Trees
Introduction and Applications
Goals

- Tree Terminology and Definitions
- Tree Representation
- Tree Application
Examples of Trees
Trees

• Ubiquitous – they are everywhere in CS

• Probably ranks third among the most used data structure:
  1. Arrays/Vectors
  2. Linked Lists
  3. Trees
Tree Characteristics

- A tree consists of a collection of nodes connected by directed arcs.
- A tree has a single root node.
  - By convention, the root node is usually drawn at the top.
- A node that points to (one or more) other nodes is the parent of those nodes while the nodes pointed to are the children.
- Every node (except the root) has exactly one parent.
- Nodes with no children are leaf nodes.
- Nodes with children are interior nodes.
Tree Characteristics

Directed Arcs

Nodes

Interior Nodes: have one or more children

Leaf Nodes: have no children
Tree Characteristics

- **b**, **c**, **d**, **f**, **g**, **h**, **i**
- **a**
- **Siblings**
- **Subtree rooted at node d**
- **Descendants of node d**
Tree Characteristics

- Nodes that have the same parent are *siblings*
- The *descendants* of a node consist of its children, and their children, and so on
  - All nodes in a tree are descendants of the root node *(except, of course, the root node itself)*
- Any node can be considered the root of a *subtree*
- A subtree rooted at a node consists of that node and all of its descendants
Tree Characteristics

- There is a single, unique path from the root to any node
  - Arcs don’t join together
- A path’s *length* is equal to the number of arcs traversed
- A node’s *height* is equal to the maximum path length from that node to a leaf node:
  - A leaf node has a height of 0
  - The height of a tree is equal to the height of the root
- A node’s *depth* is equal to the path length from the root to that node:
  - The root node has a depth of 0
  - A tree’s depth is the maximum depth of all its leaf nodes *(which, of course, is equal to the tree’s height)*
Tree Characteristics

Root:
- Height = 3
- Depth = 0

Depth = 3:
- Path length to node from the root

Height = 2:
- Path length to furthest leaf
Nodes $D$ and $E$ are children of node $B$.

Node $B$ is the parent of nodes $D$ and $E$.

Nodes $B$, $D$, and $E$ are descendents of node $A$ (as are all other nodes in the tree...except $A$).

$E$ is an interior node.

$F$ is a leaf node.
Are these trees?

Yes

No

No
• **Binary** Tree
  
  – Nodes have no more than two children
  – Children are generally referred to as “left” and “right”

• **Full** Binary Tree:
  
  – Every leaf is at the same depth
  – Every internal node has 2 children
  – Height of $h$ will have $2^{h+1} - 1$ nodes
  – Height of $h$ will have $2^h$ leaves
Perfect Full *Binary* Tree

Height of $h$ will have $2^h$ leaves
Height of $h$ will have $2^{h+1} - 1$ nodes

- $h = 1$
  - # leaves = 2
  - # nodes = 3

- $h = 2$
  - # leaves = 4
  - # nodes = 7

- $h = 3$
  - # leaves = 8
  - # nodes = 15

Perfect & Full
All leaves are at the same depth and all parents have 2 children
Complete Binary Tree

- Complete Binary Tree:
  full except for the bottom level which is filled from left to right
Not a Complete Binary Tree
Binary Tree Application: Animal Game

- **Purpose:** computer guesses an animal that you (the player) is thinking of using a sequence of questions
  - Internal nodes contain yes/no questions
  - Leaf nodes are animals (or answers!)

- **How do we build it?**
Binary Tree Application: Animal Game

Cat

Swim?

Yes

Fish

No

Cat

Swim?

Yes

Fish

No

Cat

Fly?

Yes

Bird

No

Cat
Initially, tree contains a single animal (e.g., a “cat”) stored in the root node

Guessing....

1. Start at root.

2. If internal node $\rightarrow$ ask yes/no question
   - Yes $\rightarrow$ go to left child and repeat step 2
   - No $\rightarrow$ go to right child and repeat step 2

3. If leaf node $\rightarrow$ guess “I know. Is it a …”:
   - If right $\rightarrow$ done
   - If wrong $\rightarrow$ “learn” new animal by asking for a yes/no question that distinguishes the new animal from the guess
Decision Tree

• If you can ask at most $q$ questions, the number of possible answers we can distinguish between, $n$, is the number of leaves in a full binary tree with height at most $q$, which is at most $2^q$

• Taking logs on both sides: $\log(n) = \log(2^q)$

• $\log(n) = q$ : for $n$ outcomes, we need $q$ questions

• **For 1,048,576 outcomes we need 20 questions**
Binary Search Trees

Concepts
Binary Search Tree

- Binary search trees are binary trees where every node’s value is:
  - Greater than all its descendents in the left subtree
  - Less than or equal to all its descendents in the right subtree
BST Bag: Contains Example

Alex

Abner
Abigail
Adam
Adela
Agnes

Angela
Alice
Allen
Audrey
Arthur

Object to find → Agnes
BST Bag: Add

- Do the same type of traversal from root to leaf
- When you find a null value, create a new node

```
  Alex
  /   \
Abner  Angela
   /     /   \
Abigail  Adela  Alice  Audrey
       /       /     /     \
   Adam  Agnes  Allen  Arthur
```
Before first call to `add`
BST Bag: Add Example

After first call to `add`

Next object to add: Ariel

“Ariel” should be added here
How would you remove Abigail? Audrey? Angela?
Who fills the hole?

• Answer: the leftmost child of the right subtree (smallest element in right subtree)

• Try this on a few values

• Alternatively: The rightmost child of the left subtree
Intuition...Remove 50
BST Bag: Remove Example

Before call to remove

Element to remove

Replace with: leftmost(right)
BST Bag: Remove Example

After call to **remove**
Special Case

• What if you don’t have a right child?

• Try removing “Audrey”
  – Simply return left child
• If tree is reasonably full (well balanced), searching for an element is $O(\log n)$. Why?
  
  – you’re dividing in half at each step: $O(\log n)$

• Alternatively, we are running down a path from root to leaf
  
  – We can prove by induction that in a complete tree (which is reasonably full), the path from root to leaf is bounded by floor($\log n$), so $O(\log n)$
• We’ve shown all Bag operations to be proportional to the length of a path, rather than the number of elements in the tree.
• We’ve also said that in a reasonably full tree, this path is bounded by: $\text{floor}(\log_2 n)$
• This Bag is faster than our previous implementations!
Comparison

- **Average Case Execution Times**

<table>
<thead>
<tr>
<th>Operation</th>
<th>DynArrBag</th>
<th>LLBag</th>
<th>Ordered ArrBag</th>
<th>BST Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1⁺)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Contains</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
Your Turn

• Worksheet28_BSTPractice
Binary Search Trees II
Bag Implementation
Goals

• BST Representation
• Bag Operations
• Functional-style operations
struct BSTree {
    struct Node *root;
    int cnt;
};

struct Node {
    TYPE val;
    struct Node *left;
    struct Node *right;
};

struct BSTree *createBSTree(struct BSTree *tree);
void addBSTree(struct BSTree *tree, TYPE val);
int containsBSTree(struct BSTree *tree, TYPE val);
void removeBSTree(struct BSTree *tree, TYPE val);
int sizeBSTree(struct BSTree *tree);
A useful trick (adapted from the functional programming world): Recursive helper routine that returns the tree with the value inserted

Node addNode(Node current, TYPE value)
if current is null then return new Node with value
otherwise if value < Node.value
  left child = addNode(left child, value)
else
  right child = addNode(right child, value)
return current node
Add "L"

k

Add (k, "L") =

m

Add (m, "L") =

v

Add (q, "L") =

a

Add (q, "L") =

Add(NULL, "L")

return L
void add(struct BSTree *tree, TYPE val) {
    tree->root = _addNode(tree->root, val);
    tree->cnt++;
}
Recursive Helper – functional flavor

```c
struct Node *addNode(struct Node *cur, TYPE val)
{
    struct Node *newnode;
    if (cur == NULL){
        /* base case goes here */
        return newNode
    }
    if (val < cur->val)
        /* recursive call left */
    else /*recursive call right */
        return cur;
}
```
def binary_tree_insert(node, key, value):
    if node is None:
        return TreeNode(None, key, value, None)
    if key == node.key:
        return TreeNode(node.left, key, value, node.right)
    if key < node.key:
        return TreeNode(binary_tree_insert(node.left, key, value), node.key, node.value, node.right)
    else:
        return TreeNode(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))
```java
public void insert(int data) {
    if (m_root == null) {
        m_root = new TreeNode(data, null, null);
        return;
    }
    Node root = m_root;
    while (root != null) {
        if (data == root.getData()) {
            return;
        } else if (data < root.getData()) {
            if (root.getLeft() == null) {
                root.setLeft(new TreeNode(data, null, null));
                return;
            } else {
                root = root.getLeft();
            }
        } else {
            if (root.getRight() == null) {
                root.setRight(new TreeNode(data, null, null));
                return;
            } else {
                root = root.getRight()
            }
        }
    }
}
```
Iterative (Java)

```java
else {
    if (root.getRight() == null) {
        root.setRight(new TreeNode(data, null, null));
        return;
    } else {
        root = root.getRight();
    }
}
```
How would you remove Abigail? Audrey? Angela?
Who fills the hole?

- **Answer:** the leftmost child of the right subtree (smallest element in right subtree)

- Useful to have a couple of private inner routines:

  ```c
  TYPE _leftmost(struct Node *cur) {
    ...
    /* Return value of leftmost child of current node. */
  }

  struct Node *_removeLeftmost(struct Node *cur) {
    ...
    /* Return tree with leftmost child removed. */
  }
  ```
Before call to **remove**
BST Remove Example

After call to **remove**
Node removeNode(Node current, TYPE value)
  if value = current.value
    if right child is null
      return left child
    else
      replace value with value in leftmost child of right subtree
      set right child to result of removeLeftmost(right)
  else if value < current.value
    left child = removeNode(left child, value)
  else right child = removeNode(right child, value)
  return current node
**Comparison**

- **Average Case Execution Times**

<table>
<thead>
<tr>
<th>Operation</th>
<th>DynArrBag</th>
<th>LLBag</th>
<th>Ordered ArrBag</th>
<th>BST Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>$O(1+)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Contains</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Space Requirements

• Does the functional-style recursive version require more or less space than an iterative version? Think about the call stack?
  – rebuilding as move up from recursion requires $O(\log n)$ space on average
Your Turn

• Complete the BST implementation in Worksheet #29