COMP1201 Assignment 2

Q1:

What we are given:

- Hash function with separate chaining.
- Takes an integer **(n < 100 000)**.
- Calculates the value as **$(2d_1 + 3d_2 + 5d_3 + 7d_4 + 11d_5)$ % 47**, where d_1 is the most significant digit and d_5 is the least significant digit.
- We enter **2000** numbers.

What we need to prove:

- Prove that there exists a number **x (0 <= x <= 46)** for which we can find **at least 43** numbers among the **given 2000**, whose **hash value** is exactly **x**.

Proof:

First, when a hash table has separate chaining, it mean that we build a singly-linked list at each table entry. If the keys for two elements are the same, then both of the elements are added to the corresponding singly-linked list.

The given problem can be easily coded using Java:

```java
//Create the new HashMap, which will be implemented using separate chaining.
HashMap<Integer, ArrayList<Integer>> myMap = new HashMap<>();
Random rand = new Random();

int key = 0;
int value = 0;

//Add 2000 numbers.
for (int i = 0; i < 2000; i++) {
    //Generate the 2000 random numbers.
    value = rand.nextInt(100000);

    //Isolate the different digits, in order to calculate the hashing value.
    int firstDigit = (value / 10000);
    int secondDigit = (value / 1000) % 10;
    int thirdDigit = (value / 100) % 10;
    int fourthDigit = (value / 10) % 10;
    int fifthDigit = (value % 10);

    //Add the key and value to the map.
    key = (2 * firstDigit +
            3 * secondDigit +
            5 * thirdDigit +
            7 * fourthDigit +
            11 * fifthDigit) % 47;

    //Lambda expression to add the integer to its specified ArrayList.
    myMap.computeIfAbsent(key, k -> new ArrayList<>()).add(value);
}

//Iterate through the map to see the values.
int i = 1;
Integer lastKey = 0;
ArrayList<Integer> numberOfValues= new ArrayList<>();

for (java.util.Map.Entry<Integer, ArrayList<Integer>> entry : myMap.entrySet()) {
    Integer myKey = entry.getKey();
    ArrayList<Integer> myValues = entry.getValue();
    for (Integer singleValue : myValues) {
        //Print out everything.
        //The counter is there so we can easily see the number of elements under
each key.
        if(lastKey == myKey) {
            //System.out.println("key : " + myKey + " value : " + singleValue + "
number : " + i);
            i++;
        }
        else {
            i = 1;
            //System.out.println("key : " + myKey + " value : " + singleValue + "
number : " + i);
            lastKey = myKey;
            i++;
        }
    }
    //System.out.println("Key: " + myKey + " <-> " + "Number of values: " + i);
    numberOfValues.add(i);
}

//Display the amount of values under each key, sorted, so we can prove that
//that there exists a number x between 0 and 46 for which we can find at least 43
//numbers among the given 2000, whose hash value is exactly x.
Collections.sort(numberOfValues, Collections.reverseOrder());
for(Integer number : numberOfValues) {
    System.out.println(number);
}
```

If we run this code, we can see that there is at least one number, **x**, between 0 and 46, for which there are at least 43 numbers, from the 2000, whose hash values is **x**.

Now, if we try to prove this mathematically, we can use the *pigeonhole principle*.

First, given the *hash code*, and the given *maximum value* we can have, the *values* of the *keys* may vary from 0 (included) to 46 (included). Since we input 2000 numbers (pigeons), and we have 47 key slots (pigeonholes), we see that there will be at least one key (pigeonhole) with at a minimum of 43 values (pigeons) under it. We can see that by dividing the pigeons by the pigeonholes (2000 / 47), and we get 42,5531914893617. When we the ceiling function (as per pigeonhole principle), we can see that ($\lceil 42{,}5531914893617 \rceil$ = 43) there is a pigeonhole with at least 43 pigeons in it.

Q2:

What we need to do:

- Design a data type, that supports the following operations:
  - Insert an element (**O(log(n))** time complexity).
  - Delete the biggest element (**O(log(n))** time complexity).
  - Delete the smallest element (**O(log(n))** time complexity).
  - Find the biggest element (**O(1))** time complexity).
  - Find the smallest element (**O(1))** time complexity).
- Design the data type using two heaps.

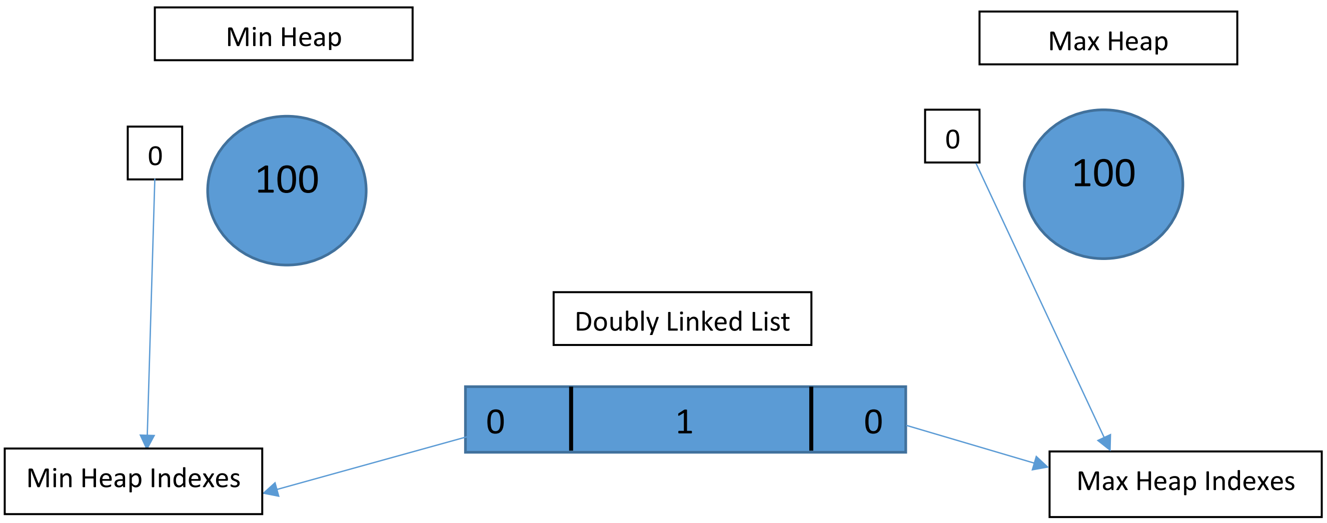
Creating the data type:

Before we start adding elements to any data types, we create two heaps. One is a **min heap** (smallest element is the root of the heap), and the other is a **max heap** (biggest element is the root of the heap). Because we want to keep the time complexity of deleting O(log(n)), we need to create another data structure to hold all our elements, so we can easily delete an element from both our heaps simultaneously. So we create a **doubly linked list**, which will contain all input items and indexes of the corresponding min and max heap nodes. We will use a doubly linked list as its insertion and deletion time complexity are O(1).

Representation of idea:

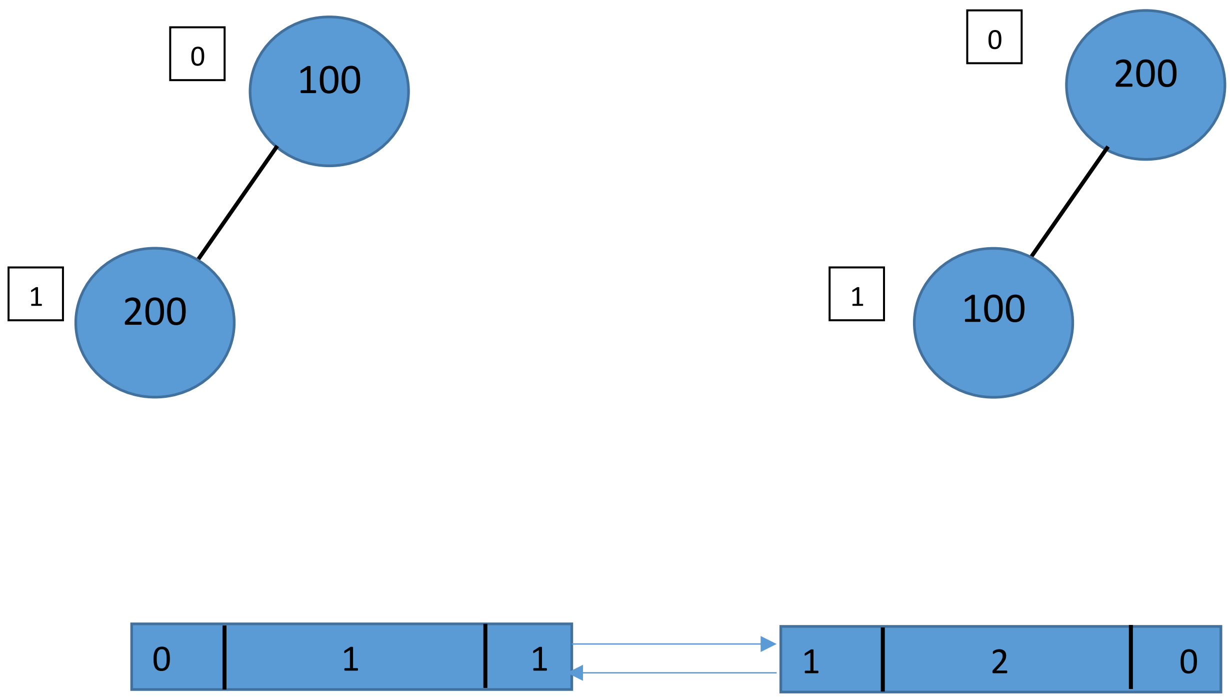
Insert element "**1**".

- First we encode the inserted value. For the sake of simplicity, we just multiply the value by 100. (So the value that will be stored in the heaps will be **100**)



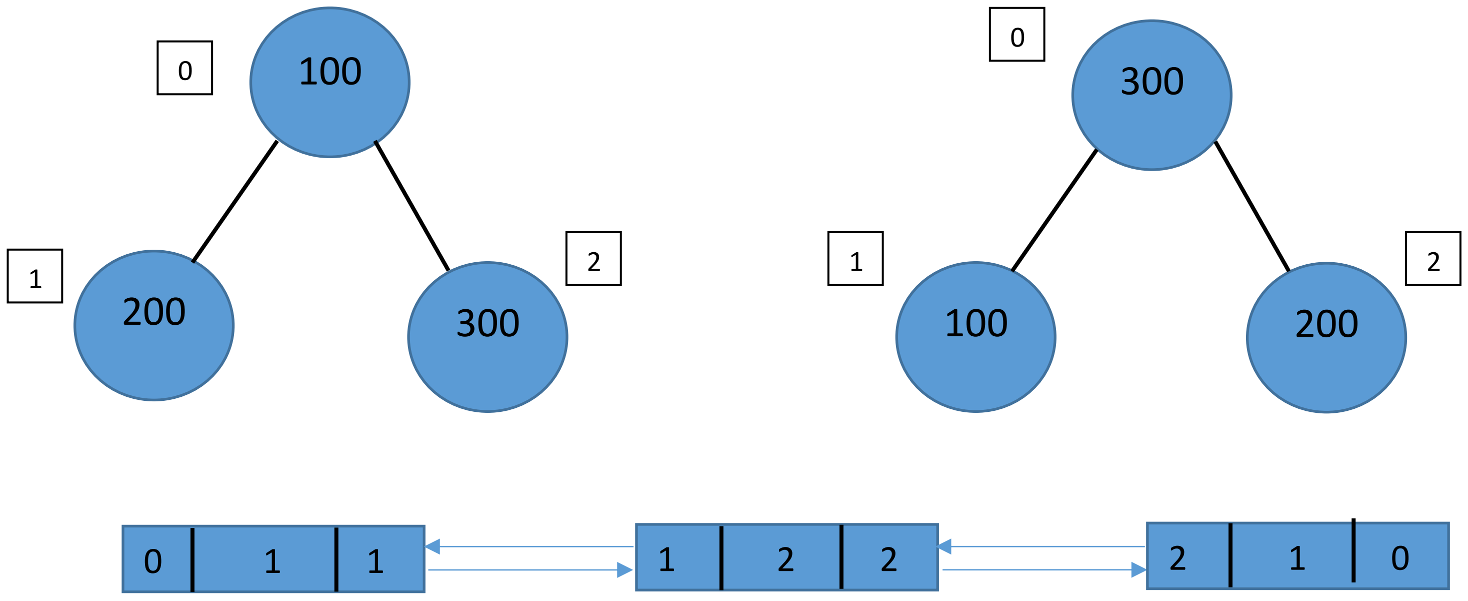
Insert element “**2**”.

- Again we encode it.



Next Insert

Insert element “**3**”.



How it works:

Inserting an element: We add the element to the start of the doubly linked list **(O(1))**. Then we insert the element to the **min** and **max heaps (Olog(n))**. Thus making the overall complexity be **O(log(n))**.

Pseudocode for inserting an element:

```
//Inserts an element into both heaps and the linked list
void insert(int key) {
    Node node = new Node();
    node.data = key;
    node.minHeapIndex = minHeap.insert(node);
    node.maxHeapIndex = maxHeap.insert(node);
    list.insertAtBeginning(node);
}
```

```
//Perculates the element through the heap.
//Helper method
int fixUpwards(int i) {
    while (i != 0 && arr[i].data > arr[parent(i)].data) {
        swap(arr, i, parent(i));
        i = parent(i);
    }

    return i;
}
```

```
//Inserting an element into the min heap.
int insert(Node node) {
    if(size == capacity) {
        System.out.println("Heap overflow");
        return -1;
    }

    arr[size] = node;
    arr[size].minHeapIndex = size;
    size++;
    return fixUpwards(size-1);
}
```

```
//Inserting an element into the max heap.
int insert(Node node) {
    if (size == capacity) {
        System.out.println("Heap overflow");
        return -1;
    }

    arr[size] = node;
    arr[size].maxHeapIndex = size;
    size++;
    return fixUpwards(size - 1);
}
```

```
//Inserts the element at the begging of the linked list.
void insertAtBeginning(Node node) {
    node.prev = null;
    node.next = head;

    if (head != null)
        head.prev = node;

    head = node;
}
int findMax() {
    return maxHeap.getMax().data;
}
```

Finding the smallest element: We look at the root of the min heap. This is a O (1) operation.

```
//Finds the smallest element.
int findMin() {
    return minHeap.getMin().data;
}
```

Deleting the biggest element: We get the biggest element using the findMax method. Then we get its address to find the node in the doubly linked list. From there we get the position of the element in the max heap. We remove the element from the three places (min heap, max heap and the linked list). Deleting the element from the doubly linked list has a O (1) time complexity. Removing the element from the heaps has a time complexity of O (log (n)). Which means that we achieve the needed overall time complexity.

```
//Deletes the biggest element.
int deleteMax() {
    Node node = maxHeap.getMax();

    int result = node.data;

    int minIndex = node.minHeapIndex;
    list.deleteNode(node);
    //extractMax() is a helper method that returns the root of the max heap.
    maxHeap.extractMax();
    minHeap.delete(minIndex);

    return result;
}
```

Deleting the smallest element: We get the smallest element using the findMin method. Then we get its address to find the node in the doubly linked list. From there we get the position of the element in the min heap. We remove the element from the three places (min heap, max heap and the linked list). Deleting the element from the doubly linked list has a O (1) time complexity. Removing the element from the heaps has a time complexity of O (log (n)). Which means that we achieve the needed overall time complexity.

```
//Deletes the smallest element
int deleteMin() {
    Node node = minHeap.getMin();

    int result = node.data;

    int maxIndex = node.maxHeapIndex;
    //extractMin() is a helper method that returns the root of the min heap.
    minHeap.extractMin();
    maxHeap.delete(maxIndex);
    list.deleteNode(node);

    return result;
}
```

Q3:

(a)

What we need to do:

- Implement **Quick-Union** in Java.
- Create a graph (A) with **256** vertices (numbered **0** to **255**).
- Connect vertices **A[i]** with **A[i+1]** for:
  - **i += 2**
  - **0 <= i <= 254**
- Connect vertices **A[j]** with **A[j+3]** for:
  - **j += 3**
  - **0 <= j <= 200**

What we need to find:

- How many connected components are there is the resulting graph (A)?
- What is the result of Find(19)?
- What is the result of Find(112)?

Solution:

```java
public class QuickUnionUF {

    private int[] id;

    public QuickUnionUF(int N) {
        id = new int[N];

        //Populate the graph.
        for (int i = 0; i < N; i++) id[i] = i;
    }

    private int root(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    //Returns true iff p and q have the same root.
    public boolean isConnected(int p, int q) {
        return root(p) == root(q);
    }

    //Returns the id of p’s root.
    public int find(int p) {
        return root(p);
    }

    //Add p’s root as a direct child to q’s root.
    public void union(int p, int q) {
        int i = root(p);
        int j = root(q);
        id[i] = j;
    }

    public static void main(String[] args) {
        //Create graph A with 256 vertices numbered from 0 to 255.
        QuickUnionUF graphA = new QuickUnionUF(256);

        int firstElement = 0;
        int secondElement = 0;
        int unionCounter = 0;

        //Connect vertices A[i] with A[i+1].
        for (int i = 0; i <= 254; i += 2) {
            firstElement = graphA.find(i);
            secondElement = graphA.find(i + 1);
            graphA.union(firstElement, secondElement);
            unionCounter++;
        }

        //Connect vertices A[j] with A[j+3].
        for (int j = 0; j <= 200; j += 3) {
            firstElement = graphA.find(j);
            secondElement = graphA.find(j + 3);
            graphA.union(firstElement, secondElement);
            unionCounter++;
        }

        //Check how many components are there is the resulting graph.
        //The rule is that each union command reduces by 1 the number of
components.
        //So to check how many components we have, we need to subtract the number
of unions we have made from the total number of starting components.
        System.out.println(256 - unionCounter);

        System.out.println(graphA.find(19));
        System.out.println(graphA.find(112));
    }
}
```

The code above provides implementation for Quick-Union in Java. After running it we can answer the questions:

- The number of components in the resulting graph is equal to the number of starting components **(256) minus** the number of union operations we have made **(195)**. Which means there are **61** components left after the operations.
- The root of node 19 is **201**.
- The root of node 112 is **113**.

(b)

What we need to do:

- Implement **Weighted Quick-Union** in Java.
- Create a graph (A) with **256** vertices (numbered **0** to **255**).

- Connect vertices **A[i]** with **A[i+1]** for:
    - **i += 2**
    - **0 <= i <= 254**
- Connect vertices **A[j]** with **A[j+3]** for:
    - **j += 3**
    - **0 <= j <= 200**

What we need to find:

- What is the result of Find(19)?
- What is the result of Find(112)?

Solution:

```
public class WeightedQuickUnionUF {
    private int[] id;
    //Number of elements in subtree.
    private int[] size;

    public WeightedQuickUnionUF(int N) {
        id = new int[N];
        size = new int[N];
        //Populate the graph.
        for (int i = 0; i < N; i++) {
            id[i] = i;
            //At the start, none of the elements have a subtree.
            size[i] = 1;
        }
    }

    private int root(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    public boolean isConnected(int p, int q) {
        return root(p) == root(q);
    }

    public int find(int p) {
        return root(p);
    }

    //We modify the union to:
    //merge the smaller tree into the larger tree, and
    //keep the size[] array updated.
    public void union(int p, int q) {
        int i = find(p);
        int j = find(q);
        if (i == j) return;

        if (size[i] < size[j]) {
            id[i] = j;
            size[j] += size[i];
```

```java
        } else {
            id[j] = i;
            size[i] += size[j];
        }
    }

    public static void main(String[] args) {
        //Create graph A with 256 vertices numbered from 0 to 255.
        WeightedQuickUnionUF graphA = new WeightedQuickUnionUF (256);

        int firstElement = 0;
        int secondElement = 0;

        //Connect vertices A[i] with A[i+1].
        for (int i = 0; i <= 254; i += 2) {
            firstElement = graphA.find(i);
            secondElement = graphA.find(i + 1);
            graphA.union(firstElement, secondElement);
        }

        //Connect vertices A[j] with A[j+3].
        for (int j = 0; j <= 200; j += 3) {
            firstElement = graphA.find(j);
            secondElement = graphA.find(j + 3);
            graphA.union(firstElement, secondElement);
        }

        System.out.println(graphA.find(19));
        System.out.println(graphA.find(112));
    }
}
```

The code above provides implementation for Weighted Quick-Union in Java. After running it we can answer the questions:

- The root of node 19 is **0**.
- The root of node 112 is **112**.

Q4:

What we need to do:

- Explain how we can modify any sorting algorithm to have a **best case** time complexity of **O(n)**.

Explanation:

For any sorting algorithm, we can make it have a best case time complexity of O(n) by simply adding a special case. This special case is true iff the input is already sorted.

For example, you can make a sorting algorithm have a best case time complexity of O(n) by checking if the input array is sorted. If it is then you return the input array as is.

Q5:

What we need to do:

- Explain how an **unstable sorting algorithm** can be turned into a **stable sorting algorithm**.

Explanation:

- Any sorting algorithm, which isn’t stable, can be modified to be stable. This can be done by modifying key comparison operation of the algorithm, so that the comparison of two keys considers position as a factor for objects with equal keys.

Q6:

What we need to do:

- Write down a recursive version of the Selection Sort algorithm.
- Write down the recurrence relation for its worst-case running time.

Solution:

```
public class SelectionSortRec {
    //Points from which index the input array will be sorted.
    private static int index = 0;

    //Helper method.
    //Used to return the minimum element of the input array.
    static int minIndex(int a[], int i, int j)
    {
        //Check if there is only one element left.
        //If so - return it.
        //(This is the base case)
        if (i == j) return i;

        else {
            //Find the smallest element of the remaining ones.
            int k = minIndex(a, i + 1, j);

            //Checks which element is smaller - the current one or the remaining
            //and then returns the smaller of the two.
            return (a[i] < a[k]) ? i : k;
        }
    }

    //Recursive selection sort.
    static void recursiveSelectionSort(int a[])
{
    //Get the length of the input array.
    int n = a.length;

    //If the array is of size 1 -> return.
    //(This is the base case)
    if (index == n) return;

    else {
        //Getting the minimum index of the array via our recursive helper
function.
        int k = minIndex(a, index, n - 1);

        //Swapping when index and minimum index are not same.
        if (k != index) {
            int temp = a[k];
            a[k] = a[index];
            a[index] = temp;
        }
        //Move the index up, as the array is being sorted.
        index++;
        //Recursively calling selection sort function.
        recursiveSelectionSort(a);
    }
}

    public static void main(String args[])
    {
        //Create the array.
        int arr[] = {4, 1, 2, 2, 12, 0};

        //Call the funcion.
        recursiveSelectionSort(arr);

        //Print out the sorted array.
        for (int i = 0; i< arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
```

The worst case time complexity of running selection sort is O(n^2).

The function to find the minimum element index takes **n** time. The recursive call is made to one less element than in the previous call so the recurrence relation for selection sort is: **T(n) = T(n-1) + n**

If the array is of size 1 (**n = 1**), we will just return the array as is (the **base case** in the *recursiveSelectionSort* method).

If the size of the array is 2 (**n = 2**), then we will have **n** operations from the **recursive clause** in the *recursiveSelectionSort* method (else construct).

Based on the above statements we can represent the recursive calls as a function T(n):

If **n = 1** then **T(n) = 1**.

Else **T(n) = T(n - 1) + n**.