# 3805ICT Advanced Algorithms
# Assignment Part Two

Ethan Leet
s5223103
ethan.leet@griffithuni.edu.au

May 22, 2022

# Contents

# 0 Introduction

## 0.1 Overview

This assignment involved writing c++ code to solve four Advanced Algorithm based problems. Test cases were then created to prove the validity of the algorithm used for each of the four problems. Each question is broken down into several sub sections throughout this report. The sub sections include; problem statement, algorithm description, complexity analysis, and experiments and results.

## 0.2 Compilation and Runtime

All programs included in this submission compile without errors nor warnings using clang++ and the c++ standard library 14. Each program contains a make file which will compile the program with all its required dependencies. There is also a make file in the parent folder which will make every program in this submission and place the binaries in the respective question folders. The following command can be executed to run the 'make' command:

```
make
```

If make is not available please check the respective questions makefile to extract the command used to compile individual programs with their respective binaries.

After successful compilation programs can then be executed with the commands described in the readme file attached to this submission. Note that in order to execute the following command you must be inside the question folder.

# 1 Maintain The List Of Smallest K

## 1.1 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 amortised analysis of your data structure.

## 1.2 Algorithm Description

This data structure was implemented as a c++ class which contains two multisets from the standard template library. The first multiset contains the $k$ smallest integers and the second multiset contains all $n + k$ integers.

A multiset was chosen as the structure due to the constant time lookup for the start and end elements of the list. This constant time lookup allows to easily check if the element in question belongs to the $k\_smallest$ integers or the remainder of the list. Insertion and deletion into the multisets can be done in worst case logarithmic time or amortized constant time as per the c++ documentation.

Non-trivial member functions are defined below:

1. **Constructor**
   This method constructs the structure from a text file. It will firstly save all elements of the text file to a temporary multiset. It then traverses this multiset and adds the first $k$ elements to $smallest$ and the remaining $n + k$ elements to $largest$.

**Algorithm 1** Constructor

```
initialise $temp$
$k \leftarrow first\ item\ in\ file$
while $file$ not $empty$ do
    $temp \leftarrow line\ in\ file$
    $file$ remove $line$
end while
$smallest \leftarrow temp[0]$ to $temp[k]$
$largest \leftarrow temp[0]$ to $temp.last$
```

2. **Insert** This method checks if the element to be added is larger than the first element present in $largest$. If it is, it simply adds the item to $largest$. If it is not, it will add the item to $smallest$. In this case, the largest element in $smallest$ will then be removed and placed into $largest$.

**Algorithm 2** Insert(element)

```
if $element \geq largest.first$ then
    $largest \leftarrow element$
else
    $smallest \leftarrow element$
    $largest \leftarrow smallest.last$
    remove $smallest.last$
end if
```

3. **Delete**
   This method firstly checks to see if the element to be deleted is present in the structure by invoking the $search$ method. If it does exist, it will then check if the element is larger than the first element in $largest$. If it is, it removes the element from $largest$. If it is not, the element is erased from $smallest$ and $largest$, then the first element in $largest$ is inserted into $smallest$.

**Algorithm 3** Delete(element)

```
if not $search(element)$ then
    return $error$
else if $element \geq largest.first$ then
    remove $element$ from $largest$
else
    remove $element$ from $smallest$
    remove $element$ from $largest$
    $smallest \leftarrow largest.first$
end if
```

## 1.3 Complexity

### 1.3.1 Space Complexity

When initialising the structure a multiset of size $n$ is created, where n is the size of file being passed $minus\ one$. This multiset is then used to create $smallest$ containing $k$ elements and $largest$ containing $n + k$ elements. This gives us a total complexity $O(n + k + n + k = 2n + 2k)$ and is of order $O(n + k)$.

The sequential method creates a multiset containing the whole structure, and is made up of the $smallest$ and $largest$ multisets. This gives a complexity of $k+n+k = n+k$ and is of order $O(n + k)$.

As such, the total worst case space complexity for this structure is $O(n + k)$ where $n + k$ is the number of elements present in the list $minus\ one$.

### 1.3.2 Time Complexity

To analyse the space complexity first consider the complexity of each method:

1. **Constructor**
   This method traverses the list passed as a command line argument and adds each item $n$ to a structure. It then traverses that structure and adds the first $k$ elements to $smallest$ and the remaining $n+k$ elements to $largest$. This method also uses the std::emplace library function which operates in logarithmic time. This gives a total complexity of $n + k + n + k + \log n = 2n + k + \log n$ and is of order $O(n)$ when $k \leq n$.

2. **Insert**
   This method checks the first element of $largest$ using the std::begin library function which operates in constant time. The method then uses std::emplace at most twice and std::erase at most once which both operate in logarithmic time giving a complexity of $1 + 2\log n + \log n = 1 + 3\log n$ which is of order $O(\log n)$.

3. **Search**
   This method used the std::find library function which operates in logarithmic time.

4. **Delete**
   This method first invokes the $search$ method. It then checks the first element of $largest$ using the std::begin library function which operates in constant time. The method then uses std::emplace at most once and std::erase at most twice which both operate in logarithmic time giving a complexity of $\log n + 2\log n = 3\log n$ which is of order $O(\log n)$.

5. **Return K Smallest**
   This method simply returns $smallest$ and operates in constant time.

6. **Sequential**
   This method uses the std::insert library function to create one multiselt containing $smallest$ and $largest$. This function operates in logarithmic time.

As can be seen, the worst case complexity for the structure is bounded by the constructor and is of order $O(n)$. Considering only the methods of the structure, the worst case complexity is logarithmic, and is of order $O(\log n)$.

## 1.4 Amortised Cost Analysis

A tighter complexity bound for this program can be achieved by considering the amortised cost in the worst case scenario:

When an item is inserted into $smallest$, three actions take place; firstly insertion of the element into $smallest$, secondly the insertion of the largest element in $smallest$ to $largest$, and finally the deletion of the largest element in $smallest$. When an item is deleted from $smallest$, three actions take place; firstly the deletion of the element, secondly the insertion of the smallest element from $largest$ to $smallest$, and finally the deletion of the smallest element from $largest$.

As can be seen, if we consider an initialised list that inserts every item into $smallest$ and then deletes every item $n$ present in the whole structure in ascending order, a total of $three$ deletions and $three$ insertions will occur for each of the $n$ items. Consider the following:

$$\underset{\substack{\text{Amortised}\\\text{Cost}}}{AM} = \underset{\substack{\text{Actual}\\\text{Cost}}}{AC} + \underset{\substack{\text{Change in}\\\text{Potential}\\\text{Energy}}}{\triangle\phi}$$

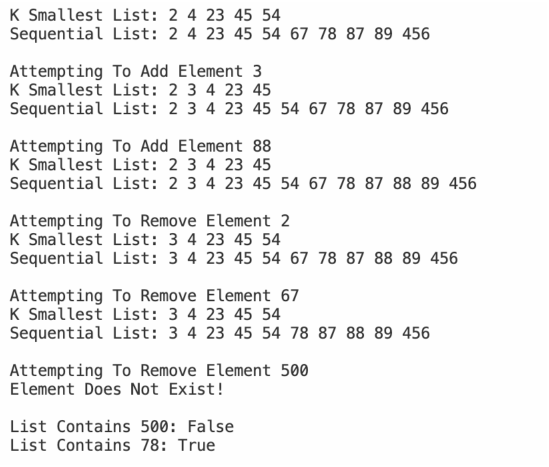
| | Action | AC | $\phi$ | $\triangle\phi$ | AM |
|---|---|---|---|---|---|
| **Insert** | Insert | 1 | 1 | 1 | 2 |
| | Insert | 1 | 2 | 1 | 2 |
| | Delete | 1 | 1 | -1 | 0 |
| **Delete** | Delete | 1 | 0 | -1 | 0 |
| | Insert | 1 | 1 | 1 | 2 |
| | Delete | 1 | 0 | -1 | 0 |

By charging $two$ for every insert there is enough potential energy $\phi$ to perform the required deletions without any cost. This results in an amortised cost of $two$ for every $n$ insert and subsequent delete.

As derived in section 1.3, the insert and delete methods operate in logarithmic time and both have an unbounded complexity of $3\log n$. When compared with the amortised costs analysis for a worst case scenario described above, a tighter bound of $2\log n$ can be expected as $2\log n < 2(3\log n)\forall n$ and gives an amortised cost of order $O(\log n)$.

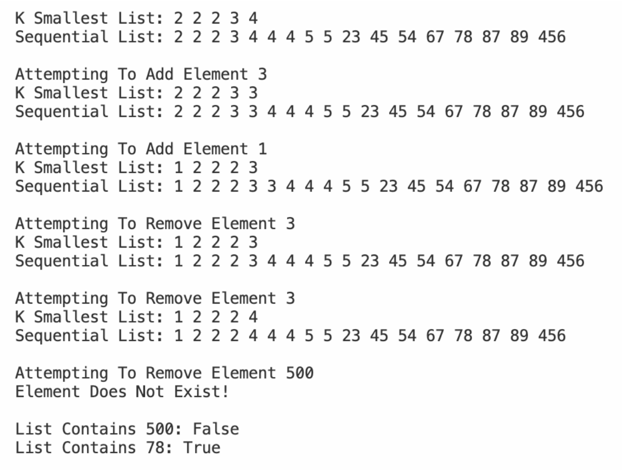
## 1.5 Experiments and Results

To test the validity of this structure a simple list of ten elements was created with $k = 5$. All methods were tested using a range of different test lists of this size. The results for one test case can be seen below:



```
K Smallest List: 2 4 23 45 54
Sequential List: 2 4 23 45 54 67 78 87 89 456

Attempting To Add Element 3
K Smallest List: 2 3 4 23 45
Sequential List: 2 3 4 23 45 54 67 78 87 89 456

Attempting To Add Element 88
K Smallest List: 2 3 4 23 45
Sequential List: 2 3 4 23 45 54 67 78 87 88 89 456

Attempting To Remove Element 2
K Smallest List: 3 4 23 45 54
Sequential List: 3 4 23 45 54 67 78 87 88 89 456

Attempting To Remove Element 67
K Smallest List: 3 4 23 45 54
Sequential List: 3 4 23 45 54 78 87 88 89 456

Attempting To Remove Element 500
Element Does Not Exist!

List Contains 500: False
List Contains 78: True
```

It was also noticed that lists which contained duplicate elements behaved differently using functions from the standard template library. Edge cases were created involving duplicate elements and duplicate elements that were present in both the $smallest$ and $largest$ structures. The results of these tests can be seen below:



```
K Smallest List: 2 2 2 3 4
Sequential List: 2 2 2 3 4 4 4 5 5 23 45 54 67 78 87 89 456

Attempting To Add Element 3
K Smallest List: 2 2 2 3 3
Sequential List: 2 2 2 3 3 4 4 4 5 5 23 45 54 67 78 87 89 456

Attempting To Add Element 1
K Smallest List: 1 2 2 2 3
Sequential List: 1 2 2 2 3 3 4 4 4 5 5 23 45 54 67 78 87 89 456

Attempting To Remove Element 3
K Smallest List: 1 2 2 2 3
Sequential List: 1 2 2 2 3 4 4 4 5 5 23 45 54 67 78 87 89 456

Attempting To Remove Element 3
K Smallest List: 1 2 2 2 4
Sequential List: 1 2 2 2 4 4 4 5 5 23 45 54 67 78 87 89 456

Attempting To Remove Element 500
Element Does Not Exist!

List Contains 500: False
List Contains 78: True
```

A c++ file (included in this submission) was created which generates a file of $10,000$ long integers with $k = 100$. The large lists were then inspected and tested with this structure to further prove its validity. In all test cases used the structure returned the right result and was observed to obey the complexities listed above.

# 2 Maze Generation

## 2.1 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.

## 2.2 Algorithm Description

To generate a maze for this problem two main data structures were created, Disjoint Sets and a Maze.

The disjoint sets structure has two main methods, find and union. The find method uses path compression (flattens the tree) to search for a passed node. The union method will unionise two nodes if they are not already in the same set based upon their rank(attach smaller depth tree under larger depth tree). This structure operates using two main members, parent and depth. The parent is a list of nodes and depth is their rank. The depth member is used for time optimisations as described below.

**Algorithm 4** Find(node)

```
if $node$ not $parent\ node$ then
    $parent\ node \leftarrow find(parent\ node)$
end if
return $parent\ node$
```

**Algorithm 5** Union(node 1, node 2)

```
if $node\ 1$ same set as $node\ 2$ then
    return
end if
if $node\ 1\ depth$ > $node\ 2\ depth$ then
    $node\ 2\ parent \leftarrow node\ 1$
    $node\ 2\ depth \leftarrow depth\ depth\ node\ 1$
else
    $node\ 1\ parent \leftarrow node\ 2$
    $node\ 1\ depth \leftarrow depth\ depth\ node\ 2$
end if
```

The Maze structure contains methods pertaining to; the creation of a maze, destroying random walls until a solution is found, and displaying the final maze. It also contains an internal private member which is a structure to hold each square in the maze. Each square has a 1D positional value and a flag for each of its four walls. The flag is $zero$ if no wall exists, $one$ if a wall can be knocked down, or $two$ if a wall exists but it is indestructible $(border)$.

**Algorithm 6** Create(size)

```
for $i \leftarrow 0$ to size do
    initialise $Square$ at $i$
    add $walls$ to $Square$ at $i$
end for
if $Square$ is $start$ then
    $Square$ at $start \leftarrow left\ wall$ destroyed
end if
if $Square$ is $end$ then
    $Square$ at $end \leftarrow right\ wall$ destroyed
end if
```

**Algorithm 7** Destroy

```
initialise $disjoint\ set\ for\ every\ node$
while $start\ node\ set \neq end\ node\ set$ do
    choose $random\ square$
    choose $random\ wall$
    if $wall\ can\ be\ destroyed$ then
        $wall \leftarrow$ destroyed
        update $neighbour\ wall$
        $node\ and\ neighbour \leftarrow$ unionised
    end if
end while
```

**Algorithm 8** Display

```
for $i \leftarrow 0$ to maze.size do
    if $square\ at\ i$ contains $wall$ then
        draw $wall$
    end if
end for
```

## 2.3 Complexity

### 2.3.1 Space Complexity

To analyse space complexity first consider the complexities of each structure and their methods:

1. Disjoint Sets This structure holds two vectors which consists of $n$ parent nodes and $n$ depths. Each method within the structure operates on these vectors. As such, a space complexity of $O(n^2)$ can be expected for this structure, where $n$ is the amount of nodes present in the graph.

2. Maze This structure holds a vector of $n$ squares which represent every cell in the maze. Each square also contains $five$ integer data types. This gives a total complexity of $5n$ or order $O(n)$ for the members of the structure.

   The destroy method creates a disjoint set for every node present in the graph, as seen above, $n$ nodes equates in a space complexity of $O(n^2)$. As such, a space complexity of $O(n^2)$ can be expected for this method and the program as a whole.

### 2.3.2 Time Complexity

To analyse time complexity first consider the complexities of each structure and their methods:

1. Disjoint Sets
   By compromising space and using a rank vector, which always adds the smaller tree to the root of the larger tree, a $logarithmic$ time complexity can be expected compared to the $linear$ complexity without using union by rank.

   A further optimisation has also been implemented, path compression. When find is called, the root of the element is returned. Path compression flattens the tree when find is called for some element $x$ by making the found root a parent of $x$. This negates any further traversals of intermediate nodes. Overall, this optimisation does not change the worst case $logarithmic$ complexity of the method, it does however provide a tighter bound of amortised constant.

2. Maze
   The $create$ method for this structure traverses the one dimensional size of the maze $(L \times W)$. As such, a complexity of $O(L \times W)$ can be expected for this method.

   The $delete$ method randomly removes walls until a path from start to end is found. Every wall that can be removed also shares a wall with its neighbouring square. As such, the total walls that can be knocked down in any maze is $(R-1) \times (2 \times (C-1)) + R - 1 + C - 1$ where $R$ and $C$ are the amount of rows and columns in the maze respectively.

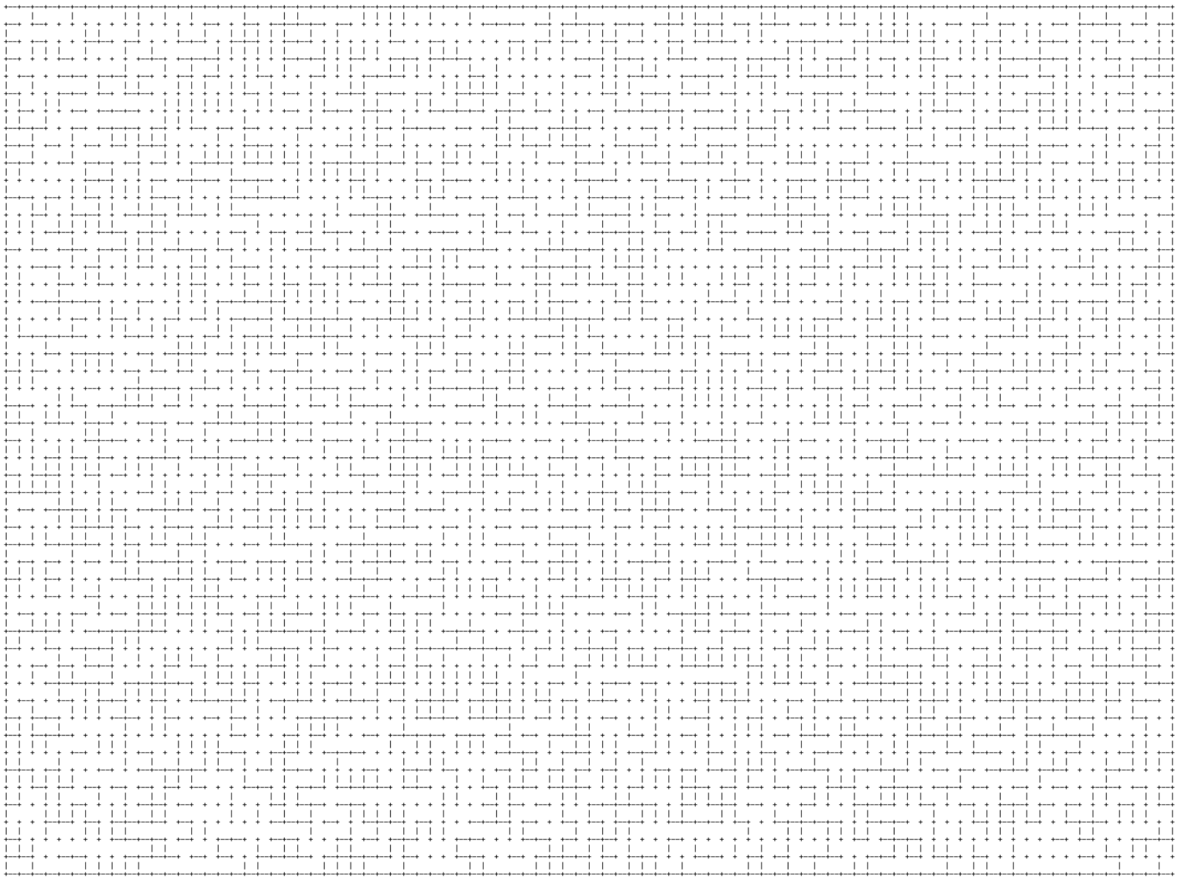
   Through theoretical experimentation it was noted that the total number of walls that could be knocked down before a solution was found was $one$ less than the total amount of walls that can be knocked down. As such, a complexity of $(R-1) \times (2 \times (C-1)) + R - 1 + C - 1$ can be expected which is of order $O(R \times C)$.

## 2.4 Experiments and Results

A variety of different maze sizes were tested to confirm the validity of this algorithm. Edge testing and test cases were also carried out to ensure the algorithm was valid for any non trivial sized maze. Maze size testing was also carried out to confirm the total number of walls that could be knocked down (described above).

It can be noted that while this algorithm generates a maze for any non trivial size passed, the maze generated is not optimal. Further experimentation would need to be carried out relating to disjoint sets to devise an algorithm which generates an optimal maze where the amount of walls removed to form a path is minimal.

The below figure stipulates the output of a $50 \times 88$ maze as requested in the problem statement.



# 3 Red-Black Trees vs. Van Emde Boas Trees

## 3.1 Problem Statement

Using C++ software obtained from the internet analyse 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.

## 3.2 Algorithm Description

A red-black tree [1] is a variation of a self-balancing binary search tree where each node stores an extra bit that represents colour, red or black. The colouring arrangement ensures that no path is more than twice as long as any other tree, resulting in a close to good balanced tree. The colouring also ensure quick operations as described below. The red-black tree chosen for this research was the std::set method from the c++ standard library.

The Van Emde Boas (vEB) tree [2] which implements an associative array of m-bit integer keys. Unlike other structures, the vEB tree operates on the maximum number of elements that can be represented in a tree rather than the amount of elements that are actually present. This is denoted as $M = 2^m$ and commonly referred to as the universe space size. The vEB tree chosen for this research was sourced [3] on the internet.

## 3.3 Complexity

### 3.3.1 Space Complexity

As derived during lecture materials and by [1] the space complexity for a red-black tree is $O(n)$ where $n$ is the total number of nodes. As can be seen by [2] the space complexity for a vEB tree is $O(M)$ where $M$ is the total number of elements which can be represented in the universe $2^m$.

### 3.3.2 Time Complexity

The worst case complexity for search, find and delete on a red-black tree is $O(\log n)$. The search function simply traverses the tree similarly to a binary search tree and has an amortised complexity of $O(\log n)$. The insert and delete have an amortised cost of $O(1)$, the proof can for this analysis is derived by [1].

The time complexities for find, insert and delete on a vEB tree are derived by [2] and are expected to obey an amortised complexity of $O(\log \log M)$.

[1] https://en.cppreference.com/w/cpp/container/set

[2] https://en.wikipedia.org/wiki/Van_Emde_Boas_tree

[3] https://github.com/dragoun/veb-tree

## 3.4 Experiments and Results

It can be seen above that for insertion and deletion the red-black tree theoretically out performs the vEB tree whereas the vEB tree theoretically out performs the red-black tree for the find operation. The goal of the below test scenarios is to practically assess this theoretical hypothesis. It is also noted that the complexity of a vEB tree is bounded by the universe size. In order to get a tight bound, an accurate universe size is needed, which can be difficult in size increasing or unknown problems.

As such, the **null hypothesis** is *red-black trees will have an overall actual performance orders better than a vEB tree for all operations other than find for an appropriately sized universe*.

In order to test the null hypothesis five tests sizes were chosen; 10,000, 100,000, 1,000,000, 10,000,000, and 100,000,000. These sizes stipulate the amount of numbers that will be operated on within the two structures. Within each test size eight experiments were carried out across four different types of orderings for the datasets totalling 160 experiments conducted on each structure.

The types of orderings tested are as follows:

1. Increasing order
2. Decreasing order
3. Random ordering and,
4. Alternating ordering (first, last, second, second last)

The experiments conducted within each order:

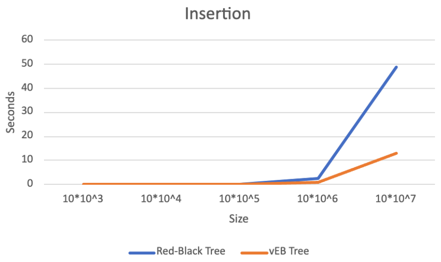
1. Insert all elements
2. Sequential access to all elements
3. Find first element
4. Find last element
5. Find middle element
6. Find random element
7. Add 10% more random elements and,
8. Empty the structure

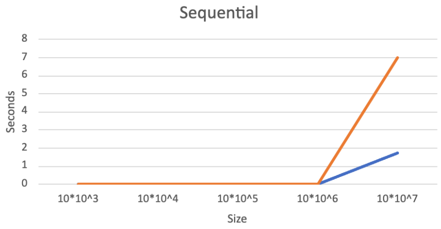
The results for a size of 100,000,000 are measured in seconds and can be seen below:

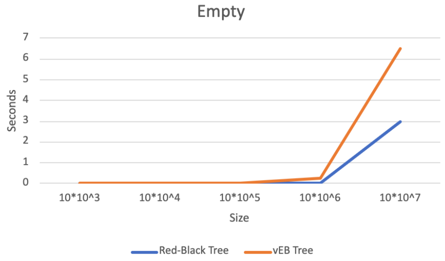
| Order | Operation | RBT | vEB |
|---|---|---|---|
| Increasing | Insert | 19 | 3 |
| | Sequential | 0 | 6 |
| | Find First | 0 | 0 |
| | Find Last | 0 | 0 |
| | Find Middle | 0 | 0 |
| | Find Random | 0 | 0 |
| | Add 10% | 9 | 0 |
| | Empty | 2 | 3 |
| Decreasing | Insert | 18 | 3 |
| | Sequential | 0 | 6 |
| | Find First | 0 | 0 |
| | Find Last | 0 | 0 |
| | Find Middle | 0 | 0 |
| | Find Random | 0 | 0 |
| | Add 10% | 0 | 0 |
| | Empty | 1 | 3 |
| Random | Insert | 125 | 11 |
| | Sequential | 7 | 8 |
| | Find First | 0 | 0 |
| | Find Last | 0 | 0 |
| | Find Middle | 0 | 0 |
| | Find Random | 0 | 0 |
| | Add 10% | 7 | 0 |
| | Empty | 7 | 2 |
| Alternating | Insert | 31 | 35 |
| | Sequential | 0 | 8 |
| | Find First | 0 | 0 |
| | Find Last | 0 | 0 |
| | Find Middle | 0 | 0 |
| | Find Random | 0 | 0 |
| | Add 10% | 8 | 0 |
| | Empty | 2 | 18 |

Upon initial inspection it appeared that the find function for either structure was not returning correct results. Upon further inspection, each call was finding a result in a time that could not be accurately measured by the CPU the tests were ran on.

Each of the four orderings were then averaged to graph how individual method times change versus size:







It was observed that for sizes less than 10,000,000 the structures are comparable. When the size starts to further increase the red-black tree outperforms the vEB tree for the sequential search and empty methods, whereas the vEB tree outperforms the red-black tree for insertion.

As the vEB tree outperforms the red-black tree for insertion, this research concludes that there is enough empirical evidence in which a failure to accept the null hypothesis is decided. Further experimentation would need to be conducted into how the insertion method changes over time for both structures as their sizes increase greater than 10,000. Further research should also be conducted into the find methods to assess, either theoretically or practically, how the two structures compare.

When considering to use either a red-black tree or vEB tree, a red-black tree should be chosen as the preferred structure when the speed of internal methods are of importance. If only a single structure is to be set up without internal methods then a vEB tree structure should be chosen.

# 4 Kevin Bacon Game

## 4.1 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.

## 4.2 Algorithm Description

It was noticed that this problem is a shortest-path graph problem. As such, a structure was created to hold the node and edge details for each actor and movie within a particular dataset. The structure holds two std::unordered_maps, one for movies and actors. The movies structure stores each movie indexed by a list of actors who played in that movie whereas the actors structure stores each actor indexed by a list of movies they played in. The overall structure also contains an internal structure which holds node information used in the search algorithms described below.

The first algorithm used for this problem is a Breadth-First Search (BFS) algorithm. This algorithm will find the shortest path between two actors and can be described below.

**Algorithm 9** Breadth First Search(source, target)

```
$visited \leftarrow source$
$node \leftarrow source$
$queue \leftarrow node$
while $queue$ not $empty$ do
    if $node$ is $actor$ then
        traverse $actors\ movies$
        if $actors[movie]$ not $in\ visited$ then
            $node \leftarrow path\ to\ movie$
            $visited \leftarrow movie$
            $queue \leftarrow node$
        end if
    else
        traverse $movies\ actors$
        if $movies[actor]$ not $in\ visited$ then
            $node \leftarrow path\ to\ movie$
            $visited \leftarrow movie$
            $queue \leftarrow node$
            if $goal$ then
                return
            end if
        end if
    end if
end while
```

The second is an exhaustive Breadth-First Search which will return the longest bacon number out of all actors present in the dataset.

**Algorithm 10** Breadth First Search(source, target)

```
$visited \leftarrow source$
$node \leftarrow source$
$queue \leftarrow node$
while $queue$ not $empty$ do
    if $node$ is $actor$ then
        traverse $actors\ movies$
        if $actors[movie]$ not $in\ visited$ then
            $node \leftarrow path\ to\ movie$
            $visited \leftarrow movie$
            $queue \leftarrow node$
        end if
    else
        traverse $movies\ actors$
        if $movies[actor]$ not $in\ visited$ then
            $node \leftarrow path\ to\ movie$
            $visited \leftarrow movie$
            $queue \leftarrow node$
        end if
    end if
end while
return $node.path.length$
```

The internal structure used by the main structure holds actor information and their current path in the graph sequence. This structure is used by the above search algorithms, when a goal state is found the length of the path size from the node (actor) to the target is returned.

## 4.3 Complexity

### 4.3.1 Space Complexity

Due to the nature of how the graph information is stored twice in the structure, via two maps. The total space complexity for this program is bounded by the number of elements $n$ present in the dataset and is of order $O(n^2)$.

### 4.3.2 Time Complexity

The constructor method simply reads the dataset passed, as such, this method has a linear complexity.

The two BFS algorithms operate in the same manner and are of order $O(V + E)$ where $V$ and $E$ are the number of vertices and edges present respectively. The second BFS algorithm is exhaustive meaning all vertices and edges are traversed. As such, a tight bound of $\Theta(V + E)$ can be expected for this structure.

## 4.4 Experiments and Results

Upon inspection it was noticed that Kevin Bacon is not present in the third dataset. As such, the first two datasets were used to test part a and part b whereas the third dataset was used to test part c.

Numerous actor combinations were tested to confirm the validity of this structure. For datasets one and two the longest bacon number actor was first found. This actor was then used for part a to confirm that their bacon number was in fact the longest that was reported, confirming that both Breadth-First Search algorithms work correctly.

A sample result from standard testing can be seen below:

1. Bacon 1
   Bacon1.txt took 2ms to load
   Dinah Shore has a bacon number of 3 and took 0ms
   Christopher Nolan has the longest Bacon number of 5 and took 2ms

2. Bacon 2
   Bacon2.txt took 9ms to load
   Ernest Abuba has a bacon number of 4 and took 5ms
   Haruko Togo has the longest Bacon number of 6 and took 9ms

3. Bacon 3
   Bacon3.txt took 511ms to load
   Grandon Rhodes has a link of 2 with John Vosburgh and took 40ms