Binary Search Tree ADT

objectives

In this laboratory you will:

- Create an implementation of the Binary Search Tree ADT using a linked tree structure
- Examine how an index can be used to retrieve records from a database file and construct an indexing program for an accounts database
- Create operations that compute the height of a tree and output the data items in a tree whose keys are less than a specified key
- Analyze the efficiency of your implementation of the Binary Search Tree ADT

Overview

In this laboratory, you examine how a binary tree can be used to represent the hierarchical search process embodied in the binary search algorithm.

The binary search algorithm allows you to efficiently locate a data item in an array provided that each array data item has a unique identifier, called its **key**, and that the array data items are stored in order based on their keys. Given the following array of keys,

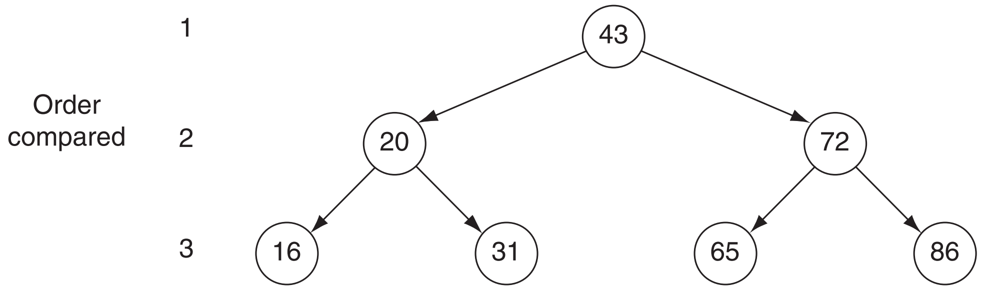
Index	0	1	2	3	4	5	6
Key	16	20	31	43	65	72	86

a binary search for the data item with key 31 begins by comparing 31 with the key in the middle of the array, 43. Because 31 is less than 43, the data item with key 31 must lie in the lower half of the array (entries 0–2). The key in the middle of this subarray is 20. Because 31 is greater than 20, the data item with key 31 must lie in the upper half of this subarray (entry 2). This array entry contains the key 31. Thus, the search terminates with success.

Although the comparisons made during a search for a given key depend on the key, the relative order in which comparisons are made is invariant for a given array of data items. For instance, when searching through the previous array, you always compare the key that you are searching for with 43 before you compare it with either 20 or 72. Similarly, you always compare the key with 72 before you compare it with either 65 or 86. The order of comparisons associated with this array is shown below.

Index	0	1	2	3	4	5	6
Key	16	20	31	43	65	72	86
Order compared	3	2	3	1	3	2	3

The hierarchical nature of the comparisons that are performed by the binary search algorithm is reflected in the following tree.



Observe that for each key K in this tree, all of the keys in K's left subtree are less than K and all of the keys in K's right subtree are greater than K. Trees with this property are referred to as **binary search trees**.

When searching for a key in a binary search tree, you begin at the root node and move downward along a branch until either you find the node containing the key or you reach a leaf node without finding the key. Each move along a branch corresponds to an array subdivision in the binary search algorithm. At each node, you move down to the left if the key you are searching for is less than the key stored in the node, or you move down to the right if the key you are searching for is greater than the key stored in the node.

Binary Search Tree ADT

Data Items

The data items in a binary search tree are of generic type DT. Each data item has a key (of generic type KF) that uniquely identifies the data item. Data items usually include additional data. Objects of type KF must support the six basic relational operators. Objects of type DT must provide a functional `getKey()` that returns a data item's key.

Structure

The data items form a binary tree. For each data item D in the tree, all the data items in D's left subtree have keys that are less than D's key and all the data items in D's right subtree have keys that are greater than D's key.

Operations

`BSTree ()`

Requirements:
None

Results:
Constructor. Creates an empty binary search tree.

`~BSTree ()`

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a binary search tree.

`void insert ( const DT &newDataItem ) throw ( bad_alloc )`

Requirements:
Binary search tree is not full.

Results:
Inserts `newDataItem` into a binary search tree. If a data item with the same key as `newDataItem` already exists in the tree, then updates that data item's nonkey fields with `newDataItem`'s nonkey fields.

`bool retrieve ( KF searchKey, DT &searchDataItem ) const`

Requirements:
None

Results:
Searches a binary search tree for the data item with key `searchKey`. If this data item is found, then copies the data item to `searchDataItem` and returns `true`. Otherwise, returns `false` with `searchDataItem` undefined.

`bool remove ( KF deleteKey )`

Requirements:
None

Results:
Deletes the data item with key `deleteKey` from a binary search tree. If this data item is found, then deletes it from the tree and returns `true`. Otherwise, returns `false`.

`void writeKeys () const`

Requirements:
None

Results:
Outputs the keys of the data items in a binary search tree. The keys are output in ascending order, one per line.

`void clear ()`

Requirements:
None

Results:
Removes all the data items in a binary search tree.

`bool isEmpty () const`

Requirements:
None

Results:
Returns `true` if a binary search tree is empty. Otherwise, returns `false`.

`bool isFull () const`

Requirements:
None

Results:
Returns `true` if a binary search tree is full. Otherwise, returns `false`.

`void showStructure () const`

Requirements:
None

Results:
Outputs the keys in a binary search tree. The tree is output with its branches oriented from left (root) to right (leaves); that is, the tree is output rotated counterclockwise 90 degrees from its conventional orientation. If the tree is empty, outputs "Empty tree". Note that this operation is intended for debugging purposes only.

Laboratory 11: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 11: Prelab Exercise

Name ____ Date ______________________

Section ____

Step 1: Implement the operations in Binary Search Tree ADT using a linked tree structure. As with the linear linked structures you developed in prior laboratories, your implementation of the linked tree structure uses a pair of classes: one for the nodes in the tree (`BSTreeNode`) and one for the overall tree structure (`BSTree`). Each node in the tree should contain a data item (`dataItem`) and a pair of pointers to the node's children (`left` and `right`). Your implementation should also maintain a pointer to the tree's root node (`root`). Base your implementation on the following declarations from the file *bstree.hs*. An implementation of the `showStructure` operation is given in the file *show11.cpp*.

```
template < class DT, class KF >
class BSTreeNode                  // Facilitator for the BSTree class
{
  private:

    // Constructor
    BSTreeNode ( const DT &nodeDataItem,
                 BSTreeNode *leftPtr, BSTreeNode *rightPtr );

    // Data members
    DT dataItem;         // Binary search tree data item
    BSTreeNode *left,    // Pointer to the left child
               *right;   // Pointer to the right child

  friend class BSTree<DT,KF>;
};

template < class DT, class KF >    // DT : tree data item
class BSTree                       // KF : key field
{
  public:

    // Constructor
    BSTree ();

    // Destructor
    ~BSTree ();

    // Binary search tree manipulation operations
    void insert ( const DT &newDataItem )         // Insert data item
        throw ( bad_alloc );
    bool retrieve ( KF searchKey, DT &searchDataItem ) const;
                                                  // Retrieve data item
    bool remove ( KF deleteKey );                 // Remove data item
    void writeKeys () const;                      // Output keys
    void clear ();                                // Clear tree
    // Binary search tree status operations
    bool isEmpty () const;                          // Tree is empty
    bool isFull () const;                           // Tree is full

    // Output the tree structure -- used in testing/debugging
    void showStructure () const;

  private:

    // Recursive partners of the public member functions -- insert
    // prototypes of these functions here.
    void showSub ( BSTreeNode<DT,KF> *p, int level ) const;

    // Data member
    BSTreeNode<DT,KF> *root;   // Pointer to the root node
};
```

Step 2: The declaration of the BSTree class in the file *bstree.hs* does not include prototypes for the recursive private member functions needed by your implementation of the Binary Search Tree ADT. Add these prototypes and save the resulting class declarations in the file *bstree.h*.

Step 3: Save your implementation of the Binary Search Tree ADT in the file *bstree.cpp*. Be sure to document your code.

Laboratory 11: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *test11.cpp* allows you to interactively test your implementation of the Binary Search Tree ADT using the following commands.

Command	Action
`+key`	Insert (or update) the data item with the specified key.
`?key`	Retrieve the data item with the specified key and output it.
`-key`	Delete the data item with the specified key.
`K`	Output the keys in ascending order.
`E`	Report whether the tree is empty.
`F`	Report whether the tree is full.
`C`	Clear the tree.
`Q`	Quit the test program.

Step 1: Prepare a test plan for your implementation of the Binary Search Tree ADT. Your test plan should cover trees of various shapes and sizes, including empty, single branch, and single data item trees. A test plan form follows.

Step 2: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Binary Search Tree ADT

Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory 11: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

A **database** is a collection of related pieces of information that is organized for easy retrieval. The following set of accounts records, for instance, form an accounts database.

Record #	*Account ID*	*First name*	*Last name*	*Balance*
0	6274	James	Johnson	415.56
1	2843	Marcus	Wilson	9217.23
2	4892	Maureen	Albright	51462.56
3	8337	Debra	Douglas	27.26
4	9523	Bruce	Gold	719.32
5	3165	John	Carlson	1496.24

Each record in the accounts database is assigned a record number based on that record's relative position within the database file. You can use a record number to retrieve an account record directly, much as you can use an array index to reference an array data item directly. The following program from the file *getdbrec.cpp*, for example, retrieves a record from the accounts database in the file *accounts.dat*.

```
#include <iostream>
#include <fstream>

using namespace std;

//--------------------------------------------------------------------
//
// Declarations specifying the accounts database
//

const int nameLength      = 11;   // Maximum number of characters in
                                  //   a name
const long bytesPerRecord = 38;   // Number of bytes used to store
                                  //   each record in the accounts
                                  //   database file

struct AccountRecord
{
    int acctID;                   // Account identifier
    char firstName[nameLength],   // Name of account holder
         lastName[nameLength];
    double balance;               // Account balance
};
```

```
void main ()
{
    ifstream acctFile ("accounts.dat");   // Accounts database file
    AccountRecord acctRec;                // Account record
    long recNum;                          // User input record number

    // Get the record number to retrieve.

    cout << endl << "Enter record number: ";
    cin >> recNum;

    // Move to the corresponding record in the database file using the
    // seekg() function.

    acctFile.seekg(recNum*bytesPerRecord);

    // Read in the record.

    acctFile >> acctRec.acctID >> acctRec.firstName
             >> acctRec.lastName >> acctRec.balance;

    // Display the record.

    cout << recNum << " : " << acctRec.acctID << " "
         << acctRec.firstName << " " << acctRec.lastName << " "
         << acctRec.balance << endl;
}
```

Record numbers are assigned by the database file mechanism and are not part of the account information. As a result, they are not meaningful to database users. These users require a different record retrieval mechanism, one that is based on an account ID (the key for the database) rather than a record number.

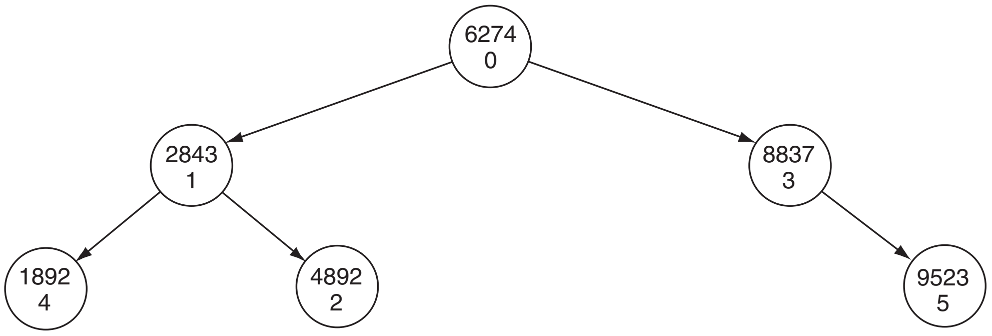
Retrievals based on account ID require an index that associates each account ID with the corresponding record number. You can implement this index using a binary search tree in which each data item contains two fields: an account ID (the key) and a record number.

```
struct IndexEntry
{
    int acctID;                           // (Key) Account identifier
    long recNum;                          // Record number

    int getKey () const
        { return acctID; }                // Return key field
};

BSTree<IndexEntry,int> index;             // Database index
```

You build the index by reading through the database account by account, inserting successive (account ID, record number) pairs into the tree as you progress through the file. The following index tree, for instance, was produced by inserting the account records shown above into an (initially) empty tree.



Given an account ID, retrieval of the corresponding account record is a two-step process. First, you retrieve the data item from the index tree that has the specified account ID. Then, using the record number stored in the index data item, you read the corresponding account record from the database file. The result is an efficient retrieval process that is based on account ID.

Step 1: Using the program shell given in the file *database.cs* as a basis, create a program that builds an index tree for the accounts database in the file *accounts.dat*. Once the index is built, your program should

- Output the account IDs in ascending order
- Read an account ID from the keyboard and output the corresponding account record

Step 2: Test your program using the accounts database in the text file *accounts.dat*. A copy of this database in given below. Try to retrieve several account IDs, including account IDs that do *not* occur in the database. A test plan form follows.

Record #	*Account ID*	*First name*	*Last name*	*Balance*
0	6274	James	Johnson	415.56
1	2843	Marcus	Wilson	9217.23
2	4892	Maureen	Albright	51462.56
3	8337	Debra	Douglas	27.26
4	9523	Bruce	Gold	719.32
5	3165	John	Carlson	1496.24
6	1892	Mary	Smith	918.26
7	3924	Simon	Becker	386.85
8	6023	John	Edgar	9.65
9	5290	George	Truman	16110.68
10	8529	Ellen	Fairchild	86.77
11	1144	Donald	Williams	4114.26

Test Plan for the Accounts Database Indexing Program

Test Case	*Expected Result*	*Checked*

Laboratory 11: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

Binary search trees containing the same data items can vary widely in shape depending on the order in which the data items were inserted into the trees. One measurement of a tree's shape is its **height**—that is, the number of nodes on the longest path from the root node to any leaf node. This statistic is significant because the amount of time that it can take to search for a data item in a binary search tree is a function of the height of the tree.

```
int getHeight () const;
```

Requirements:
None

Results:
Returns the height of a binary search tree.

You can compute the height of a binary search tree using a postorder traversal and the following recursive definition of height:

$$\mathit{height}(\mathrm{p}) = \begin{cases} 0 & \mathit{if}\ \mathrm{p} = 0\ (\mathit{base\ case}) \\ \max(\mathit{height}(\mathrm{p} \to \mathrm{left}),\ \mathit{height}(\mathrm{p} \to \mathrm{right})) + 1 & \mathit{if}\ \mathrm{p} \neq 0\ (\mathit{recursive\ step}) \end{cases}$$

Step 1: Implement this operation and add it to the file *bstree.cpp*. A prototype for this operation is included in the declaration of the BSTree class in the file *bstree.h*.

Step 2: Activate the 'H' (height) command in the test program in the file *test11.cpp* by removing the comment delimiter (and the character 'H') from the lines that begin with "//H".

Step 3: Prepare a test plan for this operation that covers trees of various shapes and sizes, including empty and single-branch trees. A test plan form follows.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the height operation, correct them and execute your test plan again.

Test Plan for the getHeight Operation

Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory 11: In-lab Exercise 1

Name ________________________________ Date ____________________

Section ________________________________

You have created operations that retrieve a single data item from a binary search tree and output all the keys in a tree. The following operation outputs only those keys that are less than a specified key.

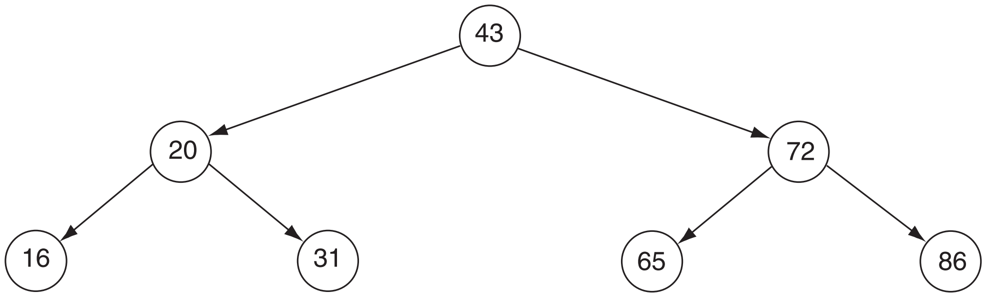
```
void writeLessThan ( KF searchKey ) const
```

Requirements:
None

Results:
Outputs the keys in a binary search tree that are less than `searchKey`. The keys are output in ascending order. Note that `searchKey` need not be a key in the tree.

You could implement this operation using an inorder traversal of the entire tree in which you compare each key with `searchKey` and output those that are less than `searchKey`. Although successful, this approach is inefficient. It searches subtrees that you know cannot possibly contain keys that are less than searchKey.

Suppose you are given a `searchKey` value of 37 and the following binary search tree:



Because the root node contains the key 43, you can determine immediately that you do not need to search the root node's right subtree for keys that are less than 37. Similarly, if the value of `searchKey` were 67, then you would need to search the root node's right subtree but would not need to search the right subtree of the node whose key is 72. Your implementation of the `writeLessThan` operation should use this idea to limit the portion of the tree that must be searched.

Step 1: Implement this operation and add it to the file *bstree.cpp*. A prototype for this operation is included in the declaration of the BSTree class in the file *bstree.h*.

Step 2: Activate the '<' (less than) command in the test program in the file *test11.cpp* by removing the comment delimiter (and the character '<') from the lines that begin with "//<".

Step 3: Prepare a test plan for this operation that includes a variety of trees and values for `searchKey`, including values of `searchKey` that do *not* occur in a particular tree. Be sure to include test cases that limit searches to the left subtree of the root node, the left subtree and part of the right subtree of the root node, the leftmost branch in the tree, and the entire tree. A test plan form follows.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the `writeLessThan` operation, correct them and execute your test plan again.

Test Plan for the `writeLessThan` Operation

Test Case	*Commands*	*Expected Result*	*Checked*