Section Handout #9

Problem 1: Choosing a Good Hash Function
Comment on the effectiveness and appropriateness of the followed suggested hash functions:

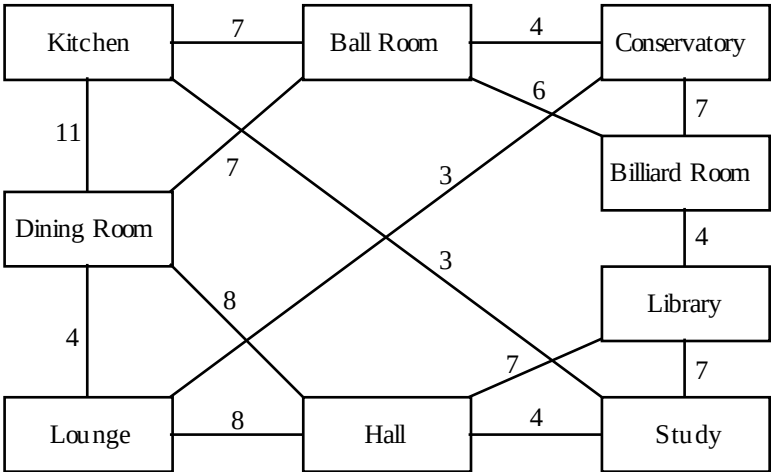
(a) The table has 2048 buckets. The search keys are peoples' names.
hash(key) = ASCII value of the first letter of key mod 2048

(b) The table has 1000 buckets. The search keys are integers in the range 0..999.
hash(key) = (Product of the digits of key) mod 1000

(c) The table has 1000 buckets, the search keys are integers in the range -32768 to +32767.
*hash(key) = (key*RandomInteger(1,100)) mod 1000*

Problem 2: Understanding graph algorithms

Those of you who have played Clue will recognize the following undirected graph, which shows the connections between the various rooms on the game board:



The numbers on the various arcs show the distance (measured in spaces on the board) between pairs of rooms. For example, the distance from the Hall to the Lounge is 4 steps, and the distance from the Ball Room to the Billiard Room is 6 steps. In this problem, the secret passages that connect the rooms at the corners of the board (the Kitchen-Study and Lounge-Conservatory arcs) are arbitrarily assumed to have distance 3.

a) Indicate the order of traversal for a depth-first search starting at the Lounge. Assume that iteration over a set chooses nodes in alphabetical order. Thus, the first step in the

depth-first search will be to the Conservatory, rather than to the Dining Room or Hall, which come later in the alphabet.

b) Indicate the order of traversal for a breadth-first search starting at the Kitchen. As before, assume that nodes in any set are processed in alphabetical order.

c) Trace the operation of Dijkstra s algorithm to find the minimum path from the Lounge to the Library.

d) Trace the operation of Kruskal's algorithm to find the minimum spanning tree of the graph. Assume that between arcs with the same cost, tie are broken in the following way:

Of all of the nodes that the arcs connect, the arc that connects the node with the name that comes first in the alphabet is processed first.

If two or more arcs connect that node, then the tie is broken with the other node and which comes first alphabetically.

For problems 3 and 4, we'll be using the following definitions for a node, arc, and graph:

```
//Forward declaration for the struct nodeT
struct arcT;

struct nodeT {
  string name;
  Set<arcT *> connected;
};

struct arcT {
  nodeT * start;
  nodeT * end;
};

struct graphT {
  Set<nodeT *> allNodes;
};
```

Problem 3: Finding Graph Cycles

A directed graph is considered to be *cyclic* if we can find a path through a graph from any node back to the same node (the definition is slightly different for undirected graphs). Using our graph traversal skills, we can find out whether an entire graph contains a cycle. Write a function:

```
bool IsCyclicGraph(graphT & graph)
```

that determines whether a given directed graph is cyclic (`true`) or acyclic (`false`). (Hint: It may be useful to keep track of which nodes have already been visited.)

Problem 4: Graph Construction

A Word Ladder is a path of words between two words. More specifically, given two words you want to find a way to turn one into the other one letter at a time where each

intermediate also is a word. For example, a word ladder between cat and run is cat -> cut -> rut -> run. cat -> cut ->cun -> run is not a valid word ladder, because cun is not a word.

a) Your first task is to convert the word ladder problem into a graph search problem. More specifically, you should write a function:

```
void ConstructGraph(graphT & graph, Lexicon & lex)
```

where graph is empty and lex has been initialized to hold all words. When `ConstructGraph` has completed, graph should be constructed so that each node is a word and each edge denotes that the words can be turned into each other by changing a single lettter. With this structure, finding the shortest word ladder between two words is equivalent to finding the shortest path between the two corresponding nodes.

For the purposes of this problem, assume that only five-letter words are allowable. Also assume that the lexicon only contains 5 letter words.

Hint: The Lexicon has a function called mapAll which allows you to call a function of your choice on every word in the Lexicon. It also allows you to pass in an additional argument to the function if you have need of other data structures. The general form of your callback function should take as arguments a string and your chosen data type and should return nothing (e.g. "void foo(string word, myType & secondArg)"). You can call mapAll like this: "lex.mapAll<myType>(foo, secondArg)"; this will call foo on every word in the Lexicon and will pass secondArg into the function call as well. This may be useful for setting up your nodes.

For parts b and c, you will use versions of breadth-first search and/or depth first search to solve the problems. For each problem, consider which search type to use and why.

b) Next, write a function

```
Stack<nodeT *> FindShortestLaddder(graphT & graph, nodeT start,
nodeT end)
```

which finds the shortest word ladder between the start word and the end word.

c) Finally write a function:

```
Stack<nodeT *> FindLongestLadder(graphT & graph, nodeT * word)
```

which, given a graph created by your function in part a finds the longest word ladder in the graph starting from the given word.