COMP1201
Assignment 2

Alberto Tamajo

Student ID: 30696844

March 2020

Electronics and Computer Science Department
University of Southampton

1 Questions

Q1 It can be proven 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, by using the generalised pigeonhole principle. The generalised pigeonhole principle states that if there exists a function $f : X \to Y$ and $|X| > k|Y|$, where $k > 0$ and $k \in \mathbb{N}$, then there exist at least $k + 1$ different elements $\{x_1, x_2, ..., x_k, x_{k+1}\} \subset X$ such that $\forall x \in \{x_1, x_2, ..., x_k, x_{k+1}\}$, $f(x) = y$ where $y \in Y$. In this question, we have an hash function $hash : X \to Y$ defined as $hash(x) = (2d1 + 3d2 + 5d3 + 7d4 + 11d5)\%47$ where % is modulo division, d1 is the most significant digit, and d5 is the least significant one (and the others are in decreasing significance order). It follows from the question that:

- $|X| = 2000$
- $|Y| = 47$

Therefore $|X| > 42|Y|$ which leads to the fact that at least 43 elements belonging to the domain will be mapped to the same element of the codomain. In other words, at least 43 numbers will be stored in the same array index in a linked list since hashing with chaining is being used.

Q2 It is possible to design a data type that supports insert, delete the maximum and delete the minimum in logarithmic time and find the maximum and find the minimum in constant time by using a max-heap and a min-heap. The difference between max-heap and min-heap is that while the max-heap is defined as a binary tree such that given any node x in such a tree, $x.left \leq x \wedge x.right \leq x$, the min-heap is defined as a binary tree such that given any node x in such a tree, $x.left \geq x \wedge x.right \geq x$. In order to guarantee $O(logn)$ time complexity for the delete maximum operation and delete minimum operation, this data structure is going to use data augmentation so that every node of both heaps doesn't store just the object value but also the index in which the respective element in the other heap is stored.Thus, it follows that when a node has changed position due to a percolate up operation or percolate down operation, the respective node in the other heap must be updated of such a change.

- **Insert operation**: inserting an element x to such a data structure involves adding x to both the min-heap and the max-heap. The individual operation of inserting an element to either the max-heap or min-heap in this data structure is identical to the normal inserting operation in a normal heap apart from the fact that when the new element is percolated up, it is necessary to update the position of the nodes moved in their respective nodes in the other heap. The time complexity of this operation is $O(logn)$ as it is possible to find a loose upper bound for this operation which is $O(2logn)$. This upper bound is loose because it considers that at most each inserting operation in both the max-heap and min-heap is $O(logn)$ with a total of $O(2logn)$ but actually this cannot never happen as if for example an element x is added to the data structure and the insert operation of the min-heap takes $\Theta(logn)$ time then x is the minimum element and so inserting x into the max-heap can never take $\Theta(logn)$ but actually would take $\Theta(1)$.
- **Delete Maximum operation**: deleting the maximum node involves swapping the root of the max-heap(node at index 0 of the max-heap array) with the last node of the max-heap array, deleting the maximum node at the last index of the max-heap array and percolating the current root node down. The time complexity of this procedure is $O(logn)$. Similarly, the maximum node must be deleted from the min-heap as well. By using data augmentation, when deleting the maximum node in the max-heap, it is possible to retrieve

the index in which the respective object is located in the min-heap. In the min-heap, this maximum object cannot have a child because otherwise it wouldn't be the maximum element and so it is sufficient to replace it with the last element of the min-heap array, delete the maximum node at the last index of the min-heap and percolate the node which has replaced the ex-maximum node up. The time complexity of this procedure is $O(logn)$. Therefore, it follows that the time complexity of the delete maximum operation is $O(logn) + O(logn) = O(2logn) = O(logn)$

- **Delete Minimum operation**:deleting the minimum object involves swapping the root of the min-heap(node at index 0 of the min-heap array) with the last node of the heap, deleting the minimum node at the last index of the min-heap array and percolating the current root node down. The time complexity of this procedure is $O(logn)$. Similarly, the minimum object must be deleted from the max-heap as well. By using data augmentation, when deleting the minimum element in the min-heap, it is possible to retrieve the index in which the respective object is located in the max-heap. In the max-heap, this minimum object cannot have a child because otherwise it wouldn't be the minimum element and so it is sufficient to replace it with the last element of the max-heap array, delete the minimum node at the last index of the max-heap array and percolate the node which has replaced the ex-minimum node up. The time complexity of this procedure is $O(logn)$. Therefore, it follows that the time complexity of the delete maximum operation is $O(logn) + O(logn) = O(2logn) = O(logn)$
- **Find minimum operation**: finding the minimum object in this data structure involves returning the root of the min-heap (node at index 0 of the min-heap array). The time complexity of this operation is $\Theta(1)$.
- **Find maximum operation**: finding the maximum object in this data structure involves returning the root of the max-heap (node at index 0 of the max-heap array). The time complexity of this operation is $\Theta(1)$

A pseudo code for these operations is provided below.

- maxHeapArray is a dynamic array which stores the elements of the max-heap
- minHeapArray is a dynamic array which stores the elements of the min-heap
- every index of both the maxHeapArray and minHeapArray stores an augmentation node which stores the value of the element added and the index in which the respective node in the other heap is stored.
  getIndexRespectiveNode() is used to return the index in which the respective node is stored in the other heap and **setIndexRespectiveNode()** is used to set the index in which the respective node is stored in the other heap.
  setElement() is used to set the element the augmentation node needs to store.

```
procedure FIND_MAXIMUM()
    return maxHeapArray[0]

procedure FIND_MINIMUM()
    return minHeapArray[0]

procedure PERCOLATE_UP_MAX−HEAP(childIndex)

    while (parentIndex ← ⌊(childIndex−1)/2⌋) ≥ 0

        childNode ← maxHeapArray[childIndex]
        parentNode ← maxHeapArray[parentIndex]

        if childNode > parentNode then

            respectiveChildNode ← minHeapArray[childNode.getRespectiveIndexNode()]
            respectiveParentNode ← minHeapArray[parentNode.getRespectiveIndexNode()]

            respectiveChildNode.setIndexRespectiveNode(parentIndex)
            respectiveParentNode.setIndexRespectiveNode(childIndex)

            swap childNode with parentNode
            childIndex ← parentIndex

        else
            exit

procedure PERCOLATE_UP_MIN−HEAP(childIndex)

    while (parentIndex ← ⌊(childIndex−1)/2⌋) ≥ 0

        childNode ← minHeapArray[childIndex]
        parentNode ← minHeapArray[parentIndex]

        if childNode < parentNode then

            respectiveChildNode ← maxHeapArray[childNode.getRespectiveNode()]
            respectiveParentNode ← maxHeapArray[parentNode.getRespectiveNode()]

            respectiveChildNode.setRespectiveNode(parentIndex)
            respectiveParentNode.setRespectiveNode(childIndex)

            swap childNode with parentNode
            childIndex ← parentIndex

        else
            exit
```

```
procedure PERCOLATE_DOWN−MAX−HEAP()

    parentIndex ← 0
    while (childindex ← (2∗parentIndex) +1) ≤ maxHeapArray.length()

        parentNode ← maxHeapArray[parentIndex]

        if maxHeapArray[childIndex]< maxHeapArray[childIndex+1] then
            childIndex ← childIndex +1

        childNode ← maxHeapArray[childIndex]

        if parentNode < childNode then
            swap parentNode with childNode

            respectiveChildNode ← minHeapArray[childNode.getRespectiveNode()]
            respectiveParentNode ← minHeapArray[parentNode.getRespectiveNode()]

            respectiveChildNode.setRespectiveNode(parentIndex)
            respectiveParentNode.setRespectiveNode(childIndex)

            parentIndex ← childIndex

        else
            exit

procedure PERCOLATE_DOWN−MIN−HEAP()

    parentIndex ← 0
    while (childindex ← (2∗parentIndex) +1) ≤ minHeapArray.length()

        parentNode ← minHeapArray[parentIndex]

        if minHeapArray[childIndex]> minHeapArray[childIndex+1] then
            childIndex ← childIndex +1

        childNode ← minHeapArray[childIndex]

        if parentNode > childNode then
            swap parentNode with childNode

        respectiveChildNode ← maxHeapArray[childNode.getRespectiveNode()]
        respectiveParentNode ← maxHeapArray[parentNode.getRespectiveNode()]

        respectiveChildNode.setRespectiveNode(parentIndex)
        respectiveParentNode.setRespectiveNode(childIndex)

        parentIndex ← childIndex
```

```
        else
            exit

procedure INSERT(element)

    create an augmentation node called minHeapNode
    create an augmentation node called maxHeapNode

    minHeapNode.setElement(element)
    maxHeapNode.setElement(element)

    minHeapArray.add(minHeapNode)
    maxHeapArray.add(maxHeapNode)

    minHeapNode.setIndexRespectiveNode(maxHeapArray.length()−1)
    maxHeapNode.setIndexRespectiveNode(minHeapArray.length()−1)

    PERCOLATE_UP_MIN−HEAP(minHeapArray.length()−1)
    PERCOLATE_UP_MAX−HEAP(maxHeapArray.length()−1)

procedure DELETE_MINIMUM()

    minimumNodeMinHeap ← FIND_MINIMUM()
    indexRespectiveNode ← minimumNodeMinHeap.getIndexRespectiveNode()

    swap maxHeapArray[indexRespectiveNode] with maxHeapArray[maxHeapArray.length()−1]

    delete maxHeapArray[maxHeapArray.length()−1]

    PERCOLATE_UP_MAX−HEAP(indexRespectiveNode)
    REMOVE_MIN()

procedure DELETE_MAXIMUM()

    maximumNodeMaxHeap ← FIND_MAXIMUM()
    indexRespectiveNode ← maximumNodeMaxHeap.getIndexRespectiveNode()

    swap minHeapArray[indexRespectiveNode] with minHeapArray[minHeapArray.length()−1]

    delete minHeapArray[minHeapArray.length()−1]

    PERCOLATE_UP_MIN−HEAP(indexRespectiveNode)
    REMOVE_MAX()

procedure REMOVE_MAX()

    swap maxHeapArray[0] with maxHeapArray[maxHeapArray.length()−1]
```

delete maxHeapArray[maxHeapArray.length()−1]
PERCOLATE_DOWN_MAX−HEAP()

procedure REMOVE_MIN()

swap minHeapArray[0] **with** minHeapArray[minHeapArray.length()−1]

delete minHeapArray[minHeapArray.length()−1]
PERCOLATE_DOWN_MIN−HEAP()

Q3 (a) The number of components in the resulting graph are 61. The root of node 19 is 201 and the root of node 112 is 113.

(b) The number of components in the resulting graph are 61. The root of node 19 is 0 and the root of node 112 is 112.

QuickUnion Class:

```
public class QuickUnion {

    int[] arrayNodes;
    int countComponents;

    public QuickUnion(int initialNumbercomponents){

        this.arrayNodes = new int[initialNumbercomponents];

        for(int i=0; i<initialNumbercomponents; i++){
            arrayNodes[i]=i;
        }

        this.countComponents=initialNumbercomponents;

    }

    private int find(int node){

        while(arrayNodes[node]!=node){
            node= arrayNodes[node];
        }
        return node;
    }

    public int getCountComponents(){
        return countComponents;
    }

    public boolean connected(int a, int b){
        return find(a)==find(b);
```

```
    }

    public void union(int a, int b){


        int rootNodeA=find(a);
        int rootNodeB=find(b);
        if(rootNodeA==rootNodeB){
            return;
        }

        arrayNodes[rootNodeA]=rootNodeB;
        countComponents--;


    }

    public static void main(String[] args) {
        QuickUnion quickUnion= new QuickUnion(256);

        for(int i=0; i<=254; i+=2){
            quickUnion.union(i,i+1);
        }

        for(int i=0; i<=200; i+=3){
            quickUnion.union(i,i+3);
        }

        System.out.println("Number components: " +  quickUnion.getCountComponents()
        System.out.println("Find 19: " + quickUnion.find(19));
        System.out.println("Find 112: " + quickUnion.find(112));
    }
}
```

WeightedQuickUnion Class:

```
public class WeigtedQuickUnion {

    int[] arrayNodes;
    int countComponents;
    int[] sizeComponentForRoot;

    public WeigtedQuickUnion(int initialNumbercomponents){

        this.arrayNodes = new int[initialNumbercomponents];
        this.sizeComponentForRoot= new int[initialNumbercomponents];

        for(int i=0; i<initialNumbercomponents; i++){
            arrayNodes[i]=i;
```

```
        }

        for(int i=0; i<initialNumbercomponents; i++){
            sizeComponentForRoot[i]=1;
        }

        this.countComponents=initialNumbercomponents;

    }

    private int find(int node){

        while(arrayNodes[node]!=node){
            node= arrayNodes[node];
        }
        return node;
    }

    public int getCountComponents(){
        return countComponents;
    }

    public boolean connected(int a, int b){
        return find(a)==find(b);
    }

    public void union(int a, int b){


        int rootNodeA=find(a);
        int rootNodeB=find(b);
        if(rootNodeA==rootNodeB){
            return;
        }

        if(sizeComponentForRoot[rootNodeA] < sizeComponentForRoot[rootNodeB]){
            arrayNodes[rootNodeA]=rootNodeB;
            sizeComponentForRoot[rootNodeB]+=sizeComponentForRoot[rootNodeA];
        }else{
            arrayNodes[rootNodeB]=rootNodeA;
            sizeComponentForRoot[rootNodeA]+=sizeComponentForRoot[rootNodeB];
        }

        countComponents--;

    }

    public static void main(String[] args) {
```

```
        WeigtedQuickUnion weightedQuickUnion= new WeigtedQuickUnion(256);

        for(int i=0; i<=254; i+=2){
            weightedQuickUnion.union(i,i+1);
        }

        for(int i=0; i<=200; i+=3){
            weightedQuickUnion.union(i,i+3);
        }

        System.out.println("Number components: " + weightedQuickUnion.getCountCompo
        System.out.println("Find 19: " + weightedQuickUnion.find(19));
        System.out.println("Find 112: " + weightedQuickUnion.find(112));
    }
}
```

Q4 It is possible to have $O(n)$ best case time complexity for any sorting algorithm based on the comparison model by initially traversing the input array so that to check whether it is already sorted. This procedure takes $O(n)$ time given that n is the number of elements inside the input array. Following this, a pseudo code of such a procedure is present (assuming an ascending order).

```
procedure CHECK_IF_SORTED(a)
    for i ← 0 to a.length−2 do
        if a_i > a_{i+1}
            return false

    return true
```

Q5 A sorting algorithm is stable if objects with same key appear in the same order in the sorted output as in the input array (this key will be called primary key for the rest of this text). Therefore, it is possible to turn an unstable algorithm into a stable one by storing the position of any object in the original array and use this additional information when two objects being compared happen to have same primary key. An example will clarify this concept. Let 2 object A and B have same primary key so that $A.primaryKey == B.primaryKey$. A sorting algorithm that makes use of the procedure described above would compare the two objects A and B and find out that $A.primaryKey == B.primaryKey$ and so it would compare A.secondaryKey with B.secondaryKey, where the secondaryKey represents the position of the object inside the input array,in order to make a comparison-based decision.
An easy way to implement the procedure described above is to use data augmentation which is easily achievable by using an OOP-language. Data augmentation consists of adding extra information to a data structure and so in an OOP-language this translates into adding an additional field to a class which stores extra information. In order to use data augmentation to add the position of each object inside the original array, $\Theta(n)$ space and $\Theta(n)$ time are needed because we need to augment each object inside the original array and add it to another array. The sorting algorithm will sort the array containing the augmented objects and once such array is sorted we need to traverse it and add each object (not-augmented) to the original array. A pseudo code of this procedure is described below. The augmented data structure contains the following operations:

- **setElement()**: copies the value stored inside the original array in the augmentation data instance
- **setSecondaryKey()**: sets the index at which the element was stored in the input array
- **getElement()**: returns the element stored

procedure SORT(inputArray)

create **a** dynamic array called augmentedArray

//Adds the elements of the input array into the augmentedArray
for i ← 0 **to** inputArray.length()−1 **do**

create augmented data structure called augmentedData

augmentedData.setElement(inputArray[i])
augmentedData.setSecondaryKey(i)
augmentedArray.add(augmentedData)

Sort using any comparison−based algorithm
which uses the secondaryKey in order **to**
make **a** comparison−based decision in case
two objects have same primaryKey

//Adds the elements sorted in the input array
for i ← 0 **to** inputArray.length()−1 **do**
inputArray[i] ← augmentedArray[i].getElement()

Q6 The pseudocode of the recursive procedure for selection-sort is provided below. It is important to notice that 2 versions of the recursive algorithm are provided. One version is not fully-recursive in the sense that uses a for loop inside it. Instead, the other version is fully recurusive because replaces the for-loop inside with a recursive auxiliary procedure,

procedure SELECTIONSORT_FULLY_RECURSIVE(**a**,index)

if index← **a**.length−1 **then**
return a

min← SELECTIONSORT_AUXILIARY(**a**,index)

swap a_{index} **with** a_{min}

return SELECTIONSORT_FULLY_RECURSIVE(**a**,index+1)

```
procedure SELECTIONSORT_AUXILIARY(a,index)

    if index← a.length−1 then
        return index

    min← SELECTIONSORT_AUXILIARY(a,index+1)

    if a_index < a_min
        return index

    return min

procedure SELECTIONSORT_RECURSIVE(a,index)

    if index← a.length−1 then
        return a

    min← index

    for i← index+1 to a.length−1 do
        if a_i < a_min
            min← i

    swap a_index with a_min

    return SELECTIONSORT_RECURSIVE(a,index+1)
```

The recurrence relation of Selection Sort in its worse case is $T(n) = T(n-1)+n$ or equivalently $T(n) = T(n-1) + O(n)$ when $n \geq 2$. $T(n) = 1$ when $n = 1$. This recurrence relation follows because given that the size of an array is equal to n and the recursive algorithm is located at index p where $p \leq (n-1)$ then the algorithm will perform $n - p - 1$ comparisons. However, in the worst case the algorithm also performs 1 swap per index. Therefore, the work done per each index is $n - p$. In other words, when the algorithm is located at index 0 the work done will be n, when the algorithm is located at index 1 the work done will be $n - 1$ and so on.