**CSCI 321 Computer Science III Summer 2019**

**Final Exam**

1. **Priority Queues**

One of the main applications of priority queues is in operating systems—for

scheduling jobs on a CPU. In this project you are to build a program that schedules

simulated CPU jobs. Your program should run in a loop, each iteration of

which corresponds to a time slice for the CPU. Each job is assigned a priority,

which is an integer between−20 (highest priority) and 19 (lowest priority), inclusive.

From among all jobs waiting to be processed in a time slice, the CPU must

work on a job with highest priority. In this simulation, each job will also come

with a length value, which is an integer between 1 and 100, inclusive, indicating

the number of time slices that are needed to process this job. For simplicity, you

may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job

runs for a number of time slices equal to its length. Your simulator must output

the name of the job running on the CPU in each time slice and must process a

sequence of commands, one per time slice, each of which is of the form “add job

name with length n and priority p” or “no new job this slice”.

Hint: Refer to Assignment 3 on how to construct a priority queue in Java.

see scheduler.java and driver.java

1. **Hash Tables**
2. Hash Tables need to deal with collisions. Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?

Hint: Think about which of the schemes use the array supporting the hash table exclusively and which of the schemes use additional storage external to the hash table.

Hash table chaining can perform efficiently with a load factor of more than 1. Tables based on open addressing schemes require a load factor that does not exceed 0.7 to be efficient. 30% of slots remain empty which leads to wasted memory.

When the table is full, linear and quadratic probing cant tolerate load. Separate chaining scheme places n item in n size table. The load factor can be greater than one which can be handled by storing them in separate chain.

1. Assume you want to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5 and store them in a hash table. Implement linear probing, and double hashing to handle collision. Print out the positions of keys stored. Hash functions for linear probing and double hashing as described as follows.

Linear probing: h(k) = (3k+5) mod 11

Double hashing: h1(k) = (3k+5) mod 11, h2(k) = 7 – (k mod 7)

Hint: Refer to Assignment 4 on implementing linear probing and double hashing.

h(12)=(3x12+5) mod 11 = 41 mod 11 = 8

h(44)=(3x44+5) mod 11 = 137 mod 11 = 5

h(13)=(3x13+5) mod 11 = 44 mod 11 = 0

h(88) = (3x88+5) mod 11 = 269 mod 11 = 5 -> collision

(h(88)+f(1)) mod 11 = (5+1) mod 11 = 6 mod 11 = 6

h(23) = (3x23+5) mod 11 = 74 mod 11 = 8 -> collision

(h(23)+f(1)) mod 11 = (8+1) mod 11 = 9 mod 11 = 9

h(94) = (3x94+5) mod 11 = 287 mod 11 = 1

h(11) = (3x11+5) mod 11 = 38 mod 11 = 5 -> collision

(h(11)+f(1)) mod 11 = (5+1) mod 11 = 6 mod 11 = 6 -> collision

(h(11)+f(2)) mod 11 = (5+2) mod 11 = 7 mod 11 = 7

h(39) = (3x39+5) mod 11 = 122 mod 11 = 1 -> collision

(h(39)+f(1)) mod 11 = (1+1) mod 11 = 2 mod 11 =2

h(20) = (3x20+5) mod 11 = 65 mod 11 = 10

h(16) = (3x16+5) mod 11 = 53 mod 11 = 9 -> collision

(h(16)+f(1)) mod 11 = (9+1) mod 11 = 10 mod 11 = 10 -> collision

(h(16)+f(2)) mod 11 = (9+2) mod 11 = 11 mod 11 = 0 -> collision

(h(16)+f(3)) mod 11 = (9+3) mod 11 = 12 mod 11 = 1 -> collision

(h(16)+f(4)) mod 11 = (9+4) mod 11 = 13 mod 11 = 2 -> collision

(h(16)+f(5)) mod 11 = (9+5) mod 11 = 14 mod 11 = 3 -> collision

h(5) = (3x5+5) mod 11 = 20 mod 11 = 9 -> collision

(h(5)+f(1)) mod 11 = (9+1) mod 11 = 10 mod 11 = 10 -> collision

(h(5)+f(2)) mod 11 = (9+2) mod 11 = 11 mod 11 = 0 -> collision

(h(5)+f(3)) mod 11 = (9+3) mod 11 = 12 mod 11 = 1 -> collision

(h(5)+f(4)) mod 11 = (9+4) mod 11 = 13 mod 11 = 2 -> collision

(h(5)+f(5)) mod 11 = (9+5) mod 11 = 14 mod 11 = 3 -> collision

(h(5)+f(6)) mod 11 = (9+6) mod 11 = 15 mod 11 = 4 -> collision

Hash table after inserting elements.

0 13

1 94

2 39

3 16

4 5

5 44

6 88

7 11

8 12

9 23

10 20

1. **Binary Search Trees (BST)**
2. List the main methods in BST ADT and their running times. Comparing BST with Hash Table, list their pros and cons.

The main methods in the abstract data type of BINARY SEARCH TREE are-

The lookup method

In general, to determine whether a given value is in the BST, we will start at the root of the tree and determine whether the value we are looking for:

is in the root

might be in the root's left subtree

might be in the root's right subtree

The insert method

The new item needs to go where you would have found it using lookup! If you don't put it there then you won't find it later.

The delete method

Deleting an item involves a search to locate the node that contains the value to be deleted.

BST vs Hash table

Hash Table supports Search,Insert,Delete operations in O(1) time whereas for binary search tree it has time complexity of O(Logn).

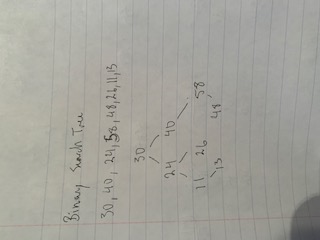
We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.

Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.

BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, O(1) is average time and some particular operations may be costly, especially when table resizing happens.

1. Insert, into an empty binary search tree, entries with keys 30, 40, 24, 58, 48, 26, 11, 13 (in this order). Draw the tree after each insertion.



1. Implement a binary search tree in Java and insert the keys above. Display all the keys stored in the tree. Hint: Refer to Lecture 6 Activity 1.

see BinarySearchTree.java

1. **AVL and Splay Trees**
2. Build an AVL tree with the following values: 12, 21, 34, 9, 12, 8. Perform the following operations: Remove 9 from the tree; Remove 8 from the tree. Show the whole process of inserting and deleting and also the final structure of the AVL tree. Hint: Refer to Lecture 6 Activity 1.
3. Implement an AVL tree in Java and redo part a. Print out the keys stored in the tree after insertion of 8, remove 9 from the tree, and remove 8 from the tree. Hint: Refer to Assignment 5.
4. Assume that you have inserted a sequence of keys into a Splay Tree: 8, 10, 88, 54, 22, 70. Draw the step-by-step workflow after searching for key 22. You need to start by showing the splay tree containing all the keys above. Hint: Refer to Assignment 5.
5. **Graphs**
6. We have discussed in class that the suitable representation of graph in DFS is adjacency list with running time O(n+m) since the main operation in DFS is to obtain the outgoing edges for a vertex. Explain why the DFS traversal runs in O(n2) time on an n-vertex simple graph that is represented with the adjacency matrix structure.

A depth first traversal will consider every rachable vertex of the graph. For each vertex considered, it must consider the set of neighbors of the vertex. It must examine the entire row of adjacency matric corresponding to the vertex. For each n of rows/vertices, it must consider the n – 1 non-diagonal entries which is O(n-2) operations.

1. Show the process of running DFS and BFS algorithm, respectively. The input graph is attached as follows.