3805ICT Advanced Algorithms

Assignment 2 (1-4)

Thien Phuc Vuong

S5193954

[Thienphuc.vuong@griffithuni.edu.au](mailto:Thienphuc.vuong@griffithuni.edu.au)

Contents

[Introduction 4](#_Toc104808410)

[Overview 4](#_Toc104808411)

[Complication and runtime 4](#_Toc104808412)

[1. Question 1 4](#_Toc104808413)

[1.1 Problem Statement 4](#_Toc104808414)

[1.2 Algorithm Description 4](#_Toc104808415)

[1.3 Complexity 4](#_Toc104808416)

[1.3.1 Space Complexity 4](#_Toc104808417)

[1.3.2 Time Complexity 4](#_Toc104808418)

[2. Question 2 7](#_Toc104808419)

[2.1 Problem Statement 7](#_Toc104808420)

[2.2 Algorithm Description 7](#_Toc104808421)

[2.3 Complexity 8](#_Toc104808422)

[2.3.1 Space Complexity 8](#_Toc104808423)

[1.3.2 Time Complexity 8](#_Toc104808424)

[3. Question 3 8](#_Toc104808425)

[3.1 Problem Statement 8](#_Toc104808426)

[3.2 Algorithm Description 8](#_Toc104808427)

[Van Emde Boas Trees: 9](#_Toc104808428)

[Red-Black Trees: 9](#_Toc104808429)

[3.3 Comparison 9](#_Toc104808430)

[Insertion 9](#_Toc104808431)

[Deletion 9](#_Toc104808432)

[Look up 9](#_Toc104808433)

[Accession 9](#_Toc104808434)

[Conclusion 10](#_Toc104808435)

[4. Question 4 10](#_Toc104808436)

[4.1 Problem Statement 10](#_Toc104808437)

[4.2 Algorithm Description 10](#_Toc104808438)

[a) Finds an actor’s Bacon number. 10](#_Toc104808439)

[b) Finds the actor with the highest Bacon number. 10](#_Toc104808440)

[c) Finds the minimum number of links between two arbitrary actors 11](#_Toc104808441)

[4.3 Complexity 11](#_Toc104808442)

[4.3.1 Space Complexity 11](#_Toc104808443)

[4.3.2 Time Complexity 11](#_Toc104808444)

# Introduction

## Overview

## Complication and runtime

# Question 1

## Problem Statement

A very large number of random numbers are added to a list. Design and implement an efficient data structure that will maintain a separate list of the k smallest numbers that are currently in the list. Space efficiency must be O(k + n). How would you handle deletions? Perform an amortized analysis of your data structure.

## Algorithm Description

Using 2 Binary heaps, one stores the k smallest number that has the largest root, and the other stores the rest number that has the smallest root. Each Heap uses a vector to store the binary three like list.

## Complexity

## Space Complexity

The complexity is O(k+n) where k is storing the first k smallest number and n is the rest of the number. For the Heap tree, a variable is used for storing the

## 1.3.2 Time Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| data | Vector<int><int> |  |  |
| Heap\_size | int | 0 |  |
| Parent(i) | int | Return (i-1)/2 | O(1) |
| Left(i) : Return left child | Int | Return 2\*I +1 | O(1) |
| Right(i) : Return right child | Int | Return 2\*I +2 | O(1) |
| Insert(k) | Bool | If(the heap is full)  Return false  I = heap\_size  Heap\_size ++  Data[i] = k  While(I is not 0 and the parent of I is less priority)  The swap I with the parent  Return true | O(log(n)) |
| Delete[i]: delete the index I of data | bool | If(Index is not available):  Return 0  Convert the index with the least priority  Loop through parent to bring this index value to the root  Extract root | O(log(n)) |
| extractRoot() | Int | Int root = data[0]  Swap the root with the last value  Converts the last value to the least priority  Hepify(0) | O(log(n) |
| Hepify(i) | void | Compare the current value to the next children and replace the current one with the higher priority. Replace the step until the current is in the correct position | O(logn) |

Amortized Analysis

Let use the Aggregate method for the Heap:

In the insertion, let call each time we insert an element, in the end, is 1. Each swap we have for each element with the parent is 1. Therefore, each element is added in a height is :

* Insertion = 1.
* Maximum swap = h = log2(n)

We have :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Height (h) | Number of Element (n) | Total of maximum operations  n\*(1+ h) | Total of the element | Average |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 2 | 4 | 3 | 1.333 |
| 2 | 4 | 12 | 7 | 1.714 |
| 3 | 8 | 32 | 15 | 2.133 |
| 4 | 16 | 80 | 24 | 3.333 |

Therefore, the insertion heap is

So, at each insertion, the heap has an average runtime at 1 <= Ai <log(n)

For deletion, let call each time replace the value at index I is 1. The swap in the index I to the root is 1. and hepify is cost log2(n) time. So each time of deleting is cost : 1 + 1 + log(n)

Therefore, the deletion heap is

Let use the Aggregate method for the kSmallest:

For generation, we need to insert a list N element to the kHeap and nHeap.

The first k insertion needs an average of k <=k\* Aki < k\*log(k) operation each.

From the next insertion, it needs to compare with the root of k, cost 1

And then it may lead to 2 cases:

When element e is less than root of k: Ai1 = 2log(k) + log(n)

We need to extract the root k: log(k)

We need to insert e: log(k)

We need to insert root to nHeap: log(n)

When e is greater than root of k, we only need to insert to nHeap: Ai2 = log(n)

For deletion, an index in kSmallest:

If i is less than k, the deletion will be Ad1 = 2\*log(k) + log(n)

We need to delete I in k = log (k)

We need to extract root of n = log(n)

We need to insert the root in k = log(k)

If i is larger than k, we only need to insert to nHeap: Ad2 = log(n)

By using the Aggregate method and a list example below we have:

Assume all log() is base 2.

As we know that when an element is inserted, we need 2\*log(k) + log(n) or log(n) for both deletion and insertion. So, every time we insert an element we prepay for the deletion as well.

Let have a random list: 8 1 9 2 0 5 7 3 4 6, and k = 5, n = 10

At the first 5 number, we need to insert 8 1 9 2 0, that we have 1 + 2 + 2 + 3 + 3 = 11 < k\*log(5) for insertion.

Then for 5 7 3 4 6, we need

5 is less than 9 (9 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(1) = 4

7 is less than 8 (8 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(2) = 5

3 is less than 7 (7 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(3) = 6

4 is less than 5 (5 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(4) = 6

6 is larger than 4 (4 is the root k), it cost log(n) = log(5) = 6

Total is 11 + 4+5+6+6+6 = 38 => A = 38/ 10 = 3.8 < log(k) + log(n) = log2(5) + log2(5) = 4.6

For deletion, the algorithm is similar to generation, the amount of insertion is as same as the amount of deletion (Ai1= Ad1 and Ai2 = Ad2)

So A = Ad = Ai =

# Question 2

## Problem Statement

A simple algorithm for maze generation is to start, apart from entry and exit points, with all walls present and randomly knock down walls until the entry and exit points are connected. Write a C++ program to implement this algorithm for an arbitrary-sized maze – test with a 50 by 88 rectangular maze.

## Algorithm Description

Using the disjoint set and Kruskal algorithm, we create a 2D array with each cell storing a pair of (w,h) coordinates. Assigning the entry and exit points. Run the Kruskal algorithm, every time the algorithms pick a random pair that was not connected to the same root parent, it will knock out a random wall around that cell.

Data structure:

Maze :

Pair<int,int> parent[w,h];

Pair<pair<int, int> , pair<int, int>> unblockedWall;

Pseudo-code:

Find : this function will recursively look for the parent most root. (find where the cell can go to another cell and return the ended cell)

|  |
| --- |
| Function find (int x, int y):  If (x,y) != parent[x,y]  Return find(parent[x,y])  Return parent[x,y] |

Union : This function is replacing 2 pairs to one parent. (checking if 2 cell can connect to a path, if not, we make it connect)

|  |
| --- |
| Function union (pair x, pair y):  Parent1 = find(x)  Parent2 = find(y)  If parent1 == parent2:  Return;  Let parent1 = parent2 |

MazeGeneration: This function will apply Kruskal Algorithm to create a map:

|  |
| --- |
| Function genrate (pair start, pair goal):  While(parent of start != parent of goal):  randomCell : pick a random cell  neighbor : find a random neighbor of this cell  if (this cell have different parent with the neighbor):  union(randomCell, neighbor)  unblockWall.add(the wall connect the cell with its neighbor) |

To print the maze, generate a maze template that does not have any path. Delete the unblocked wall.

## Complexity

## Space Complexity

Space for storing parent is O(w\*h) where w is the width and h is the height of the maze

Space for storing the unblocked wall is O(w\*h\*3) in the worst case since each cell can be opened for at most 3 walls.

## 1.3.2 Time Complexity

The solution is randomly generating the maze, it is become a non-polynomial problem. The runtime is depended in the random picked of the blocked wall list. Therefore, the Big-O in the best case can be O(w\*h+ r) when r is the random wall that was picked before.

# Question 3

## Problem Statement

Using C++ software obtained from the internet analyses and compare the performance of Red-Black

Trees and Van Emde Boas Trees using a large number of integers. This should be done for add, find,

delete and sequential access.

## Algorithm Description

For testing the 2 algorithms, I have downloaded the 2 programs in the Internet:

### Van Emde Boas Trees:

Link : <https://github.com/TISparta/Van-Emde-Boas-tree>

Authors: Check the license file.

Directory: Van-Emde-Boas-tree

My test file: Van-Emde-Boas-tree-master\test\_van.cpp

Test output: Van-Emde-Boas-tree-master\testoutput.txt

Run powershell (ps1) for generate a.exe: Van-Emde-Boas-tree-master\run\_testVan.ps1

Algorithm Test Explanation: This algorithm is running and storing the value present from the list as a BIT. It has functions for adding, erasing, looking up, successor, predecessor, min, and max.

### Red-Black Trees:

Link : <https://github.com/anandarao/Red-Black-Tree>

Authors: <https://github.com/anandarao>

Directory: Red-Black-Tree

My test file: Red-Black-Tree\testRB.cpp

Test output: Red-Black-Tree\testoutput.txt

Run powershell (ps1) for generate a.exe: Van-Emde-Boas-tree-master\testRB.ps1

Algorithm Test Explanation: This algorithm is using a Node data structure which store the value, the node color, the node parent and node left and right children. The given algorithms allow to test insertion, deletion, merge, access inorder and preorder.

## Comparison

### Insertion

|  |  |  |
| --- | --- | --- |
| Type | Time for 1000000 numbers | Time for 1 number |
| Van Emde Boas Trees | 154 milisecs | 0.000154 milisecs |
| Red-Black Tree | 443 milisecs | 0.000443 milisecs |

### Deletion

|  |  |  |
| --- | --- | --- |
| Type | Time for 1000000 numbers | Time for 1 number |
| Van Emde Boas Trees | 127 milisecs | 0.000127 milisecs |
| Red-Black Tree | 146 milisecs | 0.000146 milisecs |

### Look up

|  |  |  |
| --- | --- | --- |
| Type | Time for 1000000 numbers | Time for 1 number |
| Van Emde Boas Trees | 63 milisecs | 0.000063 milisecs |
| Red-Black Tree | N/A | N/A |

### Accession

|  |  |  |
| --- | --- | --- |
| Type | Time for 10000 numbers | Time for 1 number |
| Van Emde Boas Trees | 3 milisecs | 0.0003 milisecs |
| Red-Black Tree | 243 milisecs | 0.0243 milisecs |

## Conclusion

The Van Emde Boas Tree has a bigger advantage than the Red-Black Tree. The CPU runtime of all the methods of Van Emde Boas Tree is smaller than the Red-Black Tree. Moreover, the Red-Black Tree is not support for looking up the value of a certain node.

# Question 4

## Problem Statement

The object of the Kevin Bacon Game is to link a movie actor to Kevin Bacon via shared movie roles.

The minimum number of links is an actor’s Bacon number. For instance, Tom Hanks has a Bacon

number of 1; he was in Apollo 13 with Kevin Bacon. Sally Fields has a Bacon number of 2, because

she was in Forrest Gump with Tom Hanks, who was in Apollo 13 with Kevin Bacon. Almost all well-known actors have a Bacon number of 1 or 2. Given a list of actors, with roles, write a C++ program

that does the following:

(a) Finds an actor’s Bacon number.

(b) Finds the actor with the highest Bacon number.

(c) Finds the minimum number of links between two arbitrary actors

## Algorithm Description

Data structure: using adjacency list.

Using a map<string, set<string>> to store the key as the name of actor or movie, at each line of an actor, we get extend each character with the movie name and each movie name extend the actor.

Get author: return the author’s name from the line that separated by |

Get movie: return the movie’s name from the line that separated by |

Read file: read line by line and return a map (adjacency list) that has the keys are the author name or movies name.

Node : store an actor name, movie name, the bacon number, and the parent node.

Create Node: return a Node pointer

### Finds an actor’s Bacon number.

Using Breath-First Search, loop until the “Kevin Bacon (I)” node to the actor Node. Return the Node that have the actor’s name and Bacon number.

### Finds the actor with the highest Bacon number.

Using the Deep-First Search, loop until the queue is empty. Then, finding the Node have the biggest number of Bacon.

### Finds the minimum number of links between two arbitrary actors

Similar as the question a, but instead of finding from “Kevin Bacon (I)”, the function adds 2 actor names. It will look for the actor distance until the number links is found, or not found.

## Complexity

Let m is the number of movies.

Let a is the number of actors.

## Space Complexity

Adjacency graph: O(n\*m) is the worst case when each character is in m movies.

## 4.3.2 Time Complexity

Space complexity of all three problem has the worst case is O(n\*m).