CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 7

In this project we will create a program that is very similar in operation to pa1, this time in C++. The main program will be called Order.c, and will use a Dictionary ADT based on a Binary Search Tree.

The Dictionary ADT
The Dictionary ADT maintains a set of pairs whose members are called *key* and *value*, respectively. A state of this ADT is a set (possibly empty) of such pairs. The file Dictionary.h is posted in /Examples/pa7 and contains the following typedefs for key and value.

```
typedef std::string keyType;
typedef int valType;
```

Think of the key as being some kind of identifying information, such as an account number, and the value as being data associated with that account. The Dictionary ADT will enforce the constraint that all keys are unique, while values may occur multiple times.

The main Dictionary operations are

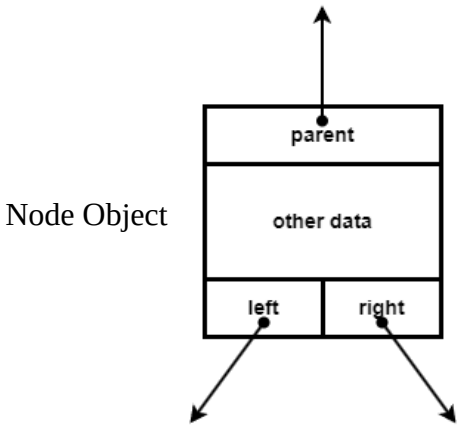
getValue(k): Return a reference to the value corresponding to key *k*. Pre: such a pair exists.

setValue(k, v): If a pair with key *k* exists, overwrite its value with *v*, otherwise insert the pair (k, v).

remove(k): Delete the pair with key *k*. Pre: such a pair exists.

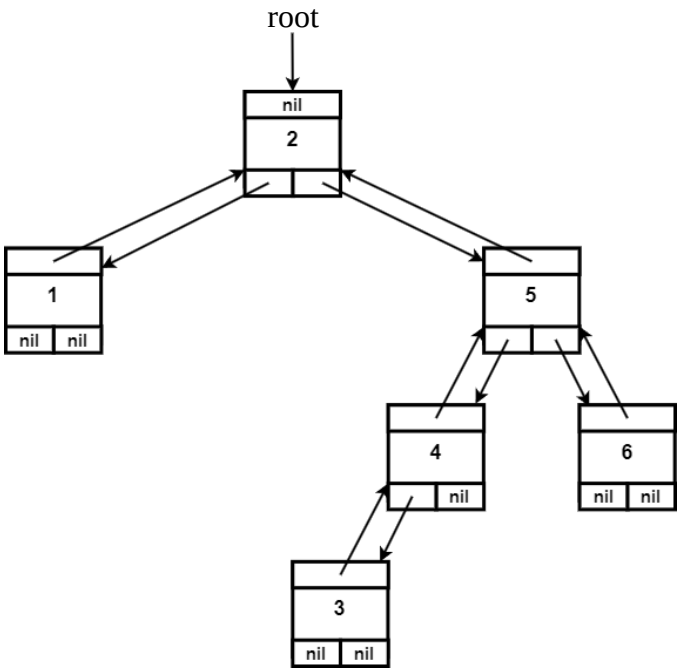
The Dictionary will also support a built-in iterator called *current*, that allows the client to step through the keys in alphabetical order, somewhat like *cursor* in the various incarnations of our List ADT. Other operations, including some suggested helper functions, along with their descriptions, are included in Dictionary.h.

Binary Search Trees
Before you begin this project, you should read Chapter 12 of our text (CLRS pp. 286-307.) Basically, a BST is a generalization of a (doubly) linked list, in which each Node has two next pointers, called *left child* and *right child*, respectively. The prev pointer in a linked list is replaced by *parent*.



In this project, the "other data" will consist of one (key, value) pair in our Dictionary. For purposes of this simplified discussion though, "other data" will be a single integer which we call key.

A binary tree assembled out of these Node objects looks something like this:



Two more conditions, called the Binary Search Tree properties, are necessary for such a structure to be a BST. Let x and y be Nodes in a BST. Then

(1) if y is in the *left subtree* of x (i.e. y is a descendant of x's left child), then key$[y] \leq$ key$[x]$,
(2) if y is in the *right subtree* of x (i.e. y is a descendant of x's right child), then key$[x] \leq$ key$[y]$.

Observe that the preceding example satisfies these properties, which make a number of sorting, searching and query algorithms possible. Other algorithms perform insertions and deletions in such a way as to maintain the BST properties. All of these algorithms will be discussed at length during lecture, and will be implemented by you in this project, some as ADT operations and some as helper functions.

Program Operation

The top-level client in this project will be called Order.cpp. It will read in the lines of an input file, each line being a (unique) single string, then insert these strings (as keys) into a Dictionary. The corresponding values will be the line number in the input file where the string was read. Your program will write two string representations of the Dictionary to the output file. The first string will consist of pairs of the form

```
"key : value\n"
```

so that when printed, each pair appears on its own line. The keys will be in alphabetical order. The second representation will consist of keys only, one to a line, with the order obtained from a pre-order tree walk. Order.cpp is quite simple to create once the Dictionary is complete. A good starting point would be the program FileIO.cpp (in /Examples/C++.) As usual, a weak test of Dictionary operations is included, called DictionaryClient.cpp. You will of course create your own tester for this project, which hopefully is somewhat more stringent. Several matched pairs of input-output files are also included in /Examples/pa7, along with a random input file generator. A file called English.txt containing over 194,00 English language words, found here, can be used by the input file generator to create input files of any desired size. Note that we will not test your program on input files with multiple identical lines. You may therefore assume that each line contains a unique string, suitable for use as a key in the Dictionary.

What to turn in
Submit the following 6 files to your pa7 directory on **git.ucsc.edu**.

README.md	Written by you, a catalog of submitted files and any notes to the grader
Makefile	Provided, alter as you see fit
Dictionary.h	Provided, you may alter the "helper functions" section, but nothing else
Dictionary.cpp	Written by you, the majority of work for this project
DictionaryTest.cpp	Written by you, your test client of the Dictionary ADT
Order.cpp	Written by you, the client for this project

Submit no other files. If you submit input files, output files, English.txt, excecutable files, or any file not listed above, you will lose points. Makefile should be capable of making the executables DictionaryTest and Order, and should contain a clean utility that removes all binary files. To get full credit, your project must implement all required files and functions, compile without errors or warnings, produce correct output on our Dictionary and Order tests, and produce no memory errors under valgrind.

As usual points are deducted both for neglecting to include required files, for misspelling any filenames and for submitting additional unwanted files. Start early, ask plenty of questions, and submit your project by the due date.