

# Set 07: Priority Queue ADT, and Heap DS

CS240: Data Structures and Data Management

Jérémy Barbay

# Outline

## Priority Queue ADT

Abstract Data Type

Data Structure: Heaps

## Heaps implemented in array

Binary Trees in Array

New operators

Heapify

## Heap Sort

Sorting with Priority queues

Sorting with a Heap implemented in an array

## Mid-Summary about Abstract Data Types

# Min (or Max) Priority Queue ADT

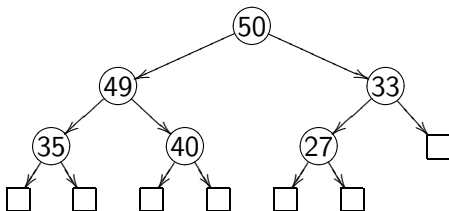
- ▶ Container of **prioritized** elements called **keys**
- ▶ supporting two operations:
  - ▶ **insert( $x$ )**: Inserts key  $x$  into the data structure.
  - ▶ **extractMin()**: (or **extractMax**) Returns the smallest (largest) key and removes it from the data structure.
  - ▶ **isEmpty()**: Returns false if the queue contains at least one key.
- ▶ In practice, an element is associated to each key.
- ▶ Here, we assume keys are distinct and totally ordered.

# Min-Heap and Max-Heap

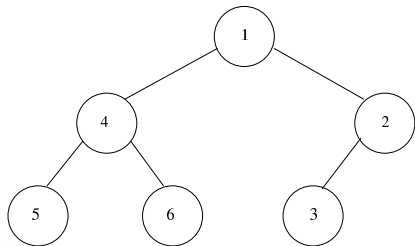
A min-heap (resp. max-heap) is a **data structure** that implements the **abstract data type** priority queue in a tree such that:

1. Value of key  $x$  is smaller (resp. larger) than the value of its descendants.
2. All levels but the last are complete.
3. Last level is filled from left to right.

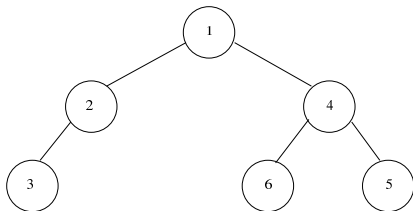
Example:



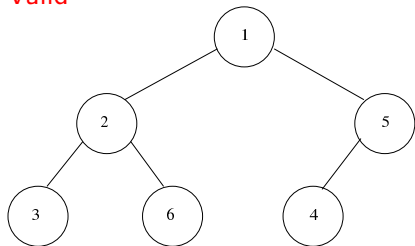
## Examples:



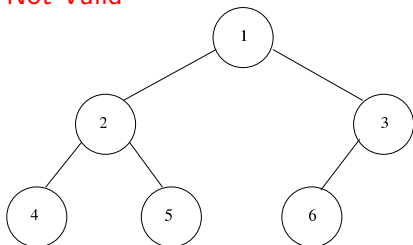
Valid



Not Valid



Not Valid



Valid

# Heap Properties

## Theorem

*A heap with  $n$  internal nodes has height*

$$h = \lceil \lg(n + 1) \rceil.$$

**Proof:** Note,  $x \leq \lceil x \rceil < x + 1$

## Corollary

*A heap with  $n$  internal nodes has height  $h \in \Theta(\log n)$ .*

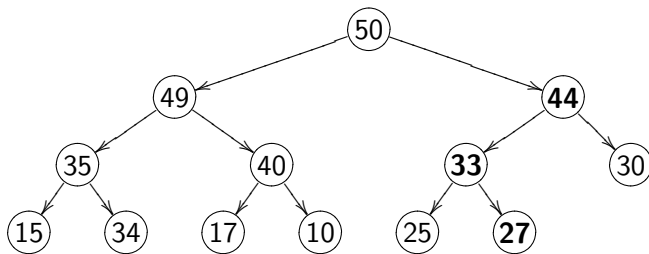
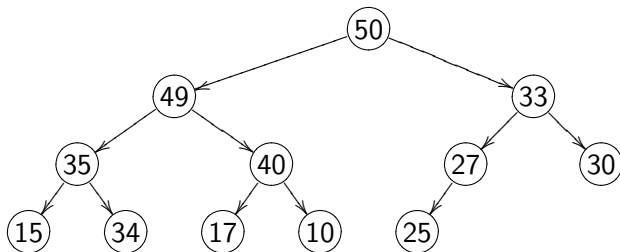
## Inserting element into heap.

- ▶ Add the element to the left-most empty position at the bottom level (or start a new level if the bottom level is full)
- ▶ This may lead to violation of the heap property
- ▶ “Sift the element up” to restore the heap property

### SiftUp( $v$ ) for a MaxHeap

```
if  $v$  has a parent then  
    if the value of  $v$ 's parent  $<$   $v$ 's value then  
        exchange their values;  
        SiftUp(parent( $v$ ));  
    end if  
end if
```

Example: insert(44)



How many swaps?  $O(\lg n)$



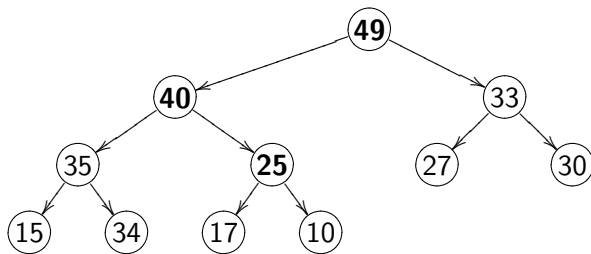
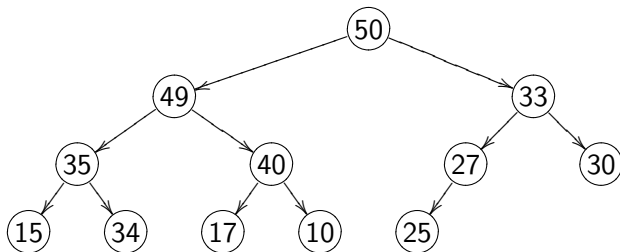
## Extracting the minimum element from the heap.

- ▶ Minimum is stored in the root of the tree: remove it.
- ▶ Fill the gap with the right-most element at the bottom level.
- ▶ Restore the heap property by “sinking” the element down.

### SiftDown( $v$ ) for a MaxHeap

```
if  $v$  has at least a child then  
  if a child of  $v$  has a larger value then  
    find the child  $w$  with the largest value;  
    exchange the value of  $v$  and  $w$  ;  
    SiftDown( $w$ );  
  end if  
end if
```

Example: `extractMax()`



How many swaps?  $O(\lg n)$ .

# Optimisation

- ▶ SiftUp and SiftDown are recursive but particular:
- ▶ only one recursive call in the function.
- ▶ Remove the recursion with a `while` loop.

Other optimisation: implement the binary tree in an array...

# Summary for Heaps

- ▶ Priority Queue is an **Abstract Data Type**
- ▶ Heap is a Data Structure based on binary trees.
- ▶ Recursive function can be **derecursived**.

# Outline

## Priority Queue ADT

Abstract Data Type

Data Structure: Heaps

## Heaps implemented in array

Binary Trees in Array

New operators

Heapify

## Heap Sort

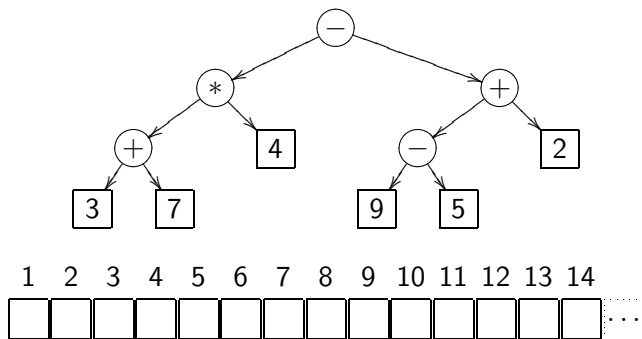
Sorting with Priority queues

Sorting with a Heap implemented in an array

## Mid-Summary about Abstract Data Types

## Binary Trees in Array

- ▶ We can represent a binary tree with an array



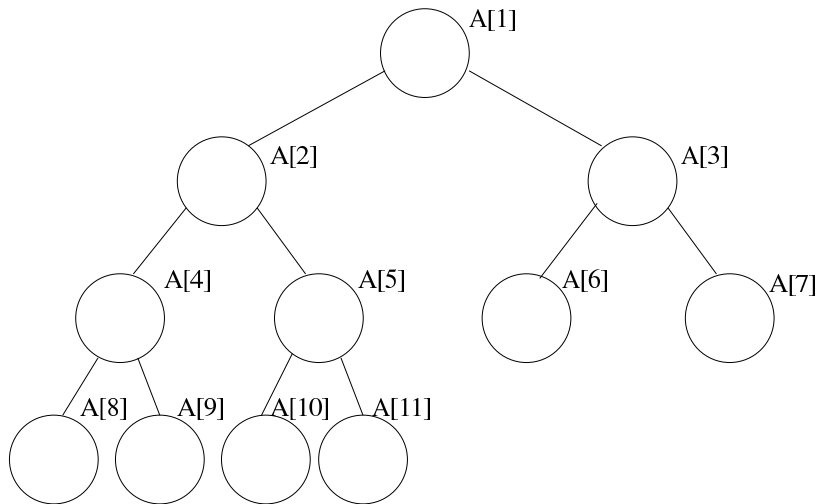
- ▶ Given an array index  $i$ , what is the index of:
  - ▶ left child:  $2i$
  - ▶ right child:  $2i + 1$
  - ▶ parent:  $\lfloor i/2 \rfloor$

## Binary Trees in Array (cont')

- ▶ Why do we not generally represent trees with arrays?
  - ▶ Only efficient for Complete Trees.
  - ▶ Only practical for trees of bounded degrees.
- ▶ A heap is a complete tree, though
- ▶ We will usually draw heaps as a tree structure
- ▶ However, we will implement the heap with
  - ▶ An array  $A[1..N]$
  - ▶ An integer representing the *size* of the current heap

## Implement heaps in an array.

Store root in  $A[1]$  and continue with elements level-by-level from top to bottom, in each level left-to-right:





# SiftDown

To Extract, use SiftDown:

$\text{SiftDown}(A[1..n], i)$

```
if  $i$  is not a leaf, and  $A[i] < \max \{ A[2i], A[2i + 1] \}$  then  
    swap ( $i, c$ ) where  $A[c] = \max \{ A[2i], A[2i + 1] \}$   
    SiftDown( $c$ )  
end if
```

Complexity:  $O(\lg n)$

# SiftUp

To Insert, use SiftUp:

**SiftUp**( $A[1..n]$ ,  $i$ )

```
if  $i$  is not the root, and  $A[i] < \text{key}(\lfloor i/2 \rfloor)$  then  
    swap ( $i$ ,  $\lfloor i/2 \rfloor$ )  
    SiftUp( $i$ )  
end if
```

Complexity:  $O(\lg n)$  To build a heap from  $n$  elements, use Insert  $n$  times?

It costs  $O(n \lg n)$ .

# Heapify

We can build a heap faster than by  $n$  insertions.

**BottomUpHeapify**( $A[1..n]$ )

```
for  $i = \lfloor \frac{n}{2} \rfloor$  down to 1 do
    SiftDown from  $A[i]$ 
end for
```

## Theorem

*The complexity of*  
**BottomUpHeapify** is  $O(n)$

## Example:

Heapify this array in a MaxHeap:

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

As an exercise, Heapify it as a MinHeap.

You should get

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

# Proof of the Complexity of BottomUpHeapify

Proof.

- ▶ In a heap of  $n$  keys, at most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$ .
- ▶ Each call to SiftDown on a subtree of height  $h$  is taking at most  $2h$  comparisons.
- ▶ BottomUpHeapify performs at most one SiftDown call per sub-tree

Hence the total complexity of BottomUpHeapify is at most

$$\begin{aligned} C(n) &\leq \sum_{h=0}^{\lfloor \lg n \rfloor} 2h \lceil \frac{n}{2^{h+1}} \rceil \\ &= 2n \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{h}{2^{h+1}} \rceil \\ &\leq 2n \sum_{h=0}^{\infty} \lceil \frac{h}{2^{h+1}} \rceil < 4n \in O(n) \end{aligned}$$

# How NOT to Heapify

Use SiftDown, and **not SiftUp!**

TopDownHeapify( $A[1..n]$ )

```
for  $i \leftarrow 1$  to  $n$  do  
    SIFTUP from  $A[i]$   
end for
```

Theorem

The complexity of  
TopDownHeapify is  $O(n \lg n)$

Exercise:

Heapify this array:

1	2	3	4	
1	9	3	6	...
				...
				...
				...
				...

## The difference between TopDown and BottomUp

- ▶ **TopDown** has a few SiftUp calls of complexity  $O(1)$  and many of complexity  $O(\lg n)$ ;
- ▶ **BottomUp** has a few SiftDown calls of complexity  $O(\lg n)$  and many of complexity  $O(1)$ .

## Summary for Heaps in arrays

Heaps implemented in arrays:

- ▶ use **less space**
- ▶ and can be build faster, using BottomUpHeapify.
- ▶ Do **not** use the other method!.

# Outline

## Priority Queue ADT

- Abstract Data Type

- Data Structure: Heaps

## Heaps implemented in array

- Binary Trees in Array

- New operators

- Heapify

## Heap Sort

- Sorting with Priority queues

- Sorting with a Heap implemented in an array

## Mid-Summary about Abstract Data Types



## Sorting with Priority queues

```
PQ-Sort( $A[1..n]$ )  
for  $i \leftarrow 1$  to  $n$  do  
    PQ.INSERT(  $A[i]$  )  
end for  
for  $i \leftarrow n$  downto 1 do  
     $A[i] \leftarrow$  PQ.EXTRACTMAX()  
end for
```

# Heap Sort

```
HeapSort( $A[1..n]$ )  
for  $i \leftarrow 1$  to  $n$  do  
    HEAP.INSERT(  $A[i]$  )  
end for  
for  $i \leftarrow n$  downto  $1$  do  
     $A[i] \leftarrow$  HEAP.EXTRACTMAX()  
end for
```

- ▶ Running Time?  $O(n \lg n)$
- ▶ Space Usage?  $O(n)$

# HeapSort and MaxHeaps

On arrays, several modifications to improve the space:

- ▶ Sort “in place”. At step  $i$ :
  - ▶ the last  $i$  elements are sorted.
  - ▶ the first  $n - i$  elements represent the heap.
- ▶ MaxHeap versus MinHeap.

## HeapSort “in place”

HeapSort:

```
for i:=n/2 downto 1
| SiftUp(i,n)
for i:=n downto 1
| Swap(A[1],A[i]);
| SiftUp(1,i-1);
```

SiftUp(node,size):

```
while (2*node<=size and A[node]<A[2*node])
    or (2*node+1<=size and A[node]<A[2*node+1])
| if 2*node+1>size or A[2*node]>A[2*node+1]
| | k:=2*node
| else
| | k:=2*node+1
| swap(A[node],A[k]); node:=k
```

# Summary for HeapSort

Heaps implemented in arrays permits to

- ▶ sort in time  $O(n \lg n)$ .
- ▶ with **space exactly  $n$** .

# Outline

## Priority Queue ADT

- Abstract Data Type

- Data Structure: Heaps

## Heaps implemented in array

- Binary Trees in Array

- New operators

- Heapify

## Heap Sort

- Sorting with Priority queues

- Sorting with a Heap implemented in an array

## Mid-Summary about Abstract Data Types

## Mid-Summary about Abstract Data Types

	Topic	Concept(s)
5	What's an ADT? Stacks (LIFO) Queue (FIFO) Graphs Adjacency list DS Adjacency matrix DS Algorithms on graphs	Abstract the "What" of the "How". Pile of plates People waiting in line (non)oriented, (un)weighted $O(m)$ , good for sparse graphs. $O(1)$ or $O(n)$ , good for dense graphs. Depth First and Breadth First Traversal.
6	Trees Binary Trees	Cardinal/Ordinal, many variants... Can represent other trees.
7	Priority Queue ADT Heap DS Sift up/down Heapify  Heapsort	Find Min or Max. Based on Binary Tree. To correct the heap. To build a heap from an array. do it BottomUp! Sort in $O(n \lg n)$ , and in place.

## Reading Materials

	Topic	GT	CLRS
5	What's an ADT? Graphs	56-74 288-306, 313-316	200-209 527-552
6	Trees Binary Trees	75-93 (same)	214-216 (same)
7	Priority Queue and Heaps	94-112	127-140

- ▶ GT = Algorithm Design, by Goodrich & Tamassia
- ▶ CLRS = Introduction to Algorithms, by Cormen, Leiserson, Rivest & Stein