Tree data structures

Guillaume Derval

April 16, 2016

Table of Contents

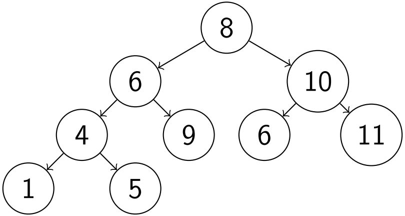
Table of Contents

Definition

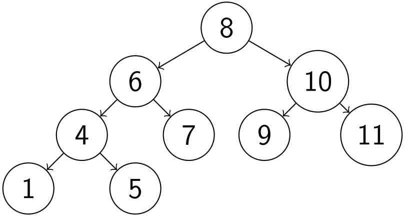
A binary search tree ...

- is a tree (:O)
- is binary (:O). So, two children maximum, that we name "left" and "right"
- such that each nodes stores a "key" and a "value" (which can be the same as the key)
- respects the "search property":
  - all nodes on the left subtree are $<$ the current node
  - all nodes on the right subtree are $>$ the current node

Invalid BST example



BST example



Balanced Search Tree

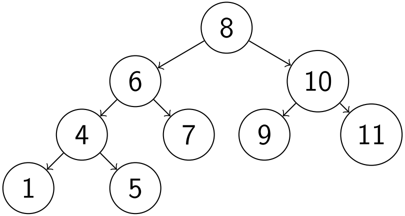
A (binary) tree is said to be balanced if its height is minimal, that is if its height is $\lfloor \log_2(n) \rfloor$
Most the of the BST are balanced. We will see later why...

Common operations

- search(key): returns the value associated with the key
- insert(key, value): insert a new key/value
- findMax(): returns the key/value associated with the biggest key
- findMin(): ...
- successor(key): returns the key/value immediately after the given key
- predecessor(key): ...

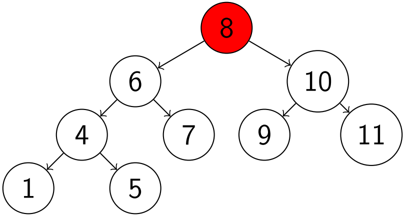
Implementing search operations

Let's say we want to find the value associated with 5 in this tree:



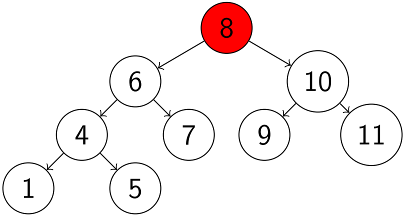
Implementing search operations

Let's start at the root:



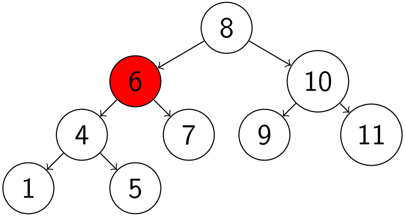
Implementing search operations

Let's start at the root:
$8 > 5$: search only on the left



Implementing search operations

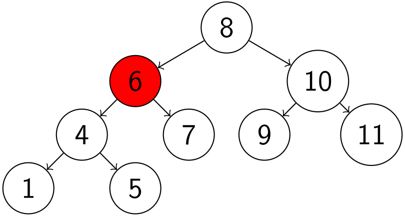
We are now at 6



Implementing search operations

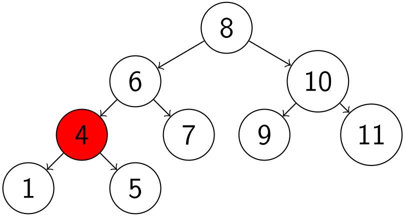
We are now at 6

$6 > 5$: search only on the left



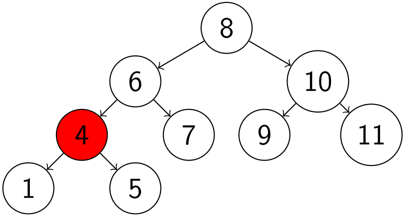
Implementing search operations

We are now at 4



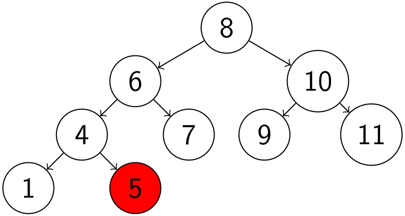
Implementing search operations

We are now at 4
4 $<$ 5: search only on the right



Implementing search operations

We are now at 5. Return the key/value pair.



Complexity?

Complexity?

$O(\text{height of the tree})$

Complexity?

$O(\text{height of the tree}) = O(\lfloor \log(n) \rfloor)$ if the tree is balanced

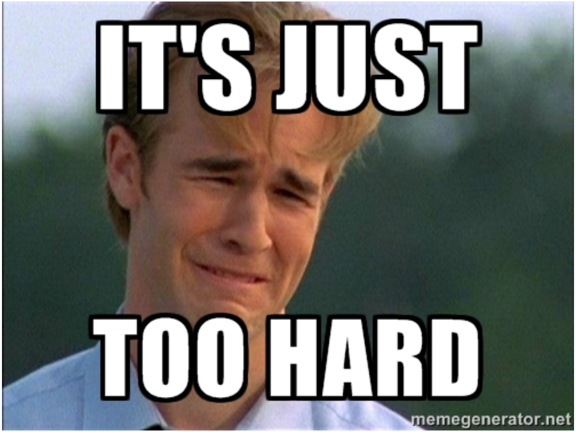
Complexity?

$O(\text{height of the tree}) = O(\lfloor \log(n) \rfloor)$ if the tree is balanced
Without a balanced tree, all the operations are $O(n)$!

Implementing a balanced BST

Ideas?

Implementing a balanced BST



Implementing a balanced BST

Implementing a balanced BST

- Too complicated to explain right now

Implementing a balanced BST

- Too complicated to explain right now
- Too complicated to remember

Implementing a balanced BST

- Too complicated to explain right now
- Too complicated to remember
- Too long to implement during a contest

Implementing a balanced BST

- Too complicated to explain right now
- Too complicated to remember
- Too long to implement during a contest
- Very prone to bugs

Still, if you have time, it's very interesting to know, but won't be useful during competitive programming contests

Use the STL

(real) languages come with implementations of BST, in two categories

- Sets: represents a mathematical set. It is, in fact, a BST with keys==values
  - C++: std::set
  - Java: TreeSet
- Maps: dictionnary
  - C++: std::map
  - Java: TreeMap

Table of Contents

Heap

A heap is a structure with (mainly) two operations:

- `push`: Add an element to the heap $O(\log n)$.
- `pop`: remove and return the biggest/greater/... element from the heap $O(\log n)$.

Heap

A heap is a structure with (mainly) two operations:

- `push`: Add an element to the heap $O(\log n)$.
- `pop`: remove and return the biggest/greater/... element from the heap $O(\log n)$.

How to implement it?

Heap

A heap is a structure with (mainly) two operations:

- `push`: Add an element to the heap $O(\log n)$.
- `pop`: remove and return the biggest/greater/... element from the heap $O(\log n)$.

How to implement it? With a tree, of course!

The heap property

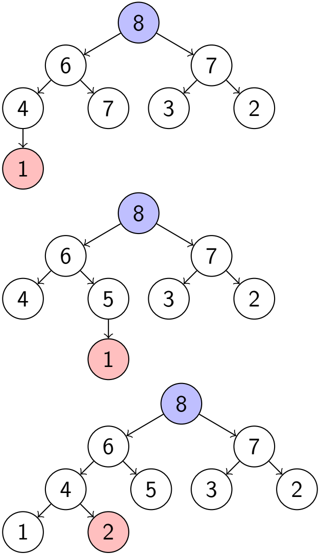
Remember the binary search tree property?

(max) Heap property: Childrens of a node in a heap tree are $\leq$ than the node itself.

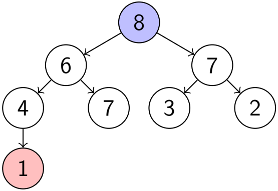
One of the most simple and efficient heap implementation are complete binary heap. Two more rules then:

- The tree is a binary one ($\leq$ 2 children per node)
- The tree is "complete": all levels of the tree are full of nodes but the last one, on which leaves are on the left

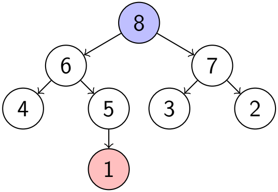
Which are valid complete binary heap trees?



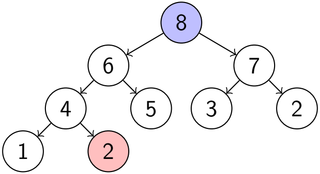
Which are valid complete binary heap trees?



(heap property not respected on 6-7)



(leaf 1 not on left)

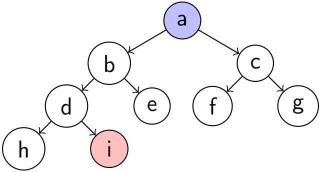


(valid!)

Storing a complete tree

Storing a complete tree is easy: use an array (or a vector)!

0	1	2	3	4	5	6	7	8
a	b	c	d	e	f	g	h	i



If p is the idx of the parent in the array, idx of the children are:

- $2p + 1$
- $2p + 2$

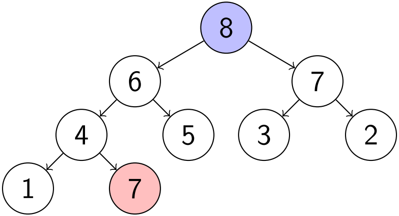
Next node to be created is always at the end of the array!

Adding a new element

1. Add new element at the end of the tree
2. If parent does not respect the heap property (== is lesser than the new node)
   2.1 Exchange the node and its parent
   2.2 Repeat from 2.

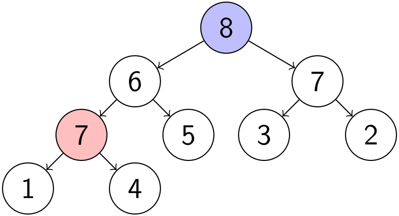
Adding a new element

Adding 7:



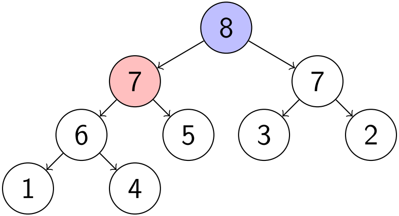
Adding a new element

Adding 7:



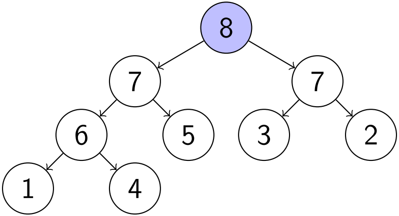
Adding a new element

Adding 7:



Adding a new element

Adding 7:

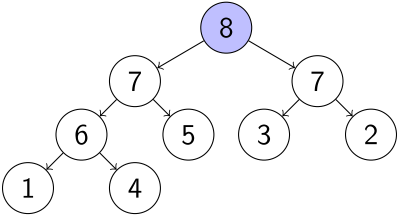


Removing the first element

1. Save somewhere the root of the tree to return it later
2. Take the last node value, and put it at the top of the tree
3. Three cases then:
   - If the heap property is respected (node $\geq$ its children), return
   - If one of the children is $>$ the node, swap them and repeat from 3.
   - If both children are $>$ the node, swap with the greatest child and repeat from 3.

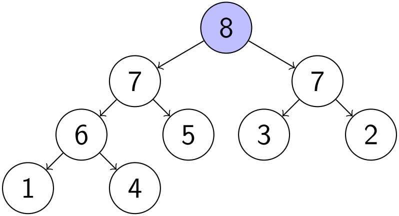
Removing the first element

Step 1: store the root somewhere (remember: root was 8)



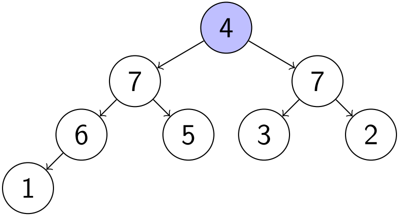
Removing the first element

Step 2: put the last node at the root



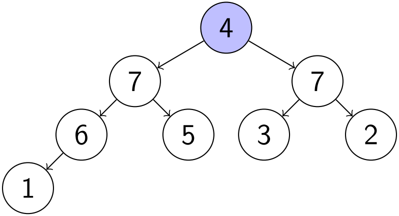
Removing the first element

Step 2: put the last node at the root



Removing the first element

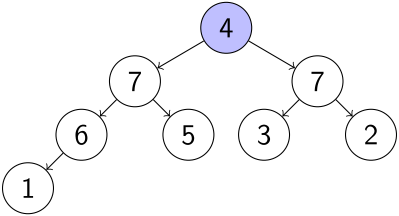
Step 3: check heap property with the children of the node.



Removing the first element

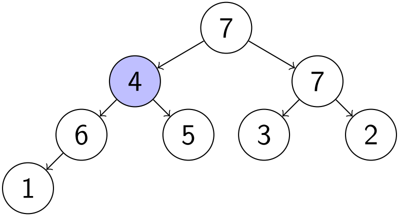
Step 3: check heap property with the children of the node.

Not respected -> swap with (the first) 7



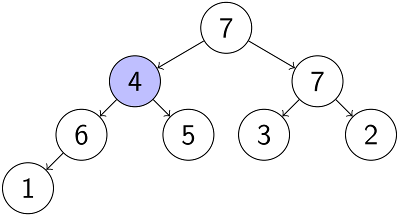
Removing the first element

Step 3: check heap property with the children of the node.
Not respected -> swap with (the first) 7



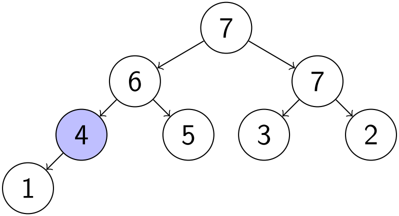
Removing the first element

Step 3: check heap property with the children of the node.
Not respected -> swap with 6 (the greatest child)



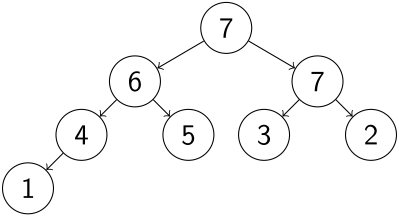
Removing the first element

Step 3: check heap property with the children of the node.
Not respected -> swap with 6 (the greatest child)



Removing the first element

Step 3: check heap property with the children of the node.
Done



Usage

- Priority queues
- Dijkstra
- Sorting
- ...

- In C++: std::priority_queue
- In Java: PriorityQueue