Datastructuren

Data Structures

Hendrik Jan Hoogeboom
Mark van den Bergh

Informatica – LIACS
Universiteit Leiden

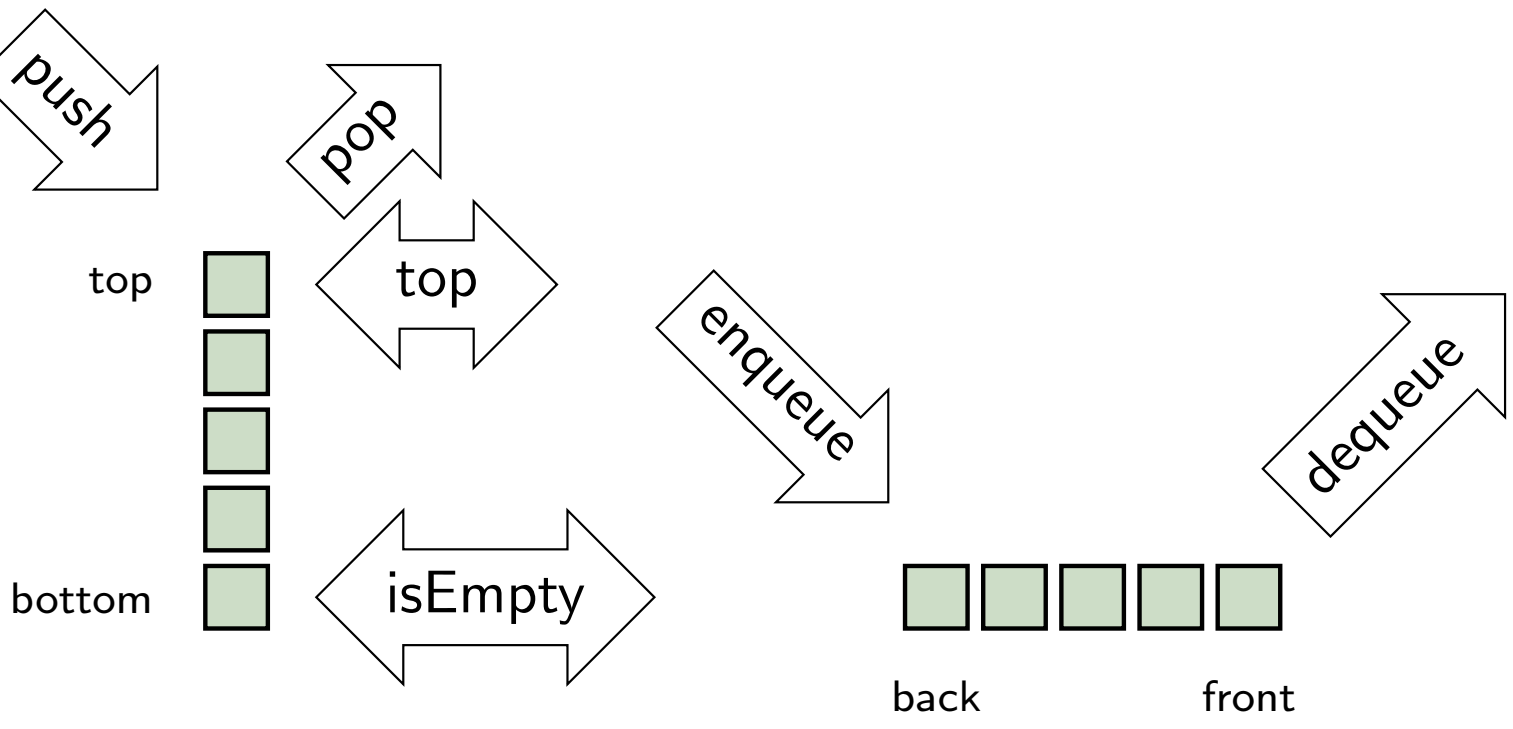
najaar 2024

Table of Contents I

Contents

Contents

stack & queue



Stack *Stapel*

LIFO

Queue *Rij*

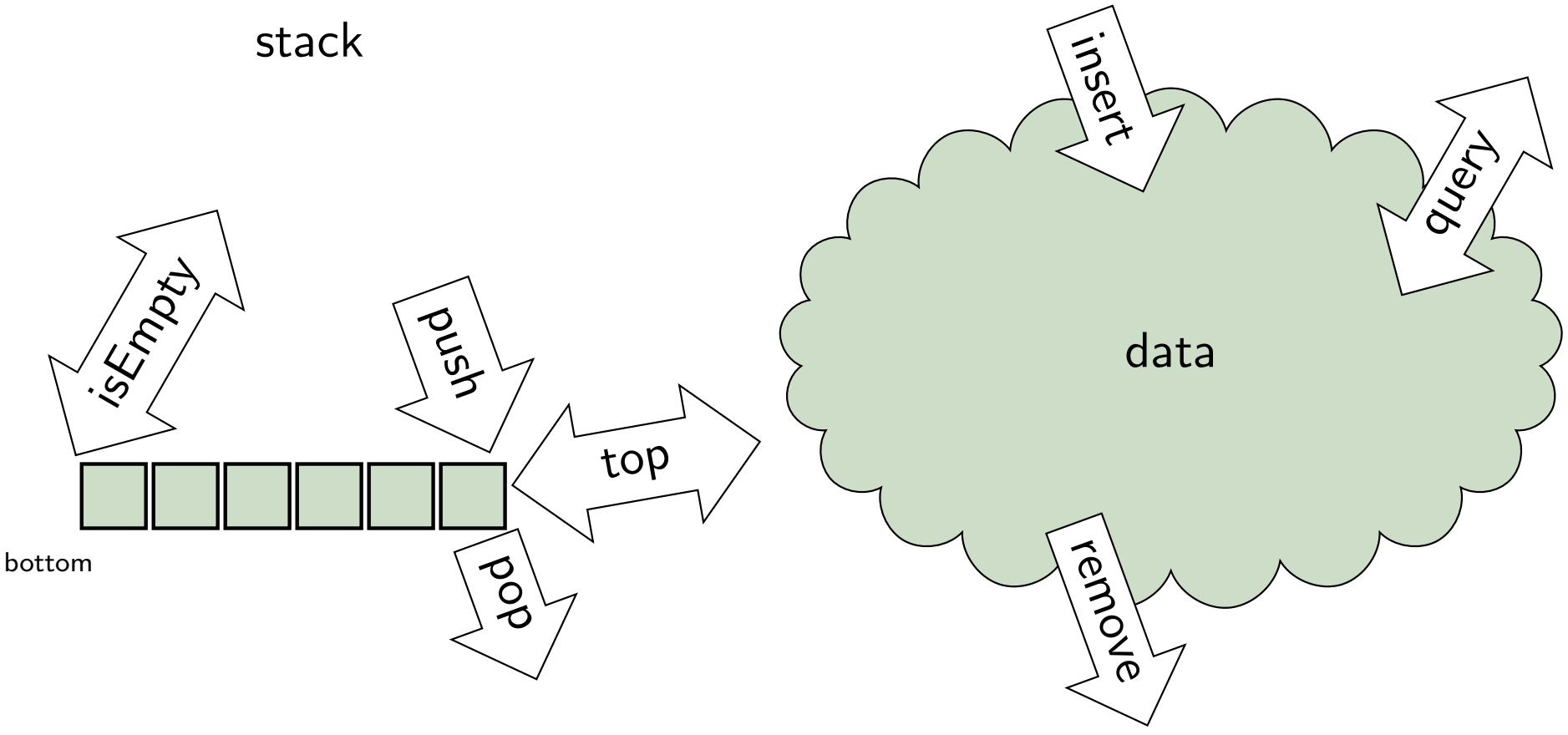
FIFO

abstract "native" data structures

- `float` $\mathbb{R}$ $+, *$
- `int` $\mathbb{Z}$

now get used to consider stacks (etc) that way

black box



abstract data structure

Definition

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.

- no *representation* or *implementation* in ADT
- "mathematical model"

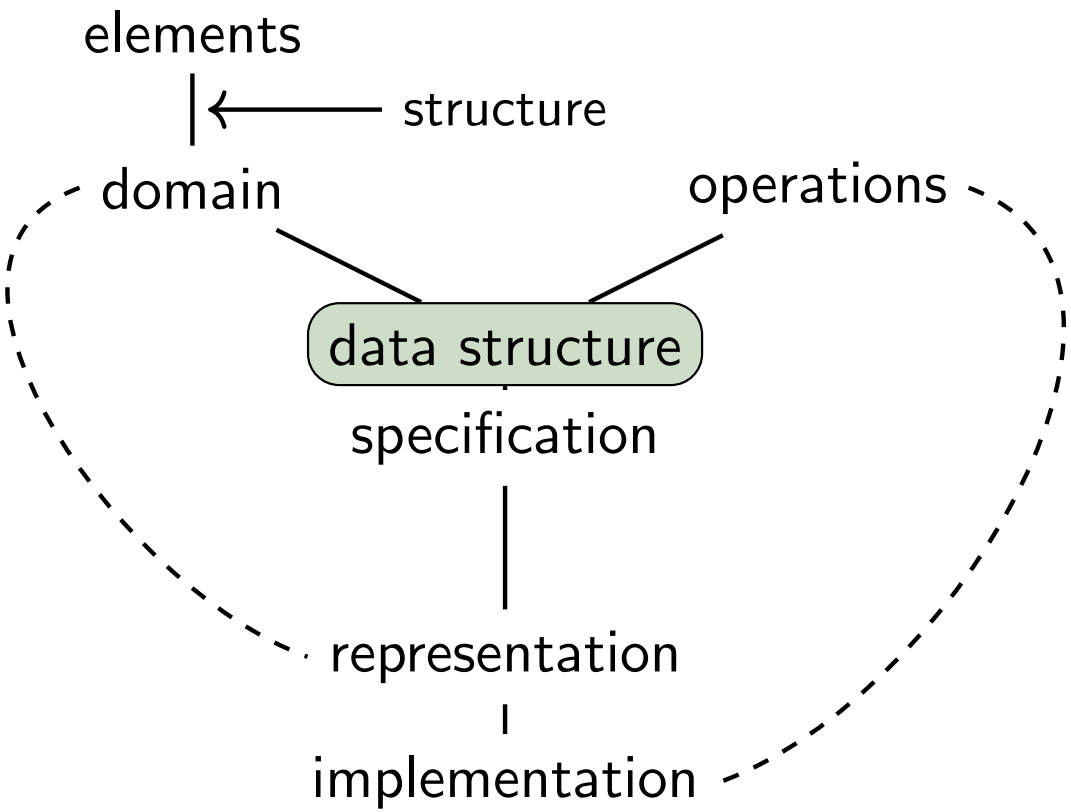
ADT Stack

- stores a (linear) *sequence* of objects (of type <T>)
- access from *one side only*, top of stack
- operations / tests
  - initialize, emptiness, size, (standard)
  - inspect "top", add "push", delete "pop" (specific)

ADT Stack

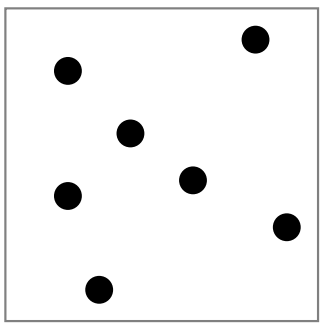
- Initialize: `void` $\rightarrow$ `stack<T>`. construct an empty sequence $()$.
- IsEmpty: `void` $\rightarrow$ Boolean. check whether there the stack is empty, i.e., contains no elements).
- Size: `void` $\rightarrow$ Integer. return the number n of elements, the length of the sequence $(x_1, \ldots, x_n)$.
- Top: `void` $\rightarrow$ `T`. returns the top x_n of the list $(x_1, \ldots, x_n)$. Undefined on the empty sequence.
- Push(x): `T` $\rightarrow$ `void`. add the given element x to the top of the sequence $(x_1, \ldots, x_n)$, so afterwards the sequence is $(x_1, \ldots, x_n, x)$.
- Pop: `void` $\rightarrow$ `void`. removes the topmost x_n element of the sequence $(x_1, \ldots, x_n)$, so afterwards the sequence is $(x_1, \ldots, x_{n-1})$. Undefined on the empty sequence.

ADT Stubbs and Webre



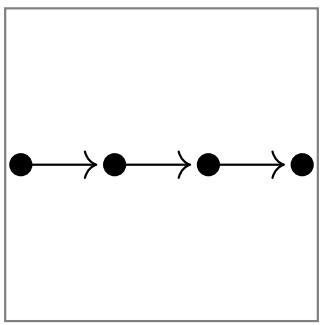
structure

unordered



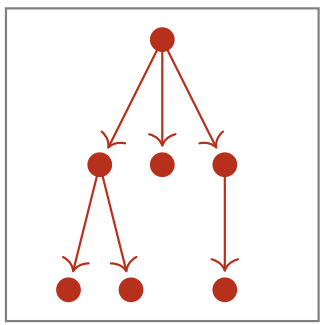
set

linear



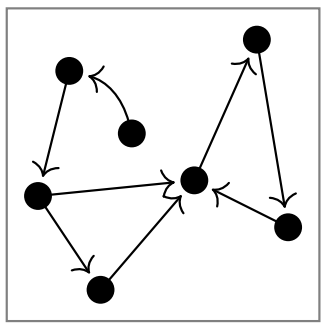
list

hierarchical



tree

network



graph

note: structure abstract ADT and its concrete implementation may differ
– `Set` is unordered, implementation as tree using linear key-order.
– `PriorityQueue` is linear, implementation as Heap=tree using linear array.

ADT Stack

- Initialize: `void` $\rightarrow$ `stack<T>`. construct an empty sequence $()$.
- IsEmpty: `void` $\rightarrow$ Boolean. check whether there the stack is empty, i.e., contains no elements).
- Size: `void` $\rightarrow$ Integer. return the number n of elements, the length of the sequence $(x_1, \ldots, x_n)$.
- Top: `void` $\rightarrow$ `T`. returns the top x_n of the list $(x_1, \ldots, x_n)$. Undefined on the empty sequence.
- Push(x): `T` $\rightarrow$ `void`. add the given element x to the top of the sequence $(x_1, \ldots, x_n)$, so afterwards the sequence is $(x_1, \ldots, x_n, x)$.
- Pop: `void` $\rightarrow$ `void`. removes the topmost x_n element of the sequence $(x_1, \ldots, x_n)$, so afterwards the sequence is $(x_1, \ldots, x_{n-1})$. Undefined on the empty sequence.

Specification ☒

VDM Vienna Development Method

`Stack of N`

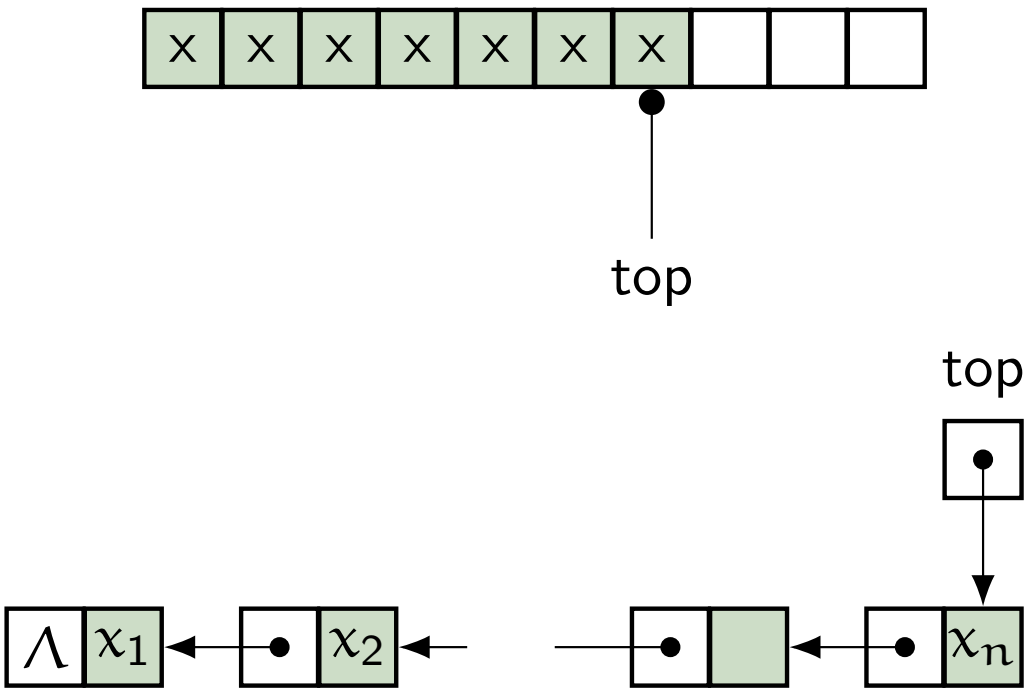
signatures

```
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
```

stack representation/implementation



Programmeermethoden

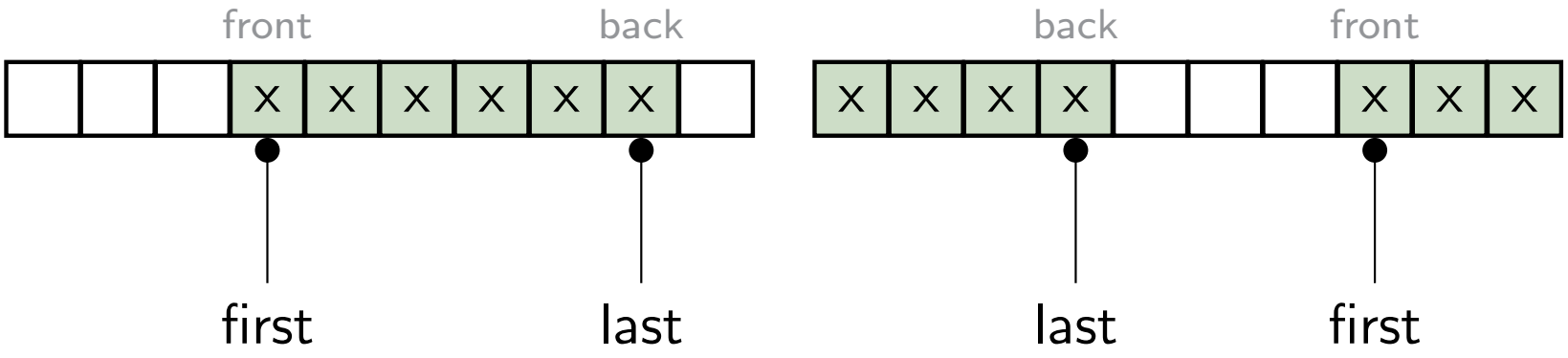
```
class stapel {  // de stapel zelf
   public:
      stapel ( ) {
         bovenste = nullptr; }  // maak lege stapel
      ~stapel ( );              // destructor
      void zetopstapel (int);       // push
      void haalvanstapel (int&);    // pop
      bool isstapelleeg ( ) {       // is stapel leeg?
         return ( bovenste == nullptr );
      }//isstapelleeg
      ...
   private:                  // het begin van de lijst is
      vakje* bovenste;       // de bovenkant van de stapel
};//stapel


void stapel::zetopstapel (int getal) {     // push
   vakje* temp = new vakje;
   temp->info = getal;
   temp->volgende = bovenste;
   bovenste = temp;
}//stapel::zetopstapel
```

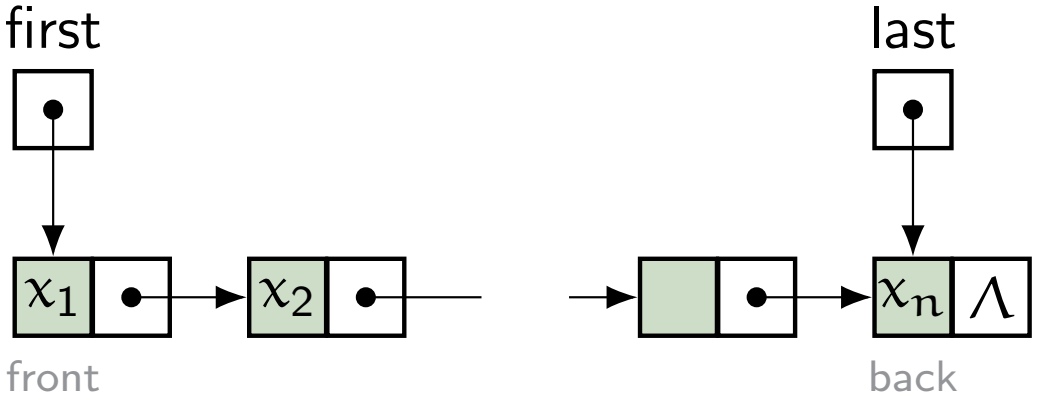
ADT Queue

- Initialize: construct an empty sequence $()$.
- IsEmpty: check whether there the queue is empty, i.e., contains no elements).
- Size: returns 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.

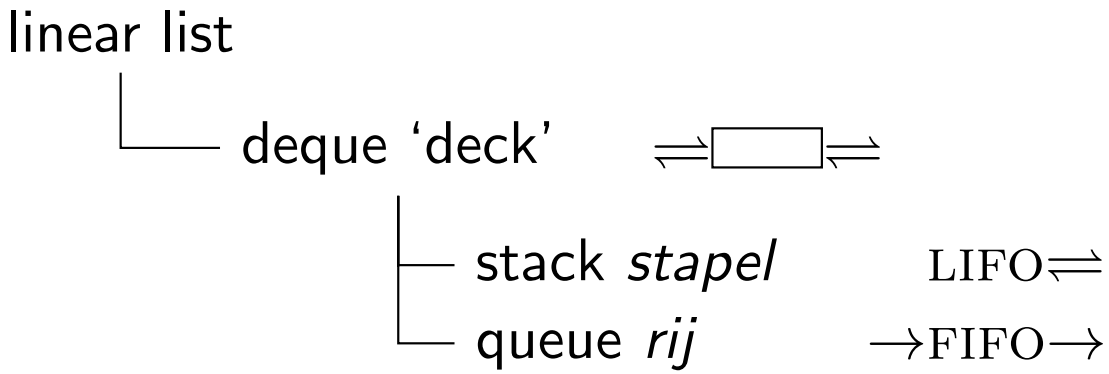
queue: circular array & list



how does the empty/full queue look like?



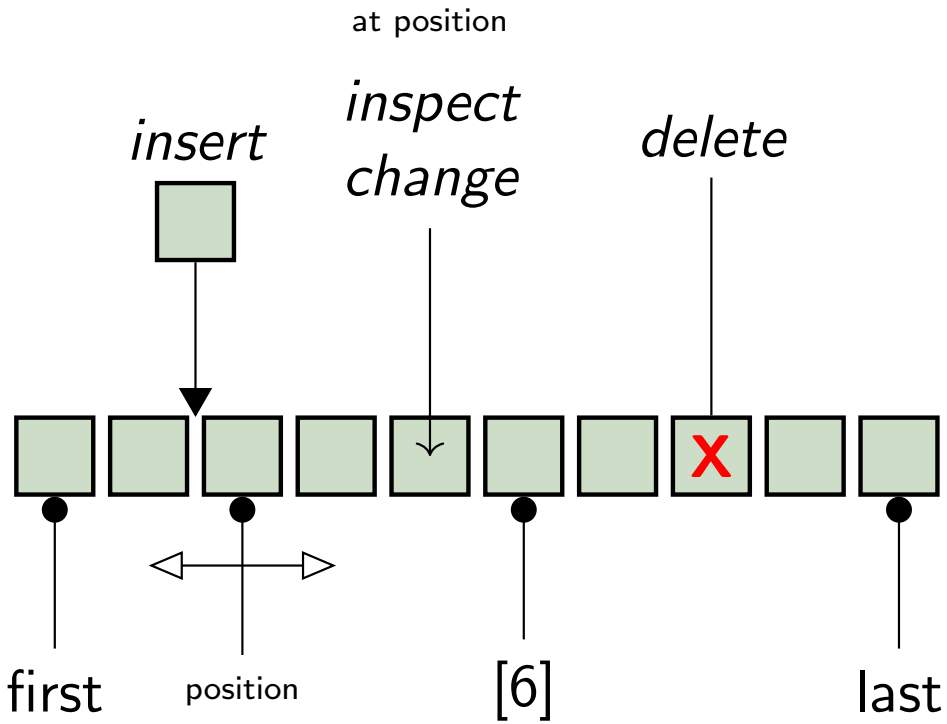
hierarchy of lists



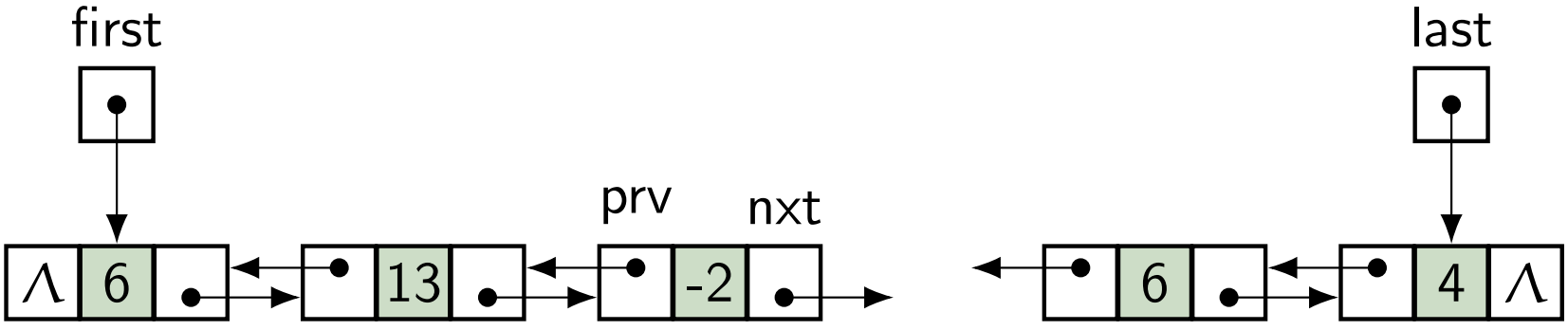
babel ☒

	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]

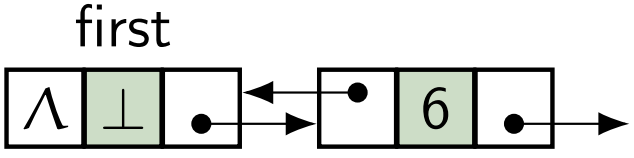
ADT linear list



doubly linked list



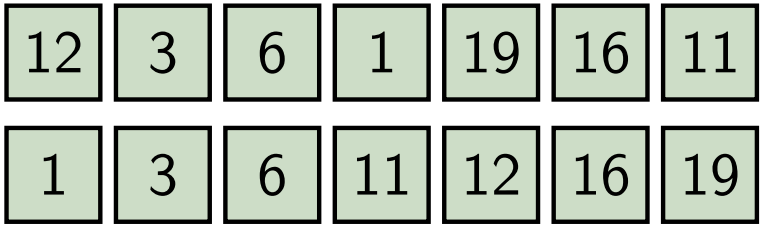
sentinel to avoid 'empty' list



list representations (of SET)

SET

$S \stackrel{?}{=} \varnothing$, $x \in S$, $|S|$, $S \cup \{x\}$, $S \setminus \{x\}$



(unsorted) (array/linked)

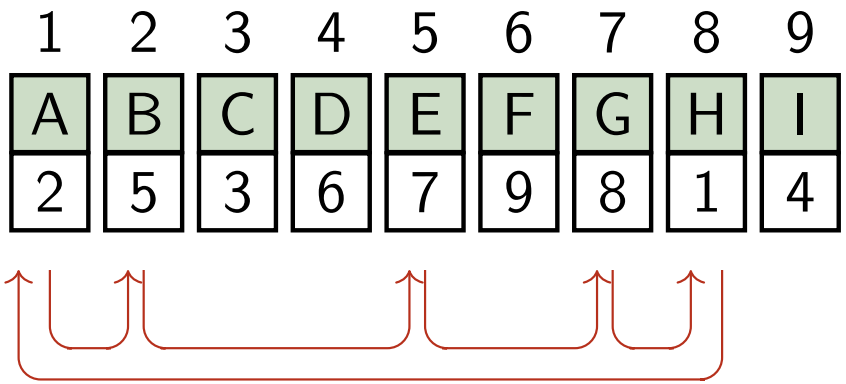
(sorted)

		$x \in S$	$S \cup \{x\}$	$S \setminus \{x\}$
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	

cyclic lists, with indices

UnionFind

{A, B, E, G, H}, {C}, {D, F, I}



ADT's

this lecture series: ADTs (and their implementation)

- Set
  (balanced) binary trees, B-trees, red-black, hash tables
- Map (key,value)
  based on Set
- PriorityQueue
  binary heap, leftist heap
- Graph
  adjacency lists, matrix
- Union-Find
  (circular) lists, (inverted) trees

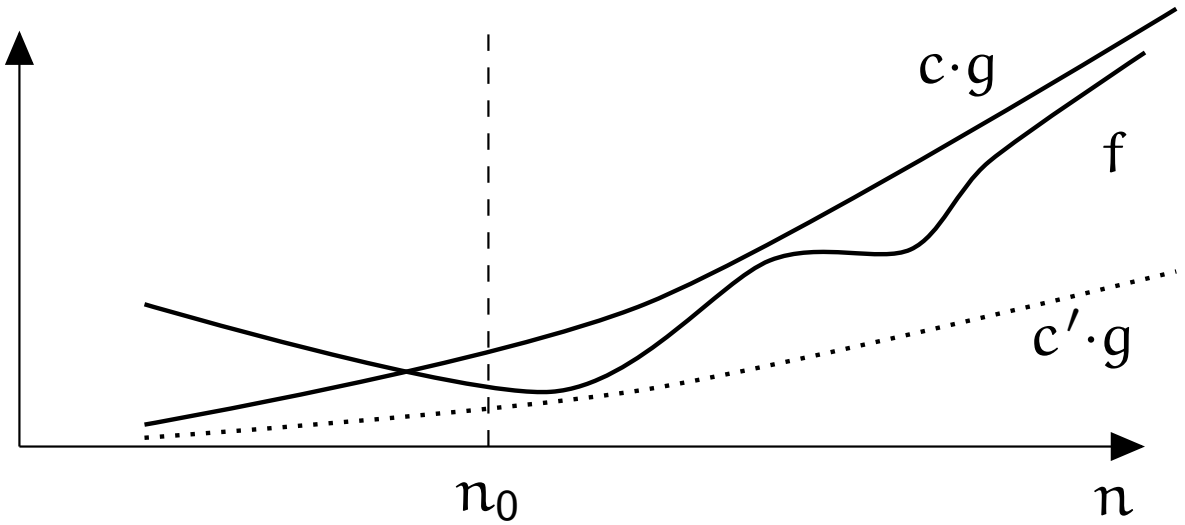
implementations (average vs. worst case)

Data Structures

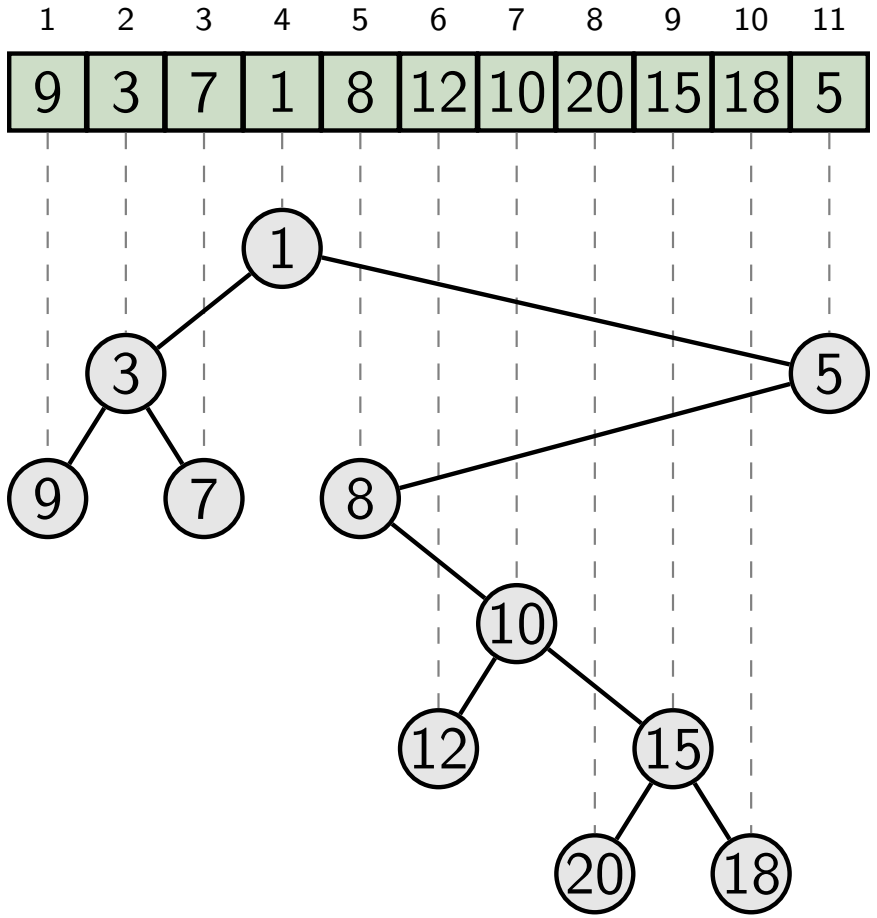
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

source: cpp-cheat-sheet

grote O en Θ



Cartesian tree ☒



see: Cartesian tree

Contents

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
```

work abstractly: hide implementation of data structure to users

OOP: object oriented programming

object class
– *data* members +
– *methods*

inheritance

- data encapsulation ⇒ nicer modelling
- localization operations ⇒ easier error finding
- information hiding ⇒ avoiding errors

see Programmeermethoden, -technieken

templates

- templated function

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

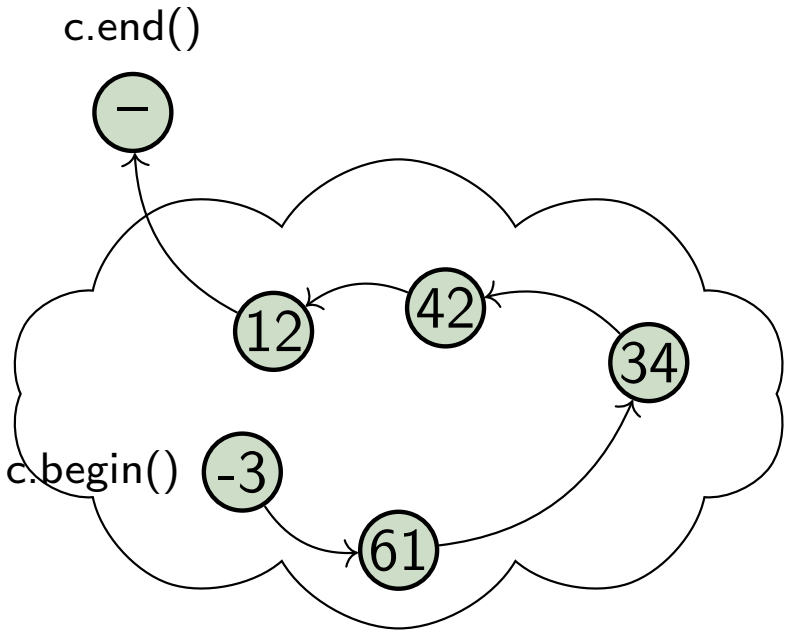
- templated class

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

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

Standard Template Library

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



STL container classes

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`

STL vector and pair

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

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)
```

Contents

tree restrictions

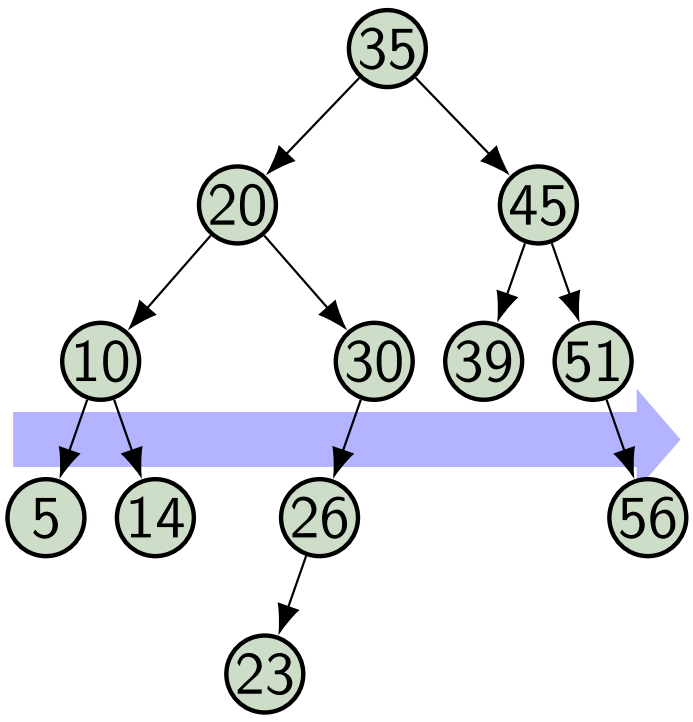
structure

- number of children
  (binary, B-tree)
- number of keys in node
  (B-tree)
- number of nodes subtree
  (*balanced*: AVL, leftist)
- complete
  (binary heap)

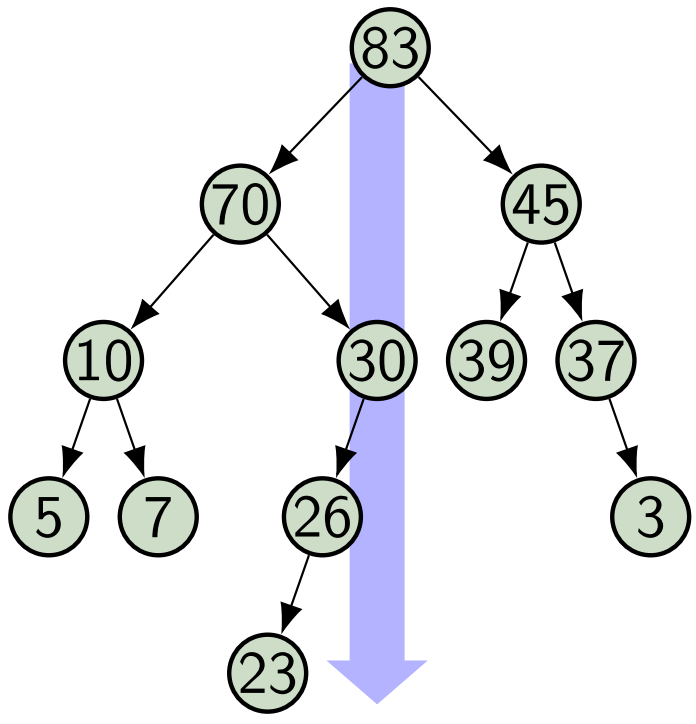
ordering keys

- search tree
- heap ordering
  (binary heap, leftist)

ordering keys

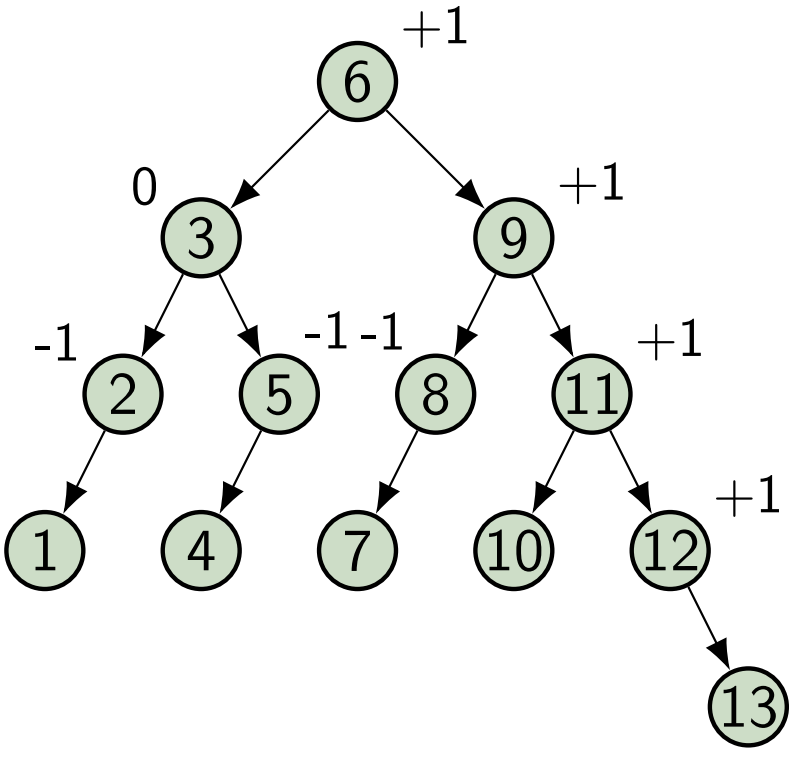


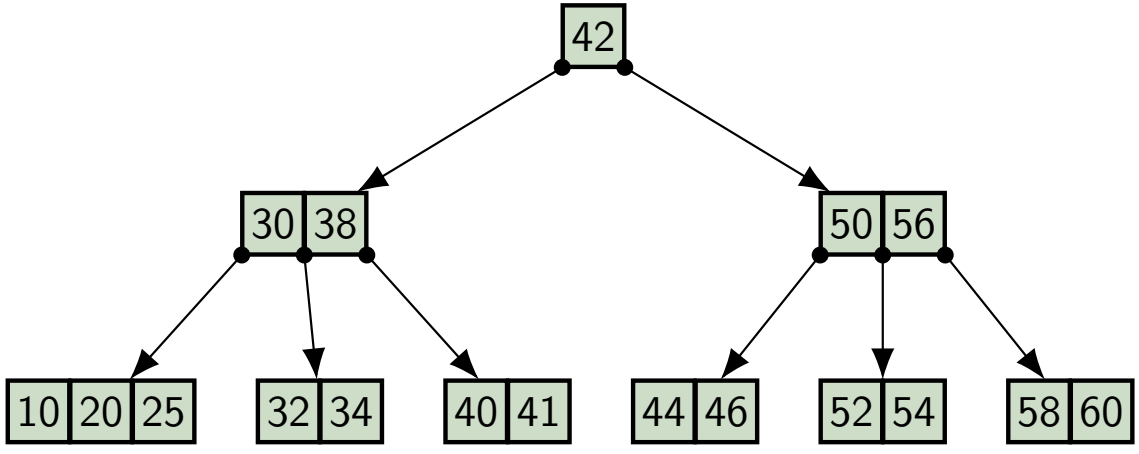
search tree



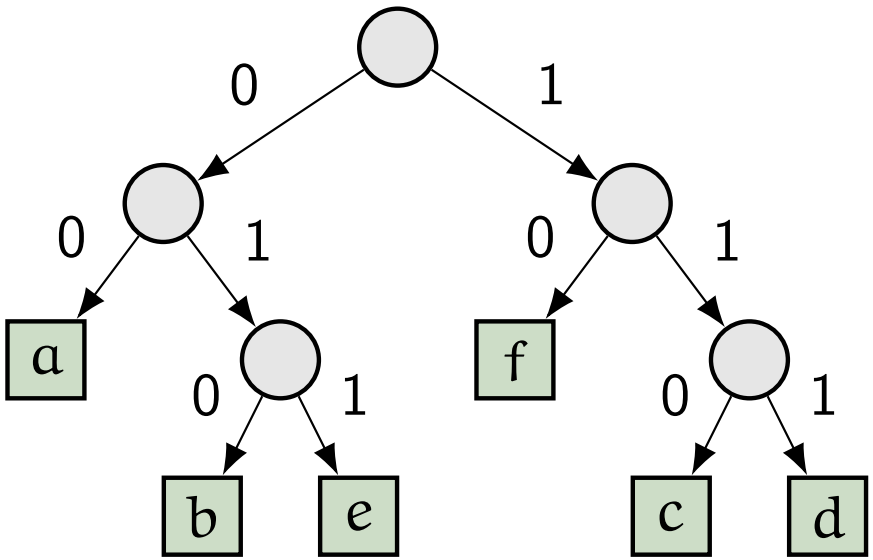
heap order

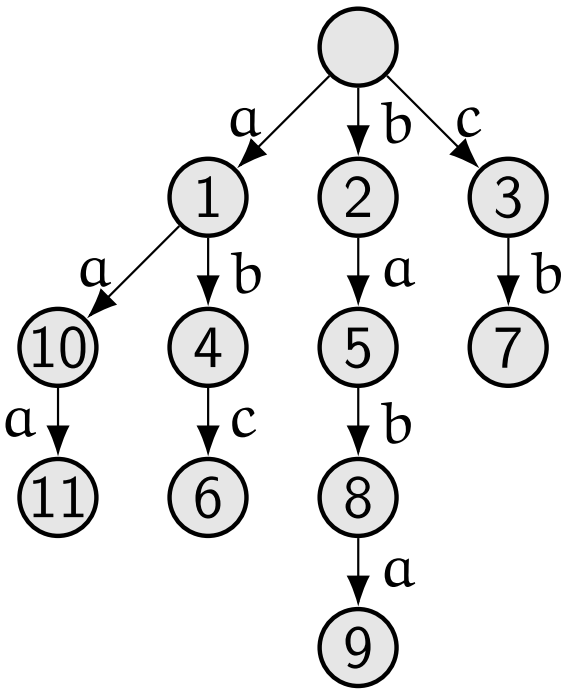
balanced trees: AVL-tree, B-tree



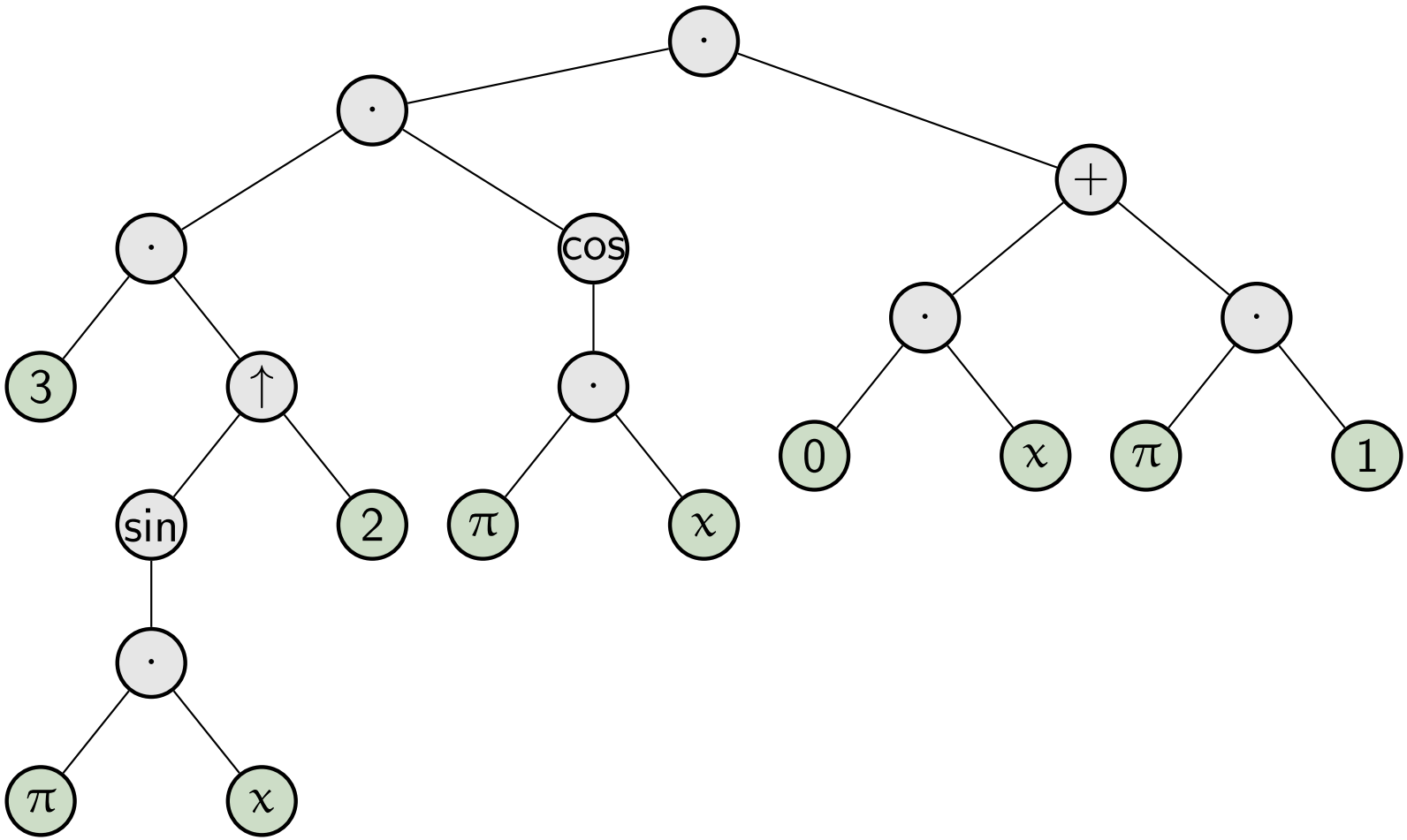


text compression: Huffman, ZLW

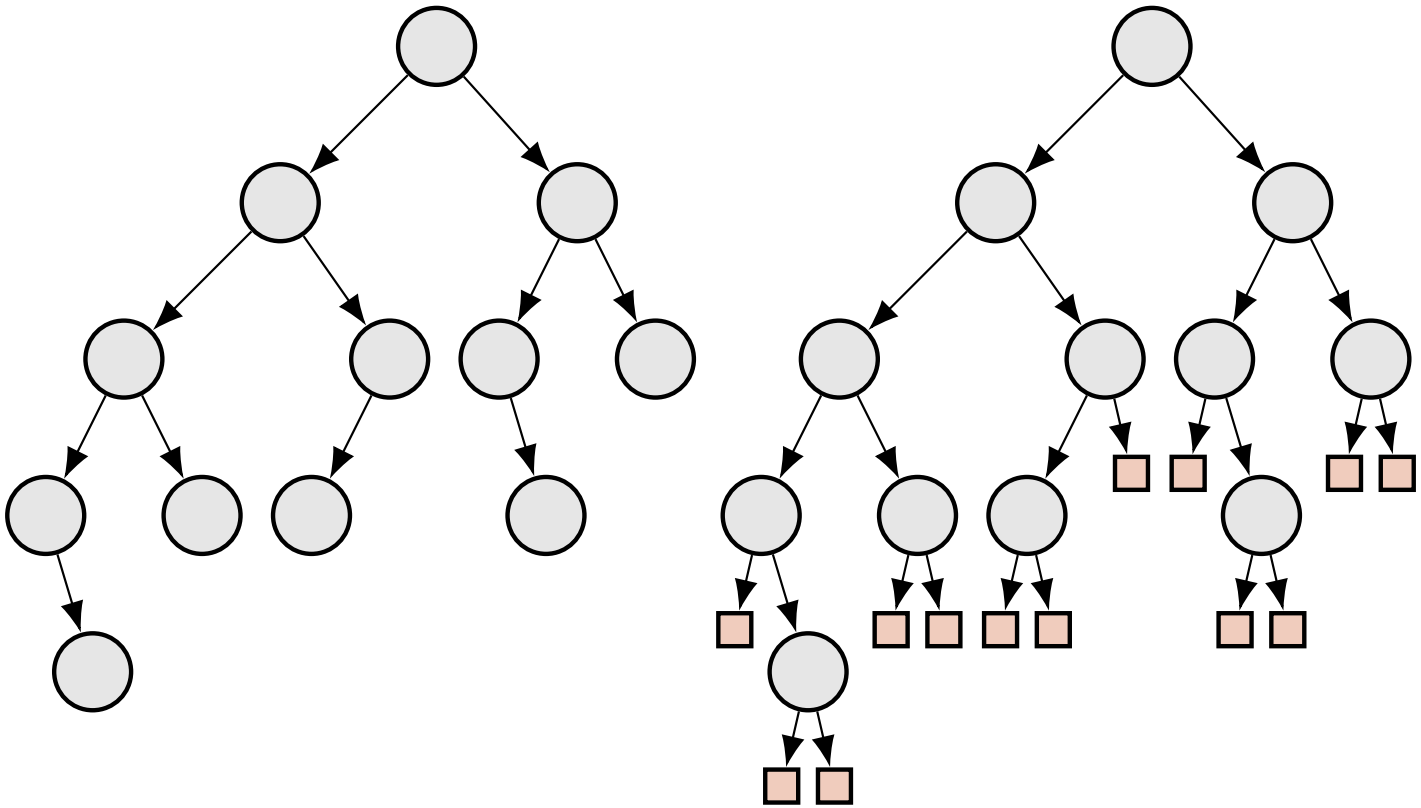




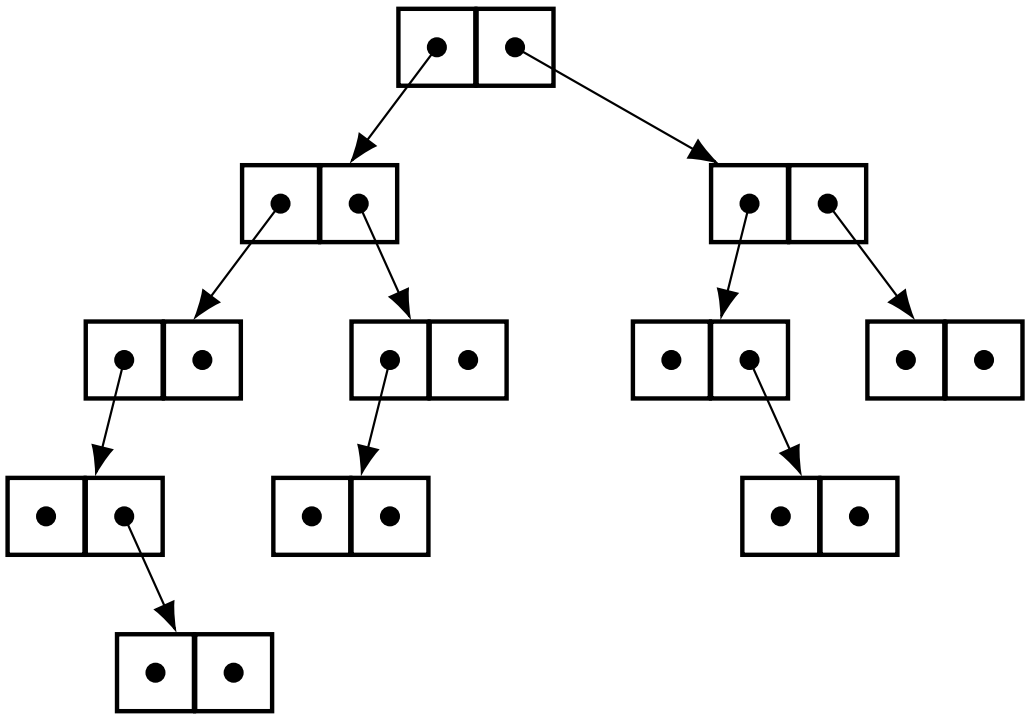
expression tree



full binary tree



pointer representation

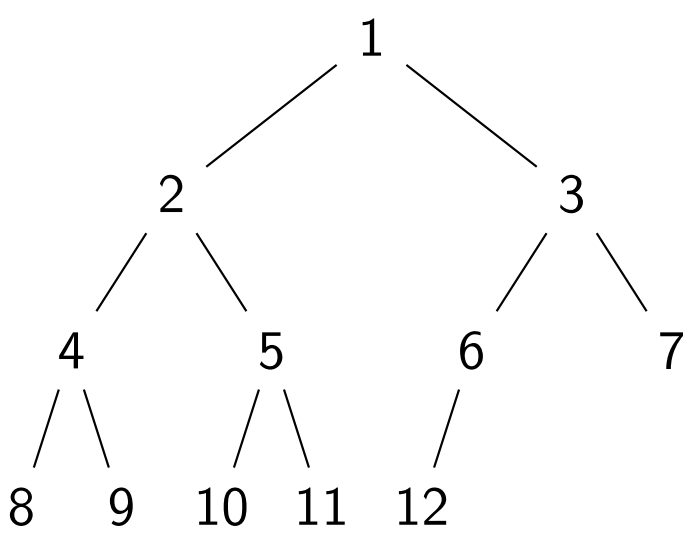


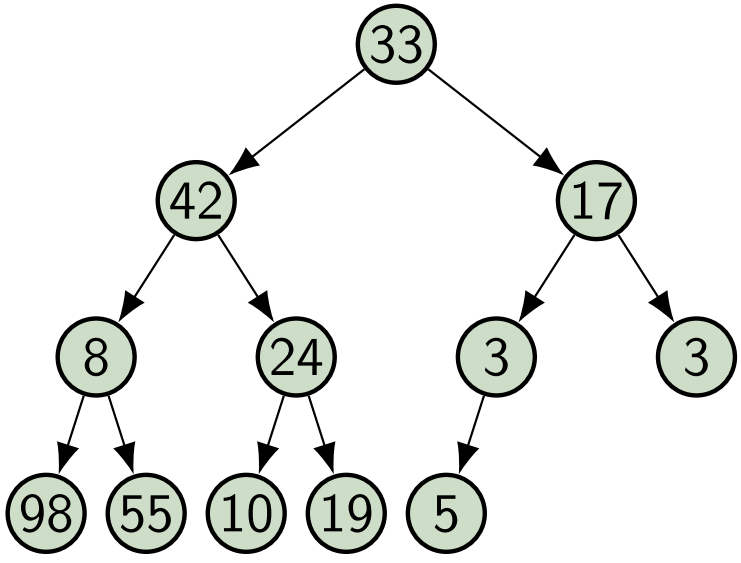
nodes and leaves

Lemma

Let T *be a full binary tree.*
If T *has* $\mathfrak{n}$ *internal (non-leaf) nodes, then* T *has* $\mathfrak{n}+1$ *leaves.*

complete binary tree



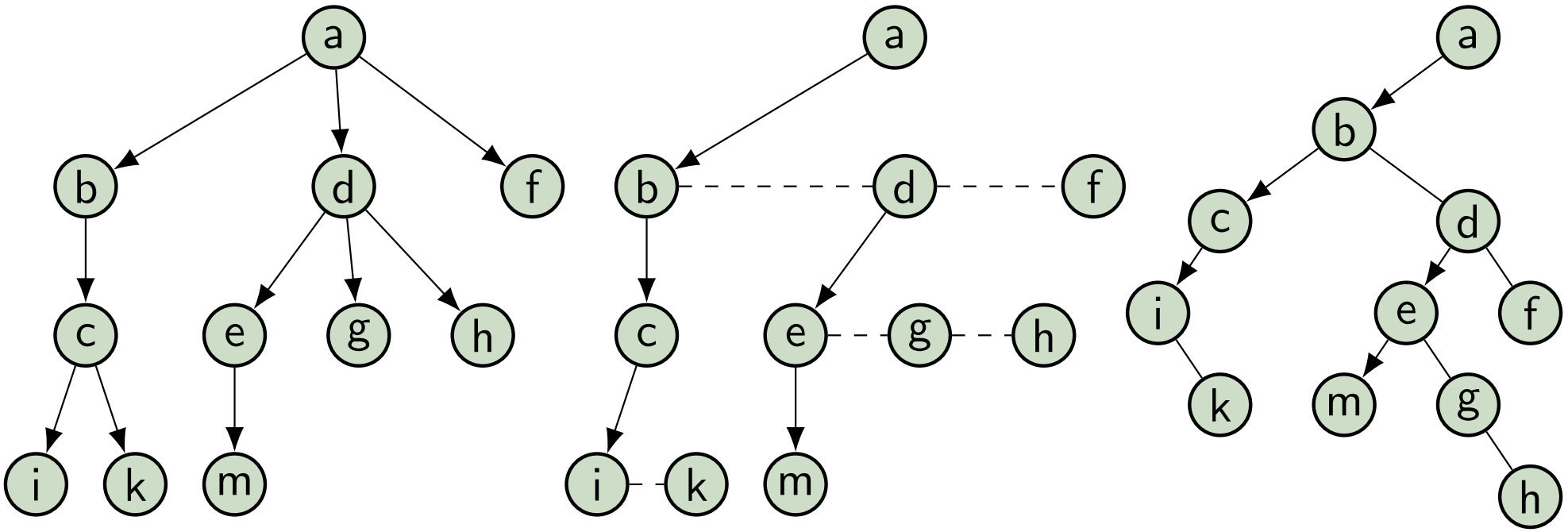


33	42	17	8	24	3	3	98	55	10	19	5
1	2	3	4	5	6	7	8	9	10	11	12

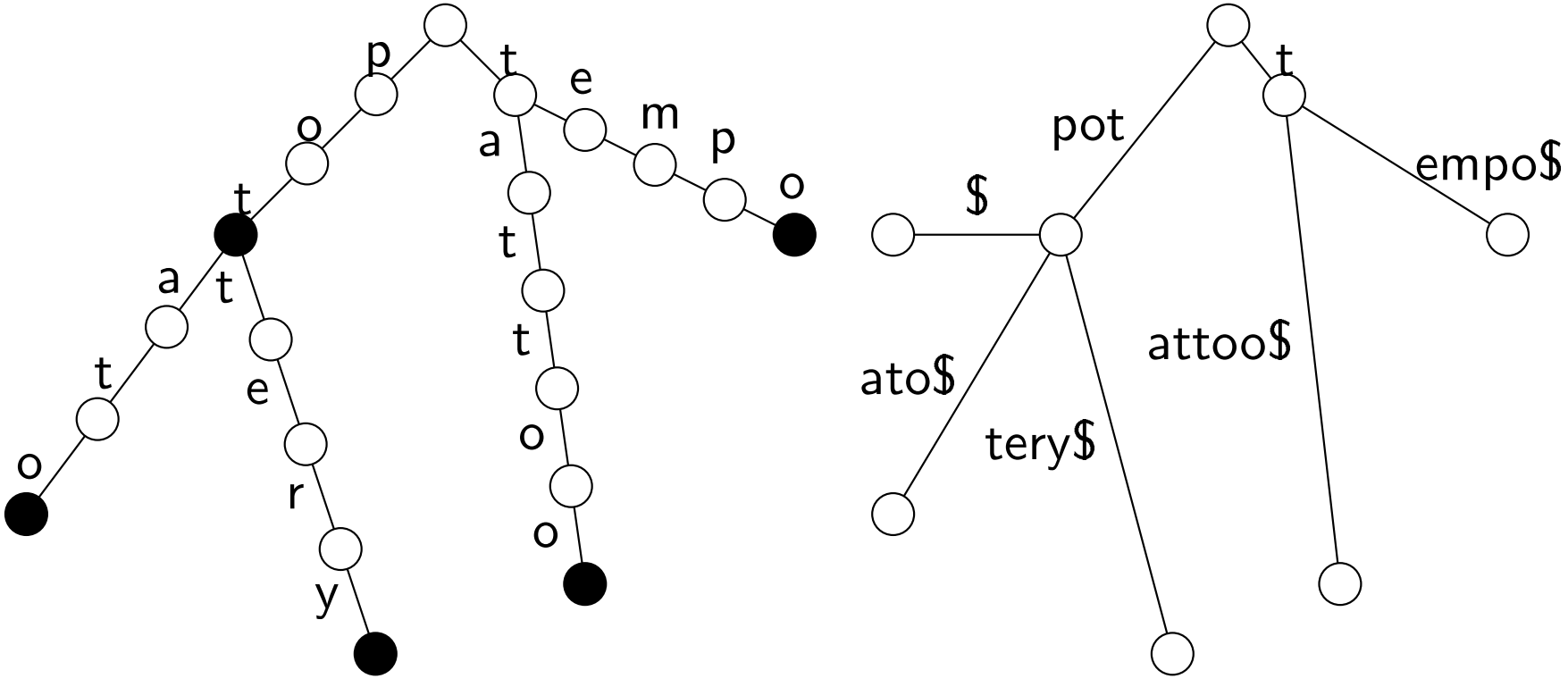
implementation binary tree: pointers

```
template <class T>
class BinKnp {
          \\ CONSTRUCTOR
    BinKnp ( const T& i,
             BinKnp<T> *l = nullptr,  \\ default
             BinKnp<T> *r = nullptr )
         : info(i)    \\ constructor of type T
         { links = l; rechts = r; }
private:   \\ DATA
    T info;
    BinKnp<T> *links, *rechts;
};
```

left-child right-sibling



trie ‘retrieval’



end.