CS 3100/5100
Programming Project 2

We can located in an unsorted data set with N elements in O(N) time by comparing the search value to each element in the set until the desired item is found. To improve the performance of the search, the data set can be sorted and a binary search can be applied to find the desired item in O(logN) time. This assumes the data is stored in an array that allows random access to any position in the data set in O(1) time. If the data set is of fixed size, this is a very efficient approach to search. On the other hand, if the data set is subject to insertions and deletions, the cost of maintaining a sorted set in an array is high. To summarize,

Operation	Unsorted Data Set		Sorted Data Set	
	Average	Worst	Average	Worst
Search	N/2=O(N)	N=O(N)	logN/2=O(logN)	logN=O(logN)
Insert Item	O(1)	O(1)	N/2=O(N)	N=O(N)
Delete Item	N/2=O(N)	N=O(N)	N/2=O(N)	N=O(N)
Ordered Display	O(NlogN)	O(NlogN)	O(N)	O(N)

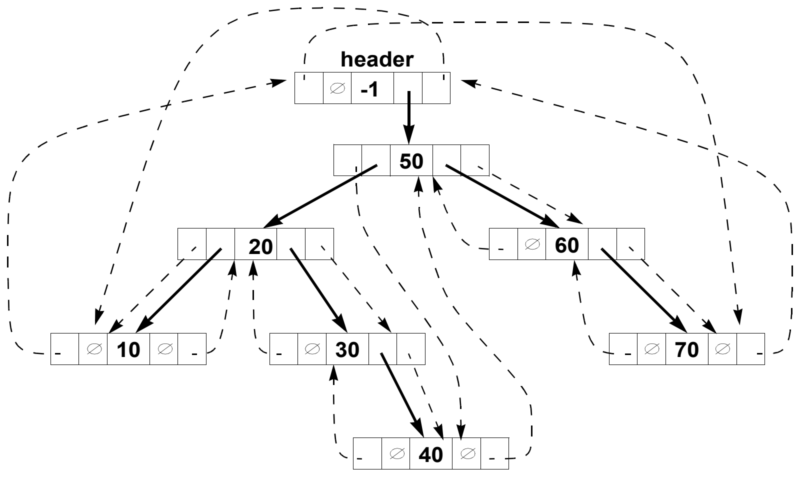
In this table, the cost of inserting or deleting an item does not include the cost of locating the item. The cost of inserting a new data item in an unsorted set is O(1) since the item can be added at the end of the array, but the cost of deleting an item is O(N) because the operation may require moving N-1 elements. If the data set is sorted the cost of search improves, but the cost of deletion and insertion increase due to the extra work needed to maintain a sorted data set. If the data set must be displayed in sorted ordered, clearly, maintaining the array in sorted order has an advantage.

An alternative to storing data in an array is to store the information in a binary tree structure. A binary tree is either empty of contains distinguished node call the root with two disjoint subtree referred to as the left and right subtrees (children) of the root, and each subtree is a binary tree. A binary search tree is a binary tree with the property that the value stored in the root node is greater than every value stored in its left subtree and less than or equal to every value stored in its right subtree. A binary search tree effectively maintains a sorted data set. The performance of a sorted array and a binary search tree are compared below.

Operation	Sorted Data Set		Binary Search Tree	
	Average	Worst	Average	Worst
Search	logN/2=O(logN)	logN=O(logN)	logN/2=O(logN)	N=O(N)
Insert Item	N/2=O(N)	N=O(N)	O(logN)	O(N)
Delete Item	N/2=O(N)	N=O(N)	O(logN)	O(N)
Ordered Display	O(N)	O(N)	kN=O(N)	kN=O(N)

The average performance of a binary search tree is better than a sorted data set for most operations. Unfortunately, a binary search tree can degenerate to a linked list. In this case, the performance approaches the behavior of an unsorted data set. Many techniques have been developed that adjust the shape of the tree to avoid the worst case behavior. We will discuss some of these techniques in class.

In this assignment, we will focus on the problem of displaying the sorted data set. Notice the cost of displaying a sorted data set is kN=O(N). As you know, to display a tree in sorted order, the tree must be traversed using a recursive algorithm or an explicit stack. Effectively, you must move up and down the tree repeatedly to visit the nodes in the proper sequence. An alternative to this approach is to add special links to the tree that can be followed to display the sorted data set without using recursion or a stack. This process is referred to as braiding a tree. The following picture should help you understand the concept of braiding a tree.



In a braided search tree, each node has four pointers. There are left and right pointers that reference the left and right subtrees respectively (solid arrows). In addition, there is another pair of pointers that are used to form a doubly linked circular list of tree nodes. By following these nodes in a forward direction, all of the elements in the tree can be displayed in sorted order without using recursion. If the nodes are traversed using the back link the data is displayed in a reverse order. In essence, the forward pointer lead to the inorder successor of a node and the back link leads to a inorder predecessor. Notice the tree begins with a dummy header node with a right pointer to the true root of the tree. The forward and back pointers in this node are initialized so the next step from the header is the smallest value in the tree and the previous step is the largest.

This problem represents a good opportunity to enhance your skill with C++ and inheritance. First, we will define a simple doubly linked circular list **node** that does not include any data:

```
class node
{
        public:
                node( void );                   // node constructor both flink and blink point to node
                virtual   ~node(void);          // node destructor
                node*    next( void );          // return pointer to the next node
                node*    prev( void );          // return a pointer to the previous node
                node*    insert( node* );       // insert this node after the argument node
                node*    remove( void );        // remove this node form the list
        protected:
                node*           flink;          // pointer to the next node (i. e.  front link)
                node*           blink;          // pointer to the previous node (i. e.  back link)
        friend class braidedTree;
};
```

Second, we will define a binary **treeNode**:

```
class treeNode
{
        public:
                treeNode( int );                // construct a tree node with an integer value
                virtual ~treeNode();            // treeNode destructor
                                                // we are not adding a copy constructor or overloaded operator=
        protected:
                int             data;           // data contents of the tree
                treeNode*       leftTree;       // pointer to the left subtree
                treeNode*       rightTree;      // pointer to the right subtree
        friend class braidedTree;
};
```

Finally, we will define a **braidedNode** that inherits both a circular list node and a treeNode:

class braidedNode : public node, public treeNode

```
{
        public:
                braidedNode( int );                                  // construct a braided node with a data value
                bradiedNode*          leftChild(void);               // return pointer to left subtree
                braidedNode*          rightChild(void);              // return pointer to right subtree
                braidedNode*          next(void);                    // return a pointer to the successor braidedNode
                braidedNode*          prev(void);                    // return a pointer to the previous braidedNode
        friend class braidedTree;
};
```

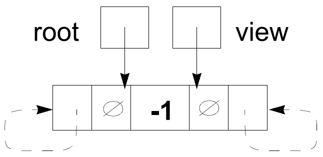
A braided tree consists of the following:

```
class braidedTree
{
        public:
                braidedTree(void);
                ~braidedTree(void);

                int next();
                int prev();
                int value();

                void display();

                boolean isFirst();
                boolean isLast();
                boolean isHead();
                boolean isEmpty();
```

```
                boolean find( int );
                int findMin( void );
                int findMax( void );

                boolean insert( int );
                boolean remove( int );

                int removeMin(void);
                int removeMax(void);

        private:
                braidedNode*          root;
                braidedNode*          view;
};
```

A braidedTree contains two pointers, a root pointer that references a dummy header node and a view pointer that references the node that will be manipulated in the next operation. The actions associated with each function are described below.

braidedTree(void); The constructor is called to allocate a braidedNode with an arbitrary data value (e. g. -1). The braidedNode is a header node for the tree. The root and view pointers should reference this header node. The node class constructor should be automatically called which results in setting both the flink and blink to reference the header node. This should leave an initial tree that looks like this:



~braidedTree(void); Destructor called to release the tree.

int next(); If the tree is not empty, this should advance the view pointer to the position of the inorder successor of the node currently referenced by view and return the value of the node. If view is currently at the header node, the next node is the node containing the smallest value in the tree.

int prev(); If the tree is not empty, this should advance the view pointer to the position of the inorder predecessor of the node currently referenced by view and return the value of the node. If view is currently at the header node, the prev node is the node containing the largest value in the tree.

int value(); Returns the value in the node referenced by view.

void display(); Display the tree data values in sorted order. This should be a non-recursive function that uses the node links.

boolean isFirst(); Returns true if it is not an empty tree and the view pointer references the node with the smallest data value.

boolean isLast();	Returns true if it is not an empty tree and the view pointer references the node with the largest data value.
boolean isHead();	Returns true if the view pointer references the header node.
boolean isEmpty();	Returns true if the tree contains only the header node.
boolean find(int x);	Locate the value x in the search tree and return true if found. As a side effect, position the view pointer to the node containing x or set it to the header node if x is not in the tree.
int findMin(void);	If the tree is not empty, locate and return the smallest value in the search tree. As a side effect, position the view pointer to the node containing the smallest value.
int findMax(void);	If the tree is not empty, locate and return the largest value in the search tree. As a side effect, position the view pointer to the node containing the largest value.
boolean insert(int x);	Insert the value x into the tree. Return if the operation succeeds. Return false if x is a duplicate of a value already in the tree. Set the view pointer to reference the newly inserted node or leave it unchanged if the operation fails.
boolean remove(int x);	Locate and remove the value x from the search tree, return true if the operation succeeds, otherwise return false. Position the view pointer to the previous node on success or header on failure.
int removeMin(void);	If the tree is not empty remove the node containing the smallest value and return the value. If view pointer was pointing to the node containing the smallest value, set it to the header node otherwise leave it unchanged.
int removeMax(void);	If the tree is not empty remove the node containing the largest value and return the value. If view pointer was pointing to the node containing the largest value, set it to the previous node otherwise leave it unchanged.

Provid a simple command interpreter to test your program. The interpreter must process the following single character commands:

N	-- move to the successor and display the value
P	-- move to the predecessor and display the value
V	-- display the value of the view node
D	-- display the inorder data
F	-- print true if view is set to first, false otherwise
L	-- print true if view is set to last, false otherwise
H	-- print true if view is set to head, false otherwise
E	-- print true if the tree is empty, false otherwise

S value	-- print true if the value is in the tree, false otherwise
<	-- print the smallest value in the tree
>	-- print the largest value in the tree
+ value	-- insert value into the tree
- value	-- remove value from the tree
M	-- remove and display the minimum value
X	-- remove and display the maximum value
Q	-- terminate the program

You may add operations to the private section of the braidedTree class, but do not alter the public interface. Files called braided. h, braided. cpp, and project2. cpp will be placed in the project2 directory along with a Makefile.