Binary Trees and Binary Search Trees—C++ Implementations

Learning Goals:

- Apply basic tree definitions to classification problems.
- Describe the properties of binary trees, binary search trees, and more general trees; and implement iterative and recursive algorithms for navigating them in C++.
- Insert and delete nodes in a binary tree.
- Compare and contrast ordered versus unordered trees in terms of complexity and scope of application.
- Describe and use preorder, inorder and postorder tree traversal algorithms.
- Provide examples of the types of problems for which tree data structures are well-suited.

In a sorted array, we can search for an element using binary search—an O(lg n) operation. In this section, we generalize this concept to binary trees.

**Definition:** A binary tree is a data structure that 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.

Note that this definition is recursive – we define a binary tree as a structure that consists of two other (sub)trees.
**Domain:** A set of nodes containing: (a) some application data (e.g., one or more of: name, student #, GPA, …), and (b) two (possibly NULL) pointers to other nodes.

**Structure:** There is a unique root node (that has no parent node) having zero, one, or two child nodes; every other node has exactly one parent node and either zero, one, or two children.

**Operations:**
- `insertLeft` - insert new node as left child of given node
- `insertRight` - insert new node as right child of given node
- `find` - find node containing given item (i.e., a “search key”)
- `findParent` - find parent of given node
- `deleteItem` - remove node containing given item
- `print` - print data in the tree

etc.

---

**Some Terminology:**

The *path* from node $N_i$ to node $N_k$ is a sequence of nodes $N_1, N_2, \ldots, N_k$ where $N_i$ is the parent of $N_{i+1}$. 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 level of the root is 0.

The *height* of a node $N$ is the length of the longest path from $N$ to a leaf node (a node with no child). The height of a leaf node is 0.

The *height of a tree* is the height of its root node. The height of the empty tree is $-1$. The root appears at level 0.

The number of nodes in a tree of height $h$ is at least $h+1$ and no more than $2^{h+1}-1$. 

---
Height of tree:
Depth of node containing B:
Height of node containing B:
# of nodes in this tree:
# of leaves (i.e., external nodes):
# of non-leaf (i.e., internal) nodes:

---

**Full Binary Tree:** Each node has exactly 0 or 2 children. Each internal node has exactly ___ children; each leaf has ___ children. Examples:

**Complete Binary Tree:** A full binary tree where all leaves have the same depth. Examples:

**Nearly Complete Binary Tree:** All leaves on the last level are together on the far left side, and all other levels are completely filled. Examples:
At this stage in the course, we assume that binary trees are arbitrarily ordered (i.e., there is no order to their keys).

The number of distinct binary trees with \( n \) nodes is given by the Catalan number having the formula:

\[
f(n) = \frac{(2n)!}{(n+1)!n!}
\]

How many distinct binary trees are there when \( n=3 \)?

Draw them.

**Implementation of a Binary Tree in C++**

We will assume that a `typedef` statement has defined `Item_type` to be the type of data to be stored in the tree. Each node contains an item, a pointer to the left subtree and a pointer to the right subtree:

```cpp
typedef string Item_type;
struct BNode
{
    Item_type item; // may have many data fields
    BNode* left;
    BNode* right;
};
```

We will now look at the implementation of some of the operations listed earlier.

To begin, we will write a `makeNode` function to create a new `BNode` as needed. Note the default assignments in the argument list:
Bnode * makeNode(const Item_type& item,
    Bnode * leftChild = NULL, Bnode * rightChild = NULL)
// PRE:  item is valid, leftChild points to a node or is
//       NULL, rightChild points to a node or is NULL
// POST: a new node is created and its address is
//       returned
{
    Bnode * temp;
    temp = new Bnode;
    temp->item = item;
    temp->left = leftChild;
    temp->right = rightChild;
    return temp;
}

Bnode * makeNode(const Item_type& item,
    Bnode * leftChild = NULL, Bnode * rightChild = NULL)

Notes:

A sample calling statement is:

Recall, when leaving a function in C++, we:

(a) Pass back the appropriate return value to the caller. In the calling
statement, the return value actually replaces the whole function call,
but only the function call (since there may be more things to do).

(b) Destroy all temporary variables in the function, but retain the
dynamically allocated ones (and any with keyword “static”).

(c) Return to the caller, and continue execution with the rest of the
calling statement, if anything in the calling statement remains.
Consider the `insertLeft` function. This function is used to insert an item in a node to the left of a current node (or to create a new binary tree). We assume that `current` does not already have a left child; so we are inserting into an “empty” spot in the tree:

```cpp
void insertLeft( BNode*& current,
    const Item_type& item )
// PRE:  current points to a node in a binary tree or
//       is NULL
// POST: if current is not null, the left child of
//       current is a new node containing item; else,
//       current points to a root node containing item
//
//    if (current) // same as: “if (current != NULL)”
//      current->left = makeNode(item);
//    else
//      current = makeNode(item);
```

Now let’s implement/complete the `find` function. This function will be used to assist in the process of deleting an item from the tree. In order to delete an item, we need to find the node that contains it.

```cpp
BNode* find( BNode* root, const Item_type& item )
// PRE: root points to the root of a binary (sub)tree
// POST: if item is in the tree, the address of the
// node containing item is returned; otherwise,
//         NULL is returned
{
  Bnode * temp;
  if (root == NULL || root->item == item)
    return root;
  temp =
```
We will now consider the `deleteNode` function. The task of removing a node from a binary tree is quite complicated; therefore, we will break the task into parts (a full version is on WebCT).

```cpp
bool deleteNode( BNode*& root, const Item_type& item )
// PRE: root points to the root of a binary tree or is NULL
// POST: if item is in tree, first instance of node containing item has been deleted and true is returned; else, false is returned (only)
{
    Bnode *temp, *parent;
    temp = find(root, item);
    if (!temp) // same as: if (temp == NULL)
        return false;
    // temp now points to the node containing item
    parent = findParent(root, temp);
    
    Now that we have a reference to the pointer that points to the node to be deleted, we proceed according to one of 4 cases:

    Case 1: node to be deleted is a leaf

    if (isLeaf(temp))
    {
        if (parent) // "if (parent != NULL)"
            if (parent->left == temp)
                parent->left = NULL;
            else
                parent->right = NULL;
        else
            root = NULL;

        delete temp; // avoid memory leak
        // see page 120 in Koffman text
        return true;
    }
```
Case 2: node to be deleted has both a left and right child

This is the tricky case. There is no obvious way to remove a node having two children and re-connect the tree. Instead, we will choose not to delete the node but rather copy data from a leaf node (which is easy to remove) into the current node. We will arbitrarily choose to copy data from the leftmost leaf of the node to be deleted (if the nodes’ keys are in arbitrary order).

See the example in the WebCT/Vista course notes, and in your lab exercise.

Case 3: node to be deleted has only a left child

if ( hasLeftChild(temp) )
{
    if (parent)
        if (parent->left == temp)
            parent->left = temp->left;
        else
            parent->right = temp->left;
    else
        root = temp->left;
    delete temp;
    return true;
}

Case 4: node to be deleted has only a right child (see WebCT Vista)
(same idea)
**Tree Traversals**

There are three common types of binary tree traversals:

*Preorder (prefix)*: “process” the current node, then recursively visit its left subtree, then recursively visit its right subtree

![Preorder Traversal Diagram]

Data printed using *preorder* traversal:

*Inorder (infix)*: visit the left subtree, then *process* the current node, then visit the right subtree

![Inorder Traversal Diagram]

Data printed using *inorder* traversal:
**Postorder (postfix):** visit the left subtree, then visit the right subtree, then *process* the current node

Data printed using *postorder* traversal:

Let’s consider how we would implement a function that traverses a binary tree using inorder traversal (recursively) and applies a function to each node visited:

```c
void inorder( BNode* root, void (*process)(BNode*) )
// PRE: root points to a binary tree or is NULL
// POST: function *process() has been applied to each node in the tree, using inorder traversal
```

The implementation of the functions to perform preorder or postorder traversal is left as an exercise.
An Application: Binary Expression Trees

Arithmetic expressions can be represented using binary trees. We will build a binary tree representing the expression:

\[(3 + 2) \times 5 - 1\]

We start by identifying the operator with the highest precedence and build a binary tree having the operator at the root, the left operand as the left subtree and the right operand as the right subtree. We continue in this fashion until all operators have been represented:

\[(3 + 2)\]

\[(3 + 2) \times 5\]

\[(3 + 2) \times 5 - 1\]

Now let’s print this expression tree using postorder traversal:

What we now have is the arithmetic expression written using Reverse Polish Notation (RPN). It turns out to be much easier to write an algorithm to evaluate an expression written in RPN than using the common arithmetic notation found at the top of this page.
Binary Search Trees (Review, and C++ Implementation):

A binary search tree (BST) is a binary tree such that for every node \( v \) in the tree: (a) all of the keys (elements) in \( v \)'s left subtree are \( \leq \) \( v \)'s key, and (b) all of the keys in \( v \)'s right subtree are \( \geq \) \( v \)'s key.

\[
\begin{array}{c}
3 \\
2 & 5 \\
\end{array}
\quad
\begin{array}{c}
5 \\
2 & 3 & 9 \\
\end{array}
\quad
\begin{array}{c}
6 \\
3 & 7 \\
\end{array}
\quad
\begin{array}{c}
6 \\
5 & 9 \\
\end{array}
\]

a binary search tree
not a binary search tree

The order of the keys in the nodes is important!

Common Operations on BST’s

Searching for a Key (called a Search Key) in a BST

Algorithm:
If tree is empty then search key is not present, and we’re done
If search key = root’s key, we’ve found it, and we’re done
If search key < root’s key then
  Search left subtree
else
  Search right subtree

\[
\begin{array}{c}
3 \\
2 & 5 \\
\end{array}
\quad
\begin{array}{c}
6 \\
3 & 7 \\
\end{array}
\quad
\begin{array}{c}
6 \\
5 & 9 \\
\end{array}
\]

CPSC 221 Binary Trees Page 23
CPSC 221 Binary Trees Page 24
In the following implementations of the Search function, we assume that the nodes of the binary search tree are represented as follows:

```cpp
template<typename Key, typename Value>
struct BNode
{
    Key key;
    Value value;
    Bnode * left;
    Bnode * right;
};
```

Example of calling sequence from `main()` or another function:

```cpp
Bnode<int, string>* myFirstTree;
Bnode<char, float>* mySecondTree;
string myString;
float num;
... // initialization and other code
search(myFirstTree, 5, myString); // see next page
search(mySecondTree, 'w', num);
```

**Recursive Implementation of Search Function:** If key is found, return true and the “value” associated with the key; else return false.

```cpp
template<typename Key, typename Value>
bool search(BNode<Key, Value>* root,
            const Key& key, Value& value)
{
    if (root == NULL)
        return false;
    if (root->key == key)
    {
        value = root->value;
        return true;
    }
    if (key < root->key) // search left subtree
        return search(root->left, key, value);
    else // search right subtree
        return search(root->right, key, value);
}
```
Alternatively, if we want to return a pointer to the node containing the found search key (or NULL if no such node exists), we can use the following code (and this time, we’ll use an iterative version):

```cpp
template<typename Key, typename Value>
BNode<Key, Value>* search( BNode<Key, Value>* root, const Key& key )
{
    while (root && root->key != key)
        if (key > root->key)
            root = root->right;  // right subtree
        else
            root = root->left;   // left subtree
    return root;
}
```

**Inserting an Item into a BST:**

**Algorithm:**
- If tree is empty, then insert item as root
- If key = root’s key, then replace root’s value with new value
- If key < root’s key, then insert into root’s left subtree
  - else insert into root’s right subtree

Here is a possible prototype for this function:

```cpp
template<typename Key, typename Value>
void insert( BNode<Key, Value>*& root, const Key& key, const Value& value );
```
**Drawing exercise:** Insert the following keys (we won’t worry about the corresponding values) into an empty binary search tree in the order given. Note that in both cases, the data is the same but the order in which we do the insertion is different.

Case 1: 7, 4, 3, 6, 9, 8

Case 2: 3, 4, 6, 7, 8, 9

Note that in Case 1, we end up with a “bushy tree”; in fact, it is a nearly complete binary tree. If we search for an item in a complete (or nearly complete) binary tree, it takes $O(______)$ worst-case time.

In Case 2, our tree is rather “unbalanced” or “one-sided”. Searching for an item in this tree would take $O(______)$ worst-case time.

Our insertion algorithm makes no attempt to balance the tree—it maintains only an ordering property, not a shape property. In this sense, a binary search tree is very different from a binary heap. (We’ll study heaps, later.)

It can be shown that, on average, with random insertion, the depth of a binary search tree is $O(______)$. 
Removing an Item from a BST

Algorithm:
- Search for the item to find the node containing the item.
- If the item was not found, we’re done.
- If the node is a leaf, delete the node.
- If node’s left subtree is empty, replace node with right child
  else if node’s right subtree is empty, replace node with left child
  else replace node with its logical predecessor (rightmost node of
  its left sub-tree)

Delete 5 from tree:

Delete 7 from tree:

Delete 6 from tree:

Note that by replacing the node with its logical predecessor (or logical
successor), we maintain the ordering property of the binary search tree.
Finding the Smallest Key in a BST, Recursively

Assuming the tree is reasonably balanced, we can access the smallest (or largest) item in O(_____) time because the smallest and largest items in a BST are stored in the extreme left and extreme right nodes of the tree.

template<typename Key, typename Value>
void findSmallest( BNode< Key, Value >* root,
    Key& key, Value& value )
//Pre: BST root is not NULL
//Post: return key AND value of smallest key
{

}

A BST implementation of a map is also beneficial if we want the data sorted by key value. Suppose we want to print the data to the screen, sorted by key. This would be a(n) ______________ traversal.

template<class Key, class Value>
void printData( BNode< Key, Value >* root )
{

}

This is a Θ(____) operation (worst case, average case, and best case).