Chapter 44 - Binary Search Trees

1. Binary Trees

Binary search is an efficient search because it eliminates half of the possible places left to search with each search cycle (or visitation). If the possible paths of a binary search in an array of 15 items were drawn, it could look something like the following:

Binary trees are data structures made of nodes that are linked together via the paths of a binary search. Dynamic binary trees are made of nodes that have two pointers each.

The top node in a binary tree is referred to as the root. All nodes have two pointers. These are referred to as the left child pointer and the right child pointer. These pointers point to nodes that are referred to as the left child and the right child. A node that contains a non-null pointer to at least one child is referred to as a parent node. A node that does not point to at least one child node (both pointers are null) is called a leaf node.

2. Levels of a Binary Tree

The root of a binary tree is said to be at level 0. The child nodes of the root are said to be on level 1. Their children are referred to as being on level 2.

The number of nodes possible to store in a tree almost doubles with each new level added. By using the level numbers as powers of two, the number of nodes that a binary tree can contain can be calculated by adding up these powers of two. For example, the most nodes a binary tree of levels 0 through 5 can have is 63. This was calculated by:

\[2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 63\]
This can be simplified to
\[ 2^6 - 1 = 63 \]

In other words, the maximum number of nodes that can be stored in a binary tree is
\[ 2^{n+1} - 1 \]

where \( n \) is the largest level number in the tree.

3. Searching and Inserting into a Binary Search Tree
   To insert a new node into a binary search tree, first the tree is searched via binary search on a value known as the key. Each node has a key value that should make it unique within the tree. By convention, if the key of the new node is less than the key of the root, the new node will be inserted on the left sub-tree. If the key of the new node is greater than the key of the root, the new node will be inserted on the right sub-tree. The search proceeds, comparing the key of the new node against the key of the root of the next sub-tree. If the search stops at a leaf without a duplicate being found, the new node is made into a child of the former leaf.

   If duplicate keys are possible, duplicate nodes become possible. Duplications can be handled by a variety of simple to complex strategies. The most simple is to record the number of duplications that are represented by the node in a field within the node. If this is not possible or desirable, a more complex solution involves storing the duplicate in an additional data structure contained within the node.

   Because of the complexity of dealing with duplicates within a binary tree, it is best to devise a key system that does not allow duplicates. For example, a binary search tree of university students will likely be keyed on student ID numbers.

   Binary search trees are named after their most prominent feature. Because the number of potential nodes in a binary tree almost double with each level added, any node can be reached in \( \log_2 n \) or less search iterations, where \( n \) is the maximum number of nodes possible in the tree. If the tree is searched heavily, this gives an upper bound of \( O(n \log_2 n) \) for binary search in the binary search tree.

   **Searching a Binary Tree**
   ```c
   nodeType *current = root;
   while (current != NULL) {
       if (current->key = = item) break;
       else if (current->key < item) current = current->rightchild;
       else current = current->leftchild;
   }
   ```

   When the search algorithm competes, if \( current \) has the value of NULL, \( item \) was not found, but if \( current \) is not NULL, the node containing \( item \) was found and the value of \( current \) is the address of that node.
Inserting into a binary tree requires that the search continue until there is a place to put the new value. This place is the first null child pointer that the search algorithm finds. In order to perform the insertion, a previous pointer to the node containing the null child pointer must be included in the search algorithm. Once the correct pointer is located, the new node can be inserted into the tree.

If root is NULL, the new node becomes the root of the tree. If current is not NULL, there is a duplicate node already in the tree. If current is NULL and previous is not NULL, the node can be inserted as a child.

**Insertion into a Binary Tree (no duplicates allowed)**

```cpp
if (root == NULL) { // if root is null, insert as root
    root = new nodeType ();
    root->dataField1 = data;
    root->dataFiled2 = data;
    ...
    root->dataFieldn = data;
    root->leftchild = NULL;
    root->rightchild = NULL;
}
else { // find location for insertion as child
    nodeType *current = root
    nodeType *previous = NULL;
    while (current != NULL) {
        if (current->key == item) break;
        previous = current;
        if (current->key < item) current = current->rightchild;
        else current = current->leftchild;
    }
    if (current == NULL && previous != NULL &&
        previous->key != item) {
        nodeType *t = new nodeType ();
        t->dataField1 = data;
        t->dataFiled2 = data;
        ...
        t->dataFieldn = data;
        t->leftchild = NULL;
        t->rightchild = NULL;
        if (item > previous->key) previous->rightchild = t;
        else previous->leftchild = t;
    }
}
```
4. **Traversing a Binary Tree**

As they are not stored in a list, binary trees are the first data structure that has been examined in this text that cannot be traversed using a simple iterative algorithm. However, they can be traversed in three different orders based on variations of a simple recursive algorithm. This algorithm performs recursion down the left sub-tree (if possible) and when that is not possible, to perform recursion down the right sub-tree (if possible). The three variations come from performing the visit (the term used for any manipulation of the data of a node) before the left and right recursive calls, between the left and right recursive calls or after the left and right recursive calls. These variations on traversal of binary tree are called pre-order, in-order, and post-order and they produce very different traversals.

**Traversal of a Binary Tree**

```c
void traverse(nodeType current) {
    if (current != NULL) {
        // ****** visit node here for pre-order traversal
        traverse(current->leftChild);
        // ****** visit node here for in-order traversal
        traverse(current->rightChild);
        // ****** visit node here for post-order traversal
    }
}
```

Given this tree:

```
Given this tree:
```

A pre-order traversal visits the nodes in this order: 7, 3, 1, 5, 11, 9, 10
An in-order traversal visits the nodes in this order: 1, 3, 5, 7, 9, 10, 11
A post-order traversal visits the nodes in this order: 1, 5, 3, 10, 9, 11, 7
5. An Example Binary Tree Class

Defining the Struct
This example of a binary tree class will have a data structure that is based on a version of the struct `daytemps` (from chapter 41) that has been modified to have two links, one called `leftchild` and one called `rightchild`.

```cpp
#include <iostream>
using namespace std;

daytemps {
    int day, hi, low;
    daytemps *leftchild, *rightchild;
};
```

Defining the Class
The example class will be a modified version of the class `temperatures` (also from chapter 41). Public member functions for the class `temperatures` will include
- a constructor to initialize the binary tree data structure
- a destructor to return all nodes in the data structure to the operating systems available memory store
- `insertDay` to insert new nodes of information into the binary tree data structure
- `returnHi` to return the high temperature of a given day
- `returnLow` to return the low temperature of a given day
- several functions (`listAllPreorder`, `listAllInorder`, `listAllPostorder`) to traverse and list the values in nodes of the binary tree.

```cpp
class temperatures {
public:
    temperatures( );
    ~temperatures( );
    void insertDay(int day, int high, int low);
    int returnHi(int day);
    int returnLow(int day);
    void listAllPreorder( );
    void listAllInorder( );
    void listAllPostorder( );

private:
    daytemps * root;
};
```

The only private data member that is needed is a pointer to struct `daytemps` called root.

```cpp
private:
    daytemps * root;
};
```
The private area of class \textit{temperatures} will be more complex that just one data member. A number of private member functions will be needed in order to preserve the encapsulation of data within the objects of class \textit{temperatures} and to simplify some of the member functions.

In order to simplify the operation of the public member function \textit{insertDay}, the separate private member function \textit{findPlace} will be created. \textit{findPlace} will return the address of the correct node to connect a new child node. In addition, \textit{findPlace} can be reused in the event class \textit{temperatures} is expanded.

The private member function \textit{find} will simplify the operation of the public member functions \textit{returnHi} and \textit{returnLow}. \textit{find} will return the address of the node that contains the temperature values of a given day. In addition to being used twice here, \textit{find} can also be reused in the event class \textit{temperatures} is expanded.

Although \textit{find} and \textit{findPlace} are very similar, the way in which they perform their functions is very different. It would be difficult to combine them into one function. Both \textit{find} and \textit{findPlace} return the address of a node inside of the binary tree data structure. They are created as private member functions so that only other member functions may call them. This is done in order to keep the data structure encapsulated inside of an object of class temperatures and prevent accidental or malicious changes to the data structure.

private:
\begin{verbatim}
    daytemps *findPlace(int day);
    daytemps *find(int day);
    daytemps *root;
\end{verbatim}

The functions that actually perform the various traversals of a binary tree will also be created as private member functions. The functions \textit{preorder}, \textit{inorder} and \textit{postorder} will be called from the public member functions \textit{listAllPreorder}, \textit{listAllInorder} and \textit{listAllPostorder}. This is done because \textit{preorder}, \textit{inorder} and \textit{postorder} will be written recursively. In order for them to function, the value of the root must first be passed to them. Since allowing the value of \textit{root} to be known outside of an object of class \textit{temperatures} would violate the encapsulation of the binary tree data structure, the public member functions \textit{listAllPreorder}, \textit{listAllInorder} and \textit{listAllPostorder} are used to access the value of root and pass it to the private member functions \textit{preorder}, \textit{inorder} and \textit{postorder}.

private:
\begin{verbatim}
    void preorder(daytemps *current);
    void inorder(daytemps *current);
    void postorder(daytemps *current);
    daytemps *findPlace(int day);
    daytemps *find(int day);
    daytemps *root;
\end{verbatim}
The destructor will call the private member function dispose for the same reasons that three listAll functions called private functions.

```cpp
private:
    void preorder(daytemps *current);
    void inorder(daytemps *current);
    void postorder(daytemps *current);
    void dispose(daytemps *current);
    daytemps *findPlace(int day);
    daytemps *find(int day);
    daytemps *root;
};
```

Here is the complete listing of the definition of class temperatures:

```cpp
class temperatures {
public:
    temperatures( );
    ~temperatures( );
    void insertDay(int day, int high, int low);
    int returnHi(int day);
    int returnLow(int day);
    void listAllPreorder( );
    void listAllInorder( );
    void listAllPostorder( );
private:
    void preorder(daytemps *current);
    void inorder(daytemps *current);
    void postorder(daytemps *current);
    void dispose(daytemps *current);
    daytemps *findPlace(int day);
    daytemps *find(int day);
    daytemps *root;
};
```

**Writing the Member Functions**

The constructor of class temperatures only has to set the value of the pointer root to NULL, as there is no data to insert when an object is created.

```cpp
temperatures::temperatures( ) {
    root = NULL;
}
```
The function `insertDay` is passed the values of day, high temperature and low temperature. If the pointer `root` is equal to NULL, the tree is empty. A new node is created for `root` to point to and all of the data is inserted into it. Pointers `leftchild` and `rightchild` are set to NULL so that other members functions searching or traversing the tree will operate correctly.

If root is not NULL, the tree is searched with a call to function `findPlace`, which returns the address of the node to receive the new child node. This is stored in the local variable `current`. If the value of `current` is not NULL and value `day` in the data to be inserted is not the same as the value in `current->day`, a new node will be created, values set and `current->leftchild` or `current->rightchild` will be set to point to the new node. Since the new node is always a leaf, the values of the pointers `leftchild` and `rightchild` in the new node are set to NULL so that other members functions searching or traversing the tree will operate correctly. However, if the value of `day` and `current->day` are the same, no action will be taken as no duplicate nodes are allowed.

```c
void temperatures::insertDay(int day, int high, int low) {
    if (root == NULL) {
        root = new daytemps();
        root->day = day;
        root->hi = high;
        root->low = low;
        root->leftchild = NULL;
        root->rightchild = NULL;
    } else {
        daytemps *current = findPlace(day);
        if (current && current->day != day) {
            daytemps *t = new daytemps();
            t->day = day;
            t->hi = high;
            t->low = low;
            t->leftchild = NULL;
            t->rightchild = NULL;
            if (day > current->day) current->rightchild = t;
            else current->leftchild = t;
        }
    }
}
```

The function `findPlace` receives the day to insert and returns the address of the node in which to link a new child node. To do this, it must keep track of the address of the current node and the address of the node previous to the current node while performing a binary search for a NULL pointer. Pointer `previous` begins with the value of NULL while pointer `current` begins with the address contained in `root`.

As long as pointer `current` is not equal to NULL, the loop will repeat. In the body of the loop, several actions must be taken. First, if the value of `current->day` equals the
value in the parameter variable *day*, the function must be terminated and the value in *current* returned. This will allow the *insertDay* function to prevent the insertion of nodes with the same day.

If the loop continues, pointer *previous* will receive a copy of the value in pointer *current*. This is because the value of pointer *current* is going to change immediately and the value in *current* would be discarded before we are sure that it is not useful. The pointer *current* is the set to the value of *current-*>rightchild if *current-*>day is less than *day* or *current* is then set to the value of *current-*>leftchild if *current-*>day is greater than *day*.

If the loop continues until the pointer *current* is equal to NULL, the value in the pointer *previous* will point to the node that will receive the new child node. The value in *previous* is then returned.

```c++
define daytemps * daytemps::findPlace(int day) {
define daytemps *previous = NULL;
define daytemps *current = root;
while (current != NULL) {
    if (current->day = = day) return current;
    previous = current;
    if (current->day < day) current = current->rightchild;
    else current = current->leftchild;
}
return previous;
}
```

The member functions *returnHi* and *returnLow* first call the function find in order to locate the correct node. Private member function *find* returns the address of the node with the value in the parameter *day* or NULL if there is no matching node. Both function *returnHi* and *returnLow* return -500 (an impossible value) if the value returned by function *find* is NULL. If the value returned by function *find* is not NULL, function *returnHi* will return the high temperature for the give day. Likewise, function *returnLow* will return the low temperature for the given day if the value returned by function *find* is not NULL.

```c++
define int daytemps::returnHi(int day) {
define daytemps *t = find(day);
define if (t != NULL) return t->hi;
define else return -500;
}
define int daytemps::returnLow(int day) {
define daytemps *t = find(day);
define if (t != NULL) return t->low;
define else return -500;
} ```
The function find, which is used by returnHi and returnLow, operates much like function findPlace. However, find does not have to keep track of the address of the previous node. It simply starts at the root and searches until it finds the node that contains a match for the value in the parameter day. If a matching node is found, its address is returned to the function that called find. If a matching node is not found, the value NULL is returned to the function that called find.

```cpp
daytemps * temperatures::find(int day) {
    daytemps *current = root;
    while (current != NULL) {
        if (current->day == day) return current;
        else if (current->day < day) current = current->rightchild;
        else current = current->leftchild;
    }
    return NULL;
}
```

The functions listAllPreorder, listAllInorder and listAllPostorder simply output column headers and call matching recursive functions (preorder, inorder and postorder), passing root as the parameter of the call, to actually perform the traversal and output of the binary tree data structure.

```cpp
void temperatures::listAllPreorder( ) {
    cout << "Pre-order\n" << "Day\tHigh\tLow\n";
    preorder(root);
}

void temperatures::listAllInorder( ) {
    cout << "In-order\n" << "Day\tHigh\tLow\n";
    inorder(root);
}

void temperatures::listAllPostorder( ) {
    cout << "Post-order\n" << "Day\tHigh\tLow\n";
    postorder(root);
}
```
Binary trees can be traversed in three separate orders - pre-order, in-order and post-order. Since the algorithms that produce these three orders are variations of the same idea, by examining one of them we will be able to understand all of them.

The in-order algorithm operates as follows:

1. the function *inorder* is called with the address of a node (the root if the first call)
2. the function *inorder* is recursively called with the address of the left child of the current node (repeat step 2)
3. visit the current node (visiting a node means to access or manipulate the data contained in the node)
4. the function *inorder* is recursively called with the address of the right child of the current node (repeat from step 2)
5. fall out of recursion (to the end of step 2 or 4 of the calling instance of *inorder*)

Here is an example of how function *inorder* is generally written.

```c
void inorder(pointertype *p) {
    if (p != NULL) {
        inorder(p->leftchild);
        *** code for visitation goes here ***
        inorder(p->rightchild);
    }
}
```

The in-order algorithm visits nodes in a binary tree favoring the left most node in any sub-tree. If the binary tree on page 11 of this chapter is traversed using the in-order algorithm, we have seen previously that nodes will be visited in the following sequence:

1, 3, 5, 7, 9, 10, 11

To make the function *preorder* from *inorder*, just move the visitation code to before the first recursive call.

```c
void preorder(pointertype *p) {
    if (p != NULL) {
        *** code for visitation goes here ***
        preorder(p->leftchild);
        preorder(p->rightchild);
    }
}
```
The pre-order algorithm visits the nodes in a binary tree favoring the root of any sub-tree. If the binary tree on page 11 of this chapter is traversed using the pre-order algorithm, nodes will be visited in the following sequence, as has been seen:

7, 3, 1, 5, 11, 9, 10

To make the function \textit{postorder} from \textit{inorder}, just move the visitation code to after the second recursive call.

```c
void postorder(pointertype *p) {
    if (p != NULL) {
        postorder(p->leftchild);
        postorder(p->rightchild);
        *** code for visitation goes here ***
    }
}
```

The post-order algorithm visits the nodes in a binary tree favoring the leaves of any sub-tree. If the binary tree on page 11 of this chapter is traversed using the post-order algorithm, nodes will be visited in the following sequence, as we have seen:

1, 5, 3, 10, 9, 11, 7

In the class \textit{temperatures}, the functions \textit{preorder}, \textit{inorder} and \textit{postorder} all perform the recursive traversal of the binary tree for which they are named.

```c
void temperatures::preorder(daytemps *current) {
    if (current != NULL) {
        cout << current->day << ',' << current->hi << ','
             << current->low << endl;
        preorder(current->leftchild);
        preorder(current->rightchild);
    }
}
```

```c
void temperatures::inorder(daytemps *current) {
    if (current != NULL) {
        inorder(current->leftchild);
        cout << current->day << ',' << current->hi << ','
             << current->low << endl;
        inorder(current->rightchild);
    }
}
```
void temperatures::postorder(daytemps *current) {
    if (current != NULL) {
        postorder(current->leftchild);
        postorder(current->rightchild);
        cout << current->day << ' ' << current->hi << ' ' << current->low << endl;
    }
}

Lastly, the destructor calls the function dispose to recursively traverse the binary tree, deleting all nodes as it goes. Because it only visits a node when it will no longer be needed to reach any other part of the tree, the post-order algorithm is ideal for use in the dispose function.

temperatures::~temperatures( ) {
    dispose(root);
}

void temperatures::dispose(daytemps *current) {
    if (current != NULL) {
        dispose(current->leftchild);
        dispose(current->rightchild);
        delete current;
    }
}
Complete Listing of Class temperatures

```cpp
#include <iostream>
using namespace std;

struct daytemps {
    int day, hi, low;
    daytemps *leftchild, *rightchild;
};

class temperatures {
public:
    temperatures( ) {
        root = NULL;
    } 
    ~temperatures( ) {
        dispose(root);
    }
    void insertDay(int day, int high, int low);
    int returnHi(int day);
    int returnLow(int day);
    void listAllPreorder( );
    void listAllInorder( );
    void listAllPostorder( );
private:
    void preorder(daytemps *current);
    void inorder(daytemps *current);
    void postorder(daytemps *current);
    void dispose(daytemps *current);
    daytemps *findPlace(int day);
    daytemps *find(int day);
    daytemps *root;
};
temperatures::temperatures( ) {
    root = NULL;
}
temperatures::~temperatures( ) {
    dispose(root);
}
```
void temperatures::insertDay(int day, int high, int low) {
    if (root == NULL) {
        root = new daytemps();
        root->day = day;
        root->hi = high;
        root->low = low;
        root->leftchild = NULL;
        root->rightchild = NULL;
    } else {
        daytemps *current = findPlace(day);
        if (current != NULL && current->day != day) {
            daytemps *t = new daytemps();
            t->day = day;
            t->hi = high;
            t->low = low;
            t->leftchild = NULL;
            t->rightchild = NULL;
            if (day > current->day) current->rightchild = t;
            else current->leftchild = t;
        }
    }
}

int temperatures::returnHi(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->hi;
    else return -500;
}

int temperatures::returnLow(int day) {
    daytemps *t = find(day);
    if (t != NULL) return t->low;
    else return -500;
}

void temperatures::listAllPreorder() {
    cout << "Pre-order\n" << "Day\nHigh\nLow\n";
    preorder(root);
}
void temperatures::listAllInorder( ) {
    cout << "In-order
" << "Day\tHigh\tLow\n";
    inorder(root);
}

void temperatures::listAllPostorder( ) {
    cout << "Post-order\n" << "Day\tHigh\tLow\n";
    postorder(root);
}

void temperatures::preorder(daytemps *current) {
    if (current != NULL) {
        cout << current->day << \t << current->hi << \t
        << current->low << endl;
        preorder(current->leftchild);
        preorder(current->rightchild);
    }
}

void temperatures::inorder(daytemps *current) {
    if (current != NULL) {
        inorder(current->leftchild);
        cout << current->day << \t << current->hi << \t
        << current->low << endl;
        inorder(current->rightchild);
    }
}

void temperatures::postorder(daytemps *current) {
    if (current != NULL) {
        postorder(current->leftchild);
        postorder(current->rightchild);
        cout << current->day << \t << current->hi << \t
        << current->low << endl;
    }
}

void temperatures::dispose(daytemps *current) {
    if (current != NULL) {
        dispose(current->leftchild);
        dispose(current->rightchild);
        delete current;
    }
}
```c
daytemps * temperatures::findPlace(int day) {
    daytemps *previous = NULL;
    daytemps *current = root;
    while (current != NULL) {
        if (current->day == day) return current;
        previous = current;
        if (current->day < day) current = current->rightchild;
        else current = current->leftchild;
    }
    return previous;
}
```

```c
daytemps * temperatures::find(int day) {
    daytemps *current = root;
    while (current != NULL) {
        if (current->day == day) return current;
        else if (current->day < day) current = current->rightchild;
        else current = current->leftchild;
    }
    return NULL;
}
```

**Exercises 44.1**
Using pencil and paper, perform a pre-order traversal of the binary tree on page 1 of this chapter. Write out the order of the nodes visited.

**Exercises 44.2**
Using pencil and paper, perform a in-order traversal of the binary tree on page 1 of this chapter. Write out the order of the nodes visited.

**Exercises 44.3**
Using pencil and paper, perform a post-order traversal of the binary tree on page 1 of this chapter. Write out the order of the nodes visited.
**Programming Exercise 44.1**
Enter the class temperatures. Use the following main to test and debug it.

```cpp
void main( ) {
    temperatures t;
    t.insertDay(2,15,3);
    t.insertDay(5,2,1);
    t.insertDay(3,37,33);
    t.insertDay(3,0,0);  // this node should not appear in any listing
    t.insertDay(1,10,5);
    t.insertDay(4,56,23);
    cout << "High Temp: " << t.returnHi(3)
         << " Low Temp: " << t.returnLow(4) << endl;
    t.listAllPreorder( );
    t.listAllInorder( );
    t.listAllPostorder( );
}
```

**Programming Exercise 44.2**
Create and use two member functions for class temperatures. One should return the minimum low temperature and the other should return the maximum high temperature.

**Programming Exercise 44.3**
Just finding the maximum and minimum temperature may not be enough. It may be necessary to know what day the min and max values occurred. One way to return multiple values is to return a struct. Alter the functions created in programming exercise 44.2 so that a node of type daytemps is returned. Be sure that this is NOT the same node that is used in the data structure. Use these altered functions.

**Programming Exercise 44.4**
Create and use a member function called replace for class temperatures that receives as parameters values for day, hi and low, finds a node for a given day and replaces the data in it with the updated data. If a node for the value day does not exist, a new node should be created in the correct place and manner.