CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 6

In this project we return to the C language to re-create the Lex.c program from pa1. This time however, the ADT used by Lex will be a *Dictionary* instead of a List. Your Dictionary ADT will be based on an underlying data structure called a Binary Search Tree (BST).

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. In general, key and value can be of any type. You will build a flexible version of the Dictionary in which these types are defined as macros in the header the file Dictionary.h. 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. With this model in mind, a Dictionary usually enforces the constraint that no two distinct pairs may have the same key. There are other situations however (like the Lex module in this project) in which it is useful to relax this constraint. Your Dictionary constructor will contain a switch that determines whether or not it will accept duplicate keys.

The main Dictionary ADT operations are

lookup(k): If a pair with key *k* exists in the Dictionary, return the associated value. Otherwise return nil.

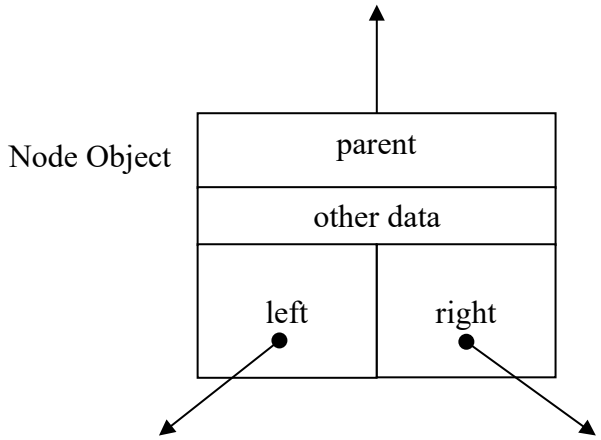
insert(k, v): Add a new pair to the Dictionary. Pre: (depending on switch) No pair with key *k* exists.

delete(k): Remove a pair with key *k* from the Dictionary. Pre: A pair with key *k* exists.

Other peripheral operations and their descriptions will be included in the Dictionary.h file for this project, which is posted on the course webpage.

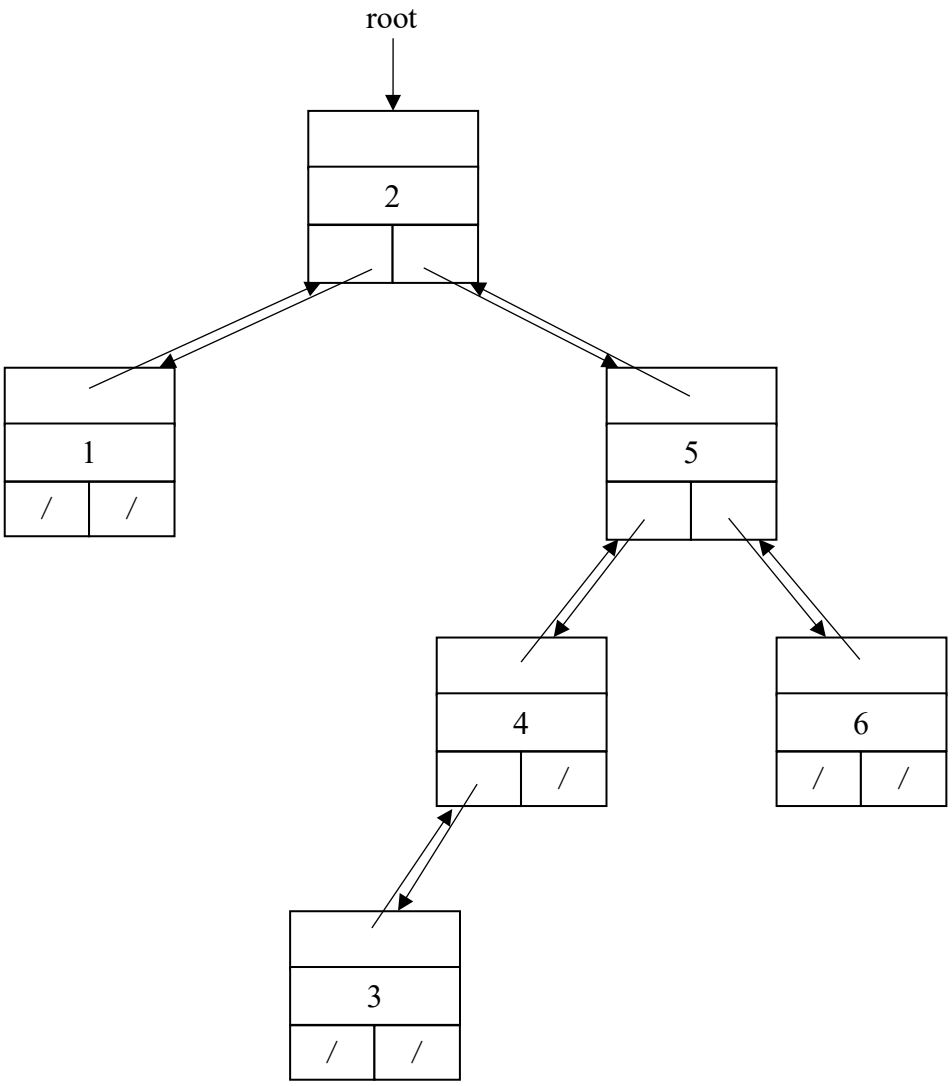
Binary Search Trees

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



Two more conditions, called the Binary Search Tree properties, are necessary for such a structure to be a BST. If x and y are 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$[x]$,

and

(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 implemented by you in this project (some as ADT operations and some as helper functions).

Your Dictionary.c file will define a struct called DictionaryObj. It should contain at minimum a Node reference to the root Node of a BST. Other possible fields in this struct will be discussed in lecture.

Program Operation
The top level client in this project will be called Lex.c, and its operation will be exactly as described in pa1. Most of the code you wrote for Lex.c in pa1 can be re-used. In particular, the code for reading lines from the input file, and storing those lines in a string array can go through unchanged. It will be up to you to figure out how to use the Dictionary ADT to accomplish the task of printing out the unsorted array in sorted order. (Hint: study the example DictionaryClient.c which is posted on the webpage.)

What to turn in
Submit the following 6 files to your pa5 directory on **git.ucsc.edu**.

README	Written by you, a catalog of submitted files and any notes to the grader
Makefile	Provided, alter as you see fit
Dictionary.h	Provided, **do not alter**
Dictionary.c	Written by you, the majority of work for this project
DictionaryTest.c	Written by you, your test client of the Dictionary ADT
Lex.c	Written by you, the client for this project

Makefile should be capable of making the executables DictionaryTest and Lex, 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 ADT and Lex tests, and produce no memory leaks under valgrind.

A number of other files are posted on the webpage **which you will not turn in**. Among these are DictionaryClient.c along with 4 matched pairs of input-output files. Also included is a file called English.txt containing 194,434 different English Language words (each occupying one line) in alphabetical order, and a python program called RandomInput5.py. This program will read English.txt, shuffle the lines, and place the result in a file you name. Use this as test input for your Lex program, and compare your output file to English.txt. (Actually the files in4 and out4 were produced in exactly this way, which is why out4 is identical to English.txt.)

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.