CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 7

In this project you will create a Dictionary ADT based on a Binary Search Tree, this time in C++, and use it to recreate the functionality of pa1. Begin by reading the Dictionary ADT handout to the end, if you have not already done so. Also read Chapter 12 of our text (CLRS pp. 286-307.)

The Dictionary ADT
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;
```

The main Dictionary operations are

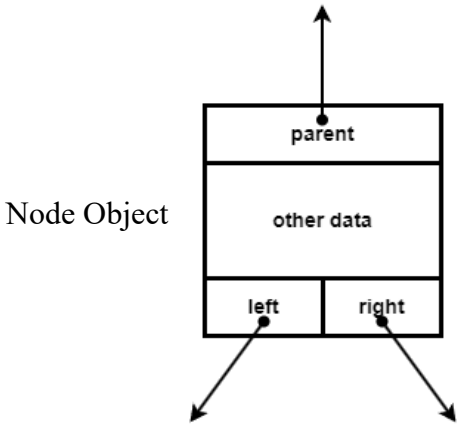
getValue(k): Return a reference to the value corresponding to key k. Pre: such a pair exists.

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.

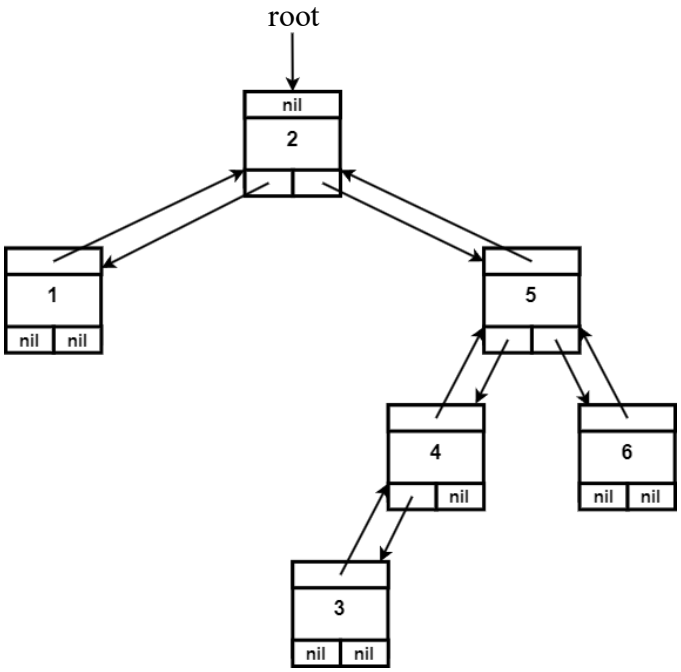
The Dictionary will also support a built-in iterator called *current*, that allows the client to step through the keys in alphabetical order, somewhat like *cursor* in the various incarnations of our List ADT. Other operations, including some suggested helper functions, along with their descriptions,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

Binary Search Trees
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 Words.cpp, and will operate identically to that Words.c from pa1. It will take two command line arguments giving the names of an input file and an output file, respectively.

```
Words <input file> <output file>
```

The input can be any text file. Words.cpp will parse the input file into individual words, discarding duplicate words. Instead of placing the unique words into an array though, it will insert them as keys into a Dictionary. The value corresponding to a key will record that key's insertion order into the Dictionary, starting with position 0. In other words, `value` is the array index at which `key` would have been placed by Words.c from pa1. Once the Dictionary is constructed, the output file (identical to that of pa1) can be constructed by iterating over it using the ADT operations `begin()`, `hasCurrent()`, `next()`, `currentKey()` and `currentVal()`, which are described in the file Dictionary.h.

The program FileIO.cpp (in /Examples/C++) would be a good starting point for the client Words.cpp. As usual, a weak test of Dictionary operations is included, called DictionaryClient.cpp along with its output DictionaryClientOut. You will of course create your own tester for this project, which contains more

rigorous tests. Several matched pairs of input-output files are also included in /Examples/pa7. These files are unchanged from pa1. You may use the random input file generator and the file English.txt posted in Examples/pa1 to create more input files for this project.

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
Words.cpp	Written by you, the client for this project

Makefile should be capable of making the executables DictionaryTest and Words, 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 Words 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.