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.

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?

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.

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.

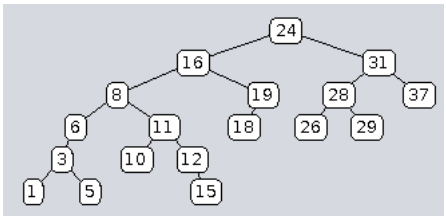
In this question, we consider a different, much simpler way of balancing trees. *Weight-balanced* trees were invented by G. Varghese, and guarantee that a sequence of n operations on an initially empty tree will run in $O(n \log n)$ time. Individual operations may **occasionally** take as long as $\Theta(n)$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

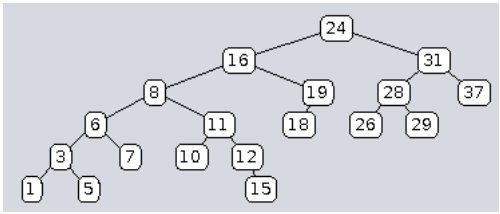
For each node N of the tree, let us denote by size[N] the number of nodes in the subtree rooted at N (including N itself). We will say that N is α-balanced if size[left[N]] $\leq \alpha \cdot$ size[N] and size[right[N]] $\leq \alpha \cdot$ size[N]. For instance, a node will be 1/2-balanced if the sizes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

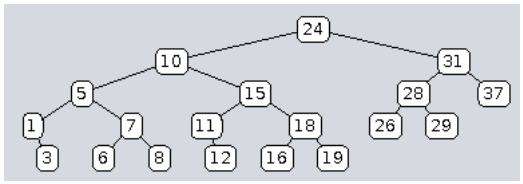
The tree will be called α-balanced if every one of its nodes is α-balanced. The following property holds:

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

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

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.

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.

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.

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.

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