Chapter 5

Binary Trees
Definitions and Properties

- A binary tree is made up of a finite set of elements called **nodes**
- It consists of a **root** and two **subtrees**
- There is an **edge** from the root to its children
- If $n_1, n_2, \ldots n_k$ is a sequence of nodes such that $n_i$ is the parent of $n_{i+1}$, then that sequence is a **path**
- The **length** of the path is $k-1$
- The **depth** of node $M$ is the length of the path from the root to $M$
More definitions

- The **height** of the tree is one more than the depth of the deepest node.
- A **leaf** node is any node with no children.
- An **internal** node is a node with at least one child.
- A **full** binary tree has each node as either an internal node with two children or as a leaf.
- A **complete** binary tree has each level filled and the last level may not be filled, but it is filled from left to right.
Binary Tree Theorems

- The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.
- The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.
- These are useful when you need to compute the amount of overhead used by a tree.
Binary Tree ADT

- There are activities that relate to nodes
  - Reach a child
  - Get a value
- There are activities that relate to trees
  - Initialization
  - Clearing
- Due to these activities the Node class should be separate from the Tree class
Binary Tree Traversals

- Any process for visiting all the nodes in some order is a traversal
- Any traversal that visits each node only one time is an enumeration
- There are three main kinds of enumerations on trees
  - Preorder
  - Postorder
  - Inorder
Traversing a tree involves selecting a function to determine the order in which the elements are processed. The traversal function is easily written as a recursive function. An important design decision is when to check for empty subtrees. You can check at the beginning of the function, returning if you are a null pointer. You can also check before you call the recursive function. The first choice is better.
Why?

- We would have to use two checks for a null tree versus one.
- We make twice as many function calls.
- We are required that an empty tree is never passed in or make an additional check for an empty tree.
- In any case, it gets complicated and error prone.
Binary Tree Node Implementations

- Normally nodes contain a data field value and two pointers to children.
- Some programmers find it convenient to add a pointer to the parent.
- This is somewhat analogous to a pointer to a previous node in a Linked List.
- However, it is almost always unnecessary and adds unneeded overhead.
An important decision is whether the same class can be used for internal and leaf nodes.

Some applications only store data in the leaf nodes.

Other applications require one type of data in an internal node and another in the leaves.

There are some choices in what to do.
enum Nodetype {leaf, internal};
class VarBinNode { // Generic node class
public:
    Nodetype mytype; // Store type for node
    union {
        struct { // internal node
            VarBinNode* left;  // Left child
            VarBinNode* right; // Right child
            Operator opx;      // Value
        } intl;
        Operand var;       // Leaf: Value only
    };
};
In the union construct, you get one or the other, but not both

So we either get the pointers to the children and the operator (internal node)

Or we get the value (leaf node)

Problem is the data structure store enough space for the biggest subtype
Better Approach

- Use inheritance
- Have a base class that implements a general node
- Inherit from the base class one for each type of node
- The method isLeaf() returns true for leaves and false otherwise
- In this option the tree handles traversing the nodes.
- This method uses polymorphism to handle which member functions get called
Third Approach

- This approach also uses inheritance.
- The major difference is that the nodes know how to traverse themselves.
- Meaning that each subclass implements a traverse function that is polymorphically called.
- This is known as a Composite design pattern
Pros and Cons

- When comparing the two inheritance methods
- The first does not require that the node classes explicitly support a traverse function
- This fact makes it easy to add new node types
- The second approach relieves the tree class from needing to know anything about the internals of the node classes.
- Secondly, the traverse function does not need to know all the node types
Space Requirements

- Assuming every node has two pointers and a data spot

- For n nodes we get 
  - n(2p + d)

- The total overhead required is 2pn for the entire tree

- The fraction of overhead is 2p/(2p+d)

- If p = d, then about 2/3 of the tree is overhead
Space for Full Trees

- If only the leaves store data, then we can get great savings in space.

\[
\frac{n/2(2p)}{n/2(2p)+d} = \frac{p}{p+d}
\]

- When \( p = d \) we get about \( \frac{1}{2} \) of the space is overhead

- This is much better.
Array Implementation for Complete Binary Trees

- We begin by assigning numbers to the node position in the complete binary tree, level by level.
- An array can store the tree’s data values efficiently, placing each value in the array position corresponding to the nodes position.
- Simple formulas can be determined to calculate siblings, parents, and children.
<table>
<thead>
<tr>
<th>Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Left Child</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Right Child</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Left Sibling</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>3</td>
<td>--</td>
<td>5</td>
<td>--</td>
<td>7</td>
<td>--</td>
<td>9</td>
<td>--</td>
</tr>
<tr>
<td>Right Sibling</td>
<td>--</td>
<td>2</td>
<td>--</td>
<td>4</td>
<td>--</td>
<td>6</td>
<td>--</td>
<td>8</td>
<td>--</td>
<td>10</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Formulas

- Parent(r) = \lfloor (r-1)/2 \rfloor \text{ if } r \neq 0
- Left Child(r) = 2r+1 \text{ if } 2r+1 < n
- Right Child(r) = 2r+2 \text{ if } 2r+2 < n
- Left sibling(r) = r-1 \text{ if } r \text{ is even}
- Right sibling(r) = r+1 \text{ is odd and } r+1 < n
Binary Search Tree

- Binary Search Tree (BST) Property: All nodes stored in the left subtree of a node whose key is K have key values less than K. All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K.
Method Example

```cpp
bool find(const Key& k) const
{
    return findHelp(root, K);
}

bool findHelp(BinNode* subRoot, const Key& k) const
{
    if(subRoot == NULL) return false;
    else if( k < subRoot-->val() )
        return findHelp(subRoot-->left(), k);
    else if( k > subRoot-->val() )
        return findHelp(subRoot-->right(), k);
    else
        return true;
}
```
Insertion

- Insertion requires that the location for the record is first located.
- The recursion ends when the subroot is null.
- Either the node is an internal node with an open child on the correct side or the node is a leaf.
- A new node is created and the node is inserted in the tree.
- Question: how does the parent know who their new child is?
Insertion Ideas

- Insert uses an insertHelper function to actually perform the insertion.
- insertHelper returns a pointer to itself
- This allows the parent to set the correct pointer to point to the new child
Removing Nodes

- This is a bit trickier than insertion.
- Let's look at the smaller problem of removing the smallest item from a given subtree.
- We will traverse down the left child pointers until we find a node that does not have a left child.
- We will call that node min and return a reference to it.
- We will also return a pointer to its right subchild.
More Deletions

- So to delete a general key value R, we first find it and check its children.
- If it is a leaf, we blow it away.
- If it has one subchild, we have the node's parent point to it and we are done.
- If it has two children, well we have to do more work.
- What can we do?
More Deletions

- We can find a value to substitute for this value that will keep the BST property

- Two choices,
  - The greatest value from the left subtree
  - The least value from the right subtree

- Well, we find the smallest value in the nodes right subtree.

- Why?
Threaded Trees

- All previously discussed tree traversals required a stack
- Either the stack is created and maintained explicitly by the user or created by the system as the run-time stack.
- So there is another way to traverse a tree without a stack?
  - Of course there is...otherwise why would we be talking about it.
Stackless Depth-First Traversal

- Threaded trees allow you to traverse the tree by following pointers stored within the tree.
- Each node would store pointers to its predecessor and successor.
- This would create a lot of overhead with the additional two pointers for a total of 4 pointers per node.
- So what can we do?
It’s all in the context

- You’ve already got two pointers in the nodes, why not use them?
- Of course you need to know what pointers are currently referring to, their context
- You can use a bit to indicate what the pointer is being used for.
- How does this work?
Inorder example

```cpp
void ThreadedTree<T>::inorder()
{
    ThreadedNode<T> *prev, *p = root;
    if( p != 0 ) {
        while ( p-->left != 0 )
            p = p-->left;
        while( p != 0 ) {
            visit( p );
            prev = p;
            p = p-->right;
            if ( p != 0 && prev-->successor == 0)
                while( p-->left != 0 )
                    p = p-->left;
        }
    }
}
```
Other Methods

- You can also reorganize the tree to traverse the tree without stacks and without threads
- It is easy to traverse a tree with no left child
- One such algorithm is:
  - MorrisInorder()
    - While not finished
      - If node has no left descendant
        - Visit it;
        - Go to the right
      - Else make this node the right child of the rightmost node in its left descendants, go to this left descendant
Heaps and Priority Queues

- There are many cases where we want to choose the next most important item from a list.
- This occurs in hospitals, OSs, etc.
- Normal queues are not efficient enough to implement a priority queue.
- We could use a binary tree to help with this problem.
Heaps

- A heap is a complete binary tree, and so it is almost always implemented in an array.
- The values stored in a heap are partially ordered.
- This means that there is a relationship between the value stored at any node and the values of its children.
- There are two variations.
Min Heaps and Max Heaps

- The **min heap** stores at every node a value that is less than or equal to that of its children.
- The root stores the smallest value
- The **max heap** stores at every node a value that is greater than or equal to that of its children.
- The root stores the greatest value
Building a Heap

(a) (4-2) (4-1) (2-1) (5-2) (5-4) (6-3) (6-5) (7-5) (7-6)
(b) (5-2), (7-3), (7-1), (6-1)
Details

- Assume you have a root and its children are already heaps.
- You “siftdown” the root to the correct location.
  - If the root is greater than its children you are done
  - If not exchange the root with its bigger child and repeat at the next level
- Where do you start this process?
  - At the last internal node.
Example
Removing

- Removing requires that we replace the root with a value so that we maintain the shape of the tree.
- Take the last node and swap it with the root.
- Call siftdown on the root.
Cost to Build a Heap

- Cost for heap construction:

\[ \sum_{i=1}^{\log n} (i - 1) \frac{n}{2^i} \approx n. \]