Datastructuren

Data Structures

Hendrik Jan Hoogeboom
Mark van den Bergh

Informatica – LIACS
Universiteit Leiden

najaar 2024

Table of Contents I

Contents

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”

seen: ADT dictionary = map = associative array

Stores a set of (key,value) pairs

- INITIALIZE, ISEMPTY, SIZE
- INSERT: add (key,value) pair, provided key is not yet present
- DELETE: deletes (key,value) pair, given the key
- FIND: returns the value associated to a given key
- SET: reassigns a new value to a (existing) given key

implementations: list, (balanced) *binary search tree*,
or *hash table* “unordered”

Contents

ADT priority queue

- 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.

additionally

- 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.

dictionary vs. priority queue

set of (key,value) pairs

{ (‘Detra’,17), (‘Nova’,84), (‘Charlie’,22), (‘Henry’,75), (‘Elsa’,29) }

based on key

map/dictionary:

Insert(‘Roxanne’,92)
Delete(‘Detra’)
Find(‘Elsa’) returns 29
Set(‘Henry’,76)

based on value

priority queue:

Insert(‘Roxanne’,92)
DeleteMax()
GetMax() returns (‘Nova’,84)

example from wikibooks

min & max queues

max-queue $\geqslant$

INITIALIZE, ISEMPTY, SIZE, INSERT, DELETEMAX, GETMAX, INCREASEKEY, MELD

min-queue $\leqslant$

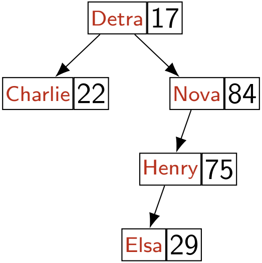
INITIALIZE, ISEMPTY, SIZE, INSERT, DELETEMIN, GETMIN, DECREASEKEY, MELD

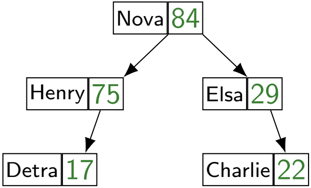
- even opletten welke ordening
- tekenen alleen prioriteit (vergeten de data)

priority queue - use cases

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

keys and values stored in tree

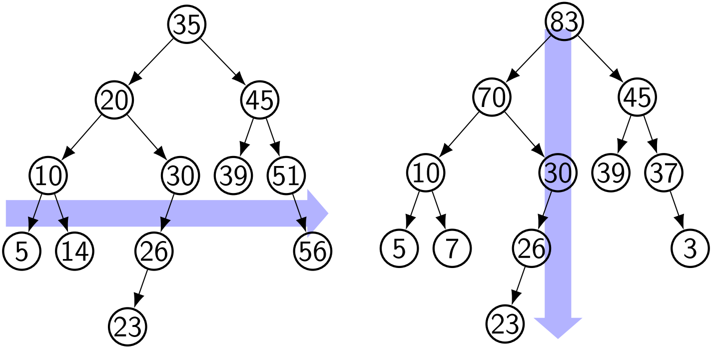




search tree/dictionary
ordered on key
examples: only keys given
(usually numbers)

heap order/priority queue
ordered on priority
examples: only priorities given

ordering keys



search tree

heap order

implementations

worst case complexity

	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

also: Binomial, Weak heap, soft heap, rank-pairing, strict Fibonacci, 2-3-heap, . . .

two prio-queue implementations

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

Contents

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`

binary heap

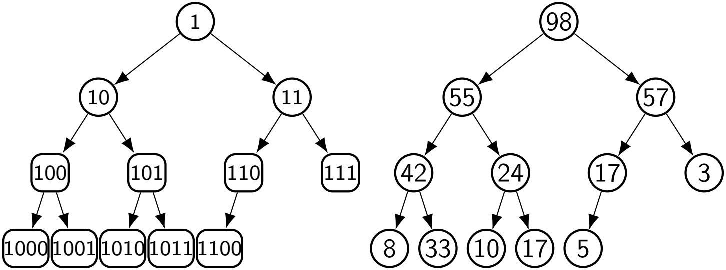
Definition

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

- (structure) *complete* binary tree
- (placement keys) *heap order*

representing binary tree with an array

root at index 1, left/right child i at index $2i/2i{+}1$.



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

works well for *complete binary trees*

waste of space when ‘missing’ nodes

binary heap: three levels

1. *functioning*: abstract (priority queue)
2. *understanding*: binary tree
3. *implementation*: array

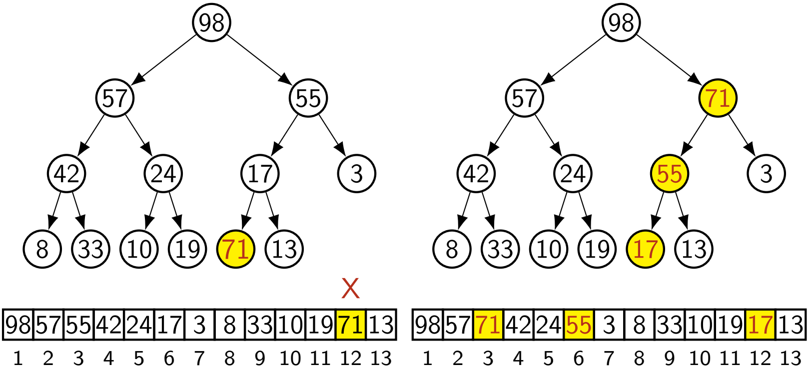
internal operations (change key at position) as binary tree:

- 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).
- 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.

"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*), . . ."
What's in a name? [Wikipedia]

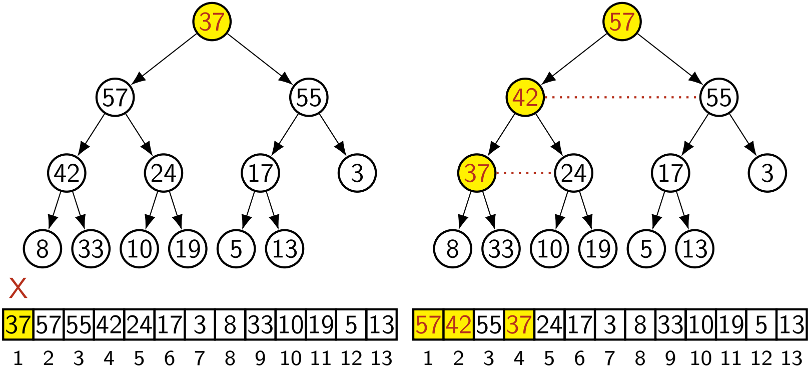
bubble up

swap with parent until heap-ordered



trickle down

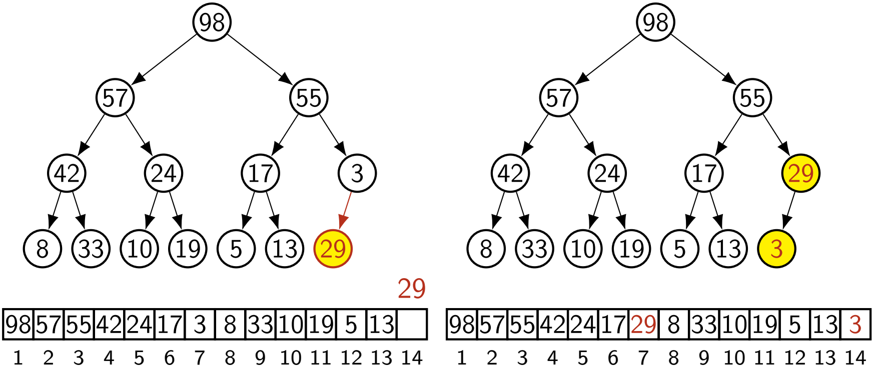
swap with *largest* child until heap-ordered



heap: implementing priority queue operations

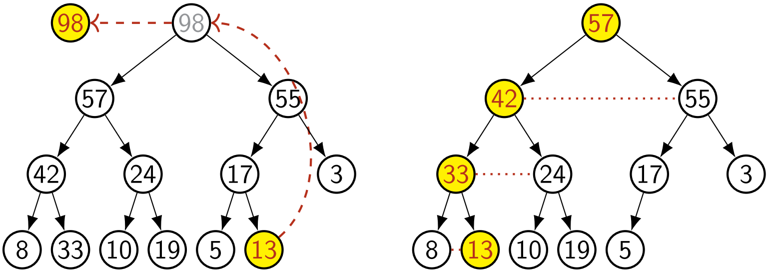
- INSERT: add the new element at the next available position at the complete tree, then BUBBLEUP.
- GETMAX: the maximal element is present at the root of the tree.
- DELETEMAX: replace the root of the tree by the last element (in level order) of the tree. That element can be moved to its proper position using TRICKLEDOWN.
- INCREASEKEY: use BUBBLEUP.

insert 29



insert: add as last element, then BubbleUp

delete max



13 x

98	57	55	42	24	17	3	8	33	10	19	5	13
1	2	3	4	5	6	7	8	9	10	11	12	13

57	42	55	33	24	17	3	8	13	10	19	5
1	2	3	4	5	6	7	8	9	10	11	12

DeleteMax: move last element to first/root, TrickleDown

heapsort

array: heap + partially sorted

– move max heap to sorted part

57	42	55	33	24	17	58	60	67	84	92	97
1	2	3	4	5	6	7	8	9	10	11	12

⟵ max heap ⟶ ⟵ sorted ⟶

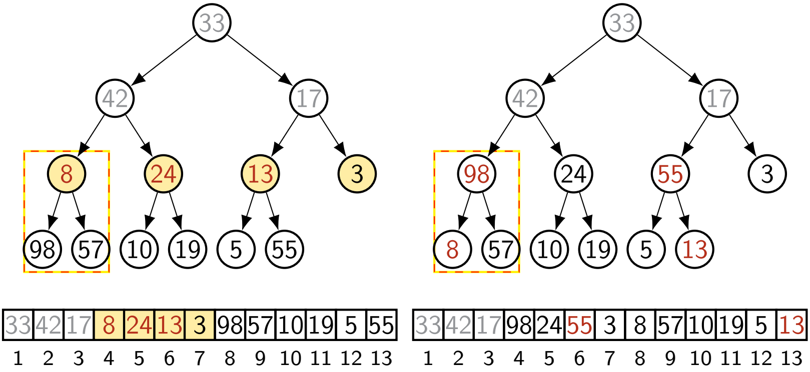
17	42	55	33	24	57	58	60	67	84	92	97
1	2	3	4	5	6	7	8	9	10	11	12

⟵ trickle down ⟶ ⟵ sorted ⟶

– first step: build the heap (linear time)

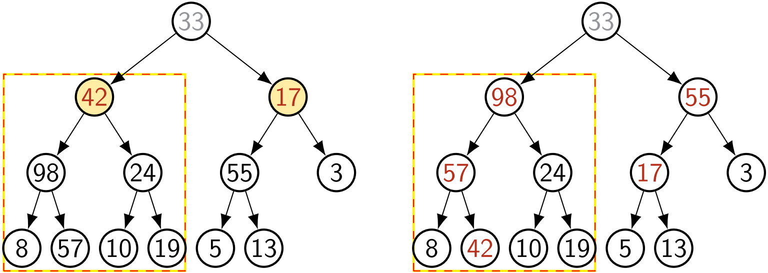
- MAKEHEAP: Given an array, reorder its elements so that the array is a binary heap. aka HEAPIFY

heapify (1)



`TrickleDown` new key: swap with parent until heap-ordered

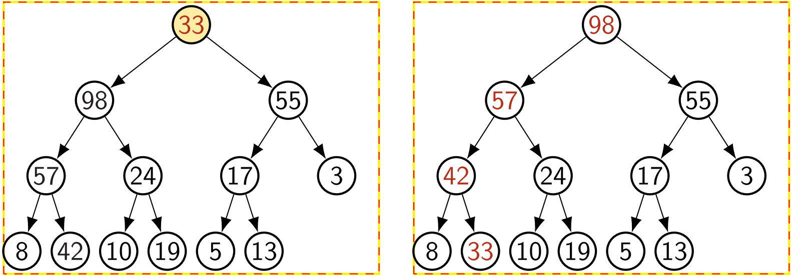
heapify (2)



33	42	17	98	24	55	3	8	57	10	19	5	13
1	2	3	4	5	6	7	8	9	10	11	12	13

33	98	55	57	24	17	3	8	42	10	19	5	13
1	2	3	4	5	6	7	8	9	10	11	12	13

heapify (3)

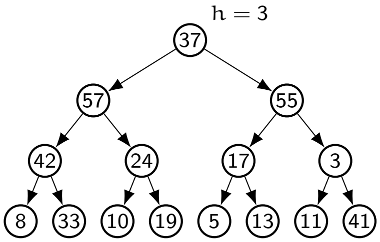


33	98	55	57	24	17	3	8	42	10	19	5	13
1	2	3	4	5	6	7	8	9	10	11	12	13

98	57	55	42	24	17	3	8	33	10	19	5	13
1	2	3	4	5	6	7	8	9	10	11	12	13

complexiteit

		topdn	botup	sum
ℓ	2^ℓ	$2^\ell \cdot \ell$		$2^\ell \cdot h$
0	1	$1 \cdot 0$	$1 \cdot 5$	$1 \cdot 5$
1	2	$2 \cdot 1$	$2 \cdot 4$	$2 \cdot 5$
2	4	$4 \cdot 2$	$4 \cdot 3$	$4 \cdot 5$
3	8	$8 \cdot 3$	$8 \cdot 2$	$8 \cdot 5$
4	16	$16 \cdot 4$	$16 \cdot 1$	$16 \cdot 5$
$h = 5$	32	$32 \cdot 5$	$32 \cdot 0$	$32 \cdot 5$
$\sum$	63	258	57	315
				$63 \cdot 5$



complexity heapify

Lemma

$\sum_{d=0}^{h} 2^d = 2^{h+1} - 1$

$\sum_{d=0}^{h} d\, 2^d = (h-1)2^{h+1} + 2$

L levels, each level 2^ℓ keys, total $n = 2^L - 1$ keys

top-down (fout)

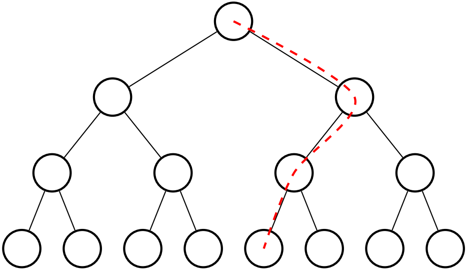
$\sum_{\ell=0}^{L-1} 2^\ell \ell = (L-2)2^L + 2 = n \lg n$ (ongeveer)

bottom-up (goed)

$\sum_{\ell=0}^{L-1} 2^\ell (L-1-\ell) = \sum_{\ell=0}^{L-1} 2^\ell (L-1) - \sum_{\ell=0}^{L-1} 2^\ell \ell = 2^L - L - 1$

which is $\Theta(n)$

visueel bewijs



Contents

leftist heaps

$\mathrm{npl}(x)$ *nil path length*, shortest distance to external leaf

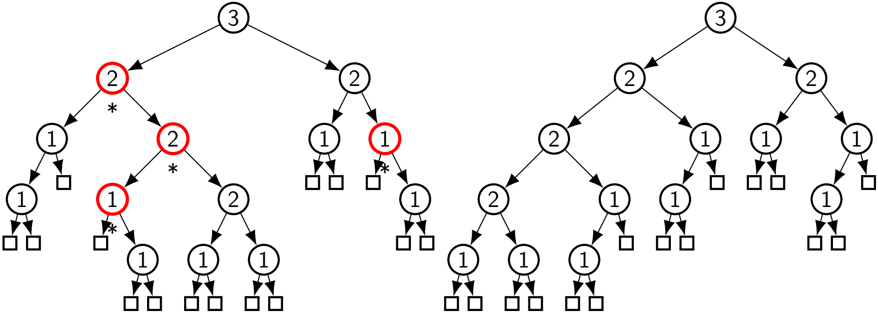
$\mathrm{npl}(x) = 1 + \min\{\ \mathrm{npl}(\mathrm{left}(x)), \mathrm{npl}(\mathrm{right}(x))\ \}$

Definition

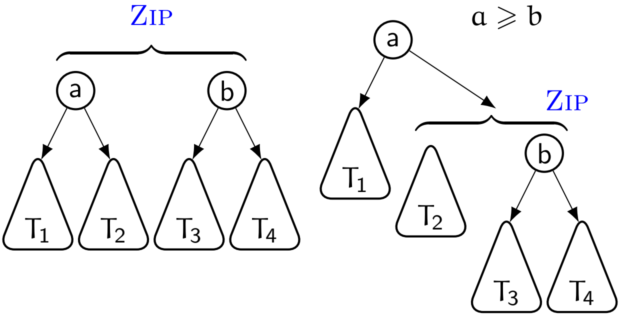
A *leftist tree* is an (extended) binary tree where for each internal node x, $\mathrm{npl}(\mathrm{left}(x)) \geqslant \mathrm{npl}(\mathrm{right}(x))$.
A *leftist heap* is a leftist tree where the priorities satisfy the heap order.

structure vs. node order

leftist tree (structure)

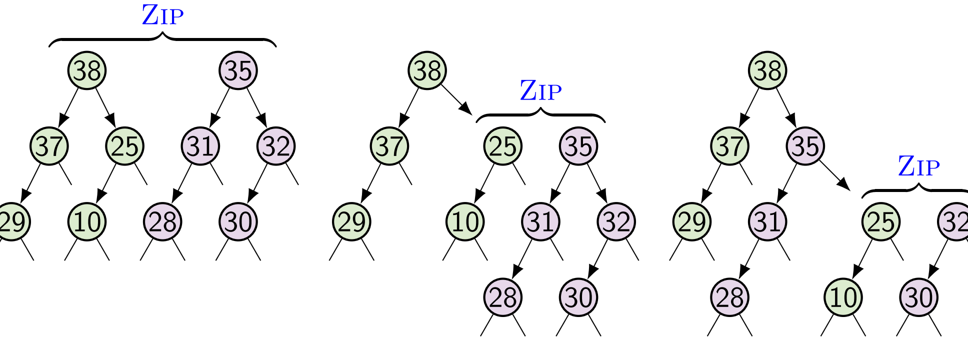


basic operation: Zip

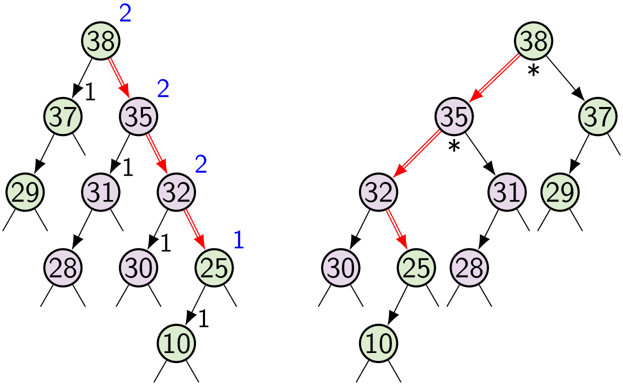


1. combine leftist heaps recursively as shown above
2. swap children at nodes where npl(left(x)) $<$ npl(right(x))

example: (1) zipping trees recursively



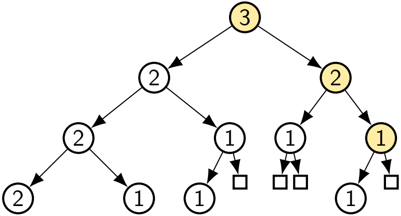
(2) restructuring leftist property (bottom-up)



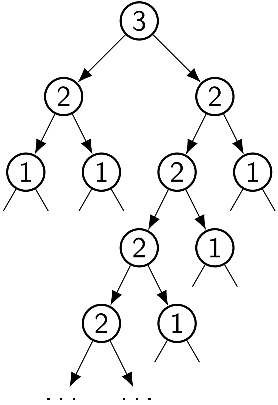
short rightmost path . . .

Lemma

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



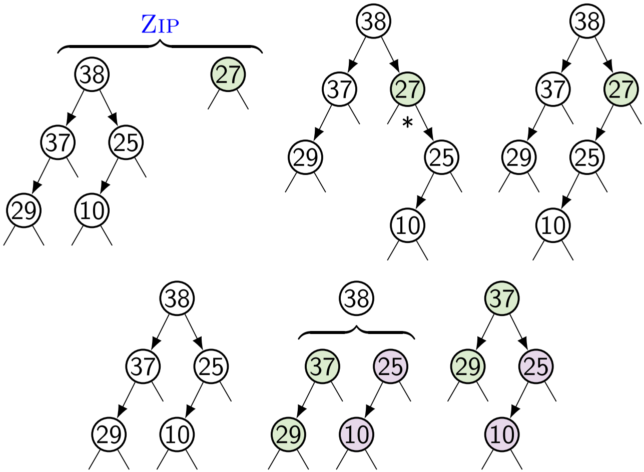
. . . but no bound other paths



priority queue operations

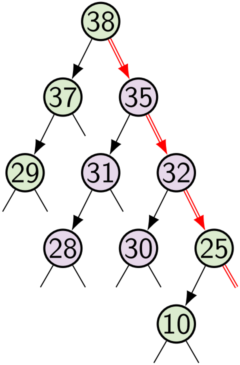
- INSERT: construct a single node tree and ZIP with the original tree.
- GETMAX: the maximal element is present at the root of the tree.
- DELETEMAX: delete the node at the root, ZIP the two subtrees of the root into a new tree.
- MELD: is performed by a ZIP.
- INCREASEKEY: cut the node with its subtree, repair npl remaining tree, ZIP the two trees. (tricky)

example heap operations



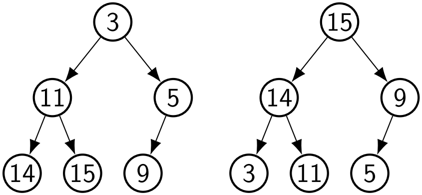
skew heap ⊠

self-adjusting heap. skew merge: *always* swap left and right.



Contents

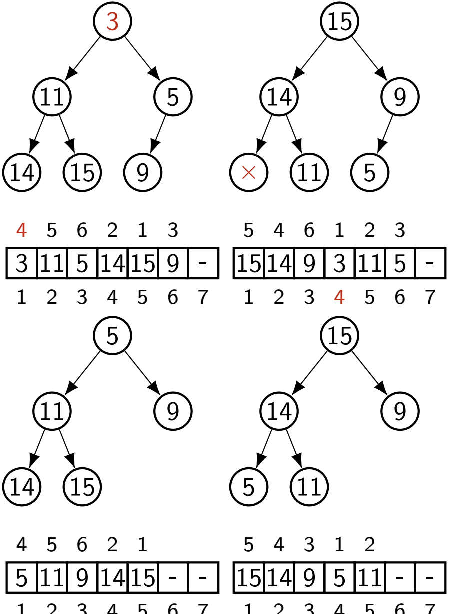
dual structure



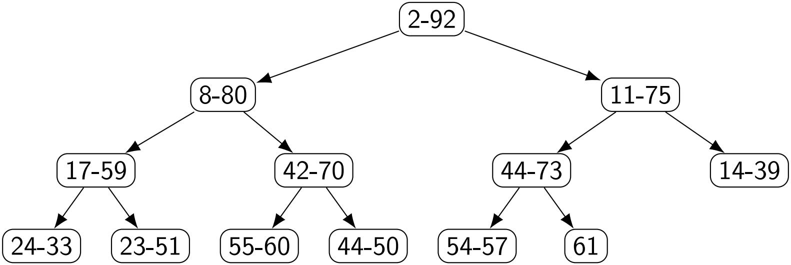
maxpos	4	5	6	2	1	3	
val	3	11	5	14	15	9	-
pos	1	2	3	4	5	6	7

minpos	5	4	6	1	2	3	
val	15	14	9	3	11	5	-
pos	1	2	3	4	5	6	7

- pointer from min-heap item to same item in max-heap
- Insertion: as in ordinary heap, but twice: once in each heap
- Deletion: find item to delete in other heap using pointer, move last element to that position and do BubbleUp

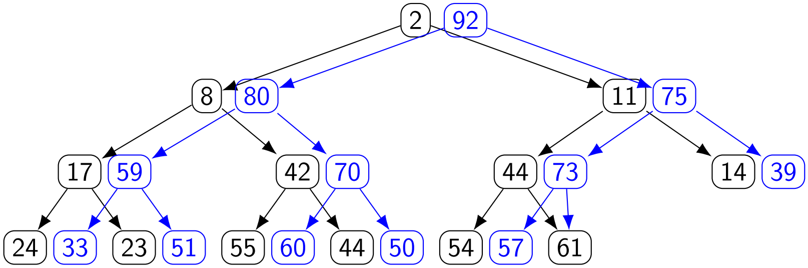

3
11 5
14 15 9
4 5 6 2 1 3
3 11 5 14 15 9 -
1 2 3 4 5 6 7
15
14 9
× 11 5
5 4 6 1 2 3
15 14 9 3 11 5 -
1 2 3 4 5 6 7
5
11 9
14 15
4 5 6 2 1
5 11 9 14 15 - -
1 2 3 4 5 6 7
15
14 9
5 11
5 4 3 1 2
15 14 9 5 11 - -
1 2 3 4 5 6 7

interval heap

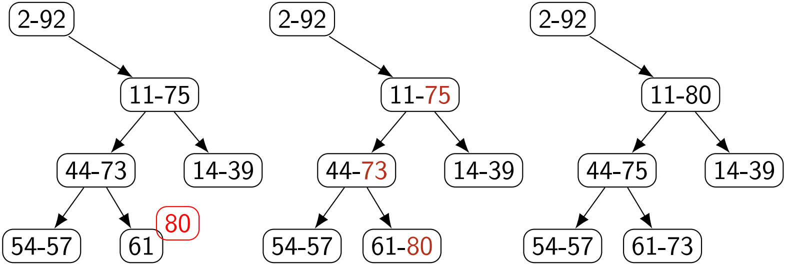


- nodes contain *two* items (minVal, maxVal) “intervals”
- child interval is subset of parent interval $[8, 80] \subseteq [2, 92]$

embedded min&max heap

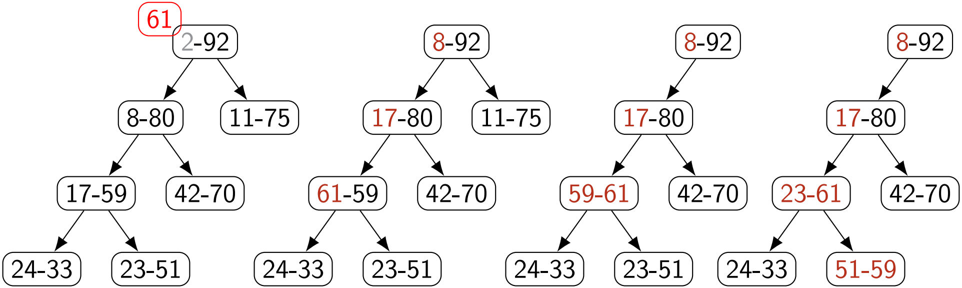


interval heap: insert



- Insert: add key in next position, (if needed) swap to ensure interval.
  Bubble up in min-heap if key smaller than parent's min-key, or in max-heap if key larger than parent's max-key

interval heap: deleteMin



- DeleteMIN: move last element to min-position in root node.
  Trickle down in min-heap and (if needed) swap elements to ensure at each node:
  `node.minVal` $\leqslant$ `node.maxVal`

Double ended priority queue - use case

wikipedia

One example application of the double-ended priority queue is **external sorting**. In an external sort, there are more elements than can be held in the computer's memory.

end.