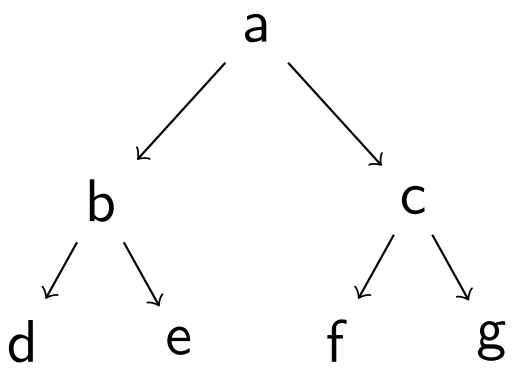
Binary Trees

Binary trees



- Each entry (a, b, c, . . .) is a **node**.
- Every node except one has a **parent** above it.
- The node with no parent (a) is the **root**.
- Each node has a **left child** or a **right child** or neither or both.
- A node with no children is a **leaf**.
- The **subtree rooted at a node** is the tree consisting of that node and all its descendants.

The (traditional) binary tree ADT

Operations and behaviour.

- `void create (String s)` creates a one-node tree that holds `s`.
- `void join (String s, Tree left, Tree right)` creates a tree with `s` at the root and the specified subtrees.
- `boolean isLeaf()` checks if the tree has just one node.
- `Tree leftChild()` and `Tree rightChild()` return the subtrees rooted at the children of the root.
- `String value()` returns the string stored at the root.

Running time of all operations is independent of the size of the tree.

ADT critique

The given operations aren't actually very useful – they require trees to be built from leaves upwards, e.g.,

```
Tree d = new Tree ("d");
Tree e = new Tree ("e");
Tree c = new Tree ("c", d, e);
...
Tree a = new Tree ("a", b, c);
```

Practical implementations typically add operations.

Java implementation (1)

```
public class BinaryTree {
    private String value;
    private BinaryTree left;
    private BinaryTree right;

    BinaryTree (String value) { this (value, null, null); }

    BinaryTree (String value, BinaryTree left, BinaryTree
        right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
    ...
```

Java implementation (2)

```
    boolean isLeaf () { return left == null && right ==
        null; }

    BinaryTree leftChild () { return left; }
    BinaryTree rightChild () { return right; }
    String value () { return value; }
}
```

Programming with binary trees

Code for binary trees often uses recursion: much easier than trying to walk up and down the tree with a cur reference.

```
boolean contains (String s) {
    return (value != null && value.equals (s))
           || (left != null && left.contains (s))
           || (right != null && right.contains (s));
}
```

Traversing binary trees

Traversing a tree refers to systematically walking through its nodes and processing them.

Three main methods:

- pre-order: process a node, then recurse on the subtrees.
- post-order: recurse on the subtrees, then process the node.
- in-order: recurse on the left subtree, process the node, then recurse on the right subtree.

`contains()` on the previous slide is essentially a pre-order traversal that stops early if it finds `s`.

Pre-order traversal

```
void preOrder () {
    System.out.println(value);
    if (left != null) left.preOrder();
    if (right != null) right.preOrder ();
}
```

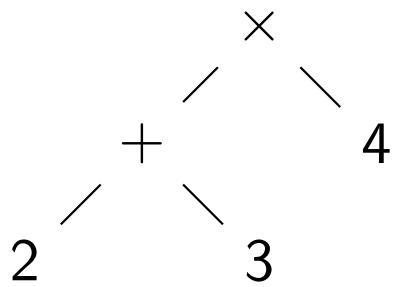
Uses:

- searching the tree;
- printing the tree, viewing each node as a heading with two subsections (e.g., "Noah begat Shem and Ham, and Shem begat Elam and Arpachshad, and Arpachshad begat...").

Post-order traversal

```
void postOrder () {
    if (left != null) left.postOrder();
    if (right != null) right.postOrder ();
    System.out.println(value);
}
```

Used to evaluate arithmetic expressions stored as trees.

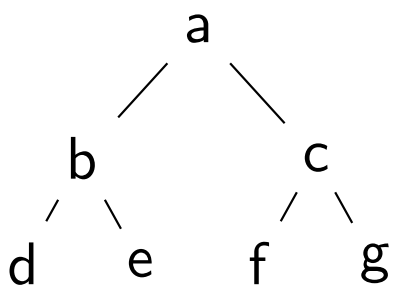


To evaluate a non-leaf node, evaluate the children and combine with the arithmetic operator at the node. $(2 + 3) \times 4 = 20$.

In-order traversal

```
void inOrder () {
    if (left != null) left.inOrder();
    System.out.println(value);
    if (right != null) right.inOrder ();
}
```

Used to print the contents of the tree "left-to-right".



In-order: d, b, e, a, f, c, g

Non-binary trees

In general, nodes in trees may have any number of children.

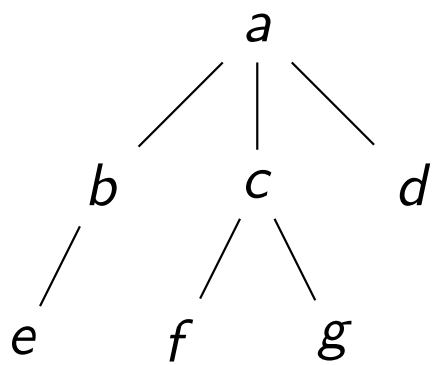
- If maximum number of children is known, each node could have an array of children. Wastes space if most nodes have fewer children. Are you *sure* it's the max? ("Nobody has more than ten children, right?")
- If number of children known is when node is created, and can't change, can make array of the right size.
- Usually better to use a list of children.

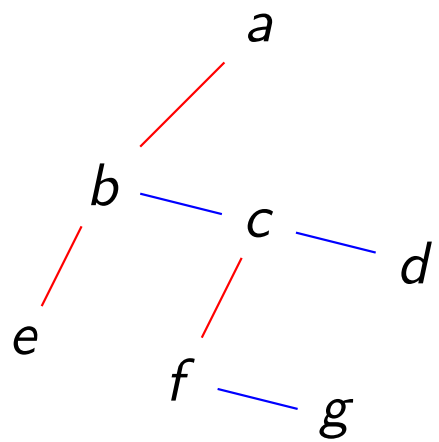
Left-child, right-sibling

To avoid needing separate tree and list classes, we can store child lists *implicitly* in a binary tree.

Repurpose the left and right child references:

- `left` points to the node's first child (in red, below);
- `right` points to the node's next sibling (in blue).





Uses of (binary) trees

- Binary search trees and priority queues are efficient ways of storing and searching sorted data (next lecture).
- 4-ary trees used to partition space in computer graphics.
- Syntax trees in compilers.
- Hierarchical data.

Binary Search Trees

Searchable sets

Suppose we want to store a changing set of data (e.g., integers) and quickly answer "is x in the set?" queries.

Searchable sets

Suppose we want to store a changing set of data (e.g., integers) and quickly answer "is x in the set?" queries.

If we store as a list:

- adding new items is fast (append in constant time);
- searching is slow (on average, have to look through half the list).

Searchable sets

Suppose we want to store a changing set of data (e.g., integers) and quickly answer "is x in the set?" queries.

If we store as a list:

- adding new items is fast (append in constant time);
- searching is slow (on average, have to look through half the list).

If we don't care about order, we can store as a sorted array:

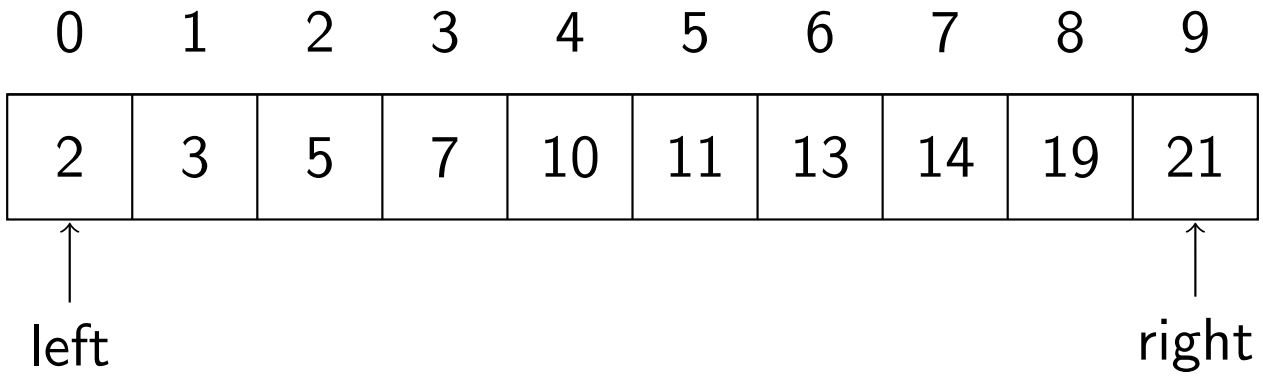
- adding new items is slow (on average, must move half the data to insert a new item);
- searching is fast (proportional to $\log_2(\mathrm{length})$).

Binary search

0	1	2	3	4	5	6	7	8	9
2	3	5	7	10	11	13	14	19	21

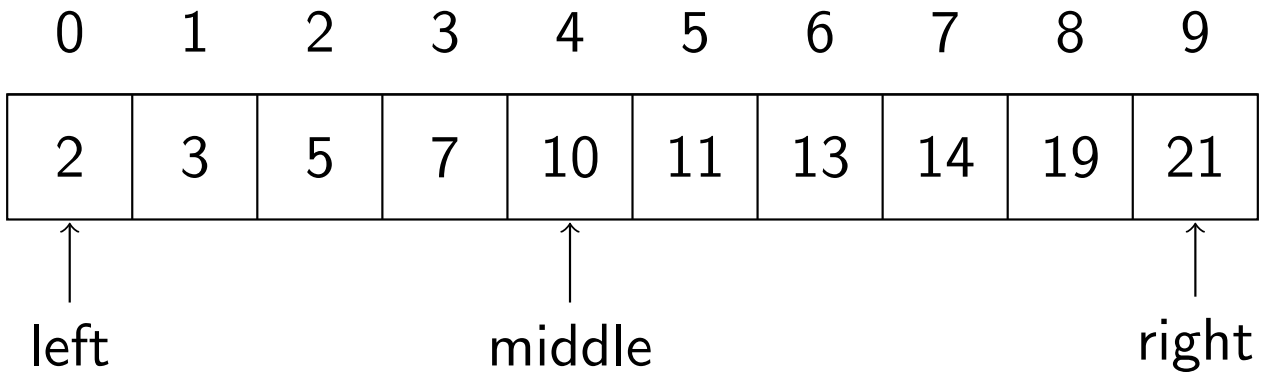
Search for 13.

Binary search



Search for 13.

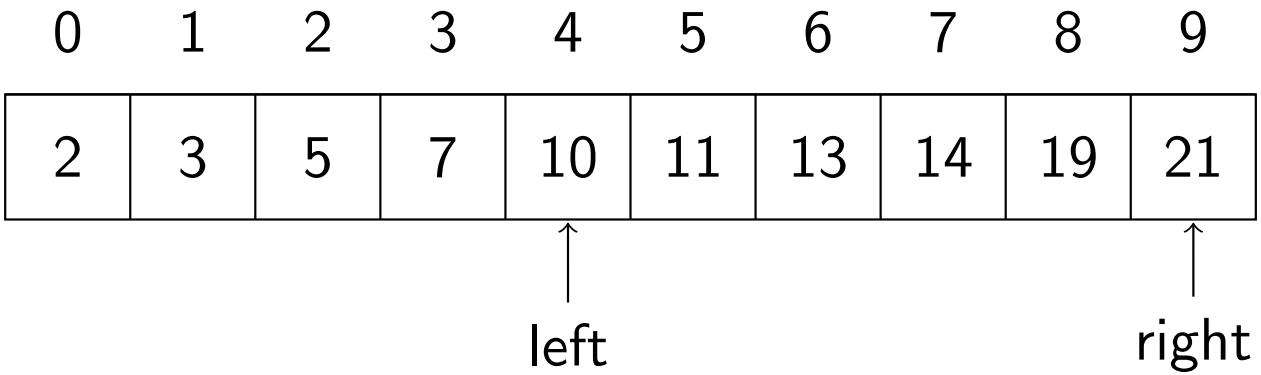
Binary search



Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

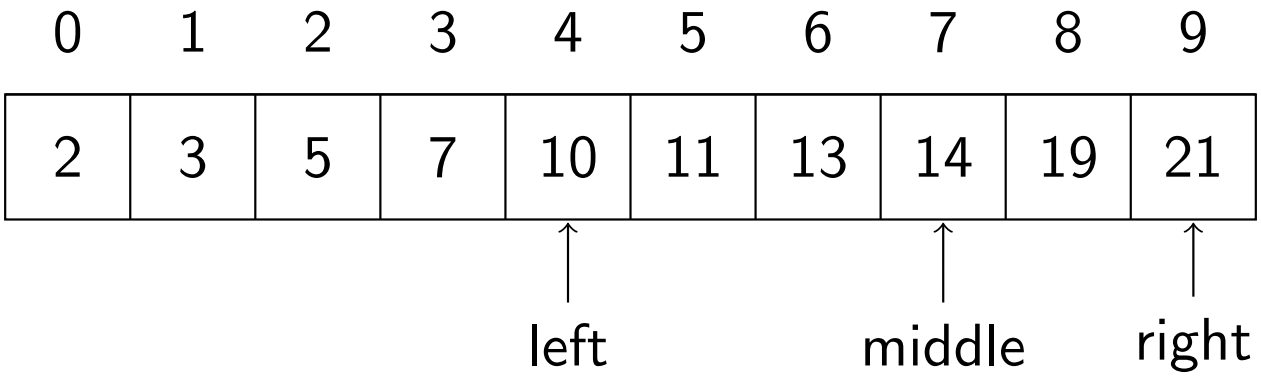
Binary search



Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

Binary search

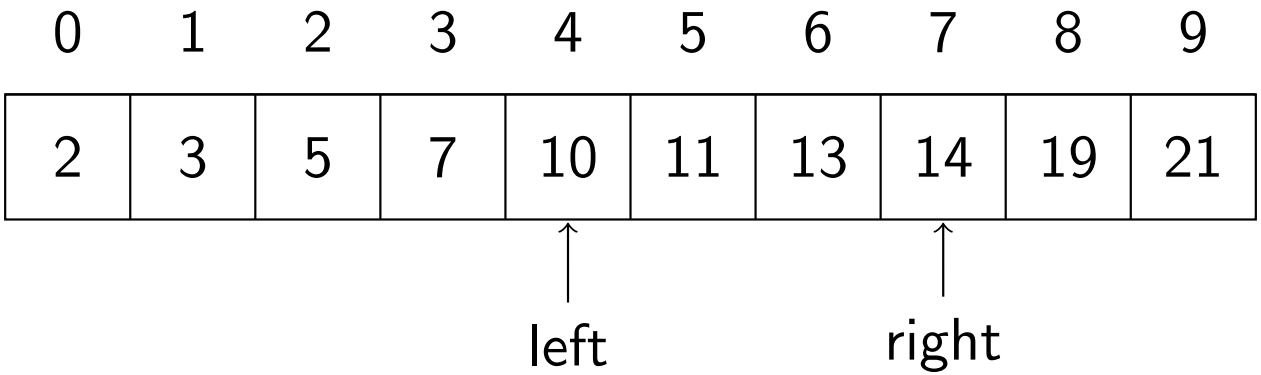


Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

$13 < 14$ so must be in left half: set right $\rightarrow$ middle.

Binary search

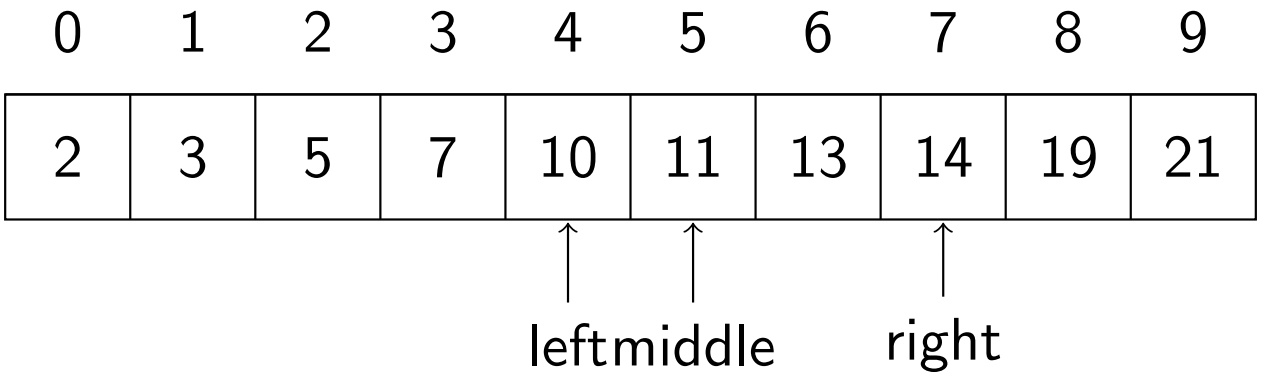


Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

$13 < 14$ so must be in left half: set right $\rightarrow$ middle.

Binary search



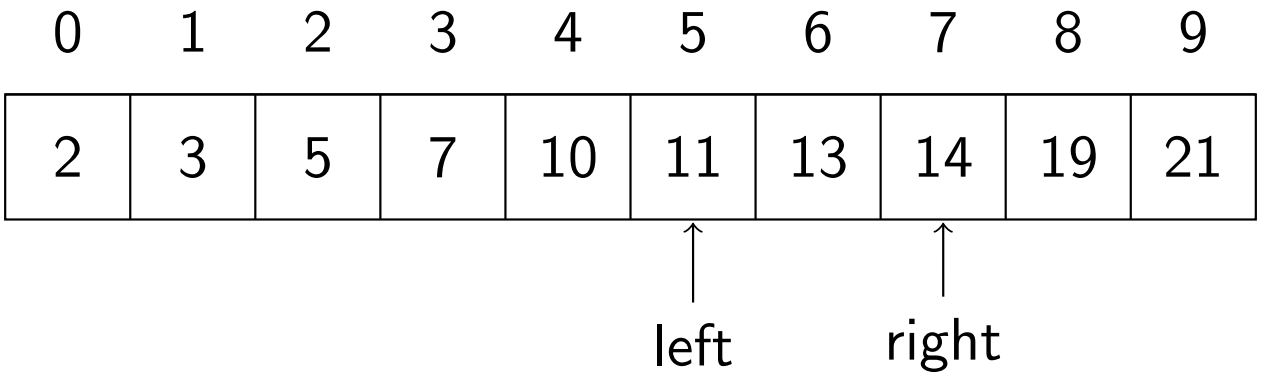
Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

$13 < 14$ so must be in left half: set right $\rightarrow$ middle.

$13 > 11$ so must be in right half: set left $\rightarrow$ middle.

Binary search



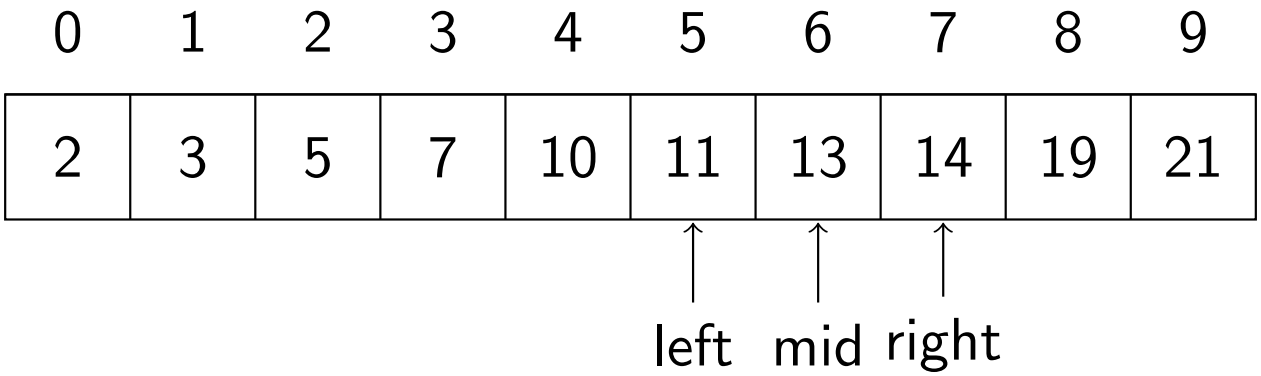
Search for 13.

$13 > 10$ so must be in right half: set left $\rightarrow$ middle.

$13 < 14$ so must be in left half: set right $\rightarrow$ middle.

$13 > 11$ so must be in right half: set left $\rightarrow$ middle.

Binary search



Search for 13.
$13 > 10$ so must be in right half: set left $\rightarrow$ middle.
$13 < 14$ so must be in left half: set right $\rightarrow$ middle.
$13 > 11$ so must be in right half: set left $\rightarrow$ middle.
Found it!

Binary search in Java

```
boolean contains (int[] arr, int i) {
    int left = 0;
    int right = arr.length-1;

    if (arr[left] == i || arr[right] == i) return true;
    while (right > left+1) {
        int middle = (left + right)/2;
        if (arr[middle] == i) return true;
        else if (i < arr[middle]) right = middle;
        else left = middle;
    }
    return false;
}
```

Binary search in Java

```
boolean contains (int[] arr, int i) {
    int left = 0;
    int right = arr.length-1;

    if (arr[left] == i || arr[right] == i) return true;
    while (right > left+1) {
        int middle = (left + right)/2;
        if (arr[middle] == i) return true;
        else if (i < arr[middle]) right = middle;
        else left = middle;
    }
    return false;
}
```

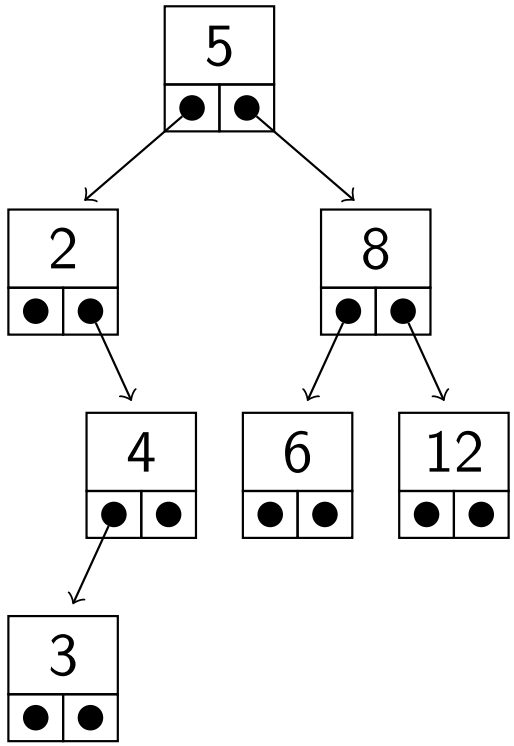
Running time on array with n items is proportional to $\log_2 n$.

("How many times can I divide n by 2 before the answer is less than 1?")

Binary search trees

Key property of binary search trees: if a node stores the number x, then:

- everything in its left subtree is less than x; and
- everything in its right subtree is greater than x.



The binary search tree ADT

Operations and behaviour.

- `void create()` makes a new empty binary search tree.
- `boolean isEmpty()` returns whether the tree is empty.
- `int size()` returns the number of items in the tree.
- `boolean contains(int i)`: is `i` in the tree?
- `void insert(int i)` inserts `i` into the tree.
- `void delete(int i)` removes `i` from the tree.

Running time of `contains()`, `insert()`, and `delete()` are proportional to the **height** of the tree.

Height of trees (1)

The **height** is the number of levels below the root.

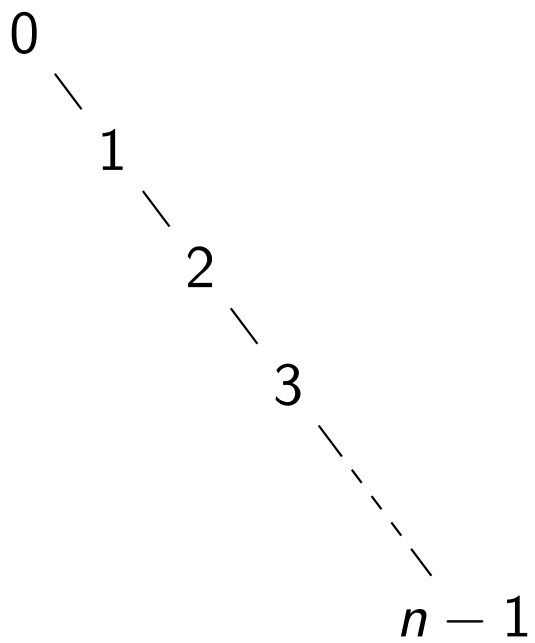
The tree on slide 84 has height 3.

Height of trees (1)

The **height** is the number of levels below the root.

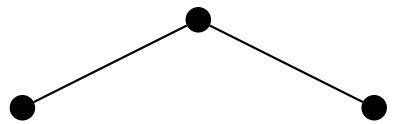
The tree on slide 84 has height 3.

A tree with n nodes can have height up to $n - 1$.



Height of trees (2)

Height	Max nodes
1	3



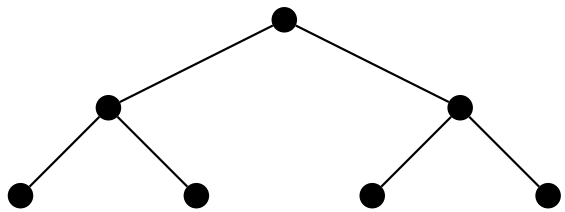
So a binary tree with n nodes has height at least $\approx \log_2 n$.

Reed has shown that a random binary search tree has height $3 \log_2 n$, on average.

Typical binary search trees look like random ones, not like lists.

Height of trees (2)

Height	Max nodes
1	3
2	7



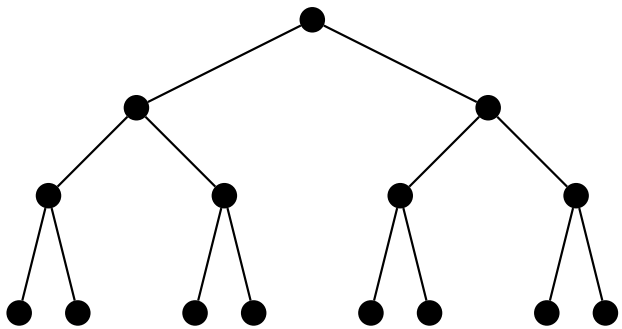
So a binary tree with n nodes has height at least $\approx \log_2 n$.

Reed has shown that a random binary search tree has height $3 \log_2 n$, on average.

Typical binary search trees look like random ones, not like lists.

Height of trees (2)

Height	Max nodes
1	3
2	7
3	15



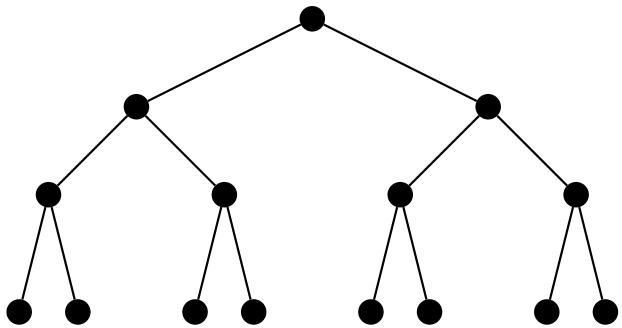
So a binary tree with n nodes has height at least $\approx \log_2 n$.

Reed has shown that a random binary search tree has height $3 \log_2 n$, on average.

Typical binary search trees look like random ones, not like lists.

Height of trees (2)

Height	Max nodes
1	3
2	7
3	15
h	$2^{h+1} - 1$



So a binary tree with n nodes has height at least $\approx \log_2 n$.

Reed has shown that a random binary search tree has height $3 \log_2 n$, on average.

Typical binary search trees look like random ones, not like lists.

Binary search tree ADT revisited

Operations and behaviour.

- `boolean contains(int i)`: is `i` in the tree?
- `void insert(int i)` inserts `i` into the tree.
- `void delete(int i)` removes `i` from the tree.

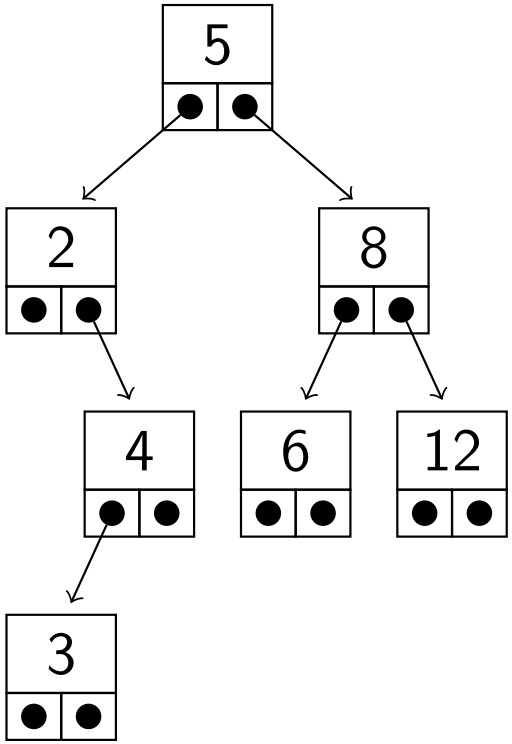
Running time of `contains()`, `insert()` and `delete()` when there are n nodes is

- typically proportional to $\log n$;
- in the worst case, proportional to n.

Java implementation of nodes

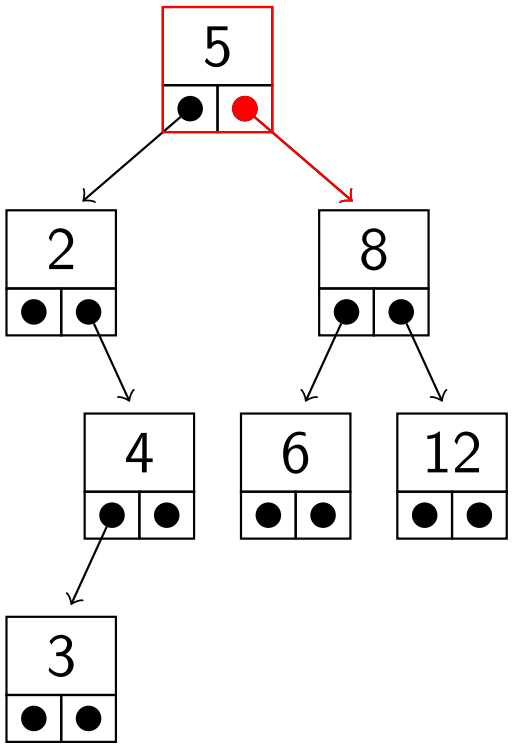
```
public class BinarySearchTree {
    private class Node {
        Node parent; /* Useful in delete() */
        Node left, right;
        int value;

        Node (Node parent, int value) { ... }
    }

    Node root = null;
    int size = 0;
    ...
}
```

Implementing `contains()` (1)



Tree contains 6?

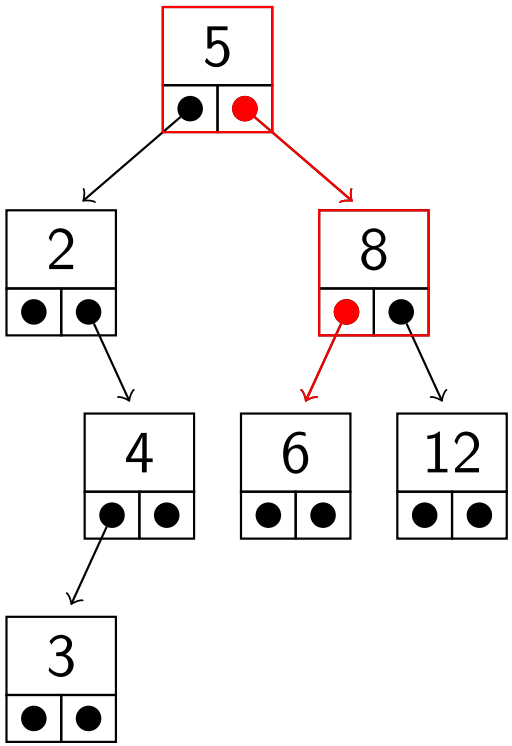
Implementing `contains()` (1)



Tree contains 6?

$6 > 5$ so go right.

Implementing `contains()` (1)

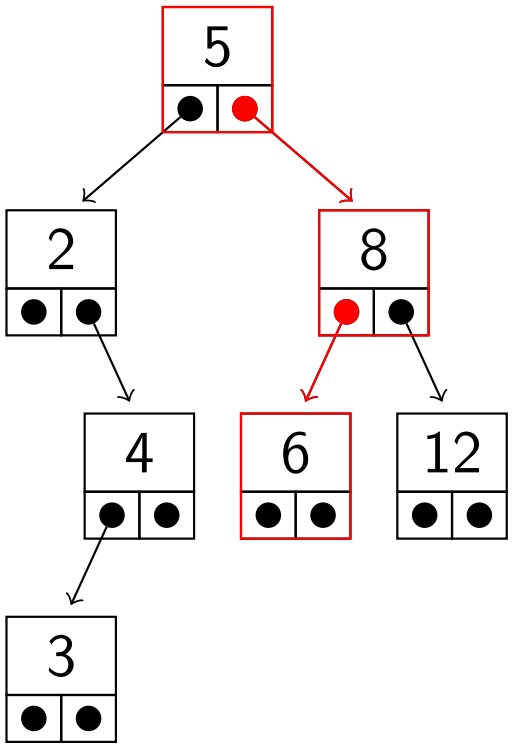


Tree contains 6?

$6 > 5$ so go right.

$6 < 8$ so go left.

Implementing `contains()` (1)



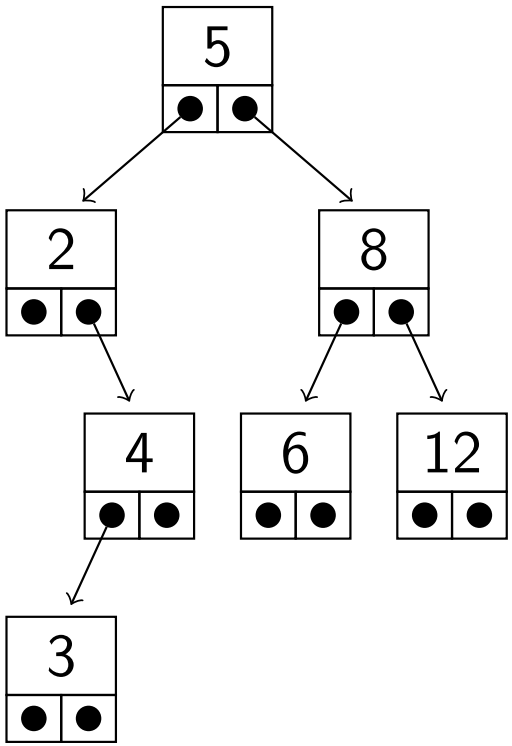
Tree contains 6?

$6 > 5$ so go right.

$6 < 8$ so go left.

6 is in the tree.

Implementing `contains()` (1)



Tree contains 6?

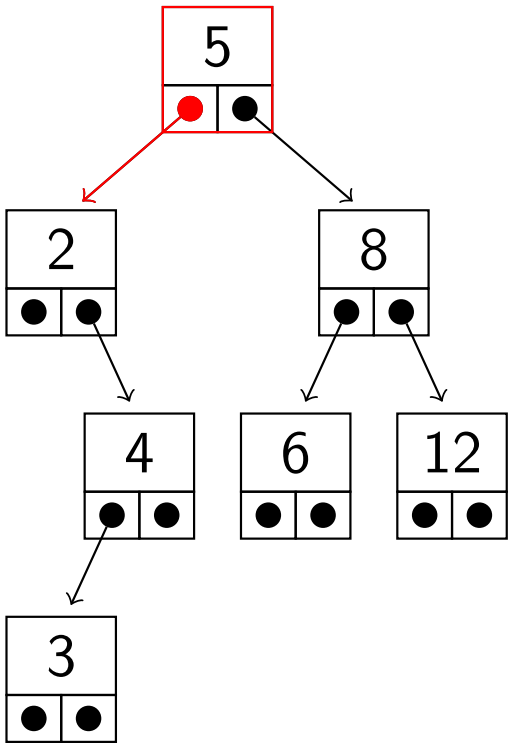
$6 > 5$ so go right.

$6 < 8$ so go left.

6 is in the tree.

Tree contains 1?

Implementing `contains()` (1)



Tree contains 6?

$6 > 5$ so go right.

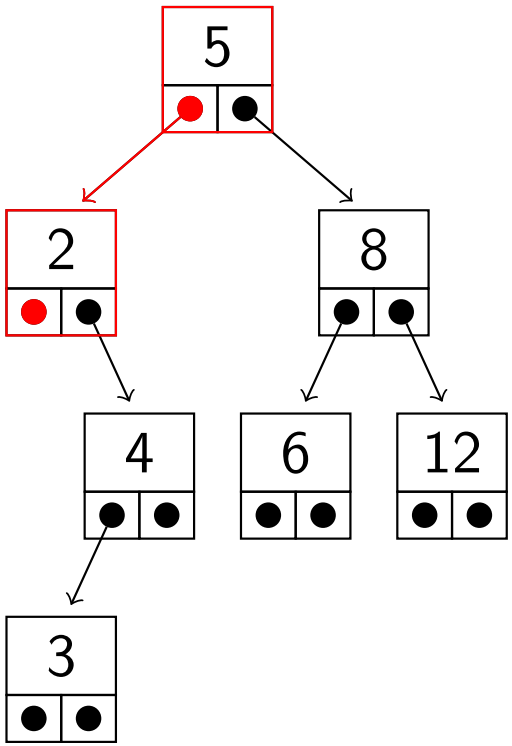
$6 < 8$ so go left.

6 is in the tree.

Tree contains 1?

$1 < 5$ so go left.

Implementing `contains()` (1)



Tree contains 6?

$6 > 5$ so go right.

$6 < 8$ so go left.

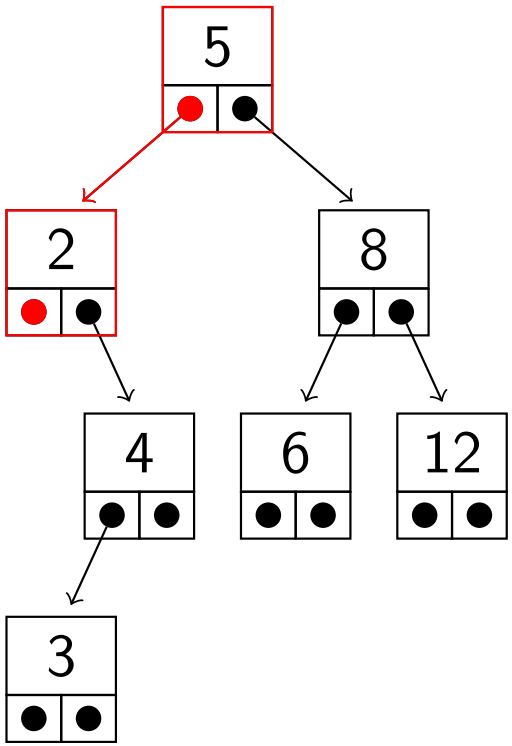
6 is in the tree.

Tree contains 1?

$1 < 5$ so go left.

$1 < 2$ but no left child.

Implementing `contains()` (1)



Tree contains 6?

$6 > 5$ so go right.

$6 < 8$ so go left.

6 is in the tree.

Tree contains 1?

$1 < 5$ so go left.

$1 < 2$ but no left child.

1 is not in the tree.

Java implementation of contains()

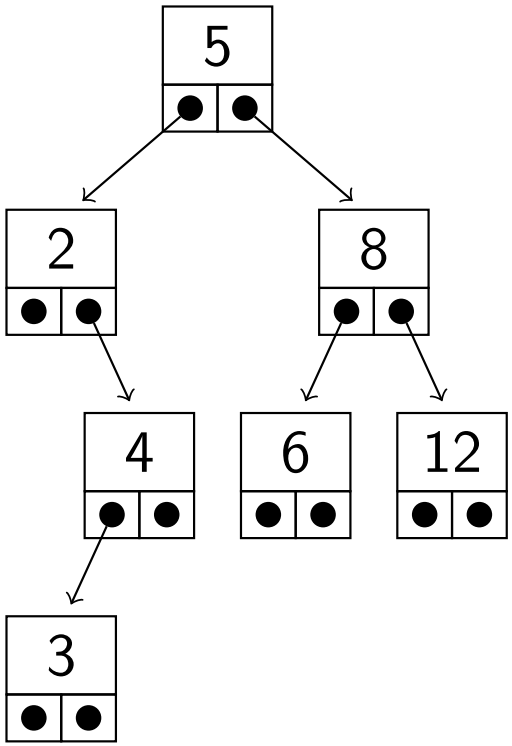
```
boolean contains (int i) {
    Node cur = root;
    while (cur != null) {
        if (i == cur.value)
            return true;
        else if (i < cur.value)
            cur = cur.left;
        else
            cur = cur.right;
    }
    return false;
}
```

Implementing `insert()`

`insert()` is very similar to `contains()`.

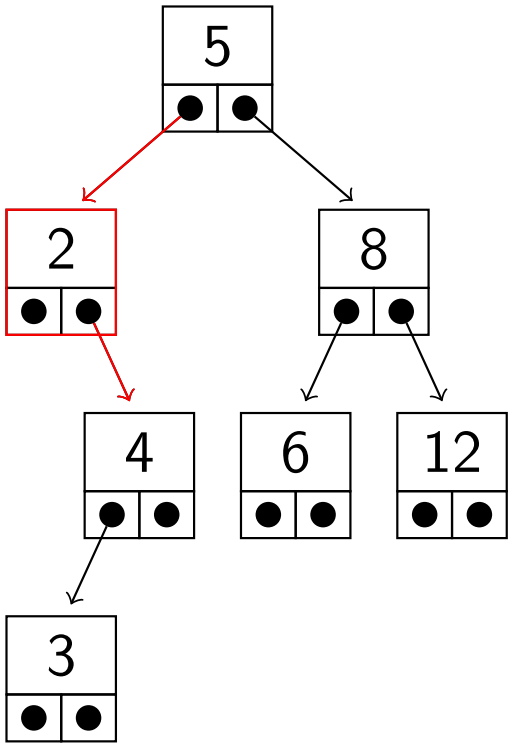
- Search for the value in the tree.
- If it is there, do nothing.
- If not, create a new child node at the node where the search failed.

Deletion 1: leaf or node with one child




```
delete(2);
```

Deletion 1: leaf or node with one child

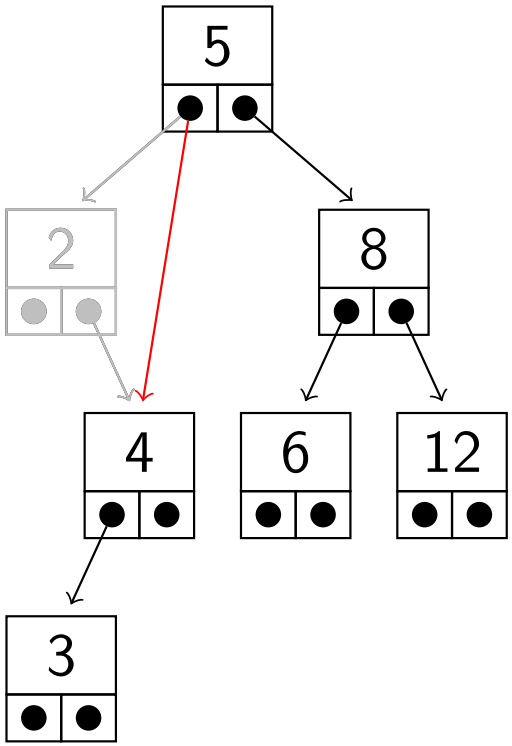


`delete(2);`

Find the node.

It, its parent and child look like a list.

Deletion 1: leaf or node with one child



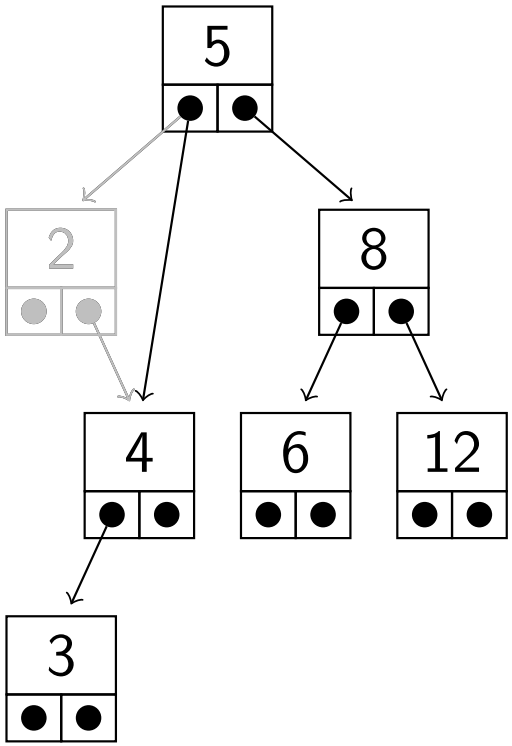
`delete(2);`

Find the node.

It, its parent and child look like a list.

Delete as if was a list.

Deletion 1: leaf or node with one child



`delete(2);`

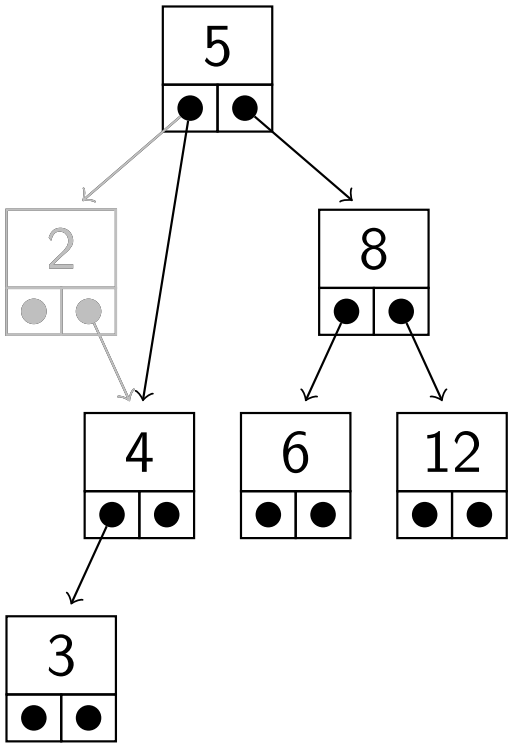
Find the node.

It, its parent and child look like a list.

Delete as if was a list.

The result is a valid BST.

Deletion 1: leaf or node with one child



`delete(2);`

Find the node.

It, its parent and child look like a list.

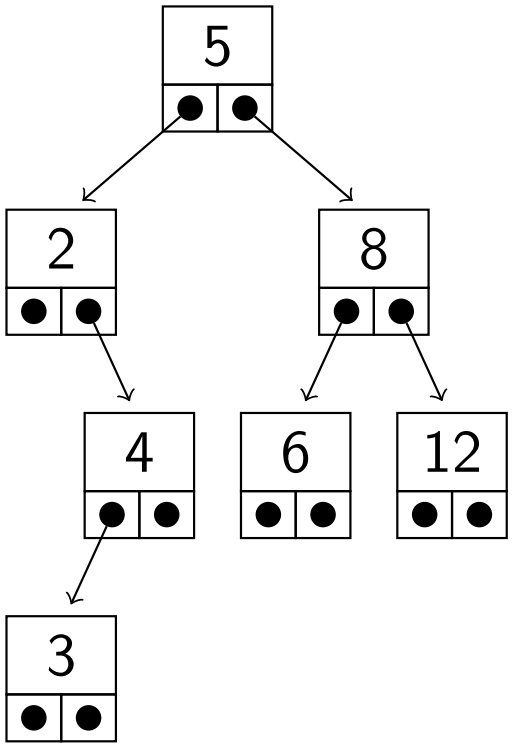
Delete as if was a list.

The result is a valid BST.

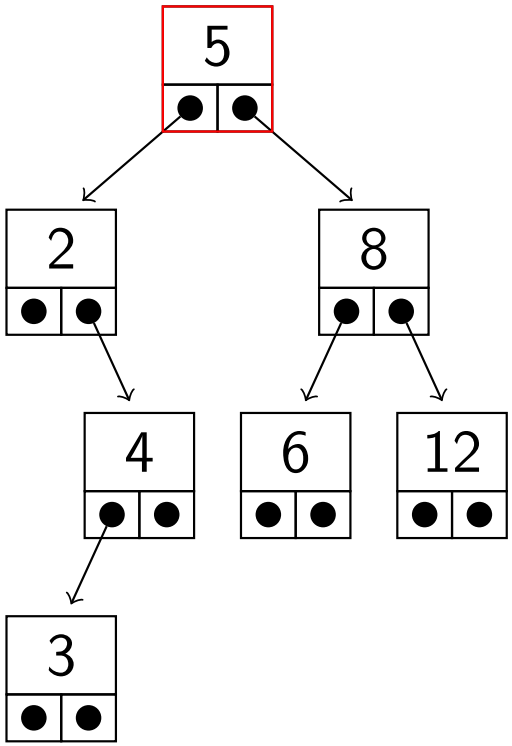
Same procedure applies if the node is the root (and has one child or none) or a leaf (no child by definition).

Deletion 2: node with two children

```
delete(5);
```



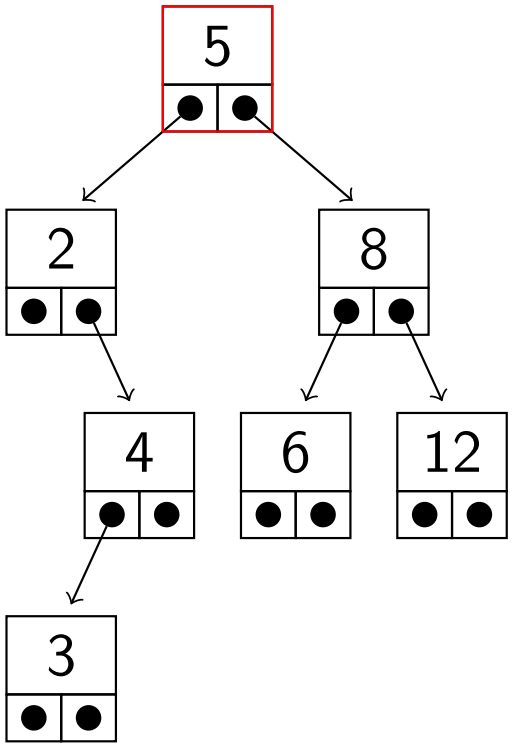
Deletion 2: node with two children




```
delete(5);
```

Find the node.

Deletion 2: node with two children

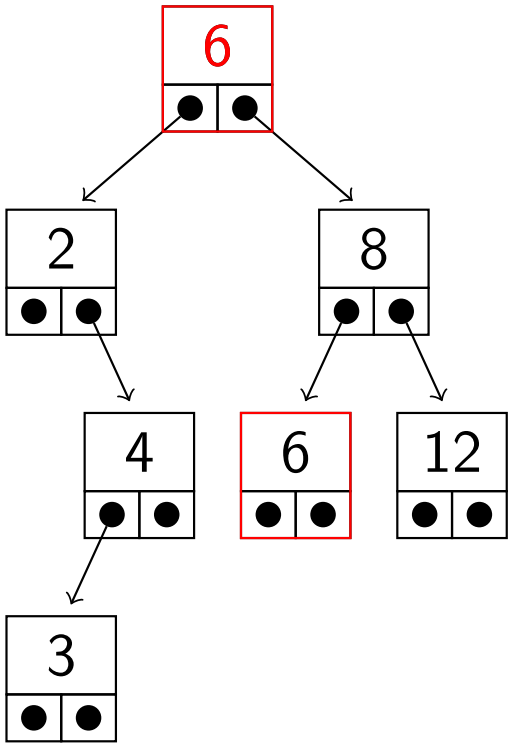



```
delete(5);
```

Find the node.

We'll replace its value with another from the tree.

Deletion 2: node with two children



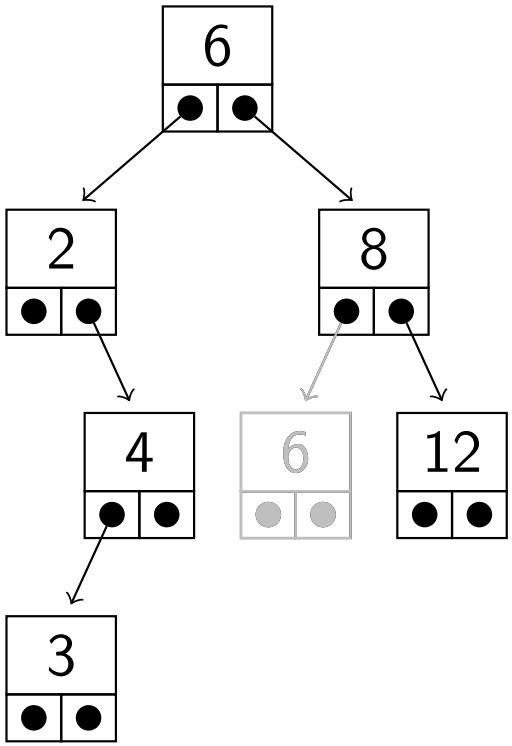
`delete(5);`

Find the node.

We'll replace its value with another from the tree.

New value must be greater than the left subtree and less than the right – use smallest value in right subtree.
The tree is temporarily invalid.

Deletion 2: node with two children



`delete(5);`

Find the node.

We'll replace its value with another from the tree.

New value must be greater than the left subtree and less than the right – use smallest value in right subtree.
The tree is temporarily invalid.

Delete the node whose value was copied (it cannot have 2 children so no cascading deletions).

Java implementation of `delete()`

The code is very fiddly; see Moodle.

You should focus on understanding the algorithm

Binary search trees – summary

Binary search trees implement sets with that support fast insertion, deletion and membership queries.

"Fast" typically means time proportional to $\log_2(\mathrm{size})$, but proportional to size in the worst case.

Can store any datatype, as long as it has a concept of "less than".

Priority queues

Priority queues

Priority queues store a set of items, each associated with a priority (i.e., "importance").

For us,

- priority will be an integer;
- smaller number = higher priority

The priority queue ADT

Operations and behaviour.

- `void create()` makes a new empty priority queue.
- `boolean isEmpty()` returns whether the queue is empty.
- `int size()` returns the number of items in the queue.
- `void insert(int p, Object obj)` inserts `obj` with priority `p`.
- `Object next()` returns the highest-priority item.

Running time `insert()` and `next()` run in time proportional to $\log_2(\text{size})$.

Simplified priority queue ADT

For simplicity, we'll just store the priorities.

Simplified operations and behaviour.

- `void create()` makes a new empty priority queue.
- `boolean isEmpty()` returns whether the queue is empty.
- `int size()` returns the number of items in the queue.
- `void insert(int p)` inserts "something" with priority `p`.
- `int next()` returns highest priority in the queue.

Running time `insert()` and `next()` run in time proportional to $\log_2(\text{size})$.

Data structure choice

As with binary search trees, implementing as sorted lists or arrays isn't fast enough:

- `next()` could run in constant time,
- but `insert()` takes time proportional to size.

Again, we use a form of binary trees.

Binary min-heaps

Binary min-heaps are binary trees with:

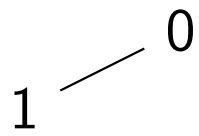
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.

0

Binary min-heaps

Binary min-heaps are binary trees with:

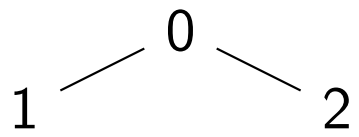
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

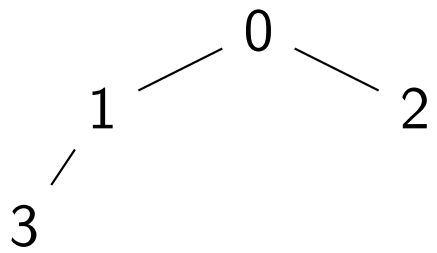
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

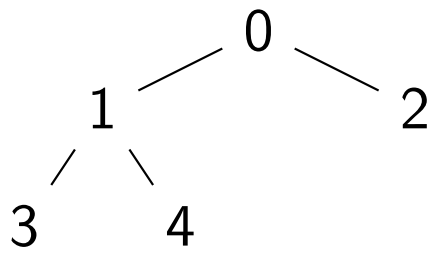
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

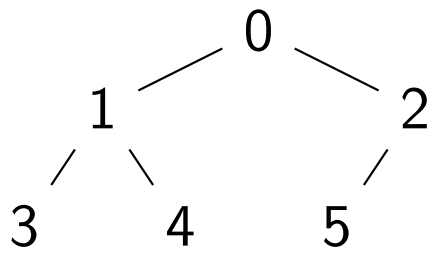
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

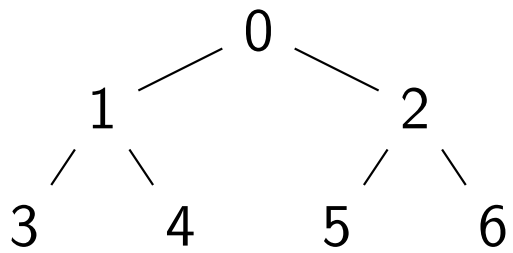
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

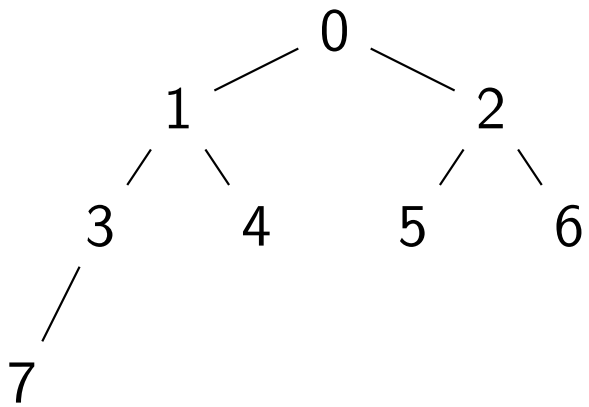
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Binary min-heaps

Binary min-heaps are binary trees with:

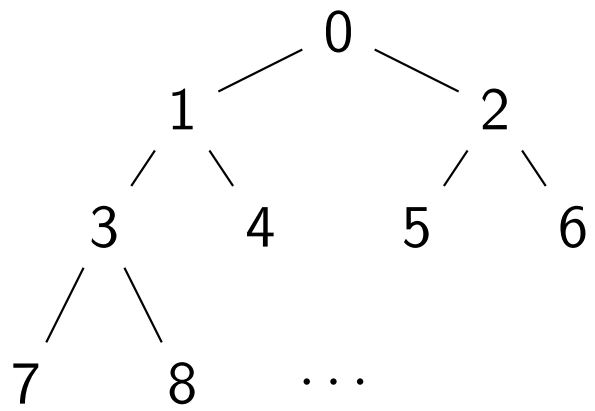
- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



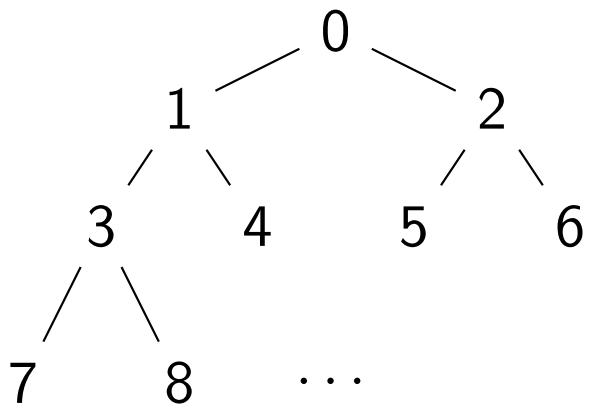
Binary min-heaps

Binary min-heaps are binary trees with:

- **heap property**: every node is less than everything in its subtrees.
- **shape property**: every level of the tree is full, except maybe the last. The nodes of the last level are filled left to right.



Storing the tree

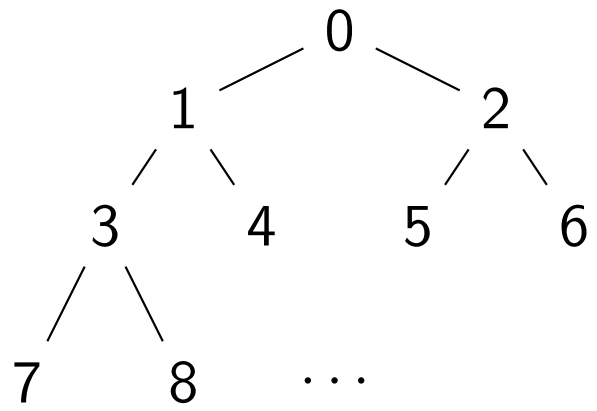


We always fill the tree in the order shown.

It is convenient to represent the tree **implicitly** in an array: cell i of the array holds the node at position i.

(Position i, **not** priority i.)

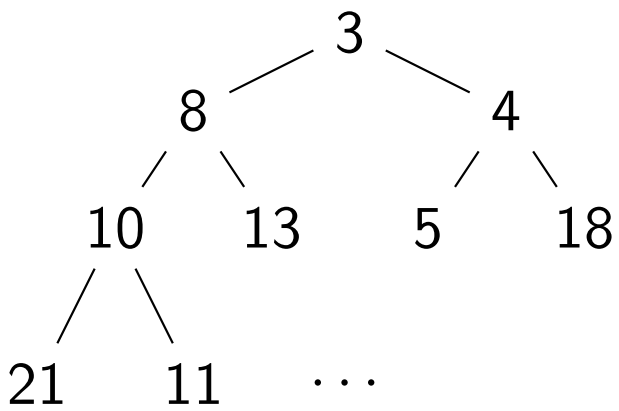
Array representation



- The **left child** of position i is at position $2i + 1$.
- The **left child** of position i is at position $2i + 2$
- The **parent** of position i is at position $(i - 1)/2$.

(These numbers would be easier if Java indexed arrays from 1. ;-))

Array representation example

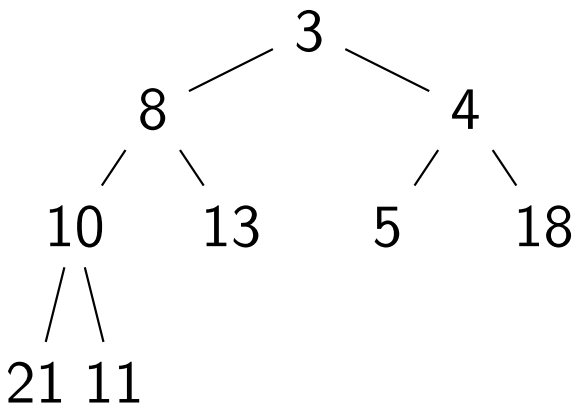


Stored as the array $\{3, 8, 4, 10, 13, 5, 18, 21, 11\}$.

Java implementation (1)

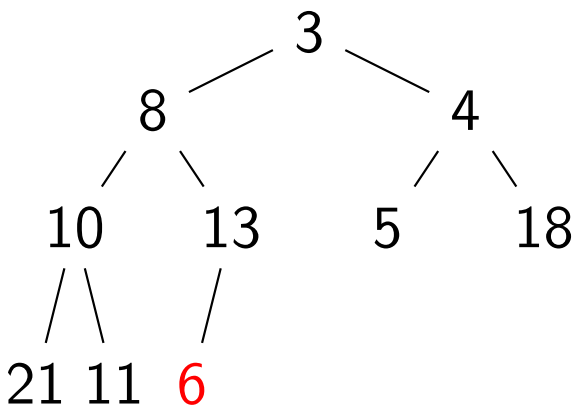
```
public class PriorityQueue {
    public class EmptyException extends RuntimeException {}

    private int[] items = new int[10];
    int size = 0;

    private static int leftChild (int index){
        return 2*index + 1;
    }
    private static int rightChild (int index) {
        return 2*index + 2;
    }
    private static int parent (int index) {
        return (index-1)/2;
    }
...
}
```

Insertion



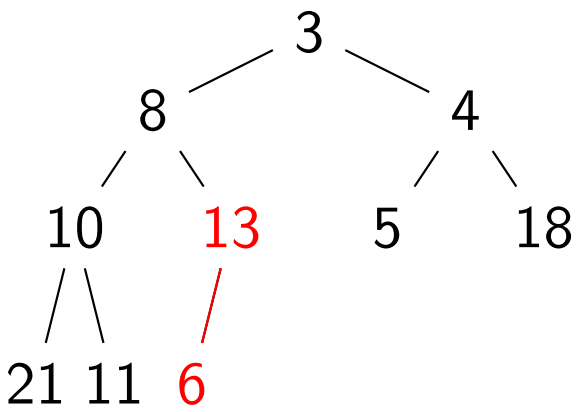
To insert, e.g., 6;

Insertion



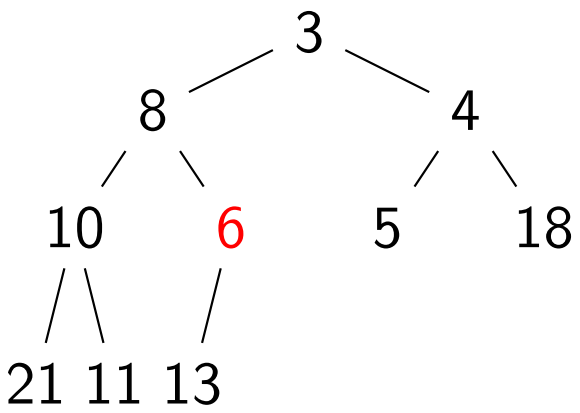
To insert, e.g., 6;
Place in next available position.

Insertion



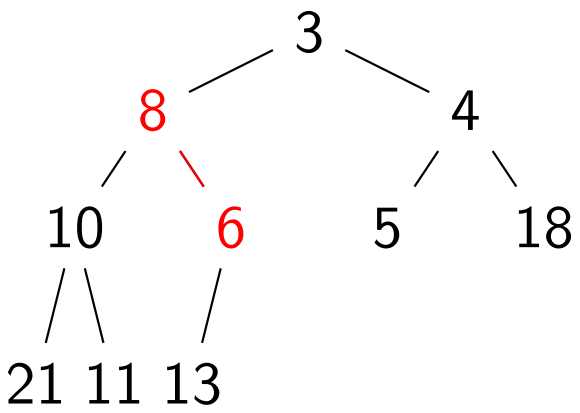
To insert, e.g., 6;
Place in next available position.
Compare with parent.

Insertion



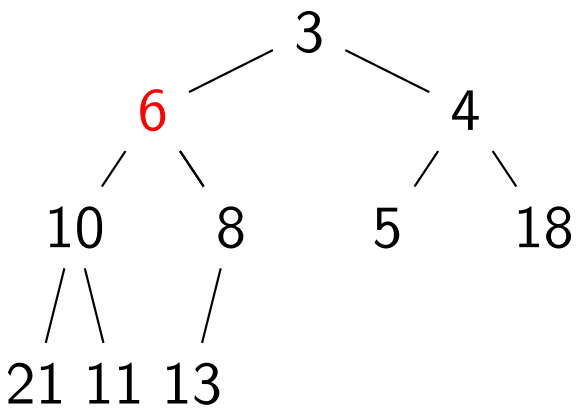
To insert, e.g., 6;
Place in next available position.
Compare with parent.
If less than parent, swap.

Insertion



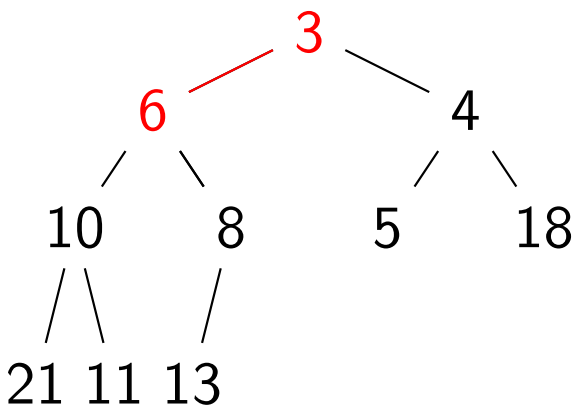
To insert, e.g., 6;
Place in next available position.
Compare with parent.
If less than parent, swap.
Compare with new parent.

Insertion



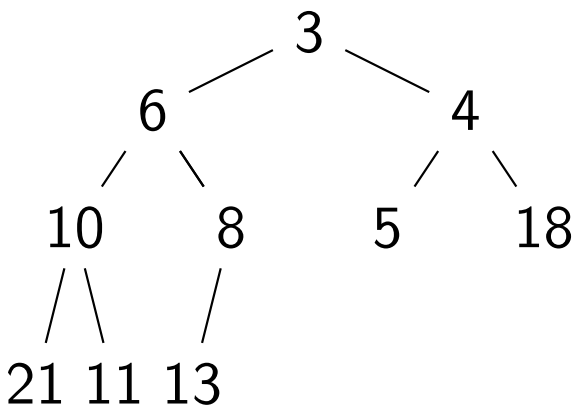
To insert, e.g., 6;
Place in next available position.
Compare with parent.
If less than parent, swap.
Compare with new parent.
If less than parent, swap.

Insertion



To insert, e.g., 6;
Place in next available position.
Compare with parent.
If less than parent, swap.
Compare with new parent.
If less than parent, swap.
Stop when not less than parent.

Insertion



To insert, e.g., 6;
Place in next available position.
Compare with parent.
If less than parent, swap.
Compare with new parent.
If less than parent, swap.
Stop when not less than parent.

Time taken is proportional to the height of the tree, which is $\log_2(\text{size})$.

Java implementation (2)

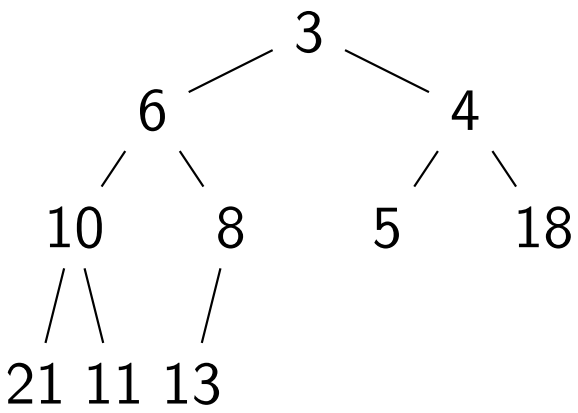
This helper method swaps the entries at positions `index1` and `index2`.

```
private void swap (int index1, int index2) {
    int temp = items[index1];
    items[index1] = items[index2];
    items[index2] = temp;
}
```

Java implementation (3)

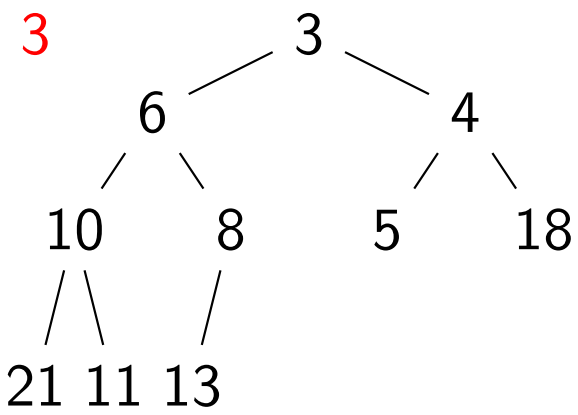
```
public void insert (int p) {
    // Extend array if full: same as ArrayStack.

    int cur = size;

    items[size++] = p;
    while (cur > 0 && items[cur] < items[parent(cur)]) {
        swap (cur, parent(cur));
        cur = parent(cur);
    }
}
```

next()



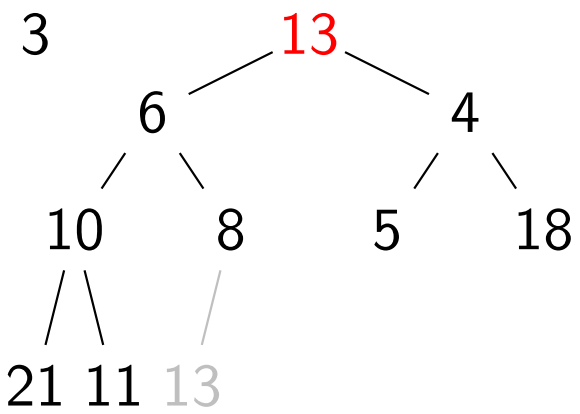
To implement `next()`

next()



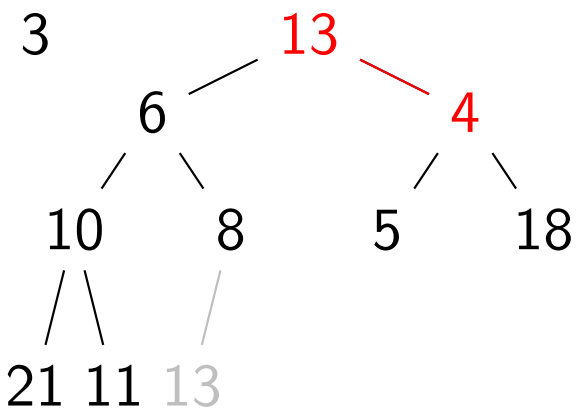
To implement `next()`
Remember the value at the root.

next()



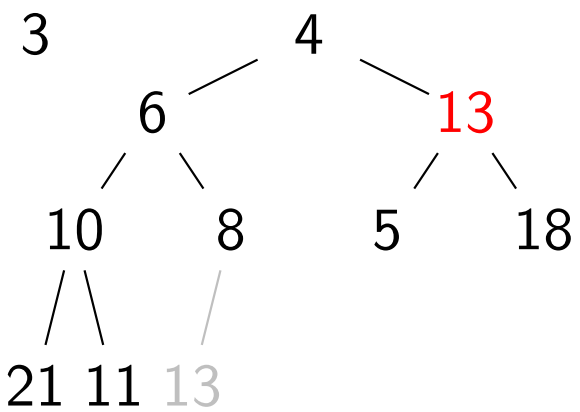
To implement `next()`
Remember the value at the root.
Replace root with last leaf.

next()



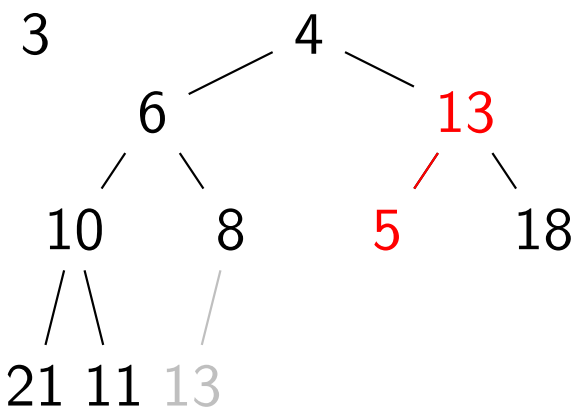
To implement `next()`
Remember the value at the root.
Replace root with last leaf.
While greater than either child,
swap with smallest

next()



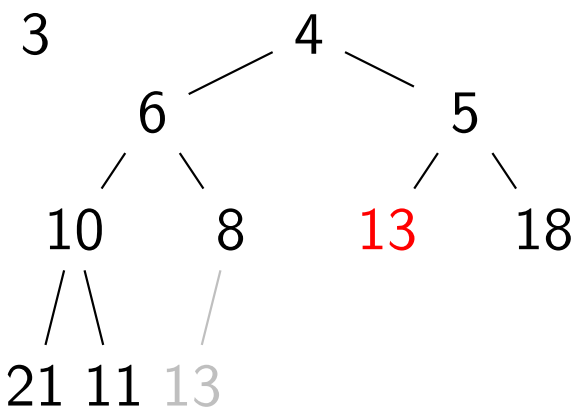
To implement `next()`
Remember the value at the root.
Replace root with last leaf.
While greater than either child,
swap with smallest

next()



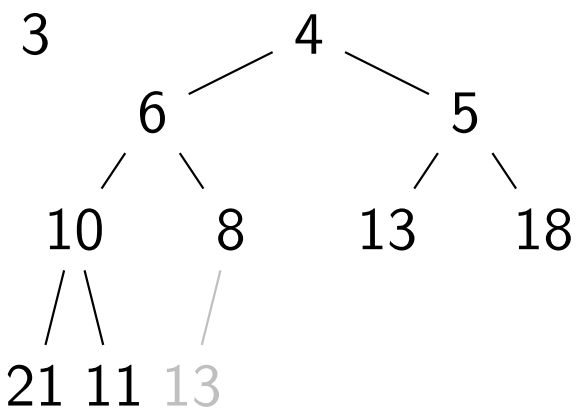
To implement `next()`
Remember the value at the root.
Replace root with last leaf.
While greater than either child,
swap with smallest

next()



To implement `next()`
Remember the value at the root.
Replace root with last leaf.
While greater than either child,
swap with smallest

next()



To implement `next()`
Remember the value at the root.
Replace root with last leaf.
While greater than either child,
swap with smallest

Time taken is proportional to the height of the tree, which is $\log_2(\text{size})$.

Java implementation (4.1)

```
public int next () throws EmptyException {
    if (isEmpty())
        throw new EmptyException ();

    int next = items[0];
    size--;

    items[0] = items[size];
    ...
```

Java implementation (4.2)

```
        ...
        int cur = 0;
        while (true) {
            if (smallestInFamily (cur) == rightChild (cur)) {
                swap (cur, rightChild (cur));
                cur = rightChild (cur);
            } else if (smallestInFamily (cur) == leftChild (cur))
                {
                swap (cur, leftChild (cur));
                cur = leftChild (cur);
            } else
                break;
        }
        return next;
    }
```

`smallestInFamily(i)` returns the index of the smallest value among node `i` and any children it has.

Uses of priority queues

- Any time you want to process items smallest/biggest first.
- e.g. informed search algorithms such as A* (see CE213).
- e.g. minimum spanning tree algorithms (see later in CE204).
- Event-based simulators, where events are scheduled to occur at times in the future.

Priority queues – summary

- Store a dynamic set of objects with priorities.
- Main operations are get highest-priority item and insert.
- Operations run in time $\propto \log_2(\text{size})$.
- The tree is implicity implemented in an array.