COMP 2140 - Data Structures

Shahin Kamali

Topic 8 - Binary Trees
University of Manitoba

Partially based on notes by S. Durocher.
Overview

- binary trees
- tree terminology
- implementing operations on binary trees

(see Open Data Structures, Chapter 6 for further reading)
Motivation

- Binary trees are extensions of linked lists
  - The memory is allocated dynamically (as opposed to arrays)
  - Using pointers helps us avoid maintaining large unused arrays.
Motivation

- Binary trees are extensions of linked lists
  - The memory is allocated dynamically (as opposed to arrays)
  - Using pointers helps us avoid maintaining large unused arrays.
- Binary trees are used in applications such as implementing dictionaries (using binary search trees) and priority queues (using heaps).
Binary Trees

Definition

A **binary tree** is either empty or consists of a node called a **root** and two binary trees called the **left subtree** and **right subtree**, one or both of which may be empty.

- Typically, a **key** is associated with each node, where each key is an element drawn from a well-ordered set.
Binary Tree

- The definition of a binary tree is a recursive structure
  - base case: The binary tree is empty.
  - recursive case: The binary tree consists of a node and two other (sub)trees.
Terminology

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.

- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.

- Node $a$ is the **parent** of nodes $b$ and $c$.

- Nodes $b$ and $c$ are **siblings** if they have the same parent.

- The **root** $a$ is the only node that has no parent.

- Nodes in the left and right subtrees of node $a$ are **descendants** of $a$.

- If node $d$ is a descendant of node $a$, then $a$ is an **ancestor** of $d$.

- A **leaf** is a node $h$ whose left and right subtrees are empty.
Terminology

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.

Nodes in the left and right subtrees of node $a$ are **descendants** of $a$.

If node $d$ is a descendant of node $a$, then $a$ is an **ancestor** of $d$.

A **leaf** is a node $h$ whose left and right subtrees are empty.
**Terminology**

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
**Terminology**

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
- Nodes $b$ and $c$ are **siblings** if they have the same parent.
Terminology

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
- Nodes $b$ and $c$ are **siblings** if they have the same parent.
- The **root** $a$ is the only node that has no parent.
Terminology

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
- Nodes $b$ and $c$ are **siblings** if they have the same parent.
- The **root** $a$ is the only node that has no parent.
- Nodes in the left and right subtrees of node $a$ are **descendants** of $a$. 
**Terminology**

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
- Nodes $b$ and $c$ are **siblings** if they have the same parent.
- The **root** $a$ is the only node that has no parent.
- Nodes in the left and right subtrees of node $a$ are **descendants** of $a$.
- If node $d$ is a descendant of node $a$, then $a$ is an **ancestor** of $d$.
Terminology

- Node $b$ is the **left child** of node $a$ if $b$ is the root of the left subtree of $a$.
- Node $c$ is the **right child** of node $a$ if $c$ is the root of the right subtree of $a$.
- Node $a$ is the **parent** of nodes $b$ and $c$.
- Nodes $b$ and $c$ are **siblings** if they have the same parent.
- The **root** $a$ is the only node that has no parent.
- Nodes in the left and right subtrees of node $a$ are **descendants** of $a$.
- If node $d$ is a descendant of node $a$, then $a$ is an **ancestor** of $d$.
- A **leaf** is a node $h$ whose left and right subtrees are empty.
Terminology

- The **path** from node $a$ to node $h$ is a sequence of nodes $n_1, n_2, \ldots, n_k$, where $n_i$ is the parent of $n_{i+1}$ (for example, $a \rightarrow b \rightarrow d \rightarrow h$).
The **path** from node $a$ to node $h$ is a sequence of nodes $n_1, n_2, \ldots, n_k$, where $n_i$ is the parent of $n_{i+1}$ (for example, $a \rightarrow b \rightarrow d \rightarrow h$).

The **length** of the path is the number of edges in the path. (Warning: Some texts use the number of nodes rather than the number of edges).
Terminology

- The **path** from node $a$ to node $h$ is a sequence of nodes $n_1, n_2, \ldots, n_k$, where $n_i$ is the parent of $n_{i+1}$ (for example, $a \rightarrow b \rightarrow d \rightarrow h$).

- The **length** of the path is the number of edges in the path. (Warning: Some texts use the number of nodes rather than the number of edges).

- The **depth** or **level** of a node $n$ is the length of the path from the root to $n$. The depth of the root is 0.
Terminology

- The **path** from node $a$ to node $h$ is a sequence of nodes $n_1, n_2, \ldots, n_k$, where $n_i$ is the parent of $n_{i+1}$ (for example, $a \rightarrow b \rightarrow d \rightarrow h$).

- The **length** of the path is the number of edges in the path. (Warning: Some texts use the number of nodes rather than the number of edges).

- The **depth** or **level** of a node $n$ is the length of the path from the root to $n$. The depth of the root is 0.

- The **height** of a node $n$ is the length of the longest path from $n$ to a leaf node. The height of a leaf node is 0. The height of a tree is the height of its root node.
Example

- height of this tree?
Example

- height of this tree?
- depth of the node with key \( f \):
Example

- height of this tree?
- depth of the node with key $f$:
- height of the node with key $c$: 
Example

- height of this tree?
- depth of the node with key $f$:
- height of the node with key $c$:
- number of nodes in this tree:
Example

- height of this tree?
- depth of the node with key $f$:
- height of the node with key $c$:
- number of nodes in this tree:
- number of leaves in this tree:
Example

- height of this tree?
- depth of the node with key \( f \):
- height of the node with key \( c \):
- number of nodes in this tree:
- number of leaves in this tree:
- number of internal (non-leaf) nodes in this tree:
Bounding the Size of a Tree

- Assume you have a tree of height \( h \)
- At least how many nodes are in the tree?
Bounding the Size of a Tree

- Assume you have a tree of height $h$
- At least how many nodes are in the tree?
  - There is at least $h + 1$ nodes in the tree
Assume you have a tree of height $h$

At least how many nodes are in the tree?
- There is at least $h + 1$ nodes in the tree

At most how many nodes are in the tree?
Assume you have a tree of height $h$.

At least how many nodes are in the tree?
- There is at least $h + 1$ nodes in the tree.

At most how many nodes are in the tree?
- The number of nodes at leave $i$ is at most $2^i$.
- There is at most $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$ nodes in the tree.
Bounding the Size of a Tree

- Assume you have a tree of height $h$
- At least how many nodes are in the tree?
  - There is at least $h + 1$ nodes in the tree
- At most how many nodes are in the tree?
  - The number of nodes at leave $i$ is at most $2^i$.
  - There is at most $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$ nodes in the tree

**Theorem**

There is at least $h + 1$ and at most $2^{h+1} - 1$ nodes in a tree of height $h$. 
Bounding the Height of a Tree

- Assume you have a tree of size $n$
- What is the minimum height that the tree can have?
Bounding the Height of a Tree

- Assume you have a tree of size $n$
- What is the minimum height that the tree can have?
  - The height is minimized when all levels are full.
  - For $h$ to be the height of the tree, we should have
    \[ 1 + 2 + \ldots + 2^h > n, \text{ i.e., } 2^{h+1} - 1 > n \text{ or } h > \log (n + 1) - 1. \]
Bounding the Height of a Tree

- Assume you have a tree of size $n$
- What is the minimum height that the tree can have?
  - The height is minimized when all levels are full.
  - For $h$ to be the height of the tree, we should have
    \[ 1 + 2 + \ldots + 2^h > n, \text{ i.e., } 2^{h+1} - 1 > n \]
    or $h > \log (n + 1) - 1$.
- What is the maximum height that the tree can have?
Bounding the Height of a Tree

- Assume you have a tree of size $n$
- What is the minimum height that the tree can have?
  - The height is minimized when all levels are full.
  - For $h$ to be the height of the tree, we should have $1 + 2 + \ldots + 2^h > n$, i.e., $2^{h+1} - 1 > n$ or $h > \log(n + 1) - 1$.
- What is the maximum height that the tree can have?
  - The height is maximized when there is only 1 node per level $\rightarrow n - 1$
Bounding the Height of a Tree

- Assume you have a tree of size $n$
- What is the minimum height that the tree can have?
  - The height is minimized when all levels are full.
  - For $h$ to be the height of the tree, we should have $1 + 2 + \ldots + 2^h > n$, i.e., $2^{h+1} - 1 > n$ or $h > \log (n + 1) - 1$.
- What is the maximum height that the tree can have?
  - The height is maximized when there is only 1 node per level $\rightarrow n - 1$

**Theorem**

The height of a tree of size $n$ is in the range $[\log (n + 1) - 1, n - 1]$. 
Number of Trees

- Given an integer $n$, there are many number of possible binary trees of size $n$.
- The **Catalan number** $b_n$ corresponds to the number of distinct binary tree structures of size $n$.

$$b(n) = \frac{1}{n + 1} \binom{2n}{n} = \frac{(2n)!}{(n + 1)! \ n!} \in O\left(\frac{4^n}{n^{3/2}}\right)$$

$$b_1 = 1, b_2 = 2, b_3 = 5, b_4 = 14, b_5 = 42, \ldots$$
Key Ordering

- The keys in a binary tree can be ordered in different ways:
  - **Binary Search Tree**: For any node in the tree, key(nodes in left-subtree) ≤ key(node) ≤ key(nodes right-subtree)
    - The keys are ordered from left to right.
    - Note, a binary tree is a data structure while a binary search tree is a binary tree whose keys have a specific order.
Key Ordering

- **Heap**: For any node in the tree, \( \text{key}(\text{node}) \leq \text{key}(\text{parent}(\text{node})) \)
  - The largest element is always at the root.
  - no particular left-to-right ordering
  - The keys along any path from the root to a leaf are in non-increasing order.
Non-Uniqueness of Binary Search Trees

- Given a set of keys, multiple different binary search trees are possible.

![Binary Search Trees Diagram]
Non-Uniqueness of Binary Search Trees

- Given a set of keys, multiple different binary search trees are possible.

- Given a fixed tree structure and a set of keys, the corresponding binary search tree is uniquely determined.
Given a set of keys, multiple different binary heaps are possible, even if the tree’s structure is fixed.
Height-balanced and Weight-balanced Trees

A tree is **height balanced** if for every node $v$ in the tree, the heights of the left and right subtrees of $v$ differ by at most one.
Height-balanced and Weight-balanced Trees

A tree is **height balanced** if for every node $v$ in the tree, the heights of the left and right subtrees of $v$ differ by at most one.

A tree is **weight balanced** if for every node $v$ in the tree, the numbers of nodes in the left and right subtrees of $v$ differ by at most one.

**Theorem**

*Any tree with $n$ nodes that is height balanced or weight balanced has height $O(\log n)$.***

We skip the proof here.
public class TreeNode {
    public int key; // node’s key
    public TreeNode left; // pointer to left child
    public TreeNode right; // pointer to right child
    public TreeNode(int newKey, TreeNode newLeft, TreeNode newRight) {
        key = newKey; // store key
        // set pointers to children
        left = newLeft;
        right = newRight;
    }

    public TreeNode(int newKey) {
        this(newKey, null, null);
    }

    public TreeNode() {
        this(0, null, null);
    }
}
Better, practical implementation

- In practice, your code should:
  - declare instance variables to be private or protected,
  - use get and set accessor and mutator methods,
  - have a generic type data member associated with each node,
  - have a parent pointer.
Public class TreeNode<E> {
    protected E data;
    // data stored at node
    protected int key;
    // node's key
    protected TreeNode<E> left, right, parent;

    public TreeNode(E newData, TreeNode<E> newLeft, TreeNode<E> newRight, TreeNode<E> newParent) {
        left = newLeft;
        right = newRight;
        parent = newParent;
        setData(newData);
    }

    public TreeNode(E newData) {
        this(newData, null, null, null);
    }

    public void setData(E newData) {
        data = newData;
        if (data != null)
            key = data.getKey();
        else
            key = -1;
    }

    public E getData() {
        return data;
    }
}
public class BinTree {
    private TreeNode treeRoot;
    public BinTree() { treeRoot = null; }
    ...
}

We implement the following operations on the classes BinTree and BST:

- height: return the height of the binary tree
- traverse: visit each node in the tree
- find: search the tree for a given key
- insert: add a key to the tree
- delete: remove a key from the tree
The height of a binary tree rooted at node $a$ can be found recursively by:

- calculating the height of the left subtree of $a$,
- calculating the height of the right subtree of $a$, and
- returning the maximum of the two, plus one.

We have $\text{height}(a) = 1 + \max\{\text{height}(a.\text{left}), \text{height}(a.\text{right})\}$

For the base case we have $\text{height}(\text{leaf}) = 0$ which is equivalent to having $\text{height}(\text{empty tree}) = \text{null}$
Measuring Height Example

\[
\text{heigh}(\text{node-9}) = 1 + \max\{\text{heigh}(\text{node-4}), \text{heigh}(\text{node-22})\} \\
= 1 + \max\{2, 3\} = 4.
\]
public int height() {
    return heightSubtree(treeRoot);
}

private int heightSubtree(
    TreeNode node) {
    /* return the height of the
       subtree rooted at node or -1
       if the subtree is empty */
    int result = -1;
    if (node != null) {
        int leftHeight = heightSubtree
            (node.left);
        int rightHeight =
            heightSubtree(node.right);
        result = 1 + Math.max(
            leftHeight, rightHeight);
    }
    return result;
}
Time Complexity of Measuring Height

- How long does it take to measure the height?
How long does it take to measure the height?

- Inside the recursive function for each node, we perform $O(1)$ operations.
- Each node is visited exactly once.
- In total, the time complexity will be $n \times O(1) = O(n)$. 

Theorem

The height of a tree of $n$ nodes can be measured in $O(n)$. 

How long does it take to measure the height?

- Inside the recursive function for each node, we perform $O(1)$ operations.
- Each node is visited exactly once.
- In total, the time complexity will be $n \times O(1) = O(n)$.

**Theorem**

The height of a tree of $n$ nodes can be measured in $O(n)$. 
Tree Traversal

Any tree traversal involves:

- Visiting the root.
- Recursively visiting all nodes in the left subtree.
- Recursively visiting all nodes in the right subtree.
Tree Traversal

• Any tree traversal involves:
  • Visiting the root.
  • Recursively visiting all nodes in the left subtree.
  • Recursively visiting all nodes in the right subtree.

• Three common tree traversal:
  • In-Order Traversal: make a recursive traversal of the left subtree, then visit the root, then make a recursive traversal of the right subtree.
Tree Traversal

- Any tree traversal involves:
  - Visiting the root.
  - Recursively visiting all nodes in the left subtree.
  - Recursively visiting all nodes in the right subtree.

- Three common tree traversal:
  - **In-Order Traversal**: make a recursive traversal of the left subtree, then visit the root, then make a recursive traversal of the right subtree.
  - **Pre-Order Traversal**: visit the root, then make a recursive traversal of the left subtree, then make a recursive traversal of the right subtree.
Tree Traversal

- Any **tree traversal** involves:
  - Visiting the root.
  - Recursively visiting all nodes in the left subtree.
  - Recursively visiting all nodes in the right subtree.

- Three common tree traversal:
  - **In-Order Traversal**: make a recursive traversal of the left subtree, then visit the root, then make a recursive traversal of the right subtree.
  - **Pre-Order Traversal**: visit the root, then make a recursive traversal of the left subtree, then make a recursive traversal of the right subtree.
  - **Post-Order Traversal**: make a recursive traversal of the left subtree, then make a recursive traversal of the right subtree, then visit the root.
Tree Traversal Example

- In-order traversal?
- Pre-order traversal?
- Post-order traversal?
public void in_order_traverse()
{
in_order_TraverseSubtree(treeRoot);
}

public void pre_order_traverse()
{
pre_order_TraverseSubtree(treeRoot);
}

public void post_order_traverse()
{
post_order_TraverseSubtree(treeRoot);
}

void in_order_TraverseSubtree(TreeNode node) {
if (node != null) {
in_order_TraverseSubtree(node.left);
System.out.println("visit "+node.key);
in_order_TraverseSubtree(node.right);
}
}
public void in_order_traverse() {
    in_order_TraverseSubtree(treeRoot);
}

public void pre_order_traverse() {
    pre_order_TraverseSubtree(treeRoot);
}

public void pre_order_traverse() {
    pre_order_TraverseSubtree(treeRoot);
}

void in_order_TraverseSubtree(TreeNode node) {
    if (node != null) {
        in_order_TraverseSubtree(node.left);
        System.out.println("visit " + node.key);
        in_order_TraverseSubtree(node.right);
    }
}

void pre_order_TraverseSubtree(TreeNode node) {
    if (node != null) {
        System.out.println("visit " + node.key);
        pre_order_TraverseSubtree(node.left);
        pre_order_TraverseSubtree(node.right);
    }
}

void post_order_TraverseSubtree(TreeNode node) {
    if (node != null) {
        post_order_TraverseSubtree(node.left);
        post_order_TraverseSubtree(node.right);
        System.out.println("visit " + node.key);
    }
}

void post_order_TraverseSubtree(TreeNode node) {
    if (node != null) {
        post_order_TraverseSubtree(node.left);
        post_order_TraverseSubtree(node.right);
        System.out.println("visit " + node.key);
    }
}
All in-order, pre-order, and post-order traversals take $O(n)$

Regardless of the structure of the tree (balanced or unbalanced):
- each node is visited exactly once
- visiting a node takes $O(1)$ time

**worst case = average case = best case for tree traversal.**
Searching for a Key in the Tree

- If keys in a tree are unordered, searching for a key s in a tree rooted at node a is similar to performing a pre-order traversal:
  - check whether s is the key of a,
  - recursively search for s in the left subtree of a,
  - recursively search for s in the right subtree of a.
If keys in a tree are unordered, searching for a key $s$ in a tree rooted at node $a$ is similar to performing a pre-order traversal:

- check whether $s$ is the key of $a$,
- recursively search for $s$ in the left subtree of $a$,
- recursively search for $s$ in the right subtree of $a$.

```java
public TreeNode find(int searchKey) {
    return findSubtree(searchKey, treeRoot);
}

private TreeNode findSubtree(int searchKey, TreeNode node) {
    // searchkey or null if the key is not in the subtree
    TreeNode searchNode = node;
    if (node != null && node.key != searchKey) {
        searchNode = findSubtree(searchKey, node.left);
        if (searchNode == null)
            searchNode = findSubtree(searchKey, node.right);
    }

    return searchNode;
}
```
Search Example

- what nodes are visited in `myTree.find(18);`?
Search Example

- what nodes are visited in `myTree.find(18);`?
- what nodes are visited in `myTree.find(22);`?
Search Example

- what nodes are visited in `myTree.find(18);`?
- what nodes are visited in `myTree.find(22);`?
- what nodes are visited in `myTree.find(25);`?
Search Time Complexity

- What is the best time that it takes to search?
Search Time Complexity

- What is the best time that it takes to search?
- What is the worst time that it takes to search?
What is the best time that it takes to search?

What is the worst time that it takes to search?

In the next module, we will see how to implement find in $O(\log n)$ time in binary-search trees in the worst case when the tree is balanced.
Search Time Complexity

- What is the best time that it takes to search?
- What is the worst time that it takes to search?

In the next module, we will see how to implement find in $O(\log n)$ in time **binary-search trees** in the worst case when the tree is balanced.

Many of the operations on trees involve searching. Consequently, several operations that require $O(n)$ worst-case time on an unordered binary tree, require only $O(\log n)$ time on a balanced binary search tree.