Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 10

Trees Everywhere

Binary Search Trees: time complexity

BSTs are a data structure for storing elements (in a sorted way).

Its basic operations are: add, search and remove an element.

How efficient are they? We start by comparing to other alternatives:

- For the same purpose, we could have used array lists
- Let n be the size of the array list, i.e. its number of elements

array and count	
unordered	*ordered*
add and element, i.e. append, is O(1) (most times)	add an element: - find its position is O($\log n$) - insert it is O(n)
searching an element is O(n)	search an element is O($\log n$) (binary search)
removing an element is O(n)	as with add, this takes O(n) overall

linked list (unordered)
add and element is O(1) (add at head position)
searching an element is O(n)
removing an element is O(n)

Binary Search Trees: time complexity

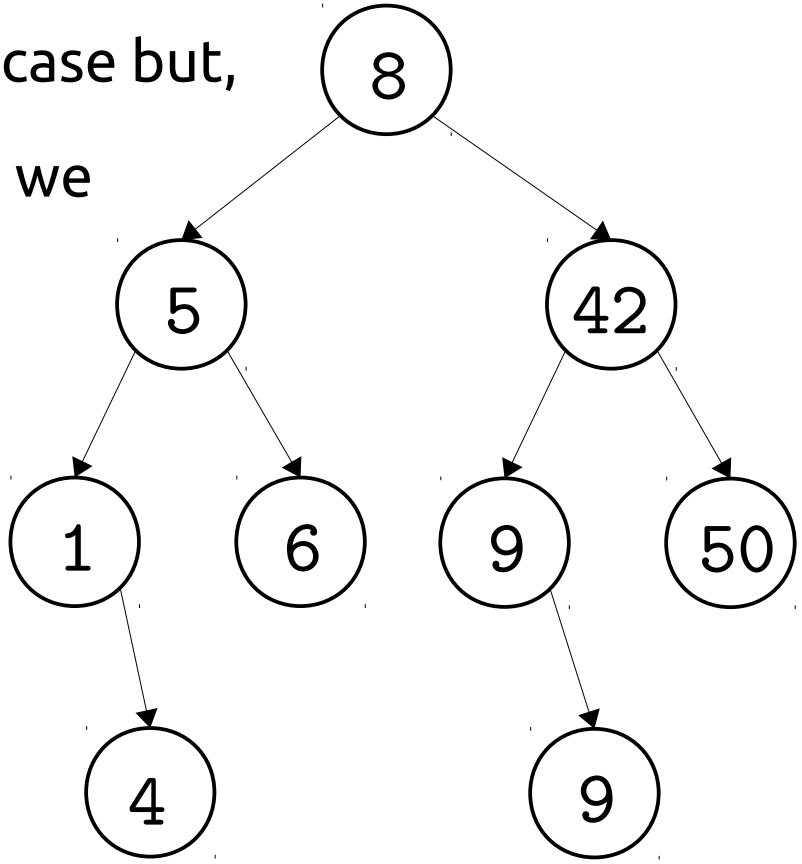
What is the time complexity of adding, searching and removing from a BST with respect to its size n ?

- we will see that these are Θ(n) in the worst case but,
- if the tree is reasonable (i.e. *balanced*) then we go down to Θ($\log n$)

It is easier to do our analysis with respect to the height h of the BST (and work with n later)

Recall that the height of the BST is the maximum level of its leaves.

E.g. this tree has height 3.



Adding an element

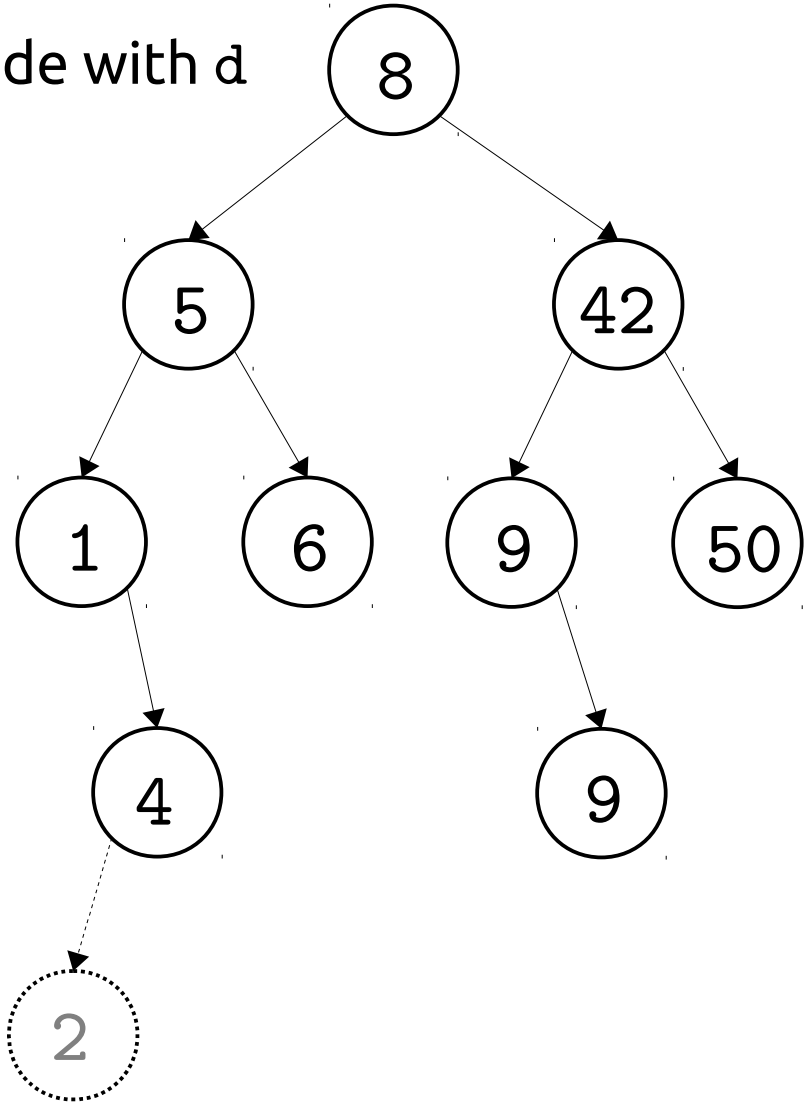
What is the time complexity of adding an element `d` to a BST with respect to its height h?

- starting from the root, we visit a bunch of nodes until we find the position where to add a new node with `d`
- each node visit takes Θ(1) – why?
- the maximum number of nodes that can be visited for adding is h+1

  This is the case when we need to arrive at a lowest leaf and add our new node below it

  In this tree, in order to add 2 we need to visit the nodes 8, 5, 1, 4, i.e. 4 nodes in total

Thus, adding a new element in the BST is in the worst case Θ(h)



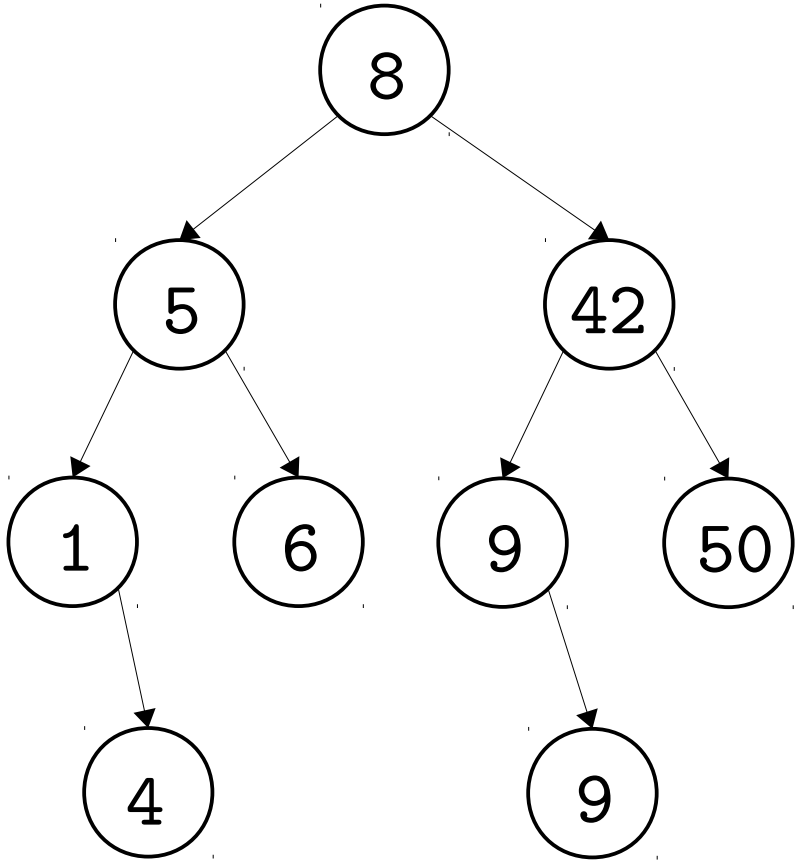
Searching for an element

What is the time complexity of searching for an element `d` to a BST with respect to its height h?

- starting from the root, we visit a bunch of nodes until we find `d` or we reach None
- each node visit takes Θ(1)
- the maximum number of nodes that can be visited for adding is h+1

  This is the case when we need to arrive at a lowest leaf and find d there, or
  not find it there and so reach None

  In this tree, if we search for 4 (or for 2) we need to visit 8, 5, 1, 4, i.e. 4 nodes in total



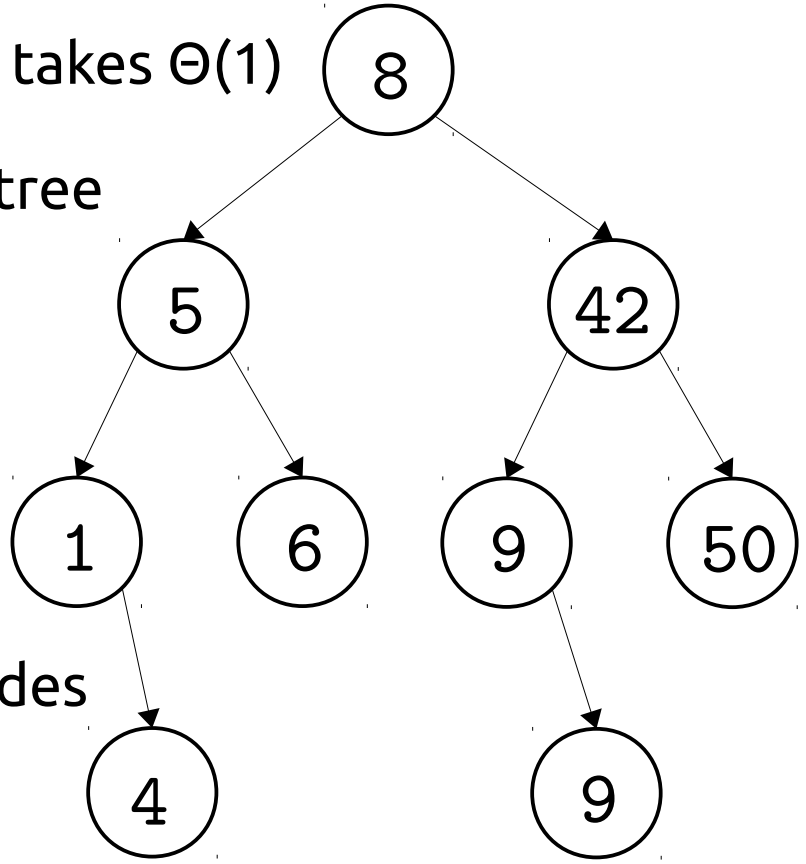
Thus, searching for an element in the BST is in the worst case Θ(h)

Removing an element

What is the time complexity of (searching and) removing an element `d` from a BST with respect to its height h ?

- starting from the root, we visit a bunch of nodes until we find `d` or we reach None, each node visit takes Θ(1)
- this gets us to a node in some level l of the tree
- To remove the node, in the worst case, we need an additional search for the minimum node on the right subtree below our node
- This latter search is from level l +1 on
- So, overall, we need to visit at most h +1 nodes

Thus, removing an element in the BST is in the worst case Θ(h)

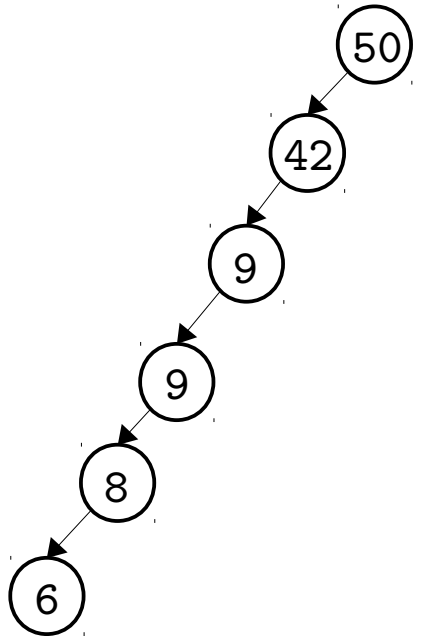


Height and size

So, all basic operations of BSTs are Θ(h) in the worst case.

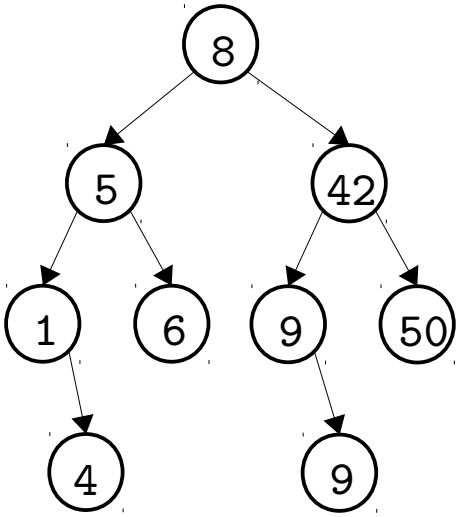
What does this mean in terms of the size n of the tree?

There are “degenerate” cases
where a tree is basically a linked list:



In such cases, h = Θ(n) and BSTs
are worse than linked lists!
-> because Θ(h) = Θ(n)

There are “reasonable” cases
where a tree is balanced:



i.e. it does not expand in height
without expanding in breadth.

In such cases, h = Θ(log$n$)
and BSTs are very efficient!
-> because Θ(h) = Θ(log$n$)

Balanced trees and efficiency

We call a tree **balanced** if, for each node in the tree, the heights of its left and right subtrees do not differ by more than 1.

For a balanced tree of arbitrary size n and height h, we can show that $h = \Theta(\log n)$.

So, for balanced BSTs:

add, search and remove are all $\Theta(\log n)$

Summing up

For a BST of arbitrary size n and height h:

- add, search and remove are all $\Theta(n)$ in the worst case
- but, if the tree is balanced: add, search and remove are all $\Theta(\log n)$ in the worst case.

Keeping a BST balanced is not always easy:

- e.g. adding 1,2,3, ..., 42 in an empty BST gives us a degenerate one
- if our operations are random enough, we will probably be balanced
- there are techniques (self-balancing trees) to keep BSTs balanced:
  - AVL trees
  - red-black trees
  - etc.

Trees, Trees everywhere!

We saw BSTs as an application of binary trees

We next look at another application: heaps!

Heaps are a data structure for storing elements so that:

- we can add an element efficiently
- we can find and remove the largest/smallest element efficiently

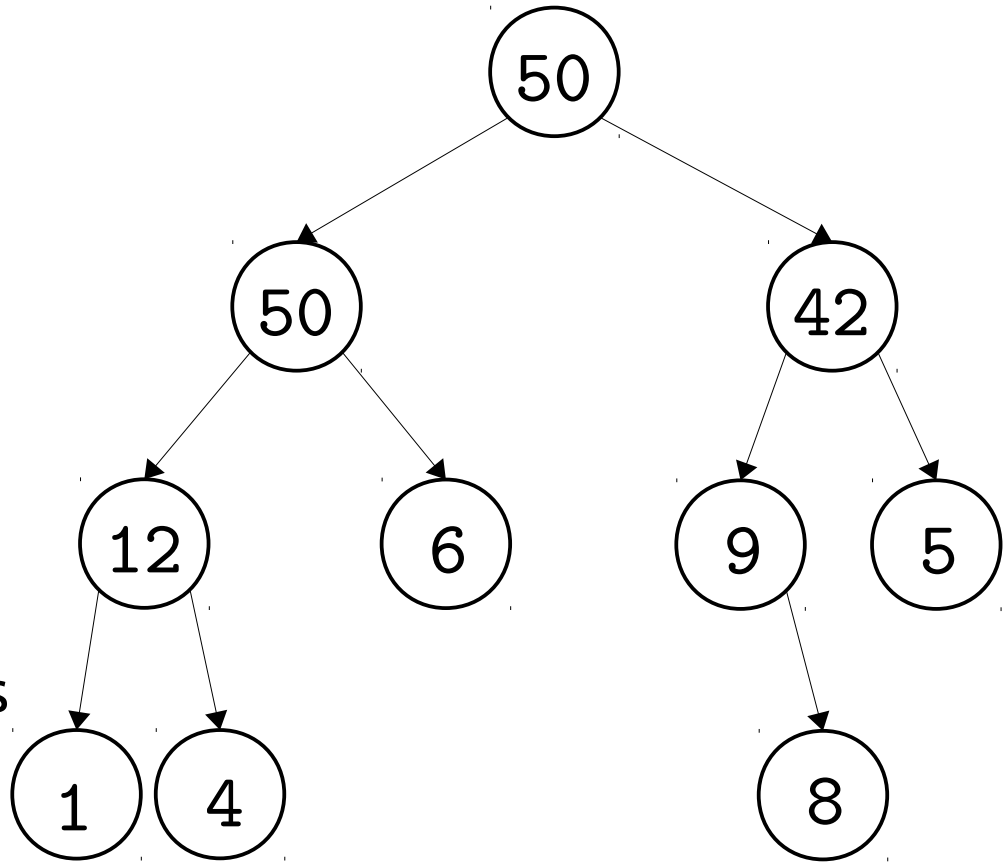
i.e. they are essentially priority queues.

Heaps

Heaps are trees whose nodes are ordered in a very specific way (this is called the **Heap property**).

From a given node `t`:

- all nodes below `t` (i.e. the children of `t`, and all their children, and all their children's children, etc.) have data values smaller or equal than that of `t`



In fact, this property is usually used to define **max heaps**,

- there are also min heaps (each node is smaller or equal than all below it)

We will assume here that all our heaps are max heaps

Heaps

Heaps are trees whose nodes are ordered in a very specific way (this is called the **Heap property**).

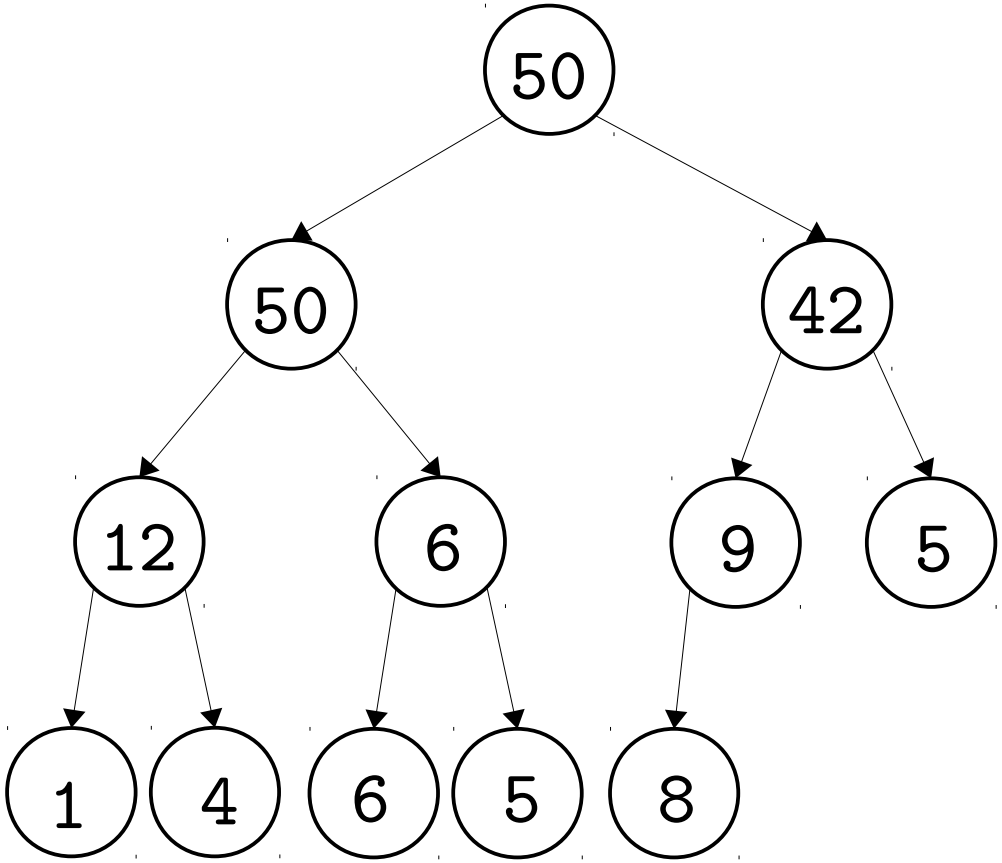
From a given node `t`:

- all nodes below `t` (i.e. the children of `t`, and all their children, and all their children's children, etc.) have data values smaller or equal than that of `t`

In addition, our heaps are binary trees and nearly complete:

- all but the bottom level of the heap are complete
- the bottom level is complete from its left end until its last leaf

The last leaf of the bottom level is called the **bottom leaf**

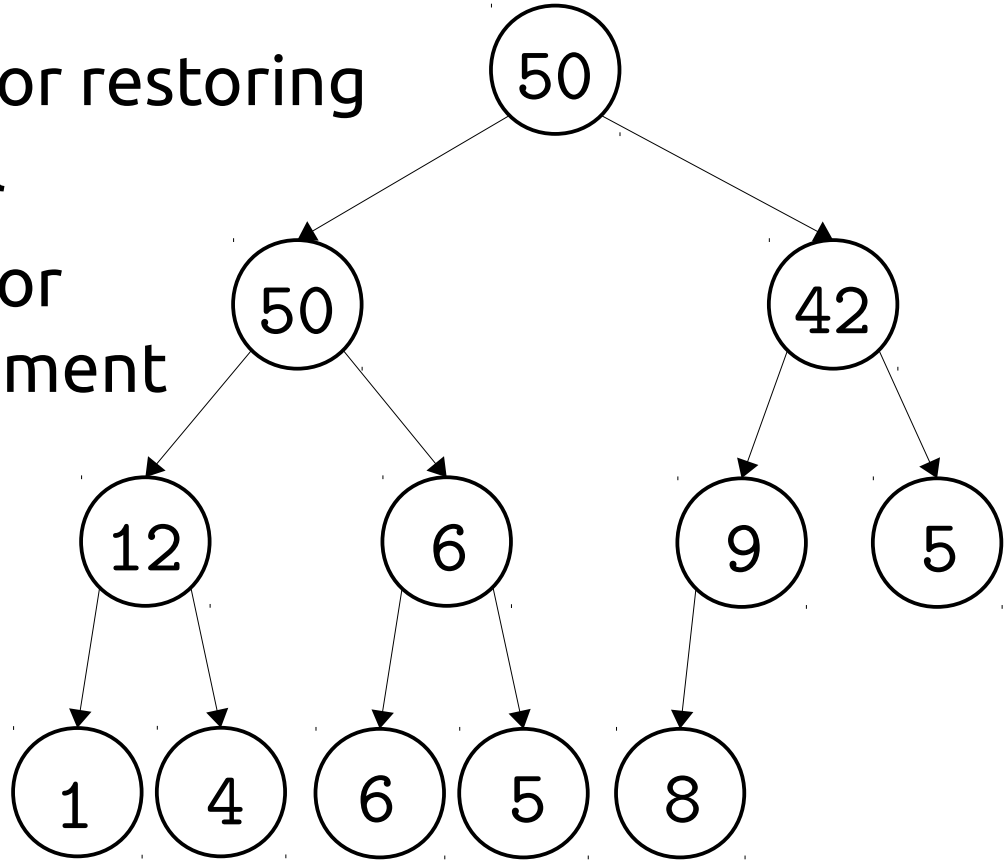


What are heaps good for?

Heaps are essentially priority queues:

- the highest element (the root) can be found in constant time
- there is an efficient mechanism for restoring the heap after each root removal
- there is an efficient mechanism for adding (i.e. enqueuing) a new element

We look at these next.



Adding a new element in a heap

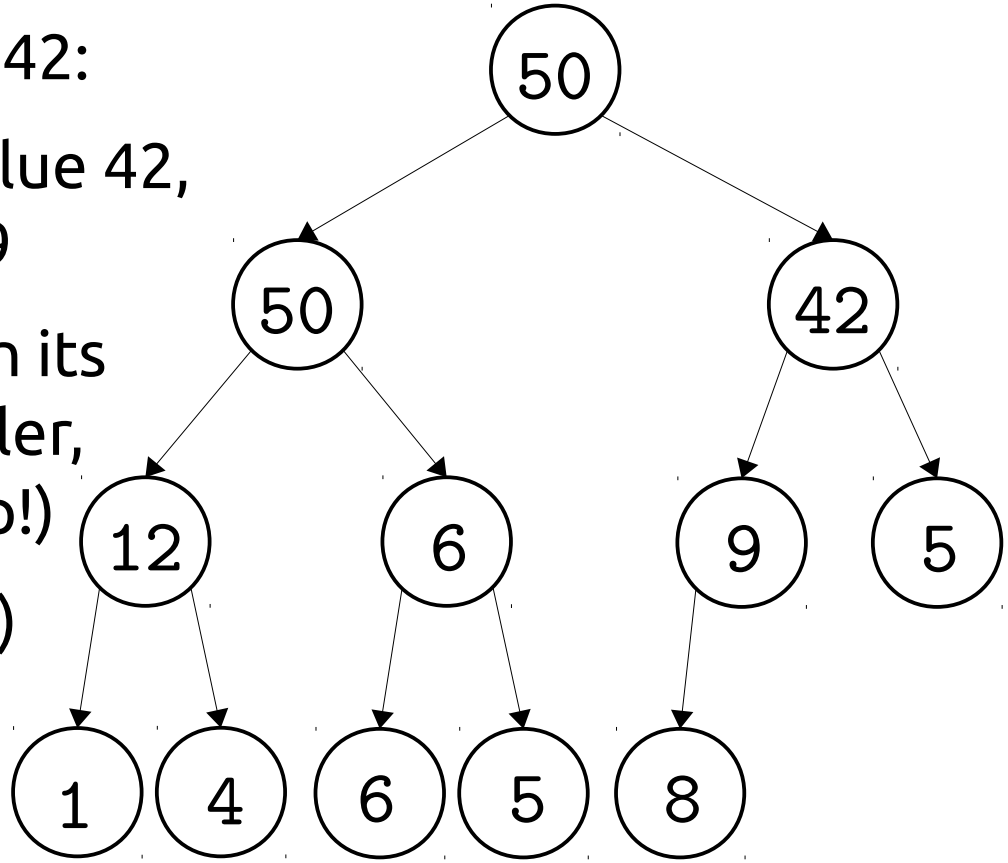
To add a new element in a heap we need to find a position such that the heap property is preserved **and** the heap is still nearly complete.

The idea is: *add element at bottom and bring up*.

For example, suppose we want to add 42:

- we add a new bottom node with value 42, i.e. this becomes the right child of 9
- we compare the new node (42) with its parent (9) and, if the parent is smaller, then we swap their values (so, swap!)
- we compare the swapped node (42) with its parent and, if the parent is smaller, then we swap their values,

  if not, we stop (so, stop!)



Adding a new element in a heap

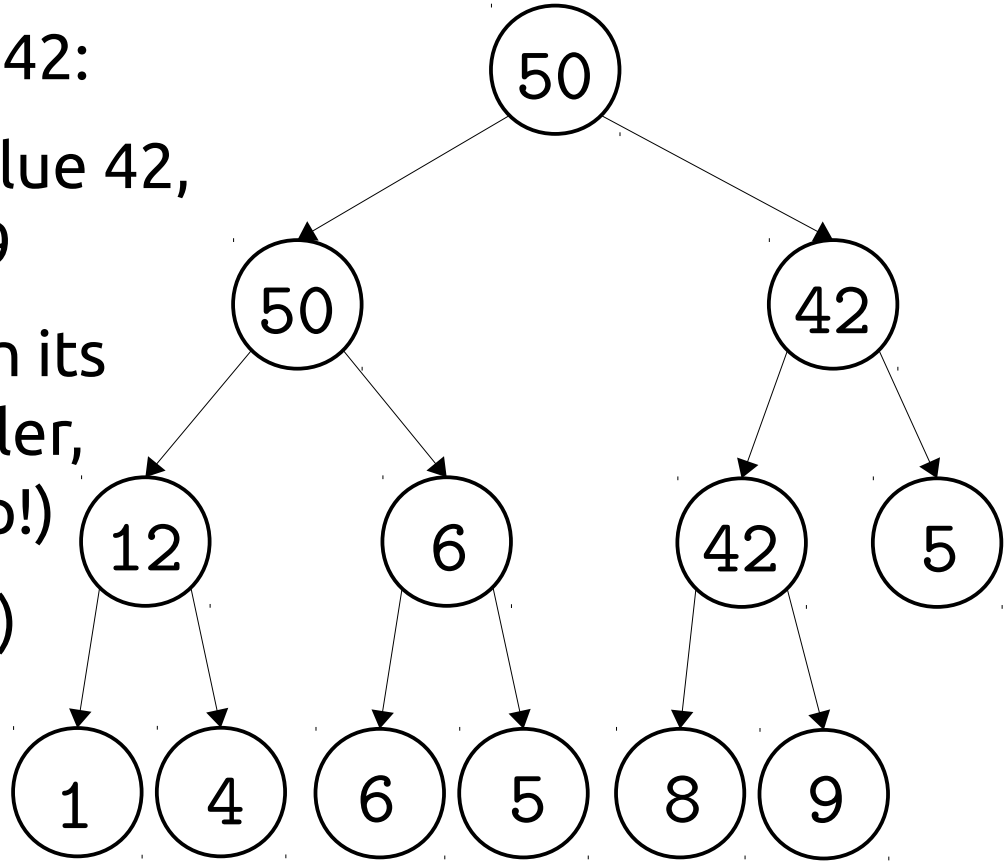
To add a new element in a heap we need to find a position such that the heap property is preserved **and** the heap is still nearly complete.

The idea is: *add element at bottom and bring up*.

For example, suppose we want to add 42:

- we add a new bottom node with value 42, i.e. this becomes the right child of 9
- we compare the new node (42) with its parent (9) and, if the parent is smaller, then we swap their values (so, swap!)
- we compare the swapped node (42) with its parent and, if the parent is smaller, then we swap their values,

  if not, we stop (so, stop!)



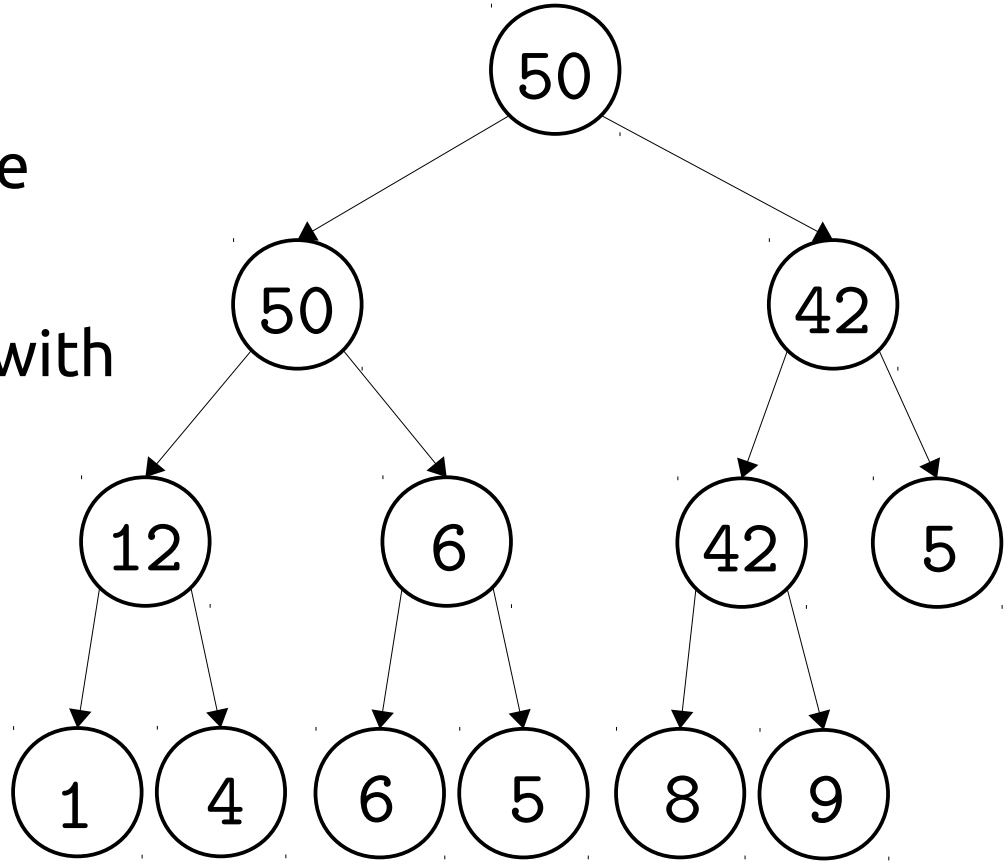
Adding a new element in a heap

To add a new element in a heap we need to find a position such that the heap property is preserved **and** the heap is still nearly complete.

The idea is: *add element at bottom and bring up*.

This means:

- we add a new bottom node with the value that we want to add
- we then keep swapping that value with its parent, going up the tree, until:
  - we reach the root node, or
  - we find a parent that is greater or equal to our node.
- Then, the resulting tree is a heap



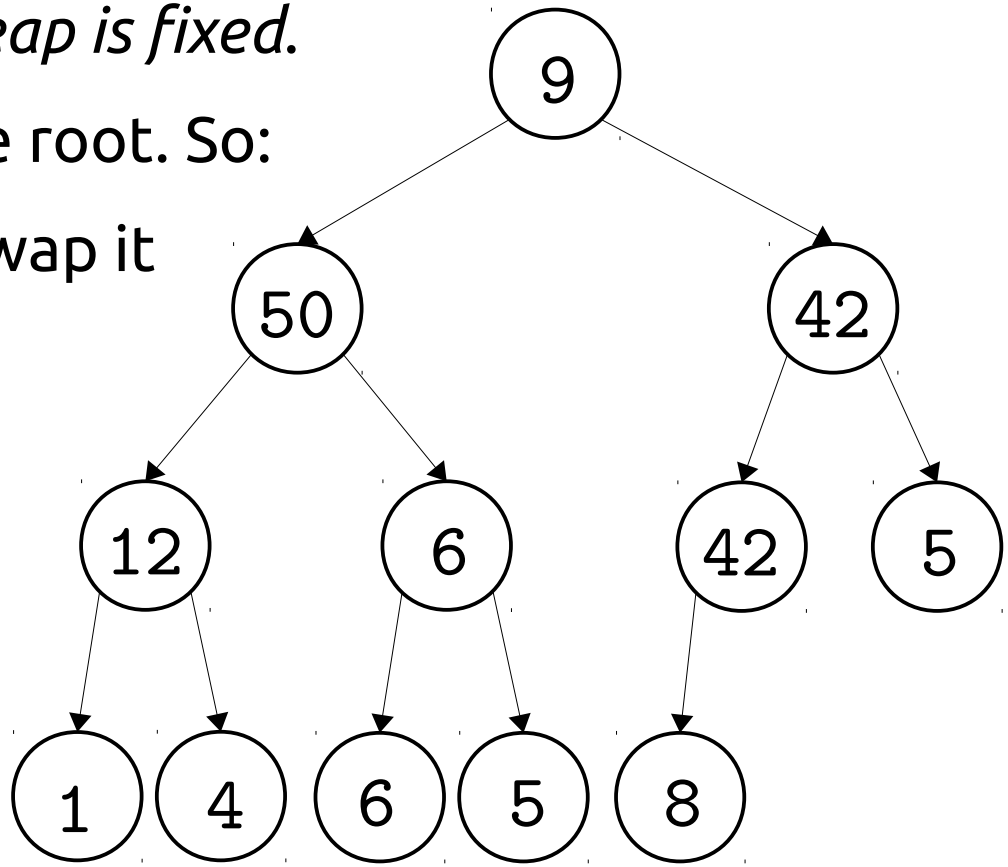
Heapify: fixing a broken heap

Sometimes (e.g. when removing) we may end up with a heap that is nearly complete but one of its nodes is smaller than one of its children. In such cases, we need to heapify (i.e. repair) the heap.

The idea is: *push element down until heap is fixed*.

For example, this heap is broken at the root. So:

- we pick the largest child of 9 and swap it with it -> i.e. we swap 9 with 50
- the heap is still broken at 9, so we pick its largest child and swap it with it -> i.e. we swap 9 with 12
- now this heap is fixed!



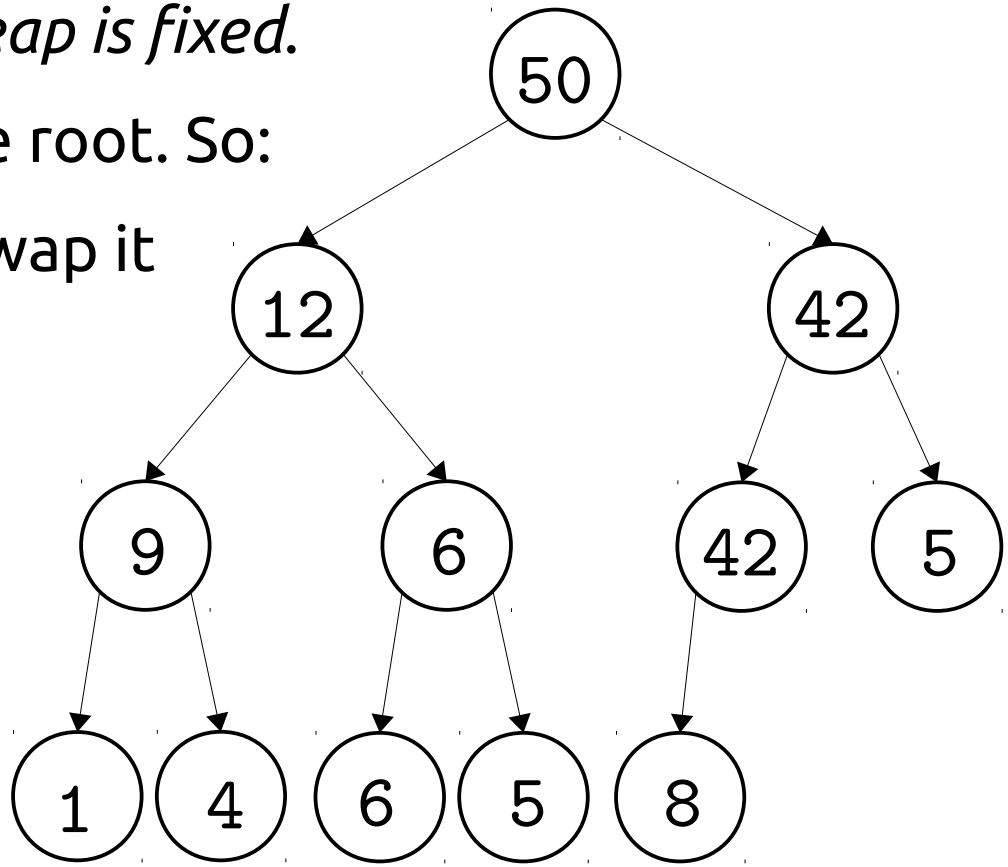
Heapify: fixing a broken heap

Sometimes (e.g. when removing) we may end up with a heap that is nearly complete but one of its nodes is smaller than one of its children. In such cases, we need to heapify (i.e. repair) the heap.

The idea is: *push element down until heap is fixed*.

For example, this heap is broken at the root. So:

- we pick the largest child of 9 and swap it with it -> i.e. we swap 9 with 50
- the heap is still broken at 9, so we pick its largest child and swap it with it -> i.e. we swap 9 with 12
- now this heap is fixed!



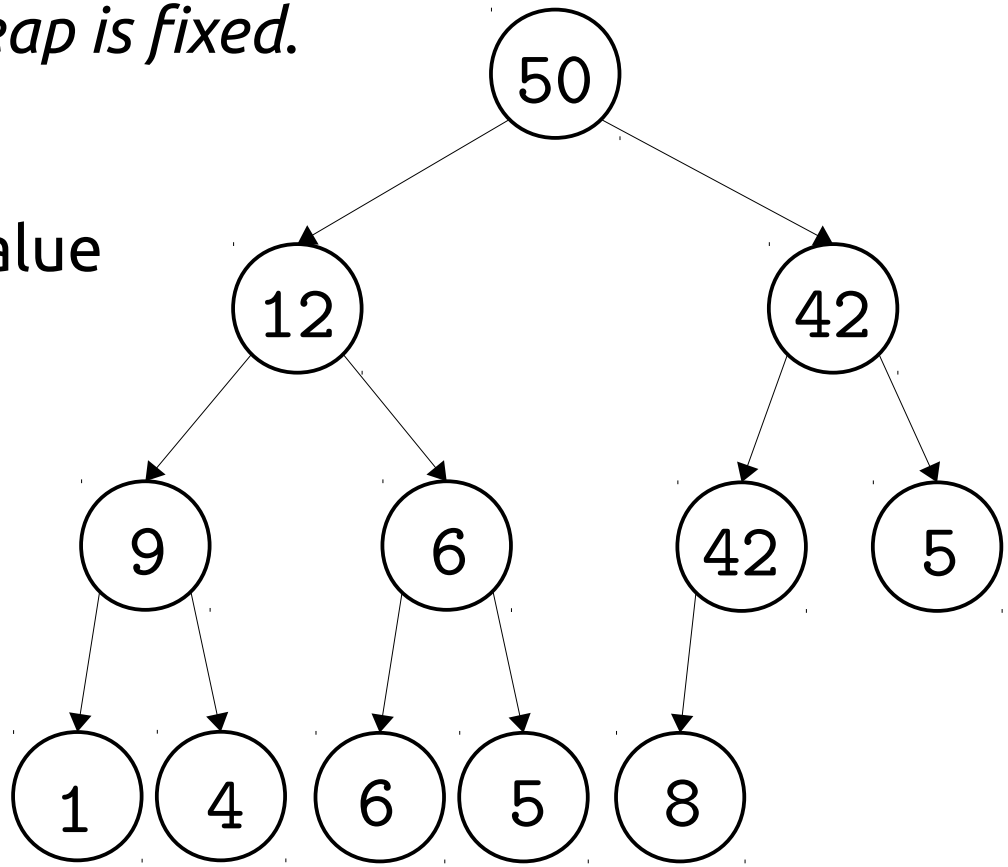
Heapify: fixing a broken heap

Sometimes (e.g. when removing) we may end up with a heap that is nearly complete but one of its nodes is smaller than one of its children. In such cases, we need to heapify (i.e. repair) the heap.

The idea is: *push element down until heap is fixed*.

This means:

- we keep swapping the offending value with its largest child
- until the heap is fixed



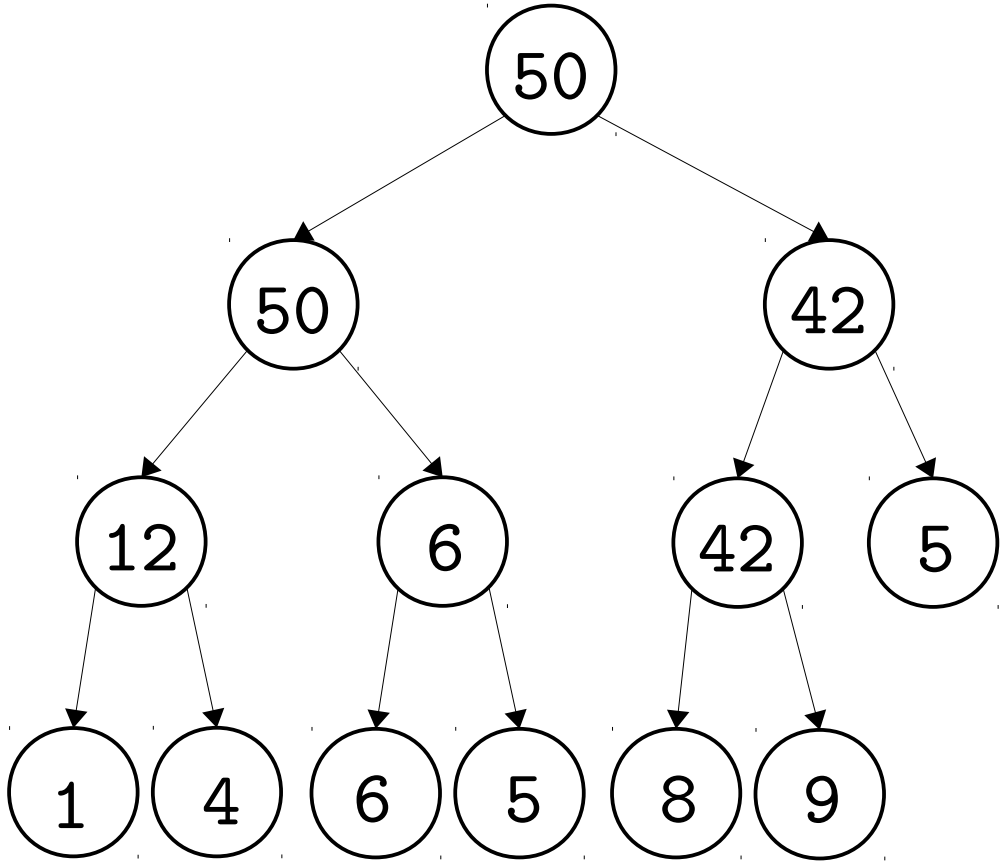
Remove root of the heap (i.e. dequeue)

So, to (find and) remove the root of a heap we simply:

- replace the root value with that of the bottom element
- remove the bottom element
- heapify (the root needs fixing)

For example,

- to remove the root of this heap



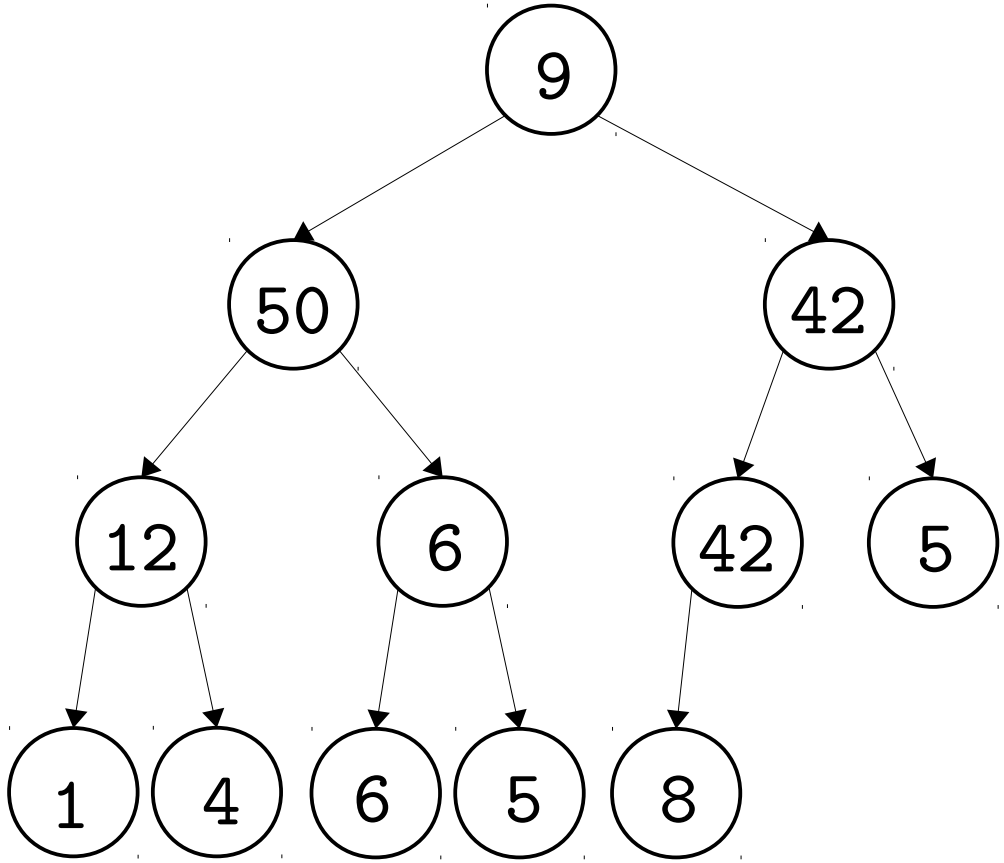
Remove root of the heap (i.e. dequeue)

So, to (find and) remove the root of a heap we simply:

- replace the root value with that of the bottom element
- remove the bottom element
- heapify (the root needs fixing)

For example,

- to remove the root of this heap
- we first bring the heap to this form



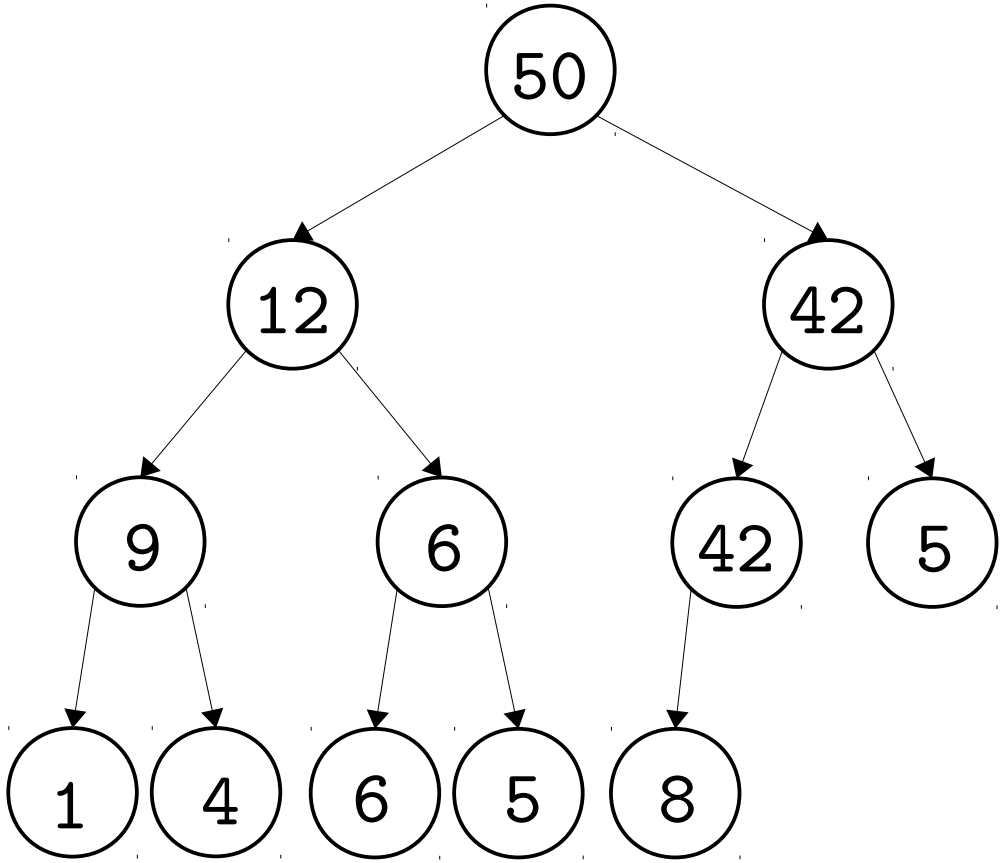
Remove root of the heap (i.e. dequeue)

So, to (find and) remove the root of a heap we simply:

- replace the root value with that of the bottom element
- remove the bottom element
- heapify (the root needs fixing)

For example,

- to remove the root of this heap
- we first bring the heap to this form
- and then heapify (as before)



Time complexity

The basic operations of heaps and their time complexities are:

- **add** an element to a heap:
  - as with BSTs, adding a new element is $\Theta(h)$, in the worst case, where h the height of the tree

    but heaps are always balanced by design (so $h = \Theta(\log n)$)

    so, adding an element is in the worst case $\Theta(\log n)$
- **find and remove** the root of the heap:
  - finding is in $\Theta(1)$

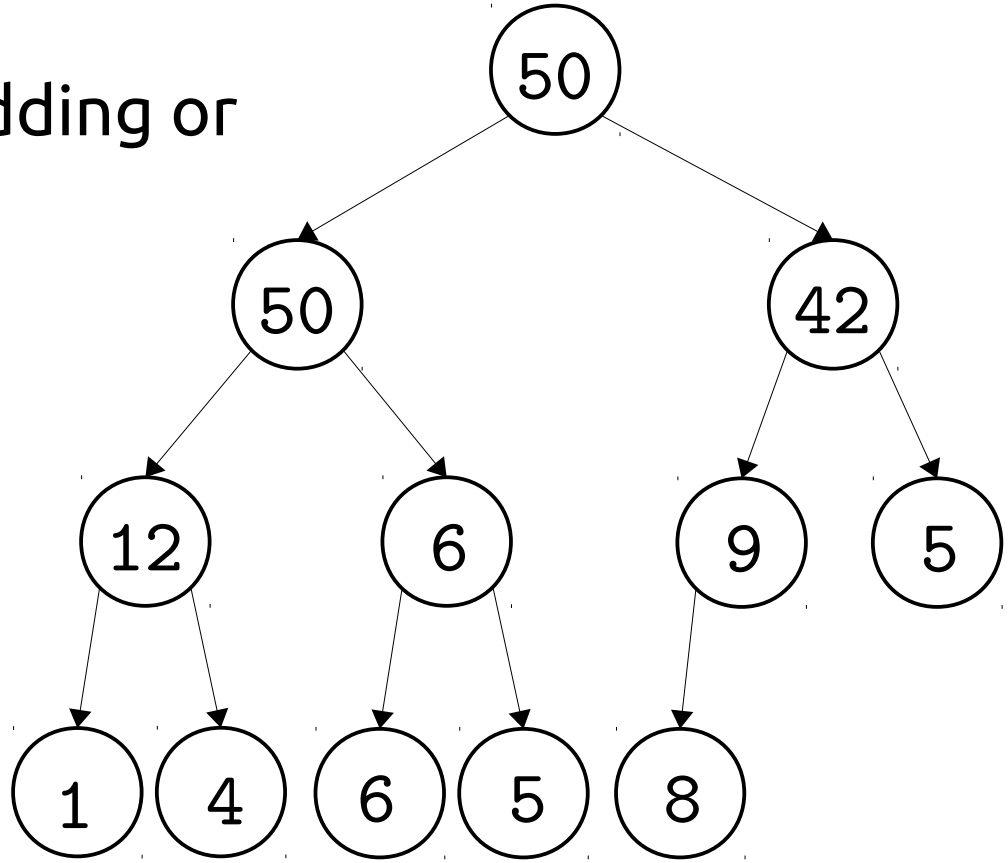
    removing is in $\Theta(h)$, in the worst case, i.e. in $\Theta(\log n)$

    so, in the worst case find and remove is $\Theta(\log n)$

How to implement heaps

Heaps are trees, so we could implement them using BTNodes:

- this works OK for forming a heap that we have already drawn on paper
- but it is **very inconvenient** for adding or removing elements from a heap!



The reason is that:

- in order to remove an element, we need to have a pointer to the bottom leaf of the heap
- in order to add an element, we need to have a pointer to the father of what will be the next bottom leaf – complicated..

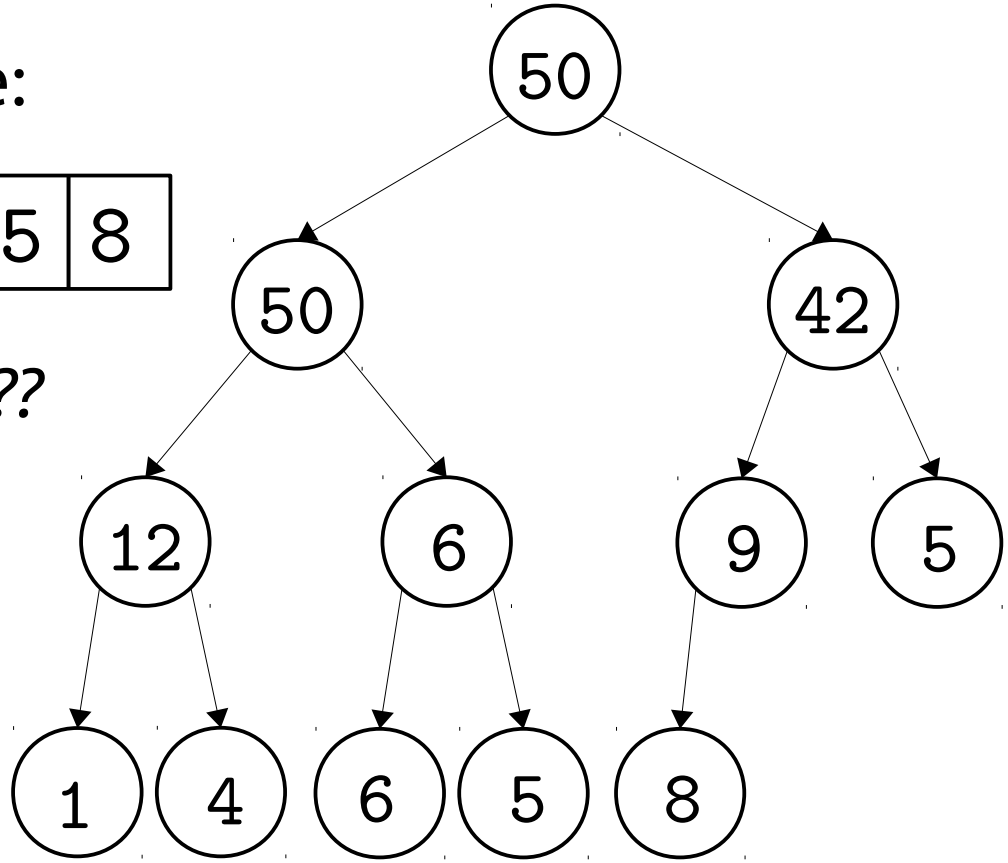
How to implement heaps

Instead, and this may come as a surprise, heaps can be easily implemented using array lists:

- the idea is simple: store a tree in an array in BFS fashion

I.e. for the tree on the right we have:

50	50	42	12	6	9	5	1	4	6	5	8



How can we ever traverse this "tree"??

E.g. how can we find what are the children of 6?

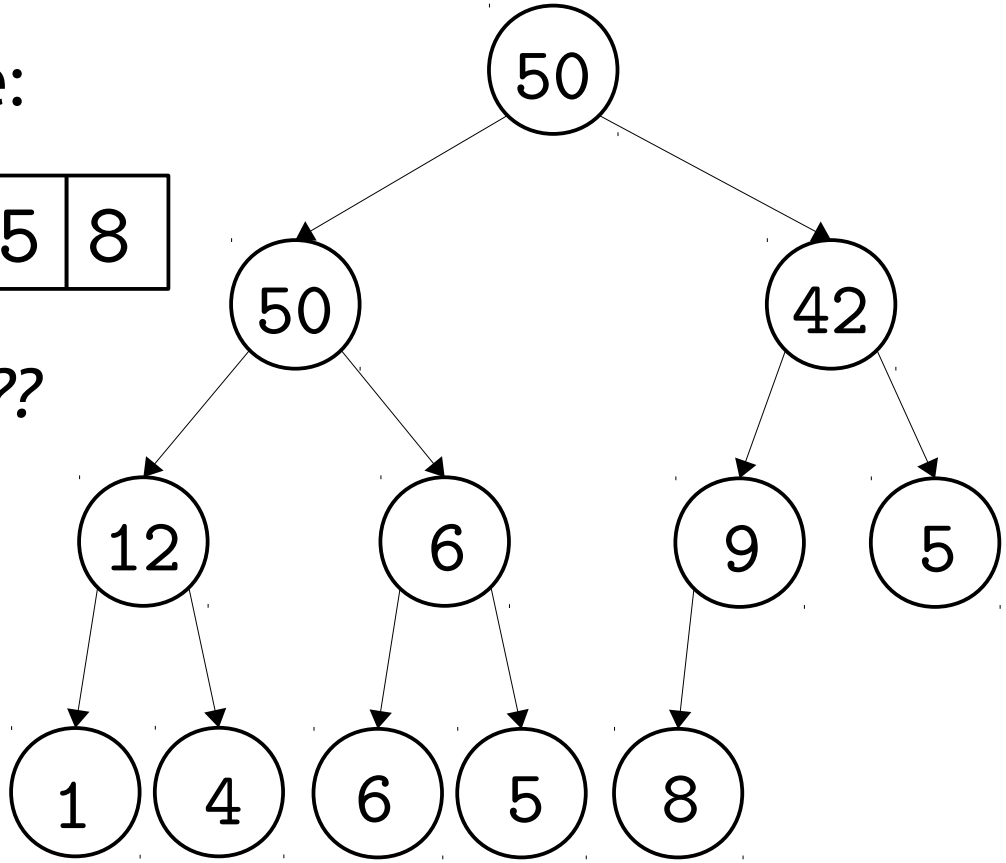
How to implement heaps

Instead, and this may come as a surprise, heaps can be easily implemented using array lists:

- the idea is simple: store a tree in an array in BFS fashion

I.e. for the tree on the right we have:

50	50	42	12	6	9	5	1	4	6	5	8



How can we ever traverse this "tree"??

E.g. how can we find what are the children of 6? It is simple:

the children of the i-th element are in positions $2i$+1 and $2i$+2

also, the parent of the i-th element is in position (i-1)//2

Heap class

Here is how we can start building our class:

```
class Heap:

    def __init__(self):
        self.inList = ArrayList()
        self.size = 0


    def add(self, d):
    # to implement

    def removeRoot(self):
    # to implement

    def heapify(self):
    # to implement
```

Application: heapsort

There is a straightforward way to sort an array `A`:

- we create an empty heap `h`
- add all the elements of the array `A` in the heap `h`

So, `h` contains all elements of `A`.

We then read the elements of `h` back in `A`, as follows:

- we find/remove the root of `h` and store it in `[len(A)-1]`
- we find/remove the root of (updated) `h` and store it in `[len(A)-1]`
- and so on

At the end, the array `A` is sorted.

Moreover, if the array has size n, sorting it takes $\Theta(n \log n)$ in the worst case.

That is, the heapsort algorithm (i.e. the one we just described) is $O(n \log n)$

Heap code

```
class Heap:
    def __init__(self):
        self.inList = ArrayList()
        self.size = 0

    def __str__(self):
        return str(self.inList)

    def add(self, d):
        self.inList.append(d)
        pos = self.size
        self.size += 1
        while pos > 0 and self.inList.get(pos) > self.inList.get((pos-1)//2):
            self.inList.swap(pos,(pos-1)//2)
            pos = (pos-1)//2

    def heapify(self, pos):
        if 2*pos+1 >= self.size: return
        if 2*pos+2 >= self.size or self.inList.get(2*pos+1) > self.inList.get(2*pos+2):
            maxChild = 2*pos+1
        else: maxChild = 2*pos+2
        if self.inList.get(maxChild) < self.inList.get(pos): return
        self.inList.swap(pos,maxChild)
        self.heapify(maxChild)

    def removeRoot(self):
        val = self.inList.get(0)
        self.inList.set(0,self.inList.set(self.size-1))
        self.inList.remove(self.size-1)
        self.size -= 1
        self.heapify(0)
        return val
```

```
def heapSort1(A):
    h = Heap()
    for i in range(len(A)):
        h.add(A[i])
    for i in range(len(A)-1,-1,-1):
        A[i] = h.removeRoot()
```

Exercises

1. Draw the heap we obtain if we start from the empty heap and add consecutively the numbers:

   24,15,1,11,45,23,65,12,5,12,67,32

2. Starting from the heap you constructed in the previous question, on your paper, remove the root of the heap.

   You should work in the same way that `removeRoot` works.

3. Explain why:
   a) removing the root of a heap takes $\Theta(\log n)$ in the worst case
   b) heapsort is $O(n \log n)$

4. Write a `Heap` function

   ```
   def append(self)
   ```

   that simply appends an element at the end of the heap, without fixing the heap afterwards

   Now, write a function

   ```
   def heapsort2(A)
   ```

   that sorts the array `A` as follows:

   - it creates an empty heap `h` and appends to it all the elements of `A`
   - it then heapifies `h` by starting at its last element and going bottom-up until the root
   - it then reads the elements of `A` back in `h` as in the heapsort that we saw in class

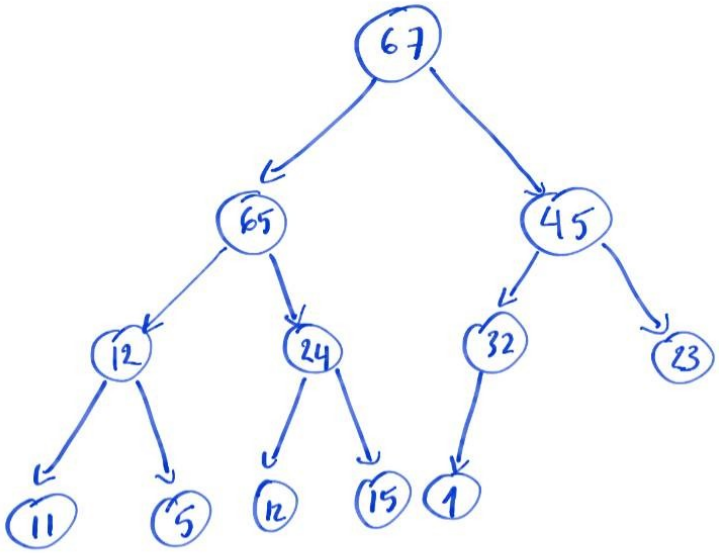
5. Write a function

   ```
   def heapsort3(A)
   ```

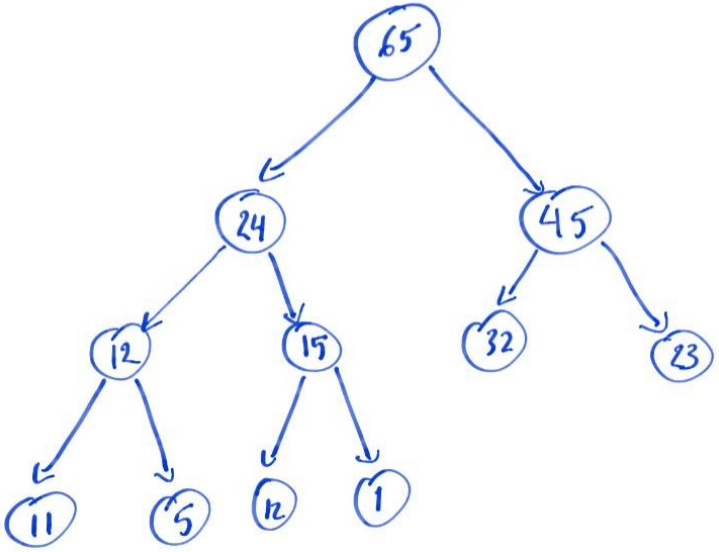
   that sorts the array `A` using the heapsort technique *in-place*, i.e. by not creating an external heap but, rather, by forming the heap inside `A` (i.e. by transforming `A` itself into a heap) and then reading out in order the elements from `A` back to `A`.

Exercises – Solutions

1. Here is the final heap we obtain:



2. After removing its root, we have:



3. For (a), an explanation along the lines of slide 23 and would suffice. For (b), we can argue as follows. Given an array of length n, the heapsort algorithm:
   - creates a heap, starting from the empty heap and adding each element of the array
     - each addition takes $\Theta(\log n)$ in the worst case, and we do n additions
     - so, altogether this takes at most $\Theta(n \log n)$
   - we then remove the elements of the heap one-by-one, using removeRoot, and store them in the array
     - each removal takes at most $\Theta(\log n)$, and we do n of them
     - so, overall this takes at most $\Theta(n \log n)$

   So, taking both stages into account, we have that heapsort takes $\Theta(n \log n)$ in the worst case.

   So, heapsort is $O(n \log n)$.

4. See `lecture10.ipynb`