Computer Science B63
University of Toronto Scarborough

Homework Exercise # 4

Handing in and marking

For this exercise, you need to submit your solutions to the pencil-and-paper exercises on crowdmark and your solutions to the programming question on MarkUs. Your pencil-and-paper solutions will be marked with respect to correctness, clarity, brevity, and readability. Your code will be marked with respect to correctness, efficiency, program design and coding style, clarity, and readability. This exercise counts for 15% of the course grade.

Question 1. Disjoint Sets Basics [10 MARKS]

Consider the following sequence of operations:

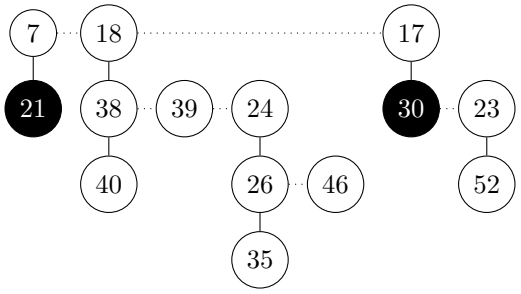
```
for i in 1,2, ..., 16: make-set(i)
for i in 1,3, ..., 15: union(i, i+1)
for i in 1, 5, 9, 13: union(i, i+2)
union(1,5); union(11,13); union(1,10)
find-set(2)
find-set(16)
```

Show the resulting data structure, if disjoint sets are implemented using:

1. linked lists (as we did in class), assuming that if set (i) and set (j) have the same size, the algorithm `union(i, j)` appends elements of (j) to elements of (i),
2. disjoint-set forests with union by rank, but without path compression, and
3. disjoint-set forests with union by rank and with path compression.

Question 2. Fibonacci Heaps Basics [10 MARKS]

Recall the Fibonacci Heap from our example in class (with some nodes now marked).



Show the heap after each of the following operations, executed in sequence:

1. `decrease-priority(`(46)`, 10)`
2. `decrease-priority(`(26)`, 12)`
3. `extract-min()`

Question 3. Variations of Expandable Arrays [20 MARKS]

Some modern implementations of expandable arrays expand by a factor of 1.5 instead of 2, i.e., `add(x)` is:

```
add(x):
0. if arr is empty:
1.   arr := new array of length 1
2. if size = capacity:
3.   capacity := ceil(1.5 * capacity)
4.   newArr := new array of length capacity
5.   copy elements of arr into newArr
6.   arr := newArr
7. arr[size++] := x
```

Prove that the amortised time of append is still $\mathcal{O}(1)$ using each of the following methods:

1. accounting method,
2. potential method.

Question 4. Average Running Time [20 MARKS]

This question is about searching an array A, of size n, for a value x. Suppose x occurs k times in A, with $1 \leq k \leq n$ (so x occurs at least once).

Two algorithms will be considered below, and their running times are proportional to the number of array accesses, so we will focus on counting the number of those accesses, and we will aim at expected values.

1. **Simple randomized approach**

   Randomly (uniformly) pick i between 1 and n, ask whether $A[i] = x$, terminate if yes, repeat if no.

   ```
   while True:
     i := random(1, n)
     if A[i] = x:
       break
   ```

2. **Linear search**

   This is the usual linear search.

   ```
   for i = 1 to n:
     if A[i] = x:
       break
   ```

We will calculate its expected number of array accesses, assuming uniformly random input, i.e., all permutations of array content are equally likely.

1. What is the expected number of times the *randomized* algorithm accesses the array?

2. For *linear search*, add the extra local assumption $k < n$.

   Let X_n be the random variable for the number of array accesses by the above linear search, with array size n. Let X_{n-1} be the random variable for the number of array accesses by a similar linear search, but with array size $n - 1$. Prove:

   $$E(X_n) = 1 + \frac{n-k}{n} E(X_{n-1})$$

3. Again let X_n be the random variable for the number of array accesses by linear search, with array size n. Use induction on n to prove:

   $$E(X_n) = \frac{n+1}{k+1}$$

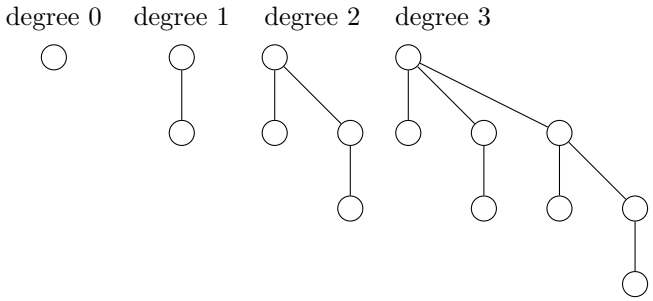
   Note that $k = n$ is possible in this part, unlike the previous part.

Question 5. Binomial Heap [20 MARKS]

A *binomial heap* is yet another way to implement a mergeable priority queue. It consists of a collection of *binomial trees* with restrictions. So let's begin with binomial trees: They are defined inductively by their degrees (of their roots):

- A binomial tree of degree 0 is one single node. (That node is the root and it has 0 children.)
- A binomial tree of degree $d+1$ (number of root's children) is built by taking two binomial trees, both of degree d, and "linking" them—adding one as a new rightmost child of the root of the other.

Here are a few example pictures (tree shapes shown, node content omitted):



There are two properties. A binomial tree of degree d has exactly 2^d nodes. A root's subtrees are ordered from left to right by strictly increasing degrees because the linking adds a new, bigger subtree at the right end.

A binomial heap is a collection of such trees with two restrictions:

- We're implementing a min priority queue, say. So each node stores a priority (and a job), and parent's priority $\leq$ child's priority.
- In the collection, no two trees have the same degree.

There is one more property: If the total number of nodes is n, then its binary notation corresponds to which degrees are present. For example if $n = 2^3 + 2^1 = 1010_2$, then we have a tree of degree 3, a tree of degree 1, and nothing else.

It will be useful to store the collection in a linked list ordered by strictly increasing degrees; likewise, each node has [a pointer to] a linked list of its children/subtrees ordered by strictly increasing degrees. Singly linked lists (with "last" pointers) are good enough.

We will calculate the amortised times of a few priority queue operations. Use the potential method; use the potential function "the number of trees (roots)".

a. [5 marks] Prove that a binomial heap with n nodes in total contains at most $\lfloor \lg(n+1) \rfloor$ trees.

b. [5 marks] insert(k, j): Add a job j of priority k.

   Create a new node to hold that job and priority (and no children). This is a tree of degree 0. Add it to the left of the list of trees. If you now have two trees of degree 0 (just check the next tree in the list), link them (whoever has a greater priority becomes the new child), resulting in a tree of degree 1. And if now you have two trees of degree 1, link them, resulting in a tree of degree 2. And if now you have... Loop over this until your new tree is the only one with its degree.

   This is analogous to computing $n + 1$ in binary.

   Prove that the amortised time of insert is $O(1)$.

c. [5 marks] union(B_1, B_2): Merge them into one binomial heap.

   Walk through both lists of trees from left to right (increasing degrees), merging them and linking same-degree trees as we go.

   This is analogous to computing $n_1 + n_2$ in binary, where n_1 and n_2 are the respective number of nodes in the two heaps.

   Prove that the amortised time of union is $O(\lg(n_1 + n_2))$.

d. [5 marks] Give an algorithm for extract-min. It should have amortised time $O(\lg n)$ (but no need to prove this). You may like to use the union operation at some point.