CPSC 320 2022W1: Assignment 6

This assignment is **for extra practice only**. So you do not have to submit it, and in fact there won't be any place where you could submit it.

Ok, time to get started...

1 Identifying Superspreader Events

(You can skip to the definition of SUPERSPREAD without missing any vital information for this question.)

Imagine a hypothetical, moderately contagious disease that spreads via respiratory droplets and aerosols. Research has shown that this disease spreads mainly via "superspreader" events, where many people are gathered together in the same space. Public health officials want you to help them create an algorithm to identify these superspreader events.

Specifically, out of a population of people, some subset has been identified as having gotten the disease. The public health officials also have a list of all events where people congregated, and they want to find a small, "highly likely" (for some definition of "likely") set of events that include all the infected people.

Formally, we define our problem SUPERSPREAD as a decision problem as follows: An instance of SUPERSPREAD consists of a set P, with $|P| = n$; a set $E = \{E_1, \ldots, E_m\}$, where each $E_i \subseteq P$; a set $I \subseteq P$; and a numerical bound k. SUPERSPREAD asks whether there exists a subset $E' \subseteq E$, such that

$$I \subseteq \bigcup_{E'_i \in E'} E'_i$$

and

$$\sum_{E'_i \in E'} \text{weight}(E'_i) \leq k,$$

where $\text{weight}(E'_i)$ is defined to be equal to

$$\text{weight}(E'_i) = \frac{|(P - I) \cap E'_i| + 1}{|I \cap E'_i| + 1}.$$

Intuitively, P is the total population of n people. E is the set of events $E_1, \ldots, E_m$, where each event is a set of people (who attended that event). I is the set of infected people. We are trying to find a subset E' of all the events, such that each infected person attended at least one event in E', and we want E' to be a "small" set of events (the sum of the weights of events in E' is $\leq k$). Without this last constraint, we could always return $E' = E$ as a solution. The weight function might seem a bit odd, but it's designed to discourage including events where a lot of people did *not* get infected (which is the number $|(P - I) \cap E'_i|$) versus the number of people who *did* get infected (which is the number $|I \cap E'_i|$). For example, for a choir practice where 52 out of 61 were infected, the weight would be $\frac{9+1}{52+1} \approx 0.19$, whereas a dental conference where 44 out of 15,000 people were infected would get a weight of $\frac{14956+1}{44+1} \approx 332.38$. It wouldn't be very useful to try to contact-trace through 15,000 attendees, so we would prefer a solution that selected smaller sub-events, even if we needed several smaller sub-events (e.g., a specific dinner, a specific seminar, a specific party, etc.) to explain all 44 attendees who got sick.

1. (4 points) Explain why SUPERSPREAD is in NP.

   Solution: As usual, we'll be more detailed in these solutions than we'd expect of you. For example, for this part of the question, an answer roughly of the form:

> SUPERSPREAD is in NP because given a subset E' that purports to be a solution, it is obviously polynomial time to just traverse all the sets in E' to make sure that all of I is included in their union, and similarly, we can traverse through the sets in E', compute each of their weights, add the weights up, and compare to k.

would be fine.

However, let's be a bit more precise here, just to be sure. We'll assume we represent the set P by the integers $1, \ldots, n$. This means the input instance doesn't have to list P, but just provide the value n (and to be really formal, we might specify that n is given in binary). The set E is then given by m lists of numbers between 1 and n inclusive. Then, the set I is also a list of numbers. And k is a number, also given in binary. This means, if we wanted to be super formal, that the length of the input is bounded by the number of elements in all the sets E_i plus the number of elements in I, all times $\lg n$ (plus $2 \lg n$ for n and k and maybe some formatting bits, both those are all lower-order terms). Let's call this input length N.

Now, we can read the input and store the data in any convenient data structure. Reading all the input will take $O(N)$ time, as we don't have to re-read anything. We can store I as a linked list of numbers, and E as an array of linked lists of numbers.

Now, given this (rather inefficient, but good enough) representation, what is our runtime to check that $I \subseteq \bigcup_{E'_i \in E'} E'_i$? We can copy the list I, and then make a single pass through all the E', deleting elements from the copy of I for each element in each E'_i. Since the length of I and the length of all of E' are both bounded by N, we can obviously bound the runtime by $O(N^2)$.

Similarly, to check the sum of the weights, we can traverse all of E' again. For each element in each E'_i, we can check in $O(N)$ time whether the element is in I or not, which lets us compute the weight for each subset E'_i. So, again, we get a bound of $O(N^2)$.

Combined, this bounds our overall checking runtime by $O(N^2)$, which is polynomial.

(Again, we emphasize that we do NOT expect you to write out such a detailed solution. We provided this just so you can see that it's possible to account for things carefully, and so you can convince yourself it really is polynomial if it wasn't obvious to you.)

2. (8 points) Complete the proof that SUPERSPREAD is NP-complete by providing a reduction from the SET COVER problem to SUPERSPREAD, and proving your reduction correct. The SET COVER problem is defined in your textbook in Section 8.1 as follows:

> Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?

Solution: A typical reduction/proof like this will consist of four parts: (1) explaining the reduction, (2) proving that if the original instance is a "yes" instance, then the instance produced by the reduction is also a "yes" instance (for the different problem), (3) proving that if the instance proved by the reduction is a "yes" instance, then the original instance was a "yes" instance, and (4) explaining that the reduction is polynomial time.

(1) (We'll first give some intuition, but this wouldn't be formally part of the answer.) SUPERSPREAD looks a lot like SET COVER, so the reduction shouldn't be too complicated. In SUPERSPREAD, we have to cover the set I, which is a subset of the overall population P; in SET COVER, we have to cover the entire set U. In SUPERSPREAD, we do the covering with events E_i, which are subsets of P that can contain members who are in or not in I; in SET COVER, we do the covering with the subsets S_i, which are subsets of U, all of which we have to cover. In SUPERSPREAD, we add up the weights of the subsets E'_i and make sure the sum is less than or equal to k; in SET COVER, we count the subsets we use in the cover, and make sure that we use at most k subsets. Note that this is equivalent

to giving each subset weight 1 and adding up the "weights". So, to reduce a SET COVER instance into a SUPERSPREAD instance, it seems like we'd want to make $I = U$ (since we have to cover all of U), and then somehow fiddle with the subsets E'_i to make their weights equal to 1. And that's exactly what we'll do. Here's the formal reduction:

Given an instance of SET COVER with set U with n elements, subsets $S_1, \ldots, S_m$ of U, and a number k, we construct an instance of SUPERSPREAD as follows: Let $P = U \cup V$, where the set V contains n new elements, in 1-1 correspondance with the elements in U, i.e., if $U = \{u_1, \ldots, u_n\}$, then let $V = \{v_1, \ldots, v_n\}$, with each v_i considered to be the "match" to u_i. Next, let $I = U$. Now, create m subsets $E_1, \ldots, E_m$ of P as follows: Let each $E_i = S_i \cup T_i$, where $T_i = \{v_j \mid u_j \in S_i\}$, i.e., E_i is just S_i unioned with a set containing the corresponding elements to S_i taken from V. Notice that this makes the weight$(E_i) = 1$ for all i, because each E_i has equal numbers of elements from U (which is equal to I) and from V (which are not in I). The bound for our SUPERSPREAD instance will be the same bound k as in the SET COVER instance.

(2) If the SET COVER instance is a "yes" instance, then there are k of the subsets from $S_1, \ldots, S_m$, whose union is equal to all of U. Without loss of generality, let those k subsets be $S_1, \ldots, S_k$. (If they weren't, we could renumber the subsets.) Now, let $E' = \{E_1, \ldots, E_k\}$. Since each subset E_i includes all of S_i, and the union $\bigcup_{i \in 1,\ldots,k} S_i = U = I$ then we have:

$$I \subseteq \bigcup_{E'_i \in E'} E'_i.$$

And since the weight of each E'_i is 1, then the sum:

$$\sum_{E'_i \in E'} \text{weight}(E'_i) = \sum_{i \in 1,\ldots,k} 1 = k \leq k.$$

Therefore, E' is a witness that the SUPERSPREAD instance is a "yes" instance.

(3) If the SUPERSPREAD instance is a "yes" instance, then there is a subset $E' \subseteq E$, such that

$$I \subseteq \bigcup_{E'_i \in E'} E'_i$$

and

$$\sum_{E'_i \in E'} \text{weight}(E'_i) \leq k.$$

Recall that the reduction guarantees that the weight of any E_i is equal to 1, so the fact that the sum of the weights $\leq k$ tells us that there are at most k subsets in E'. Now, for each $E'_i \in E'$, select the correponding subset in the SET COVER instance (e.g., you can compute it by intersecting the set with U). As noted there are at most k selected from the original collection $S_1, \ldots, S_m$ of subsets of U. And since the union of the $E'_i \in E'$ covers all of I, and $I = U$, the union of the selected subsets of U also covers all of U. Thus, we have a solution to the SET COVER instance.

(4) The reduction is obviously polynomial time (linear time, really), as we are essentially making two copies of the SET COVER instance.

2 Septimated queues

Recall that a standard queue maintains a sequence of items subject to the following operations:

- `Enqueue(x)`: adds an item `x` to the end of the sequence.
- `Dequeue()`: removes and returns the item `x` at the ront of the sequence.
- `Size()`: returns the number of elements of the sequence.

It is easy to implement a queue using a doubly-linked list (or even using a pair of stacks) so that it uses $\Theta(s)$ space, where s is the size of the queue, and the worst-case (amortized) time for each of these operations is in $\Theta(1)$.

Consider the following new operation, that removes every seventh element from the queue, starting at the beginning, in $\Theta(s)$ time:

```
Function Septimate()
    s ← Size()
    for i ← 0 to s-1 do
        if i mod 7 = 0 then
            Dequeue()
        else
            Enqueue(Dequeue())
```

1. (8 points) Prove that any intermixed sequence of n `Size`, `Enqueue`, `Dequeue` and `Septimate` operations on an initially empty queue runs in $\Theta(n)$ time in the worst case. Hint: use the potential method.

   Solution:

   Suppose the worst-case running time of Septimate() is at most cs where s is the size of the queue before the call to Septimate(). We define $\Phi(Q_i) = 7cs$ where s is the number of elements of the queue after the i^{th} operation on it. Clearly $\Phi(Q_i) \geq 0$, and since the queue is initially empty $\Phi(Q_0) = 0$. Now we analyze the amortized cost of each operation:

   - Size: its real cost is in $\Theta(1)$, and it does not change the potential of the queue, and hence the amortized cost of Size() is in $\Theta(1)$.
   - Enqueue: its real cost is in $\Theta(1)$, and its potential difference is $\Phi(Q_{i+1}) - \Phi(Q_i) = 7c(s+1) - 7cs = 7c$ where s was the size of the queue just before the Enqueue() operation. Hence the amortized cost of Enqueue() is $\Theta(1) + 7c$ which is in $\Theta(1)$.
   - Dequeue: its real cost is in $\Theta(1)$, and its potential difference is $\Phi(Q_{i+1}) - \Phi(Q_i) = 7c(s-1) - 7cs = -7c$ where s was the size of the queue just before the Dequeue() operation. Hence the amortized cost of Dequeue() is $\Theta(1) - 7c$ which is in $\Theta(1)$.
   - Septimate: its real cost is at most cs, and the potential goes down by $7c\lceil s/7 \rceil \geq cs$. Thus the amortized cost of Septimate() is ≤ 0.

   Therefore the worst-case running time of a sequence of n operation is at most

   $$\sum_{i=1}^{n} \max\{\Theta(1), \Theta(1), \Theta(1), 0\}$$

   which is in $O(n)$.

3 Balancing trees using weight instead of height

As you know, binary search trees are fast on average, allowing `search`, `insert` and `delete` operations to run in $O(\log n)$ time, where n is the size of the tree. Unfortunately, some sequences of operations, like inserting n values in increasing order, result in a tree where **every** operation takes $\Theta(n)$ time. Balanced trees, such as AVL trees or B-Trees, avoid this problem and every operation run in $O(\log n)$ time in the worst case. Unfortunately, as most if not all of you will recall unhappily from CPSC 221, the implementations of `insert` and `delete` become much, much more complicated.

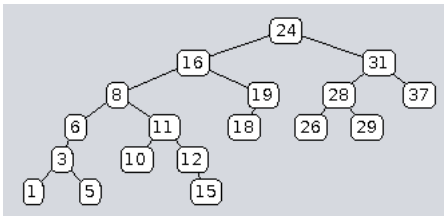
In this question, we consider a different, much simpler way of balancing trees. *Weight-balanced* trees were invented by G. Varghese, and guarantee that a sequence of n operations on an initially empty tree will run in $O(n \log n)$ time. Individual operations may **occasionally** take as long as $\Theta(n)$ time.

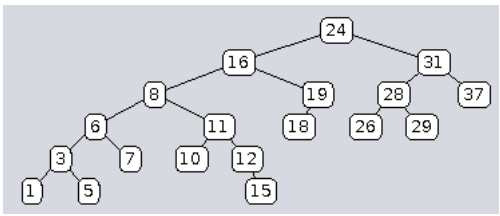
For each node N of the tree, let us denote by size[N] the number of nodes in the subtree rooted at N (including N itself). We will say that N is α-balanced if size[left[N]] $\leq \alpha \cdot$ size[N] and size[right[N]] $\leq \alpha \cdot$ size[N]. For instance, a node will be 1/2-balanced if the sizes of its left and right subtrees differ by at most 1.

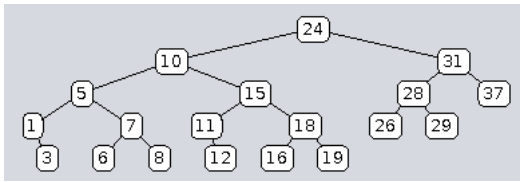
The tree will be called α-balanced if every one of its nodes is α-balanced. The following property holds:

Property 1 *An α-balanced tree has depth at most $\lceil \log_{1/\alpha} n \rceil$.*

For the remainder of this problem, we will assume that $\alpha = 3/4$. The implementations of `insert` and `delete` will remain the same as usual, but after every call to `insert` or `delete`, if one or more nodes in the tree is/are no longer 3/4-balanced, then the subtree rooted at the highest such node will be rebuilt so it becomes 1/2-balanced. For example, the tree on the left of the figure is 3/4-balanced. After inserting 7, we get the tree on the right, which is not 3/4-balanced: the node N with key 16 has 10 nodes in its left subtree, but the subtree rooted at N contains 13 nodes in total, and $10 > 13 \cdot 3/4$. We will thus rebalance this subtree, and end up with the tree on the bottom.







We will analyze this rebuilding scheme using the potential method. For a node N in the binary search tree T, let us define

$$\Delta(N) = |\text{size[left[}N\text{]]} - \text{size[right[}N\text{]]}| \,.$$

We then define the potential of T as

$$\Phi(T) = 2 \left(\sum_{N \in T : \Delta(N) \geq 2} \Delta(N) \right)$$

That is, the potential of the tree is obtained by adding the Δs at each node, but only for the Δs that are 2 or larger, and multiplying the sum by 2.

1. (3 points) A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node N in an arbitrary binary search tree, show how to rebuild the subtree rooted at N so that it becomes 1/2-balanced. Your algorithm should run in time $O(\text{size}[N])$ and it can use $O(\text{size}[N])$ additional space.

   Solution:

   We use the fact that an inorder traversal starting at node N will return the keys stored in the subtree rooted at N, in increasing order:

   - We perform an inorder traversal from N, and store each key in an array.
   - We then rebuild the subtree rooted at N as follows, by placing the median of the array at the root, and the smaller (larger) elements in its left (right) subtree:

```
Function makebalancedtree(A, first, last)
  if first ≤ last then
    mid ← ⌊ (first + last) / 2 ⌋
    N2 ← new node with key A[mid]
    left[N2] ← makebalancedtree(A, first, mid - 1)
    right[N2] ← makebalancedtree(A, mid + 1, last)
    return N2
  endif
  return null
```

2. (8 points) Assume that the real cost of rebuilding a subtree with k nodes is at most k. Prove that the `insert` operation has an amortized cost in $O(\log n)$ (the analysis for the `delete` operation would be more or less the same, and we won't ask you to do it).

   Hint: consider the amortized cost of the rebalancing (if there is one) separately from the amortized cost of the actual insertion.

   Solution: Let us first show that the amortized cost of a rebalancing operation is ≤ 0. Suppose that we rebalance a subtree with k nodes, rooted at a node N. This only occurs if N wasn't 3/4-balanced. Thus, before the rebalancing, one of N's two subtrees contained more than $3k/4$ nodes, with the other subtree therefore containing fewer than $k/4-1$ nodes (the -1 term is for N itself). Hence, before the rebalancing, we had

   $$\Delta(N) \geq 3k/4 - (k/4 - 1) = k/2 + 1 > k/2$$

   Thus the potential in the subtree rooted at N was at least $2 \cdot k/2 = k$. After the rebalancing, the potential in that subtree is 0. Hence the potential difference was $\leq -k$. Therefore the real cost k plus the potential difference is ≤ 0.

   Let us now consider an `insert` operation. By Property 1, its real cost is in $O(\log_{4/3}(n+1))$, as there are at most $\log_{4/3}(n+1)$ nodes on the path from the root to the new node. Moreover, the Δ of each of them increases by at most 1. Thus the potential goes up by at most 2 for each node on the path from the root to the new node (the Δ only goes up by 1, but if it goes from 1 to 2 then it suddenly starts being included in the potential). Therefore the total potential increase is at most $2\log_{4/3}(n+1)$. This means that the amortized cost of the `insert` operation is in $O(\log_{4/3} n)$. The rebalancing (if it happens) does not add anything to the amortized cost.

   Finally, we can prove that the amortized cost of a `delete` operation is in $O(\log_{4/3} n)$ using exactly the same argument.

4 More fun with Stacks

In worksheet 9, we implemented a stack using a pair of queues. Consider now a new data structure that combines properties of both stacks and queues, which we will call a Duck[1]. It can be viewed as a list of elements written left to right such that three operations are possible:

- `DuckPush(x)`: add a new item `x` to the left end of the list;
- `DuckPop()`: remove and return the item on the left end of the list;
- `DuckPull()`: remove the item on the right end of the list.

1. (6 points) Show how to implement a duck using three stacks which we will call `L`, `R` and `TMP`, and $O(1)$ additional memory. In particular, each element in the Duck must be stored in exactly one of `L`, `R` and `TMP`. The amortized costs for any `DuckPush`, `DuckPop`, or `DuckPull` operation should be in $O(1)$. You cannot access the elements of `L`, `R` and `TMP` except through the functions `Push` and `Pop`.

   Give pseudo-code for each of the three operations. Hint: you want *most* calls to `DuckPop` and `DuckPull` to need only $O(1)$ steps.

   Solution: First, as promised, here is the explanation for the name of the data structure. This problem was taken from Jeff Erickson's web site where the data structure is called a *Quack* (the name is a combination of the words *Queue* and *Stack*). However using that name made it too tempting to google the answer, and ducks quack, so I decided to rename the data structure to *Duck*.

   Now onto the ansewr: the idea is that we will normally keep half the elements of the Duck on stack `L`, with the newest element (most recently inserted) at the top of the stack, and the other half of the elements on stack `R`, with the oldest element at the top of the stack. So `L` will act like a stack, while `R` will behave like a queue. We will also keep track of the size of each of stacks `L` and `R`.

   Insertions will happen on `L`. If we are trying to extract an element from the Duck, and the stack we would like to pop from (`L` for `DuckPop`, `R` for `DuckPull`) is empty, then we use the `TMP` stack to move *half* the elements of the Duck to the empty stack. This ensures we won't need to move elements between stacks too often (we will need to `DuckPop` or `DuckPull` half the elements of the Duck before we need to move elements again).

   Here is the implementation:

```
DuckPush(x):
    L.push()

DuckPop():
    if L.size = 0 then
        n ← R.size
        for i ← 0 to ⌊n/2⌋
            TMP.push(R.pop())
        for i ← 0 to ⌈n/2⌉
            L.push(R.pop())
        for i ← 0 to ⌊n/2⌋
            R.push(TMP.pop())
    return L.pop()

DuckPull():
    if R.size = 0 then
```

[1] The reason for the name will be explained in our solutions.

```
        n ← L.size
        for i ← 0 to ⌊n/2⌋
            TMP.push(L.pop())
        for i ← 0 to ⌈n/2⌉
            R.push(L.pop())
        for i ← 0 to ⌊n/2⌋
            L.push(TMP.pop())
    return R.pop()
```

2. (8 points) Using the potential method, prove that every sequence of n operations on an initially empty Duck (where each operation is one of `DuckPush`, `DuckPop`, or `DuckPull`) runs in $O(n)$ time. Hint: with my answer to question 1, the potential function $\Phi(D_i) = 3\max\{\textit{size of } L, \textit{size of } R\}$ works if I consider moving an element from one stack to another to be a single step.

   Solution:

   As stated in the hint, we will use $\Phi(D_i) = 3\max\{\textit{size of } L + \textit{size of } R\}$. We start by computing amortized costs for each of `DuckPush`, `DuckPop` and `DuckPull`.

   Let us first consider `DuckPush`. Its real cost is 1. The potential difference will be either 0 or 3, depending on whether or not $\max\{\textit{size of } L + \textit{size of } R\}$ increases (if it does, it will only increase by 1). So

$$\begin{aligned} cost_{am}(\texttt{DuckPush}) &= cost_{real}(\texttt{DuckPush}) + (\Phi(D_i) - \Phi(D_{i-1})) \\ &\leq 1 + 3 \\ &= 4 \end{aligned}$$

   Now we consider `DuckPop`. The real cost of popping stack 1 is 1, and this reduces $\max\{\textit{size of } L + \textit{size of } R\}$ by at most 1. So, in the case where `L` wasn't empty, the amortized cost is at most $1+3=4$. But what about the case where `L` was empty? Suppose `R` contained n elements. Then

   - The real cost of moving the elements is $\lfloor n/2 \rfloor + \lceil n/2 \rceil + \lfloor n/2 \rfloor$.
   - The potential *before* the move is $3n$ (recall that `L` is empty, so $\max\{\textit{size of } L + \textit{size of } R\} = n$). The potential *after* the move is $3\textit{size of } L = 3\lceil n/2 \rceil$.
   - So the amortized cost encurred by moving elements is

$$\begin{aligned} cost_{am}(move) &= cost_{real}(move) + (\Phi(D_i) - \Phi(D_{i-1})) \\ &\leq \lfloor n/2 \rfloor + \lceil n/2 \rceil + \lfloor n/2 \rfloor + (3\lceil n/2 \rceil - 3n) \\ &= \lfloor n/2 \rfloor + \lceil n/2 \rceil + \lfloor n/2 \rfloor - (3n - 3\lceil n/2 \rceil) \\ &= \lfloor n/2 \rfloor + \lceil n/2 \rceil + \lfloor n/2 \rfloor - 3\lfloor n/2 \rfloor \\ &= \lceil n/2 \rceil - \lfloor n/2 \rfloor \\ &= \leq 1 \end{aligned}$$

   Therefore, $cost_{am}(\texttt{DuckPop}) \leq 4 + 1 = 5$.

   Finally, the analysis of `DuckPull` is identical to that of `DuckPop`, which means that its amortized cost is also at most 5. Consequently

$$\sum_{i=1}^{n} cost_{real}(op_i) \leq \sum_{i=1}^{n} cost_{am}(op_i) \leq 5n$$

   which means that every sequence of n operations on an initially empty Duck runs in $O(n)$ time.