# CSE 250: Binary Search Trees

## Lecture 25

Oct 30, 2023

# Reminders

- PA2: Implement **Map Routing**
  1. Create an adjacency list (discussed today)
  2. Find a path from A to B with the fewest intersections
  3. Find a path from A to B with the shortest distance
- PA2 implementation due Sun, Nov 5 at 11:59 PM

# Trees

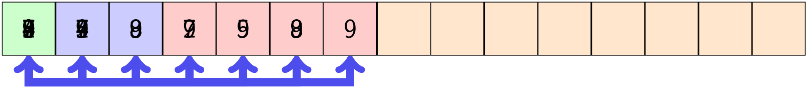
- **Child**
  An adjacent node connected by an out-edge
- **Leaf**
  A node with no children
- **Depth** of a node
  The number of edges from the root to the node
- **Depth** of a tree
  The maximum depth of any node in the tree
- **Level** of a node
  The depth $+ 1$

# Collections

- **Sequence/List**
  An **ordered** collection of **non-unique** elements
- **Set**
  An **unordered** collection of **unique** elements
- **Bag**
  An **unordered** collection of **non-unique** elements

# Heap Sort

**Inputs:** 7 4 8 2 5 3 9



**Outputs:** 2 3 4 5 7 8 9

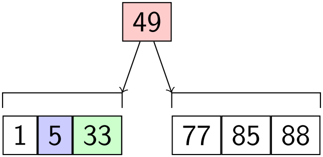
# Heaps

- **Binary** Tree
  - Each element has (at most) 2 children.
- **Heap** Constraint
  - Each node is lesser than its descendants.
- **Complete** Tree
  - Each level (except the last) is full.

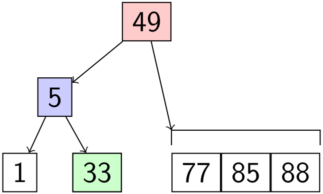
# Binary Search

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 5 | 33 | 49 | 77 | 85 | 88 |

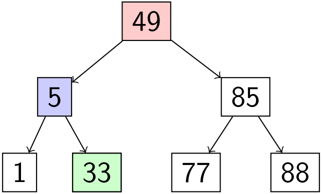
# Binary Search



# Binary Search



# Binary Search [Tree]



# Binary Search Trees

- **Binary** Tree
  - Each element has (at most) 2 children.
- **Binary Search Tree** Constraint
  - Each node has a value.
  - Each node's value is greater than its left descendants
  - Each node's value is lesser than (or equal to) its right descendants
- **Set** Constraint [optional]
  - Each node's value is unique.

We'll work with sets at first.

# Binary Search Trees

```java
public class TreeNode<T>
{
  T value;
  Optional< TreeNode<T> > left = Optional.empty();
  Optional< TreeNode<T> > right = Optional.empty();

  /* ... */
}
```

# Binary Search Trees - Find

**Find**

For the current node (starting at the root):

- Is the target element...
  - ... equal to the value at this node?
    - Return the value at this node
  - ... lesser than the value at this node?
    - Recur down the left tree
  - ... greater than the value at this node?
    - Recur down the right tree

# Binary Search Trees - Find

```
public Optional<T> find(T elem)
{
  if(elem.equals(value)){ return Optional.of(value); }
  if(elem.compareTo(value) < 0){
    if(left.isPresent){  return left.get().find(elem); }
    else {               return Optional.empty(); }
  } else {
    if(right.isPresent){ return right.get().find(elem); }
    else {               return Optional.empty(); }
  }
}
```

# Binary Search Trees - Find

What's the worst-case (Big-$O$) complexity of `find`?

$$T(\texttt{node}) < \begin{cases} 0 & \textbf{if } \texttt{node.isEmpty} \\ 1 + \max(T(\texttt{node.left}), \\ \qquad T(\texttt{node.right})) & \textbf{otherwise} \end{cases}$$

This is the **depth** of the tree.

# Binary Search Trees - Insert

**Insert**

For the current node (starting at the root):

- Is the target element...
    - ... equal to the value at this node?
        - Ignore: No duplicates in a **set**
          (If **bag**, recur right instead)
    - ... lesser than the value at this node?
        - If the left tree is empty, insert there
        - Otherwise, recur down the left tree
    - ... greater than the value at this node?
        - If the right tree is empty, insert there
        - Otherwise, recur down the right tree

# Binary Search Trees - Insert

```
public void insert(T elem)
{
  if(elem.equals(value)){ return; }
  if(elem.compareTo(value) < 0){
    if(left.isPresent){ return left.get().find(elem); }
    else { left = Optional.of(new TreeNode(elem));
           return; }
  } else {
    if(right.isPresent){ return right.get().find(elem); }
    else { right = Optional.of(new TreeNode(elem));
           return; }
  }
}
```

# Binary Search Trees - Insert

What's the worst-case (Big-$O$) complexity of `insert`?

$$T(\texttt{node}) < \begin{cases} 0 & \textbf{if } \texttt{node.isEmpty} \\ 1 + \max(T(\texttt{node.left}), \\ \qquad\qquad T(\texttt{node.right})) & \textbf{otherwise} \end{cases}$$

This is the **depth** of the tree.

# Binary Search Trees - Remove

For the current node (starting at the root):

- Is the target element...
  - ... equal to the value at this node?
    - Remove the node from its parent
    - Replace it with the left subtree
    - Insert the right subtree under the left subtree
  - ... lesser than the value at this node?
    - If the left tree is empty, insert there
    - Otherwise, recur down the left tree
  - ... greater than the value at this node?
    - If the right tree is empty, insert there
    - Otherwise, recur down the right tree

# Binary Search Trees - Remove

What's the worst-case (Big-$O$) complexity of `remove`?

- Find the node to remove — $O(\text{depth})$
- Reinsert the right subtree
    1. Option 1: Insert every element in the right subtree — $O(N)$
    2. Option 2: Insert the right subtree as a batch — $O(\text{depth})$

**Total:** $O(\text{depth})$

# Binary Search Trees

| Operation | Runtime |
|---|---|
| `find` | $O(d)$ |
| `insert` | $O(d)$ |
| `remove` | $O(d)$ |

What's this in terms of $N$? ($O(N)$)
Does it need to be that bad?