Binary Trees

Binary trees

A tree is a binary tree if each node has *at most 2* children.

- What is the maximal size of a binary tree of height *n*?
- How many leaves such a tree has?
- What is the minimal height of a binary tree of size *n*?
- What is the minimal height of a binary tree with *n* vertices?

5

1. $2^{n+1} - 1$:

Why is it not $2^n - 1$?

- A **full binary tree** of height n has nodes at levels 0 through n.
- Each level i has 2^i nodes.

- The total number of nodes is the sum of a geometric series:

$$1 + 2 + 4 + \cdots + 2^n = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

- If it were $2^n - 1$, it would imply the tree has height n−1.

2. 2^n:

   Why?

   - A full binary tree of height n has its leaves at depth n.
   - The number of nodes at depth n is 2^n, as each internal node doubles its children.

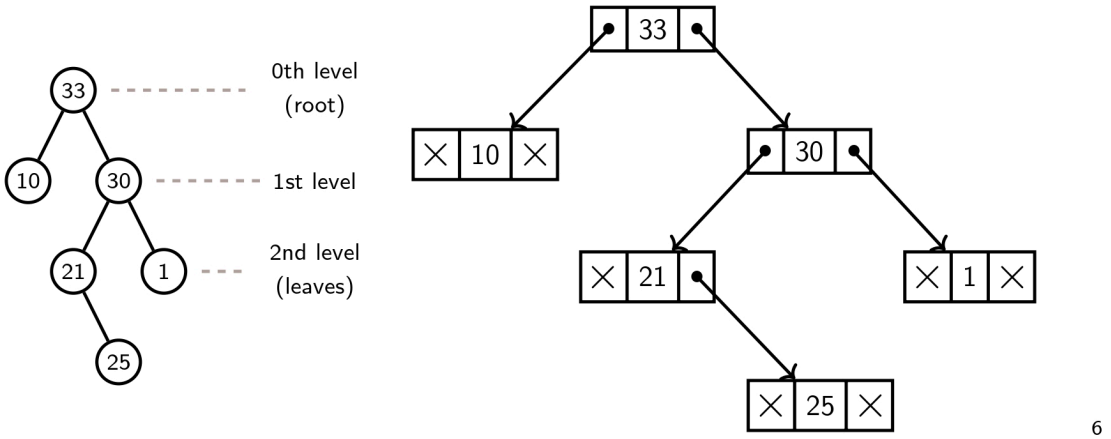
3. $\lceil \log_2(n-1) \rceil$:

Why?

- A perfectly balanced binary tree has the least height.
- The number of nodes in a full binary tree of height h is at most $2^{h+1} - 1$
- Solving $2^{h+1} - 1 \geq n$ for h, we get $h \geq \log_2(n-1)$, so the minimal height is the ceiling function $\lceil \log_2(n-1) \rceil$

Binary tree as a data structure

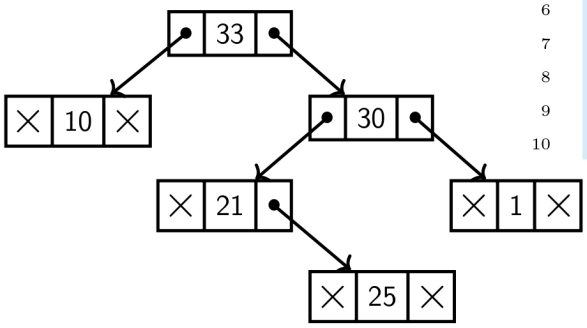
A tree is a very flexible and powerful data structure that, like a linked list, involves connected nodes, but has a hierarchical structure instead of the linear structure of linked lists.



6

Binary Tree: Code example flattening a tree into a list

```java
class Node {
  int val;
  Node left;    // left child
  Node right;   // right child
}
```




```java
int[] flatten(Node tree) {
  if (tree == null) {
    return new int[] {};
  } else {
    return appendArrays(
      flatten(tree.left),
      new int[] {tree.val},
      flatten(tree.right));
  }
}
```

$\mapsto [10, 33, 21, 25, 30, 1]$

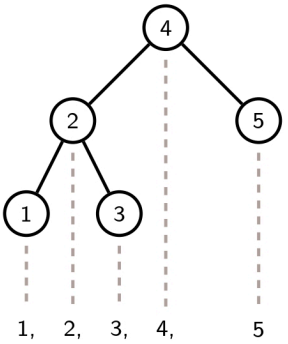
7

Binary Search Trees

Binary Search Trees

A type search tree is either **empty**, or

1. values in the left subtree are **smaller** than in the root,
2. values in the right subtree are **larger** than in the root, and
3. root's left and right subtrees are also *binary search trees*.



$\Longrightarrow$ values in the flattened Binary Search Tree are in order!

A Binary Search Tree is a Binary Tree, so the same constructors and accessors apply. It is just that there is an extra constraint that the node value ordering must be maintained during construction and manipulation.

1

Searching Binary Search Trees

Starting from the root node, how do we determine whether a value `x` is in the tree?

If the tree is empty, `x` is not in the tree! Otherwise, compare `x` and the value stored in the root. There are three possibilities:

- They are equal $\Longrightarrow$ we found it!
- `x` is smaller $\Longrightarrow$ we search the left subtree.
- `x` is larger $\Longrightarrow$ we search the right subtree.

2

Binary Search Trees: Search

```
1  class BSTree {
2    Node root;
3
4    class Node {
5      int val;
6      Node left, right;
7
8      public Node(int val, Node lf, Node rt) {
9        this.val = val;
10       this.left = lf;
11       this.right = rt;
12     }
13   }
14
15   public isEmpty() {
16     return (this.root == null);
17   }
18
19   // more methods here
20 }
```

```
1  bool search(int value) {
2    curnode = this;
3    while (curnode != null
4        && curnode.val != value) {
5      if (curnode.val < value)
6        curnode = curnode.left;
7      else
8        curnode = curnode.right;
9    }
10   return (curnode != null);
11 }
```

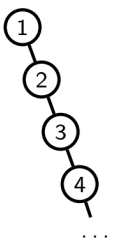
Time complexity: $O(\text{height})$!

Exercise:
Write a *recursive* implementation of `search`.

3

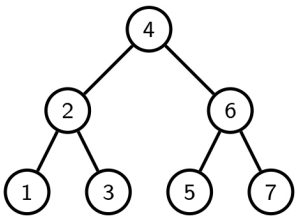
Searching Binary Search Trees

Compare complexities:



...

vs.



- For the left case, complexity of search is $O(n)$.
- For the right case, complexity of search is $O(\log n)$.
- For the average case (assuming the tree was created by a *random* sequence of insertions), complexity of search is also $O(\log n)$ (not proven here).
- Average height of a general binary tree is $O(\sqrt{n})$.

4

💡

Average Case Complexity of Search: $O(\log n)$

- The best-case scenario for a BST is a **perfectly balanced tree**, where each level has roughly **half** of the remaining nodes.
- If the BST is built from a **random sequence of insertions**, it is likely to be **reasonably balanced** rather than forming a skewed (linked-list-like) structure.
- In such a balanced BST, the height is approximately $O(\log n)$, meaning the search complexity is also $O(\log n)$, since at each step, we cut the problem size in half.
- Example: If you have $n = 1,000,000$ elements in a balanced BST, the search depth would be around $log_2(1,000,000) \approx 20$, which is much faster than searching in an unbalanced tree.

2. Average Height of a General Binary Tree: $O(\sqrt{n})$

- Unlike a well-balanced BST, a **general binary tree** is **not necessarily balanced**.
- If nodes are inserted randomly **without rebalancing**, the tree is expected to have a height of around $O(\sqrt{n})$ instead of $O(\log n)$.
- Intuition:
  - If nodes were completely random, the tree wouldn't be strictly balanced but also wouldn't form a straight line.
  - The height of such a tree follows the **random binary tree height distribution**, which is proven to be about $O(\sqrt{n})$.
- This is **worse** than the balanced case but **better** than the worst-case unbalanced case of O(n).

Comparison of BST Heights:

Type of BST	Worst-Case Height	Average-Case Height
Completely Skewed (like a linked list)	$O(n)$	$O(n)$

Balanced BST (ideal case)	$O(\log n)$	$O(\log n)$
General Binary Tree (random insertions)	$O(n)$	$O(\sqrt{n})$

Conclusion

- If a BST is **balanced**, search runs in $O(\log n)$
- If it's **randomly built**, it has an expected height of $O(\sqrt{n})$, which is **worse than log but much better than linear**.
- If it's **completely skewed**, search degrades to $O(n)$, which is why **self-balancing BSTs** $(\text{like AVL trees and Red-Black trees})$ are used in practice.

Binary Search Trees: Insertion

```
1  void insert(int value) {
2    if (this.root == null) { this.root = new Node(value, null, null); }
3    else { insert(value, this.root); }
4  }
5
6  private void insert(int value, Node ptr) {
7    if (value < ptr.val) {
8      if (ptr.left == null) { ptr.left = new Node(value, null, null); }
9      else { insert(value, ptr.left); }
10   }
11   else if (value > ptr.val) {
12     if (ptr.right == null) { ptr.right = new Node(value, null, null); }
13     else { insert(value, ptr.right); }
14   }
15   else          // ptr.val == value
16     throw new Error("Value already in tree");
17 }
```

5

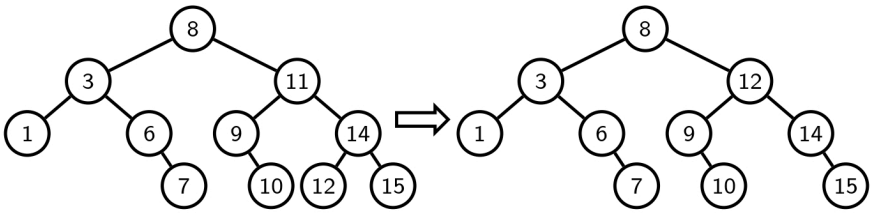
Deleting from a Binary Search Tree

- First Option:
  - Insert all items except the one to be deleted into a new BST
  - n inserts of the optimistic complexity $O(\log n)$, results in complexity $O(n \log n)$
  - Worse than deleting from an array: $O(n)$
- Second Option:
  1. Find the node containing the element to be deleted:
  2. If it is a leaf, just remove it.
  3. Else, if only one of the node's children is not empty, replace the node with the root of the non-empty subtree.
  4. Else,
     - 4.1 find the left-most node in the right sub-tree (this contains the smallest value in the right sub-tree);
     - 4.2 replace the value to be deleted with that of the left-most node;
     - 4.3 replace the left-most node with its right child (may be empty).
  - Complexity $O(\log n)$

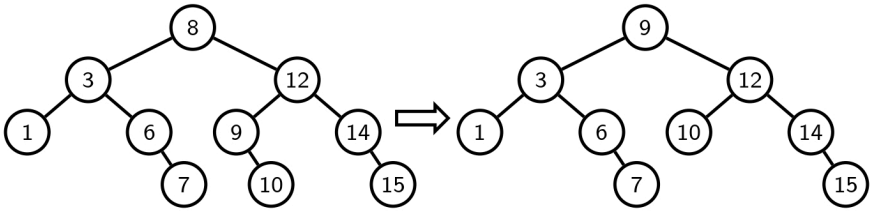
6

Deleting from a Binary Search Tree

Delete 11:



Delete 8:



7

Binary Search Trees: Deletion

```
Node delete(int value, Node tree) {
  if (tree == null)
    throw new Error("Value not in tree");

  if (value < tree.val) {
    delete(value, tree.left);
    return tree;
  } else if (value > tree.val) {
    delete(value, tree.right);
    return tree;
  } else if (tree.left == null)
      return tree.left;
  else if (tree.right == null)
      return three.left;
  else {
    smallest = smallest_node(tree.right);
    return Node(smallest,
                tree.left,
                delete(smallest, tree.right));
  }
}
```

```
int smallest_node(Node tree) {
  // Precondition: tree is non-empty!
  if (tree.left == null)
     return tree.val;
  else
     return smallest_node(tree.left);
}
```

8

Exercise: Sorting using BSTs

1. Given a BST `tree`, print all of its values **in order**!
2. Modify your solution to output a **sorted array** of all its values.
3. Use the above together with the insertion function to sort an array.
4. **Challenge!** What is the complexity of this sorting algorithm? In the worst case and in average case (assuming all orders are equally likely).

9