Assignment 3

Student ID: 480048691

April 8, 2020

1 Problem 1

1.1 PickLargest

Let there be 4 games:

- $\mathrm{G_A}$ with fun factor = 80 and space = 13.
- $\mathrm{G_B}$ with fun factor = 81 and space = 7.
- $\mathrm{G_C}$ with fun factor = 75 and space = 4.
- $\mathrm{G_D}$ with fun factor = 40 and space = 3.

Let our space limit on the shelf be equal to 20.

Using the **PickLargest** algorithm, we will first sort the list of games in the order of $\mathrm{G_B}$, $\mathrm{G_A}$, $\mathrm{G_C}$, $\mathrm{G_D}$ (given that $f_B >= f_A >= f_C >= f_D$). The algorithm will then choose $\mathrm{G_B}$ and $\mathrm{G_A}$ with the combined fun factors of 80 + 81 = 161, and the space taken is 20 out of 20. These two games, however, does not maximise the amount of fun we can have. Indeed, if we had placed $\mathrm{G_B}$, $\mathrm{G_C}$, and $\mathrm{G_D}$ on the shelf, we will have a higher combined fun factor of 81 + 75 + 40 = 196 with the space taken is also within the space limit of 20. Thus, **PickLargest** does not always return the maximum fun we can have.

1.2 PickLargestRatio

Let there be 4 games:

- $\mathrm{G_A}$ with fun factor = 43 and space = 45. $\frac{f_A}{s_A} = \frac{43}{45} = 0.96$
- $\mathrm{G_B}$ with fun factor = 75 and space = 70. $\frac{f_B}{s_B} = \frac{75}{70} = 1.07$
- $\mathrm{G_C}$ with fun factor = 46 and space = 46. $\frac{f_C}{s_C} = \frac{46}{46} = 1$

Let our space limit on the shelf be equal to 100.

Using the **PickLargestRatio** algorithm, we will first sort the games in the order of $\mathrm{G_B}$, $\mathrm{G_C}$, $\mathrm{G_A}$ (given that $\frac{f_B}{s_B} >= \frac{f_C}{s_C} >= \frac{f_A}{s_A}$).

The algorithm will then choose $\mathrm{G_B}$ as it has the largest fun/space ratio. Thus, the fun factor returned by the **PickLargestRatio** algorithm would be 75, with the space taken of 70 out of 100. Since the space required by A and C are larger than that available on the shelf, they were not added.

The game $\mathrm{G_B}$ alone, however, does not maximise the amount of fun we can have. Indeed, if we had chosen the two games $\mathrm{G_C}$ and $\mathrm{G_A}$, we will get a higher fun factor of 43 + 46 = 89 and we would have used 45 + 46 = 91 space capacity, which is also within the given space limit. Therefore, the **PickLargestRatio** algorithm also does not always return the correct solution.

2 Problem 2

2.1 Initialising the data structure

Assuming that the doubly-linked lists are sorted in an ascending order, with the minimum element of each list at the start and maximum at the end. Initially, we add the head of each list to a priority queue, queue A, and the tail of each list to a priority queue, queue B. Queue A is implemented with a **min-heap** (we call heap A) while queue B is implemented with a **max-heap** (heap B). Heap A will have the property

$$key(m) >= key(parent(m))$$

for m != root and the root always holds the smallest key in the heap. Heap B, on the other hand, has the property

$$key(m) <= key(parent(m))$$

for m != root and the root always holds the largest key. We always keep at most k items in each heap.

We add the head of all k doubly-linked lists to an array. We build heap A from scratch by adding the head of each doubly-linked list one by one, by means of k successive insert operations. Every time a new node is added, a 'heapify' operation called **upheap(e)** is carried out to restore the heap-order property of a **min-heap** by swapping keys (if needed) from the newly added node e towards the root. Constructing this heap A of k elements will take O(k) time (according to lectures).

Similarly, heap B will also be built from scratch by adding the tail of each doubly-linked list. Every time a new node is added, the operation **upheap(e)** is also carried out to restore the heap-order property of a **max-heap**.

REMOVE-MIN

Our data structure supports the operation **REMOVE-MIN** in the following manner. Firstly, we store the root of the min-heap as a variable **u**. We know that this root node is a head node in one of the doubly-linked lists L_i. We then remove the root from the min-heap and replace it with the u.next in the corresponding linked list. If there happens to be a duplicate of a node (one node exists in both heaps, then we must also remove the node in the max-heap, this happens as we gradually remove almost all elements of a list). The min-heap property will be restored by calling a **downheap** operation at the root. We then remove the head node in the linked list by setting new head = u.next. Since we know that the root of a min-heap always holds the smallest value, we will return the removed head (u) as it is the smallest element in all n elements.

In the case where u.next is None in the corresponding linked-list (implying we are at the last element of a list L_i), we replace the root by the last node of the mid-heap. We then again remove u (essentially removing the list) and call **downheap(root)** to restore the heap-order property.

```
function removeMin():
    u <- heapA.root
    remove u from the heap A
    remove u from the heap B (if u is in heap B)
    if u.next is None:
        heapA.root <- last node in the min-heap
        head <- null
    else if u.next is not None:
        heapA.root <- u.next in the corresponding linked list
        head <- head.next
        head.prev <- null
    downheap(root)
    return u
```

REMOVE-MAX

The **REMOVE-MAX** operation is carried out in almost the same manner. However, we are now working with the tails of the doubly-linked lists. We first store the root of the max-heap as variable **v**. v is the tail of one of the k lists since this is how we constructed the max-heap at the start. We remove v from heap B and replace it by v.prev in the corresponding linked list. Once again, we restore the max-heap property by calling downheap at the root. We then remove v from its linked list by setting the new tail = v.prev. Since the root of a max-heap always holds the largest value, we return v.

```
function removeMax():
    v <- heapB.root
    remove v from the heap B
    remove v from the heap A (if v is in heap A)
    if v.prev is None:
        root <- last node in the max-heap
        tail <- null
    if v.prev is not None:
        root <- v.prev in the corresponding linked list
        tail <- tail.prev
        tail.next <- null
    downheap(root)
    return v
```

Once again, in the case where v.prev is None (implying we are at the last node of a list), the root of the max-heap will be replaced by the heap's last node. We then remove v (essentially removing the list) and perform a downheap operation from the root.

2.2 Argue the correctness

Regarding the initialisation of the data structure, we first have an array of all the head elements from k lists. Given our assumption that the lists are sorted in an ascending manner, while this array is unsorted, we can ensure that the minimum element of a **n** elements is within this array. Based on the property of a min-heap (mentioned above), we can ensure that the minimum value is the root of the min-heap. Applying the same logic for the tail elements from k lists, we know that the maximum value of all n elements is the root of the max-heap. The **upheap(e)** operation being performed every time a node is added is what gives us this certainty.

Every time REMOVE-MIN or REMOVE-MAX is called, the root of the min-heap is replaced with either u.next (or the root of the max-heap being replaced with v.prev) or the last node of the heap. The head of the corresponding linked list will also be updated to u.next (or u.prev) or the linked list will be removed and the current node u will be set to null. This ensures that we do not count one element twice. We also covered the case where the same node exists in both heaps. Performing **downheap(root)** after replacing the root is what helps us maintaining the heap-order property and allows us to performing REMOVE-MIN and REMOVE-MAX repeatedly.

2.3 Analyse the running time of the operations

According to lectures, building a min-heap or a max-heap from an unsorted list of k elements will require O(k) time. In the worst case where k = n, that is, when the number of lists is equal to the total number of elements, this will take O(n) time to build each heap. Thus, to initialise our data structures will require O(n) + O(n) = O(n) time.

Replacing the root node and updating the corresponding linked lists altogether takes O(1) time. It is the **downheap(root)** operation at the end that requires O(log k) time because the height of the heap is log k. Occasionally, we delete a node from one of the trees that also takes O(log k) time. Therefore, **REMOVE-MIN** and **REMOVE-MAX** will each take at most O(log k) time as required since we always keep at most k elements in each heap.

3 Problem 3

If one of the trees has size n and the latter has size 0, we can implement the function **rank(k)** on a node u in order to find the kth element (in sorted order) stored in the tree. Note that each node u has a subtree value that of the number of elements of the subtree rooted at u.

Starting at the root, we check if $u.left.stv >= k$, if yes, we know that the kth element is in u.left and continue searching for k in u.left. If $u.left.stv = k - 1$, the kth element we are looking for is at u. If $u.left.stv < k - 1$, then the kth element is in the right subtree and we search for k in u.right.

Given the property of an AVL tree and that there n odd distinct integers, the median of the elements in the tree can quickly be found by carrying out rank($\frac{root.stv+1}{2}$).

Given that we do O(1) time at each work on n elements and the height of the tree is at most log(n) (property of an AVL tree). The above operation will take at most O(log n) time.