Handout Data Structures

Contents

1 Basic Data Structures **1**
1.1 Abstract Data Structures . . . 3
1.2 C++ programming . . . 10
1.3 Trees and their Representations . . . 15

2 Tree Traversal **22**
2.1 Recursion . . . 22
2.2 Euler traversal . . . 25
2.3 Using a Stack . . . 25
2.4 Using Link-Inversion . . . 31
2.5 Using Inorder Threads . . . 35

3 Binary Search Trees **44**
3.1 Representing sets . . . 44
3.2 Augmented trees . . . 49
3.3 Comparing trees . . . 52

4 Balancing Binary Trees **60**
4.1 Tree rotation . . . 60
4.2 AVL Trees . . . 62
4.3 Adding a Key to an AVL Tree . . . 65
4.4 Deletion in an AVL Tree . . . 69
4.5 Self-Organizing Trees . . . 72
4.6 Splay Trees . . . 73

5 Priority Queues **78**
5.1 ADT Priority Queue . . . 79
5.2 Binary Heaps . . . 81
5.3 Leftist heaps . . . 87
5.4 Double-ended Priority Queues . . . 92

6 B-Trees **98**
6.1 B-Trees . . . 98
6.2 Deleting Keys . . . 102
6.3 Red-Black Trees . . . 103

7 Graphs **111**
7.1 Representation 111
7.2 Graph traversal 112
7.3 Disjoint Sets, ADT Union-Find 116
7.4 Minimal Spanning Trees 121
7.5 Shortest Paths 126
7.6 Topological Sort 132

8 Hash Tables **139**
8.1 Perfect Hash Function 140
8.2 Open Addressing 141
8.3 Chaining 146
8.4 Choosing a hash function 147

9 Data Compression **151**
9.1 Huffman Coding 151
9.2 Ziv-Lempel-Welch 156
9.3 Burrows-Wheeler ☒ 160

10 Pattern Matching **163**
10.1 Knuth-Morris-Pratt 164
10.2 Aho-Corasick 170
10.3 Comparing texts ☒ 171
10.4 Other methods ☒ 179

Standard Reference Works **181**

HJ Hoogeboom
Universiteit Leiden
January 22, 2024

1 Basic Data Structures

We start with some preliminary notions on pseudocode and complexity.

Pseudocode. For the algorithms in this text we mostly prefer to use pseudocode, to stress the programs work independently of the chosen programming language. There is no real language-independent form of pseudocode: even for assignment one has various choices (for instance `x:=5` or `x=5` or `x←5`).

We will explicitly delimit blocks using keywords, like in **if**...**then**...**else**...**fi**, **while**...**do**...**od** (both legal ALGOL68), or **repeat**...**until**.... Additionally we try to adhere to indentation.

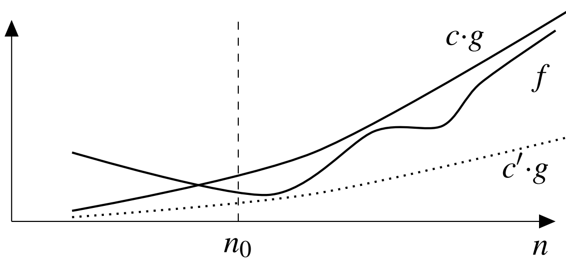
Don't worry: you do not have to *learn* this particular pseudocode. The compiler checking your exam is human, and tries hard to understand variants with indentation, or block structure using curly brackets (like in C++).

Often the notation nil is used to indicate a null-pointer (probably taken from the programming language Pascal). In pictures the symbol Λ might be used, but in many case we just draw a dot without outgoing arcs. In C++ originally the value `0` or the macro `NULL` was used as null-pointer (inherited from C), but in modern C++11 an explicit constant `nullptr` is provided to avoid unwanted conversions in the context of function overloading.

Complexity: *O* is for Ordnung. [Zie Algoritmiek] Usually we consider *worst-case* complexity of operations in data structures. The operation has worst case complexity $f(n)$ if at most $f(n)$ steps are needed to compute the operation whenever the input structure contains n items. In some cases the computation might be faster (the object we search for is at the top of the list, or the tree is nicely balanced) but we can always achieve the $f(n)$ steps.

The notion of 'steps' is rather imprecise and depends on the detail in which we look into the implementation. To avoid long discussions we use the *big-O notation*. This notation only looks at aymptotic behaviour (for large input n) and ignores constant factors.

We write $f \in O(g)$ if f is bounded above by g (asymptotically and up to constant factor) which more formally means that there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for $n > n_0$. Sometimes a variable is added in the notation and we write $f(n) \in O(g(n))$.



When we can bound the function f both from above and below by g we write $f \in \Theta(g)$. Assuming f and g are well-behaved functions this is the case iff $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and non-zero.

Then we say the operation *has complexity* $O(g)$ if its actual complexity $f(n)$ satisfies $f \in O(g)$. The complexity is *logarithmic*, *linear* or *quadratic* when $g(n)$ equals $\lg n$, n or n^2, respectively. We will also encounter $g(n) = n \lg n$, for which Wikipedia suggests 'linearithmic', but I prefer *n-log-n*.

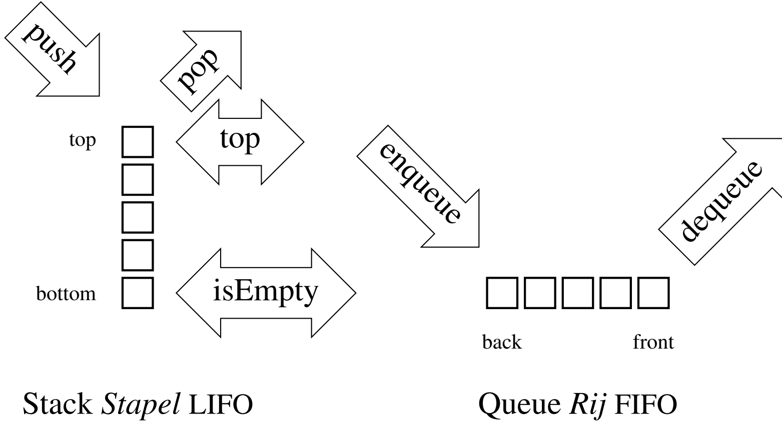
To analyse some simple repeating tasks we use some closed form expressions for summations. Well-known are $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$ and $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$. In particular $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \in \Theta(2^n)$. In Lemma 3.8 we will encounter the summation $\sum_{i=0}^{n} i \cdot 2^i = (n-1) \cdot 2^{n+1} + 2$.

We will also use $\sum_{i=1}^{n} \lg i \in O(n \lg n)$ because $\sum_{i=1}^{n} \lg i \leq \sum_{i=1}^{n} \lg n \leq n \lg n$. Actually $\sum_{i=1}^{n} \lg i \in \Theta(n \lg n)$ because also $\sum_{i=1}^{n} \lg i \geq \sum_{i=\frac{n}{2}}^{n} \lg n \geq \frac{n}{2} \lg \frac{n}{2} \geq \frac{n}{4} \lg n$ (for the last step $\frac{n}{2} \geq \sqrt{n}$ and thus $\lg(\frac{n}{2}) \geq \frac{n}{2} \lg n$). This also follows fom Stirling's approximation for factorials: $\sum_{i=1}^{n} \lg i = \lg(n!) = n \lg n - n \lg e + \varepsilon(n)$, where $\varepsilon(n) \in \Theta(\lg n)$.

The constant $\lg e \approx 1.44$ is the binary logarithm of base e of the natural logarithm, a conversion between logarithms.

1.1 Abstract Data Structures

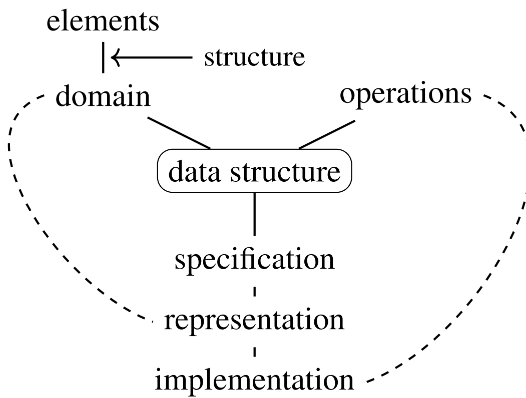
Intro. The notion of a stack or a queue can be defined independently of its implementation (using array or pointer.) The data structure stack for example represents a linear sequence of data elements, from where we can add new elements or remove existing ones, but only at a specific end of the sequence, called the top.



Definition 1.1. An *abstract data structure* (ADT) is a specification of the *values* stored in the data structure as well as a *description* (and signatures) of the operations that can be performed.

Thus an ADT is fixed by its *domain* which describes the values stored, the data elements and their structure (like set or linear list), and the *operations* performed on the data.

In present day C++ the most common abstract data structures are implemented as *template containers* with specific *member functions*. In general if we want to code a given ADT, the *specification* fixes the 'interface', i.e., the functions and their types, the *representation* fixes the constructs used to store the data (arrays, pointers), while *implementation* means writing the actual code for the functions [picture after Stubbs and Webre 1985].



The *structure* of the elements in the domain can be unordered (like in a set), linear (as in a list), hierarchical (as in a tree) or more complicated like a network or graph.

The structure in the ADT itself and that in its implementations can be different:

– The SET is unordered, while an implementation might be as tree, using linear key-order.

– The PRIORITYQUEUE is linear with values form small to large, but we implement it as Heap, which is a (hierarchical) tree stored in a linear array.

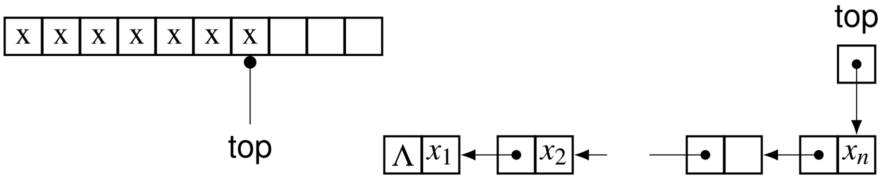
Stack. The *Stack* keeps a linear sequence of objects. This sequence can be accessed from one side only, which is called the *top* of the stack. The behaviour is called LIFO for *last-in-first-out*, as the last element added is the first that can be removed.

The ADT STACK has the following operations:

- INITIALIZE: `void` $\to$ `stack<T>`. construct an empty sequence $()$.
- ISEMPTY: `void` $\to$ Boolean. check whether there the stack is empty, i.e., contains no elements).
- SIZE: `void` $\to$ Integer. return the number n of elements, the length of the sequence $(x_1,\dots,x_n)$.
- TOP: `void` $\to$ `T`. returns the top x_n of the list $(x_1,\dots,x_n)$. Undefined on the empty sequence.
- PUSH(x): `T` $\to$ `void`. add the given element x to the top of the sequence $(x_1,\dots,x_n)$, so afterwards the sequence is $(x_1,\dots,x_n,x)$.
- POP: `void` $\to$ `void`. removes the topmost x_n element of the sequence $(x_1,\dots,x_n)$, so afterwards the sequence is $(x_1,\dots,x_{n-1})$. Undefined on the empty sequence.

Stacks are commonly implemented using an array/vector or a (singly) linked list. In the array the topmost symbol is stored as an index 'top' to the array,

whereas in the linked list there is a pointer to the topmost element. There is no need to maintain a pointer to the bottom element.



Specification. ☒ There are several possible notations for pushing x to the stack S. In classical imperative programming style we write push(S,x), where the stack S is changed as result of the operation. In object oriented style we tend to write S.push(x). In functional style we also may write push(S,x), but here S itself is not changed, instead push(S,x) denotes the resulting stack. We have to make a distiction between the *concept* of the stack versus its instantiation in our preferred programming language.

Languages have been developed to specify the syntax and semantics of ADT's. One (rather old) example is the VDM. Here the specification for the Stack of N:

```
signatures

init: → Stack
push: N × Stack → Stack
top: Stack → ( N ∪ ERROR )
pop: Stack → Stack
isempty: Stack → Boolean
```

```
semantics

top(init()) = ERROR
top(push(i,s)) = i
pop(init()) = init()
pop(push(i, s)) = s
isempty(init()) = true
isempty(push(i, s)) = false
```

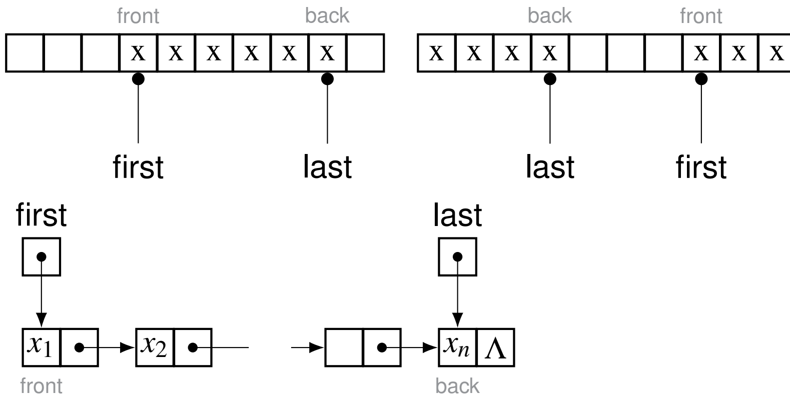
Queue. The *Queue* keeps a linear sequence of objects. Elements can be retrieved from one end (*front*), and added to the other end (*back*). The behaviour is called FIFO for *first-in-first-out*, as the elements are removed in the same order as they have been added.

The ADT QUEUE has the following operations:

- INITIALIZE: construct an empty sequence ().

- ISEMPTY: check whether there the queue is empty, i.e., contains no elements).
- SIZE: return the number n of elements, the length of the sequence $(x_1,\ldots,x_n)$.
- FRONT: returns the first element x_1 of the sequence $(x_1,\ldots,x_n)$. Undefined on the empty sequence.
- ENQUEUE(x): add the given element x to the end/back of the sequence $(x_1,\ldots,x_n)$, so afterwards the sequence is $(x_1,\ldots,x_n,x)$.
- DEQUEUE: removes the first element of the sequence $(x_1,\ldots,x_n)$, so afterwards the sequence is $(x_2,\ldots,x_n)$. Undefined on the empty sequence.

Queues are implemented using a circular array or a (singly) linked list. In a circular array we keep track of the first and last elements of the queue within the array. The last element of the array is considered to be followed by the first array element when used by the queue; hence circular. Be careful: it is hard to distinguish a 'full' queue from an empty one.

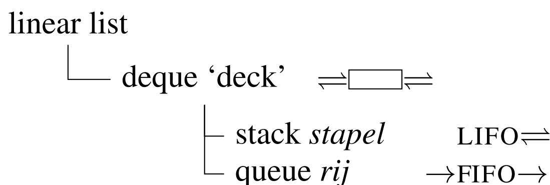


A *Deque* ('deck') or *double-ended-queue* is a linear sequence where only operations are allowed on either end of the list, thus generalizing both stack and queue. Usually implemented as circular array or doubly linked list (see also below).

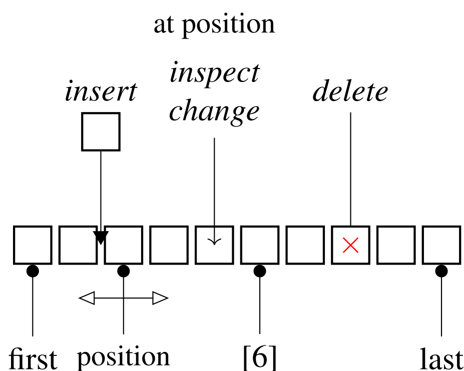
There is no common naming convention for the operations in various programming languages.

	insert		remove		inspect	
	back	front	back	front	back	front
C++	push_back	push_front	pop_back	pop_front	back	front
Perl	push	unshift	pop	shift	[-1]	[0]
Python	append	appendleft	pop	popleft	[-1]	[0]

The most general from of linear list is where we have access to all positions in the list. Thus we have the following hierarchical picture.



List. A *Linear List* is an ADT that stores (linear) sequences of data elements. The operations include Initialization, EmptyTest, Retrieval, Change and Deletion of the value stored at a certain position, Addition of a new value "in between" two existing values, as well as before the first or after the last position.

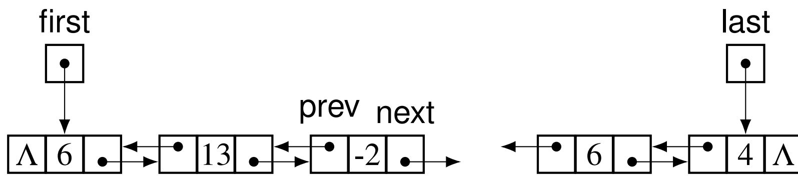


Hence, in order to specify the linear list we also require the notion of a *position* in the list, to indicate where operations should take place. We also need instructions that move the position one forward or backward if we want to be able to traverse the list for inspection. (Such a traversal is not considered necessary for stacks and queues, where we can only inspect the topmost/front element.)

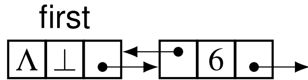
It is rather tedious to give a good description of a general list and its operations. We could start with the mathematical concept of a tuple and consider a list $(x_1, x_2, \ldots, x_n)$, with positions $1, 2, \ldots, n$. If we now add a new value at the start of the list it means all other values also have changed position. Not a natural choice for a programmer who might like to see positions as pointers.

The most common implementation of the linear list is a *doubly linked list*, with pointers to the first and last element of the list, and for each element a pointer to its predecessor and successor (if it exists).

Sometimes it is more convenient to treat the first and last pointers as special elements of the list. Such a special link that marks the ends of the list is called a *sentinel*. Using a sentinal might avoid distinguishing special cases for the empty list.



The version with a sentinel:

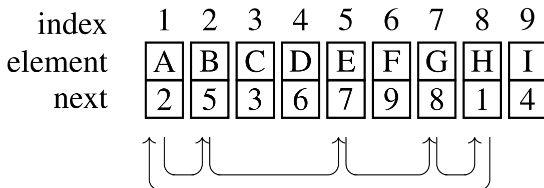


Problem. Find an efficient way of generating the Hamming numbers $2^i 3^j 5^k$, $i, j, k \geq 0$ in sorted order: $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, \ldots$.

Hybrid representations. It is a little deceptive to distinguish strictly between array and linked representations. It is sometimes useful to consider hybrid representations, where an array is used to store the links. Then the links are not pointers into memory, but indices referring to array positions.

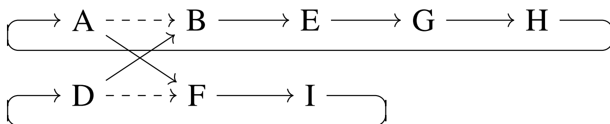
Example 1.2. We can represent the partition of a finite domain into disjoint subsets by putting the elements of each of the partition classes of the domain into a single cyclic list, connected by references to the next element in the list. In this example the references are indices to array elements. (This is a possible representation of the UNIONFIND ADT.)

The partition is $\{A, B, E, G, H\}$, $\{C\}$, $\{D, F, I\}$. We indicate the list representing the first set.



index	1	2	3	4	5	6	7	8	9
element	A	B	C	D	E	F	G	H	I
next	2	5	3	6	7	9	8	1	4

Recall that cyclic lists can be efficiently joined by swapping a pair of pointers:



Set and Dictionary. A *Set* is the ADT that matches the mathematical set. So a Set represents a (finite) collection in a fixed domain (or universe in mathematical terminology). Basic operations are EmptyTest ($S \stackrel{?}{=} \varnothing$), reporting Size ($|S|$), Adding and Deleting elements ($S \cup \{x\}$, $S \setminus \{x\}$), and a Membership test to check whether an object belongs to the set ($x \in S$).

Note this definition is a rather restricted view of sets. We can only handle single elements, and do not have set-union $A \cup B$ and set-intersection $A \cap B$ as 'native' operations.

A *Dictionary* (or Associative Array, Map) is a Set that stores (key,value) pairs, for each key at most one value, so the relation is functional. We may retrieve the value for a key in the Dictionary, and also delete the pair by specifying the key (and not its value).

In Chapter 3 we will study the Set and its basic implementation, the binary search tree. In later chapters we will see many (better) alternatives: in particular balanced binary trees (AVL-trees), B-trees, red-black trees, and hash tables.

But already with linear lists we can implement Set in a simple way. This might be sufficient for sets that are rather small. With lists we already have four choices. The list can be implemented with array or linked. Then the items can be stored either sorted or unsorted. This has implications for the efficiency of the operations. (See below for a little complexity recapitulation.)

		$x \in S$	$S \cup \{x\}$	$S \setminus \{x\}$
array	unsorted	$\mathcal{O}(n)$	$\mathcal{O}(n) + \mathcal{O}(1)$	$\mathcal{O}(n) + \mathcal{O}(1)$
	sorted	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n) + \mathcal{O}(n)$	$\mathcal{O}(n)$
linked	unsorted	$\mathcal{O}(n)$	$\mathcal{O}(n) + \mathcal{O}(1)$	$\mathcal{O}(n) + \mathcal{O}(1)$
	sorted	$\mathcal{O}(n)$	$\mathcal{O}(n) + \mathcal{O}(1)$	$\mathcal{O}(n) + \mathcal{O}(1)$
			locate+adapt	

Priority queue. A *Priority Queue* contains a set of items with their priority. Different from the Set we can not check whether an arbitrary item is present, we can only retrieve the largest (or smallest) item. See Chapter 5 for its definition, and implementations like binary heap or leftist heap. It is also part of the STL, see below.

1.2 C++ programming

This is not a course in C++.

Take some time to review the Leiden course Programmeertechnieken considering Advanced C++ programming and Modern C++ programming. In particular we are reminded that C++ is to be considered a federation of programming languages. This means it still inherits features of good old C, but also includes (1) modern object-oriented constructs, (2) templates, and (3) the Standard Template Library (see next section). These latter three concepts are essential for programming data structures. And, the other way around, choosing the right types in the STL assumes some knowledge of basic implementations of data structures.

Object orientation teaches us to model a data structure as an object, storing data which can be accessed using specific functions or operators. This has several advantages. The OOP code is more intuitive to read; compare both code fragments below: one easily recognizes the 'pop' instruction. Nicer code means less errors. Also the implementation of the operations is localized to a specific module or file. Thus error checking that implementation, or changing the implementation (from array to linked list for example) is done at a specific location, rather than all over the program. A third advantage is the information hiding principle. The user (of the implemented structure) does not know its details and cannot access the data other than with approved methods. This avoids data inconsistency.

pop-all

```
while ( S.boven >= 0 )
do   a = S.vakje[S.boven] ;
     S.boven-- ;
     ...
od
```

pop-all

```
while ( not S.isleeg() )
do   S.pop( a ) ;
     ...
od
```

Object Oriented Programming. Get acquainted with important OOP features: Classes, objects = data + methods. Inheritance. Virtual functions, dynamic binding, polymorphism. Operator overloading. Friend functions.

Use accessor methods ('getter') and mutator methods ('setter') to interface the data in your objects whenever possible. Distinguish them using `const` at appropriate places.

Prefer `vector` and `string` above the C-style array with its pointer arithmetic. They are 'first-class objects' meaning that they come with copy and assignment operators, and comparing them with `==` will give the expected result (i.e., it checks whether the strings are equal rather than their addresses).

Templates. Rather than implementing various variants of the same data type for different data items (stack of integers, stack of strings) modern C++ introduces a form of generic programming by way of templates. A template is a type parameter `T` that is used to define a family of data structures: stack of `T`.

One distinguishes *templated functions* and *templated data types* (classes). A function `max` that is defined for every type (as long as it allows comparison of elements by `<`) can be defined as in Slide 1, top.

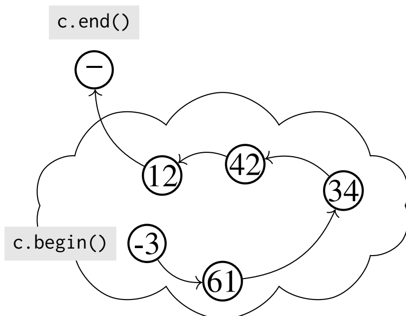
When we use `max(3,7)` in our program, the compiler will generate code for function `max(int,int)`, when we use `max(3.0,7.0)` we get code for the function `max(float,float)`. When the parameters are mixed and need a type conversion, which otherwise is applied automatically, here the compiler might need some help, and we explicitly write `max<float>(3,7.0)`.

In a similar way one defines and instantiates templated classes, see Slide 1, bottom.

Standard Template Library

The Standard Template Library STL for C++ gives its users some ready-to-use data structures matching the ADT's we have seen in the previous section. The technical details differ at some points though. For example, the STL `set` is assumed to be over an ordered universe and a simple membership $a \in A$ is not provided. The concepts used in defining the STL include that of container, iterator, and algorithm.

- *container*: holds the data
- *iterator*: walks over the container
- *algorithms*: sorting, counting, reversing



The *container* is like an ADT. It specifies the domain of the data structure, the data elements and the internal structure of the data (e.g., it may be linear, linked, ordered, etc.). Like its name promises, the STL is implemented using templates, making the library very flexible. With each container comes a collection of member functions to manipulate and access the data. Naming of the functions is consistent, so that the equivalent functions on different containers have the same name. As an example all (but one?) containers have `size` and `empty` member functions.

Slide 1 Templates.

- templated function

```
template <typename T>
T max(T a, T b) { return a>b ? a : b ; }
```

- templated class

```
template <typename Typ>
class Stack {
   ...
   private:
      vector<Typ> storage;
}

Stack<int> intStack;
Stack<string> stringStack;
```

Slide 2 STL containers

helper: `pair`

sequences:

contiguous: `array` (fixed length),
`vector` (flexible length),
`deque` (double ended),
linked: `forward_list` (single), `list` (double)

adaptors: *based on one of the sequences*:
`stack` (LIFO), `queue` (FIFO),
based on binary heap: `priority_queue`

associative: *based on balanced trees*:
`set`, `map`, `multiset`, `multimap`

unordered: *based on hash table*:
`unordered_set`, `unordered_map`, `unordered_multiset`,
`unordered_multimap`

An *iterator* provides access to the separate elements in a container. Although internally the iterator may be implemented as either an index to an array or as a pointer to a linked structure this is not visible in the definitions. This makes changing internal implementation feasible.

The container class templates can be divided into several categories.

Sequences. The `array<T, N>` is fixed size, 'contiguous' sequence (of `N` elements of type `T`), while `vector<T>` is flexible sequence, efficiently resizing when needed. The underlying implementation is a classical array. It is possible to insert elements at an arbitrary position but this will not be efficient since all successive ellements have to be moved. The `at(·)` operator references an element with bound check and may signal out of range. The classic operator `[·]` still works, but does not care about bounds.

The double ended queue `deque<T>` in the STL is a little bit peculiar. It promises to behave like a vector that also efficiently allows adding elements in front, but these elements are not guaranteed to be stored consecutively. Different from the ADT we have seen in the previous section, each element of the deque can be accessed by `[·]` and we may insert and delete at any position (this may be inefficient though).

Linked implementations of sequences are provided by the singly linked `forward_list<T>`, and the doubly linked `list<T>`.

Adaptors. These provide a new interface to an existing class. For instance a `stack<T>` provides functions `empty`, `size`, `top`, `push`, and `pop` based on a `deque<T>`, while hiding the old functionality of the vector. In particular, except for the top element we cannot access any of the elements. This mechanism is called *encapsulation*. It is possible to build the `stack` on top of a `vector` or `list`.

In a similar way `queue<T>` and `priority_queue<T>` are provided. Note encapsulation is particularly important for the priority queue. In order to operate efficiently the data structure is implemented as binary heap, see Section 5.2, and it is essential that the user will not distroy that ordering.

Associative. These containers are called *associative* as they store data items that are retrieved by the value of that element, not by its position in the container.

The `set<T>` implements the mathematical set, containing unique 'keys'. Unlike the mathematical concept it is assumed that the set is from an ordered universe. The elements stored in ordered fashion, typically implemented as balanced search tree, like the AVL tree or red-black tree.

The ordering of the elements ensures that iterating over a set will visit its elements in order. A `multiset<T>` allows multiple occurrences of a single key. For (key,value) pairs we have `map<Key, T>`. The operator `[·]` returns the value

Slide 3 STL example with `pair`, `vector`, and `priority_queue` of `pair`'s, where the second component gives the priority. Also note the cool C++11 features highlighted in the comments.

STL vector of pair

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

using paar = pair<string, unsigned int>;   // replacing typedef

int main() {
    vector <paar> club                      // 'modern' initialization
    { {"Jan", 1}, {"Piet", 6}, {"Katrien", 5}, {"Ramon", 2}, {"Mo", 4} };

    for (auto& mem: club) {                 // range based for-loop
        cout << mem.first << "   " ;
    }
    cout << endl;
    return 0;
}
```

```
Jan   Piet   Katrien   Ramon   Mo
```

STL priority_queue

```
class Comp {
public:
    int operator() ( const paar& p1, const paar& p2 ) {
        return p1.second < p2.second;
    }
};

int main() {
    vector <paar> club                      // 'modern' initialization
    { {"Jan", 1}, {"Piet", 6}, {"Katrien", 5}, {"Ramon", 2}, {"Mo", 4} };

    using pqtype = priority_queue< paar, vector <paar>, Comp >  ;

    pqtype pq (club.begin(), club.end() ); // wow! converts into
                                           // priority_queue
    while ( !pq. empty() ) {
        cout << pq.top().first << " (" << pq.top().second << ")  ";
        pq.pop();
    }
    return 0;
}
```

```
Piet (6) Katrien (5) Mo (4) Ramon (2) Jan (1)
```

for a given key. (Curiously it also adds that key to the map when it is not present! To test whether the key is present one seems to use `find` or `count`. Strange.)

Also available `multimap<Key, T>`.

Unordered. If one does not need to access elements in an ordered fashion implementations based on hash tables are provided, like `unordered_set<T>`, `unordered_multiset<T>` (hash table with buckets), `unordered_map<Key, T>` and `unordered_multimap`.

In the unordered associative containers the user can provide a specific hash-function, like in the ordered containers one may provide a key comparison operator `<`.

Note this is the C++11 version of the STL: `array`, `forward_list` and unordered containers where added in that edition.

Collections Framework. ☒ A similar approach is taken in Java, in the Collections Framework. Here an overview table of data structures and their implementations. Quite similar.

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash + Linked
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

1.3 Trees and their Representations

Trees are graphs with a hierarchical structure used to represent data structures. In the following chapters we will see many of these types of trees.

Some trees have *structural* restrictions:

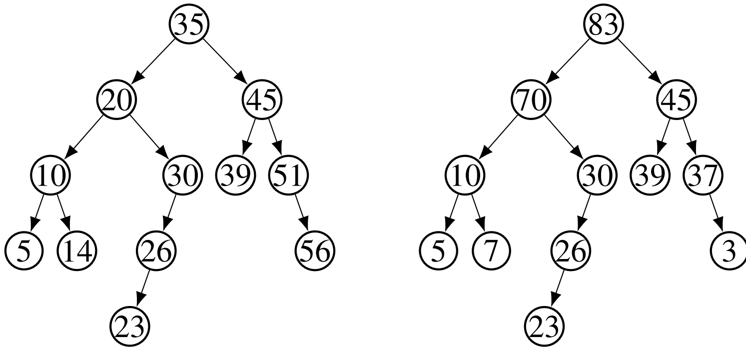
- on the number of children in each node (binary tree, B-trees)
- on the number of keys in each node (B-trees)
- on the number of nodes in each subtree (balanced trees: AVL-tree, B-tree)
- complete trees (binary heap)

Some structural restrictions are designed to make access of keys fast, but we have to take care that updating the data structure after insertion of deletion is not too complicated as we have to adhere to the restrictions.

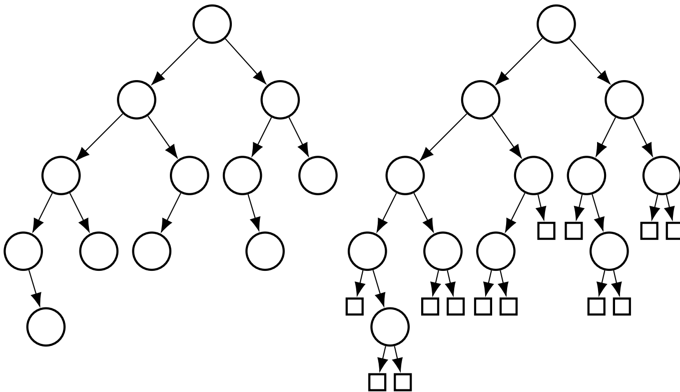
Also trees usually have restrictions on the placement of data in the nodes.

- In a *binary search tree* the keys in the left subtree are smaller than the key at the root of the subtree, and the keys in the right subtree are larger than the key at the root. This is a good organization to find arbitrary keys in the tree.

- In a tree with *max-heap ordering* [not necessarily complete] the keys at the children of a node are smaller than the key at the node. There is no left-right ordering. This is geared towards finding the maximal element, *not* for finding arbitrary keys.



Full binary trees. A binary tree is *full* if each node is either a leaf (has no children) or has two children. (This is sometimes called a 2-tree.) Every binary tree can be made into a full tree in a systematic way by adding leaves below the current nodes (at every position where a child is missing). Sometimes these new leaves are depicted in a different way to show they are special "external" nodes. In general non-leaf nodes are called *internal*.

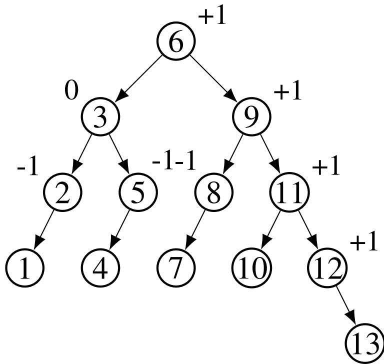


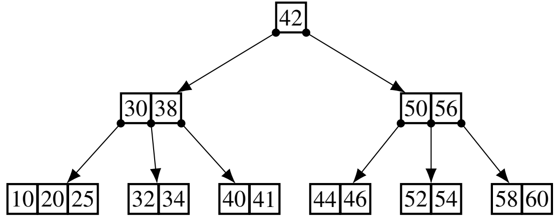
It is a mathematical fact that if the tree has n nodes, its extended tree has $n+1$ leaves.

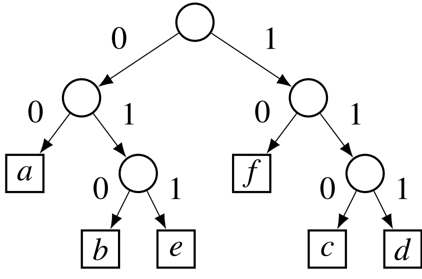
Lemma 1.3. *Let T be a full binary tree. If T has n internal nodes with at least one child, then T has $n+1$ leaves.*

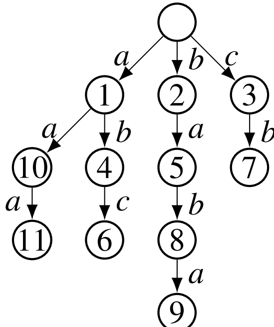
Proof. By induction, either top-down or bottom-up. *Top-down*: the root has right and left subtrees that satisfy the property. We may assume the left subtree has

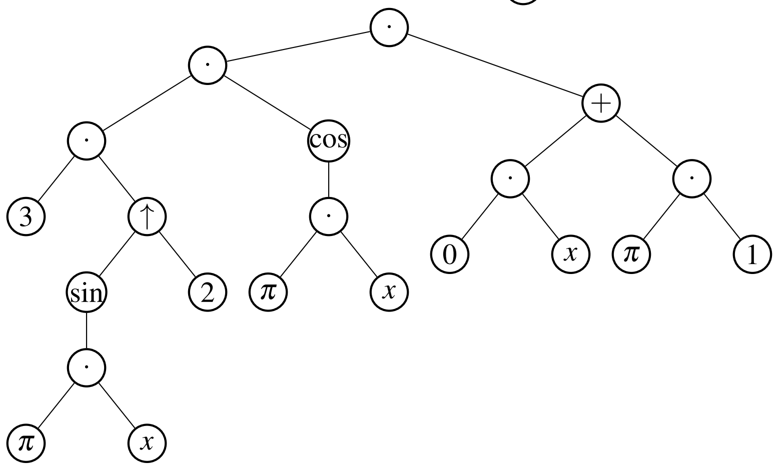
Slide 4 Various trees: (1) height-balanced AVL-tree and B-tree, used to represent sets; (2) trees for text compression: Huffman and ZLW algorithms; (3) expression tree.





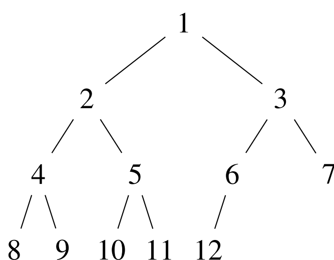


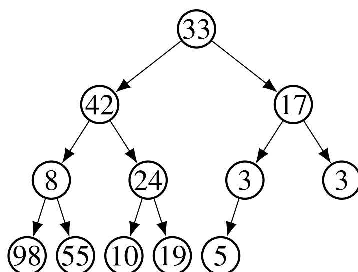




n_1 internal nodes and n_1+1 leaves, and similarly n_2 internal nodes and n_2+1 leaves for the right subtree. The total number of internal nodes is n_1+n_2+1 as we add the root. The total number of leaves is n_1+1+n_2+1. This is indeed one more than the number of internal nodes. *Bottom-up*: the tree has a node with two children both of which are leaves. When replacing this node with its children by a leaf we obtain a smaller tree that has one node and one leaf less than the original tree. □

Complete binary trees. Binary trees can be represented in an array (of nodes) using a numbering known from genealogy[1].





33	42	17	8	24	3	3	98	55	10	19	5
1	2	3	4	5	6	7	8	9	10	11	12

In this way the children of node i are $2i$ and $2i+1$, the parent of node i equals $\lfloor i/2 \rfloor$. (Yes, some people start numbering the root with 0: the changes are simple. Writing the indexes in binary shows why choosing 1 here makes sense.) In this way we can store the information of the each node in the corresponding position of an array, without providing explicit links between parents and children. This approach will waste a lot of space when the tree has relatively many missing children. It will work well for a *complete* binary tree, which is a binary tree where each level has all its nodes, except perhaps the last level, which is filled from left to right. The representation is used in the *binary heap*, Section 5.2.

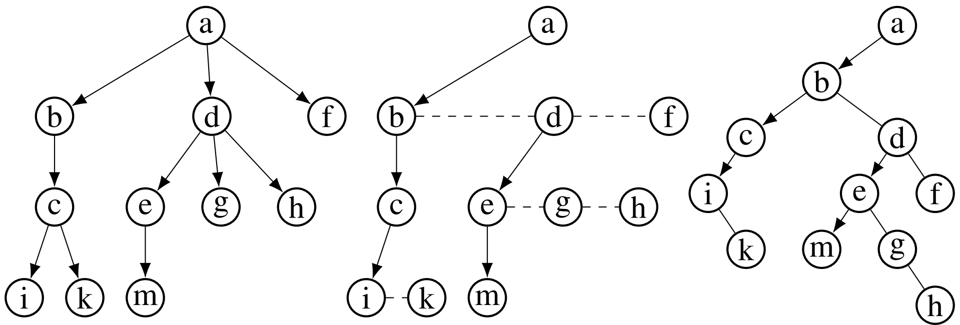
Binary tree representation. The 'common' tree in computer science is the binary tree as represented by nodes with left and right links to children.

```
template <class T>
class BinKnp {
          \\ CONSTRUCTOR
    BinKnp ( const T& i,
             BinKnp<T> *l = nullptr,    \\ default
```

[1]Wiki knowledge: It seems the first *Ahnentafel* using this numbering system was published in 1590 by Michael Eytzinger in *Thesaurus principum hac aetate in Europa viventium* Cologne/Köln

```
            BinKnp<T> *r = nullptr )
        : info(i)    \\ constructor type T
        { links = l; rechts = r;
        }
private:   \\ DATA
    T info;
    BinKnp<T> *links, *rechts;
};
```

Left-child right-sibling binary tree. A trick to represent trees with no bound on the number of children as binary trees. Following links to the left we find the list of siblings, the children of a node. The link to the left points to the list of children of that node. Although this looks like a binary tree, the *meaning* of left and right are replaced by a new interpretation. [zie Algoritmiek]

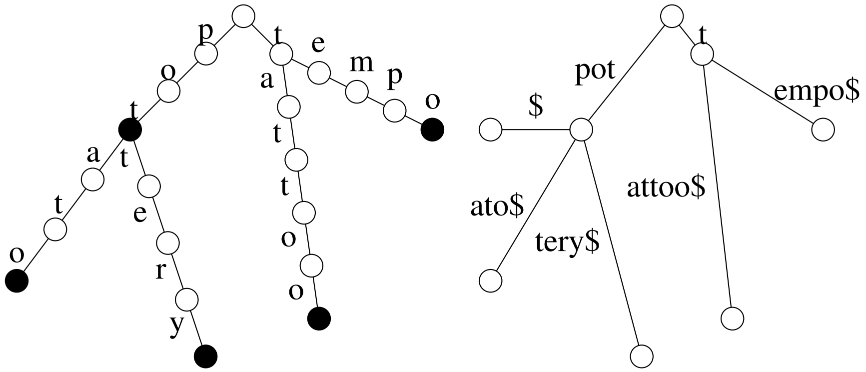


Trie. A *trie*, pronounced like tree as in retrieval, is a tree structure to store a set of strings. The edges of the tree are labelled by letters, and the words that can be read from root to leaf belong to the stored set of strings. The problem with efficiently implementing a trie is that some nodes have only a single successor, while others have many, so it is not wise to reserve a pointer for every possible child (letter). A *compact trie* compresses the labels of paths that do not bifurcate into a single edge; the label of that edge will be a string.

Not all strings that belong to the set of strings will end in a leaf when the set contains prefixes, like ban and banana. In that case we either label the 'end' nodes, or we attach a special symbol $ to all strings.

In the area of computational molecular biology a set of algorithms was developed based on an efficient representation of the set all suffixes of a given string, the suffix tree.

Example 1.4. Representations for the set { pot, potato, pottery, tattoo, tempo }. Left a trie (with nodes that correspond to the set of strings marked) and right a compact trie, with $ end markers.



◊

References. See PROGRAMMEERMETHODEN Week 12: Datastructuren: stapels, rijen en binaire bomen

See PROGRAMMEERTECHNIEKEN Week 4, 5: Advanced C++ programming; Week 6: Modern C++ programming

See C++ REFERENCE `http://www.cplusplus.com/reference/stl/`

Opgaven

1.a) Wat is een abstracte datastructuur (ADT) ?

b) Beschrijf de ADT *stapel* (=*stack*) en schets twee geschikte implementaties.

c) Beschrijf de ADT *union-find* (=*disjoint sets*) en geef een passende implementatie. Bij welk algoritme heeft union-find een belangrijke toepassing?

Jan 2015

2.a) **(i)** Beschrijf de abstracte datastructuur *Rij* (=*Queue*): wat wordt opgeslagen, wat zijn de standaard operaties (en wat doen ze)?
(ii) Schets twee bekende implementaties.

b) **(i)** Beschrijf de abstracte datastructuur *Priority Queue*: wat wordt opgeslagen, wat zijn de standaard operaties (en wat doen ze)?
(ii) Leg uit hoe een binaire zoekboom gebruikt kan worden om de Priority Queue te implementeren. *Mrt 2015*

2 Tree Traversal

A *tree traversal* is a systematic way to visit all nodes of a tree. In this way we

- can obtain information about the values stored in the tree,
- can verify or update the structure of the tree,

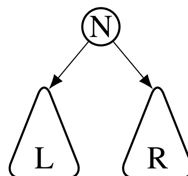
but most of all, we learn techniques that can be extended to general graphs.

2.1 Recursion

For binary trees there are three natural, recursive, orders to visit the nodes in the tree. For preorder we start by visiting the node, then recursively we visit the left and right subtrees of the node, or NLR. Similarly for inorder LNR and postorder LRN.

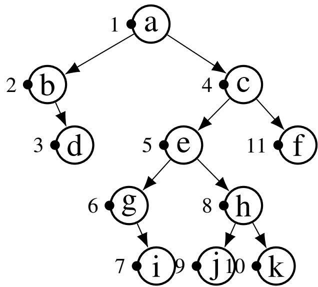
These order have different applications. Preorder corresponds to depth-first search DFS in graphs, inorder lists the nodes from left to right (from small to large in a binary search tree), while postorder is useful in a bottom-to-top evaluation of the tree.

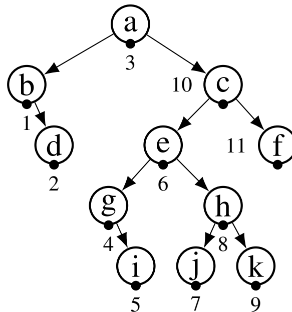
- *preorder* (NLR, node-left-right)
- *inorder*, or *symmetric order* (LNR)
- *postorder* (LRN)

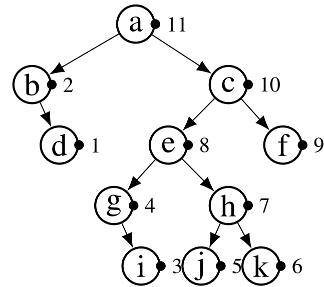


In this section we consider binary trees only, that is trees in which each node has a left and a right subtree, where each of these can be empty. Preorder and postorder can easily be defined for non-binary trees, where the number of subtrees is not fixed or where that number is larger than two. For preorder we visit the node N followed by their subtrees from left to right. Similarly for postorder. There is no obvious definition for inorder for general trees.

Example 2.1. Preorder, inorder, and postorder numbers for the nodes of a binary tree. Below the diagrams the nodes are listed in the respectice orders. (The alphabetical order of the nodes corresponds to the level-order.)







NLR = preorder
a b d c e g i h j k f

LNR = inorder
b d a g i e j h k c f

LRN = postorder
d b i g j k h e f c a

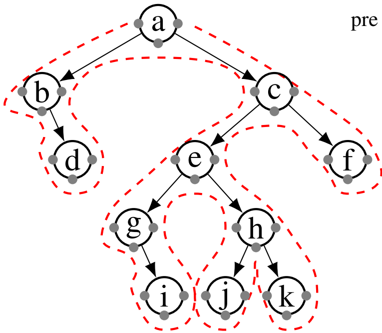
The recursive definitions immediately translate into recursive algorithms for the three methods. By inspecting nodes at the first, second or third visit we obtain preorder, inorder and postorder evaluations.

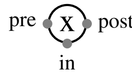
recursive

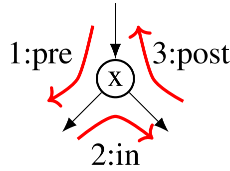
```
traversal( node )
    if (node != nil)
    then
        pre-visit( node )   // first
        traversal( node.left )
        in-visit( node )    // second
        traversal( node.right )
        post-visit( node )  // third
    fi
end // traversal
```

In the next sections we will give nonrecursive algorithms to traverse trees. First with the help of an additional data structure, the stack. Then without stack, by (temporarily) storing traversal information inside the tree.

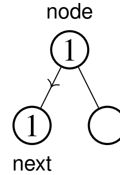
Slide 5 Euler traversal. Top: Euler traversal in example tree, following the 'outline'. Middle: Table of the cases moving from current node to the next position. Bottom: ☒ Translating the table into a while-structured pseudo-program.

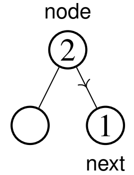


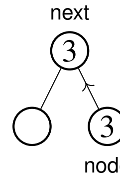


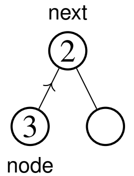


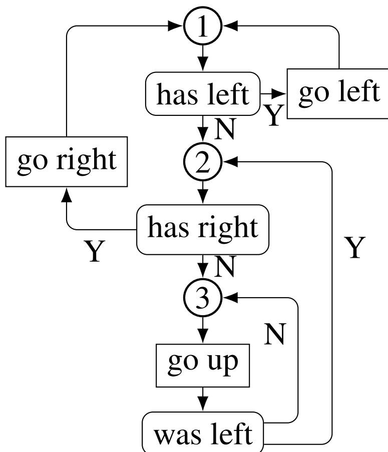
visit	test	direction	next
1	has left-child	down-left	1
		(stay)	2
2	has right-child	down-right	1
		(stay)	3
3	at left-child	up	2
	at right-child	up	3
	at root	(exit)	–












```
                 Euler traversal
start at root
while (node not nil)
do  pre-visit (1)
    while (has left child)
    do  go left "push"
        pre-visit (1)
    od
    in-visit (2)
    while (not has right)
    do  repeat
            post-visit (3)
            go up "pop"
            if nil then exit fi
        until (was left)
        in-visit (2)
    od
    go right "push"
od
```

2.2 Euler traversal

An *iterative* algorithm walks over the tree, stepping from node to node, inspecting ("visiting") the node at the proper moment. The *Euler traversal* is a generic tree-walk for binary trees that visits each node three times, scanning the 'boundary' of the tree. At each of the visits we move to the next node in a fixed order. See for an example Slide 5(top).

We list these steps in a table as in Slide 5(middle). For example, if we visit a node for the first time, we go to the left child and visit that node, also for the first time. If the current node has no left child, we stay and visit the current node for the second time. When the node is visited for the third and last time we move up in the tree. The parent is then visited for the second or third time, depending on whether we came from a left or a right child.

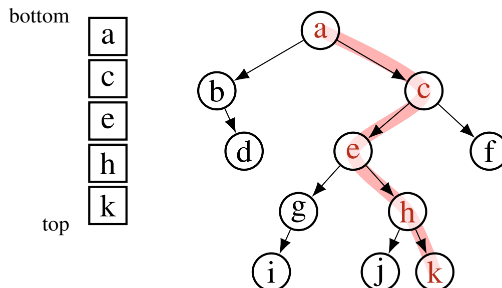
We can make this into a traversal algorithm suitable for each of the three recursively defined orders, by adding a **while** loop. In Slide 5(bottom) the old fashioned flowchart for such an algorithm is given, which we then can reframe using common control-structures.

If we have a tree representation where each node has three pointers, to children ánd parent, this already is a technique to walk over trees: just follow the pointers up and down the tree.

In the standard representation of binary trees going *down* to either of the children is not a problem. However we cannot directly *move up* to the parent of a node. The favourite solution for this is to use a stack. We show specific traversal algorithms with a stack in the next paragraph. The postorder traversal will use ideas of the Euler traversal.

2.3 Using a Stack

To be able to return to nodes in higher levels of the tree we use a stack that stores a selection of the nodes we have seen before during traversal. In the small diagram below *all* nodes on the path from root a to leaf k are pushed on the stack. The choice of which nodes that we store depends on the traversal method, as we will explain below.



Slide 6 *Preorder traversal using a stack.* Top: recursive algorithm. Middle: Straightforward translation pushing children onto a stack. Bottom: wrapped version, with diagram indicating when the right children that are popped from the stack.

recursive

```
traversal( node )
    if (node != nil)
    then
        pre-visit( node )
        traversal( node.left )
        traversal( node.right )
     fi
end // traversal
```

pre-order

```
iterative-preorder( root )
    S : Stack
    S.push( root )
    while ( not S.isEmpty() )
    do  node = S.pop()
        if (node != nil)
        then visit( node ) // pre-order
            S.push( node.right )
            S.push( node.left )
        fi
    od
end // iterative-preorder
```

pre-order (wrapped)

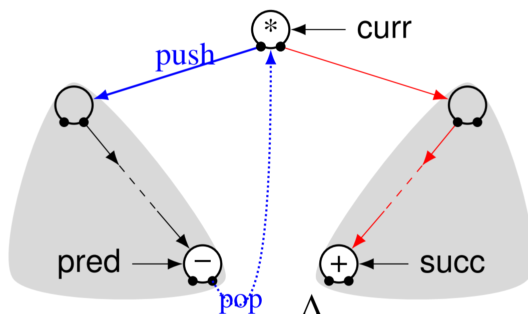
```
iterative-preorder( root )
    S : Stack
    S.create()
    S.push( root )
    while ( not S.isEmpty() )
    do  node = S.pop()
        while (node != nil)
        do  visit( node ) // pre-order
            S.push( node.right )
            node = node.left
        od
    od
end // iterative-preorder
```

Preorder. For a preorder walk we may treat the stack as if it is a stack of recursive function calls. Preorder means visit a node, then visit its two subtrees. This is implemented in a staightforward way with a stack: Pop a node, visit it, and push both its children. Repeat. See Slide 6(Middle).

Alternatively we may unroll the tail recursion: the last (left) node pushed is immediately popped, so it seems more efficient to move directly to the next left child. In that way we push the right children of the visited nodes onto a stack, while continuing to the left. When a leftmost branch ends we pop the last node from the stack (which will be the right child of the last node where we went left). Slide 6(Bottom).

In general this will also push nil-pointers for missing right children, but these are ignored when popped. Of course if you want, this can be "optimized" by not pushing nil-pointers onto the stack.

Inorder. For inorder (=symmetric) traversal we have to (re)visit a node when the last (rightmost) node of its left subtree has been visited. Thus we construct a stack, containing the predecessors of the currently visited node at the points where the path moves to the left. See Slide 7 for the pseudo code.



Example 2.2. Traversal in a binary tree using a stack, comparing preorder (left) and inorder (right). When moving left into a new subtree the algorithm pushes the right child of the node (preorder) or the node itself (inorder).

Dashed arrows indicate nodes that are popped from the stack when the node has no right child. For the preorder we move from visit 7 (node *i*) to visit 8 (node *h*). Also the right child from *g* is pushed onto the stack (when moving left to *i*) but this nil-node is ignored, and we pop a new node from stack.

Slide 7 *Inorder traversal using a stack.* The stack contains the nodes where the path moves left. In that way we can pop and return to the root of a completely traversed left subtree for its inorder visit. Bottom: cleaner version with function to walk to along the left path to first node in subtree. Also: the *Wikipedia* version, which is very compact and symmetric (5.2'22). Be careful, both left and right nil-pointers signal popping.

in-order

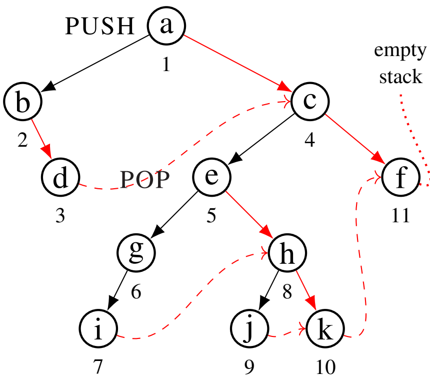
```
iterative-inorder( root )
    S : Stack
    S.create()
    node = root
    // move to first node (left-most)
    while (node != nil)
    do  S.push( node )
        node = node.left
    od
    while ( not S.isEmpty() )
    do  node = S.pop()
        visit( node ) // inorder
        node = node.right
        while (node != nil)
        do  S.push( node )
            node = node.left
        od
    od
end // iterative-inorder
```

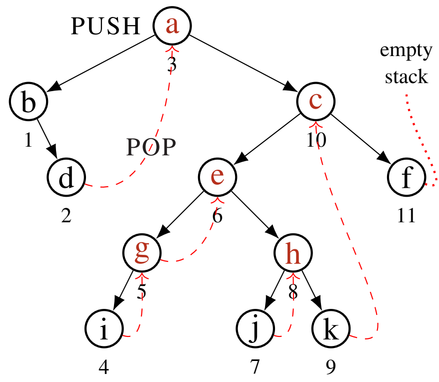
in-order (netter)

```
iterative-inorder( root )
    S : Stack
    S.create()
    // move to first=left-most
    node = root;
    walkLeft( node, S )
    while ( not S.isEmpty() )
    do  node = S.pop()
        visit( node )
        node = node.right
        walkLeft( node, S )
    od
end // iterative-inorder

walkLeft( node:Node, S:Stack)
    while (node != nil)
    do  S.push( node )
        node = node.left
    od
end // walkLeft
```

in-order (wikipedia)

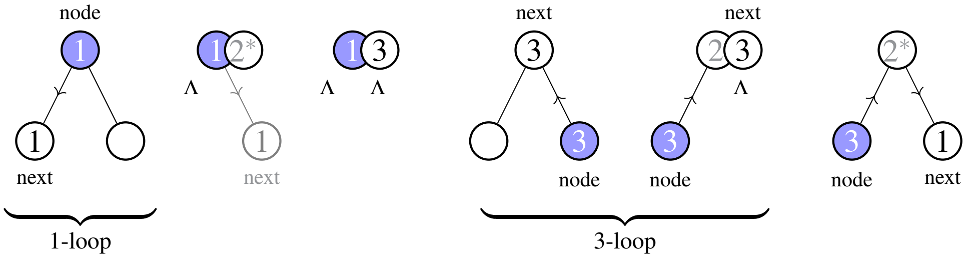
```
iterative-inorder( root )
    S : Stack
    S.create()
    node = root;

    while ( not S.isEmpty()
                or node != nil )
    do  if (node != nil)
            S.push( node )
            node = node.left
        else
            node = S.pop()
            visit( node )
            node = node.right
        fi
    od
end // iterative-inorder
```





Postorder. This seems to be the most complicated traversal when programmed iteratively. We basically follow the generic Euler traversal as described above. The stack contains the complete path to the present node. We push the new node when we go down (i.e., left or right, first visit) and we pop the node when we go up to the parent (which is always stored on top of the stack, third visit).

When we go up, we need to distingish whether we start from a left child or a right child. To do this, we may "peek" at the top of the stack (with the father of the current node) to see at which side the current node is with respect to its father. This determines whether we have visited the current node for the second or for the third time. In these cases we either go right, or visit the current node, and go up in the tree (last three diagrams below).



Two implementations of this algorithm are presented.

First the version of Drozdek, as in Slide 8. The parent of the current node is popped from the stack. Here two consecutive loops are repeated: one while-loop going down to a left child, first visit. The other while-loop goes up and checks this is from the right parent. Otherwise we end in the last situation, when node.right==last-visited, where we move from left to right subtree. If you compare this to the Euler traversal (omitting first and second visits) you will see a striking resemblance.

Slide 8 Postorder traversal from Drozdek. Bottom: pseudocode.

postorder (Drozdek)

```
template<class T>
void BST<T>::iterativePostorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p->right == 0 || p->right == q) {
            visit(p);
            q = p;
            if (travStack.empty())
                return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

postorder (pseudo Drozdek)

```
iterativePostorder
    S : Stack
    node = root, last = root
    while (node != nil)
    do      // go down left, 1st visit
        while (node.left != nil)
        do  S.push(node)
            node = node.left
        od
            // go up, 3rd visit
        while (node.right == nil or node.right == last)
        do  visit(node);
            last = node;
            if (S.isEmpty() ) then return fi // exit
            node = S.pop()
        od
            // (2nd visit) go right via parent
        S.push(node)
        node = node.right
    od
end // iterativePostorder
```

The other implementation is from Wikipedia, see Slide 9. The algorithm distinguishes going down (visit 1) from going up (visit 3) by checking whether node==nil. As long as we are in the upward mode node is kept nil and we visit the node peek instead. This peek is the parent node, presently the top of the stack. (Nasty trick, it took me quite some time to realize why going up we were not updating node.)

What's on the stack. Each of the three traversal algorithms can be better memorized by noting which nodes are on the stack during traversal. In the diagrams below we depict the path to the current node * from the root, for preorder, inorder and postorder, respectively. nodes that are already visited are marked by a checkmark ✓, those that are on the stack are marked by an **X**.

Understanding which nodes are kept on the stack is understanding the algorithm.

pre-order in-order post-order

2.4 Using Link-Inversion

Link-inversion is a binary tree-traversal technique where no additional stack is used. Instead, the stack is threaded through the tree by temporarily inverting the links.

Link-Inversion in linear lists. We first illustrate the technique using a linear list. During traversal the links form two paths, one back to the initial node, one forward to the last node. Using this method we can only perform *one* traversal at a time in the list. Note we need two pointers to the list: to keep track of the

Slide 9 *Postorder traversal using stack, as in Wikipedia.* Bottom: pseudo-code

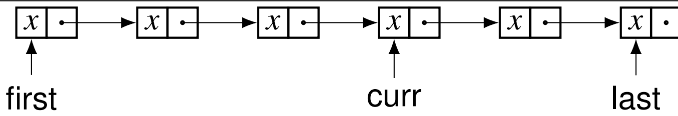
post-order (wikipedia 20.9'21)

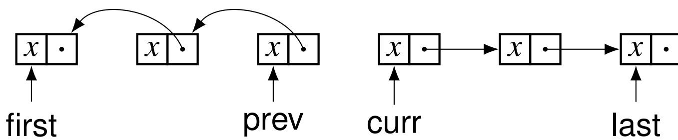
```
procedure iterativePostorder( node )
    stack ← empty stack
    lastNodeVisited ← null
    while not stack.isEmpty() or node ≠ null
    do  if node ≠ null
            stack.push(node)
            node ← node.left
        else peekNode = stack.peek()
            // if right child exists and traversing node
            // from left child, then move right
            if peekNode.right ≠ null
                    and lastNodeVisited ≠ peekNode.right
                node ← peekNode.right
            else visit(peekNode)
                lastNodeVisited ← stack.pop()
```

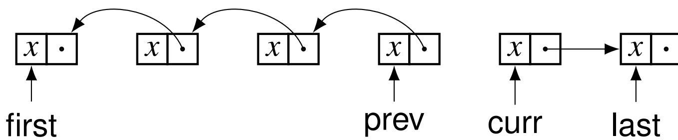
post-order (pseudo-wikipedia)

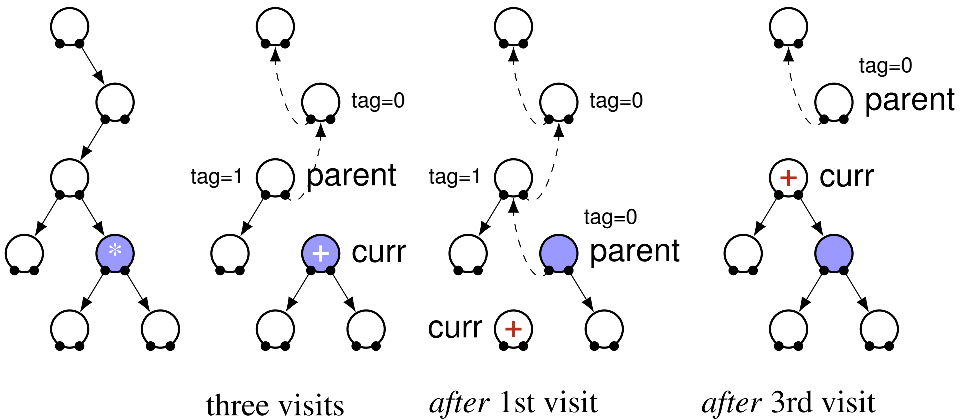
```
iterative-postorder( root )
    S : Stack // contains path from root
    S.create()
    last = nil
    node = root
    while (not S.isEmpty() or node != nil)
    do  if (node != nil)
        then S.push(node)
            node = node.left
        else // if right child exists and traversing node
            // from left child, then move right
            peek = S.top()   // peek = parent
            if (peek.right != nil and last != peek.right)
            then
                node = peek.right
            else visit(peek)
                last = S.pop()
                // node == nil   nog steeds!
            fi
        fi
    od
end // iterative-postorder
```

Slide 10 Link-inversion. Top: In a linear list. Bottom: In a tree. **(1)** Fragment of binary tree. **(2)** Three visits at coloured node $*$. **(3)** After first visit: move down left. **(4)** After third visit: move up.









visited node curr and its predecessor prev. We can move up and down the list, the situation is symmetrical. See Slide 10(Top)

Link-Inversion in Binary Trees

Features:

- Euler style: global visit counter 1,2,3.
  can be used in pre-order, in-order or post-order fashion.
- no external stack
- path to visited node *link-inverted*, to function as stack
- *tag*: one bit per node (along the inverted path) to distinguish left/right children
- keep pointer to parent (gap)
- structure is disturbed: only a single traversal at a time

We perform the Eulerian iterative traversal for binary trees where each node is visited three times, see Section 2.2. At each moment we know the visit number at the current node, and we act accordingly, moving left, right or up.

Along the path from the root to the currently visited node curr the links are inverted. This means the pointer no longer indicates its child along the path but rather points to its parent. When we back-up to the root we need to be able to distinguish which of the children pointers is inverted, and we use one bit of information at each node, called the *tag*. The tag has value 0 if the path to the current node continues with the left child, and has value 1 otherwise. Tags not on the path from the root to the current node are undefined.

Like the linear list case, we cannot reach the current node from the root (and vice-versa) and we keep an additional pointer to the parent of the visited node.

When going left-down, the next node is the left child of the current node, and we cyclicly swap curr.left, curr and parent pointers. When going up, we determine the direction using the tag-field of the parent. If that tag is 0 we move up from left, and we cyclicly swap parent.left, curr and parent pointers. See Slide 10(Bottom).

Exercise. The algorithm uses the tag at the parent pointer to distinguish whether to return from a left or right child. Now imagine the tree is a binary search tree, that is, the keys are ordered consistent with the inorder, see Section 3. The question is now: can we avoid the tag when doing link-inversion?

2.5 Using Inorder Threads

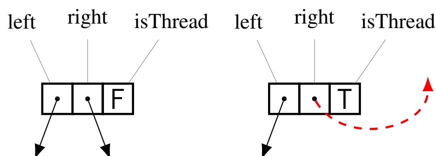
Features:

- *threads*:
  replace nil-pointers to indicate inorder successors
- can be used to perform stack-less traversal
- need one bit [boolean] per node to mark thread

- *Morris-variant*: temporary threads, no extra bit

From basic tree combinatorics we know that a tree with n nodes has $n-1$ edges. Now, a binary tree with n nodes has $2n$ pointers, of which $n-1$ are used as edges. That means that half of the pointers in a binary tree implementation are nil, and have no direct use.

We enrich the standard representation by adding *threads* pointing to inorder successors. So, in each node that has no right child we *replace* the right nil-pointer by a reference to the inorder successor of the node. Additionally we introduce a new field for nodes, a boolean, to distinguish threads from edges. For consistency the successor of the last node (in inorder) is taken to be nil. It is marked as thread. Here we consider right (forward) threads only; there is the possibility to consider also left threads to inorder predecessors.

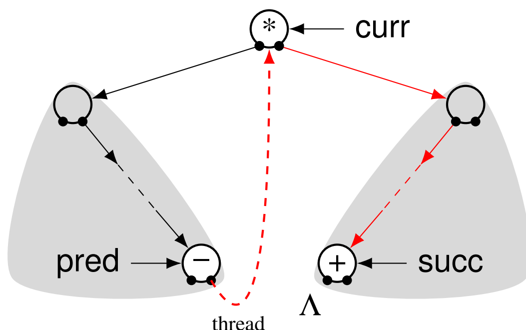


In Slide 11 we show an example of a binary tree where right threads have been inserted at nodes without right child. Threads are represented by dashed lines (and point upwards). Recall that they are stored in the right pointer field of the binary node. (Again: there is no 'physical' difference between child-pointer or thread, both are located at the same memory position. A Boolean isThread indicates that the right pointer is indeed a thread and not an edge of the tree. In diagrams threads are red, dashed, and go up.)

The inorder successor succ for node curr when that node has a right child is the leftmost node in the right subtree of curr. Otherwise, when curr has a right thread, we directly follow the thread to find the successor. In that case curr itself is the rightmost node in the left subtree of succ. Note the symmetry.

Traversal. The additional information stored in threaded trees makes tree traversal possible without recursion or external stack. To find the inorder successor of a node we consider two cases. When the node has a right child the successor of

that node is the leftmost node of the right subtree; this is not necessarily a leaf, the leftmost node may have itself a right subtree. When the node has no right child the successor is directly stored as thread. See the diagram below. In Slide 12 one finds the traversal algorithm using (inorder) threads.



Dynamic trees. Usually trees are dynamic: nodes are added and deleted. When a tree is stored using threads we have to update the threads when the trees is changed. This does not only mean we have to add threads when a node is added, but also, e.g., that we have to think what happens to a thread that points *to* a node that is deleted.

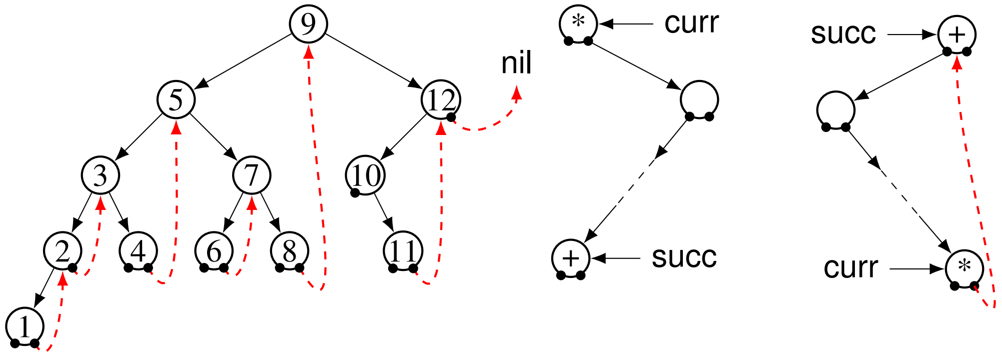
Exercises. How to locate the parent of a node in a threaded tree? Can we perform other traversals (preorder, postorder) using inorder threads? Can we add preorder threads, and do preorder traversals? Same question with postorder.

Morris traversal. Using the thread technology it is possible to traverse the tree in inorder by *temporarily* adding threads to the tree (different from the threaded tree representation, where the threads are marked ánd remain in the tree). Each time we step downwards into a left subtree we add a thread from the rightmost node in the subtree back to the current node. This thread will enable us to exit from the subtree when it has been visited. The moment the thread is followed it is removed. Of course we constantly have to check whether right links are either tree edge or thread as this is not indicated at the node.

Features (Morris):

- do not know threads: move right and check afterwards
- (pre-order visit) arrive from parent via child-link;
  add thread to current node
- (inorder) leave subtree, via thread; remove thread

Slide 11 Left: Binary tree with forward inorder threads (dashed) Right: From current node to inorder successor: via edges, or via thread



Slide 12 Binary tree traversal using inorder threads.

```
inorder threads

    // assuming root != nil
curr = root;
walkLeft( root ); // to first in inorder
while (curr != nil)
do  inOrderVisit( curr );
    if (curr.isThread)
    then     // follow thread
        curr = curr.right;
    else     // down to first in subtree
        curr = curr.right;
        walkLeft (curr)
    fi
od

walkLeft( node : Node)
    while (node.left != nil)
    do  node = node.left
    od
end // walkLeft
```

There is a very clever implementation of this idea[2], see Slide 14. Observe that when we reach a node from above from the parent, or from below from the successor via a thread, this will always be the first or second visit to this node. So we want to distinguish between these two, see Slide 13. **(a)** When the node has no left child, we cannot return to it from the left subtree, so we have consecutive first and second visits: do the inorder visit, and go right (we do not know whether that follows an edge or a thread, but we will find out).

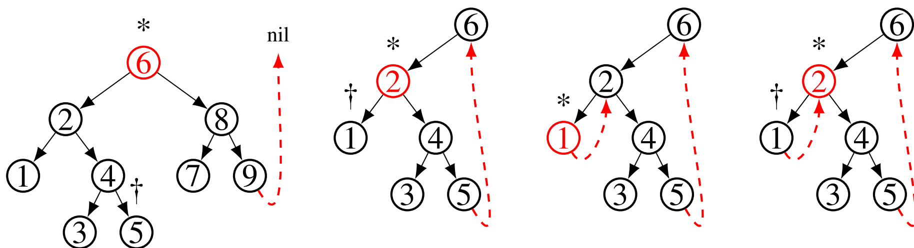
Otherwise, from the current node, look for the predecessor (the rightmost node in the left subtree, which exists as the left child is present). **(b)** In case this rightmost node, the predecessor, has a right nil-pointer, we have never been in the left subtree and we add a thread from predecessor to current node, and continue with the left child (in its first visit). **(c)** Alternatively, the predecessor has a right link to the current node, a thread. Thus we have completed that left subtree. hence we remove the thread, visit the current node, and move right from the current node.

Example 2.3. The traversal of a binary tree and the configuration with temporary threads when visiting successive nodes in the tree. When the node ($*$ in the diagrams) has a left child, we determine the predecessor (†) of that node (left, then repeatedly move right until the right link is either nil, or leads to the current node.)
1. At (6), no link from predecessor (5), first visit, add link, move left.

2. At (2), no link from predecessor (1), first visit, add link, move left.

3. At (1), no left child, first and *second visit*, move right. (We follow a thread, but we can't see this.)

4. At (2), has link from predecessor (1), *second visit*, remove link, move right.



5. At (4), no link from predecessor (3), first visit, add link, move left.

6. At (3), no left child, first and *second visit*, move right.

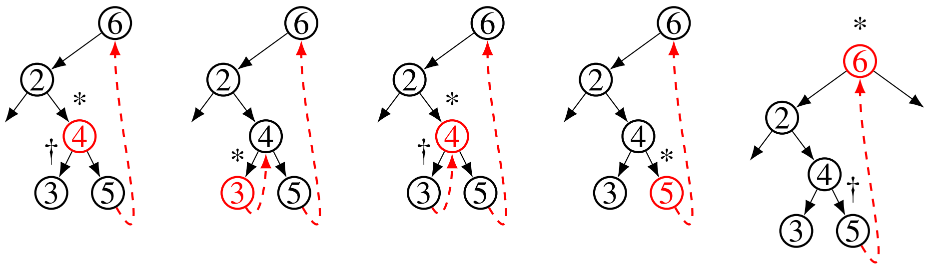
7. At (4), has link from predecessor (3), *second visit*, remove link, move right.

8. At (5), no left child, first and *second visit*, move right.

9. At (6), has link from predecessor (5), *second visit*, remove link, move right (to node (8) not shown here).

---

[2]To be honest, I got this from YouTube.



At this point all threads in the left subtree of the root have been removed, and we continue our traversal in the right subtree of the root.

References. A. PERLIS & C. THORNTON. Symbol manipulation by threaded trees. Communications of the ACM **4** 195–204 (1960) doi:10.1145/367177.367202

J.M. MORRIS. Traversing binary trees simply and cheaply. Information Processing Letters **9** 197–200 (1979) doi:10.1016/0020-0190(79)90068-1

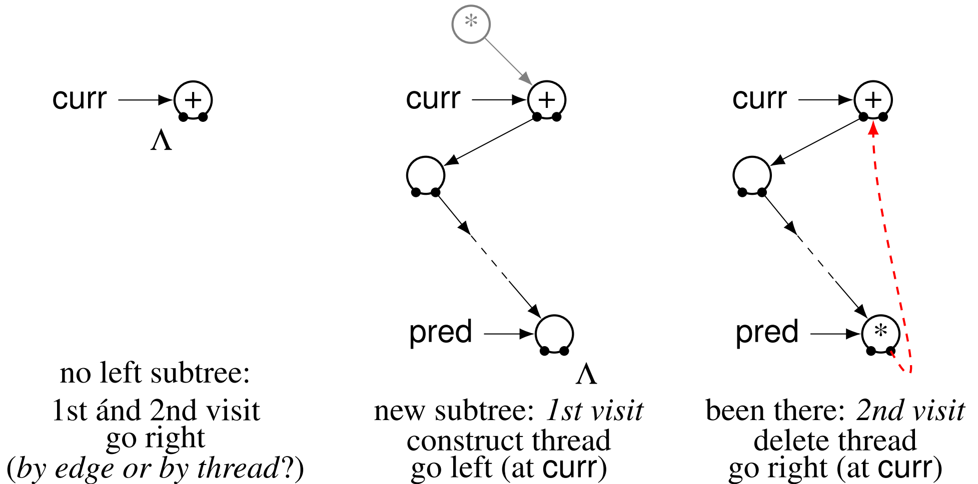
See DROZDEK Section 6.4.3: Traversal through tree transformation. Note the book considers this traversal as ‘tree transformation’, which is technically correct, but the connection to threads may help understanding the algorithm.

See KNUTH Section 2.3.1: Traversal binary trees. Algorithm S.

D. GORDON. Eliminating the flag in threaded binary search trees. Information Processing Letters **23** 209–214 (1986) “. . . allows us to distinguish efficiently between a thread and a right-son pointer during searching or insertion” [*I have to check this out. HJH*] doi:10.1016/0020-0190(86)90137-7

We will meet J.M. Morris and D.E. Knuth, both mentioned above, in Section 10.1, where we present the Knuth-Morris-Pratt algorithm for text search.

Slide 13 *Morris tree traversal.* After moving to node curr via a right link, we need to check whether this was via a child-link (first visit) or via a thread (second visit).

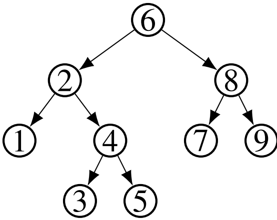


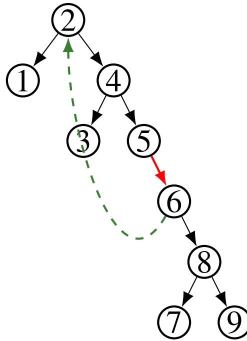
Slide 14 ALGORITHM: Morris tree traversal using temporary threads

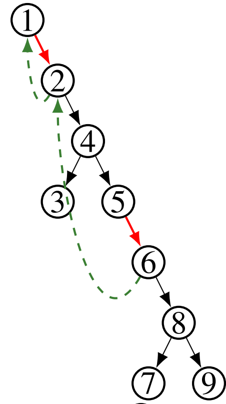
Morris traversal

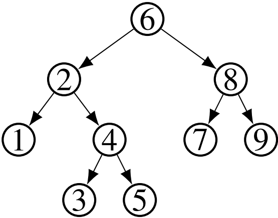
```
curr = root;
while (curr != nil)
do  if (curr.left = nil)
    then inOrderVisit( curr )
         curr = curr.right
    else // find predecessor
         pred = curr.left
         while (pred.right != curr and pred.right != nil)
         do  pred = pred.right
         od
         if (pred->right=nil)
         then // no thread: subtree not yet visited
              pred.right = curr
              curr = curr.left
         else // been there, remove thread
              pred.right = nil
              inOrderVisit( curr )
              curr = curr.right
         fi
    fi
od
```

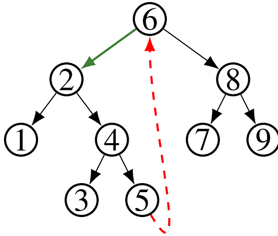
Slide 15 Morris tree traversal. ☒ As tree transformation (top, Drozdek) and as temporary threads (bottom).

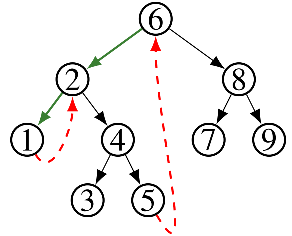












Opgaven

1. Beredeneer dat in een binaire boom met knoop node en inorde opvolger succ precies één van de knopen node.right en succ.left bestaat.

2.a) **(i)** Leg uit welke knopen van de boom er op de stapel staan gedurende de iteratieve pre-orde wandeling die gebruik maakt van een stapel.

(ii) Idem voor de iteratieve in-orde (symmetrische) wandeling.

b) Bekijk het generieke Euler boom-wandel algoritme waarin elke knoop driemaa wordt bezocht (zoals we uitgewerkt hebben bij link-omkering). Maak daarvoor de volgende tabel af, waarbij visit het bezoek aan de huidige knoop telt.

visit	node-test	direction	new visit
1	...	...	...
...	...	...	...

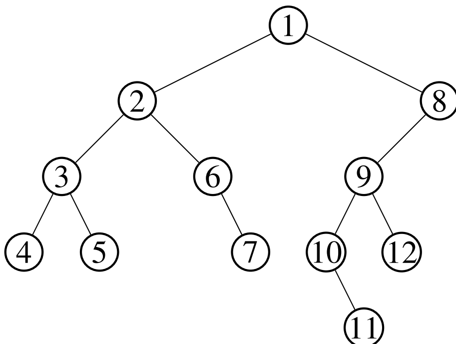
(Alleen de tabel is voldoende, met enige uitleg.) *Jan 2017*

3. Geef een boomwandelalgoritme voor een binaire boomimplementatie waar elke knoop ook een parent-pointer heeft. Dit kan zonder gebruik te maken van recursie of stapel. Kijk naar pre-, in- en postorde varianten van de wandeling.

4. Link-inversie, zie p. 34. Als de boom een binaire zoekboom is, kunnen we dan de waarden in de knopen gebruiken om zonder *tag* door de boom te wandelen met link-inversie?

5. We kijken naar een symmetrisch(=inorde) bedrade boom (*inorder threads*) met alleen draden naar rechts.

a) Neem onderstaande binaire boom over en voeg passende draden toe.



b) Geef een algoritme dat een symmetrische wandeling uitvoert op een bedrade boom, zonder stapel of recursie te gebruiken.

c) Neem aan dat de boom een binaire zoekboom is. Laat zien hoe we een waarde aan de boom toevoegen. Zorg ervoor dat de bedrading van de boom correct blijft. *Jan 2015*

6. Vragen over wandelen in bomen met vaste draden. Zie eerder in de tekst.

a) How to locate the parent of a node in a threaded tree? (Neem hier standaard draden, dus inorder.)

b) Can we perform other traversals (preorder, postorder) using inorder threads?

c) **(i)** Can we add preorder threads, and do preorder traversals?

(ii) Same question with postorder.

3 Binary Search Trees

Sets and data

In this section we consider binary search trees, a convenient way to store sets. A classical view of a (relational) data base consists of tables. Each of the rows indentifies a record with its fields (or attributes).

code	vak	docent
CMPA6	Computerarchitectuur	Rietveld
CNPRE	Concepts of Programming Languages	Hiep
DATAS	Datastructuren	Hoogeboom
AUTTH	Automata Theory	van Vliet
SECU6	Security	Gadyatskaya

This table stores a relation, mathematically that is a set consisting of tuples, $\{\,(\text{CMPA6}, \text{Computerarchitectuur}, \text{Rietveld}),\ (\text{CNPRE}, \text{Concepts}, \text{Hiep}), \dots\}$.

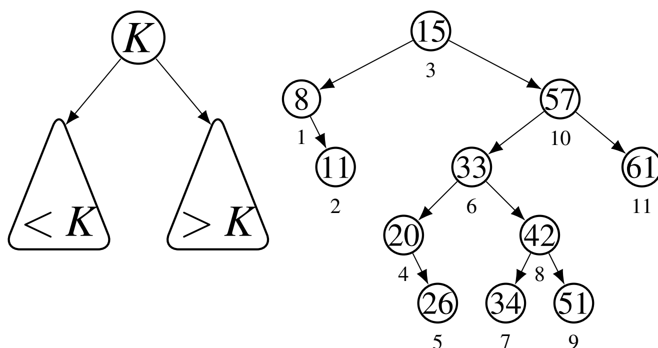
In this lecture we will consider that each record has a *key*, for example the code of a course, and we will store records by their keys. The other fields will be called the *value* of the key.

If we are only interested in the set of keys, then the abstract data type is called the SET. If we are interested in the key-value pairs then the ADT is called a MAP or DICTIONARY. Given the key, we are able to locate it in the set and we can retrieve the corresponding value.

In the examples the keys will be simply numbers, but they can come from any domain that can be ordered, like strings or calendar dates.

3.1 Representing sets

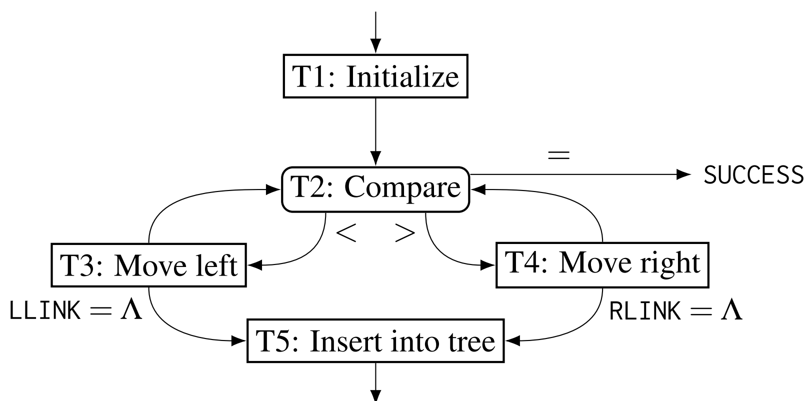
A *binary search tree* (BST) is a binary tree (with keys in a totally ordered domain) such that if a node holds key K, then all descendants to the left hold a value less than K, and all descendants to the right a value larger than K.



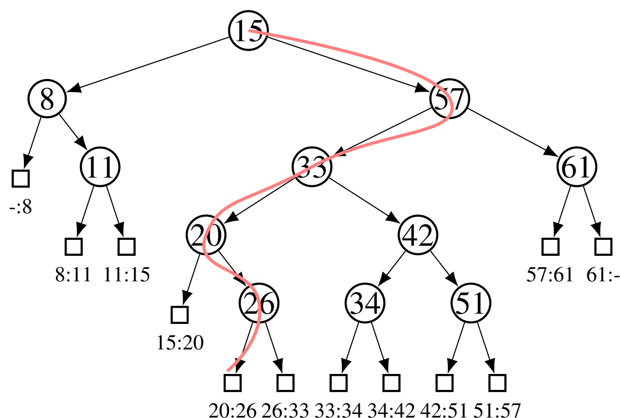
Lemma 3.1. *Let T be a binary search tree. Then the inorder traversal of T visits all the nodes in increasing order.*

Proof. By induction on the structure of T. If T has root K, and subtrees T_1 and T_2 then the inorder traversal visits T_1, K, T_2. By induction we may assume the inorder traversals of T_1 and T_2 are increasing. Since all keys in T_1 are smaller than K, and those in T_2 are larger, we obtain an increasing order. □

Binary search trees have a simple algorithm to search for keys, or for the position to add new keys. For fun we give the diagram from TAoCP by Knuth (see the 'Standard Reference Works' on Page 181).



Note that a successful search ends in one of the internal nodes of the BST, whereas an unsuccessful search (failure to find the key) leads to an external leaf in the tree. That external leaf corresponds to an interval between the keys.



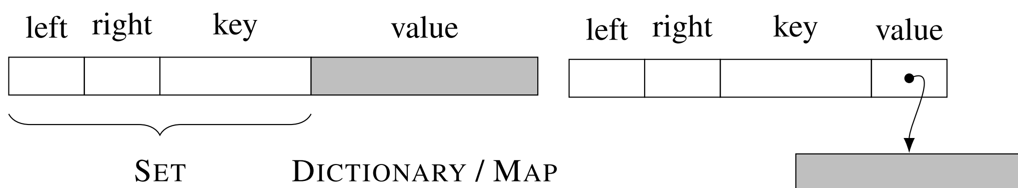
ADT Set and Dictionary. Binary search trees are the basis for many implementations for the abstract data structure SET. Such a set represents a [finite] subset of a *totally ordered* domain D: for each pair $u, v \in D$ we have either $u < v$, $u = v$ or

$u > v$. (This is not an obscure requirement: both numbers and strings are totally ordered. It allows us to go left/right at each vertex depending on the value of the key compared to the value stored at the vertex.)

The ADT SET has the following operations:

- INITIALIZE: construct an empty set: $A = \varnothing$.
- ISEMPTY: check whether there the set is empty (contains no elements): $A \stackrel{?}{=} \varnothing$.
- SIZE: return the number of elements, the cardinality of the set.
- ISELEMENT: returns whether a given object from the domain belongs to the set: $a \in A$.
- INSERT: add an element to the set (if it is not present): $A \cup \{a\}$
- DELETE: removes an element from the set (if it is present): $A \setminus \{a\}$.

When SET is implemented using a binary search tree traversal algorithms can be used to iterate over the elements of a set. We can define an abstract kind of *iterator* to the set by adding the operation SUCCESSOR, which gives the next element in the set order (of course we also need constants for the 'first' and 'last' element of the set to start and stop the traversal).



The ADT DICTIONARY (or Map, or Associative Array) stores a set of (key,value) pairs. For each key in the dictionary there is only one value stored (so it is a *functional* relation). The generalization is called a MultiMap, where several pairs with the same key can be stored.

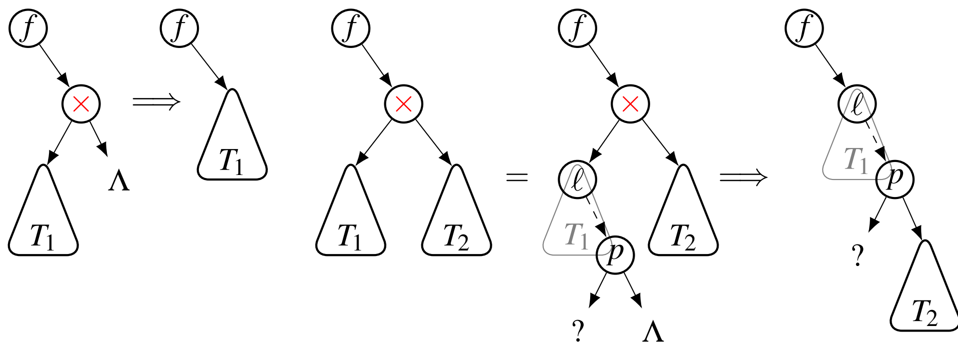
The operation RETRIEVE returns the value for a given key in de dictionary (and a proper error when it does not exist). Likewise, the operations ISELEMENT and DELETE work as in a SET and need only the key as input. Of course, for INSERT we need both key and value to add the (key,value) pair to the set, provided no pair with the given key is already in the dictionary. (Otherwise, depending on the particular details, we may either overwrite the existing key, of return an error.)

In the next section, Chapter 4, we improve the efficiency of BST implementation for SET by considering balanced trees. For other possible implementations of SET and DICTIONARY, see Chapter 6 on B-trees and Chapter 8 on Hash Tables. Of course, even a basic implementation using linear lists is possible.

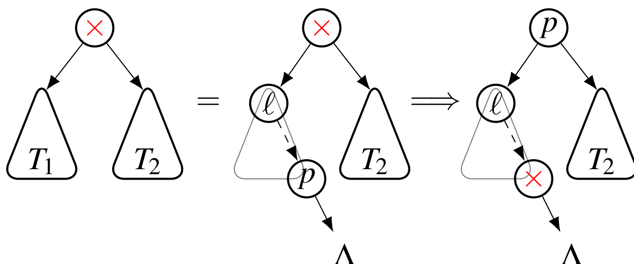
Deletion. The task is to delete a key x while preserving the inorder of the remaining keys in the tree. The method depends on the number of children of x. **(0)** Of course a leaf can be safely removed. **(1)** Also, for a node with a single child the process is simple: the subtree of the child is linked to the parent f of x, replacing x. Note that in both cases we need not only need the node x, but also its parent f, to adjust one of its children.

(2) The last case, deleting node x with two children can be handled in two different ways, *by merging*, or *by copying*. In both cases we locate the inorder predecessor p of x. This node is the right-most node in the left subtree T_1 of x. As p is rightmost, it does not have a right child, marked with Λ in the diagrams. We don't know whether p has a left child, but that is not important.

Deletion by merging. **(2a)** This solution merges the two subtrees T_1 and T_2. T_2 becomes the right subtree of p. Now x has only a single child, and T_1 becomes the right subtree of parent f, like in case (1) above.

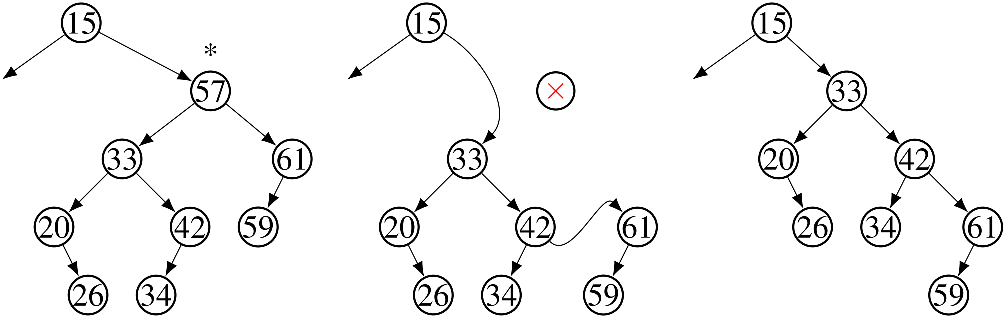


Deletion by copying. **(2b)** Again x has two children, and we locate its inorder predecessor p. Now copy p into x. We now can remove the original node p [marked x in the diagram below, for deletion], which has at most one child, using the previous cases. Note that only temporarily p and x are in the wrong order in the BST, but since x is deleted afterwards, this does not harm the search tree property.

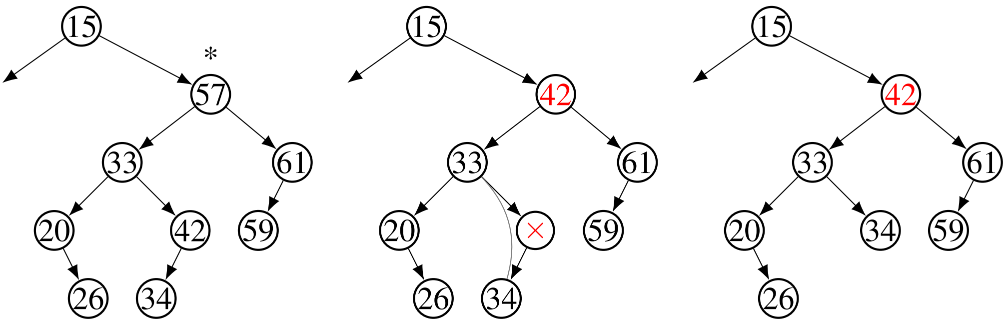


Example 3.2. We want to delete key 57 from the tree below. Node 57 has two children. Its predecessor is 42.

Deletion *by merging* moves the right subtree of 57 to become the right subtree of 42. Now 57 has only a left child, and can be removed while linking its only child 33 in its place.



In deletion *by copying* the key 42 is moved to the node 57. The old 42 has only a left child, and can be removed while linking its only child 34 in its place to become a child of 33.



◇

3.2 Augmented trees

Binary search trees are a classic way to store a set of keys, using an ordering on those keys. As we have such an ordering it might be logical to ask for the kth key in the ordered set, like we can ask for the kth element in an array (k is variable here, not fixed). [Of course, storing the set in an ordered array would make this a constant-time operation, unfortunately, updates to the set would then cost linear-time.] Note that in a binary tree we can efficiently search for a given key, but we do not know what is the inorder number of each key in the tree.

An elegant solution is to *augment* the binary tree with additional information. With each node we store its *size*, the number of nodes in its subtree (including the node itself). An empty subtree has size 0.

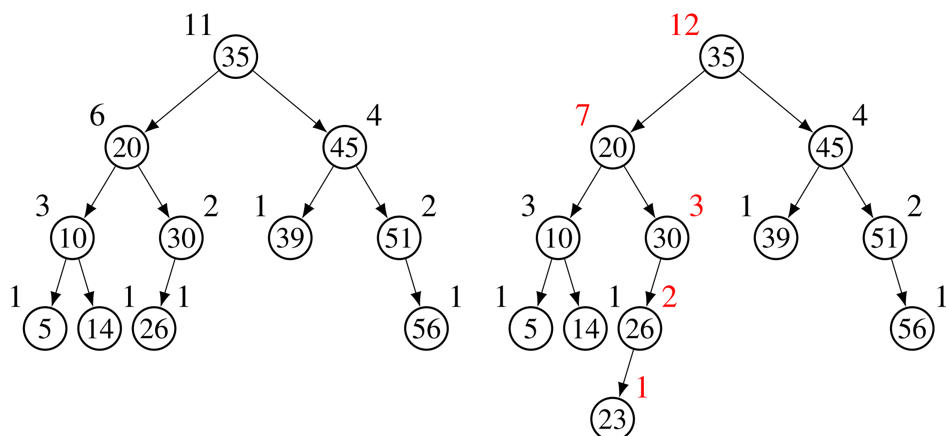
Now the kth key can be located based on this information. At the node we determine the number r, the size of its left subtree plus one.

- if $k = r$ we have located the key,
- if $k < r$, locate kth element in left subtree, and
- if $k > r$, locate $(k - r)$-th element in right subtree.

This concludes the algorithm, except that we have to stop with an error message at empty trees.

Updating sizes when adding a key to the tree is easy and can be done while inserting (if we know in advance that the key is not already in the tree). At each node on the path to the leaf where the new key is inserted we add one to the rank. A new leaf gets rank 1.

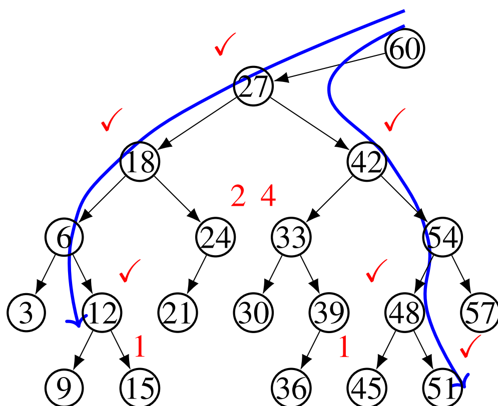
Example 3.3. Binary tree, nodes marked with their rank. Same tree, with updated ranks, after inserting 23.



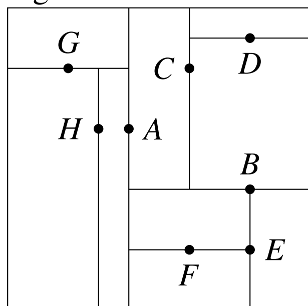
◇

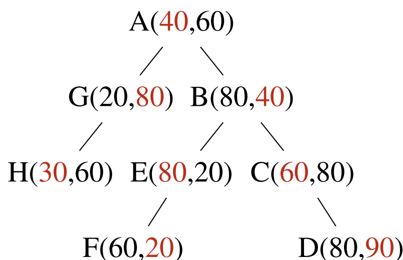
Range Query. Binary trees can be used to perform *range queries*, answering the question which keys in the tree fall in a given interval $[\ell, r]$. A related task is to report only the number of these keys, not the actual keys. To find the keys in the interval (or their number) search in the tree for both ℓ and r. The last node where the paths to ℓ and r are the same is interesting, it is the *split node* (which must belong to the interval). We start reporting/counting from there. All branches that point 'inside' the right and left paths are included, thus counted or reported, as well as the nodes on the paths that are in the interval. In essence we either report full subtrees, and single nodes on the boundary paths. The implementation has to distinguish whether a key has actually been found or not. In the first case the search does not reach a leaf. For frequent counting tasks we store rank information, see above.

Example 3.4. We want to count the number of nodes in the interval $[12, 52]$. The split node is 27. On the boundary paths we count six nodes, marked ✓. Apart from these we have subtrees with $1+2+4+1$ nodes, for a total of 14 in the interval.

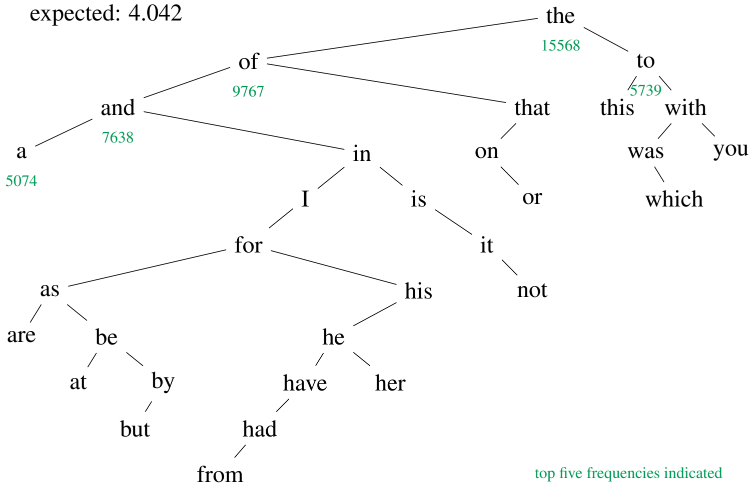


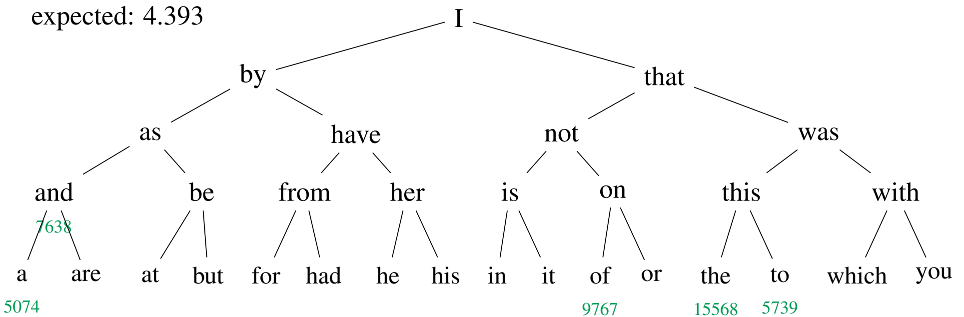
Range Query. *k*-d-tree. Storing multidimensional data in a BST by alternatingly sorting on one of the coordinates.

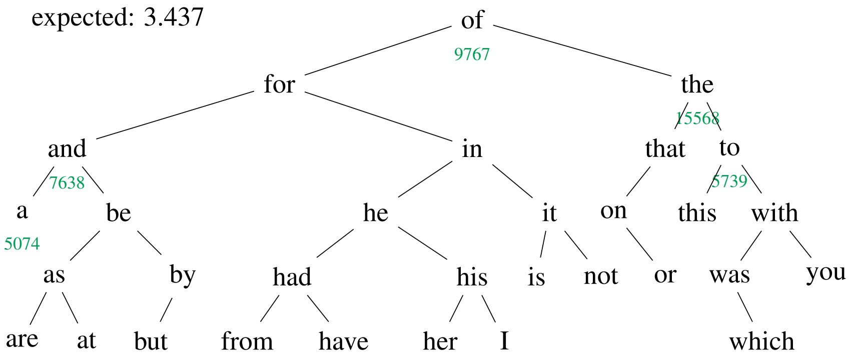




Slide 16 Knuth (Section 6.2.2) constructs BST's with 31 most common words in English. (1) Adding the words in decreasing order of frequency, average number of comparisons of 4.042. (2) A perfectly balanced search tree, 4.393 comparisons. (3) Using dynamic programming 3.437 in the optimal tree. In green, the frequenceis of the five most occurring words.







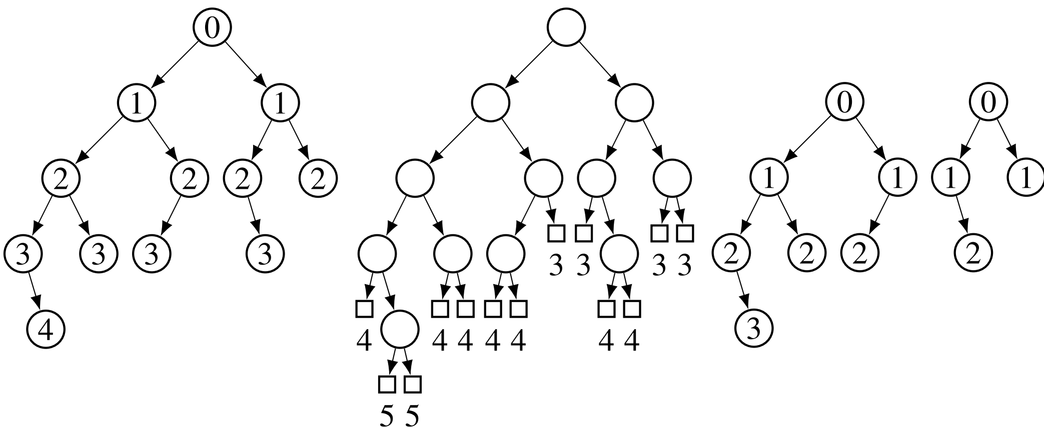
3.3 Comparing trees

In this section we develop a measure for (binary search) trees. It indicates the efficiency of the tree and is related to the number of steps (key comparisons) needed to locate a key in the tree.

An *extended binary tree* is a binary tree where the empty left and right subtrees of nodes missing one or more children are replaced by specially marked leaves. Such an extended tree is *full* (every node is either a leaf or has two children).

Definition 3.5. The *internal path length* I of a binary tree is the sum of the pathlengths (number of edges from the root) to each of the nodes in the tree.
The *external* path length E is the total pathlength to each of the leaves in the extended binary tree.

Example 3.6. Left: A binary tree T with $n = 12$ nodes. Its internal path length equals $I = 3+4+2+3+1+3+2+0+2+3+1+2 = 26$. Middle: Its extended binary tree. The external path length of T equals $E = 4+5+5+4+4+4+4+3+3+4+4+3+3 = 50$.



Right: the two subtrees of T. Their internal path lenghts are $I_1 = 2+3+1+2+0+2+1 = 11$ and $I_2 = 1+2+0+1 = 4$, respectively.

◇

Lemma 3.7. *Let T be a binary tree with n nodes, with internal path length I and external path length E. Then $E = I + 2n$.*

Proof. By induction on the structure of the tree.

Basis. Start with the empty binary tree, with 0 nodes. Its extended version has a single leaf (as root!). For this tree we have $n = I = E = 0$, as the distance is measured in edges, so the level of the root is 0.

Induction. Assume T is a binary tree with root and subtrees T_1, T_2. We assume as hypothesis that for both subtrees the equation $\mathrm{E}_i \overset{(H)}{=} \mathrm{I}_i + 2n_i$ holds (with obvious meaning of the variables; '(H)' is the 'name' of the equation we use below).

We observe that $n \overset{(N)}{=} n_1 + n_2 + 1$. All nodes in the subtrees of T are one level deeper than in their original position (as separate trees). Thus we have to add 1 for each node to move from I_1 and I_2 for the subtrees to I for T, hence in total $n_1 + n_2$. The new root has zero path length, so has no contribution. (See Example 3.6, left and right diagrams.) So $\mathrm{I} \overset{(I)}{=} \mathrm{I}_1 + \mathrm{I}_2 + n_1 + n_2$.

Similarly $\mathrm{E} \overset{(E)}{=} (\mathrm{E}_1 + n_1 + 1) + (\mathrm{E}_2 + n_2 + 1)$, each extended leaf is one level deeper in the combined tree; recall that the (extended) binary tree with n_i nodes has $n_i + 1$ external leaves.

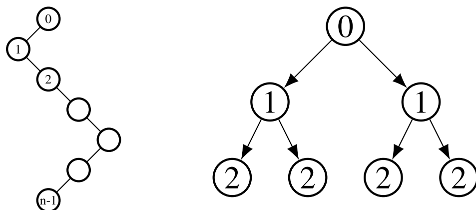
We combine these formulas: $\mathrm{E} \overset{(E)}{=} \mathrm{E}_1 + \mathrm{E}_2 + n_1 + n_2 + 2 \overset{(H)}{=} (\mathrm{I}_1 + 2n_1) + (\mathrm{I}_2 + 2n_2) + n_1 + n_2 + 2 = (\mathrm{I}_1 + \mathrm{I}_2 + n_1 + n_2) + 2n_1 + 2n_2 + 2 \overset{(I)}{=} \mathrm{I} + 2n_1 + 2n_2 + 2 \overset{(N)}{=} \mathrm{I} + 2n$. □

The path length to a certain node on a tree is one less than the number of nodes visited, or the number of key comparisons to find that key in the tree. So, the *average* number of key comparisons for a *successful* search in a given binary tree equals $\frac{\mathrm{I}}{n} + 1$, where n is the number of nodes (keys) and I the internal path length. Here we assume that every node has the same probability of being searched for.

In the same way the average number of key comparisons in the case of *failure* (key not in the tree) equals $\frac{\mathrm{E}}{n+1}$, assuming each interval between keys has the same probability.

Extremal trees. The worst tree in terms of path length is linear. The keys are on successive levels. With n keys the internal path length of a linear tree equals $\mathrm{I}_n = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

Thus the average search time for keys in a linear tree with n keys equals $\frac{\mathrm{I}_n}{n} + 1 = \frac{n+1}{2}$.



At the other end is the perfectly balanced tree, with completely filled levels. Say the tree has height h, counting the number of edges, then it has $n = \sum_{i=0}^{h} 2^i =$

$2^{h+1}-1$ nodes. Thus conversely, its height in terms of the number of nodes equals $h = \lg(n+1)-1$.

If we sum the height of all the nodes by level, we find one (2^0) node on level 0, two (2^1) nodes on level 1, four (2^2) nodes on level 2, and in general 2^i nodes on level i. Thus the internal path length of the perfect binary tree of height h corresponds to the following formula.

Lemma 3.8. $\sum_{i=0}^{h} i \cdot 2^i = (h-1)\cdot 2^{h+1}+2$.

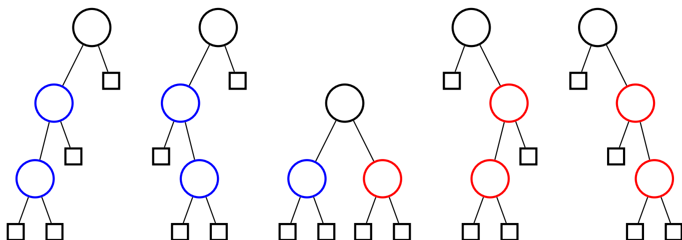
Proof. By induction on h. Basis for $h=0$: $0\cdot 2^1 = (-1)\cdot 2^1+2$.
Induction step. Assume the formula holds for h. We show the formula for $h+1$. $\sum_{i=0}^{h+1} i\cdot 2^i = \sum_{i=0}^{h} i\cdot 2^i + (h+1)\cdot 2^{h+1}$. We can use the induction hypothesis for the summation. We get $\big((h-1)\cdot 2^{h+1}+2\big)+(h+1)\cdot 2^{h+1} = 2h\cdot 2^{h+1}+2 = h\cdot 2^{h+2}+2$. □

In terms of the number of nodes, $n = \sum_{i=0}^{h} 2^i = 2^{h+1}-1$, we obtain the internal path length $I_n = (h-1)\cdot 2^{h+1}+2 = [\lg(n+1)-2]\cdot(n+1)+2 = (n+1)\lg(n+1)-2n$.

A simpler way to obtain the internal path length is via the external path length, and the correspondence between the two values. The perfect tree has $n+1 = 2^{h+1}$ leaves, each at depth $h+1$, so $\mathrm{E}_n = 2^{h+1}\cdot(h+1) = (n+1)\lg(n+1)$. Then $\mathrm{I}_n = \mathrm{E}_n - 2n = (n+1)\lg(n+1)-2n$. Same result.

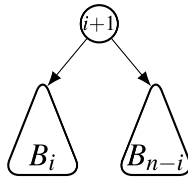
We obtain an avarage search time for a key in the perfect binary tree of $\frac{\mathrm{I}_n}{n}+1 = \frac{n+1}{n}\lg(n+1)-1 \sim \lg n$ (for large n).

Example 3.9. Five binary trees with 3 internal nodes and 4 leaves.



◇

Number of binary trees. We count the number of possible binary trees with n (internal) nodes and $n+1$ (external) leaves. Let B_n be this number. The example above shows the five possible binary trees with three nodes. Thus $B_3 = 5$.



There is an elegant recursive formula for the numbers. Any tree with $n+1$ nodes splits in a root, and two subtrees with i and $n-i$ children. Hence we have the recursion

$$B_{n+1} = \sum_{i=0}^{n} (B_i \cdot B_{n-i}),$$

with $B_0 = 1$.

The sequence starts with $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$.

And indeed we can verify one step in the recursion $B_5 = \sum_{i=0}^{4}(B_i \cdot B_{n-i}) = B_0 \cdot B_4 + B_1 \cdot B_3 + B_2 \cdot B_2 + B_3 \cdot B_1 + B_4 \cdot B_0 = 1 \cdot 14 + 1 \cdot 5 + 2 \cdot 2 + 5 \cdot 1 + 14 \cdot 1 = 42$.

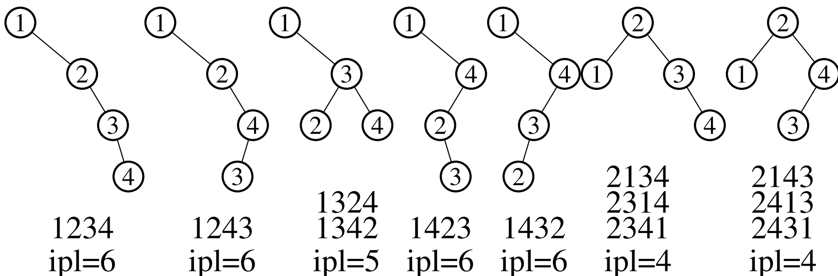
This sequence of numbers is named after Eugène Catalan, a Belgian mathematician. The closed formula for the numbers is

$$B_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

The last approximation of this number is based on Stirling's approximation for factorial numbers. It is very good: the symbol $\sim$ here indicates that the fraction of the two numbers approaches 1 (as n goes to infinity).

There is a huge number of objects that when counted result in the same sequence. See Wikipedia for many examples. Also: OEIS/A000108.

Example 3.10. The BST's resulting from adding a permutation of $1, 2, 3, 4$; remaining trees are the mirror image of these. There are $4! = 24$ permutations and 14 binary trees with 4 nodes. This illustrates (for a very small tree size only!) that the number of permutations that lead to bad linear trees (4 permutations with ipl=6) is less than the better ones with ipl=4 (6 permutations).



(The numbers in the nodes indicate the values entered into the BST, and follow the inorder numbering.)

If we avarage over these trees, taking into account the number of permutations generating them, the average BST has an ipl of $\frac{6\cdot4+2\cdot5+4\cdot6}{12} = 4.83$.

Average tree. Binary search trees are generally built by inserting keys in some random order. The permutation of keys determines the structure of the tree. There are more permutations than possible keys. Linear trees can only be obtained with one permutation, while there are many permutations for more balanced trees, see the above example.

When we consider average trees we mean the average over all possible permutations. It turns out that the average tree has logaritmic search time, close to the optimum. In this paragraph we try to get an idea why that is.

To investigate the searching time in the 'average' binary tree, we get intuition by looking at trees with 2^4-1 nodes. With 15 nodes there is exactly one completely balanced tree (with four levels, the minimum possible) but there are $2^{14} = 16,384$ 'linear' trees (with 15 levels, the maximum). Searching in such a linear tree takes around 7 steps, the height of the average node.

If we start with 15 given keys $1,2,\dots,15$ then there is exactly one permutation that yields each given linear tree structure, if we add the keys in that order in a standard way to an initially empty binary search tree. There are however many permutations (21,964,800 to be precise, oeis.org/A076615) that yield the four level balanced tree, where each of the nodes can be found in at most four steps.

Example 3.11. Consider the BST defined by the permutation (5246173). The root 5 determines that numbers less than 5 go into the left subtree, and those larger go right. That also means that the subpermutations (2413) and (67) determine the form of the two subtrees. The form of the tree is determined by two permutations. One of $1,2,3,4$ and one of $6,7$. If we interleave these two permutations and start with 5 we always get the same tree, as depicted here. The second permutation (67) is equivalent to a permutation of $(1,2)$ (simply by renumbering starting from 1).

In this way, by splitting at the root, we can reduce a large permutation into smaller ones, and compute the average IPL for binary trees.

Telescoping☒. Let $\hat{I}_n$ denote the IPL for a BST of n nodes *averaged* over all permutations. The example above can be extended into an argument for a recursion formula for $\hat{I}_n$. Rather than averaging over permutations we average over subpermutations after the root k has been fixed. (This is not immediate: we have to check that all subpermutations of a certain size occur the same number of times.)

$$\hat{I}_n = (n-1) + \frac{1}{n}\sum_{k=1}^{n}(\hat{I}_{k-1} + \hat{I}_{n-k})$$

With this formula a rather precise evaluation of the average tree can be obtained. First we subtract two successive values to obtain a recursive formula.

so	$I_n = (n-1) + 2(\mathrm{I}_0 + I_1 + \cdots + I_{n-1})/n$
also	$I_{n-1} = (n-2) + 2(I_0 + \mathrm{I}_1 + \cdots + I_{n-2})/(n-1)$
subtract	$nI_n - (n-1)I_{n-1} = 2n - 2 + 2I_{n-1}$
thus	$nI_n = (n+1)I_{n-1} + 2n - 2$

$$\frac{I_n}{n+1} = \frac{I_{n-1}}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)}$$

$$\frac{I_{n-1}}{n} = \frac{I_{n-2}}{n-1} + \frac{2}{n} - \frac{2}{(n-1)n}$$

$$\cdots$$

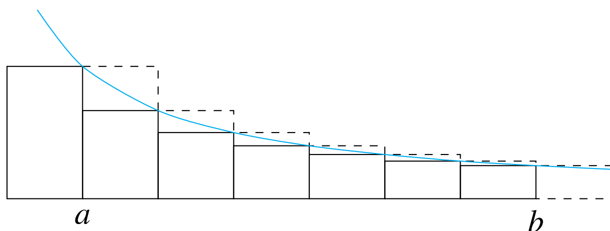
$$\frac{I_1}{2} = \frac{I_0}{1} + \frac{2}{2} - \frac{2}{1\cdot 2}$$

$$\frac{I_n}{n+1} = \frac{I_0}{1} + O(\ln n) - \frac{2n}{n+1}$$

Now substitute each time the next value in the previous one, or add all the successive formulas. We then obtain a telescoping series. One part can be approximated with a logarithm, the other sequences has a precise value, viz. $\sum_{k=1}^{n}\frac{1}{k(k+1)} = \frac{n}{n+1}$.

Approximation. We explain the logarithm in the above formula. The integral of a function is defined as the limit of Riemann sums, computing the area under and above the function. In reverse we can approximate certain summations by their integral. In the diagram below we have a decreasing function f over the interval $[a,b]$. Then

$$\int_a^{b-1} f(x)\,dx \le \sum_{k=a}^{b} f(k) \le \int_{a-1}^{b} f(x)\,dx$$



We can apply this to partial sums of the 'harmonic series' and obtain for instance

$$\frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{n}=1+\sum_{k=2}^{n}\frac{1}{k}\leq 1+\int_{1}^{n}\frac{1}{x}\,dx=1+\lg n$$

References. See DROZDEK Appendix. A.4 Average path length in a random binary tree.

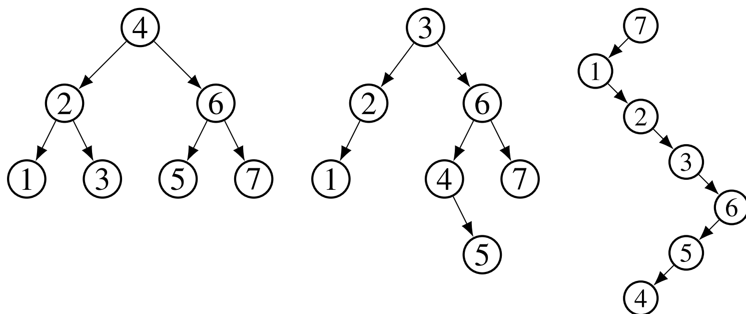
Opgaven

1. Bij het verwijderen in een binaire boom hebben de varianten *by merging* en *by copying* gezien. Beschrijf hoe de algoritmes moeten worden aangepast voor een bedrade binaire boom (als in vorig hoofdstuk). Zorg dus dat na verwijderen weer een correcte binaire boom ontstaat.

4 Balancing Binary Trees

Introduction. From the chapter on binary trees we have learned that perfect binary trees are much more efficient than linear trees when we use them to store sets. A perfect tree retrieves a key in logaritmic time, a linear tree in linear time.

It is however complicated to keep the tree perfect when deleting and adding keys to the trees. This chapter will study height balanced trees: they have both logarithmic search time and logaritmic update time.

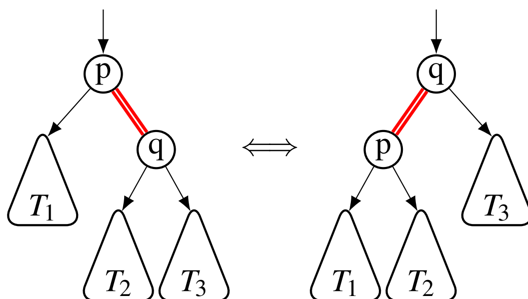


4.1 Tree rotation

Features:

- A tree rotation locally restructures a binary tree, without changing the inorder of the nodes.

In order to obtain new tree structures for the same set of keys we use a simple operation which changes the structure of a tree, while keeping the inorder of the nodes intact. Below we see *single rotation at p to the left*, or (read in reverse) a single rotation at q to the right. Note the operation changes the root of the (sub)tree, a detail which is important when implementing it: we need access to the parent in order to change the link.

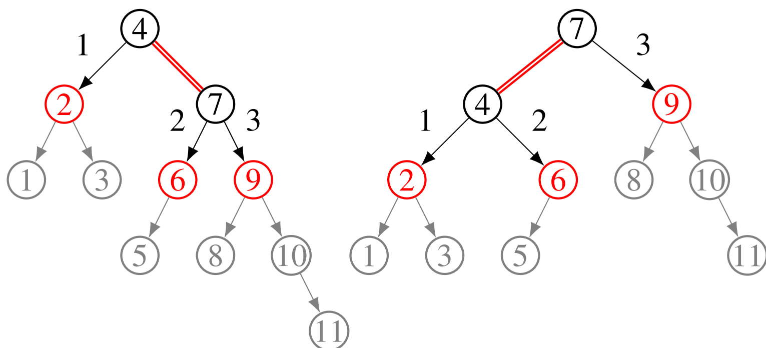


The structure of the trees can be described as $T_1\, p\, (T_2\, q\, T_3)$ and $(T_1\, p\, T_2)\, q\, T_3$, respectively. Thus tree rotations are a form of associativity. It is a crucial obser-

vation that the inorder in both trees reads $T_1\, p\, T_2\, q\, T_3$. Thus, if we start with a BST the result will again be a BST, but with a slightly changes shape.

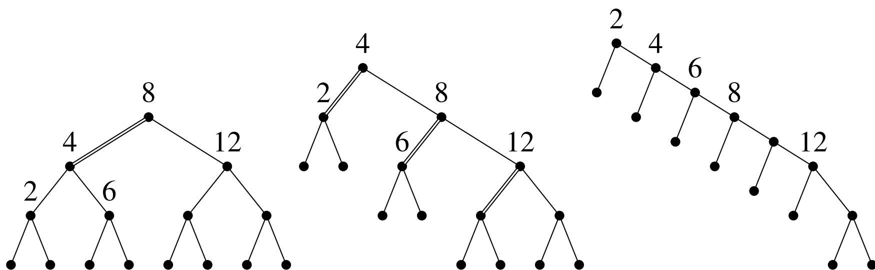
Rotations are "universal", in the sense that every binary tree can be transformed into another binary tree (with the same number of nodes) using rotations. This is because each binary tree can be transformed into a right-linear one (which has only descendants to the right) by repeatedly rotating to the right at any node that has a left child.

Example 4.1. Rotating a tree at the root over the edge $(4,7)$. The three subtrees of the two nodes 4 and 7 are indicated by their roots $2,6,9$. Note that before and after rotation these subtrees keep their shape and order.



Perfect trees. ☒ A *perfect* binary tree T_n of height n has n levels, each of which is completely filled. Such a tree has $2^n - 1$ nodes, we assume here they are numbered $1, 2, \ldots, 2^n - 1$. Rotating it into a right-linear tree can be done systematically: First rotate halfway, at the root which has number 2^{n-1}. Then repeat at the new root and two descendant to the right at $k \cdot 2^{n-2}$, $k = 1, 2, 3$; again at the new root and six descendants to the right at $k \cdot 2^{n-3}$, $k = 1, \ldots, 7$; etcetera.

Reversing the process we can make a perfect balanced tree out of a linear one in a very systematic way, alternatingly rotating nodes on the rightmost "backbone" of the tree.



This is the heart of the DSW algorithm presented in Drozdek. Any binary tree can be rotated into an almost perfect tree, where all levels are completerly filled

except possibly the last one. First rotate the tree into a linear tree, by rotating all edges into edges to the right. Then conversely rotate into a tree of optimal form.

References. See DROZDEK 6.7.1 The DSW Algorithm.

T.J. ROLFE. One-Time Binary Search Tree Balancing: The Day/Stout/Warren (DSW) Algorithm. Inroads (ACM SIGCSE) **34** (2002) 85–88. [Special Interest Group for Computer Science Education]

4.2 AVL Trees

Features:

- Height balanced binary search tree, logarithmic height and logarithmic search time.
- Rebalancing after adding a key using (at most) one single/double rotation at the lowest unbalanced node on the path to the new key.
- Rebalancing after deleting a might need a rotation at every level of the search path (bottom-up).

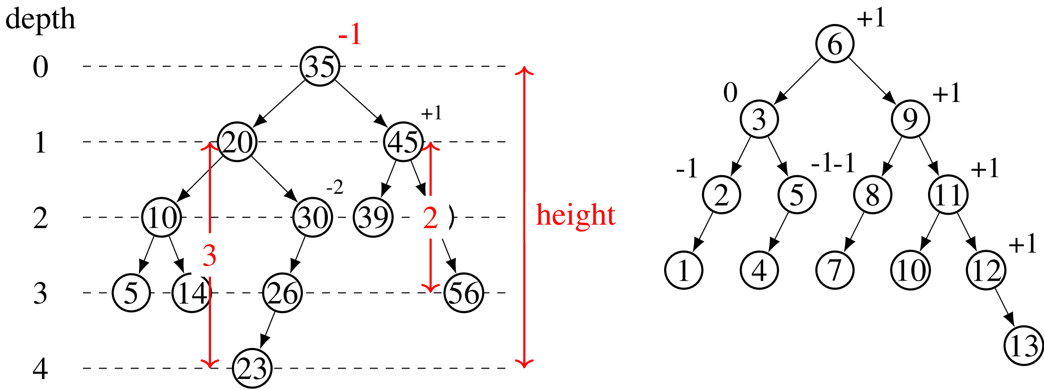
Definition 4.2. An *AVL-tree* is a binary search tree in which for each node the heights of both its subtrees differ by at most one. The difference in height at a node in a binary tree (right minus left) is called the *balance factor* of that node.

These trees are named after their inventors, Adelson-Velsky and Landis who presented their idea already in 1962[3]. Enforcing the structure ensures the tree is balanced, in the sense that the height of the tree is roughly logaritmic in the number of nodes, guaranteeing logarithmic tree search, while on the other hand the structure is not too strict, and updates for adding and deleting a node can also be performed in logarithmic time.

Example 4.3. (Left) Here we illustrate the height of binary trees, measured in number of edges from root to the deepest leaf. The height of a subtree is measured from its root.

(Right) An AVL-tree; nodes numbered in inorder. With each node we give its balance factor (except for the leaves, where it is 0).

[3] Адельсон-Вельский and Ландис – the transcription of the Russian into English may vary (I can't type Cyrillic, but copy-paste works fine in modern LaTeX).

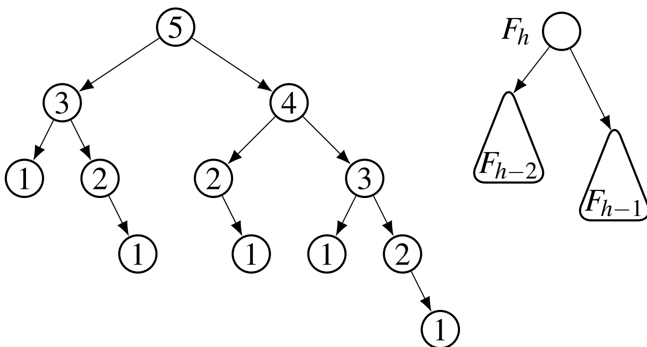


◇

Minimal and maximal trees. For a fixed height h, the AVL tree with the maximal number of nodes has all its balance factors equal to 0, the tree is perfectly balanced, and contains $2^h - 1$ nodes.

For a given height h, the AVL tree with the minimal number of nodes has all its balance factors equal to ± 1 (except the leaves). A tree with this shape is called a *Fibonacci tree*. It can be defined recursively. The Fibonacci trees F_0 and F_1 consist of no nodes, and one node, respectively. Fibonacci tree F_{h+1} consists of a root, and two subtrees which are equal to F_h and F_{h-1}, respectively.

Example 4.4. In the picture below we draw a Fibonacci tree of height 5 – each node marked h is the root of a Fibonacci tree of height h.



◇

Exact analysis. If we denote the number of nodes in F_h by f_h, then we have $f_{h+1} = 1 + f_n + f_{h-1}$, or $(f_{h+1} + 1) = (f_h + 1) + (f_{h-1} + 1)$, with $(f_0 + 1) = 1$ and $(f_1 + 1) = 2$: This is the recursion of the well-known Fibonacci numbers! But starting two positions later.

Let Φ_n be the nth Fibonacci number (starting with $\Phi_0 = 0$ and $\Phi_1 = 1$). According to Binet's formula the Φ_n equals $\Phi_n = \frac{1}{\sqrt{5}}(\varphi^n + (1-\varphi)^n)$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.62$. Note that $(1-\varphi)^n \sim 0.62^n \to 0$ as $n \to \infty$. So, for large n, $\Phi_n \sim \frac{\varphi^n}{\sqrt{5}}$.

Then $f_h + 1 = \Phi_{h+2} \sim \frac{\varphi^2}{\sqrt{5}}\varphi^h \approx 1.17 \cdot 1.62^h$. This is the minimal number of nodes in an AVL-tree of height h. As we have seen earlier that the maximum number of nodes is $2^n - 1$, both bounds grow exponentially with h. Conversely, this shows the height h of the AVL tree is logarithmic in the number of nodes.

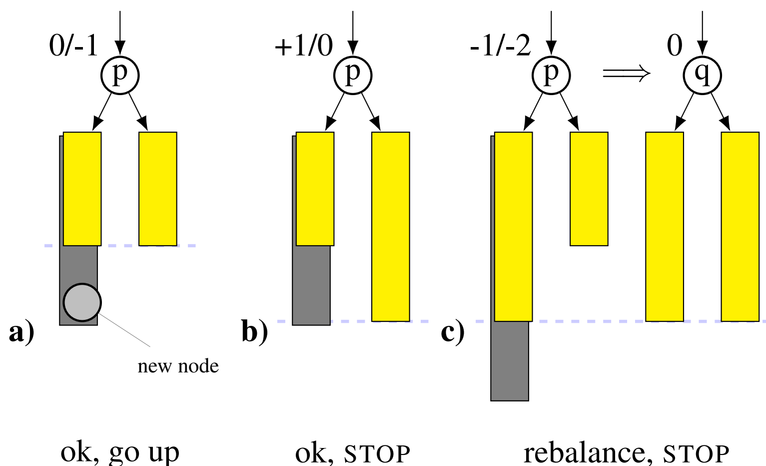
Quick and easy analysis. The above analysis gives a very precise value for the minimal number of nodes in an AVL tree of height h. A quick and easy observation shows that in an AVL tree of height $2h-1$ or height $2h$ the top h levels are completely filled: if the height at the root is h then at least one of the children has height $h-1$, the other has at least height $h-2$.

So $f_{2h} \geq 2^h - 1$, and we easily obtain a logarithmic bound on the length of the longest path in the tree given its number of nodes. If an AVL tree has height $2h$ (or more) than it has at least $n = 2^h - 1$ nodes. Contrapositive: If an AVL tree has less than n nodes, it must have height less than $2h = 2(\lg n - 1)$.

4.3 Adding a Key to an AVL Tree

As an AVL tree is a binary search tree, adding a key has to take place at a leaf which is in the proper position of the inorder of keys. After adding the key the AVL property may no longer hold. We then have to restructure the tree. It turns out that it suffices to perform an operation at a single node on the path from root to the fresh leaf. We discuss here how that position can be found.

Bottom-up view. After inserting the new key we follow the path from the new leaf back to the root, and see how the new balance factors are computed; and in particular where rebalancing has to take place. For simplicity we assume that the new key has been inserted into the left subtree, and also that the balance has been decreased (i.e., the height of left subtree has increased). Note that balance factors outside the path from the root to the new leaf are not changed.



We distinguish three cases:

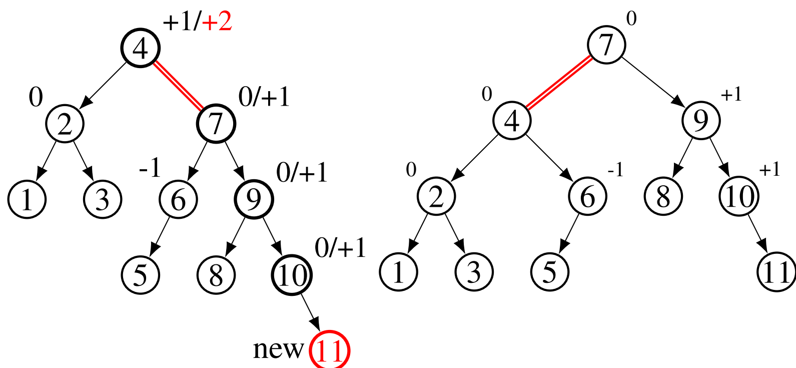
- **a)** balance old/new $0/-1$. Balance OK, the height of this tree increased, CONTINUE to father.
- **b)** balance old/new $+1/0$. Balance OK, the height of this tree did not change: STOP, the balance factors above this point do not change.
- **c)** balance old/new $-1/-2$, then changed to 0. New balance out of bounds. A rotation will restore balance (explained later) and the new balance at this point will become 0. Moreover the rebalanced tree at this point has the same height as the original tree; STOP, the balance factors above this point do not change.

From this overview we learn an important fact: *at most a single rebalance operation* is needed, at the lowest point on the path from the root to the new leaf

where the original balance was not 0. At this point the orignal balance before inserting ± 1 either improves to 0 after inserting (and we stop) or changes to inadmissible ± 2, where we rebalance (and stop). The new balance factors below this point become ± 1 depending on the side the path continues.

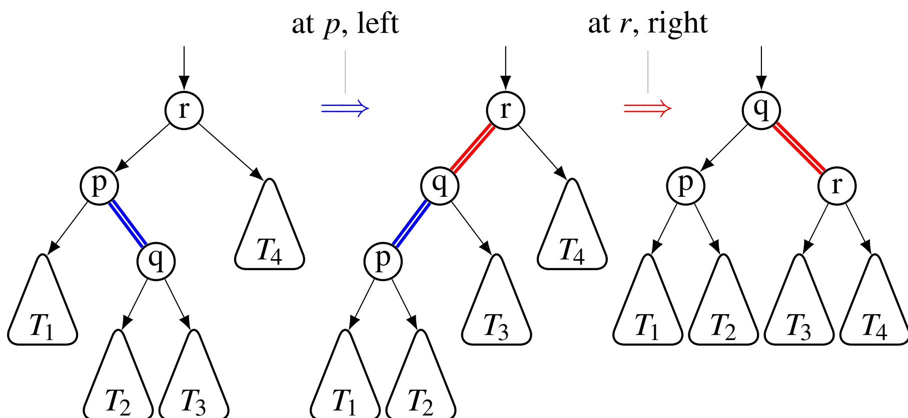
We did not yet explain *how* to rebalance, which we will do shortly.

Example 4.5. A tree that results from adding a key 11 to an AVL-tree, with before and after balance factors given. (Changes occur only along the path from the root to the inserted leaf.) The resulting tree is out of balance at the root. It can be rebalanced by a single rotation at the root to the left. (This is like Example 4.1, but in an AVL context.)



We give the final details of the algorithm that rebalances the tree after addition of a key, by specifying the operation performed in case **c)** above, at the lowest node that is out of balance. In order to do so, we use the *double rotation* as described in the diagram below.

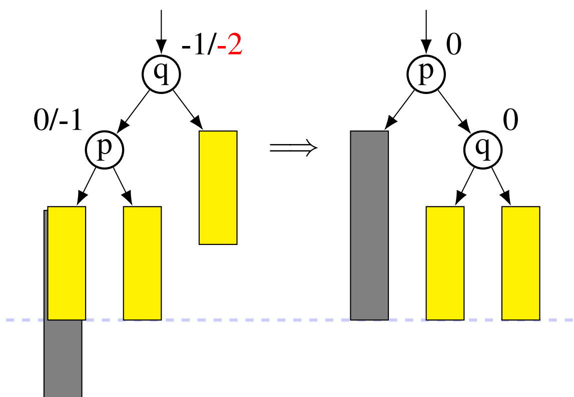
Depicted is a double rotation at r to the right; it is composed of two single rotations. One at p to the left, followed by one at r to the right.



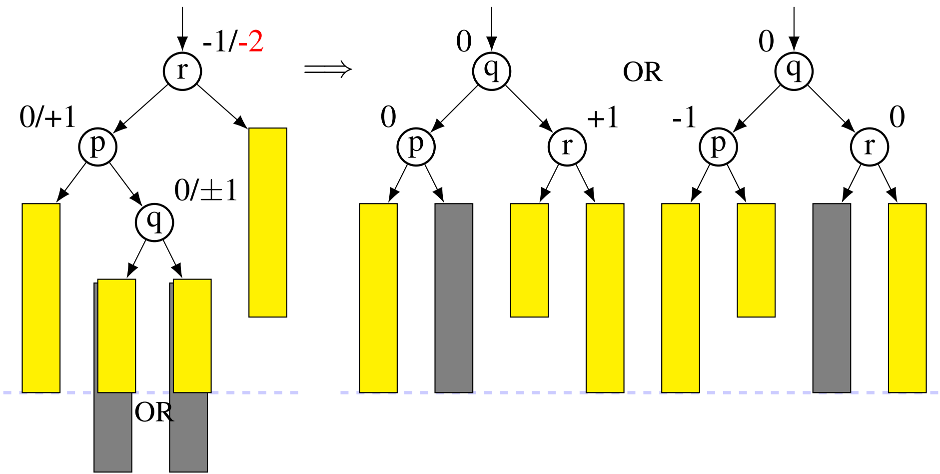
An example of double rotation can be seen in Example 4.6, somewhat below. (We usually draw the result of a double rotation in one step, omitting the middle tree diagram above.)

The precise rotation chosen by the insertion algorithm depends not only on the subtree where the key was added, but also on the subtree another a level below that. We distinguish the **LL** and **LR** cases. Note that the diagrams do not only specify the rotation chosen, but also the new balance factors. These can be “hard coded” and do not have to be recomputed in the tree (as they follow from the type of rotation). The **RR** and **RL** cases are not specified here, they are mirror images of the given two cases.

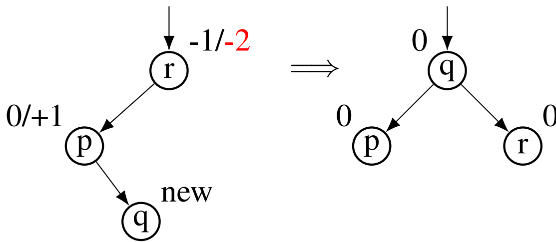
LL-case. At the lowest non-zero balance, the key moves left-left. We perform a **single rotation** to the right, as sketched. Balance OK, the height of this tree did not change: STOP, the balance factors above this point do not change.



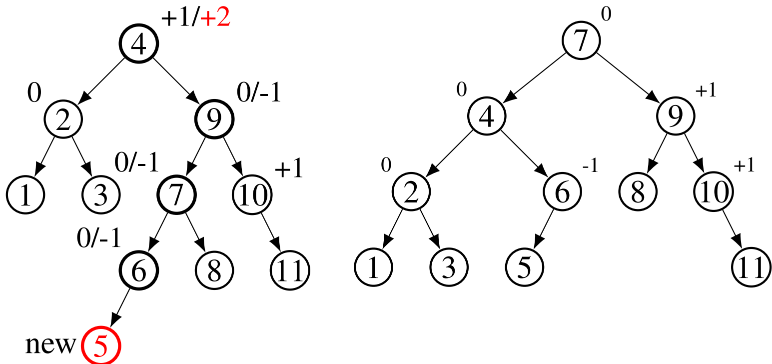
LR-cases. At the lowest non-zero balance, the key moves left-right. We perform a **double rotation** to the right, as sketched. If we want to know the new balance factors we have to look at the next level, see the diagrams. Balance OK, the height of this tree did not change: STOP, the balance factors above this point do not change.



This LR-case includes an additional situation where q itself was inserted, see next diagram.

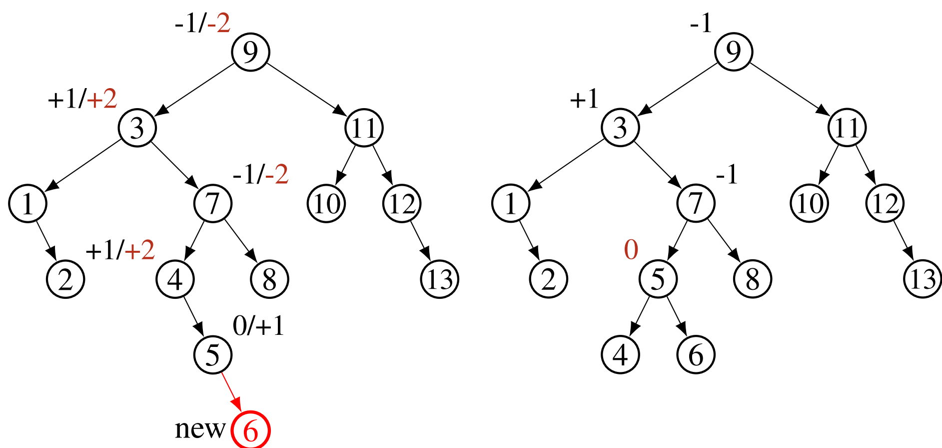


Example 4.6. A tree that results from adding a key 5 to an AVL-tree, with before and after balance factors given. The resulting tree is out of balance. It can be rebalanced by a double rotation at the root to the left.



In the second example we add node 6 to an AVL tree. In the path from root to new leaf all balance factors change. Moreover at almost all these levels the tree is out of balance. Following the algorithm, the balance can be restored using one

rotation, at the lowest level of unbalance, node 4. The subtree at 5 replaces the old subtree at 4, and these trees have the same height. That means that all balance factors above this point keep their original values.



◊

Overview. Adding a key. Adjusting balance factors and structure bottom up:

1. $0 \mapsto \pm 1$ (*go up*)
2. $\pm 1 \mapsto 0$ (*done*)
3. $\pm 1 \mapsto \pm 2$ (lowest position of unbalance in tree)
   (a) LL RR single rotation
   (b) LR RL double rotation
   (*then done*)

Implementation. At each node the balance factor has to be known. It is sufficient to store the values $0, \pm 1$ as we have shown here, because the new balance factors can be predicted from the old ones, as the case analysis above has shown. However some implementations find it worthwhile to explicitly store the height at each subtree. Our sketch is close to an iterative implementation, but elegant recursive solutions are possible.

4.4 Deletion in an AVL Tree

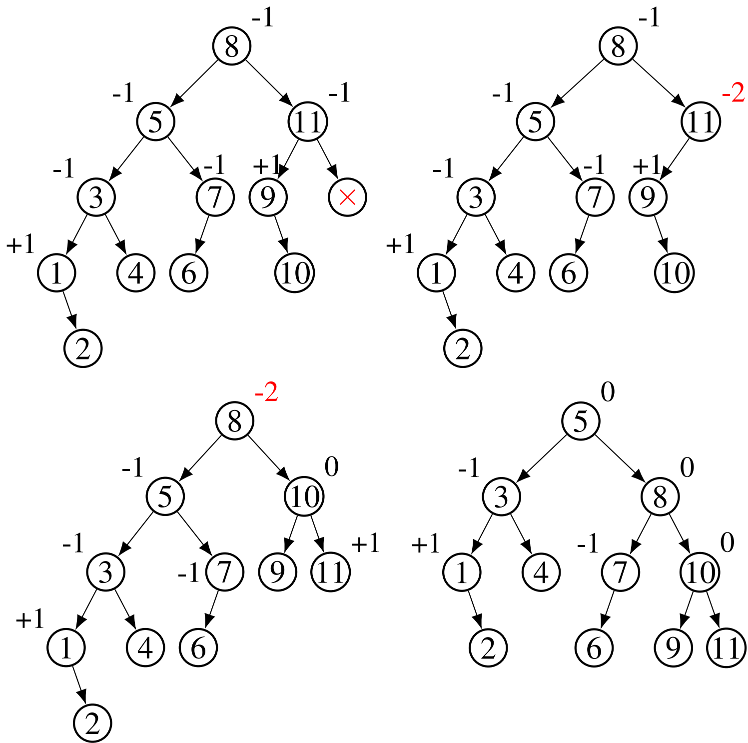
When deleting a node in an AVL tree one usually reduces to the case where the node has only a single child, like for general BST's. However apart from maintaining the inorder of keys here we additionally need to restructure the tree when it has become unbalanced. This can be done by careful case analysis, like in the case of addition. See Slide 17. Now however it is not enough to rebalance at a single node. Instead after one rotation the final subtree at that node may be less tall than

the original subtree. That means that also one level up the balance factors need to be addressed. Etcetera. Nevertheless, as the height of the tree is logarithmic, also the number of operations here is logarithmic at most.

Some people prefer *lazy deletion*, where a key keeps its position in the tree, but is marked as "deleted". The drawback of this choice is that the tree will keep growing when insertions and deletions are used intensively.

Example 4.7. Consider the first AVL tree below. (Note it not-accidentally is 'minimal', i.e., of Fibonacci proportion, see Section 4.2.) Delete key 12 from the AVL tree. At its parent, node 11, the tree becomes unbalanced. We are in a LR situation (which can be seen by the balance factors at 11 and 9), hence we do a double rotation to the right at 11.

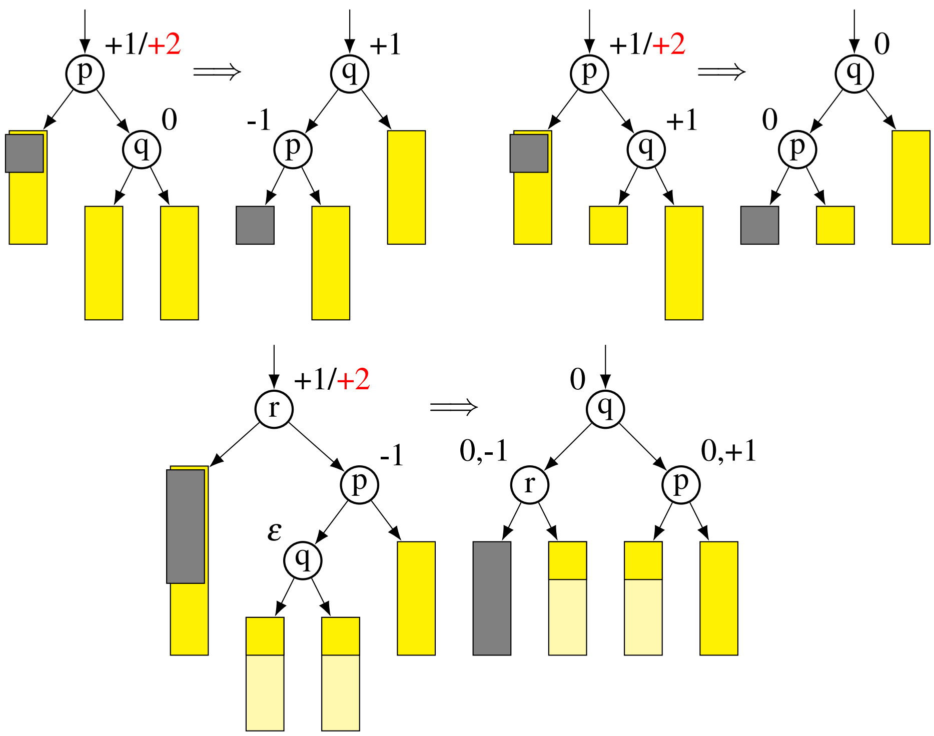
In the resulting tree (third diagram) the height at 11 has decreased by 1. As a result the balance at parent 8 changes to -2. We now have to rebalance at 8, the root. If we inspect the balance factors at 8 and 5 we see we are in a LL case: a single rotation at 8 to the right is applied.



◇

References. G.M. ADEL'SON-VEL'SKIĬ, E.M. LANDIS: An algorithm for organization of information (in Russian), Doklady Akademii Nauk SSSR **146** (1962) 263–266.

Slide 17 Deletion in AVL-trees. Top: two RR-cases, below: three RL-cases (for $\varepsilon = -1, 0, +1$). Yellow = height original subtree, gray = height subtree after deletion of node.



(Доклады Академии Наук, hier ook bekend als: Proceedings of the USSR Academy of Sciences, in vertaling uitgegeven door de American Mathematical Society, Soviet Mathematics 3 (1962) 1259–1263.)

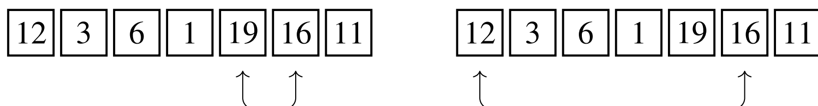
See DROZDEK 6.7 Balancing a tree; 6.7.2 AVL Trees.

See LEVITIN 6.3 Balanced Search Trees: AVL Trees.

See WEISS 4.4 AVL Trees. [Recursive implementation, removal seems easy, but it is not clear that insertion performs only one rotation?]

4.5 Self-Organizing Trees

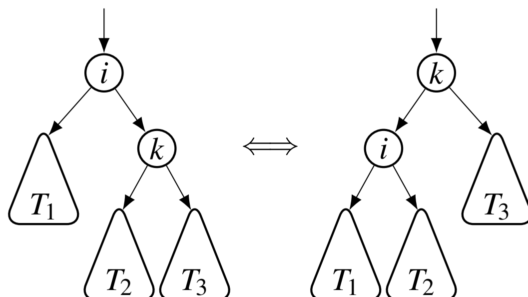
For linear lists we have heuristics for self-organization: whenever a key is found after search it is swapped with its predecessor (or even moved to the first position at once).



The same idea can be used as *self-organizing trees*[4]. Whenever a key is found after search we move it upwards either one level or completely to the root, using rotations. In this section we study the move to front heuristics, when accessing a key.

Lemma 4.8 (☒). *Let $i < j$, while j is a descendant of i in the tree. After accessing k key j is no longer a descendant of i iff $i < k \leq j$.*

Proof. First note that if k is not a descendant of i then accessing k does not change the subtree below i. If $k > j$ then it in the right subtree of i. While moving k to the root it meets i just before their rotation, see figure below. Now j is in either T_2 or in T_3. Note that j will cease to be a descendant of i iff it belongs to subtree T_3, i.e. iff it satisfies $k < j$, or if $k = j$.

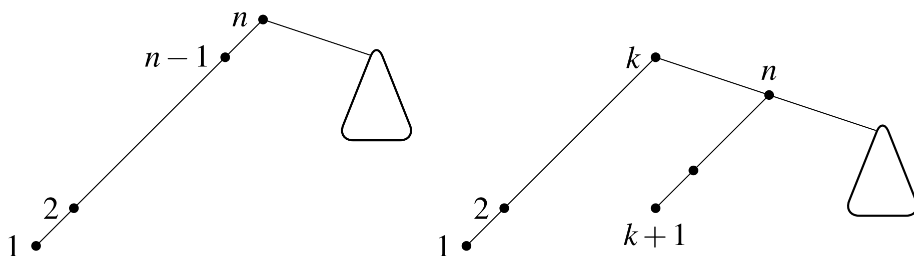


[4]This means the tree is not enforced with structural restrictions, but in general searches and insertions are implemented in such a way that the tree (hopefully) will gradually optimize its structure

□

This observation is useful if we want to argue how the subtree relations are after accessing a series of keys.

We can now point at a worst case scenario for this heuristics. When accessing the keys repeatedly in order, we obtain a linear time behaviour. Starting with *any* tree, accessing the keys $1,2,\ldots,n$ in that order will lead to a tree of the form in the first figure below. If we then again access one of these keys, the tree looks like the second figure when key k has been just accessed. Clearly moving this linear row of keys up one by one needs a quadratic number of rotations: $n+n-1+\cdots+1$. This is a linear number of rotations per key.



4.6 Splay Trees

Splay trees are a variant of the move to root technique. Again we move a key to the front, but this time not always bottom-up with single rotations, but in one case at the grandparent of the node that is being moved, *two levels* at a time. This leads to better balance. The "zig-zag" terminology originates from the original paper.

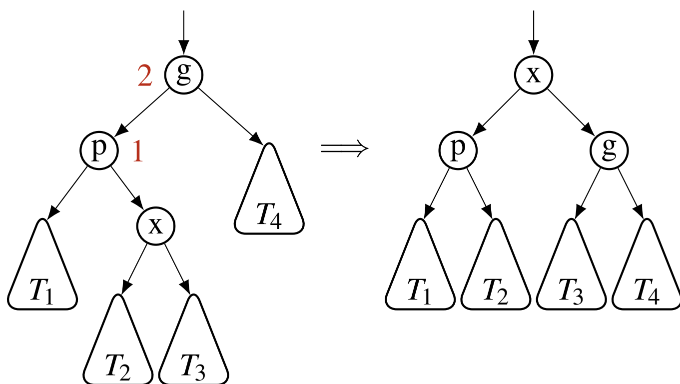
Features:

- Simple implementation, no bookkeeping. *self organizing*
- Any sequence of K operations (insert, find) has an *amortized* complexity of $\mathcal{O}(K \log n)$
- move item to root two levels at a time
- zig-zig step differs from bottom-up rotation

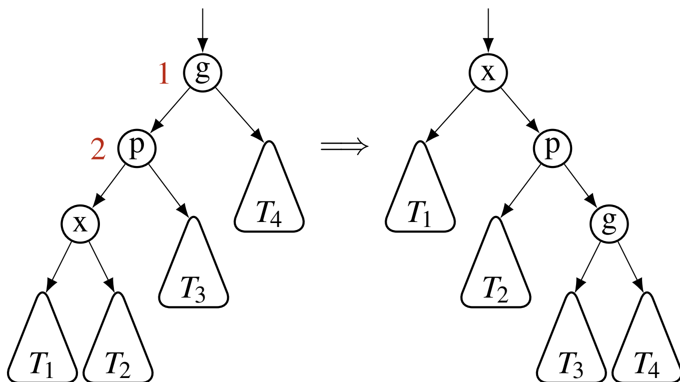
In *amortized* complexity we do not measure the complexity of a single operation, but we average over a seuqence of operations. This means that no single operation is guaranteed to perform in logaritmic time, but averaged over the complete sequence it does.

Below let x be the node that is moved to the root. We distinguish to which subtree x belongs, as seen from its grandparent g.

Zig-Zag step. Double rotation to the right at the grandparent, like in AVL-trees. This is actually the same as two single rotations at the parent and grandparent of x (first p to the left then g to the right).

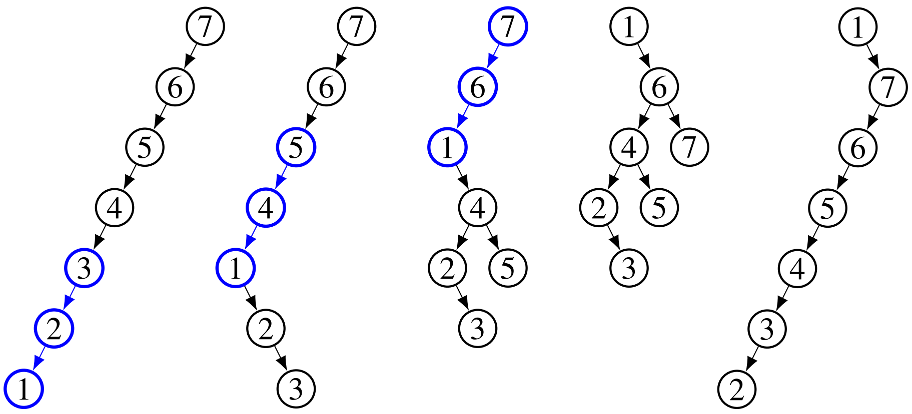


Zig-Zig step. Two rotations to the right: but here top-down! (So, first at g to the right, then at p to the right.) Here the splay tree differs in the move to root approach.



Zig step. When node x is a child of the root we perform a last single rotation. This is a standard rotation, and we do not repeat the picture here, see Section 4.1.

Example 4.9. As an example consider a seven node "linear" tree, with all descendants to the left. After searching for 1 is is splayed to the top in four steps. (Rightmost picture) When we would rotate 1 to the root with ordinary rotations, the tree would still be linear whereas with splaying the height practically halves (imagine a larger example).



◇

Implementation. There are many implementation choices. Most importantly, when do we splay? Possibilities are at search, unsuccessful search, insertion and deletion. In deletion splaying can be useful. To delete key x first splay x to the root, remove x to obtain two subtrees T_1 and T_2. Then splay the last key y in T_1 to the root (the predecessor of x). Clearly y has no successor in T_1, so no right child in the root of the new tree. At that position we attach the original right subtree T_2.

References. D.D. SLEATOR, R.E. TARJAN: Self-Adjusting Binary Search Trees, Journal of the ACM **32** (1985) 652—686 [ACM = Association for Computing Machinery] doi:10.1145/3828.3835

B. ALLEN, I. MUNRO: Self-organizing search trees, Journal of the ACM **25** (1978) 526-–535 doi:10.1145/322092.322094

See DROZDEK 6.8 Self-Adjusting Trees.

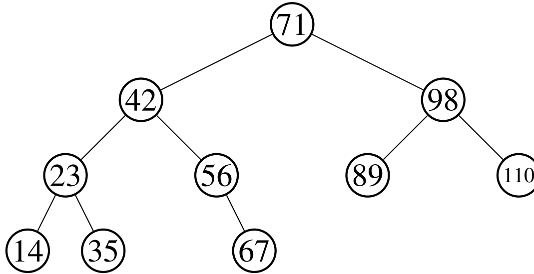
See WEISS 4.5 Splay Trees.

Opgaven

1. Deze opgave gaat over AVL-bomen.

a) Waarom zijn AVL-bomen te prefereren boven standaard binaire zoekbomen?

We gaan sleutels toevoegen en verwijderen. Beschouw de volgende boom B.



b) **i.** Wat is het resultaat van het toevoegen van sleutel 7 aan boom B?

ii. Wat is het resultaat van het toevoegen van achtereenvolgens 70 en 69 aan (de oorspronkelijke) boom B?

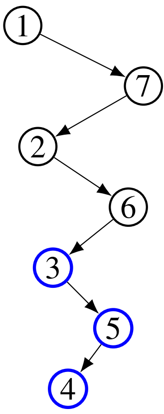
Jan 2016

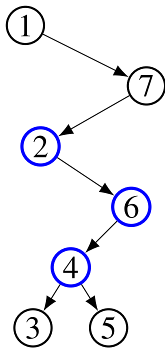
2. Bij de evaluatie van Fibonacci bomen wordt gezegd (pagina 64):

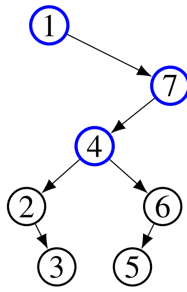
> in an AVL tree of height $2h - 1$ or height $2h$ the top h levels are completely filled

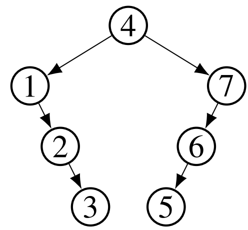
Bewijs dat dit klopt.

3. In Example 4.9 wordt getoond hoe de structuur van een lineaire boom na een Splay operatie verandert. Wat gebeurt er bij een ‘zig-zag’-boom, waar steeds één kind per knoop is maar nu afwisselend links en rechts.


1
7
2
6
3
5
4


1
7
2
6
4
3
5


1
7
4
2
6
3
5


4
1
7
2
6
3
5

5 Priority Queues

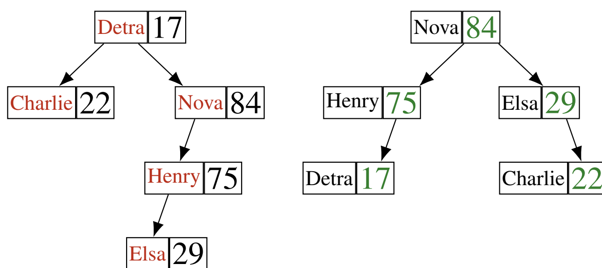
Ordering items in a set

A database consists of a set of key-value pairs. We can retrieve a record (element) by its key, and in this way we can update the value, or even delete the record from the set.

In a priority queue we also consider key-value pairs. The difference is that the value here has a specific meaning, it is the *priority* associated with the record. We add new record-priority pairs to the priority queue, and we can locate and delete the item with largest priority. There is no (efficient) way to find other items based on this priority. Also we cannot locate the elements based on their key, unless we build a separate structure to keep track of that key.

Example 5.1. Consider the following relation of persons with their ages:
{ ('Detra',17), ('Nova',84), ('Charlie',22), ('Henry',75), ('Elsa',29) }

We can store these key-value pairs according to their names, and we obtain a dictionary from which we are able to retrieve the age given the name of the persons. This uses a binary search tree, see the left diagram.



Alternatively we can store these items more suitable for a priority queue. This enables us to retrieve the persons in order based on their age, without actually completely sorting them. In this chapter we will see two different implementations of the priority queue. Both of these use a *heap-ordered* binary tree, see the right diagram. (The implementations also have an additional *structural restriction* to the form of the tree.)

◇

In the toy examples of this lecture usually only the priorities are given in the diagrams. The associated records are omitted. See for instance in Example 5.2 below, where two binary trees are given, one as a search tree, the other heap-ordered. These trees consists of numbers only, and have no associated data-values.

5.1 ADT Priority Queue

In a PRIORITY QUEUE the data elements are assigned with a priority. The priority values are totally ordered: for each pair we have exactly one of $p > q$, $p = q$ or $p < q$. We can add any element to the queue but we can only retrieve a maximal one. Elements can have the same priority, a fact that is often ignored for simplicity.

In practice it may be the case that we want to retrieve the element with lowest costs, hence priority goes up with lower values. We distinguish max-queues and min-queues when we want to make that explicit.

- INITIALIZE: construct an empty queue.
- ISEMPTY: check whether there are any elements in the queue.
- SIZE: returns the number of elements.
- INSERT: given a data element with its priority, it is added to the queue
- DELETEMAX: returns a data element with maximal priority, and deletes it.
- GETMAX: returns a data element with maximal priority.

Isempty and Size are mentioned here for convenience only: they are easily implemented by keeping a counter. Less common operations are:

- INCREASEKEY: given an element *with its position in the queue* it is assigned a higher priority.
- MELD, or Union: takes two priority queues and returns a new priority queue containing the data elements from both.

The definition of INCREASEKEY is tricky. We assume its position (address of, pointer to, index in, ...) is known when the operation is applied. The reason for this is that there is no operation to locate an element in a priority queue. It is however an important part of Dijkstra's algorithm for shortest paths to be able to update priorities when better connections in the graph are discovered. That means that we keep an inverse data structure that keeps track of the positions of all elements.

In examples involving priority queues often only the priorities of the elements is shown, not the associated data. The particular information does not help in understanding their mechanics.

Many types of priority queues have been developed, with the goal of minimizing the complexities of the operations. In this text we will discuss the *binary heap* and the *leftist heap*. Both are based on binary trees: one with array representation, the other with a linked trees. The binary heap is famous for one of its applications, *heapsort*.

We give a list of complexities for various implementations of the priority queue, from standard to quite involved. In fact the last type mentioned here (Brodal queue) is of theoretical interest only. Its implementation is very involved, and its complexity bounds have huge constants.

	Binary	Leftist	Pairing	Fibonacci	Brodal
GETMAX	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\mathcal{O}(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
DELETEMAX	$\Theta(\log n)$	$\Theta(\log n)$	$\mathcal{O}(\log n)^\dagger$	$\mathcal{O}(\log n)^\dagger$	$\mathcal{O}(\log n)$
INCREASEKEY	$\Theta(\log n)$	$\Theta(\log n)$	$\mathcal{O}(\log n)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)$
MELD	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

† amortized complexity

Fibonacci queues have their complexities as *amortized* bounds, the other are worst case.

Use cases. We mention some examples where priority queues are used.

- sorting (heapsort)
- graph algorithms (Dijkstra shortest path, Prim's algorithm)
- compression (Huffman)
- operating systems: task queue, print job queue
- discrete event simulation

Overview. In the next two sections we present two tree implementations of the priority queue data structure. Here a short overview of their distinctive features.

	binary heap	**leftist heap**
structure	binary tree	
restriction	complete	leftist
keys	heap ordered	
representation	array	pointers
internal	trickledown bubbleup	zip
advantage	heapsort	efficient meld

References. G.S. BRODAL: Worst-Case Efficient Priority Queues. In Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), (1996) 52–58. ACM:10.1145/313852.313883 "*. . . is based on heap ordered trees where [. . .] nodes may violate heap order.*" "*The data structure presented is quite complicated.*"

G.S. BRODAL, G. LAGOGIANNIS, R.E. TARJAN: Strict Fibonacci heaps. Proceedings of the 44th symposium on Theory of Computing (STOC12), (2012) 1177–1184. doi:10.1145/2213977.2214082

5.2 Binary Heaps

Features:

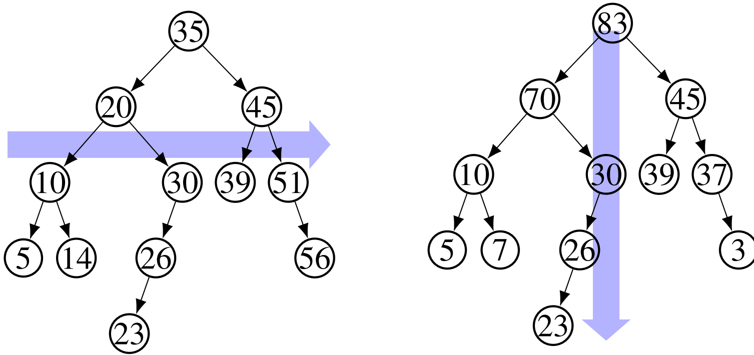
- Implementation of ADT PRIORITYQUEUE
- Uses a complete binary tree, stored as an array/vector.
- Keys are ordered in heap order.
- Internal functions TrickleDown and BubbleUp for restructuring
- Application: Heap sort.

In a *binary heap* the queue elements with their priorities are represented using a binary tree. The tree in use has two structural restrictions. First the tree is *complete*, second elements are stored in *heap order*, in decreasing priority when moving from root to leaf. There is no left-right order between priorities enforced. Since the tree is complete, it will be stored as array. Assuming the root at position 1, the children of node i are found at positions $2i$ and $2i+1$.

The following example is similar to Example 5.1, but with simple keys. (To stress, once again, the difference between heap-order and in-ordered search-tree.)

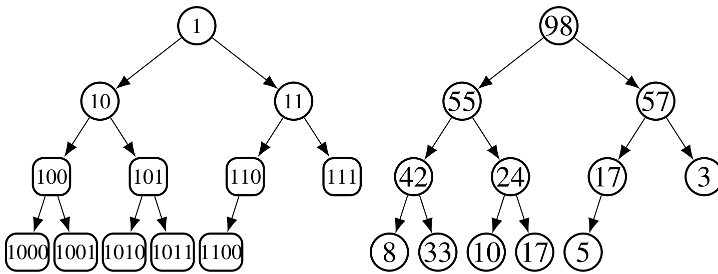
Example 5.2. As seen in Section 1.3, two binary trees. To the left the *binary search tree* (ordering from left-to-right) and to the right the *heap-order* (keys ordered from top-to-bottom). Note that in the search tree we can easily locate each item, going left or right each step, whereas in the heap-ordered tree it is not possible to decide in which branch a key is stored.

Note the heap-ordered tree is not complete. This extra requirement is needed to make the tree a binary heap.



◇

Definition 5.3. A *binary heap* is a complete binary tree with elements from a partially ordered set, such that the element at every node is larger than (or equal to) the element at its left child and the element at its right child.



98	55	57	42	24	17	3	8	33	10	17	5
1	2	3	4	5	6	7	8	9	10	11	12

As a consequence of the definition, by transitivity, *all* descendants of a node have a value less than the node itself.

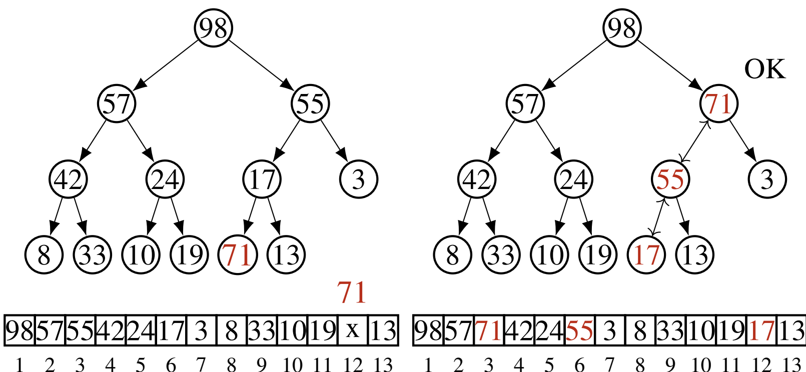
The priority queue operations are implemented using two basic internal functions:

- TRICKLEDOWN: Swap an element (given by its position in the heap) with the largest of its children, until it has a priority that is larger than that of both children.
- BUBBLEUP: Swap an element (given by its position in the heap) with its father until it has a priority that is less than that of its father (or is at the root).

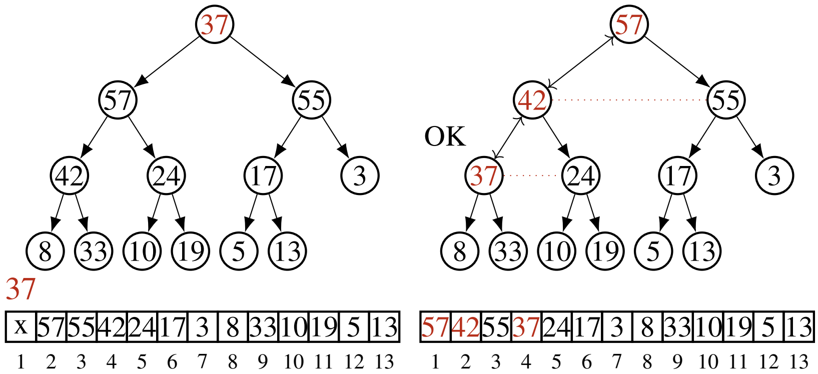
The restriction that an element is given by its position is essential: there is no efficient way to locate an arbitrary element in the heap.

These two names seem appropriate, but Wikipedia lists many more: "To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *swim-up*, *heapify-up*, or *cascade-up*), ..."

Example 5.4. (1) The leaf 71 is too large for its position (in a max-heap). It moves up in the tree, using BUBBLEUP.



(2) Root 37 is too small for its position. Its moves down, using TRICKLE-DOWN. In each step swap node with its largest child.

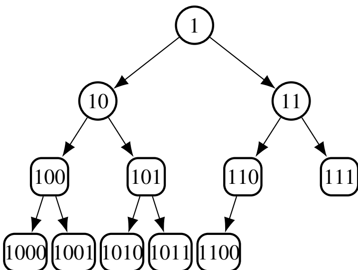


◇

The operations are now carried out as follows.

- INSERT: add the new element at the next available position at the end of the array/tree, then BUBBLEUP.
- GETMAX: the maximal element is present at the root of the tree.
- DELETEMAX: replace the root of the tree by the last element of the array/tree. That element can be moved to its proper position using TRICKLE-DOWN.
- INCREASEKEY: use BUBBLEUP. (We need the position of the key, as we cannot efficiently search for it.)

The restriction to consider only complete trees means that the most natural implementation uses an array (instead of pointers) to store the tree. If the root is at position 1 then the children of node x are at $2x$ and $2x+1$. Starting at 1 seems natural if one looks at the regularity of the addresses in binary.



Example 5.5. (1) In the heap below we insert key 29. Place it after the last leaf, and apply BUBBLEUP.

(2) Next we delete the maximal element (98) from the original heap. Replace the root by last element (13) and TRICKLEDOWN.

◇

Constructing an heap. If we randomly put elements in an array it can be seen as a complete binary tree, but if we want to use the data as an heap we have to reorder the keys. This is a useful operation: it can be used in a sorting method, heapsort, and it can be used to more efficiently merge two heaps.

- MAKEHEAP: Given an array, reorder its elements so that the array is a binary heap.

This operation is sometimes called HEAPIFY. It can be done in linear time, when the right approach is taken.

For the precise analysis, recall that $\sum_{i=0}^{h} i \cdot 2^i = (h-1) \cdot 2^{h+1} + 2$, see Lemma 3.8, and of course $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

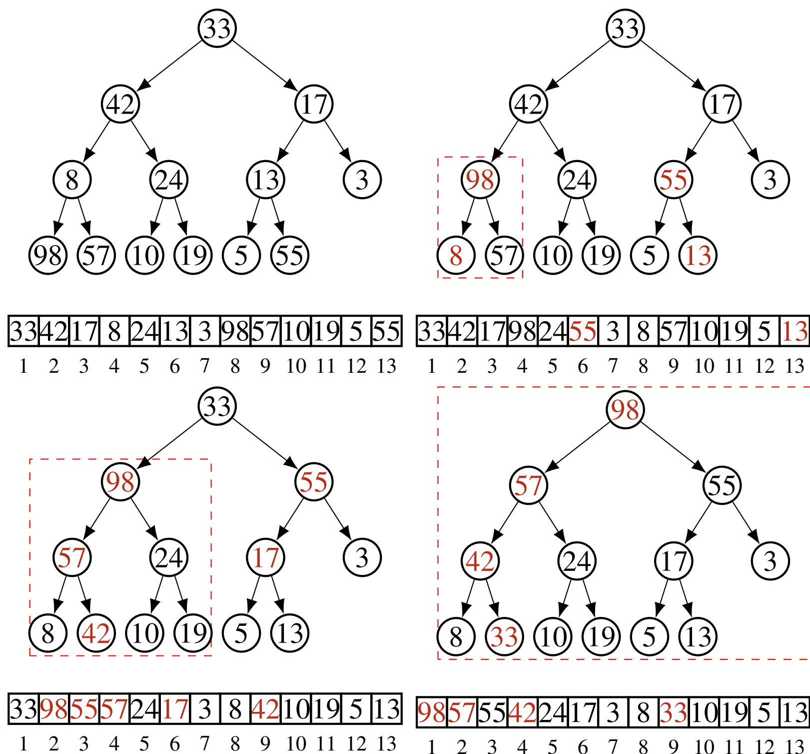
First approach, not optimal. We can construct the heap by adding one by one the elements of the array to the heap. Each element will bubble up in the tree, *top-down*. Assuming the heap is complete with the bottom level completely filled, it contains $n = 2^L - 1$ elements, and has levels $0, 1, \dots, L-1$, with level k containing 2^k elements. The elements at level k will move at most k levels up, so the total amount of key swaps is $\sum_{k=0}^{L-1} k2^k = (L-2) \cdot 2^L + 2$; see Lemma 3.8.

Thus expressed in n, the complexity is $\mathcal{O}(n \lg n)$.

Optimal. We can improve on this, by adding the elements *bottom-up* to the heaps, rather than top-down. So, we add each element to the heaps that were constructed below it. Thus the elements trickle down to their position. Now an element at level k will move down at most $L-k-1$ levels. Note half of the elements, those in the leaves, will not move down at all. The complexity now is $\sum_{k=0}^{L-1} 2^k(L-k-1)$. First split the summation into terms with 2^k and those with $k2^k$. We get $(L-1)\sum_{k=0}^{L-1} 2^k - \sum_{k=0}^{L-1} k2^k$. The first summation yields another power of 2, the second summation is again Lemma 3.8. The important parts cancel: $(L-1)(2^L-1) - (L-2) \cdot 2^L - 2 = 2^L - L - 1$.

Thus expressed in n, the complexity is $\mathcal{O}(n)$.

Example 5.6. We construct an heap-structure from an unordered array, working upwards at level, 2,1,0 and trickling down the nodes at those levels.



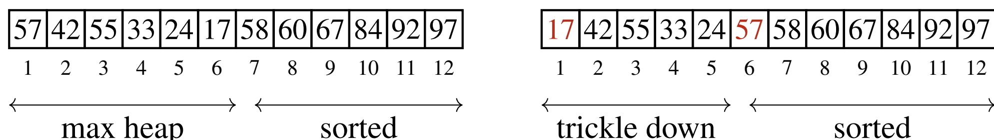
Merging. If we want to merge two heaps, we must move their data into a single array. The cost of moving is linear in the number of elements of the smallest heap. When we append the items of one heap after the other heap, the result is not automatically an heap. The operation Heapify can reorder the items in linear time to become a merged heap.

Heapsort. The binary heap is the key part of the *in situ* sorting method *heapsort*. This means that no extra space is used, apart from array where the data is stored. The array will have two functions: during the sorting process part of the array will be sorted, while the remaining values are stored as a binary max-heap. Each step the maximal value is removed from the heap (using DeleteMax of course) and stored at its position in the array.

To start the sorting, the initial array must be changed into an heap, that is, we apply Heapify, in linear time.

Sorting step. We illustrate a single step in the heapsort of an array. In the example diagram below (left) the elements $[7 \cdots 12]$ are already at their sorted position in the array, while the initial elements $[1 \cdots 6]$ form a max-heap.

The maximal element of the heap is at position 1, extract it from the heap while moving it to its sorted position by swapping positions 1 and 6, and then trickle down the 1st element to its proper position in the heap.



The complexity of heapsort is $O(n \lg n)$, where n is the number of elements that are sorted. This can be seen as follows. We repeatedly apply DeleteMax (using TrickleDown) on a shrinking heap, each step logaritmic in the size of the heap. That gives complexity $O(\sum_{k=1}^{n} \lg k) = O(\lg n!)$. Then according to the approximation by Stirling, $\lg(n!) = n \lg n - n + \Theta(\lg n)$.

References.
J.W.J. WILLIAMS: Algorithm 232: Heapsort. Communications of the ACM **7** (1964) 347–348. doi:10.1145/512274.512284

5.3 Leftist heaps

Features:

- Implementation of ADT PRIORITYQUEUE
- Proper binary tree, with structural restriction on path lenghts.
- Keys are ordered in heap order.
- Internal function Zip for restructuring.
- Queues can be efficiently melded (joined).

A leftist heap is a binary tree implementation of the priority queue where the nodes are *heap ordered* like in a binary heap (children have a lower priority than their parent) but does not maintain its complete form. Instead it tries to be out of balance, paths to the left tend to be longer than those to the right; hence the name.

To define the data structure, we first introduce some concepts. Given a binary tree its extended binary tree is obtained by adding a special leaf at positions where a node had no child. The resulting tree is *full*: every node either has two children, or is a leaf.

For each node x we define its *nil path length*, denoted $\text{npl}(x)$, as the shortest distance to an external leaf. Recursively that can be written as $\text{npl}(x) = 0$, when x is an external leaf, and

$$\text{npl}(x) = 1 + \min\{\ \text{npl}(\text{left}(x)),\ \text{npl}(\text{right}(x))\ \},$$

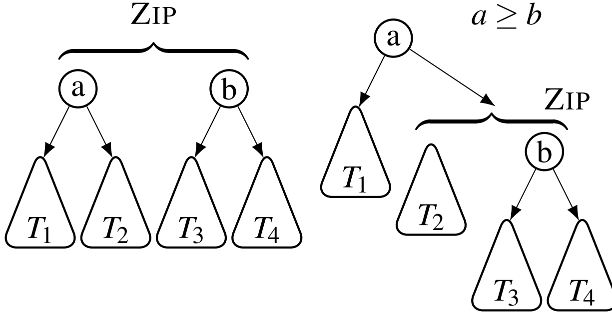
when x is an internal node.

Definition 5.7. A *leftist tree* is an (extended) binary tree where for each internal node x, $\text{npl}(\text{left}(x)) \geq \text{npl}(\text{right}(x))$. A *leftist heap* is a leftist tree where the priorities satisfy the heap order.

Example 5.8. A binary tree with its extended leaves. Numbers indicate npl. All leaves have npl 0, the other npl's are indicated inside the nodes. At four nodes the leftist-tree property is violated. We swap the two subtrees at these positions.

◇

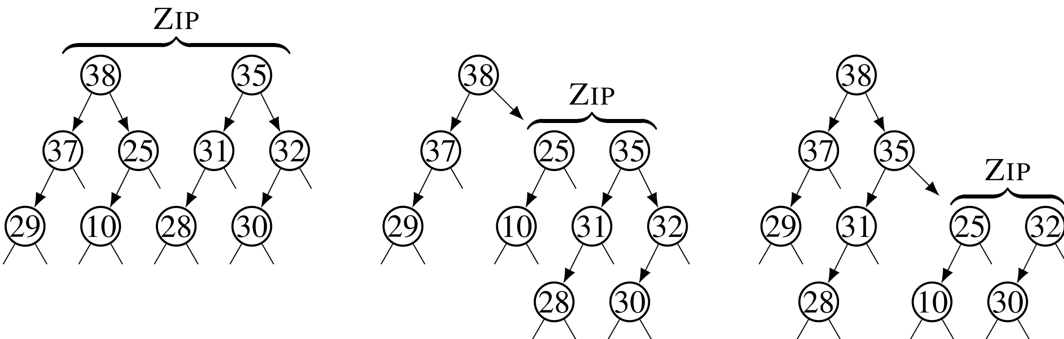
We *recursively* define an internal function ZIP to this data structure. It takes two leftist trees T_a, T_b with roots a and b, where we assume the priority(a) is at least priority(b), swapping the trees otherwise. The result is the tree with root a, where the right subtree T_2 of T_a is replaced the (recursively obtained) ZIP of T_2 and T_b. Additionally the children T_1 and $\text{ZIP}(T_2, T_b)$ are swapped when necessary to maintain the leftist tree property. Perhaps a diagram is helpful.

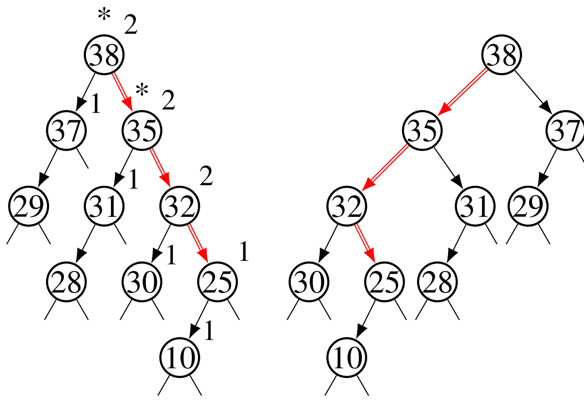


The trees T_i to the left are copied from the old to the new structure, so the npl in their nodes did not change. For the nodes on the rightmost path the npl do change. The new values can be computed bottom-up.

The tree constructed this way has the heap-order, but might not be leftist. The proper structure can be obtained by swapping children at the same rightmost path when necessary. This can be done at the same time as the new npl are computed.

Example 5.9. We ZIP two leftist trees, data represented in the nodes, by first joining nodes along the rightmost paths, each time choosing the largest priority. Then we restore the leftist property, bottom-up, by swapping children along the path where the two npl are in the wrong order.





◇

One easily shows by induction that the number of nodes in a leftist tree is (at least) exponential in the length of the rightmost path. As the ZIP-operation is performed along that rightmost path of two trees, the number of nodes visited is logarithmic in the total number of nodes in the trees.

Lemma 5.10. *Let T be a leftist tree with root v such that* $\mathrm{npl}(v) = k$*, then*
(1) *T contains at least $2^k - 1$ (internal) nodes, and*
(2) *the rightmost path in T has exactly k (internal) nodes.*

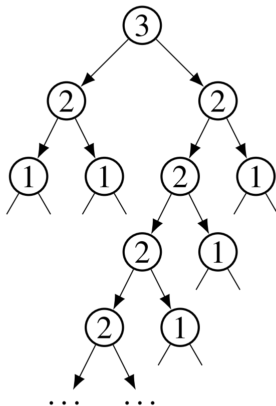
Proof. (1) Using induction on k, the $\mathrm{npl}(x)$ of the root. If $k = 1$, then the tree contains one node, the root itself.

Assume that the $\mathrm{npl}(x) = k+1$ for the root x. Then both children must have npl at least k. Using the inductive hypothesis we know that both subtrees must contain at least $2^k - 1$ nodes. The total number of nodes is at least $2 \cdot (2^k - 1) + 1 = 2^{k+1} - 1$.

(2) The npl of the right child is precisely one less than that of its parent. We can prove this as follows. In general the formula $\mathrm{npl}(x) = 1 + \min\{\,\mathrm{npl}(\mathrm{left}(x)), \mathrm{npl}(\mathrm{right}$ holds. The leftist property requires that $\mathrm{npl}(\mathrm{left}(x)) \geq \mathrm{npl}(\mathrm{right}(x))$. Thus the minimum is always the right child, and $\mathrm{npl}(x) = 1 + \mathrm{npl}(\mathrm{right}(x))$. □

The time complexity of zipping two trees is proportional to the sum $k = k_1 + k_2$ of the lengths of both rightmost paths in the trees. By the above lemma these lengths are equal to the npl at the root of the trees. Again by the lemma, the total number of nodes in the trees is at least $n = 2^{k_1} + 2^{k_2}$ (minus two). Thus $k = O(\lg(n))$.

Example 5.11. In leftist trees the length paths to the left *cannot* be bounded by a function of the npl at the root. In the example below the root has npl 3, while the long path of nodes with npl = 2 can be made into any length. Although in general paths to the left tend to be longer than those to the right, the longest path is not necessarily the one to the left starting at the root, see again the example. Numbers indicate npl.

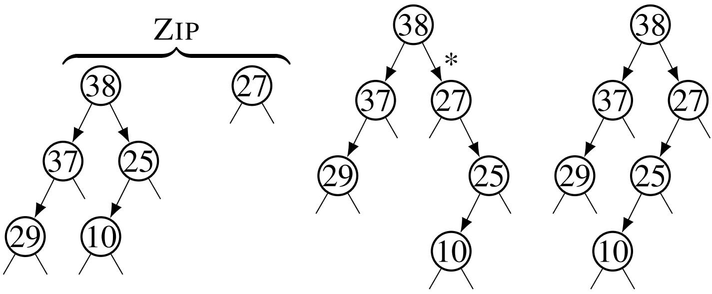


◇

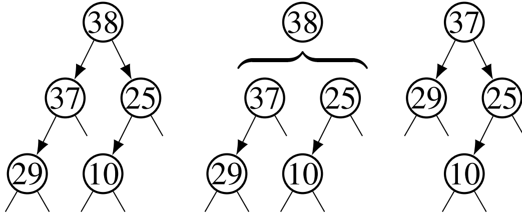
The priority queue operations are now carried out using ZIP, as follows. Since ZIP is of logarithmic complexity in the number of keys (nodes) also the priority operations are log-time efficient. For the special operation INCREASEKEY we have to be careful: as we have seen in the above example, the (leftmost) paths cannot be bounded as a logarithm of the number of keys.

- INSERT: construct a single node tree and ZIP with the original tree.
- GETMAX: the maximal element is present at the root of the tree.
- DELETEMAX: delete the node at the root, ZIP the two subtrees of the root into a new tree.
- MELD: is performed by a ZIP.
- INCREASEKEY: Cut the node and its subtree from the tree. Repair the remaining tree as npl on the path to the root has been changed. Zip the two trees. (See Example 5.13 for explanations about the complexity.)

Example 5.12. We insert 27 into a leftist heap with five elements, by zipping the heap with a single element heap. After the zip we swap children at node 27 to restore the leftist property.



Then we delete the maximal element from the same (original) five element leftist heap, by removing the root and zipping its two subtrees.

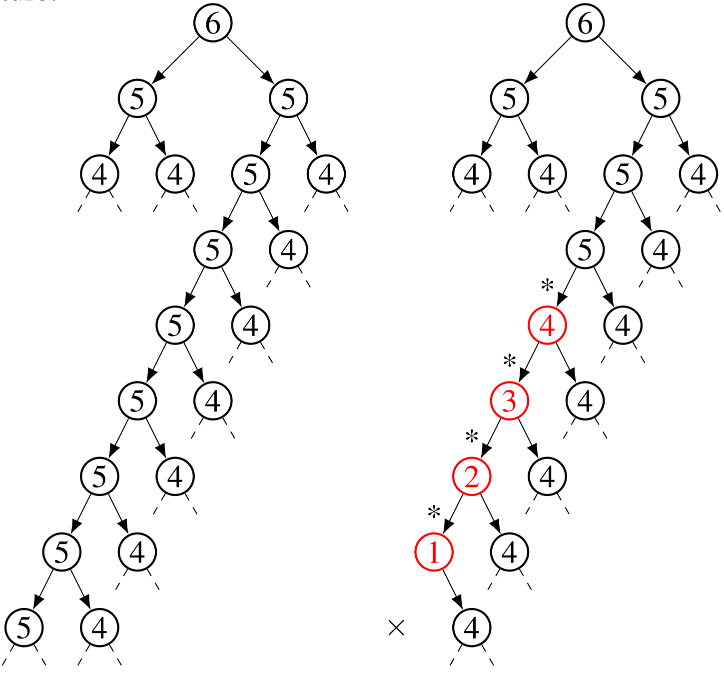


Example 5.13. The left figure shows a leftist tree with a rather large rightmost subtree, like in Example 5.11. The numbers indicate the npl at each node, not the priority of the node. If we want to apply INCREASEKEY to one of the lower nodes with npl equal to 5, we *cannot* increase the priority and swap with the parent in this data structure (like we did for the binary heap) as this does not guarantee a logarithmic bound. The reason we cannot guarantee this is because we cannot bound the length of the linear path in the subtree. Instead cut the node with its subtree at the position where we want to increase. That subtree is an leftist heap by itself, and the npl at its root is at most the logarithm of the number of its successors.

The original tree now most probably no longer is leftist, and we have to recalculate npl(x) at every node upwards (swapping left and right where necessary). Again we have to verify that this recalculation can be done in a reasonable number of steps. At each step we increase this value by one until at some point we reach the original value. So this has again a logarithmic bound. Now ZIP the two trees to obtain the new leftist tree.

In the diagram below, a leftist heap before and after cutting a subtree (for simplicity subtrees indicated by dashed lines were omitted). After recomputing

the npl the starred nodes need swapping their children to reobtain their leftist structure.



References. C.A. CRANE: Linear lists and priority queues as balanced binary trees. Technical Report STAN-Cs-72-259, Computer Science Dept, Stanford Univ., Stanford, CA, 1972

See WEISS 6.6 Leftist Heaps

Pairing Heap ☒

Wishful thinking. Tot nu toe nooit behandeld.

References. M.L. FREDMAN, R. SEDGEWICK, D.D. SLEATOR, R.E. TARJAN: The pairing heap: a new form of self-adjusting heap. Algorithmica **1** (1986) 111–129. doi:10.100

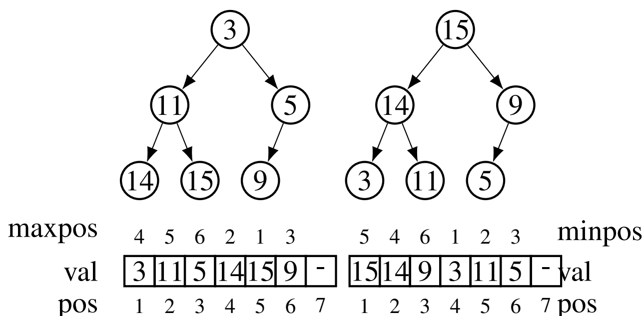
5.4 Double-ended Priority Queues

A *double-ended priority queue* is a data structure extending the priority queue, in that it can handle request for both minimal and maximal elements of the set (instead of only one of these).

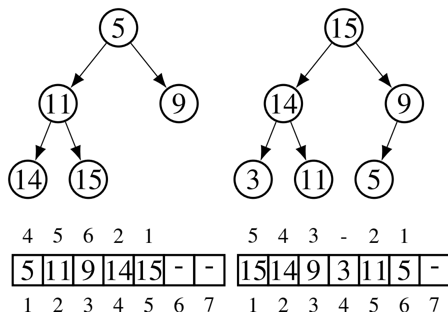
Many implementations involve two binary heaps (one for min and one for max) that are cleverly balanced.

Dual Structure Method. Store keys twice, once in a min-heap and once in a max-heap. If the minimum is deleted from the min-heap, it has to be deleted from the max-heap too. Hence for each key in one heap we have to know its position in the other in order to be able to find and remove it. Similarly, when during insertion of a key some other keys change position in one heap, we have to update this information in the other heap. Not very hard to do, but a lot of book-keeping.

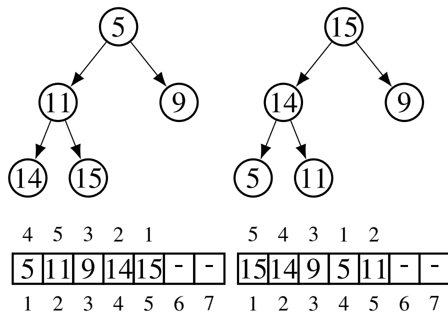
Example 5.14. Consider a double ended priority queue with values $\{3, 5, 9, 11, 14, 15$ stored both in a min-heap and a max-heap. In each heap we also store the position of the keys in the *other* heap, to locate that key when its info has to be updated.



Delete min 3 from min-heap, move last key 9 to root, then trickle down, and swap with 5. When moving values the positions of those values in the max-heap move with them. Also update the min-heap positions of the elements stored in the max-heap.



As 3 has been deleted from the set of values its copy in the max-heap has to be removed too. In the original min-heap the position of 3 in the max-heap was given as 4, so we can find it. Move the last element 5 of the max-heap to that position (it fits there as it is smaller than the parent, so no bubble-up needed). Update the max-heap position of 5 in the min-heap.



◇

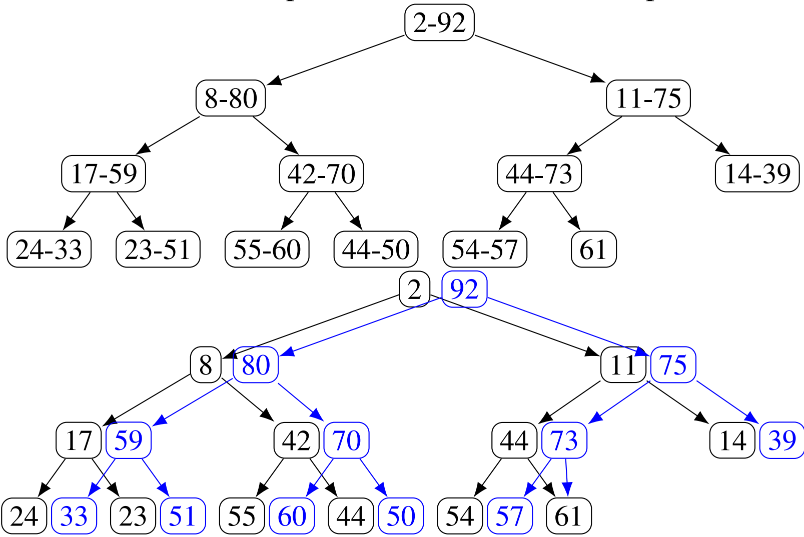
We can also divide the keys over a min-heap and a max-heap. In that case we have to balance the two heaps, and always make sure that the minimal element is indeed in the min-heap (and similar for max). We present a structure that ensures this.

Interval Heaps. An *interval heap* is like a binary heap except that its nodes contain two numbers instead of one. The numbers are ordered like intervals. This when x', y' is a child of x, y, then $x \leq x' \leq y' \leq y$, or $[x', y'] \subseteq [x, y]$. When the set stored in the heap contains an odd number of elements, the last node contains only one element, and is considered a one element interval $[x.x]$.

It is easy to see that the left elements of the nodes form a min-heap. while the right elements form a max-heap. These two embedded heaps form the heart of the procedures for adding or deleting keys. A key is added at the proper node at the end of the tree. When the key is smaller than the min-key of its parent, we insert the key in the min heap. Similarly, when the key is larger than the max-key of its parent, we insert the key in the max heap. This is done as in binary heaps: we swap an element with its parent as long as it is smaller than (larger than, for the max heap) that parent.

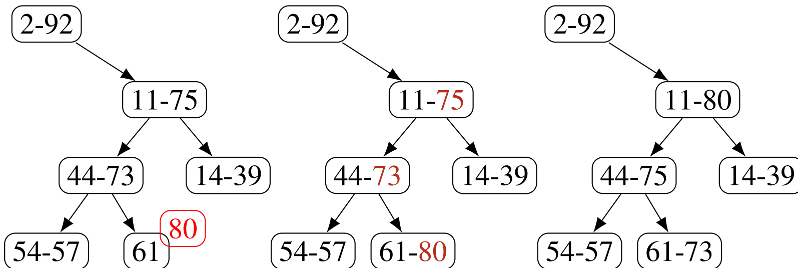
Deletion of the min element likewise is similar to deletion in a binary heap. Remove the min element, and replace it with the element from the last leaf. As long as larger than one of its children swap with the smallest child. There is one complication. There is no guarantee that the left and right elements form an interval at each step. When the left element (min heap) is larger than the right element (max element) we interchange the two elements, and continue (with the min heap).

Example 5.15. An interval heap and its embedded min-heap and max-heap.



Inserting an element to the interval heap is done, by starting at the last position, and inserting like in a ordinary binary heap. We have to swap keys when this is necessary to keep the left value smaller than the right value.

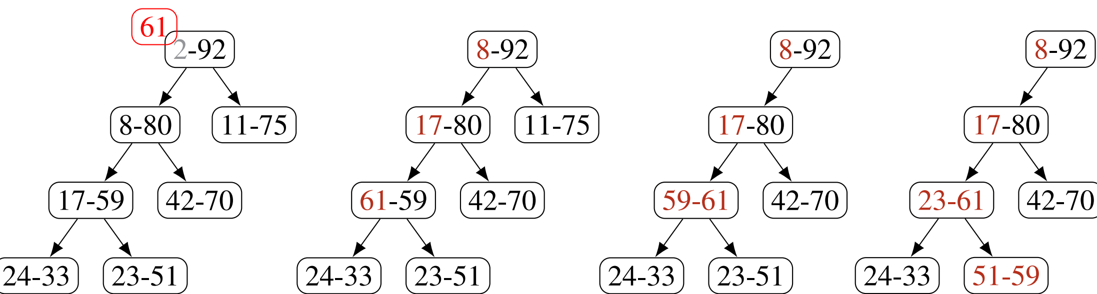
First we *insert* 80 in the heap above. It is added to the rightmost leaf as that has only a single value. Since 80 $>$ 61 the new interval is [61-80] and we add 80 to the max heap.



Now *delete the minimal value* (from the original set). Thus we delete the root of the min-heap, and like in the binary heap, we replace it by 61, the last item in the tree.

Then we sift-down in the min-heap, swapping the key with that in its smallest child. When we reach [61-59] the two values have to be swapped and we get the new interval [59-61]. Note the max-heap is still in order. For the parent 61 $<$ 80, the value came down and fits inside the parent interval. For the child: we have increased the right boundary of the interval from 59 to 61, so the heap property still holds.

The new value in the min-heap is 59, which is further sifted down. At the leaf we once more swap both elements.



◇

References. J. van Leeuwen, D. Wood: Interval Heaps. The Computer Journal **36** (1993) 209–216. doi:10.1093/comjnl/36.3.209

Opgaven

1.a) Voeg de getallen 1 tot en met 7 toe aan een lege min-Priority Queue. Hoe ziet de boom eruit in geval van een Binary Heap en in geval van een Leftist Heap?

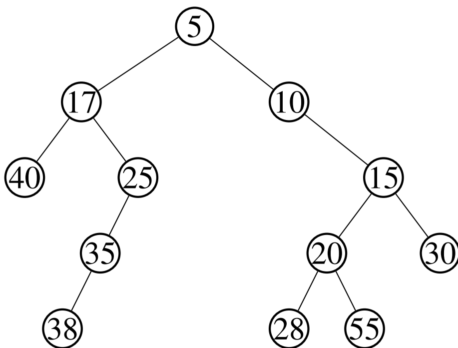
b) Idem, maar nu in het geval van een max-Priority Queue.

2. De operatie Heapify/Makeheap zet een willekeurig array om in een binary heap.

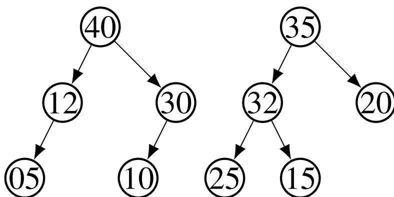
a) Leg uit waarom dit altijd lineair veel werk kost.

b) Geef aan hoe een array eruit ziet waarvoor de top-down methode (*first approach*) inderdaad $O(n \lg n)$ veel vergelijkingen gebruikt. De sleutels schuiven dan maximale lengte door.

3. Maak de volgende boom leftist door op de juiste plekken de kinderen van een knoop om te wisselen.



4.a) Rits (Zip) de onderstaande Leftist Heaps, zodat er weer een Leftist Heap ontstaat.

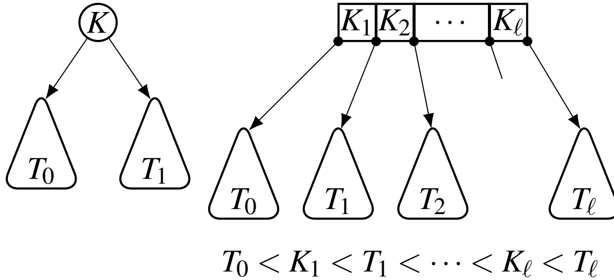


b) Verwijder het maximum uit de resulterende Leftist Heap.

5. Onderzoek hoe een AVL-boom gebruikt kan worden als Priority Queue. Kun je iets zeggen over de complexiteit van de Priority Queue operaties?

6 B-Trees

In binary search trees a node contains a single key, and the left and right subtrees have keys that are less and larger than the key in the node (respectively). This can be generalized to *multi-way search trees* with nodes that have a sequence of several keys, say ℓ keys $K_1, K_2, \ldots, K_\ell$, and $\ell + 1$ subtrees $T_0, T_1, \ldots T_\ell$ at that node such that K_i is larger than the keys in T_{i-1} while it is less than the keys in T_i, for $0 < i < \ell$.



6.1 B-Trees

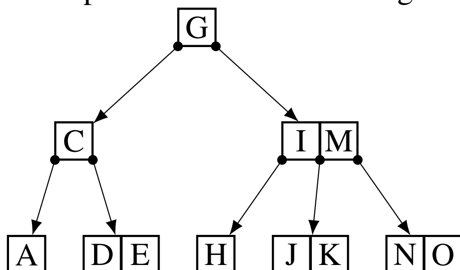
Nodes with a large number of keys are preferred when the search tree is stored on disk. Obtaining new data from disk is several orders of magnitude slower than from internal memory. This means that the number of disk reads should be minimized: getting a node with a single key at each step is slowing down the process. Thus trees are considered with a larger number of keys at each node.

Definition 6.1. A *B-tree of order m* is a multi-way search tree such that

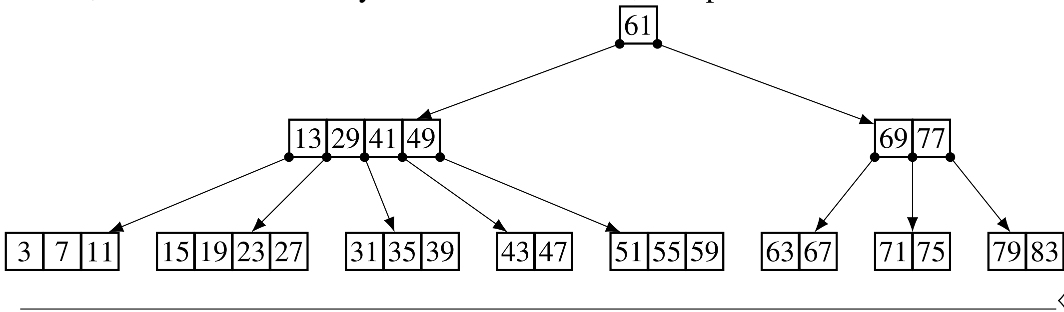
- every node has at most m children (contains at most $m - 1$ keys),
- every node (other than the root) has at least $\lceil \frac{m}{2} \rceil$ children (contains at least $\lceil \frac{m}{2} \rceil - 1$ keys),
- the root contains at least one key, and
- all leaves are on the same level of the tree.

This means that there is a strict depth restriction on the B-tree, every path from root to leaf has the same length. On the other hand, the number of keys per node may vary by a factor of two, for flexibility.

Example 6.2. In a B-tree of order 3, the number of children in a node may vary between 2 and 3, and the number of keys either is 1 or 2. There is no restriction for the root. Here is an example of such a tree with eight nodes on three levels.



In a B-tree of order 5, the number of children in a node may vary between 3 and 5, so the number of keys is between 2 and 4, except for the root.



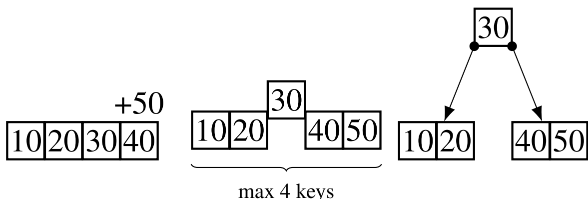
◇

Adding Keys. The algorithm to add a key to a B-tree can be summarized as follows. The example below will illustrate the various cases.

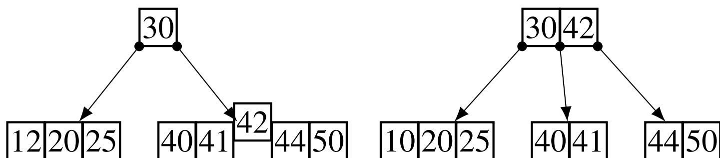
- Always add the new key to a leaf which is found by following the path from the root determined by the keys at the internal nodes. If the key can be fitted into the leaf without exceeding the maximal capacity we are done.
- When that leaf is already at its maximal capacity, the key is added but the leaf is split into two leaves; the middle key is moved to the parent to serve as a separator between the leaves. If the parent already is at maximal capacity we repeat and split there.
- Eventually splits can reach the root. We then obtain a new root with a single key and two children (which are formed when splitting the old root).

Example 6.3. We store a sequence of numbers (10, 20, 40, 30, 50, 25, 42, 44, 41, 32, 38, 56, 34, 58, 60, 52, 54 and 46, in that order) in an initially empty B-tree of order 5. Each node (except the root) must hold at least 2 keys, and each node may hold at most 4 keys.

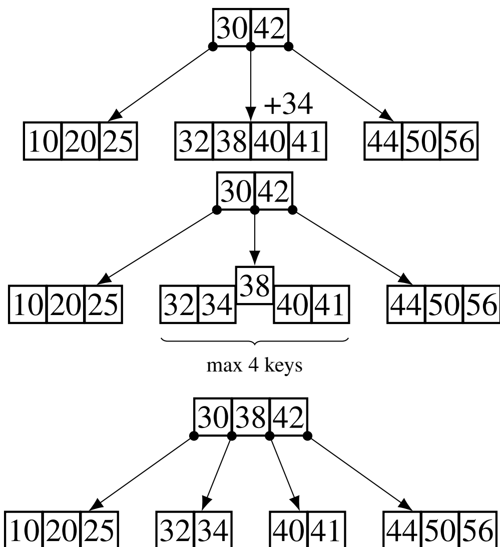
After adding the first four keys 10, 20, 40, 30 the tree consists of a single root, which is now full. Adding the fifth key 50 means we have to split the five keys into two new nodes, and promote the middle key 30 one level up, to become the new root.



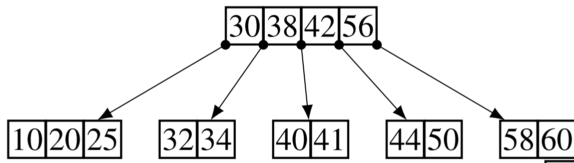
The next three keys 25, 42, 44 can be added in the leaves. The next key 41 arrives in a leaf which already contains four keys, the leaf splits, and its middle key 42 moves one level up to the root. The root contains two keys, and we have three leaves below the root.



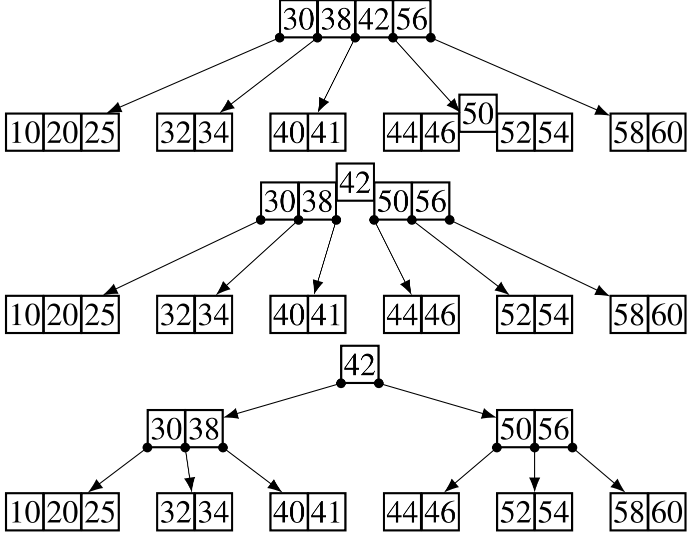
We now add four keys 32,38,56 and 34 where the last key causes a split of the middle leaf.



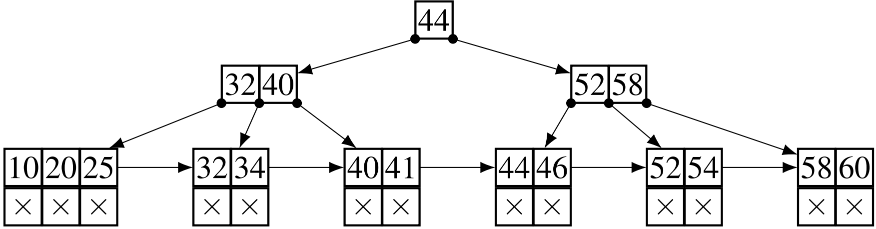
Now add two keys 58,60 which both end up in the rightmost leaf. With five keys 44 50 56 58 60 it has to split. The middle key 56 moves to the parent (the root).



The two keys 52,54 are added, ending up in the same leaf 44 50 52 54. Adding yet another key 46 destined to that leaf causes it to split. Middle key 50 moves up to the parent which already is at maximal capacity and splits too, yielding a new root 42.



Variations. ⊠ The B-tree has keys *and* (links to) data at every level of the tree. A B+-tree has only keys in its internal nodes, no data, to find the proper leaves. At the leaves all keys are stored together with the data (directly or via a pointer). This means that some keys are present in both the internal nodes and the leaves. Additionally the leaves in the B+-tree are usually linked as a linear list so that the data can easily be accessed in sequential order.



There is also the B*-tree in which the nodes are 2/3 full instead of 1/2, by stronger balancing. It turns out that deletion is much more complicated that way.

6.2 Deleting Keys

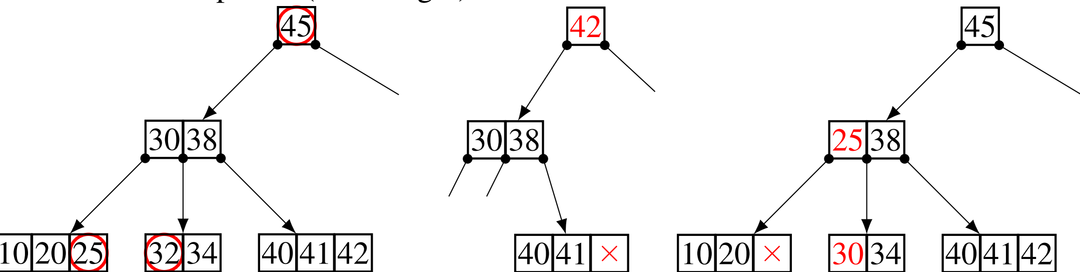
In short:

- For non-leaves: swap key with predecessor (key moves to a leaf)
- If below minimal capacity, get key from sibling with surplus, *via* parent.
- If no siblings with surplus: merge with sibling and get separating key from parent. Recurse with parent.
- Due to recursion, deletion may reach the root, and can collapse a level.

When deleting a key from a B-tree we restrict ourselves to leaves. For keys in an other position we use a trick also used for binary trees: locate the predecessor of the key and copy that to the key to be deleted. Now we may delete the original copy of the predecessor, which is located in a leaf.

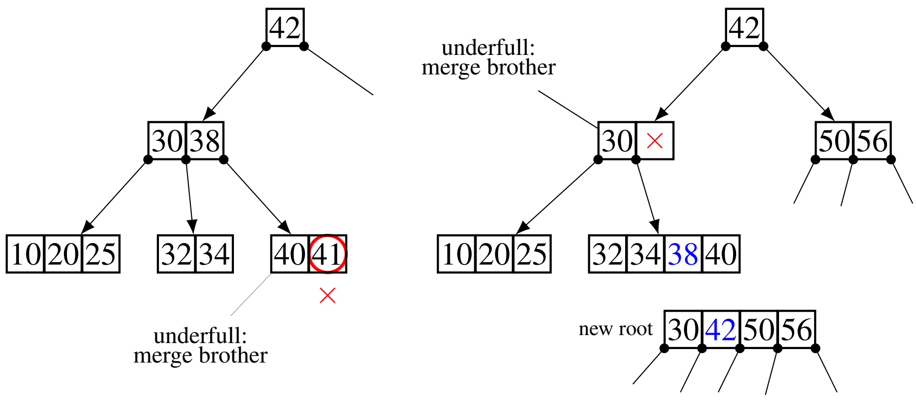
When deleting a key from a leaf we consider several cases. If, after removing the key the leaf still contains enough keys, we can stop. Otherwise, if the contents of the leaf drop below the minimal threshold, we first investigate whether one of the immediate siblings has and surplus key that can be moved to the leaf. If this is possible we can solve the deletion by moving the key. We must take care however: the key is first moved to the parent in the position between the sibling and target leaf and then the key originally at the parent moves to the leaf.

Example 6.4. We start with the tree fragment below (left) and we will discuss deletion of each of the three keys indicated by a red circle.
(1) When deleting 45 we delete its predecessor 42 instead, after copying it to the position of 45 (left & middle). The predecessor is at a leaf, where the actual deletion starts.
(2) Deletion of 25 is done by just removing the key in the leaf. There are enough remaining keys in the node.
(3) Deletion of 32 (in the original tree to the left) will leave a leaf with just a single key. Its sibling to the left has three keys, one of which can be moved via their common parent (left & right).



Finally, if the leaf has not enough keys left and is unable to obtain a key from a sibling we merge the leaf with that sibling. Note we also have to move the key in the parent between the leaves: as the parent looses a child it also has to reduce by one key. Now the parent has one key less, which means it can drop below minimum. In that case repeat at that level: try to move from a sibling, otherwise merge. Eventually we may loose one level when the two only children of the root are merged.

Example 6.5. When deleting 41 we cannot move a key from a sibling. Hence we merge the two leaves. Unfortunately the parent now is left with just a single key. We iterate one level up where we again have two merge two nodes into a new root.



References. R. BAYER, E. MCCREIGHT: Organization and Maintenance of Large Ordered Indexes, Acta Informatica 1 (1972) 173–189. doi:10.1007/bf00288683

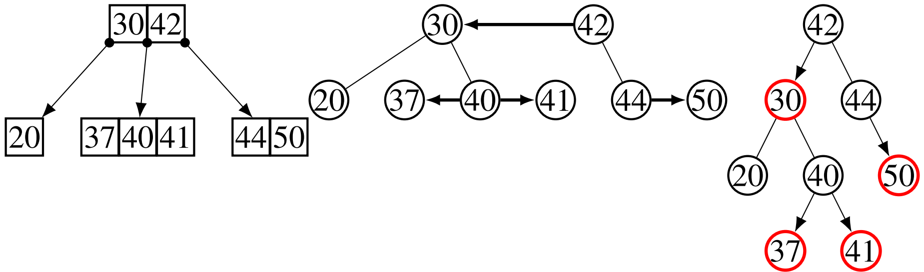
See DROZDEK Section 7.1: The Family of B-Trees

See WEISS Section 4.7: B Trees

6.3 Red-Black Trees

We focus on B-trees with $m = 4$. These have 1 to 3 keys in each of their nodes and 2 to 4 children, and are called *2–4-trees*. In this section we study a representation of those trees as binary search trees. Additionally they will have a balanced behaviour, like the B-trees themselves.

Example 6.6. We show a 2–4-tree and its representation as binary search tree. Nodes that are originating from the same B-node are grouped, by 'horizontal' edges. Such trees are called *VH-trees*. As a third figure the red-black representation of the same configuration. Red nodes are "dependent" of their black parents, and belong to the same B-node.

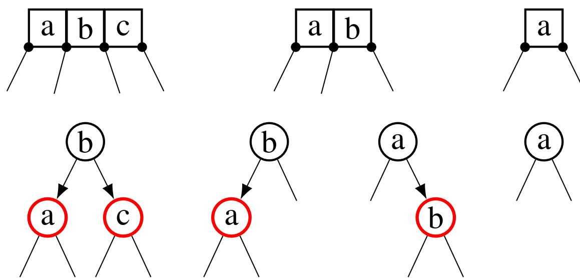


VH-trees and red-black trees are very similar. In VH-trees there are two types of edges, in red-black trees two types of nodes.

◇

Implementation. The property of being a red node can be coded in the node itself, but also in its parent, where we can store the colour for each of the children. This is somewhat logical as the node may want to know whether the children belong to the same B-node.

For nodes with 4,3,2 keys we have a simple correspondence to red-black trees as in the diagram below. For a B-node with two keys we have a choice with a red child to the left or to the right.



Definition 6.7. A *red-black tree* is a

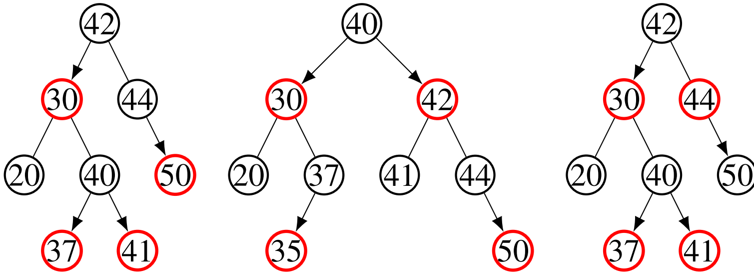
- binary search tree

such that each node is either black or red, where

- the root is black,
- no red node is the child of another red node,

- the number of black nodes on each path from root to extended leaf (NIL-pointers) is the same.

Example 6.8. The first two trees shown are red-black trees: all paths from root to leaf contain exactly two black nodes. The third tree is not red-black: the path that ends at the left (nil-)child of node 44 has only one black node, unlike the other paths.



It can be shown that every AVL-tree, and in particular the Fibonacci trees, can be red-black coloured. See Opgaven.

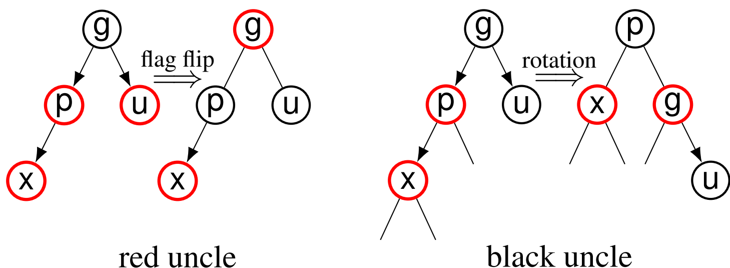
Inserting a key to a red-black tree. Like in any binary search tree we add the new key as a leaf in the proper position of the tree. That leaf must be red, as otherwise we will destroy the property that all paths to leaves have the same number of black nodes. On the other hand, the parent of the new red node may also be red. We show how to remedy that.

When a node x and its parent p are both red we first look at the *uncle* u which is the brother of the parent. The uncle may very well be an extended leaf, in wich case we consider it to be black.

- If the uncle is red, perform a *flag-flip*. Parent and uncle become black, grandparent becomes red. This will not effect the number of black nodes on paths from root to extended leaf, so locally solves the problem. However we have to check at the grandparent what is the colour of its parent, and repeat the procedure.

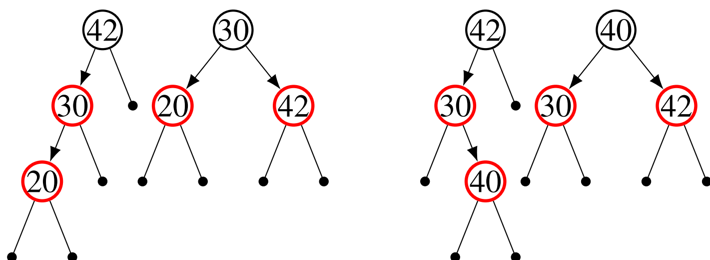
- If the uncle is black, perform a *rotation*, locally restructuring the tree. As with AVL trees we distinguish LL, LR etc cases leading to single or double rotations, either to the left or to the right. We can stop. Observe the uncle u is not involved in the operation, except as part of the subtrees involved in the rotation.

  Below a flag-flip and a single rotation to the right.

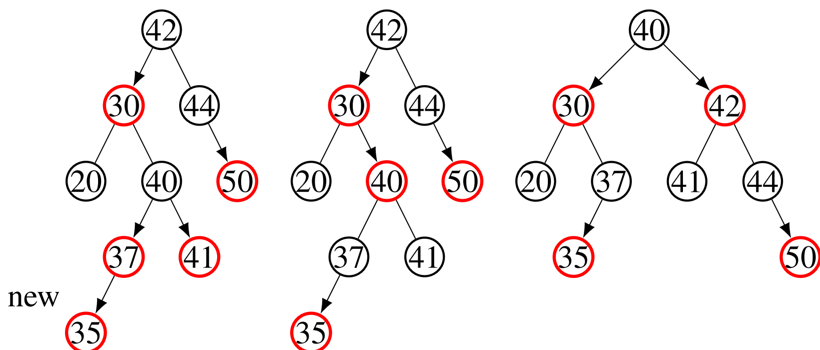


- If the root has been coloured red, make it black.

The rotations to restructure red-red conflicts (with black uncle) are just the classical binary tree rotations that preserve the search tree order. Likewise we can have single or double rotations (in both directions). Note the rotation also chooses proper new colours. The new root will be black, the two children will be red.



Example 6.9. We add 35 to the red-black tree from Example 6.6, as a red leaf, left child of 37 which itself is red. Uncle 41 is also red, so we perform flag-flip. The newly red coloured node 40 is the child of red 30. Uncle 44 is black. We restructure by rotation at 40 (double, to the right) with the recolouring of the three nodes involved. Done.



It is worthwhile to draw the same trees, but now as B-trees.

◇

Example 6.10. From the introductory comments in stl_tree.h for the GNU C++ Library.

"Red-black tree class, designed for use in implementing STL associative containers (set, multiset, map, and multimap). The insertion and deletion algorithms are based on those in Cormen, Leiserson, and Rivest, Introduction to Algorithms (MIT Press, 1990), except that ..."

The Linux kernel seems to be full of red-black trees.

"There are a number of red-black trees in use in the kernel. The anticipatory, deadline, and CFQ I/O schedulers all employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the 'hierarchical token bucket' scheduler." lwn.net/Articles/184495/

Final remarks. ☒ Red-black trees are considered to originate from the symmetric binary B-trees proposed by Bayer, one of the inventors of the B-tree. They were developed by Guibas and Sedgewick.

Implementations sometimes use a parent pointer to locate the parent when updating the tree, but other implementations are recursive, where the updates can be performed on the way up.

Sedewick now advocates *left-leaning red-black trees*, where a single red child can only occur at the left. (Two red children are OK.) Such trees are more restrictive, which may lead to less cases to consider. Others argue that because the asymmetry the code will become more complex. Also studied are *AA-trees*, where a red node can only be at the right. They form an implementation for 2–3 trees (with one or two keys in a node).

References. R. BAYER: Symmetric binary B-Trees: Data structure and maintenance algorithms, Acta Informatica (1972) 290–306. doi:10.1007/BF00289509

L.J. GUIBAS, R. SEDGEWICK: A Dichromatic Framework for Balanced Trees, Proceedings of the 19th Annual Symposium on Foundations of Computer Science (1978) 8–21. doi:10.1109/SFCS.1978.3

See CLRS Chapter 13: Red-Black Trees

See DROZDEK Section 7.1.8: 2-4 Trees

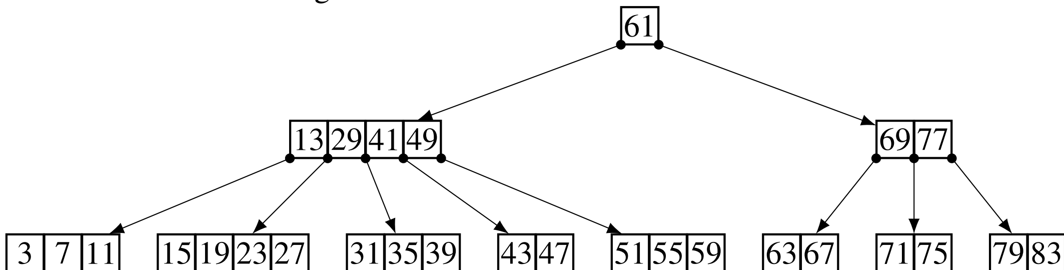
See WEISS Section 12.2: Red-Black Trees

R. RAMAKRISHNAN, J. GEHRKE: Database Management Systems, Ch10: Tree indexes (Lecture Databases)

Opgaven

1.a) Wat is het maximale en minimale aantal sleutels dat geplaatst kan worden in een B-boom van orde m en drie nivo's diep?

Beschouw de volgende B-boom B van orde 5:



Zorg ervoor dat we bij het toevoegen en verwijderen steeds een B-boom van orde 5 behouden. Geef een korte toelichting en (zo nodig) tussenresultaten. Ongewijzigde delen van de boom kunnen schematisch worden aangegeven.

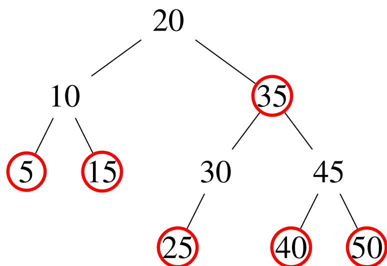
b) (i) Voeg aan B de sleutel 24 toe.

(ii) Voeg aan B (de originele boom dus) de sleutel 50 toe.

c) Verwijder de sleutel 77 uit B (alweer de originele boom). *Mrt 2016*

2.a) Wat zijn de definiërende eigenschappen van de rood-zwart boom (*red-black tree*)?

b) Gegeven is de volgende rood-zwart boom, waarbij 'rode' knopen een (rode) cirkel hebben gekregen.



Welke boom ontstaat als we hieraan toevoegen

(i) eerst 55 en vervolgens 7.

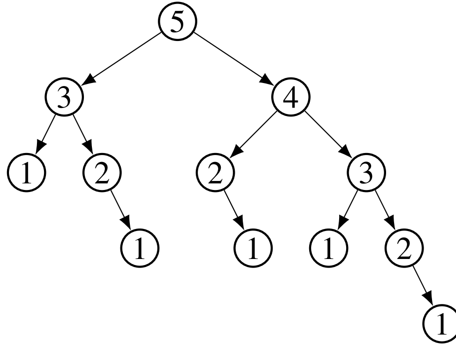
(ii) eerst 7 en vervolgens 55.

Benoem de achtereenvolgende operaties en geef relevante tussenresultaten.

Jan 2017

3. Volg de suggestie uit Example 6.9, en vertaal het voorbeeld naar 2-4-bomen. Waarmee correspondeert een flag-flip?

4. Consider the (Fibonacci) tree in the diagram below, where node values indicate the ordinary height of the subtree. Colour the nodes to obtain a red-black tree; indicate the black height for each black node.



7 Graphs

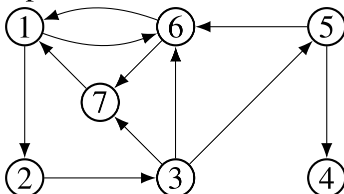
7.1 Representation

A *graph* $G = (V, E))$ consists of a set V of *vertices* (or *nodes*) and a set E of *edges* (or *arcs*, *lines*). We distinguish between directed and undirected graphs. Standard representations are the *adjacency matrix* and the *adjacency lists*. If we want to easily query the existence of an edge $(u, v) \in E$, then the matrix is efficient. If we need to check all outgoing edges of a given vertex, then the lists work well.

If the graph contains n vertices and e edges, the size of its adjacency matrix equals $O(n^2)$ while the adjacency lists representations uses space $O(n+e)$. Recall that e is at most n^2, or $\frac{n(n-1)}{2}$ for undirected graphs.

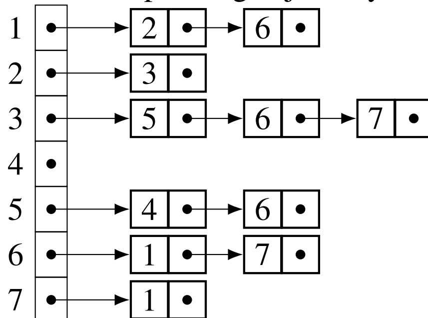
Note that these standard representations need not always be the best choice. In Example 7.5 we have a kind of rectangular map where each vertex has at most four neighbours. In that case each vertex can be represented implicitly by a pair of coordinates (implicit meaning that we do not directly associate objects to the vertices) and the existence of horizontal and vertical edges can be written in an array (which is different from the adjacency matrix!).

Example 7.1. A directed graph.



Left: the adjacency matrix of the above graph, where in order to preserve legibility the dots are used to indicate zero's. Right: the corresponding adjacency lists.

	1	2	3	4	5	6	7
1	·	1	·	·	·	1	·
2	·	·	1	·	·	·	·
3	·	·	·	·	1	1	1
4	·	·	·	·	·	·	·
5	·	·	·	1	·	1	·
6	1	·	·	·	·	·	1
7	1	·	·	·	·	·	·



7.2 Graph traversal

Graph traversals are techniques to visit the nodes of a graph. *Depth-first search* and *breadth-first search* generalize the tree traversals pre-order and level-order, using a stack and a queue, respectively. As graphs may contain cycles, we have to take care that the traversals do not end in an infinite loop. This is done by marking nodes as visited.

Depth-first search

With depth-first search (DFS) we try to move forward along a path as much as possible, storing the unvisited neighbors on a stack. When reaching a dead-end (or already visited nodes) the algorithm back-tracks and pops a node from the stack.

Recursive DFS

```
DFS(v)
    visit(v)
    mark(v)
    for each w adjacent to v
    do  if w is not marked
        then DFS(w)
        fi
    od
end // DFS
```

Iterative DFS

```
// start with unmarked nodes
S.push(init)
while S is not empty
do  v = S.pop()
    if v is not marked
    then mark v
        for each edge from v to w
        do  if w is unmarked
            then S.push(w)
            fi
        od
    fi
od
```

The algorithm implicitly builds a *DFS-tree*: when we pop and visit a vertex w that was pushed on the stack at the time v was visited we actually follow edge (v,w), which is part of the tree. It can be the case that during a traversal a vertex is "seen" at several moments from different neighbours. It is pushed to the stack each time, as it is the nature of depth-first to back-track from the last time we have seen the vertex[5].

There is no unique DFS tree: its structure will depend on the order in which the adjacent nodes are processed at each node.

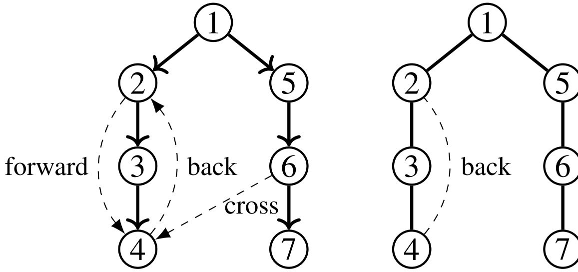
The (directed) graph may not be strongly connected and the DFS when started at a vertex will not visit all the vertices. Therefore, in general, we start the DFS at all vertices in succession. This can be done by pushing all vertices on the stack.

The edges in the original (directed) graph are categorized into four types based on the DFS. Tree edges belong to the DFS-tree; *forward* edges point to successors

[5] Marking vertices when they are added to the stack, and not adding marked vertices yields a valid graph traversal algorithm. It does not have the structure of depth-first.

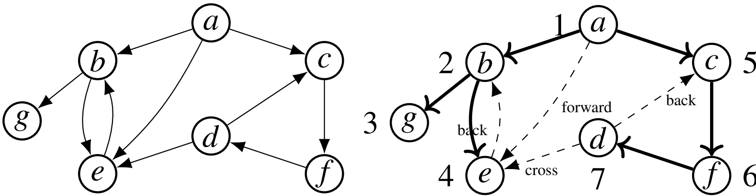
in the tree, while *back* edges point to predecessors; finally *cross* edges point to vertices in another subtree (which was searched earlier).

For undirected graphs there is no distiction between forward and back edges. Also, cross adges cannot exist: an edge connecting two subtrees would be traversed by the algorithm to visit that subtree.

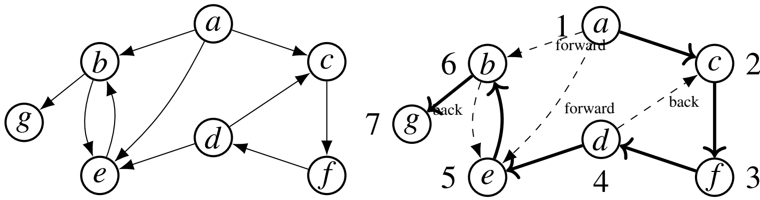


Example 7.2. We perform a DFS on the given graph, starting at vertex a. When giving the stack the leftmost vertex is at the top of the stack. Vertices have their parents (at the time they were pushed) as a superscript. This helps to reconstruct the spanning tree of the traversal.

		push initial a	a
pop a	visit and mark	push adjacent b,e,c (reverse order)	b^a, e^a, c^a
pop b	visit and mark	push adjacent g, e	g^b, e^b, e^a, c^a
pop g	visit and mark	no children	e^b, e^a, c^a
pop e	visit and mark	adjacent b is marked/visited, not pushed	e^a, c^a
pop e	marked, already visited		c^a
pop c	visit and mark	push adjacent f	f^c
pop f	visit and mark	push adjacent d	d^f
pop d	visit and mark	adjacent e, d are already marked	
	empty stack	done	



The DFS tree is not unique. It depends on the order in which the successors of a node are pushed on the stack. In our current example we might start with node c after the initial a, then the DFS tree would look as follows.



◇

Two important applications of DFS are *topological ordering* and structural properties like connectedness in the graph. An example of the latter is the detection of nodes the deletion of which will split the graph.

Articulation points. We present an application of depth-first traversal of a graph. The *articulation points* (or cut points) of an undirected graph are those vertices that removing one of them them (with their incident edges) will increase the number of components of the graph.

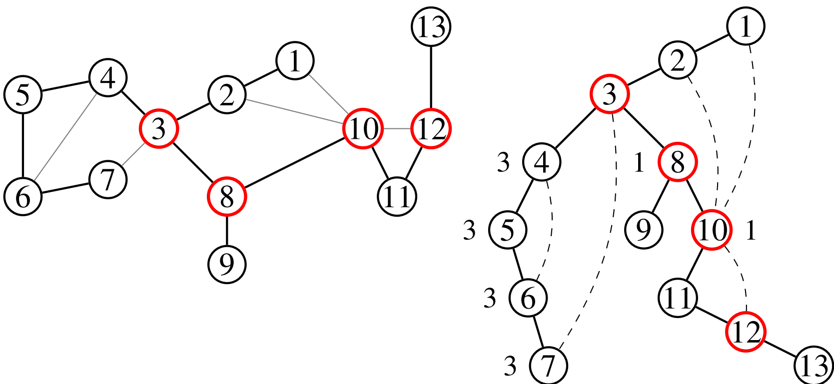
The articulation points can be found using the depth-first tree of the graph. We use the following characterization.

Lemma 7.3. *A vertex v is an* articulation *point if either*

- *v is the root, and has two or more children, or*
- *v has a (strict) subtree, and no node in the subtree has a back edge that reaches above v.*

Example 7.4. An undirected graph and its DFS tree. The articulation points are marked in red. Cutting for example 10 from the graph will separate nodes $11, 12, 13$ from the rest of the graph.

Consider vertex 3. Its right subtree in DFS (with root 8) has a successor 10 with a back edge to 2 which is above 3. Its left subtree in DFS (with root 4) however, has no successors with a back edge jumping above 3.



◇

Breadth-first search

Breadth-first search BFS visits all vertices in "concentric circles" around the initial vertex. Thus the vertices are selected based on their distance from the initial vertex.

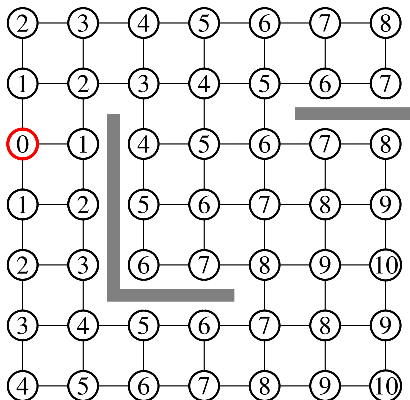
We can change the moment vertices are marked. Vertices that are rediscovered later do not have priority over earlier discoveries. Hence we can mark vertices as soon they are found and put in the queue.

Iterative BFS

```
    // Q is a queue of vertices
    // start with unmarked nodes
Q.enqueue(init)
dist[init] = 0
while Q is not empty
do  v = Q.dequeue()
    newdist = dist[v] + 1
    for all edges from v to w
    do  if w is not marked
        then Q.enqueue(w)
            mark w
            dist[w] = newdist
        fi
    od
od
```

Example 7.5. Various tasks, like "flood-fill" (colouring pixels in a closed area) and robot motion planning can be seen as the application of breadth-first search on (implicit) graphs. The graphs are not always explicitly specified, e.g., vertices may be pixels or 'tiles' in a map, and edges indicate their neighbour relation.

In the picture below a graph with one marked node and other nodes with their 'breadth-first' distance to the marked node over the allowed edges.



Note that this 'graph' can be represented by matrices representing horizontal and vertical edges:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad V = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

References. J. HOPCROFT, R. TARJAN: Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM **16** (1973) 372–378, doi:10.1145/362248.362272 (describes an algorithm for biconnected components, which uses articulation points)

See DROZDEK Chapter 8.2: Graph Traversals; 8.6: Connectivity

See WEISS Chapter 9.6: Applications of Depth-First Search

7.3 Disjoint Sets, ADT Union-Find

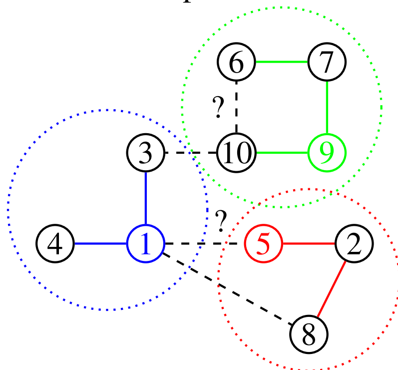
The data structure for disjoint sets is equipped for keeping track the connected components in an undirected graph while adding edges. Initially no vertices are connected, so each vertex is a component in itself. When an edge is added between two different components, the two are joined, UNION. Another operation FIND is able to retrieve the component in which a vertex belongs. In this way it can be checked whether or not two vertices belong to the same component. To make this possible we assign to each component a “name” which usually is a representative vertex in the component.

As the relative ordering of elements is irrelevant, it is customary to consider a domain $D = \{1, 2, \ldots, n\}$ of vertices. The components form a *partition* of the domain.

- INITIALIZE: construct the initial partition; each component consists of a singleton set $\{d\}$, with $d \in D$.
- FIND: retrieves the name of the component, i.e, $Find(u) = Find(v)$ iff u and v belong to the same set in the partition.
- UNION: given two elements u and v the sets they belong to are merged. Has no effects when u and v already belong to the same set.
  Usually it is assumed that u, v are representatives, i.e., names of components, not arbitrary elements.

Example 7.6. We want to find a spanning tree of a graph, by repeating the following step. Start with an empty forest. Take an edge in the graph. Check whether an edge can be added to the forest without causing a cycle. When this is possible add the edge, and continue with the new forest.

In the graph below, the forest connects three components. The edge $(6,10)$ is within one of these components and cannot be added as it would form a cycle. Edge $(1,5)$ connects two different components and can be added.



Let $D = \{1,2,\dots,10\}$. Then $\{\{\underline{1},3,4\},\{6,7,\underline{9},10\},\{2,\underline{5},8\}\}$ is a partition for D in three sets. The names (representing elements) of each set are underlined.

Thus $Find(6) = 9 = Find(10)$, indicating 6 and 10 belong to the same set. At the same time $Find(3) = 1 \neq 5 = Find(5)$, indicating 3 and 5 do not belong to the same set.

Now $Union(1,5)$ will result in the partition $\{\{1,2,3,4,5,8\},\{6,7,9,10\}\}$, with some assignment of representing elements.

Linked lists. In the naive approach we keep an array which lists for each vertex the set it belongs to. Obviously this gives a very efficient FIND operation. For UNION we have to rename the representative x for one of the sets into the representative y of the other. Without further bookkeeping we have to loop over the full array, and at each position check for x.

Thus, the complexity for each operation: FIND $O(1)$, and UNION $O(n)$ time.

1	2	3	4	5	6	7	8	9	10	
1	5	1	1	5	9	9	5	9	9	find

UNION$(9,5)$

1	2	3	4	5	6	7	8	9	10	
1	9	1	1	9	9	9	9	9	9	find

By adding a linked list for each of the sets in the partition we do not have to look at every position. We can perform a faster renaming, by iterating over the

shortest list. Joining lists can be done in constant time (e.g., if we keep a circular list).

Now the worst case *total* complexity of n UNION operations is $O(n\log(n))$. Worst case builds lists of size 2,4,8, etc.. Each FIND is performed in constant time.

Example 7.7. Start with a collection of ten objects. The lists are chained via the next field. Recall Example 1.2. The initial configuration starts thus.

1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	find
1	2	3	4	5	6	7	8	9	10	next
1	1	1	1	1	1	1	1	1	1	size

After $Union(1,3)$, $Union(9,10)$, $Union(5,8)$, and $Union(6,7)$ the arrays look as follows. Dots indicate don't cares: size is only important for representative names.

1	2	3	4	5	6	7	8	9	10	
1	2	1	4	5	6	6	5	9	9	find
3	2	1	4	8	7	6	5	10	9	next
2	1	.	1	2	2	.	.	2	.	size

Then $Union(9,6)$ relabels all elements from the list of 6 to 9, the new name of the combined set, and joins the two lists.

1	2	3	4	5	6	7	8	9	10	
1	2	1	4	5	9	9	5	9	9	find
3	2	1	4	8	10	6	5	7	9	next
2	1	.	1	2	.	.	.	4	.	size

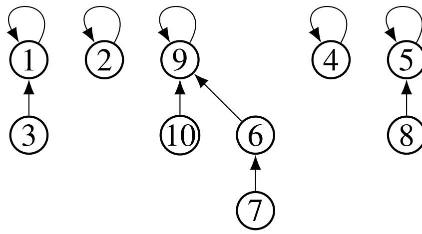
◇

References. See DROZDEK Chapter 8.4.1: Union-Find Problem

Inverted trees. The operation Union can be improved, at the cost of Find, by representing the sets, by upside-down trees. The root of such a tree is the name of the set. Union can be performed in constant time by adding a simple link. In order to optimize Find we can store the heights of the trees, so choose which tree to join to the other. The trees can be represented by a simple array indicating the parent for each node. The root of a node either points to itself (or to a special value).

Example 7.8. After $Union(1,3)$, $Union(9,10)$, $Union(5,8)$, $Union(6,7)$ and $Union($ the arrays look as follows.

1	2	3	4	5	6	7	8	9	10	
1	2	1	4	5	9	6	5	9	9	parent
2	1	.	1	2	.	.	.	3	.	height



◇

Path compression. Now Find will take several steps to locate the root of the tree. Luckily the worst case tree has an height logarithmic in the number of nodes. Assuming that the same node is searched several times we can improve the complexity of Find. When we have located a root-node, revisit the path and set all parents along the path to the root we have just found. This will double the number of steps, which does not change the order of complexity, but in subsequent Finds's to the same node we find the root in constant time.

Here is a schematic picture for UNION(v,w) and for FIND(v) before and after the operation, see Slide 18. All nodes along the path benefit from the Find operation, as well as their subtrees.

The height values of the trees after path compression do not always match the new trees, but this turns out to be not a real problem.

It can be shown that the complexity of any sequence of n Union and Find operations is 'almost linear' in a mathematically well defined sense. The complexity is $\alpha(n)\cdot n$, where the extra factor $\alpha(n)$ is the inverse of the Ackerman function. This function grows very slow: for all practical purposes $\alpha(n) \leq 5$.

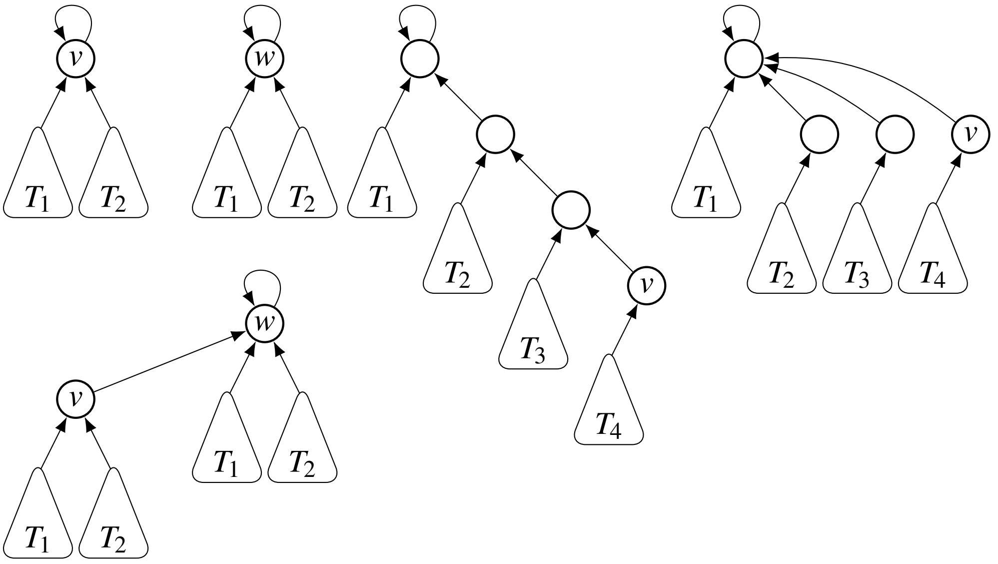
References. B.A. GALLER, M.J. FISCHER: An improved equivalence algorithm. Communications of the ACM **7** (1964) 301–303, doi:10.1145/364099.364331. The paper originating disjoint-set forests.

See CLRS Chapter 21: Data Structures for Disjoint Sets

See WEISS Chapter 8: The Disjoint Sets Class

On cstheory.stackexchange there is a poll to vote for the "Algorithms from the Book", a collection of the most elegant algorithms. Union-Find is in first place, presently with 121 votes, just beating runner-up Knuth-Morris-Pratt with 112 votes. (accessed 15-10-2021)

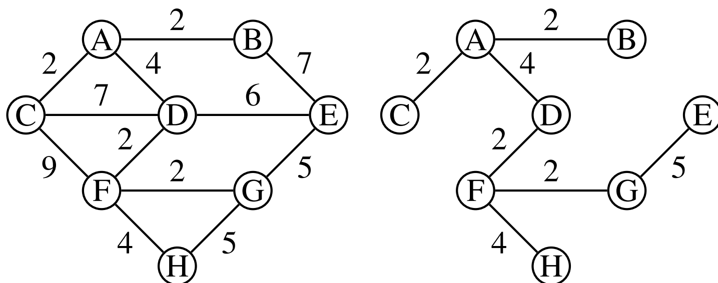
Slide 18 Union-Find. Left diagram: $\textsc{Union}(v,w)$. Middle and right: $\textsc{Find}(v)$ with path compression (before and after).



7.4 Minimal Spanning Trees

A minimal spanning tree in a undirected weighted graph is a tree connecting all the vertices in such a way that the total weight of the vertices is minimal considering all spanning trees. We consider two greedy algorithms for finding minimal spanning trees in graphs: *Prim* and *Kruskal*.

Example 7.9. An (undirected) graph and a minimal spanning tree.



Kruskal's algorithm

The algorithm of Kruskal considers edges sorted on their weight one by one and adds them to the constructed spanning tree as far as they do not lead to a cycle with the already chosen edges. The basic difficulty is checking whether the next edge added will cause a cycle. An important data structure for keeping track of the different components in a dynamically growing graph is Union-Find, which was the topic of Section 7.3 above.

```
Kruskal (high level)

start with forest without edges
repeat
    consider edge with smallest weight
    if it does not yield a cycle
    then add it to the forest
    else discard the edge
    fi
until no edges left
```

Prim's algorithm

Unlike the algorithm of Kruskal, the algorithm of Prim forms a tree in every stage. New edges always have one end inside the current tree, and one end outside.

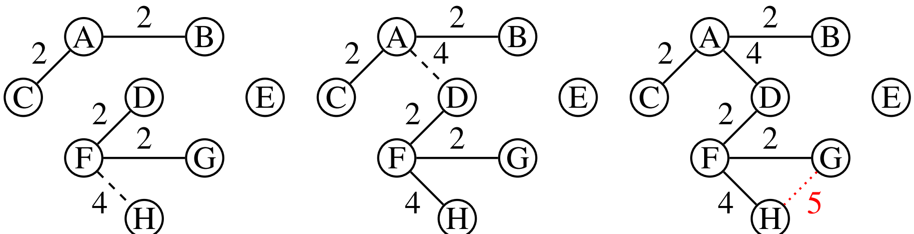
```
                    Prim (high level)
start with single node
repeat
    consider edge with smallest weight
            connecting node in tree with one outside
    add new node+edge to the tree
until all nodes in tree
```

Note however, that if v is a vertex outside the partial spanning tree T, and both u_1 and u_2 are vertices inside the tree, then not both (u_1,v) and (u_2,v) are part of the final spanning tree: adding both will cause a cycle. This means that we can save on the efforts needed to select edges. For each vertex outside of the tree we have to store at most one incident edge, the one of minimal weight. Now rather than to find the minimal edge among all edges, we look for one edge among all edges associated to the remaining vertices. That saves work, as there are less vertices than there are edges.

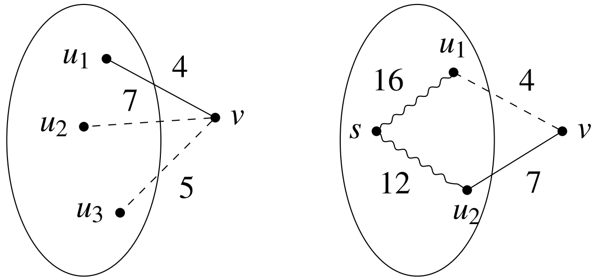
In the algorithm below, the edge associated with node v is defined by the other side of the edge, node parent[v]. The length of that edge is cost[v].

```
                         Prim
cost[source] = 0         // infinite for other vertices
parent[source] = 0       // code for the root
PQ = V                   // all vertices
while PQ is not empty
do  u is vertex in PQ with minimal cost[u]
    remove u from PQ
    for each edge (u,v)
    do  if length(u,v) < cost[v]
        then cost[v] = length(u,v)
            parent[v] = u
        fi
    od
od
```

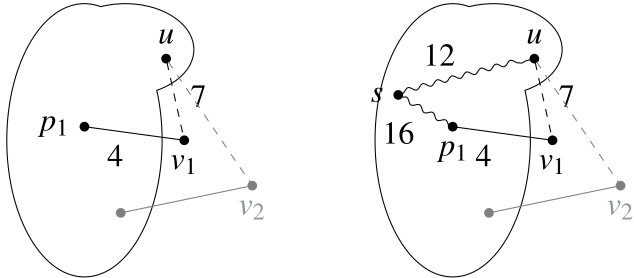
Example 7.10. Minimal spanning tree, comparing Kruskal (top) and Prim (bottom) algorithms. Zie Algoritmiek.

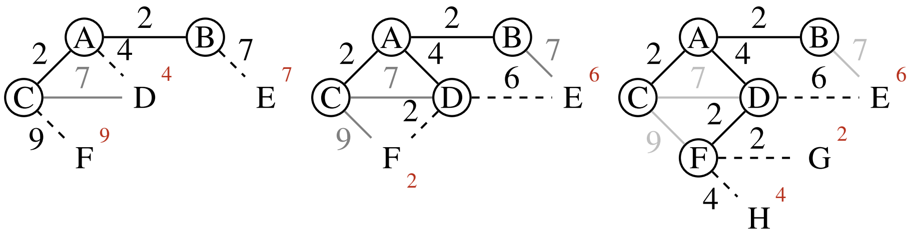


Slide 19 Keeping the best 'incoming' connection to nodes outside the tree. Left: Prim choosing minimal weight edges. For node v the optimal connection is from u_1 with cost 4. Right: Dijkstra: minimal distance includes the path from source to the node. The optimal connection to node v is via u_1, as the total distance $12+7 < 16+4$.



Slide 20 Node u just has been added to tree. Updating adjacent nodes outside tree. Left: Prim. For v_1 the previous best connection is from 'parent' p_1. The new connection via u does not improve this, as $7 \not< 4$. Right: Dijkstra. The distance to u has been fixed to 12. The best path to v_1 until now is via parent p_1 and has length $20 = 16+4$. The new connection via u has length $19 = 12+7$ and improves the previous path. The parent for v_1 will now be set to u.





◇

Correctness. The Prim algorithm constructs a set of successive trees $T_1 \subseteq T_2 \subseteq \cdots \subseteq T_n$ by adding in each step a vertex with an incident edge, where T_i contains i vertices, and n is the number of vertices in the given graph. We show that in each step there is a minimal spanning tree T such that $T_i \subseteq T$.

This is true for T_1, as that tree has no edges we can start choosing T_1 to be any mst of the graph.

Assume that in T_i we choose an edge e. If e is also in T we proceed. Otherwise, if e is not in T we construct a new 'goal' tree of the same total weight, so it also is minimal. Adding edge e to T we must find a cycle. Let X be the set of vertices of T_i. As e has only one endpoint in X this cycle is partly within X and partly outside. Follow the cycle, and look at the edge e' where the cycle re-enters X. Note e' was not chosen by the algorithm, so weight$(e') \geq$ weight(e). If we replace e' by e in T we again obtain a tree T', the weight of which is not larger than that of T, so it must be minimal too. Our new goal for Prim is the tree T' and we proceed.

Implementation. For implementation and the resulting complexity of the algorithm we refer to the section on Dijkstra below. The algorithms are structurally similar.

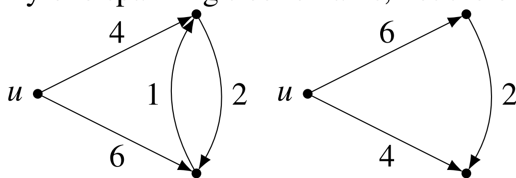
Note that we repeatedly have to choose a minimal cost edge associated to the vertices. That can be done by a linear search among those edges, or we can keep the edges (or rather the vertices that are associated with the edges) in a min priority queue. Note that we need to be able to perform the operation DECREASEKEY on a vertex in the queue: when a new incident edge is found that is less costly than the previous one, we have to update the cost.

Example 7.11. Both Prim and Kruskal are defined for undirected graphs. Here we will see these algorithms cannot be extended to directed graphs in a simple way. First note the directed problem only makes sense if we consider a node u, and there actually exist spanning trees starting with u.

The algorithms of Prim and Kruskal make choices in a greedy way. Once the choice is made, it will not be reconsidered. We show how this choice will go wrong for certain directed examples.

The left picture has three spanning trees, with weight $4+2$, weight $6+1$, and weight $6+4$. Kruskal will choose the lightest edge 1. Now, only one spanning tree remains, not the best one.

The right picture has two spanning trees, with weight $6+2$ and weight $6+4$. Starting with node u Prim will consider the two edges 4 and 6 and choose the lightest one. Now, only one spanning tree remains, not the best one.



Kruskal fails   Prim fails

A solution for the directed spanning tree problem was given by Edmonds.

◇

References. J. EDMONDS: Optimum Branchings. J. Res. Nat. Bur. Standards **71B** (1967) 233–240, doi:10.6028/jres.071b.032

J.B. KRUSKAL: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7** (1956) 48–50. doi:10.1090/S 9939-1956-0078686-7

R.C. PRIM: Shortest connection networks and some generalizations. Bell System Technical Journal **36** (November 1957) 1389–1401,
doi:10.1002/j.1538-7305.1957.tb01515

The algorithm was discovered and published much earlier, unfortunately in a less accessible venue.

V. JARNÍK: O jistém problému minimálním. (Z dopisu panu O. Borůvkovi). Práce moravské pří rodovědecké společnosti **6**, (1930) 57–63. (in Czech: On a certain problem of minimization, Proceedings of the Moravian Scientific Society.) doi:10338.dmlcz/500726

See CLRS Chapter 23.2: The Algorithms of Kruskal and Prim

See DROZDEK Chapter 8.5: Spanning Trees

See LEVITIN Chapter 9.1& 2: Prim's & Kruskal's Algorithm

See WEISS Chapter 9.5: Minimum Spanning Tree

7.5 Shortest Paths

Dijkstra's algorithm

Given a startnode source in a graph G with positive edge lengths we compute the distances to all other nodes in the graph, together with the tree of shortest paths starting from source. The tree of shortest distances is coded via the parents.

```
Dijkstra

dist[source] = 0    // infinite for other vertices
parent[source] = 0  // code for the root
Q = V               // start with all vertices
while Q is not empty
do  u is vertex in Q with minimal dist[u]
    remove u from Q
    for each edge (u,v)
    do  if dist[u] + length(u,v) < dist[v]
        then dist[v] = dist[u] + length(u,v)
             parent[v] = u
        fi
    od
od
```

Complexity. Let $n = |V|$ and $e = |E|$ be the number of vertices and of edges of the graph, respectively. Initialization of the arrays costs $O(n)$. There are n iterations of the main loop where we find the minimal element (*minimal dist*). If we have to look inside an array this will cost a total of $O(n^2)$. Then we look at all outgoing edges from all vertices, which will cost $O(e)$ assuming we use adjacency lists, and $O(n^2)$ using adjacency matrix representation (*for each edge*). The total complexity is $O(n+n^2+e) \subseteq O(n^2)$.

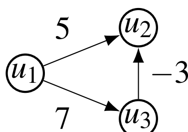
We may improve on this by using a priority queue, in the form of a binary heap, see Section 5.2, which allows the operation DECREASEKEY (*dist* < *dist*). The number of heap operations totals at most $n+e$ when each edge finds a shorter distance. The complexity then equals $O(e\log(n))$, which is better than $O(n^2)$ if the number of edges is relatively small.

Clever implementations of a priority queue have $O(\log(n))$ complexity for DELETEMIN and constant time DECREASEKEY. This results in an overall complexity of $O(n+n\log(n)+e) \subseteq O(n\log(n)+e)$.

	list	heap	clever	
initialize	n	n	n	
minimal dist	n^2	$n\cdot \lg n$	$n\cdot \lg n$	DeleteMin
each edge	e	$e\cdot \lg n$	e	DecreaseKey(!)
	n^2	$e\cdot \lg n$	$e+n\cdot \lg n$	

Dijkstra (in this implementation) does *not* work when the graph has negative edge-lengths. It assumes that the distance to a vertex never is found via another vertex that has a larger distance.

Example 7.12. We start with source u_1. The source has distance 0, are other are 'infinite'. Dijkstra updates the distances to u_2 and u_3 to 5 and 7 respectively. Minimal distance node now is u_2 and Dijkstra fixes it at 5. Then last node u_3 is selected, distance is fixed at 7. The algorithms does not look at the edge (u_3, u_2) which would shorten the distance to u_2.



◇

Bottleneck paths. With a little thought Dijkstra's algorithm can be changed to another version of optimal paths, where we try to find the "widest path" to every vertex (or the path with "maximum flow"). Here the value of an edge indicates a restriction (maximal width, height, capacity) and the restriction along the path is the minimum over the edges. Thus a path with edges $3,6,5$ has bottleneck $\max\{3,6,5\} = 3$. We want to avoid the bottleneck, thus we look for the maximal possible value over all paths. Whereas Dijkstra computes

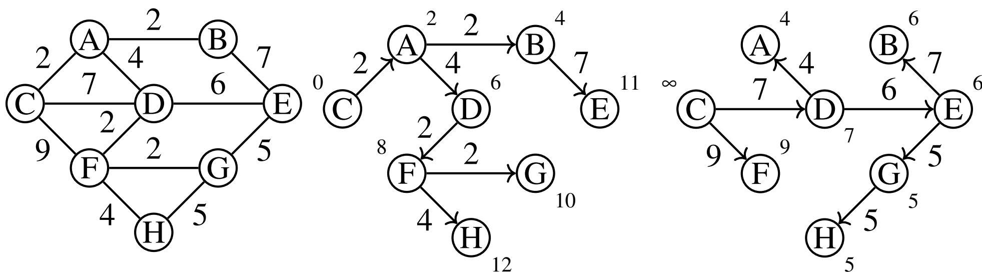
$$\text{dist}[v] = \max\{\ \text{dist}[v], \text{dist}[u] + \text{length}(u,v)\ \}$$

for the bottleneck variant we choose

$$\text{bott}[v] = \max\{\ \text{bott}[v], \min\{\text{bott}[u], \text{width}(u,v)\}\ \}$$

when considering a new edge (u,v).

Example 7.13. An (undirected) graph and with initial vertex C. (Middle) A shortest distance tree, and (Right) an optimal bottleneck tree. The values at the vertices indicate final distances and bottleneck restrictions, respectively.



◇

References. E.W. DIJKSTRA: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271. doi:10.1007/BF01386390

D.B. JOHNSON: Efficient algorithms for shortest paths in sparse networks. Journal of the ACM. **24** (1977) 1—13. doi:10.1145/321992.321993 for the implementation using a heap

See LEVITIN Chapter 9.3: Dijkstra's Algorithm

See WEISS Chapter 9.3.2: Dijkstra's Algorithm

All-Pairs Shortest Paths

The problem of *all-pairs shortest paths* asks to determine a complete matrix of distances between every pair of vertices. With a little care not only the distances, but also the corresponding shortest paths themselves can be retrieved. As there are n pairs of nodes we will not explicitly store all these paths, but enough information to retrieve them.

Floyd-Warshall

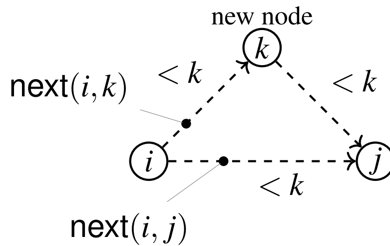
```
// initially dist equals the adjacency matrix
for each edge (i,j)
do  next[i,j] = j
od
for k from 1 to n
do  for i from 1 to n
    do  for j from 1 to n
        do  if dist[i,k] + dist[k,j] < dist[i,j]
            then dist[i,j] = dist[i,k] + dist[k,j]
                 next[i,j] = next[i,k]
            fi
        od
    od
od
```

The algorithm of *Floyd* uses the technique of *dynamic programming* to compute the shortest distances. With n vertices, we assume the vertices are the numbers $1,\dots,n$. For each $k = 0,1,\dots,n$ a matrix A^k of partial solutions is computed, where $A^k[i,j]$ denotes the distance from vertex i to vertex j using paths that do not pass any vertex with number larger than k. In particular the paths for matrix A^0 do not pass any vertex, they can only be paths of a singe edge (i,j) (where no vertices are passed). The algorithm then adds the vertices $1,\dots,n$ one by one as possible intermediate stops.

The node $\text{next}(i,j)$ is the first node on the current shortest path from i to j.

When matrix A^{k-1} has been determined we compute A^k. The shortest path from i to j without using vertices $k+1$ or larger either uses vertex k or not. The distance via k equals $A^{k-1}[i,k] + A^{k-1}[k,j]$, while the distance that does not use k already has been determined as $A^{k-1}[i,j]$.

Hence $A^k[i,j]$ is the minimum of $A^{k-1}[i,j]$ and $A^{k-1}[i,k] + A^{k-1}[k,j]$.



When a shorther distance is found, via the new node k, the first node next(i,j) must be updated, it now lies on the segment from i to k.

Computations for A^k do not change the values at $[k,j]$ or $[i,k]$ in the matrix A^{k-1} so when implementing this algorithm we may in fact use only a single matrix.

Example 7.14. We refer to Slide 21 for an example where the algorithm is applied to a graph which has partially directed edges. The distance from 3 to 1 is computed as 5 according to the final matrix. The actual path from 3 to 1: next$(3,1) = 2$: this means the first edge is $3 \to 2$, and we have to resolve the remaining path from 2 to 1. Thus next$(2,1) = 4$, then next$(4,1) = 1$ (done), so $3 \to 2 \to 4 \to 1$.

◇

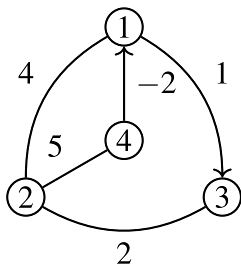
This algorithm is closely related to that defined by *Warshall*, which computes the transitive closure T of a directed graph, i.e., a Boolean matrix for which $T[i,j]$ holds iff there is a path from i to j. We start with a Boolean matrix, an unweighted adjacency matrix, and replace the operations min and $+$ by $\vee$ and $\wedge$.

The algorithm is also strongly related to Kleene's algorithm to transform a finite state automaton into a regular expression. The basic recursion there is $L(i,j) = L(i,j) + L(i,k) \cdot L(k,k)^* \cdot L(k,j)$.

Warshall

```
    // initially conn equals the adjacency matrix
    // with additionally 1=true on the diagonal
for k from 1 to n
do  for i from 1 to n
    do  for j from 1 to n
        do  conn[i,j] = conn[i,j] or ( conn[i,k] and conn[k,j] )
        od
    od
od
```

Slide 21 Voorbeeld uitwerking Floyd, opgave uit het tentamen van maart 2017. In de linkerkolom staan steeds de (op dat moment kortste) afstanden via knoop i, achtereenvolgens $i = 0, \ldots, 4$. Kijk dan steeds of die afstand een verbetering is, resultaat in de middelste kolom. Gegeven zijn daarnaast de matrices $\mathsf{next}(i, j)$, óók als er nog geen pad gevonden is.



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 4 & 1 & \infty \\ 4 & 0 & 2 & 5 \\ \infty & 2 & 0 & \infty \\ -2 & 5 & \infty & 0 \end{pmatrix} \end{array}$$

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 4 & 1 & \infty \\ 2 & 4 & . & 2 & 5 \\ 3 & \infty & 2 & . & \infty \\ 4 & -2 & 5 & \infty & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 4 & 1 & \infty \\ 2 & 4 & . & 4+1 & 4+\infty \\ 3 & \infty & \infty+4 & . & \infty+\infty \\ 4 & -2 & -2+4 & -2+1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 2 & 3 & 4 \\ 2 & 1 & . & 3 & 4 \\ 3 & 1 & 2 & . & 4 \\ 4 & 1 & 2 & 3 & . \end{array}$$

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 4 & 1 & \infty \\ 2 & 4 & . & 2 & 5 \\ 3 & \infty & 2 & . & \infty \\ 4 & -2 & \boxed{2} & \boxed{-1} & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 4 & 4+2 & 4+5 \\ 2 & 4 & . & 2 & 5 \\ 3 & 2+4 & 2 & . & 2+5 \\ 4 & 2+4 & 2 & 2+2 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 2 & 3 & 4 \\ 2 & 1 & . & 3 & 4 \\ 3 & 1 & 2 & . & 4 \\ 4 & 1 & 1 & 1 & . \end{array}$$

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 4 & 1 & \boxed{9} \\ 2 & 4 & . & 2 & 5 \\ 3 & \boxed{6} & 2 & . & \boxed{7} \\ 4 & -2 & 2 & -1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 1+2 & 1 & 1+7 \\ 2 & 2+6 & . & 2 & 2+7 \\ 3 & 6 & 2 & . & 7 \\ 4 & -1+6 & -1+2 & -1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 2 & 3 & 2 \\ 2 & 1 & . & 3 & 4 \\ 3 & 2 & 2 & . & 2 \\ 4 & 1 & 1 & 1 & . \end{array}$$

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & \boxed{3} & 1 & \boxed{8} \\ 2 & 4 & . & 2 & 5 \\ 3 & 6 & 2 & . & 7 \\ 4 & -2 & \boxed{1} & -1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 8+1 & 8-1 & 8 \\ 2 & 5-2 & . & 5-1 & 5 \\ 3 & 7-2 & 7+1 & . & 7 \\ 4 & -2 & 1 & -1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 3 & 3 & 3 \\ 2 & 1 & . & 3 & 4 \\ 3 & 2 & 2 & . & 2 \\ 4 & 1 & 1 & 1 & . \end{array}$$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 3 & 1 & 8 \\ 2 & \boxed{3} & . & 2 & 5 \\ 3 & \boxed{5} & 2 & . & 7 \\ 4 & -2 & 1 & -1 & . \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & . & 3 & 3 & 3 \\ 2 & 4 & . & 3 & 4 \\ 3 & 2 & 2 & . & 2 \\ 4 & 1 & 1 & 1 & . \end{array}$$

Example 7.15. ☒ This is how Floyd's algorithm was first presented. (From the journal style guide: *Algorithms should be in the Reference form of ALGOL 60 and written in a style patterned after the most recent algorithms appearing in this department. For the convenience of the printer, please underline words that are delimiters to appear in boldface type.*)

ALGORITHM 97 SHORTEST PATH
ROBERT W. FLOYD Armour Research Foundation, Chicago, Ill.

procedure shortest path (m,n); **value** n; **integer** n; **array** m;
comment Initially $m[i,j]$ is the length of a direct link from point i of a network to point j. If no direct link exists, $m[i,j]$ is initially 1010. At completion, $m[i,j]$ is the length of the shortest path from i to j. If none exists, $m[i,j]$ is 1010. Reference: WARSHALL, S. A theorem on Boolean matrices. *J ,ACM 9*(1962), 11–12;
begin integer i, j, k; **real** inf, s; inf := 1010;
for i := 1 **step** 1 **until** n **do**
for j := 1 **step** 1 **until** n **do**
if m[j,i] < inf **then**
for k := 1 **step** 1 **until** n **do**
if m[i,k] < inf **then**
begin s:= m[j,i] + m[i,k];
if s < m[j,k] **then** m[j,k] := s;
end
end shortest path

◇

Although conceptually very simple, it is more efficient to compute the *strongly connected components* of the graph to determine the connectivity between vertices. Algorithms to do this are based on depth-first traversal of the graph.

References. ROBERT W. FLOYD. Algorithm 97: Shortest Path. Communications of the ACM **5** (June 1962) 345. doi:10.1145/367766.368168

STEPHEN WARSHALL. A theorem on Boolean matrices. Journal of the ACM **9** (January 1962) 11–12. doi:10.1145/321105.321107

S.C. KLEENE. Representation of Events in Nerve Nets and Finite Automata. Automata Studies, Annals of Math. Studies. vol. 34, Princeton Univ. Press. 1956, pages 3–42. Kleene's algorithm transforms a nondeterministic finite automaton into an equivalent regular expression. It has the same (algebraic) structure as the Flod-Warshall algorithm.

See DROZDEK Chapter 8.3: Shortest Paths

See LEVITIN Chapter 8.4: Warshall's and Floyd's Algorithms

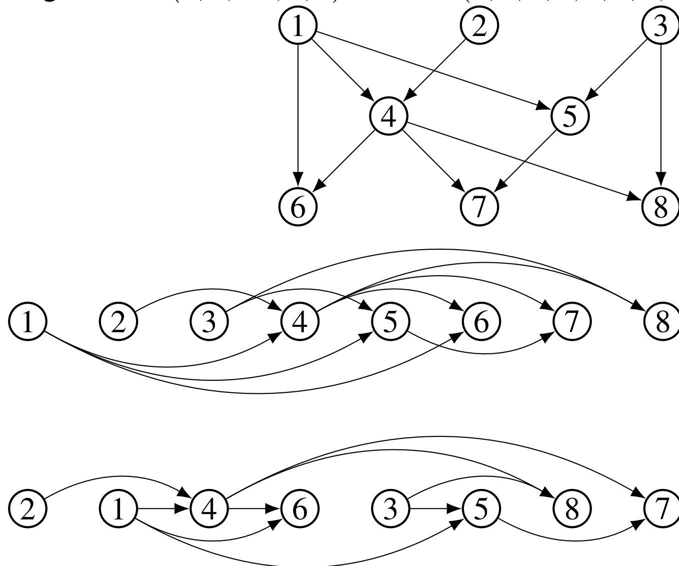
See WEISS Chapter 10.3.4: All-Pairs Shortest Path

7.6 Topological Sort

Definition 7.16. Let $G = (V, E)$ be a directed graph. A *topological ordering* [or *sort*] of G is an ordering $(v_1, \ldots, v_n)$ of V, such that if $(v_i, v_j) \in E$ then $i < j$.

In technical terms the topological sort is a linear ordering that is compatible with a partially ordered set. More informally it is an ordering on the nodes such that all edges are from left to right.

Example 7.17. Topological sorts are usually not unique. The given graph has a topological sort $(1, 2, \ldots, 7, 8)$ but also $(2, 1, 4, 6, 3, 5, 8, 7)$. And more.



--- ◇

Clearly a graph that has a topological ordering cannot have cycles. The converse is also true.

Theorem 7.18. *Let $G = (V, E)$ be a directed graph that has no cycles. Then G has a topological ordering.*

Proof. We start with an observation: G has a source, a node without incoming edges. Assume to the contrary that G has no source, every node has an incoming edge. Then following edges backwards we start a path, that eventually enters the same node twice. Hence we found a cycle; a contradiction. Using this observation we prove the theorem using induction. Take a source v of G. Then $G - \{v\}$ is also an acyclic directed graph. Hence by the induction hypothesis $G - \{v\}$ has a topological ordering $(v_1, \ldots, v_n)$ of $V - \{v\}$. Writing v in front we obtain a topological ordering $(v, v_1, \ldots, v_n)$ for G: any edge involving v must be outgoing. □

A useful observation is that the proof above in fact gives an algorithm that performs a topological sort. Find a source in the graph, write it as first element in the topological ordering, delete its (outgoing) edges, and repeat. Wikipedia calls this *Kahn's algorithm*.

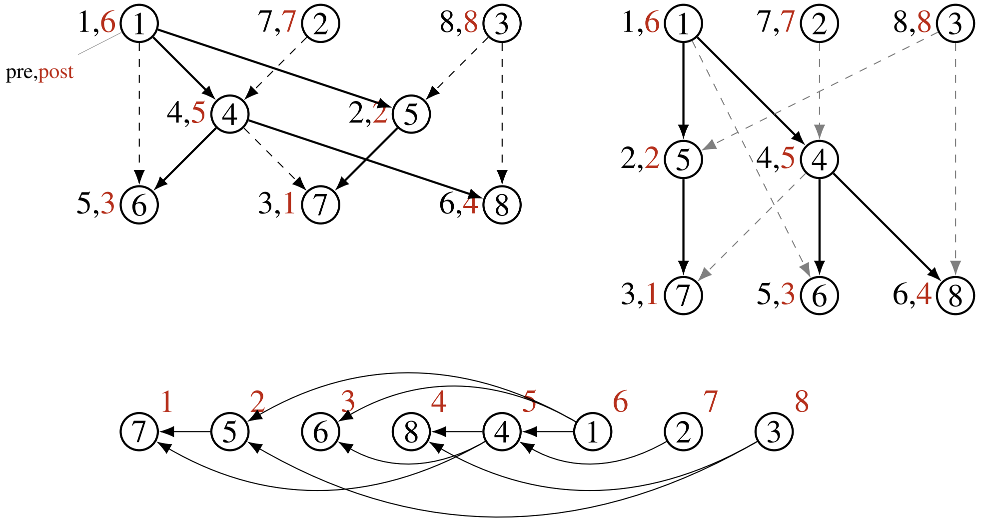
An efficient implementation does not really remove the edges. It iterates over the edges (in an adjacency lists representation) and for each node *counts* the *incoming* edges. We keep a list of sources. Choose a source, and when deleting it, look at its outgoing edges and decrease the count for the targets of the edges. When the count drops to 0 add the vertex to the list of sources. This algorithm considers every edge exactly twice, once when counting ingoing edges, a second time when each edge is 'removed' with its source: linear time.

Example 7.19. Count incoming edges in Example 7.17. The sources are kept in a list. Repeatedly remove a source and update counts and add new sources to the list.

1	2	3	4	5	6	7	8	sources	remove
0	0	0	2	2	2	2	2	1,2,3	1
	0	0	1	1	1	2	2	2,3	2
		0	0	1	1	2	2	4,3	4
		0		1	0	1	1	6,3	6
		0		1		1	1	3	3
				0		1	0	5,8	5
						0	0	7,8	7
							0	8	8

Another approach to find topological orderings is to use depth first-search. The ordering is found as the *reverse post-order* of the DFS spanning tree (or better, forest). When a node is popped in DFS no further descendants of the node are found, so all edges are to a node with lower DFS number.

Example 7.20. DFS applied to the graph of Example 7.17. For example, node 7 is the first node found to have no decendants, and is the last in the topological order.



References. ARTHUR B. KAHN. Topological sorting of large networks. Communications of the ACM **5** (1962) 558–562. doi:10.1145/368996.369025

See CLRS Chapter 22.4: Topological Sort

See DROZDEK Chapter 8.7: Topological Sort

Maximum flow ☒

Er zijn nog veel andere problemen die geformuleerd kunnen worden voor grafen. Een belangrijk voorbeeld is het *maximum flow problem*. Hier geven de gewichten de capaciteit van verbindigen weer, en de vraag is hoe we zoveel mogelijk kunnen laten stromen tussen twee gegeven knopen in de graaf, waarbij de capaciteiten gerespecteerd worden. Een flow heeft ook de eigenschap dat de totale flow (ingaand min uitgaand) in elke knoop nul is, behalve in de source en sink (net als Kirchhoff's law).

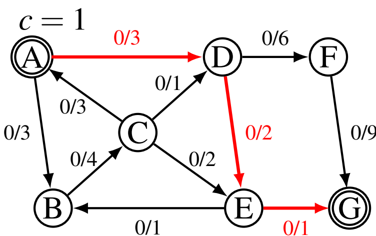
Een vroeg algoritme daarvoor is van Ford en Fulkerson. Het idee is dat herhaald een pad in de graaf gezocht wordt waarlangs nog extra capaciteit bestaat. Dit is een *augmenting path*. Een probleem is dat er langdurig kleine verbeteringen gevonden kunnen worden. Betere methoden om augmenting paths the bepalen zijn van Edmonds–Karp en van Dinitz, met complexiteit $O(|V||E|^2)$ en $O(|V|^2|E|)$, respectievelijk.

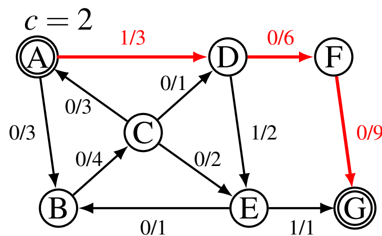
Hieronder een voorbeeld van het bepalen van de maximum flow.

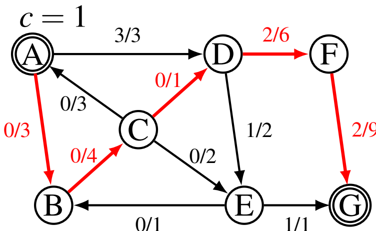
Example 7.21. Overgenomen van wikipedia. In de onderstaande gerichte graaf zoeken we een maximum flow van A naar G. Een tak label x/c geeft een verbinding mat capaciteit c weer, waarvan er in de gegeven graaf op dat moment x gebruikt worden.

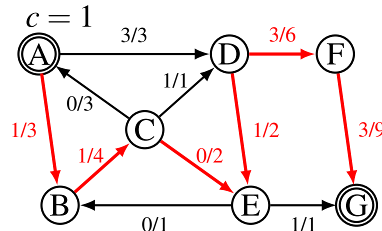
We starten met de nul flow. Begin met het augmenting path $A-D-E-G$. Da vrije capaciteit over dat pad is het minimum van $3, 2, 1$, dus 1. En we passen de flow in die richting aan. In de tweede stap is het augmenting path $A-D-F-G$ en de vrije capaciteit op dat pad is 2, namelijk het minimum van $3-1, 6, 9$.

In de vierde stap is het augmenting path $A-B-C-E-D-F-G$. Merk op dat een van de takken tegen de richting in gevolgd wordt. Als we daar de flow aanpassen wordt de flow in die tak kleiner gemaakt in plaats van groter.

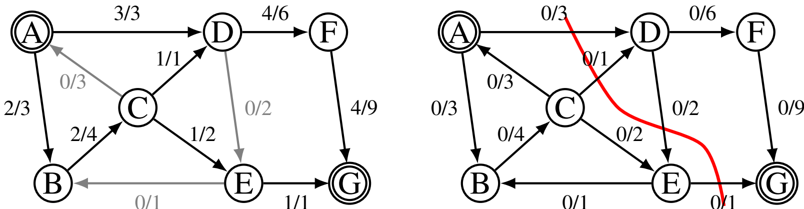








Het eindresultaat is als hieronder, met een totale flow van 5 tussen A en G.



Het is ook te bewijzen dat er geen grotere totale flow bestaat, met behulp dan de *max-flow-min-cut* stelling. Er is een *snede* in de graaf waarlangs in totaal niet meer dan 5 verplaatst kan worden (in de goede richting). Zie boven.

Travelling salesman problem ☒

Het Handelsreizigersprobleem is één van de bekendste optimalisatieproblemen. Gegeven een graaf met afstanden tussen elk paar steden, bepaal de kortste route die elke stad precies één keer aandoet en eindigt in de eerste stad. Er is geen efficient algoritme voor dit probleem bekend. Om precies te zijn is het probleem NP-compleet.

Een beslissingsprobleem is een algoritmische vraag waarvoor het antwoord alleen ja/nee is. De (beslissings-)problemen die in polynomiale tijd kunnen worden opgelost vormen de klasse P. Technisch gebruiken we daarvoor het rekenmodel van de Turing machine maar dat is hier niet essentieel. Een probleem behoort tot de klasse NP als een oplossing in polynomiale tijd kan worden geverifieerd. De afkorting staat voor *niet-deterministisch polynomiaal*. Het niet-determinisme slaat erop dat we een oplossing mogen gokken en die dan controleren.

Het is een belangrijk open probleem of P gelijk is aan NP. De moeilijkste problemen uit NP heten NP-compleet. Dat betekent dat als één NP-compleet probleem in polynomiale tijd kan worden opgelost, dat dat voor alle NP-problemen geldt, en daarmee P=NP.

Het Handelsreizigersprobleem kan als beslissingsprobleem worden omgezet: bestaat er een handelsroute die alle steden precies een keer aandoet met lengte maximaal ℓ'. In die versie is het Handelsreizigersprobleem NP-compleet. Het probleem is zelfs NP-compleet als de afstanden die gebruikt worden in de graaf de gewone afstanden van punten in het platte vlak zijn, het zogenaamde Euclidische TSP. Een gerelateerd NP-compleet probleem is het *Hamiltonpadprobleem*: bepaal of een ongerichte ongewogen graaf een gesloten pad heeft dat elke knoop precies één keer aandoet.

Opgaven

1.a) Wat is een topologische ordening op de knopen in een gerichte graaf $G = (V,E)$? Wees precies.

b) Hieronder staat een schematische recursieve functie voor *depth-first search*.

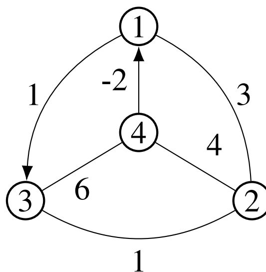
```
DFS (KnoopType x)
{   // aanname: knoop x nog niet bezocht
    bezoek x
    markeer x als bezocht
    for (elke knoop w bereikbaar vanuit x)
    {    if (w niet bezocht)
         {   DFS (w)
         }
    }
    // knoop x volledig afgehandeld:
    push x op stapel S
}
```

Beredeneer dat de knopen op de stapel `S` een topologische ordening van de acyclische graaf vormen nadat `DFS` op de achtereenvolgende knopen zonder inkomende takken is aangeroepen.

c) Gegeven is een adjacency-list representatie van een acyclische graaf G en een lijst met een topologische ordening voor G.

Beschrijf hoe het langste pad in G efficient kan worden bepaald. Pseudo-code is welkom. *Mrt 2016*

2. Onderstaande graaf heeft negatieve gewichten (en is slechts gedeeltelijk gericht).



a) Laat zien dat het algoritme van Dijkstra *niet* toepasbaar is op grafen met negatieve gewichten, door het algoritme toe te passen op bovenstaande graaf. Kies een geschikte beginknoop, voer Dijkstra uit en controleer de uitkomst.

b) Het algoritme van Floyd (*all pair distances*) is wel geschikt voor negatieve gewichten, zolang er geen negatieve kringen zijn in de graaf.

Geef het algoritme van Floyd. *Jan 2017*

c) Pas het toe op de gegeven graaf.

3. Bij de uitleg van Floyd staat: *Computations for* A^k *do not change the values at* $[k, j]$ *or* $[i, k]$ *in the matrix* A^{k-1} *so when implementing this algorithm we may in fact use only a single matrix.*

Controleer deze uitspraak.

4.a) Geeef het algoritme van Dijkstra, maar nu aangepast om de *bottleneck* waarden te berekenen, dus het pad waarlangs de minimale waarde van de pijlen maximaal is.

b) Idem, maar nu voor Floyd.

8 Hash Tables

Hash tables form an implementation for ADT's SET and DICTIONARY (or associative array) see Section 3.1, as an alternative to the several types of balanced trees, see Chapter 4 and Chapter 6.

The C++ STL has `set` and `map` as implementations for SET and DICTIONARY where the keys are stored in order (from small to large). Typically this is done using some kind of balanced tree. It also has `unordered_set` and `unordered_map` where this order is not respected. Typically this is done using hash-tables, the topic of this chapter.

	find av	*find* wc	*insert* av	*insert* wc	*delete* av	*delete* wc	order
unordered list		n		1		n	no
bin tree	$\log n$	n	$\log n$	n	$\log n$	n	yes
balanced		$\log n$		$\log n$		$\log n$	yes
hash table	1	n	1	n	1	n	no

av=average, wc=worst case

We like to store a number of keys from a huge domain (strings, numbers) in a table where necessarily the number of available addresses is much less than the number of possible keys. We investigate ways of directly computing the address where each key is stored with a simple function $h(K)$ based on the key K alone. As the number of possible keys is relatively large compared to the number of addresses we have to consider the case that two keys K, K' are destined in the same address, $h(K) = h(K')$. Such keys are called *synonyms*. In case actually two synonyms are supposed to be stored in the same table, we speak of a *collision*.

Here we consider three cases.

8.1 The keys are fixed and known in advance, so we can avoid collisions: **perfect hashing**.

8.2 After a collision the key is stored elsewhere: **open addressing**.

8.3 The table is designed to store various keys at the same address: **chained hashing**.

Hashing basics

- store keys of arbitrary size (usually large domains) in table of fixed size (usually small)
- store passes, checksums (MD5, CRC32)
- implement ADT unordered set: find, insertion and deletion in (avg) constant time
- *hash function* calculates position in table: $h(K) \bmod T$ (TableSize)

- *collision*: attempt to store key K when $h(K)$ is occupied

8.1 Perfect Hash Function

A *perfect hash function* is a mapping from the set K of keys into the address space of table T without collisions. This can only be attempted when the set of keys that is to be placed is known in advance. Examples of these are keywords of a specific programming language, or a dictionary of most occurring words in a natural language.

With such a perfect hash function h it is possible to test membership in K in constant time (plus the time needed to compare two keys), assuming $h(w)$ can be computed in constant time for any possible key w. We can use the observation that $w \in K$ iff $T(h(w))$ contains w.

An example. Cichelli computes the set of 36 keywords of the programming language Pascal as an example of the method. He proposes an hash function for strings of the form $h(w) = |w| + v(\text{first}(w)) + v(\text{last}(w))$, where v is a mapping from characters to integers, and first(w) and last(w) are the first and last letters of w. Suitable values can usually be found using clever backtracking.

Example 8.1. An example is the set of keywords in the programming language Pascal. The letters of the alphabet get their value according to the following table. Letters not mentioned get value 0.

a	b	c	f	g	h	i	l	m	n	p	r	s	t	u	v	w	y
11	15	1	15	3	15	13	15	15	13	15	14	6	6	14	10	6	13

Thus, for example $h(\text{case}) = 4+1+0 = 5$. Also $h(\text{label}) = 5+15+15 = 35$. We get the following table.

2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
do	end	else	case	downto	goto	to	otherwise	type	while	const	div	and	set	or	of	mod	file	record	packed	not	then	procedure	with	repeat	var	in	array	if	nil	for	begin	until	label	function	program

◇

References. R.J. Cichelli. Minimal Perfect Hash Functions Made Simple, Communications of the ACM **23** 17–19 (January 1980)

8.2 Open Addressing

With this method we have an array `T[M]` of M possible slots, each which can hold just a single key (or more precisely for Dictionaries a single key-value pair). The hash function h returns the intended address of the key. Collisions are resolved by looking for an alternate unused location in the array. The process of checking at various locations for an empty slot is called *probing*. The probing sequence for each key K is fixed, but depends on the specific hash method and its parameters. We discuss several variants: linear, quadratic, and double hashing.

The operation ISELEMENT checks whether a key is present, and is implemented by following the probing sequence for the key. It is successful if the key is found, it is unsuccessful when we happen to hit an empty slot (we can conclude the key is not present as it would have been stored at that position).

Both the INSERT (add a key, or key-value pair) and the RETRIEVE (for a Dictionary, find the value for a give key) operations follow the *same path* over the address space. INSERT is successful if an empty slot is found, and the key is stored (and needs special attention if we find the key instead, we do not allow duplicate keys); RETRIEVE on the contrary fails if we discover an empty slot.

There are several ways to choose the probe sequence $h(K,0) = h(K), h(K,1), h(K,2), \ldots$, i.e, the sequence of addresses that are checked consecutively. Note we theoretically wish to make this sequence a permutation of the addresses in the table, as we want to visit each address exactly once. In practise however we sincerely hope that the number of visits is bounded by a constant to fulfill the promise of hashing.

(1) With **linear probing** we search for an available position (or the key itself) in the positions immediately before the original address: for key K after $i \geq 0$ steps we check address $h(K,i) = h(K) - i \pmod{M}$. Recall that M is the table size. The probe sequence wraps to the last element when we failed at the 0'th position, because of the modulo. Mind the negative sign.

Unfortunately linear probing is sensitive to *primary clustering*: segments of occupied spots are filling the table. Note that larger segments have a larger probability to grow: each collision to one of the elements in the cluster means that the new key will end up in the free position just before the cluster.

This means we have to be extremely careful that the input distribution of keys will not form clusters using the chosen address function. As an example: variable sequences like help1, help2, help3 are common, and they should not be placed at consecutive places in the hash table as that would already start a cluster.

Example 8.2. We use a table $T[11]$ to store integers and hash-function $h(K) = K \pmod{11}$ using linear hashing. We add the keys

60(5), 29(7), 74(8), 3(5), 19(8), 23(1), 40(7)

(in that order) to an empty table. Here the values in parentheses are the computed hash values: for example $60 = 5 \cdot 11 + 5 \equiv 5 \pmod{11}$.

The keys 60, 29, and 74 are stored on their intended addresses, the address determined by the hash-function. Then we have a collision of key 38 at slot 5. We try one slot to the left, and place 38 at 4.

Key 19 again is in collision at 8, and is placed after two more probes at slot 6.

Key 40 cannot be placed at address 7, and we need several steps to find a free slot at 3.

0	1	2	3	4	5	6	7	8	9	10
				38	60		29	74		
					38					
	23			38	60	19	29	74		
								19		
	23		40	38	60	19	29	74		
							40			

Sometimes it might be a good idea to change the step size for linear hashing from 1 to another constant c. That can be done provided that c has no divisors in common with the table size M.

It might seem that large c is profitable as it will jump over clusters. The problem is that with $c \neq 1$ the clusters are not formed in consecutive cells.

Example 8.3. Starting with $h(K,i) = (K \bmod 10) - 3i$ we have the following table. One needs many steps to find 65. (Start at address $h(65) = h(65,0) = 65 \bmod 10 = 5$. This is occupied by 55. Continue three steps to the left at $5-3=2$, which again is occupied by another key. Etc.)

0	1	2	3	4	5	6	7	8	9
65		32	43		55	72			19

The reason is that the table is one big cluster, visible when we show the neighbours relative to the step size 3.

0	3	6	9	2	5	8	1	4	7
65	43	72	19	32	55				

This is actually equivalent to hashing according to $h'(K,i) = (7K \bmod 10) - i$.

0	1	2	3	4	5	6	7	8	9
65	43	72	19	32	55				

In this last version of the table, the cluster that does not appear to be present in the first version, is plainly visible.

◇

---

(2) Ideally we would like to use a random permutation $r_0 = 0, r_1, r_2, \ldots$ of all addresses to avoid primary clustering and visit the slots as $h(K,i) = h(K) - r_i \pmod{M}$. It is however hard to devise a random sequence for every table size. A solution which is close to randomness is **quadratic probing**. Here the squares are used to substitute random numbers: $0, \pm 1^2, \pm 2^2, \pm 3^2, \pm 4^2, \ldots$. Technically it can only be guaranteed this forms a permutation of the address space if M is a prime number, which is $3 \pmod 4$.

This avoids primary clustering, but still all keys at a given address follow the same path, a phenomenon called *secondary clustering*.

Example 8.4. We use a table $T[11]$ to store integers and hash-function $h(K) = K \pmod{11}$ using quadratic hashing. We add the keys 60(5), 29(7), 74(8), 38(5), 19(8), 23(1), 40(7) (in that order) to an empty table. Again in parentheses are the computed hash values.

The steps we take, each time starting from the initial address (not the last address) are $+1, -1, +4, -4, +9, -9, \ldots,$

The keys 60, 29, and 74 are stored at free addresses immediately as we have no synonyms. Then we have a collision of key 38 at slot 5. The slot $5 - 1 = 4$ is free so we store 38 there.

Key 19 again is in collision at 8, and we try at $8 - 1 = 7$, and then $8 + 1 = 9$ is free. No problems for 23.

Key 40 cannot be placed at address 7, but we find a slot at $7 - 1 = 6$.

0	1	2	3	4	5	6	7	8	9	10
				38	60		29	74		
	23			38	60		29	74	19	
	23			38	60	40	29	74	19	

◇

---

(3) **Double hashing.** In this method we use a second hash function to determine the step size. This *probe function* $p(K)$ must be 'independent' from $h(K)$ so that keys hashed to the same address do not follow predictable paths.

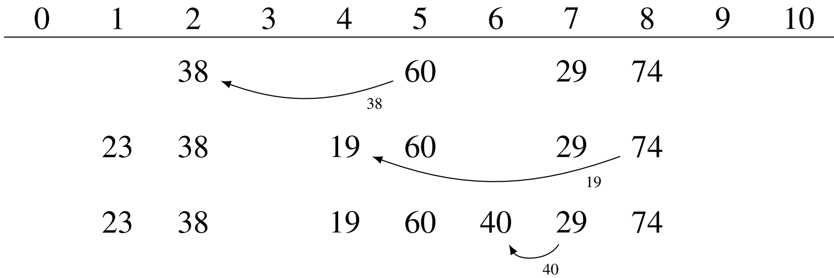
The step sequence equals $h(K,i) = h(K) - i \cdot p(K) \pmod{M}$. As mentioned with linear probing, the values of p must be relatively prime to the table size M.

Example 8.5. We repeat Example 8.2 now using the probe function $p(K) = (K \bmod 4) + 1$. This leads to five different step sizes for each location. Note the table size is a prime, so the step sizes do not have a divisor with the table size.

The keys 60, 29, 74, 38, 19, 23 and 40 have the following hash values $h(K)$ and $p(K)$:

Key	60	29	74	38	19	23	40	K
Address	5	7	8	5	8	1	7	$h(K) = K \bmod 11$
Step size	1	2	3	3	4	4	1	$p(K) = (K \bmod 4) + 1$

The table is filled in a similar way as before, except we now have different sized steps when looking for a free spot.



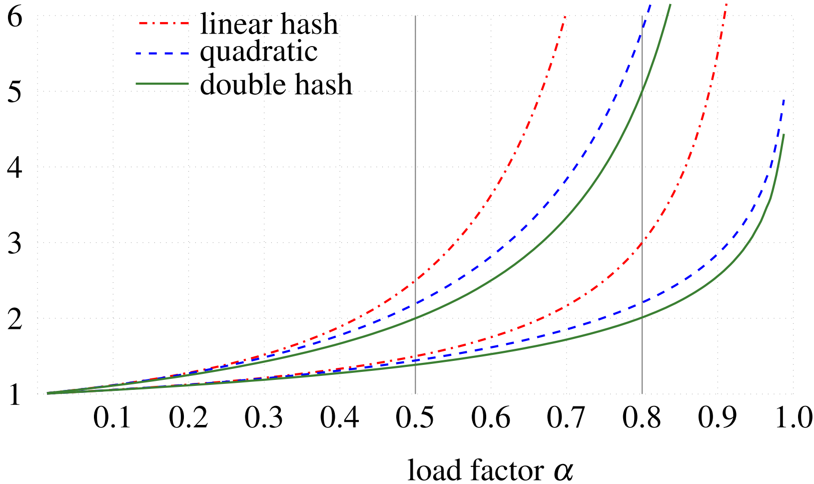
Observe that we have a cluster of five keys stored in a contiguous interval. This is however not always a cluster from the viewpoint of each key. When looking for key 30 (hash value 8, step 3) we first look at 8, then $8 - 3 = 5$, 2 and finally 10. The last slot is empty and shows key 30 is not in the table.

$\diamond$

Expected number of steps. According to Knuth we have the following expected number of steps for three hashing methods above, and both unsuccessful search (or adding) and successful search. The variable α represents the *load factor*, the fraction of occupied slots in the table. It determines the expected number of probes to find or store a key in the table.

We give the function values where $\alpha = 0.5 - 0.8$. Not too bad at 50% it seems.

	find / successful		add / unsuccessful	
linear	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	1.5–3	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$	2.5–13
secondary	$1 + \ln(\frac{1}{1-\alpha}) - \frac{\alpha}{2}$	1.4–2.2	$\frac{1}{1-\alpha} - \alpha + \ln(\frac{1}{1-\alpha})$	2.2–5.8
double	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$	1.4–2.0	$\frac{1}{1-\alpha}$	2–5



Note that for open addressing we have to choose a table size that is appropriate for the number of expected keys! If the load factor α reaches 1 the table starts to behave like a unordered list.

What to remember of this?

- Finding is faster than adding (because we stop early along the path).
- Double is faster than secondary, which is faster than linear.
- Avoid large load factors.
- (But do not memorize the exact functions given here...)

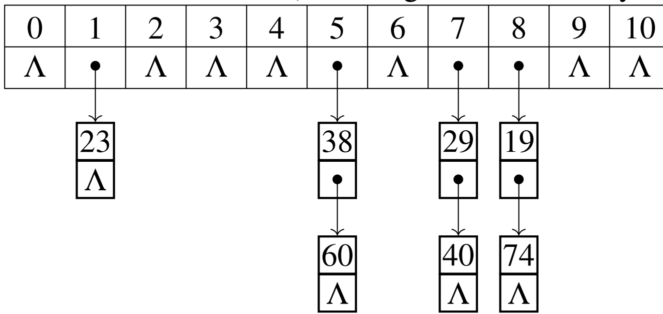
Deletion. Removing a key to make room for another one is a bad decision in open hash tables. If the key K is on the search path for another key K' between its home address and the slot where it was placed, we will conclude the key K' is not present after removal of K.

Usually a wise solution is to use *lazy deletion*, by marking the slot as 'deleted'. When searching for a free slot we can use it as if empty, when searching for a given key we pretend it is in use (by another key).

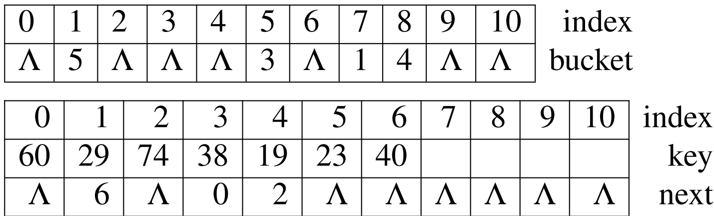
8.3 Chaining

Instead of storing keys directly in the table, here we keep a linked list of keys for each table element. Such a list usually is called a *bucket*. That means that collision is solved by adding the new elements to the bucket at the hash address. We can choose to just add the new elements at the beginning of the list or to keep the list ordered (which is slightly faster when searching elements not in the table). Deletion is not a problem in chained hashing: just remove the key from the proper bucket.

Example 8.6. Consider the table $T[11]$ of size 11 and the address function $h(K) = K \bmod 11$. After adding the keys 60(5), 29(7), 74(8), 38(5), 19(8), 23(1) and 40(7) the table looks as follows (assuming we store the keys in ascending order).



Recall that a list does not automatically implies a pointer implementation. One can also use an array to store the buckets. We then get the following picture. (Here Λ indicates an end-of-list value.)



0	1	2	3	4	5	6	7	8	9	10	index
Λ	5	Λ	Λ	Λ	3	Λ	1	4	Λ	Λ	bucket

0	1	2	3	4	5	6	7	8	9	10	index
60	29	74	38	19	23	40					key
Λ	6	Λ	0	2	Λ	Λ	Λ	Λ	Λ	Λ	next

8.4 Choosing a hash function

A good hash function $h(K)$ should

- be fast to compute,
- evenly distribute the keys over the table, and
- depend on all ‘distinctive bits‘ of the key.

One of the main concerns is to counteract any expected ‘behaviour’ of the keys. If we hash bank account numbers, several digits might be predictable in case the number includes a code for the bank. If we hash student accounts, the initial digits might represent year or discipline. If we hash variables we might expect sequences like `x1`, `x2`, . . . , that should not be hashed on consecutive addresses.

We mention some techniques. We assume the key is in fact a (possibly large) number. Any string can be read as number.

Division. Return the key modulo the table size (as we have done in several examples). When the table size is a power of two, then division amounts to choosing the last bits of the key, see Extraction. This seems to have good results when the table size is prime (or without small divisors).

Folding. The key is chopped into parts, which are then added/xor-ed. The result is then again returned modulo the table size.

Mid-squaring. Take the square of the key, and select the middle bits as result (assuming the table size is a power of two). In several contexts squaring behaves rather randomly. Take care when the key contains many zeros at its start or end.

Extraction. The address is computed based on selected bits of the key. These bits should be as random as possible, and preferably not be somewhat predictable like year or check-bits.

Example 8.7. Developing a good hash function is hard work. In professional context it should be fast and very thoroughly mix the bits of a key. The resulting code matches the architecture of the processor to achieve speed. Just as illustration we give MurmurHash, on which google’s CityHash is based, in Slide 22.

The GNU C++-compiler defines `hash_bytes`, “a primitive used for defining hash functions. Based on public domain MurmurHashUnaligned2”. (stackoverflow: What is the default hash function used in C++ `std::unordered_map`?)

Slide 22 ☒ MurmurHash as presented on Wikipedia

MurmurHash

```
Murmur3_32(key, len, seed)
    // integer arithmetic with unsigned 32 bit integers.
    c1 := 0xcc9e2d51
    c2 := 0x1b873593
    r1 := 15
    r2 := 13
    m := 5
    n := 0xe6546b64

    hash := seed
    for each fourByteChunk of key
        k := fourByteChunk
        k := k * c1
        k := (k « r1) OR (k » (32-r1))
        k := k * c2
        hash := hash XOR k
        hash := (hash « r2) OR (hash » (32-r2))
        hash := hash * m + n

    with any remainingBytesInKey
        // (also do Endian swapping on big-endian machines.)
        remainingBytes := remainingBytesInKey * c1
        remainingBytes := (remainingBytes « r1) OR (remainingBytes » (32 - r1))
        remainingBytes := remainingBytes * c2
        hash := hash XOR remainingBytes

    hash := hash XOR len
    hash := hash XOR (hash » 16)
    hash := hash * 0x85ebca6b
    hash := hash XOR (hash » 13)
    hash := hash * 0xc2b2ae35
    hash := hash XOR (hash » 16)
```

For double hashing we also choose a second hash function $p(K)$ (the probe function, which determines the step size). As discussed earlier that function should have no divisors in common with the table size (to ensure we visit all addresses) and it should be independent of the address function (to avoid secondary clustering).

Note that passwords are usually stored using hash functions. Examples of these are MD5 (Message Digest Algorithm 5) or SHA-1 (Secure Hash Algorithm). To check the given password it is hashed and compared with the stored value. Needless to say such hashes must be very unpredictable.

Wikipedia gives the example MD5("Pa's wijze lynx bezag vroom het fikse aquaduct") = `b06c0444f37249a0a8f748d3b823ef2a`, while MD5("Ma's wijze lynx bezag vroom het fikse aquaduct") = `de1c058b9a0d069dc93917eefd61f510` .

References. See DROZDEK Chapter 10: Hashing.

See LEVITIN Section 7.3: Hashing.

See CLRS Chapter 11: Hash Tables.

Opgave

1. Beschouw hashen in een zgn. ‘open’ hashtabel met twee hash-functies h en p. Het $i+1$-ste bezochte adres $h(K,i)$ is zoals gewoonlijk $h(K) - i \cdot p(K)$ (modulo M).

a) Welke twee punten zijn belangrijk bij de keuze van de adresfunctie h ?

Waarop moeten we letten bij de keuze van de stapfunctie (*probe function*) p?

b) Welke twee soorten clustering onderscheiden we bij hashen met open adressering? Geef een korte beschrijving.

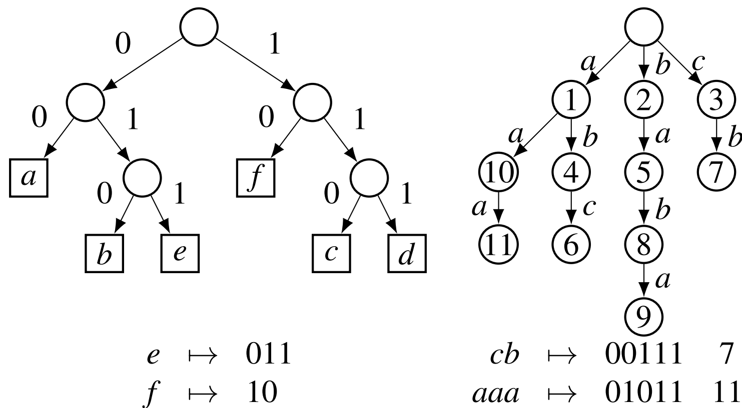
c) In een tabel `T[0..10]`, dus $M = 11$, worden achtereenvolgens de sleutels 10, 22, 31, 4, 15, 28, 17, 88, 59 geplaatst, met adresfunctie $h(K) = K \bmod 11$ en lineair hashen (stapgrootte 1).
Laat zien welke tabel ontstaat, maar geef op een overzichtelijke manier ook alle plekken waar de sleutels geprobeerd worden.

d) Idem, nu met een stapfunctie $p(K) = 1 + (K \bmod 10)$. *Jan 2015*

9 Data Compression

Data compression can save both time (transmission of data) and space (storage). We study *lossless* data compression, so the coded data can be decoded back into the data that was used to generate it. We present two well-known techniques to code a text in binary: *Huffman coding* and *Ziv-Lempel-Welch*. The first method codes each single letter by a fixed binary string such that frequently occurring letters get short codes. Ziv-Lempel-Welch tries to find codes for sequences of substrings that occur more often in the input.

Example 9.1. (Left: Huffman coding) The binary tree representation of a prefix code. The coded symbols are represented as leaves. The binary code follows the path from node to leaf. (Right: Ziv-Lempel-Welsh) The code is given as a trie, with letters along edges. The numders inside the nodes represent binary codes of a fixed length. Here, e.g., number 7 is the five letter code 00111.



$e \mapsto 011$ $\qquad$ $cb \mapsto 00111 \quad 7$

$f \mapsto 10$ $\qquad$ $aaa \mapsto 01011 \quad 11$

◇

9.1 Huffman Coding

In the ASCII code every 'letter' is encoded as a bit string of 8 bits. In Morse code every letter is encoded by a variable length sequence of dots ('dit') and dashes ('dah'): E, the most common letter in English, is represented by a single dot. Decoding ASCII is easy. Decoding Morse is only possible because besides dots and dashes one also uses 'spaces' between letters and words. These spaces are important because Morse is not a prefix code (see below). Without spacing letter I ('dit dit') has the same code as two I's.

Informally the *binary compression* problem we solve is as follows. We start with a message over some alphabet Σ. We know the alphabet and the frequency of the symbols in the message in advance. We want to replace the symbols in the

message by a binary code (each letter in Σ gets a fixed code, but the length of the code may vary between letters) such that the encoded binary message is of minimal length. Of course the compressed binary message must be easily decodable, where the binary code for each letter is known at the time of decoding.

Features:

- variable length code for single letters
  $a_1,\ldots,a_n \in \Sigma \mapsto w_1,\ldots,w_n \in \{0,1\}^*$
- based on character frequencies (known in advance) $f_1,\ldots,f_n$
- optimal expected code length (for prefix code) $\sum_{i=1}^{n} f_i \cdot |w_i|$
- code has to be known by decoder

Given a set of letters $\Sigma = \{a_1,\ldots,a_n\}$ and their (relative) frequencies $f_1,\ldots,f_n$, we want to find a variable length (binary) code $w_1,\ldots,w_n$ of strings over $\{0,1\}^*$ that is easily decodable and has minimal expected length.

The code of a message $a_{i_1}a_{i_2}\ldots a_{i_m}$ equals $w_{i_1}w_{i_2}\ldots w_{i_m}$, i.e., is obtained by a string morphism. It is decodable if no two messages generate the same code, i.e., the homomorphism is injective. The expected length of a message, given the letter frequencies, equals $\sum_{i=1}^{n} f_i \cdot |w_i|$.

Example 9.2. The code $a \mapsto 10, b \mapsto 1, c \mapsto 00$ is uniquely decodable. However we have to look ahead. Each sequence in 10^* has to be completely read before it can be decoded. If the number of 0's is even then 10^{2n} is decoded as bc^n, if the number is odd then 10^{2n+1} is decoded as ac^n.

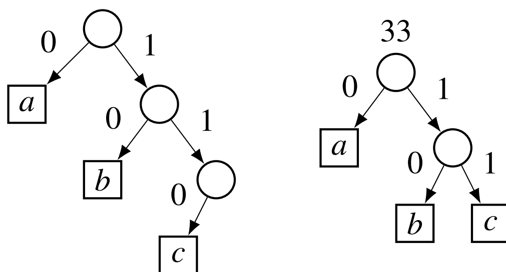
The code $a \mapsto 0, b \mapsto 01, c \mapsto 011$ is uniquely decodable: each letter can be recognized by the initial 0. Still we have a (small) lookahead: a letter is only decoded when the first 0 bit of the next letter is seen.

◇

In order to be able to decode messages without lookahead we restrict ourselves to *prefix codes*. This means no codeword w_i is a prefix of another codeword w_j. These codes can be represented as binary trees, with the letters of the input alphabet at the leafs. Internal nodes do not represent codewords, because of the prefix property. The expected length of a message equals the *weighted external path length* in the tree: the sum over all leaves of the depth of the leave times its frequency. We may assume the tree is *full* (i.e., each internal node has two children), as we can find a shorter code by omitting the nodes with a single child.

Decoding a prefix code is trivial. Start at the root and follow the edges until a leaf is reached; output the letter at the left; return to the root, and continue.

Example 9.3. Reversing the codewords in the second example above we get $a \mapsto 0, b \mapsto 10, c \mapsto 110$, which is a prefix code with the tree representation below (left). Obviously it is not optimal (the tree is not full) as the code for c can be changed into 11 without losing the prefix property (right). String 010111110001011 is decoded via $0 \cdot 10 \cdot 11 \cdot 11 \cdot 10 \cdot 0 \cdot 0 \cdot 10 \cdot 11$, i.e., as *abccbaabc*.



◇

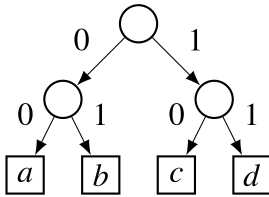
Huffman's solution for constructing an optimal code tree is an elegant greedy *bottom-up* algorithm.

Huffman

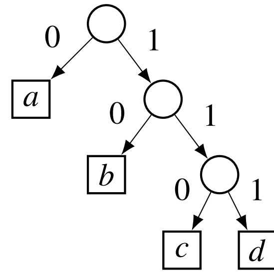
```
// initialize:
for each input letter a tree with that letter,
    and its frequency
repeat
    take two trees of minimal frequencies
    join these as children in a new tree,
        with combined frequency
until one tree left
```

The algorithm is not deterministic. We may swap left and right branches without changing the efficiency of the code (only the corresponding bits 0 and 1 are swapped for the code words in the tree). A simple example shows we can have two 'structurally different' code trees for a given alphabet (with frequencies).

Consider the alphabet $\{a,b,c,d\}$ with respective frequencies 10,5,5, and 5. The following two optimal trees both have expected code length 50: $2 \cdot 10 + 2 \cdot 5 + 2 \cdot 5 + 2 \cdot 5$ resp. $1 \cdot 10 + 2 \cdot 5 + 3 \cdot 5 + 3 \cdot 5$. Both of them can be obtained from the algorithm.



$2 \cdot 10 + 2 \cdot 5 + 2 \cdot 5 + 2 \cdot 5 = 50$



$1 \cdot 10 + 2 \cdot 5 + 3 \cdot 5 + 3 \cdot 5 = 50$

Example 9.4. We construct a Huffman code for the alphabet a, b, c, d, e, f, the letters of which have relative frequencies 18, 8, 12, 13, 7, and 21, respectively.

With the given (relative) frequencies the expected code length equals $18 \cdot 2 + 8 \cdot 3 + 12 \cdot 3 + 13 \cdot 3 + 7 \cdot 3 + 21 \cdot 2 = 198$.

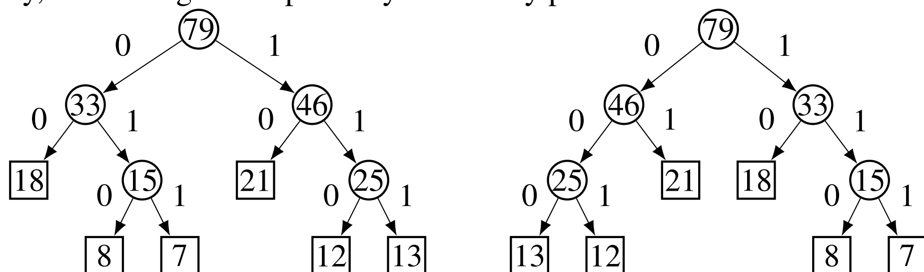
$\diamond$

Complexity. Clearly the number of steps taken by the algorithm is linear, as we lose one tree each step. However, the complexity is slightly more than that, $\mathcal{O}(n \lg n)$, as we have to take the two least frequencies in each step. This can be done in two ways. First, we can keep the trees in a priority queue. We then have a

linear number of insertions and deletions, each of complexity $O(\lg n)$. As a second possibility we can sort the initial frequencies and keep them in a list. The trees that are joined are kept in a separate list. The second list can be kept in sorted order, as each new tree has a larger frequency over time. In each step we choose the minimal frequency from the two (sorted) lists. This has the same complexity as sorting, which is $\mathcal{O}(n \lg n)$.

Dynamic frequencies. ☒ If we add the frequencies of each subtree (which are just the total frequencies of the leaves in the subtree), then we obtain the expected path length (see the left tree in the example below, $79+33+46+15+25=198$). This is a consequence of the fact that the frequency of each leaf is counted in each of its predecessors.

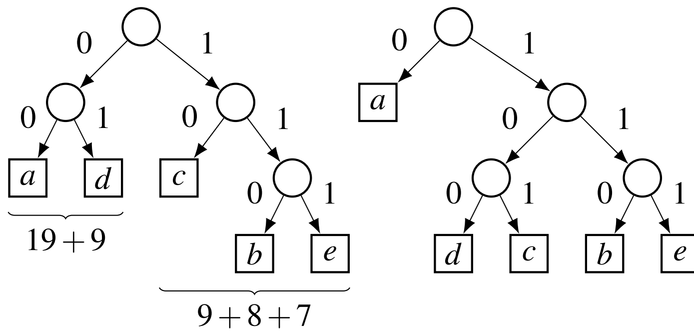
We can rearrange the Huffman tree, swapping left and right branches where necessary, to obtain a tree where the frequencies are ordered in breadth-first order. In fact, this is a characteristic of a Huffman tree. This observation is the basis of the *dynamic Huffman algorithm*, where the frequencies are not known in advance and the tree is restructured while counting. The restructuring process is rather costly, and the algorithm probably is not very practical.



Remarks. ☒ Despite its simplicity, Huffman's algorithm was a major breakthrough. The previous best method was the Shannon–Fano coding which basically worked in a top-down manner, repeatedly dividing the frequencies into two subsets.

The solution found by Huffman's algorithm does not try to keep the codes in *alphabetical order*. If that is required and one is looking for optimal alphabetic binary trees, there is an algorithm by *Hu-Tucker*. This works in $\mathcal{O}(n \lg n)$ time, just like Huffman.

Example 9.5. With frequencies 19, 8, 9, 9, 7 the Shannon–Fano coding gives a total expected length $2 \cdot (19+9+9)+3 \cdot (8+7) = 119$, whereas Huffman yields $1 \cdot 19 + 3 \cdot (9+9+8+7) = 118$. Tiny difference, but enough to show that Shannon–Fano is not optimal.



◇

References. D. HUFFMAN. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE **40** (1952) 1098–1101. doi:10.1109/JRPROC.1952.273898

T.C. HU, A.C. TUCKER. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. SIAM Journal on Applied Mathematics **21** (1971) 514–532. doi:10.1137/0121

See DROZDEK Chapter 11.2: Huffman Coding.

See LEVITIN Section 9.4: Huffman Trees.

9.2 Ziv-Lempel-Welch

Features:

- fixed length code for repeating patterns in input
  $x_1, \ldots, x_n \in \Sigma^* \mapsto w_1, \ldots, w_n \in \{0,1\}^k$
- strings x_i learned while reading input
- code is also learned by decoder and does not have to be transmitted

Before we start the coding and decoding algorithms, we have to agree on the input-alphabet, and on the number of bits used to send each code word. We start by setting the codes for each single letter in the alphabet. The number of bits is important as the decoder has to read that number of bits to translate the code back into its original. The coding algorithm keeps a 'dictionary' of input segments that have received a code. It scans repeatedly over the input to find the next segment that can be coded. When the longest segment has been discovered, the code is output. At the same time a new code is entered into the dictionary. The current segment plus the next letter from the input get the next available bit string. In this way the encoding algorithm 'learns' from what it reads.

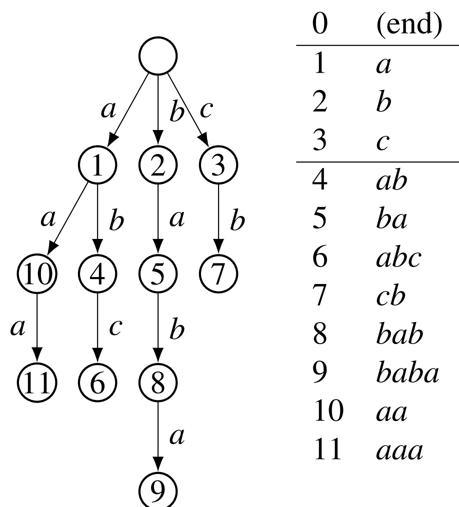
```
                    ZLW compression
initialize dictionary with single characters

w = "";
while ( not end of input )
do  read character c
    if w+c exists in the dictionary
    then w = w+c;
    else
        add w+c to the dictionary;
        output the code for w;
        w = c;
    fi
od
output the code for w
```

In the toy example below we include and 'end-of-string' symbol (to signal the end of the message). This is a technicality, and there might be other ways to recognize the end of a sequence of bits.

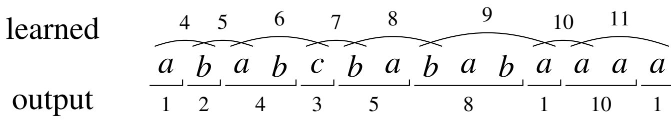
The dictionary can be represented as a *trie*, where nodes have (possible) children for each input letter.

Example 9.6. Input string *abab cbab abaa aa* over the alphabet $\{a,b,c\}$. We do not fix the number of output bits, but use numbers to indicate output. For example 3 denotes 0011 or 00011 for 4 or 5 bits, respectively. In this example we assume to have at least five bits (to accommodate 12 code words).

w	c	dict?	out	new code
	a	✓		
a	b	×	1	$4 \mapsto ab$
b	a	×	2	$5 \mapsto ba$
a	b	✓		
ab	c	×	4	$6 \mapsto abc$
c	b	×	3	$7 \mapsto cb$
b	a	✓		
ba	b	×	5	$8 \mapsto bab$
b	a	✓		
ba	b	✓		
bab	a	×	8	$9 \mapsto baba$
a	a	×	1	$10 \mapsto aa$
a	a	✓		
aa	a	×	10	$11 \mapsto aaa$
a	$\perp$		1	



0	(end)
1	a
2	b
3	c
4	ab
5	ba
6	abc
7	cb
8	bab
9	$baba$
10	aa
11	aaa



If the dictionary becomes full during the algorithm, we have to agree on the proper action. There are several conventions.

1. Continue "as is", no further codes are entered into the dictionary.
2. Add one bit to code space, and continue.
3. Many of the leaves in the code tree represent codes that are no longer used as they were not extended. Reclaim this space by pruning the tree, cutting the last letters from each branch and reusing the codes.

Decoding. The reverse operation is performed by virtually the same algorithm, adding new codes into the dictionary. If binary code n is read we output its original w and we should add new code for wb, where b is the single letter look-ahead in the coding algorithm. However b is not known until the next binary code n' has been processed. Thus, decoding lags behind by one character.

This can have an unfortunate effect. It may happen that the next code n' is not yet in the dictionary, as it actually is the one that should have been added the previous step. This can only happen if the (unknown) lookahead letter b actually is the first letter of the new code wb, i.e., the first letter of the last decoded word w. With this knowledge we can close the gap.

ZLW decompression

```
initialize dictionary with single characters
read first code in variable prev and output str(prev)
while ( not end of input )
do  read w
    if w exists in the dictionary
    then output str(w)
        add to dict: str(prev) + firstchar(str(w))
    else
        // special case
        output str(prev) + firstchar(str(prev))
        add to dict: str(prev) + firstchar(str(prev))
    fi
    prev = w
od
output the code for w
```

Example 9.7. We decode the output we found in the previous example, i.e., the bit sequence corresponding to the numbers $1, 2, 4, 3, 5, 8, 1, 10, 1$.

As before we start with the dictionary $1 \mapsto a$, $2 \mapsto b$, and $3 \mapsto c$.

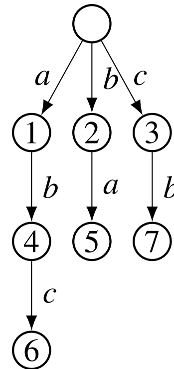
First code 1 is decoded to a using the initial dictionary. We cannot learn the new code, as we need to know the next symbol. Second code 2 is decoded into b. Now we nknow the next symbol and learn $4 \mapsto ab$.

This continues until we have to decode 8. Yet the code for 8 is not in our dictionary: it is the next one that should have been learned. For the new code we know $8 \mapsto ba?$ because we extend the previous code. Since the first letter read is used for the new code, and we now know the letter is b, we can complete the code $8 \mapsto bab$.

learned: 4 5 6 7 8
decoded: a b a b c b a ? ? ?
input: 1 2 4 3 5 8 1 10 1

We complete the table, as below. Each step we add the new code for the string decoded in the last step plus the first letter of the present string. The special case is marked by '!'. The tree to the right is the code-tree at the moment we read code 8.

code		means	add
1		a	
2		b	$4 \mapsto ab$
4		ab	$5 \mapsto ba$
3		c	$6 \mapsto abc$
5		ba	$7 \mapsto cb$
8	!	bab	$8 \mapsto bab$
1		a	$9 \mapsto baba$
10	!	aa	$10 \mapsto aa$
1		a	$11 \mapsto aaa$



◇

Remarks. The ZLW compression is part of the GIF file format ("Graphics Interchange Format"), and was patented. In the 1990s the owners of this patent started to ask licence fees for developers that incorporated the algorithm in their tools. As a consequence the conversion to GIF was removed from many distributions, and a new file format PNG was developed. Around 2003 the patent on LZW expired.

References. T. WELCH. A Technique for High-Performance Data Compression. Computer **17** 8–19 (1984). doi:10.1109/MC.1984.1659158

J. ZIV, A. LEMPEL. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** 530–536 (1978). doi:10.1109/TIT.1978.1055934 (The original paper has “Ziv-Lempel” but now “Lempel-Ziv” seems common.)

See DROZDEK Chapter 11.4: Ziv-Lempel Code.

9.3 Burrows-Wheeler ☒

Start with a string, like *MISSISSIPPI*. For clarity of the presentation we add an end marker, like $ here. List all rotations of the string (moving the last letter to front repeatedly). Then write the rotated strings in alphabetical order, where the marker comes last in the alphabet.

$MISSISSIPPI \rightsquigarrow S^2MP\$PIS^2I^3$

1	M I S S I S S I P P I -	8	I P P I - M I S S I S S	S_1	I_1
2	I S S I S S I P P I - M	5	I S S I P P I - M I S S	S_2	I_2
3	S S I S S I P P I - M I	2	I S S I S S I P P I - M	M_1	I_3
4	S I S S I P P I - M I S	11	I - M I S S I S S I P P	P_1	I_4
5	I S S I P P I - M I S S	1	M I S S I S S I P P I -	$	M_1
6	S S I P P I - M I S S I	10	P I - M I S S I S S I P	P_2	P_1
7	S I P P I - M I S S I S	9	P P I - M I S S I S S I	I_1	P_2
8	I P P I - M I S S I S S	7	S I P P I - M I S S I S	S_3	S_1
9	P P I - M I S S I S S I	4	S I S S I P P I - M I S	S_4	S_2
10	P I - M I S S I S S I P	3	S S I S S I P P I - M I	I_2	S_3
11	I - M I S S I S S I P P	6	S S I P P I - M I S S I	I_3	S_4
12	- M I S S I S S I P P I	12	- M I S S I S S I P P I	I_4	$

Read the last column, which is a permutation of the original string. The result *SSMP$PISSIII* is the *Burrows-Wheeler transform* of the original string. It is the basis of compression techniques like bzip2 as in general there will be segments of the same letter repeated: $S^2MP\$PIS^2I^3$. To this we can apply run length encoding.

It is rather surprising that we can decode the transform, and obtain the original string. This is based on the observation that the copies of a single letter are in the same order if we look at the in either the first or the last column of the sorted array: Aw is before Au iff wA is before uA. Also note that the letter in the last column is followed by the letter in the first column.

Next to the last column we write the first column. The first column can be found from the first column just by sorting the letters. (And even this sorting is efficient as we just can count the number of each symbol in the columns: 4 copies of I, etc.) We have just seen that the order of each copy of a letter is the same in those two columns. In order to decode, we number the occurrences of each letter for clarity. Now we can distinguish and identify each letter. Thus M_1 is the first

letter as it 'follows' \$. Next character is I_3 as it follows M_1. Following I_3 is S_4, etc. Finally we get $M_1 I_3 S_4 S_2 I_2 S_3 S_1 I_1 P_2 P_1 I_4$.

Decoding BW with the counting trick is quite efficient. Efficiently encoding BW is more difficult. It uses techniques that are similar to the suffix array, which is also mentioned in Chapter 10.4.

References. M. BURROWS AND D.J. WHEELER. A Block-sorting Lossless Data Compression Algorithm. Digital Systems Research Center Research Report **124** (1994). HP Labs Technical Reports "*The algorithm described here was discovered by one of the authors (Wheeler) in 1983 while he was working at AT&T Bell Laboratories, though it has not previously been published.*"

Opgave

1.a) Welk probleem lost het algoritme van Huffman op?

Als er n sleutels gegeven zijn, wat is dan de complexiteit? Geef enige toelichting.

b) Pas Huffman toe op symbolen $a, b, \ldots, f$ met respectievelijke frequenties $10, 7, 2, 8, 11, 5$.

c) Er bestaat een *dynamische* versie van Huffmans algoritme, waar de frequenties kunnen veranderen. Daartoe wordt gebruik gemaakt van een bijzondere eigenschap van de bomen die geconstrueerd worden: de waardes in de knopen kunnen *breadth-first* van groot naar klein gekozen worden. Leg uit dat dit inderdaad lukt. *Jan 2016*

2. *Ziv-Lempel-Welsh codering.* Het alfabet heeft vier letters `a`, `b`, `c`, `d`. Letter `a` krijgt code 1.
Geef telkens welke letters 'geleerd' worden, en de uiteindelijke codeboom.

a) Codeer `cdbc acac daaa bc`, spaties staan hier alleen voor de leesbaarheid.

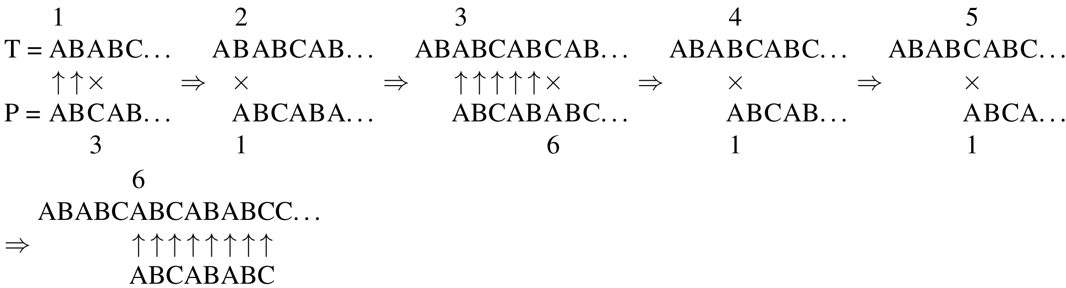
b) Decodeer 2 4 4 5 1 3 7 11 1 2 14 10. *Jan 2017*

10 Pattern Matching

There are many forms of string searching. The most basic form is finding a rather small substring P (the pattern) in a rather large string T (the text).

We start by illustrating the straightforward *naive* approach, aligning the first characters of pattern and text and comparing the consecutive letters one by one. At a mismatch we shift the pattern one to the right and start again at the first character of the pattern, again comparing the letters one by one.

Example 10.1. We try to locate pattern $P =$ 'ABCABABC' in text T which starts with 'ABABCABCABABCC...'. arrows indicate matches, crosses the first mismatch at that position.



This set-up might take a long time on special examples like $P = A^m B$ and $T = A^n B$. Each iteration all letters A of pattern P are matched, until the last letter B is not found in the text. Then the pattern is shifted by one. The method compares all letters of P to all letters of T, and thus is of complexity $O(m \cdot n)$.

In this chapter we present the *Knuth-Morris-Pratt* algorithm which uses preprocessing of the pattern to avoid returning to an earlier position in the text. Other important algorithms are *Rabin-Karp* which computes a running hash of the text and compares it with the hash of the pattern. and *Boyer-Moore* which also preprocesses the pattern, but additionally observes the letters occurring in the text (and matches the pattern backwards).

Even in searching for a single string in a long string many algorithms are available. Some points that can be taken into account in choosing the best one.

- *Size of the alphabet.* Some techniques are applicable to smaller alphabet sizes, like the binary alphabet $0,1$, are the alphabet of DNA bases A,C,G,T. In a smaller alphabet relatively more repetitions will occur.

- *Number of applications.* If we use the same long text multiple times for searching, we might consider preprocessing the text. This usually takes either a long time or huge storage space but modern techniques like the *suffix-tree* and the suffix-array give an efficient approach to answering questions like 'which is the longest substring that occurs twice in the text?'.

The notion of 'pattern' can also be generalized from a single string to a set of strings, or even a pattern in a general sense, like 'palindrome' or 'regular expression'.

Computational biology. In some applications we want to measure the *similarity* of two strings x and y. In a general setting the similarity of two strings is the number of basic operations that is neede de to transform the first string into the second. In bioinformatics this is formulated as finding an *alignment* between the strings. Another typical requirement in bioinformatics is the robustnest against errors, we are not always looking for precise answers, but fast and good approximations are usually preferred.

10.1 Knuth-Morris-Pratt

Features:

- *failure links*
- linear time preprocessing pattern
- search will never back-up in text

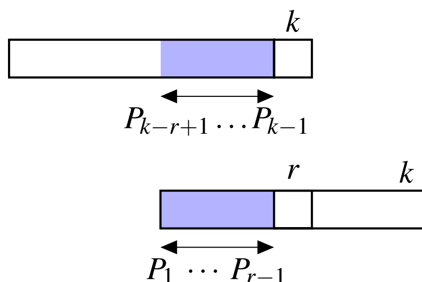
If (during the naive implementation of pattern matching) a mismatch occurs at the k-th position of the pattern, we start again with the first position of the pattern at the next position of the text. We know however that the first $k-1$ positions of the pattern match the text when we tested the pattern at the previous position. We can use this knowledge to predict how the pattern matches at the next position, basically by matching the pattern against itself as a *preprocessing step*. This approach is used in the KMP algorithm: if the shifted pattern does not match the pattern then the shifted will not match the text.

Example 10.2. We consider again the pattern $P =$ 'ABCABABC', and assume a mismatch has been made at position 8. We match the 7-letter prefix ABCABAB against itself. We observe that the first possible match is when the first and last two letters of the prefix are aligned.

```
ABCABAB ?     ABCABAB ?     ABCABAB ?       ABCABAB ?        ABCABAB ?
 ×               ×                ×               ×
. ABCABAB     . . ABCABAB   . . . ABCABAB   . . . . ABCABAB  . . . . . ABCABAB
```

From this we conclude: if a mismatch occurs at position 8 then we can shift the pattern such that the present position of the text aligns with position 3 of the pattern. Always, regardless of the text, it is not useful to shift less.

The *failure link* at position k for the pattern P is the maximal $r < k$ such that $P_1 \dots P_{r-1} = P_{k-r+1} \dots P_{k-1}$. Thus, the maximal overlap of a prefix and a suffix of the pattern $P_1 \dots P_{k-1}$ up to this position. When during the matching process an error occurs at position k, we can only expect a new match if we continue the match of the pattern at the position indicated by the failure link and the *present position* of the text.



For $k = 1$ we set the failure link equal to 0. If an error occurs at the first position, we have to shift the pattern by one position to the right and continue at the first position of the pattern at the next position of the text.

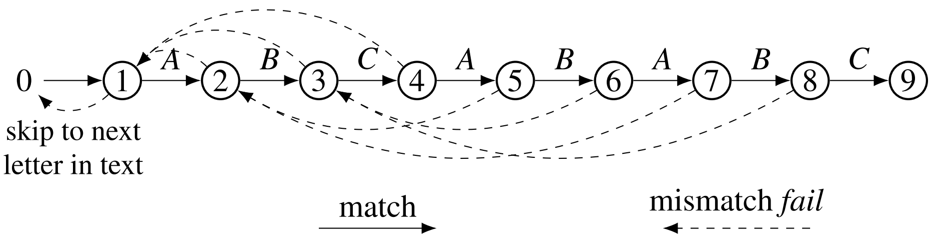
Note that the failure link does *not* take into account which letter was missed, so we do not verify whether or not $P_k = P_r$. This means that if an error occur we may shift the pattern to a new position with the same letter that was rejected in the step before.

Example 10.3. We determine 'by hand' the failure links for the pattern of our running example. For every position we show which is the longest prefix which matches a suffix at the position. (This is slow. We learn an efficient algorithm later.)

2	3	4	5	6	7	8
A	AB	ABC	ABCA	ABCAB	ABCABA	ABCABAB
. A	. . AB	. . . ABC	. . . $\overline{\text{A}}$BCA	. . . $\overline{\text{AB}}$CAB	. . . . . $\overline{\text{A}}$BCA	. . . . . $\overline{\text{AB}}$CABAB
1	1	1	2	3	2	3

Hence we obtain the following table of failure links.

k	1	2	3	4	5	6	7	8
P[k]	A	B	C	A	B	A	B	C
FLink[k]	0	1	1	1	2	3	2	3



◇

The following pseudo-code searches the text using the KMP algorithm assuming the failure links are already computed.

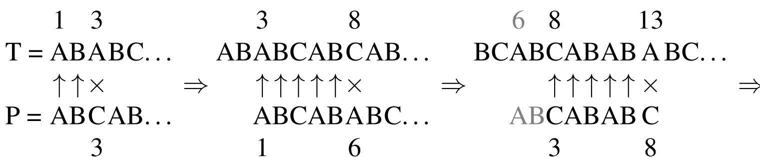
KMP using failure links

```
Pos = 1      // position in pattern
TPos = 1     // position in text
while ((Pos <= PatLen) and (TPos <= TextLen))
do  if (P[Pos] == Text[TPos])
    then Pos ++
         TPos ++
    else Pos = FLink[Pos]
         if (Pos == 0) // next position in text
         then Pos = 1
              TPos ++
         fi
    fi
od
if (Pos > PatLen) // found?
then Pattern found at position TPos-PatLen+1 in Text
fi
```

Example 10.4. We locate the pattern P = 'ABCABABC' from earlier examples, in text T = 'ABAB CABC ABAB ABC...' using the algorithm of Knuth-Morris-Pratt and the failure links obtained above.

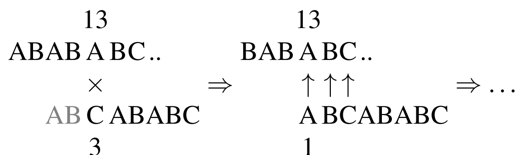
k	1	2	3	4	5	6	7	8
P[k]	A	B	C	A	B	A	B	C
FLink[k]	0	1	1	1	2	3	2	3

(1) Pattern at position 1. First two positions match with text. Mismatch at position 3 of pattern. Failure link at 3 equals 1.



(2) Put 1st position of pattern at present position 3 in text. Positions 3–7 of text match positions 1–5 of pattern. Mismatch at position 6 of pattern. Failure link at 6 equals 3.

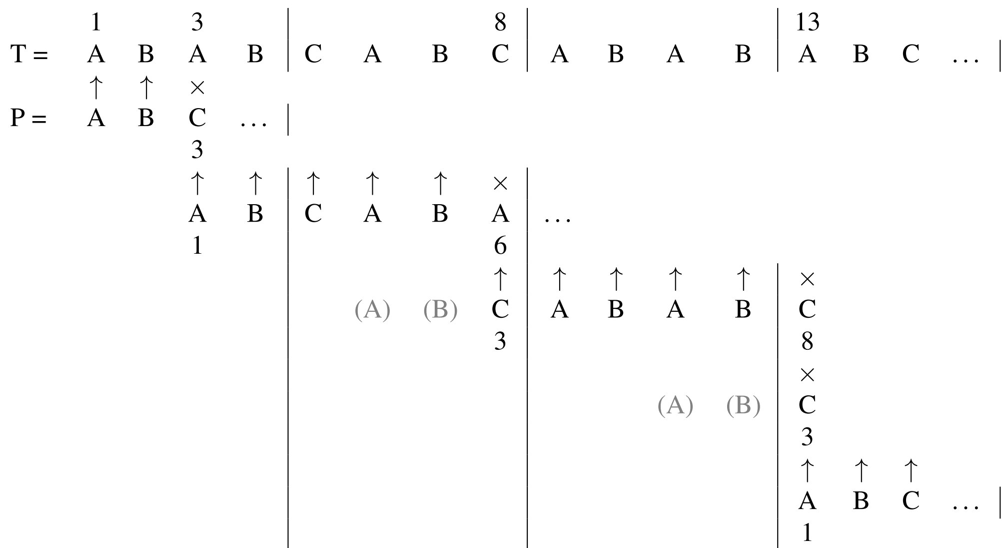
(3) Put 3rd position of pattern at present position 8 in text. Thus the pattern now starts at position 6 in text, but we do not reconsider the first two positions of the pattern: we know they already match the text. Continue comparing letters at position 3 of pattern and position 8 of text. Mismatch at position 8 of pattern, position 13 of text. Failure link at 8 equals 3.



(4) Continue comparing letters at position 3 of pattern and current position 13 of text. Immediate mismatch at position 3. Failure link at 3 equals 1.

(5) Put 1st position of pattern at current position 13 in text. Three matches in a row. (But we do not know continuation of text, so stop here.)

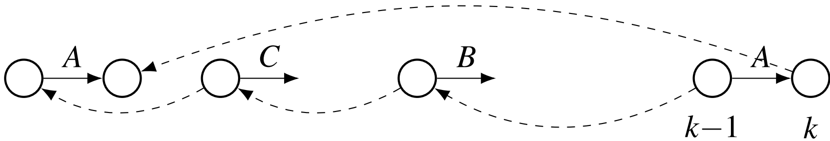
All together, in one diagram (the grayed symbols are *not* compared to the text at that position).



Constructing the failure links. We now present the algorithm to efficiently construct the failure link table of a pattern. In words, at each position pos we try to locate the letter P[pos-1] from the *previous* position following the failure links, starting at Flink[pos-1], the failure link of the previous position.

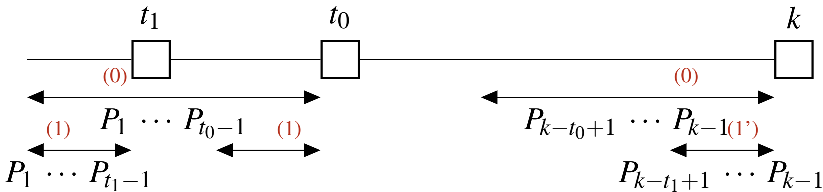
computing KMP failure links

```
Pos = 1 // position in pattern
FLink[1] = 0
for Pos = 2 to PatLen
do  Fail = FLink[Pos-1]
    while ( (Fail > 0) and (P[Fail] != P[Pos-1]) )
    do  Fail = FLink[Fail]
    od
    FLink[Pos] = Fail+1
od
```

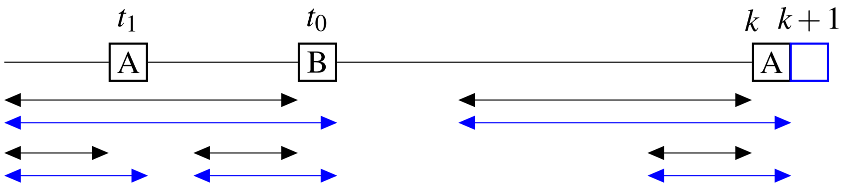


Example 10.5. In our example pattern $P =$ 'ABCABABC' we construct FLink[7] at Pos=7, based on the previous links. Fail = Flink[6] = 3. But P[3] $\neq$ P[6]. So we follow the Flink at that position: Fail = Flink[3] = 1. As P[1] = P[6] we stop and set Flink[7] = Fail+1 = 2.

Correctness. Consider the set of *all* positions $t < k$ such that $P_1 \ldots P_{t-1} = P_{k-t+1} \ldots$ i.e., all prefixes of $P_1 \ldots P_{k-1}$ that are also a suffix of that string. Assume $t_0 < t_1 < t_2 < \ldots$ is this sequence. By definition $t_0 = \text{FLink}[k]$. We claim that $t_i = \text{FLink}[t_{i-1}]$. This follows from the fact that $P_1 \cdots P_{t_i-1} = P_{k-t_i+1} \ldots P_{k-1}$ is a prefix of $P_1 \cdots P_{t_{i-1}-1}$ but also a suffix of $P_{k-t_{i+1}} \cdots P_{k-1}$.



Note that if we delete the last letter from a string that is both a prefix and suffix before position $k+1$ (as we look for in the failure links) then we get a prefix/suffix at position k. If we now want to find the failure link P at position $k+1$, i.e., the longest prefix of $P_1 \ldots P_k$ which is also a suffix of $P_1 \cdots P_k$, then we try to extend the possible prefix/suffixes of $P_1 \ldots P_{k-1}$ by the letter P_k, i.e., we test whether $P_t = P_k$ for the values t as above. Thus we follow the failure links for the previous position, like in the algorithm.



Improving the failure links. A failure link may point to the same character that has been rejected in the previous step, thus P[k] = P[FLink[k]]. This happens in the example *ABCABABC* where FLink[5] = 2. If a mismatch occurs at position 5 (because the text does not have a *B*) in the next step we will again check a *B* (the one at position 2) That is a consequence of the fact that we did not look at the character itself when constructing failure links. This can be fixed in one simple sweep over the array of failure links. (It can also be done in the failure link construction itself, but to me that is slightly confusing.) Note the inner loop is a simple `if`, it is not necessary to use another `while` as the failure link at the previous position already points to a different letter.

```
improving KMP failure links

for Pos = 2 to PatLen
do  if ( P[Pos] == P[FLink[Pos]] )
    then FLink[Pos] = FLink[FLink[Pos]]
    fi
od
```

Example 10.6. Improving failure links in the running example, before and after.

k	1	2	3	4	5	6	7	8
P[k]	A	B	C	A	B	A	B	C
FLink[k]	0	1	1	1	2	3	2	3
FLink′[k]	0	1	1	0	1	3	1	1

◇

Complexity. The total complexity of determining the failure links is linear (in the length of the pattern). In a single step we might have to follow a sequence of failure links, but on this averages over a sequence of operations. The argument follows the ideas of *amortized complexity analysis*. The value of Fail over the various loops may go down several times, but in total cannot be decreased more than it is increased, which is once for every letter of the pattern, in the instruction FLink[Pos] = Fail+1 which sets the Fail for the next letter.

References. D. KNUTH, J.H. MORRIS, V. PRATT: Fast pattern matching in strings. SIAM Journal on Computing **6** (1977) 323–350. doi:10.1137/0206024. For fun (and education) read Section 7, Historical remarks "*The pattern-matching algorithm of this paper was discovered in a rather interesting way*"

See CORMENLRS Section 32.4: The Knuth-Morris-Pratt Algorithm

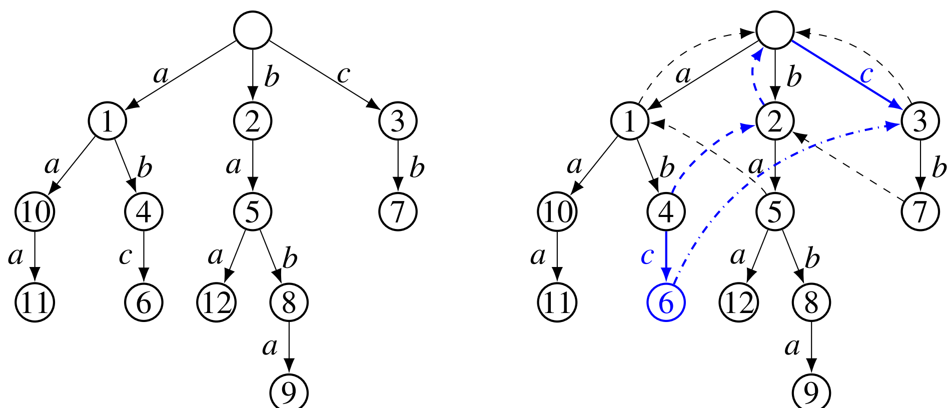
See DROZDEK Section 13.1.2: The Knuth-Morris-Pratt Algorithm

10.2 Aho-Corasick

The algorithm of **Aho-Corasick** builds on KMP, but instead considers a finite set of strings that are searched in parallel. The set of strings is stored in a *trie*. Failure links from one string in the pattern can end somewhere in one of the other strings. They are computed like for KMP, but breadth-first.

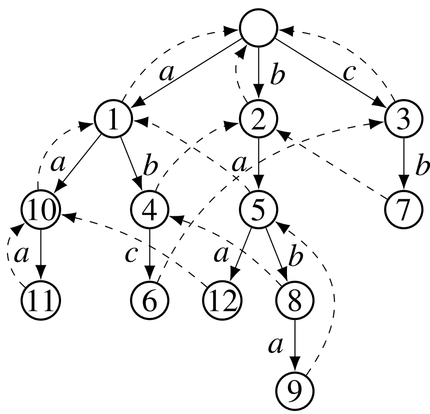
Example 10.7. Consider the set $P = \{aaa, abc, baa, baba, cb\}$. We want to search for occurrence of any of these patterns in a long text, in parallel.

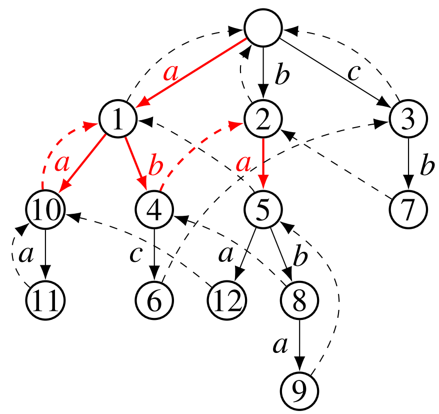
Start by building a trie for $\{aaa, abc, baa, baba, cb\}$ (left). Then add failure links, similar to KMP. For example (right) in blue: constructing failure link at 6. Incoming edge form 4 has label c, now follow the failure links, starting at parent 4 and then search for a node which has outgoing c. First to node 2 which has only outgoing a, then to the root which has outgoing c to node 3. This then is the the failure link at 6. As we see here, this might end up in a different subtree of the trie than where we started.



In this way we obtain the following tire with failure links (left). Then, right in red, we follow the path for text $T = aaba\ldots$. If the wanted outgoing letter is not available, we follow the failure-links to try elsewhere.

Thus after four letters of the text we end in node 5 meaning that the letters ab leading to that node correspond to the prefix ab of the two patterns baa and $baba$, and this is the longest such overlap at this moment.





This has been useful, quote from colleague in paper: “A massive thanks [..] for pointing us to the Aho-Corasick algorithm which resulted in a speed-up of several orders of magnitude.”

References. ALFRED V. AHO, MARGARET J. CORASICK: Efficient string matching: An aid to bibliographic search. Communications of the ACM **18** (June 1975) 333–340. doi:10.1145/360825.360855

See DROZDEK Section 13.1.6: Matching Sets of Words

10.3 Comparing texts ☒

Geen tentamenstof. *Originally written for a course in Molecular Computational Biology.*

The similarity (or rather dissimilarity) between two strings can be measured in the number of operations needed to transform one into the other. There are three basic operations we consider here: changing one character into another, inserting a character, or deleting a character. In the context of molecular biology these operations correspond to mutations (point mutations or insertions and deletions) in the genome.

When we assume that no two operations take place at the same position (like changing a character, then removing it) the operations used to transfer one string into another can be represented by an *alignment* of the two strings. Corresponding symbols are written in columns, marking positions where a symbol was deleted or inserted with a dash in the proper position.

Example 10.8. We have recreated an example of alignment given at wikipedia. It consists of sequences AAB24882 and AAB24881, and was generated using the ClustalW2 tool at the European Bioinformatics Institute, where all settings were left as default. The symbol * in the bottom row indicates that the two sequences are equal at that position, whereas : and . indicate decreasing similarity of the amino acids at that position.

```
82 TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
81 --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                       ****: .***:  * *:** * :****.:* *******..

82 PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
81 HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
    **** *:***********:***:**.: .***************    :  *.: :
```

$\diamondsuit$

Formally an *alignment* of strings x and y over alphabet Σ is a sequence of letter vectors, $\binom{x_1}{y_1}\binom{x_2}{y_2}\dots\binom{x_\ell}{y_\ell}$, with $x_i, y_i \in \Sigma \cup \{\varepsilon\}$ and $\binom{x_i}{y_i} \neq \binom{\varepsilon}{\varepsilon}$ such that $x = x_1x_2\dots x_\ell$ and $y = y_1y_2\dots y_\ell$. Note that $\ell \leq |x| + |y|$. Usually the empty string ε is represented by a dash $-$.

Given two sequences x and y it is an algorithmic task to determine the alignment where the number of operations involved has been minimal, counting the number of positions in the alignment where the two rows have unequal content. This value is called the *edit distance* of x and y.

In general one considers a *weighted* version of this problem by adding a *scoring system*. This in general consists of a similarity matrix σ (or substitution matrix) specifying a value $\sigma(a,b)$ for all a,b in the alphabet (representing the cost of changing a into b) and gap-penalty $\sigma(a,-)$ and $\sigma(-,b)$ for deleting a or inserting b. Thus the score for the general alignment above is given by $\sum_{1=1}^{\ell} \sigma(x_i, y_i)$, where the empty string ε is equated with the dash $-$.

Given a scoring system, the *similarity* of strings x and y is defined to be the maximal score taken over all alignments of x and y. An alignment that has this score is called an *optimal* alignment of x and y.

In simple examples the distances are given by just three values, one fixed value (typically positive) for matches $\sigma(a,a)$, one (typically negative) for mismatches $\sigma(a,b)$, $a \neq b$, and one (also negative) for the 'insdels' (insertions and deletions) $\sigma(a,-)$ and $\sigma(-,b)$. This latter is sometimes referred to as the *gap penalty*.

Example 10.9. The scoring system on the alphabet $\{A, C, G, T\}$ of nucleotides is defined here by the values $+2$ and -1 for match and mismatch, and -1 for gaps.

For the strings TCAGACGATTG and TCGGAGCTG a possible alignment is

TCAG - ACG - ATTG
TC - GGA - GC - T - G

It consists 7 matches and 6 insdels, so its score is $14 - 6 = 8$.

Similarly, the alignment

TCAGACGATTG
TCGGA - GCT - G

consists of 6 matches, 2 mismatches, and 2 insdels. Consequently the score is $12 - 2 - 2 = 8$. Both alignments have the same score, and the similarity of the strings is at least 8.

As stated above, one usually has $\sigma(a,a) > 0$. In some applications also $\sigma(a,b)$ may be positive when a and b are different (but have some similarity). For instance, the BLOSUM62 scoring system, used for amino acids, has this feature. In general σ will be symmetric: $\sigma(a,b) = \sigma(b,a)$. In the sequel the value for insdels is assumed to be given by a fixed gap penalty $g \leq 0$, which does not depend on the symbol that is deleted or introduced.

As noted above, an alignment gives at most a single operation at each position, which seems reasonable in general. Consider the case where deleting A and C costs -5 and -2, respectively, whereas substituting A by C costs -2. Now the two operations $A \to C \to -$ have total cost -4 which is better than the direct deletion of A. In such a case the algorithms of this section will compute the optimal alignment, however this might not correspond to the optimal score set of operations from one string to the other. Usually the scoring system avoids this kind of problems.

Global Alignment Given a pair of strings x and y over an alphabet Σ and a scoring system for Σ, we want to compute the similarity of x and y, and an optimal alignment for the strings.

We use a dynamic programming approach for this problem. The algorithm computes the similarity for each pair of prefixes of the two strings starting with short prefixes, storing the values in a table, and reusing them for the longer prefixes. When the scores of the partial alignments are determined, the second phase starts. The alignment itself is computed from the numbers stored, working backwards. This is called a *traceback*. In the context of molecular biology this method is known as the **Needleman-Wunsch** algorithm.

A recursive implementation of the problem is easily given. Consider the last position of an optimal alignment of strings xa and yb. We have only three possibilities: $\binom{-}{b}$, $\binom{a}{b}$, or $\binom{a}{-}$. Hence the similarity of x and y, the value $\text{sim}(xa, yb)$ of an optimal alignment is found by recursively computing

$$\text{sim}(xa,yb) = \max \begin{cases} \text{sim}(xa,y) & + & g \\ \text{sim}(x,y) & + & \sigma(a,b) \\ \text{sim}(x,yb) & + & g \end{cases}$$

Boundary values (when one of the sequences is empty) can be obtained from the identities $\text{sim}(xa,\varepsilon) = \text{sim}(x,\varepsilon) + g$, and $\text{sim}(\varepsilon,\varepsilon) = 0$.

Let $x = x_1 \dots x_m$ and $y = y_1 \dots y_n$ be two strings that we want to align. Denote the value of the optimal alignment of the prefixes $x_1 \dots x_i$ and $y_1 \dots y_j$ by $A[i, j]$, to stay close to programming style. (In our program strings start at position 1, index $i = 0$ or $j = 0$ corresponds to the empty string.)

The first phase of the algorithm computes the values $A[i, j]$ as follows. The value of the optimal alignment, the similarity of x and y, can be found as $A[m,n]$.

$$\begin{array}{lcll} A[i,0] & = & i \cdot g & 0 \le i \le m \\ A[0,j] & = & j \cdot g & 0 \le j \le n \\ A[i,j] & = & \max \begin{cases} A[i,j-1] & + \; g \\ A[i-1,j-1] & + \; \sigma(x_i,y_j) \\ A[i-1,j] & + \; g \end{cases} & 1 \le i \le m, 1 \le j \le n \end{array}$$

For the second phase, traceback, we assume that for each position (i, j) in the matrix A the cases were stored for which the value of that element was obtained, either $(i, j-1)$, $(i-1, j-1)$, or $(i-1, j)$ – as the three cases in the maximum, representing $\binom{-}{y_j}$, $\binom{x_i}{y_j}$, or $\binom{x_i}{-}$. Now start at the bottom-right position (m,n), and return to the cell of the matrix that resulted in that value. This is repeated until the first cell $(0,0)$ is reached. In many cases the maximum was obtained not for one of the arguments, but for two or even three arguments. In that case we can choose to store just a single of these, or to store all of them, and trace all alignments rather than single one.

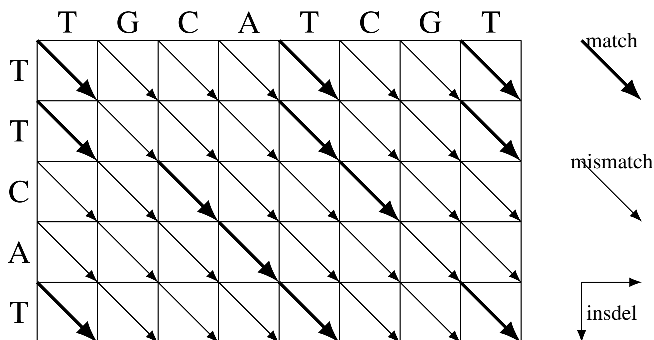
Alternatively, the trace is not followed from stored values, but is recomputed from the values in the matrix at the current position $A[i, j]$ and three neighbouring positions $A[i-1, j]$, $A[i-1, j-1]$, and $A[i, j-1]$.

Complexity. We assume the given strings have length m and n, respectively. The matrix takes space $\mathcal{O}(mn)$, and computing all its elements takes time $\mathcal{O}(mn)$. The traceback is computed in time $\mathcal{O}(m+n)$. If we do not need the alignment itself, but only its score, it is not necessary to store all elements of the matrix but

only the last column (or last row). This reduces the space complexity to $\mathcal{O}(m)$, but time complexity still is $\mathcal{O}(mn)$.

Example 10.10. Global alignment of TTCAT and TGCATCGT with scoring system match 5, mismatch -2, and insdel -6.

A graph represention the problem is as below. The task is to find the optimal path from top-left to bottom-right, where the costs of traversing an edge are related to the label of the edge (negative values represent costs, while positive values can be seen as rewards). Bold diagonal edges represent matches, horizontal and vertical edges represent insdels.

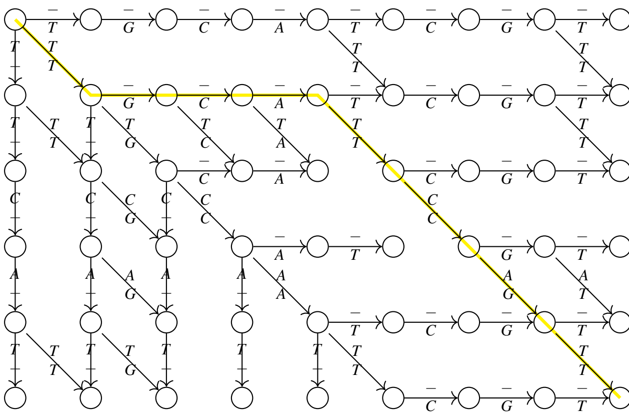


The following matrix is computed by the dynamic programming algorithm. It indicates that the score of the optimal global alignment equals 0.

	–	T	G	C	A	T	C	G	T
–	0	−6	−12	−18	−24	−30	−36	−42	−48
T	−6	5	−1	−7	−13	−19	−25	−31	−37
T	−12	−1	3	−3	−9	−8	−14	−20	−26
C	−18	−7	−3	8	2	−4	−3	−9	−15
A	−24	−13	−9	2	13	7	1	−5	−11
T	−30	−19	−15	−4	7	18	12	6	0

The alignment itself can be traced back from the final position, following incoming edges that represent the direction over which the maximal score was obtained. These edges and a possible traceback are as follows, giving the alignment

T	-	-	-	T	C	A	T
T	G	C	A	T	C	G	T
+5	-6	-6	-6	+5	+5	-2	+5



Two other alignments with optimal score can be read from the diagram.

```
TTCA---T     TTCAT---
TGCATCGT     TGCATCGT
```

Edit Distance. The edit distance between two strings, also called **Levenshtein distance**, counts the minimum number of operations to change one string into the other. Possible operations are inserting or deleting a character, as well as changing a character into another. This corresponds with alignment with match score 0, while mismatch and insdel are both -1.

LCS. A string z is a *subsequence* of string x if z can be obtained by deleting symbols from x. Formally z is a subsequence of $x = x_1 \dots x_m$ if we can write $z = x_{i_1} x_{i_2} \dots x_{i_k}$ for $i_1 < i_2 < \cdots < i_k$. A *longest common subsequence* of strings x and y is a string z of maximal length such that z is subsequence of both x and y. Although z itself may not be unique, the *length* of a longest common subsequence of given strings is.

The problem of finding a longest common subsequence can be answered by computing the alignment where match is rewarded by $+1$ while mismatch and insdel penalty are both 0.

```
A T T G C C - A - - T T
A - T - C C A A T T T T
```

Local Alignment Global alignments attempts to align every character in both sequences. This is useful when the sequences are similar and of roughly equal size. In some cases one expects only parts of the strings to be similar, e.g., when

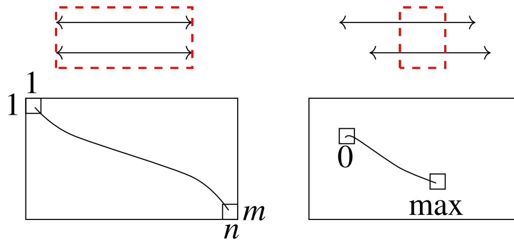


Figure 1: Global versus local alignment.

both strings contain a common motif. In such cases one tries to find segments of both strings that are similar, using local alignment (known as **Smith-Waterman**). This uses a simple adaptation of the global approach, and is the main topic of this section.

Another variant is when one wants to determine whether one string can be seen as extending the other, following a partial overlap. This is motivated by sequence reconstruction based on a set of substrings. Also this can be solved by an adaptation of the dynamic programming technique.

Given two strings x and y, and a scoring system, we want to find substrings x' and y' (of x and y respectively) such that the similarity of x' and y' is maximal.

The main difference with the global version of the algorithm is that we can forget negative values. Whenever a partial alignment reaches a negative value it is reset to zero. As we want to find substrings with maximal alignment we can drop the pieces that give negative contribution. In the same vein the value of the local alignment is not found in the bottom-right corner of the matrix, but rather it is the maximal value found in the matrix. Indeed, extending the maximum alignment will add negative contribution to the value so far obtained, and this part is skipped when stopping at the maximum.

This means we obtain the following algorithm for the computation of local alignment. It differs in two aspects from global alignment. The initialization sets the borders to zero (not the multiples of the gap penalty), and the maximum for the computation of the non-border cells now includes zero, to avoid negative values.

$$
\begin{array}{rcll}
A[i,0] & = & 0 & 0 \leq i \leq m \\
A[0,j] & = & 0 & 0 \leq j \leq n \\
A[i,j] & = & \max \begin{cases} A[i,j-1] & + \quad g \\ A[i-1,j-1] + & \sigma(x_i,y_j) \\ A[i-1,j] & + \quad g \\ 0 \end{cases} & 1 \leq i \leq m, 1 \leq j \leq n
\end{array}
$$

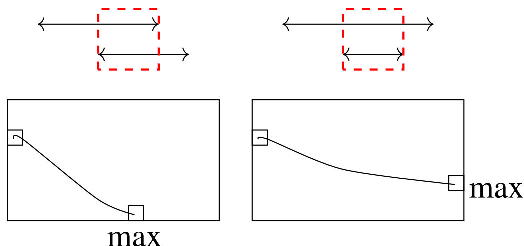


Figure 2: Semi-global alignment: finding overlap or containment.

Example 10.11. Local alignment of ATTCAT and TGCATCGT with scoring system match 2, mismatch -1, and insdel -1.

The following matrix is computed by the dynamic programming algorithm. It indicates that the score of the optimal local alignment equals 7, which is the maximal value in the matrix.

	–	T	G	C	A	T	C	G	T
–	0	0	0	0	0	0	0	0	0
A	**0**	0	0	0	2	1	0	0	0
T	**0**	**2**	1	0	1	4	3	2	2
T	0	**2**	**1**	0	0	3	3	2	4
C	0	1	1	**3**	2	2	5	4	3
A	0	0	0	2	**5**	4	4	4	3
T	0	2	1	1	4	**7**	6	5	6

Tracing back the matrix from the maximal position until a zero is reached one finds one of the following alignments.

```
TTCAT      T-CAT
TGCAT      TGCAT
```

Semi-Global Alignment. In some contexts we are interested in specific overlap between the strings x and y. For instance, when we have a set of (overlapping) random segments of a long string we may reconstruct the original long string using the segments, after we have determined their order using the overlap between the strings. Hence we are interested in determining the maximal overlap consisting of a suffix of x and a prefix of y. As another example when y is much shorter than x it is not very useful to consider the global alignment of x and y to find the possible position of y within x. See Fig. 2 for a pictorial representation of these two cases.

Both these cases share a property with local alignment that gaps at the beginning or the end of one of the strings should not be penalized. As a consequence these problems can be solved in a similar manner. Where initial gaps are free we can include this in the initialization phase of the algorithm (see local alignment), not counting the gap panelty in the first row or column of the matrix. Where final gaps are free we solve this in the final phase of the algoritm. The solution then is not found in the bottom-right cell, but rather is the maximal value on either bottom row or rightmost column.

References. V.I. LEVENSHTEIN: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10 (1966) 707–710.

S.B. NEEDLEMAN, C.D. WUNSCH: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology **48** (1970) 443–53. doi:10.1016/0022-2836(70)90057-4

T.F. SMITH, M.S. WATERMAN: Identification of Common Molecular Subsequences. Journal of Molecular Biology **147** (1981) 195–197. doi:10.1016/0022-2836(81)90087-5

10.4 Other methods ☒

The method of *Boyer-Moore* also takes into account the letters seen in the text at a mismatch. Moreover it works backwards, starting from the last letter of the pattern. If the last letter of the pattern does not fit the letter in the text, and that letter does not occur in the pattern at all, then we can shift the pattern completely over its present interval. Except for finding the first occurrence of the pattern in a string one might ask related questions, like *how many* instances of pattern can be found in text, or what is the longest string that appears at least twice in text. This leads to a surprising efficient data structure that preprocesses the text (not the pattern), called *Suffix Tree* and *Suffix Array*.

Opgave

1.a) Bij het algoritme van Knuth-Morris-Pratt berekenen we *failure links* voor ieder van de posities in het patroon dat we zoeken. Wat weten we van het patroon als gegeven is dat de failure link van positie k gelijk is aan r?

b) Bereken op efficiënte wijze de failure links voor het patroon P = abacaaba.

c) We zoeken naar P in de tekst T = abab caab aaab acab acaa baba (hier staan de spaties louter voor de leesbaarheid). Geef nauwkeurig aan hoe het zoeken volgens het KMP-algoritme gebeurt. Welke letters worden telkens met elkaar vergeleken?

d) Geef een voorbeeld van een type patroon waar KMP significant sneller werkt dan het naïeve algoritme.

(Het naïeve algoritme om patronen in een tekst te zoeken werkt als volgt: we leggen het patroon aan het begin langs de tekst en we vergelijken patroon en tekst letter voor letter. Wanneer er verschil optreedt, schuiven we het patroon één letter verder en we beginnen opnieuw.) *Mrt 2016*

Standard Reference Works

Some of the sections carry references to chapters of the books that are listed below. Other references may carry a DOI (digital object identifier) linking to the original paper of a certain data structure or algorithm. These papers are not always open access, but most of these can be retrieved within the university.

The Leiden University course DATASTRUCTUREN 20XX is based on the books by Drozdek and Weiss. Some (otherwise logically fitting) topics are skipped, as they were part of the predecessor course ALGORITMIEK which uses the book by Levitin.

T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST, C. STEIN. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill 2009. [new edition in 2022]

A. DROZDEK. Data Structures and Algorithms in C++, 4th / International Edition. Cengage Learning 2013.

D.E. KNUTH. The Art of Computer Programming (TAoCP), Volume 1: Fundamental Algorithms, 2nd ed., Volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973. [latest third 'boxed' edition 2011]

A. LEVITIN. Introduction to the Design and Analysis of Algorithms. (2nd Edition) Pearson International 2007. [This is not the latest edition]

M.A. WEISS. Data Structures and Algorithm Analysis in C++ (4th edition). Pearson 2014.

Links

Cyan links lead to outside the document, to the original publications or Wwikipeda, for example. Red links lead to other positions in the document. If you are using Acroread, then you can return to the previous position by pressing alt-←.

Dank

Dit college werd in eerdere jaren ook (mede-)gegeven door Rudy van Vliet, Stijn de Gouw, Jonathan Vis en Fenia Aivaloglou. Ze hebben diverse nuttige verbeteringen in de presentatie aangebracht. Bedankt!

Vorige jaargangen studenten hebben al veel krakkemikkige zinnen, halfbakken uitleg en domme tiepfouten helpen opsporen. De docent en toekomstige studenten houden zich aanbevolen voor verdere opmerkingen.