Chapter 4. Priority queues

Data Structures and Algorithms

FIB

Q2 2018–19

Jordi Delgado
(slides by Antoni Lozano)

Perfect binary trees

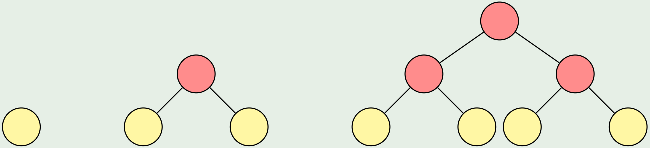
Definition

The level of a node in a tree is the distance from the root to the node.

Definition

A binary tree is perfect if all leaves are at the same level.

Examples



Definition

The height of a tree is the maximum level of its nodes.

Perfect binary trees

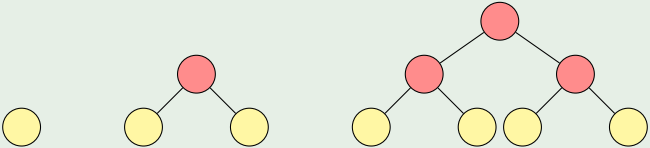
Definition

The level of a node in a tree is the distance from the root to the node.

Definition

A binary tree is perfect if all leaves are at the same level.

Examples



Definition

The height of a tree is the maximum level of its nodes.

Perfect binary trees

Proposition

A perfect binary tree of height h has $2^{h+1} - 1$ nodes.

Proof

Induction on the height. Let T be a perfect binary tree of height h.

- Base case: $h = 0$.
  The tree has a single node, and $1 = 2^{0+1} - 1$.
- Induction step: $h > 0$.
  Left and right subtrees have height $h - 1$ and, by induction hypothesis, they have $2^h - 1$ nodes each. The number of nodes of T is the sum of these nodes plus one (the root):

$$\text{nodes of } T = 2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1.$$

Perfect binary trees

Proposition

A perfect binary tree of height h has $2^{h+1} - 1$ nodes.

Proof

Induction on the height. Let T be a perfect binary tree of height h.

- Base case: $h = 0$.
  The tree has a single node, and $1 = 2^{0+1} - 1$.
- Induction step: $h > 0$.
  Left and right subtrees have height $h - 1$ and, by induction hypothesis, they have $2^h - 1$ nodes each. The number of nodes of T is the sum of these nodes plus one (the root):

$$\text{nodes of } T = 2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1.$$

Complete binary trees

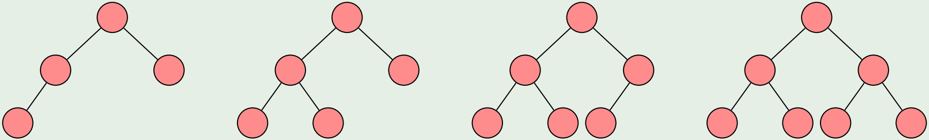
Definition

A binary tree of height h is complete if

1. all leaves are at level $h-1$ or h and
2. the number of leaves of the left subtree of any node is greater or equal than the leaves of the right one.

Hence, a complete binary tree is one which has:
(1) the first $h-1$ levels full and
(2) level h with the leaves as much to the left as possible.

Examples



Complete binary trees

Proposition

A complete binary tree with height h has between 2^h and $2^{h+1} - 1$ nodes.

Proof

Let T be a complete binary tree with height h:

- The minimum number of nodes for T corresponds to having a unique node at height h. Since up to height $h - 1$, T has $2^h - 1$ nodes, adding the only node at height h, we obtain 2^h nodes.
- The maximum number of nodes of T corresponds to a perfect binary tree of height h, which has $2^{h+1} - 1$ nodes.

Complete binary trees

Proposition

A complete binary tree with height h has between 2^h and $2^{h+1} - 1$ nodes.

Proof

Let T be a complete binary tree with height h:

- The minimum number of nodes for T corresponds to having a unique node at height h. Since up to height $h - 1$, T has $2^h - 1$ nodes, adding the only node at height h, we obtain 2^h nodes.
- The maximum number of nodes of T corresponds to a perfect binary tree of height h, which has $2^{h+1} - 1$ nodes.

Complete binary trees

Corollary

The height of a complete binary tree with n nodes is $\lfloor \log n \rfloor \in \Theta(\log n)$.

Proof

By the previous proposition, a complete binary tree of height h and n nodes fulfills:

$$2^h \leq n \leq 2^{h+1} - 1.$$

If we take logarithms en base 2, we have

$$h \leq \log n < h + 1.$$

And taking the root of the logarithm,

$$h = \lfloor \log n \rfloor.$$

Hence, $h \in \Theta(\log n)$.

Complete binary trees

Corollary

The height of a complete binary tree with n nodes is $\lfloor \log n \rfloor \in \Theta(\log n)$.

Proof

By the previous proposition, a complete binary tree of height h and n nodes fulfills:

$$2^h \leq n \leq 2^{h+1} - 1.$$

If we take logarithms en base 2, we have

$$h \leq \log n < h + 1.$$

And taking the root of the logarithm,

$$h = \lfloor \log n \rfloor.$$

Hence, $h \in \Theta(\log n)$.

Introduction

Several applications require to process the input following a partial order given by priorities.

- Task scheduling in computer systems: shorter tasks should be processed before.
- Simulation systems where event should be simulates in chronological order.
- Sorting algorithms. All elements are first inserted and then we iterate by always removing the minimum of the remaining ones.

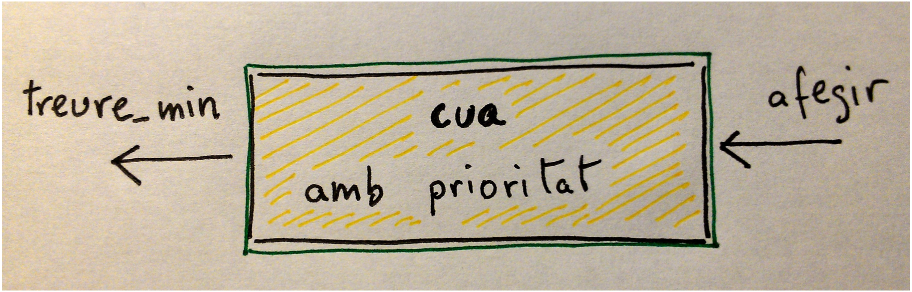
Priority queues are a key ingredient in algorithms design.

Operations

Definition

A priority queue is a data structure that supports two basic operations:

- **add**: add an element (key and information) and
- **remove_min**: remove and return the element with the smallest key.



implementations	**add**	**remove_min**
unordered sequential	$\Theta(1)$	$\Theta(n)$
ordered sequential	$\Theta(n)$	$\Theta(n)$
ord. seq. (decreasing)	$\Theta(n)$	$\Theta(1)$
ordered circular vector	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$

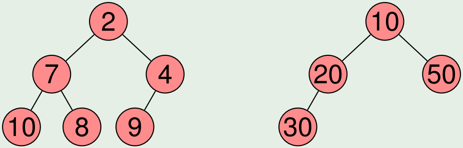
Heaps

Definition

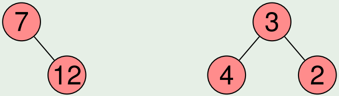
A *min-heap* is a complete binary tree where the key of any node is always smaller than the keys of its children.

Examples

Min-heaps:



Not min-heaps:



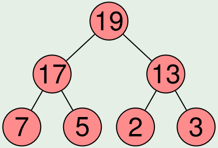
Heaps

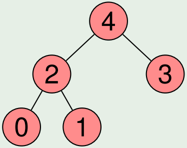
Definition

A *max-heap* is a complete binary tree where the key of any node is always larger than the keys of its children.

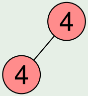
Examples

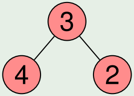
Max-heaps:





Not max-heaps:





Heaps

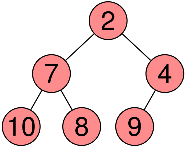
Terminology

- When we say heaps without any further detail, we will refer to min-heaps.

Heaps

Heaps are represented in a compact way using vectors.

The heap



is represented by the vector

	2	7	4	10	8	9		
0	1	2	3	4	5	6	7	8

Pointers are not necessary because:

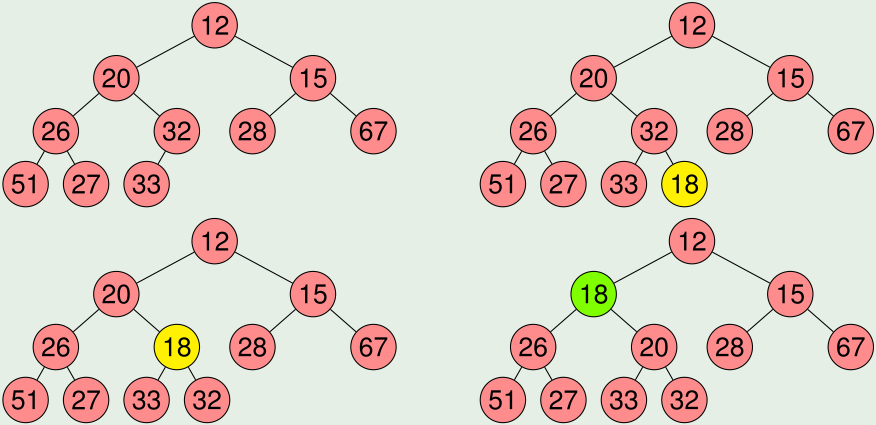
- the father of the node at position i is at position $\lfloor i/2 \rfloor$
- the left child of the node at position i is at position $2i$, the right one at $2i + 1$

Basic operations

Operation **add**

We add the element to the first free position of the vector and move it up until the heap property is satisfied again.

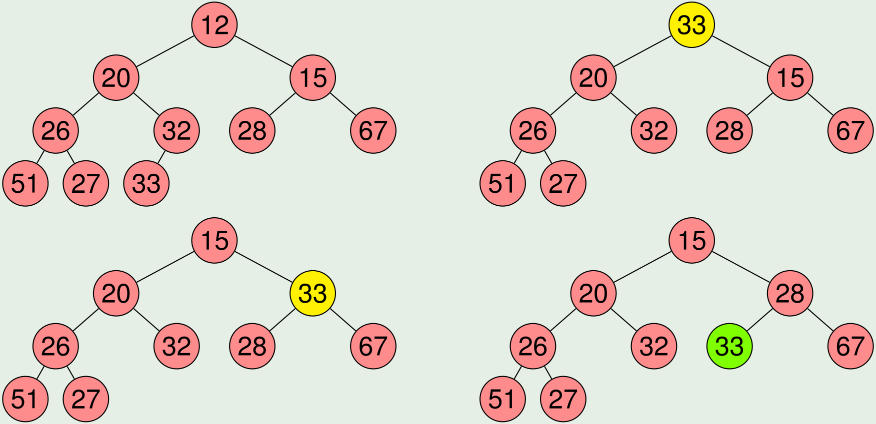
Example (adding key 18)



Operation **remove-min**

The element in the last position is moved to the first one and is moved down until its position is found. The former root is returned.

Example



Recursive implementation

Definition of class `PrioQueue`

```
template <typename Elem>
class PrioQueue {

private:
vector<Elem> t; // Table where the heap is stored
                // (position 0 is not used)
```

Recursive implementation: public functions

Constructor

Creates an empty priority queue. Cost: $\Theta(1)$.

```
PrioQueue () {
    t.push_back(Elem());
}
```

Asking about the size

Returns the size of the priority queue. Cost: $\Theta(1)$.

```
int size () {
    return t.size()-1;
}
```

Recursive implementation: public functions

Emptiness check

Determines whether the priority queue is empty. Cost: $\Theta(1)$.

```
bool empty () {
    return size()==0;
}
```

Return the minimum

Returns the element with minimum priority. Cost: $\Theta(1)$.

```
Elem minimum () {
    if (empty()) throw ErrorPrec("Empty PrioQueue");
    return t[1];
}
```

Recursive implementation: public functions

add

Adds a new element. Cost: $\Theta(\log n)$.

```
void add (Elem& x) {
    t.push_back(x);
    move_up(size());
}
```

Recursive implementation: public functions

remove_min

Removes and returns the minimum element. Cost: $\Theta(1)$.

```
Elem remove_min () {
    if (empty()) throw ErrorPrec("Empty PrioQueue");
    Elem x = t[1];
    t[1] = t.back();
    t.pop_back();
    move_down(1);
    return x;
}
```

Recursive implementation: private functions

move_up

An element is moved up until the heap ordering condition is satisfied. Cost: $\Theta(\log n)$.

```
void move_up (int i) {
    if (i!=1 and t[i/2]>t[i]) {
        swap(t[i],t[i/2]);
        move_up(i/2);
}   }
```

Recursive implementation: private functions

move_down

An element is moved down until the heap ordering condition is satisfied.
Cost: $\Theta(\log n)$.

```
void move_down (int i) {
    int n = size();
    int c = 2*i;
    if (c<=n) {
        if (c+1<=n and t[c+1]<t[c]) c++;
        if (t[i]>t[c]) {
            swap(t[i],t[c]);
            move_down(c);
} } }
```

Iterative implementation

The operations to be changed are **add** and **remove_min**, where **move_down** and **move_up** are now optimized. Asymptotic costs do not change: $\Theta(\log n)$.

add

```
void add (Elem& x) {
    t.push_back(x);
    int i = size();
    while (i!=1 and t[i/2]>x) {
        t[i] = t[i/2];
        i = i/2;
    }
    t[i] = x;
}
```

Iterative implementation

remove_min

```
Elem remove_min () {
    if (empty()) throw ErrorPrec("Empty PrioQueue");
    int n = size();
    Elem e = t[1], x = t[n];
    t.pop_back(); --n;
    int i = 1; c = 2*i;
    while (c<=n) {
        if (c+1<=n and t[c+1]<t[c]) ++c;
        if (x<=t[c]) break;
        t[i] = t[c];
        i = c;
        c = 2*i;
    }
    t[i] = x;
    return e;
}
```

Basic algorithm

Priority queues can be used to sort in time $\Theta(n \log n)$.

The algorithm is called heapsort and was introduced in 1964 by J.W.J. Williams. Given a vector with n elements,

1. the n elements are added to a *heap*: $\Theta(n \log n)$
2. n **remove_min** operations are used to construct a sorted vector: $\Theta(n \log n)$

Total time is $\Theta(n \log n)$, the minimum asymptotic time for a sorting algorithm.

Basic algorithm

Heapsort

With different vectors for the *heap* and input/output.
Time: $\Theta(n \log n)$.
Space: $\approx 2n$.

```
template <typename elem>
void heapsort (vector<elem>& T) {
    PrioQueue<elem> h;
    for (int i=0; i<n; ++i)
        h.add(T[i]);
    for (int i=0; i<n; ++i)
        T[i] = h.remove_min();
}
```

Basic algorithm

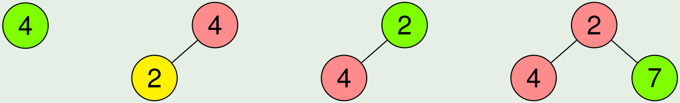
Example

Let us assume that we start with the vector:

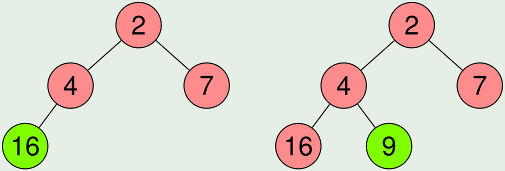
4	2	7	16	9	3	1	5

and we add the elements to the heap, one at a time.

$+4$, $+2$, $+7$:

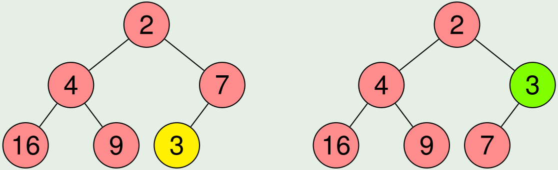


$+16$, $+9$:

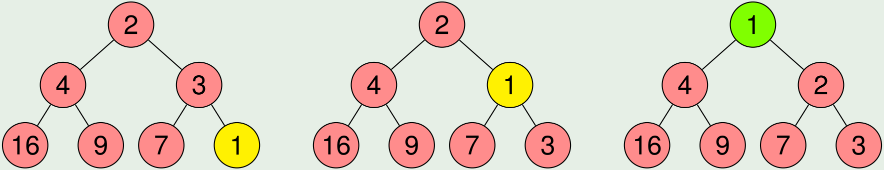


Basic algorithm

$+3$:

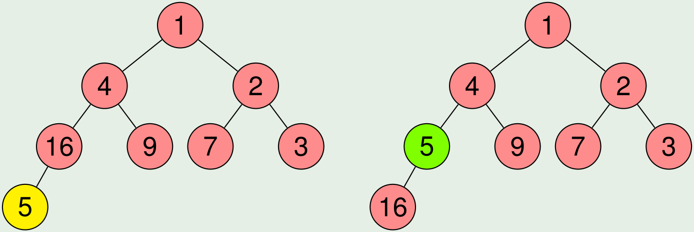


$+1$:



Basic algorithm

+5:

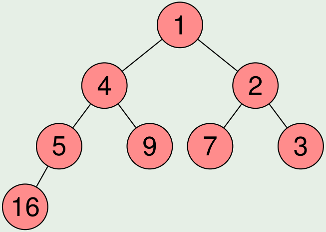


The resulting *heap* is stored in the vector:

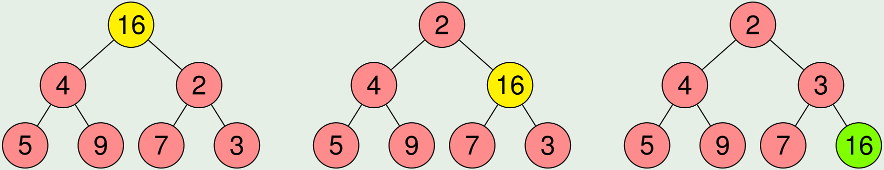
1	4	2	5	9	7	3	16
1	2	3	4	5	6	7	8

We now move the elements in order to the original vector.

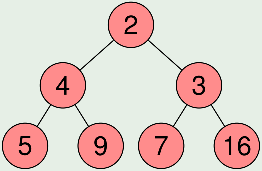
Basic algorithm



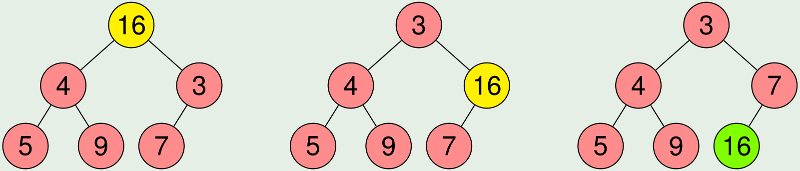
-1:



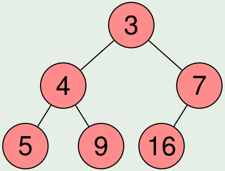
Basic algorithm



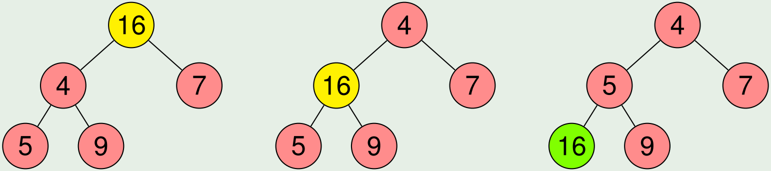
-2:



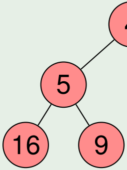
Basic algorithm



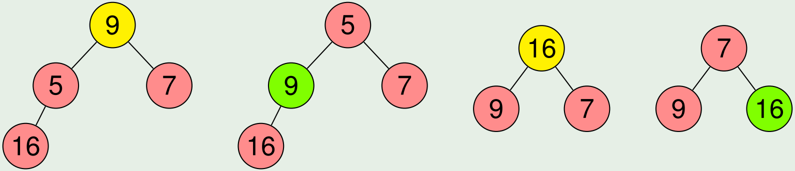
−3:



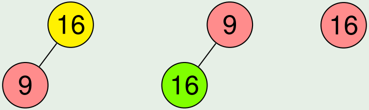
Basic algorithm



−4,−5:



−7,−9, −16:



Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

4	2	7	16	9	3	1	5
1	2	3	4	5	6	7	8

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

	2	7	16	9	3	1	5
1	2	3	4	5	6	7	8
4							

heap

Example: evolution of vectors (operation **add**)

input/output

1	2	3	4	5	6	7	8
		7	16	9	3	1	5

heap

1	2	3	4	5	6	7	8
2	4						

Example: evolution of vectors (operation **add**)

input/output

			16	9	3	1	5
1	2	3	4	5	6	7	8
2	4	7					

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

1	2	3	4	5	6	7	8
				9	3	1	5
2	4	7	16				

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

					3	1	5
1	2	3	4	5	6	7	8
2	4	7	16	9			

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

						1	5
1	2	3	4	5	6	7	8

2	4	3	16	9	7		

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

							5
1	2	3	4	5	6	7	8
1	4	2	16	9	7	3	

heap

Basic algorithm

Example: evolution of vectors (operation **add**)

input/output

1	2	3	4	5	6	7	8
1	4	2	5	9	7	3	16

heap

Basic algorithm

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3	4	5	6	7	8
1	4	2	5	9	7	3	16

heap

Basic algorithm

Example: evolution of vectors (operation **remove_min**)

input/output

1							
1	2	3	4	5	6	7	8
2	4	3	5	9	7	16	

heap

Basic algorithm

Example: evolution of vectors (operation **remove_min**)

input/output

1	2						
1	2	3	4	5	6	7	8
3	4	7	5	9	16		

heap

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3					
1	2	3	4	5	6	7	8

4	5	7	16	9			

heap

Basic algorithm

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3	4				
1	2	3	4	5	6	7	8
5	9	7	16				

heap

Basic algorithm

Example: evolution of vectors (operation remove_min)

input/output

1	2	3	4	5			
1	2	3	4	5	6	7	8
7	9	16					

heap

Basic algorithm

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3	4	5	7		
1	2	3	4	5	6	7	8
9	16						

heap

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3	4	5	7	9	
1	2	3	4	5	6	7	8
16							

heap

Example: evolution of vectors (operation **remove_min**)

input/output

1	2	3	4	5	7	9	16
1	2	3	4	5	6	7	8

heap

Improvements over the basic algorithm

First improvement

Implement the algorithm on a single vector, distinguishing:

- a left part to store the *heap*
- a right part for the input/output

Each time a **remove_min** is called, the minimum is written as the first element of the right part. Elements end up sorted in a decreasing way.

If we want them sorted increasingly, a max-heap can be used.

Improvements over the basic algorithm

Second improvement

Construct the heap in time $\Theta(n)$ instead of $\Theta(n \log n)$ following the steps:

1. Add the elements to the heap in whatever order (and lineal time).
2. If the heap has h levels, for $i = h - 1, h - 2, \ldots, 1$:
    - **move_down** all elements of level i

The fact that most treated subheaps are small makes the number of swaps needed by **move_down** a lineal amount.

Improvements over the basic algorithm

Example

For a heap with 127 nodes, there are

- 32 heaps of size 3
- 16 heaps of size 7
- 8 heaps of size 15
- 4 heaps of size 31
- 2 heaps of size 63
- 1 heap of size 127

Swaps in perfect trees

Number of key swaps then when the tree is perfect with $n = 2^h - 1$ nodes:

$$\sum_{1 \le i < h} 2^{h-i-1} \cdot i = 2^h - h - 1 < n.$$

(For complete trees, the same bound can be proved.)

The selection problem

Selection problem

Given a list S of natural numbers and $k \in \mathbb{N}$, find out the k-th smallest element in S.

Using heaps, we can find a new algorithm:

1. Construct a min-heap from S. $\Theta(n)$
2. Perform k **remove_min** operations to the min-heap. $\Theta(k \log n)$
3. Return the last extracted element. $\Theta(1)$

Total cost: $\Theta(n + k \log n)$.

The median corresponds to $k = n/2$. Cost: $\Theta(n \log n)$.
When $k = n/\log n$, cost is $\Theta(n)$.

The selection problem

Selection problem

Given a list S of natural numbers and $k \in \mathbb{N}$, find out the k-th smallest element in S.

Using heaps, we can find a new algorithm:

1. Construct a min-heap from S. $\Theta(n)$
2. Perform k **remove_min** operations to the min-heap. $\Theta(k \log n)$
3. Return the last extracted element. $\Theta(1)$

Total cost: $\Theta(n + k \log n)$.

The median corresponds to $k = n/2$. Cost: $\Theta(n \log n)$.

When $k = n/\log n$, cost is $\Theta(n)$.