Trees

Chapter #4

pp. 121-155, 163-164, 170
Trees

• **Motivation**: O(N) time to access arrays or linked lists.

• **Goal**: O(log N) time for all operations.

• A **tree** is a collection of nodes. The collection may be empty. If it isn’t empty, then the tree consists of a **distinguished node** \( r \), called a **root** and zero or more non-empty distinct (sub)trees \( T_1, \ldots, T_k \), each of whose root are connected by a **directed edge** from \( r \).
Visualizing Trees

- root of each subtree is a child of $r$.
- $r$ is the parent of each subtree root.
Tree terminology

- A **leaf** has no children.
- **Siblings** have the same parent.
- A **path** is a sequence of nodes \( n_1, n_2, \ldots, n_k \) such that \( n_i \) is the parent of \( n_{i+1} \) for \( 1 \leq i < k \).
- The **length** of a path is the number of edges in the path.
- The **depth** of a node is the length of the path from the root to the node.
- The **height** of a tree: length of the longest path from root to a leaf.
Tree example

- C:\
  - CS216
    - lab1
      - list.h
    - lab2
      - list.cpp
    - lab3
      - calc.cpp
  - CS120
  - MyMail
    - school
    - pers
Example: HTML document

<HTML>
<HEAD>…</HEAD>
<BODY>
<H1>My Page</H1>
<P> Blah
<pre>blah blah</pre>
End
</P>
</BODY></HEAD>
</HTML>

How is this a tree?
Trees are Everywhere

• Lab 3 – calculator
• folders/files on a computer
• HTML and XML document structures
• compilers: parse tree
  \[ a = (b + c) \times d; \]
First child/next sibling

```c
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
}
```

![Diagram showing a tree structure with nodes and edges labeled with courses and files.](image-url)
Tree traversals

TreeNode::printTree(TreeNode tnode) {
    tnode.print();
    for each child c of tnode
        c.printTree();
}

• preorder/postorder traversal
int TreeNode::numNodes(TreeNode tnode) {
    if (tnode == NULL)
        return 0;
    else
        sum = 0;
        for each child c of tnode
            sum += numNodes(c);
        return 1 + sum;
}
Binary trees

- A **binary tree** is a tree where all nodes have at most two children.

```c
struct BinaryNode
{
    Object element;
    BinaryNode *left;
    BinaryNode *right;
}
```

![Binary tree diagram]
Binary Search Trees

• Associated with each node is a **key** value that can be compared.

• **Binary search tree property:**
  – every node in the left subtree has key whose value is less than the value of the root’s key value, and
  – every node in the right subtree has key whose value is greater than the value of the root’s key value.
Example

BINARY SEARCH TREE
Counterexample

NOT A BINARY SEARCH TREE
find

• **Basic idea:** compare the value to be found to the key of the root of the tree.
  – If they are **equal**, we are done.
  – If they are **not equal**, recurse depending on which half of the tree the value to be found should be in if it is there.
find

BNode<int> *
BST::find(const int x, BNode<int> *t){
    if (t == NULL)
        return NULL;
    else if (x < t->element)
        return find(x, t->left);
    else if (x > t->element)
        return find(x, t->right);
    else
        return t;  // match
}
BST Insert

• To insert an element, we essentially do a **find**. When we reach a NULL pointer, we create a new node there.

```c++
void BST::insert(const Comp & x, BinaryNode<Comp> * & t) {
    if (t == NULL)
        t = new BinaryNode<Comp>(x, NULL, NULL);
    else if (x < t->element)
        insert(x, t->left);
    else if (x > t->element)
        insert(x, t->right);
    else
        ;  // if duplicate; do appropriate thing
}
```
findMin, findMax

• To find the maximum element in the BST, we ...

• To find the minimum element in the BST, we ...
BST remove

• Removing an item disrupts the tree structure.

• Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.

• Three cases:
  – node has no children
  – node has one child
  – node has two children
No children, one child

[Diagram of a tree with nodes labeled 1, 3, 4, 5, 7, and 11]
Two children
• Replace the node with its successor. Then remove the successor from the tree.
Height of BSTs

• $n$-node BST: Worst case depth: $n-1$.

• **Claim:** The maximum number of nodes in a binary tree of height $h$ is $2^{h+1} - 1$.

**Proof:** The proof is by induction on $h$. For $h = 0$, the tree has one node, which is equal to $2^{0+1} - 1$. Suppose the claim is true for any tree of height $h$. Any tree of height $h+1$ has at most two subtrees of height $h$. By the induction hypothesis, this tree has at most $2 \cdot (2^{h+1} - 1) + 1 = 2^{h+2} - 1$. 
Height of BSTs, cont’d

• If we have a BST of $n$ nodes and height $h$, then by the Claim,
  \[ n \leq 2^{h+1} - 1. \]
  So, $h \geq \log (n+1) - 1$.

• **Average** depth of nodes in a tree.
  Assumptions: insert items randomly (with equal likelihood); each item is equally likely to be looked up.

• **Internal path length**: the sum of the depths of all nodes.
traversals

```cpp
void BST::print(BNode<int> *t) {
    if (t != NULL) {
        print(t->left);
        cout << t->element;
        print(t->right);
    }
}
```
AVL Trees

• **Motivation:** we want to guarantee $O(\log n)$ running time on the find/insert/remove operations.

• **Idea:** keep the tree balanced after each operation.

• **Solution:** AVL (Adelson-Velskii and Landis) trees.

• **AVL tree property:** for every node in the tree, the height of the left and right subtrees differs by at most 1.
AVL tree

not an AVL tree
AVL trees: find, insert

• AVL tree **find** is the same as BST find.
• AVL **insert**: same as BST insert, except that we might have to “fix” the AVL tree after an insert.
• These operations will take time $O(d)$, where $d$ is the depth of the node being found/inserted.
• What is the maximum height of an $n$-node AVL tree?
AVL tree insert

• Let $x$ be the deepest node where an imbalance occurs.
• Four cases to consider. The insertion is in the
  1. left subtree of the left child of $x$.
  2. right subtree of the left child of $x$.
  3. left subtree of the right child of $x$.
  4. right subtree of the right child of $x$.

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.
Single rotation example
Single rotation in general

\[ X < b < Y < a < Z \]

\[ h \geq -1 \]
Cases 2 & 3 — try a single rotation…

\[ X < b < Y < a < Z \]

single rotation fails!
Double rotation, step 1
Double rotation, step 2
Double rotation in general

\[ W < b < X < c < Y < a < Z \]

\[ h \geq 0 \]
AVL tree: Running times

- **find** takes $O(\log n)$ time, because height of the tree is always $O(\log n)$.
- **insert**: $O(\log n)$ time because we do a find ($O(\log n)$ time), and then we may have to visit every node on the path back to the root, performing up to 2 single rotations ($O(1)$ time each) to fix the tree.

- **remove**: $O(\log n)$ time. Left as an exercise.