CSE 40113: Algorithms
Homework 0

A note about homework 0: A large goal of this particular assignment is to force you to review some of what was covered in discrete mathematics and in data structures, both of which are important prereqs for this course. You can't design and analyze algorithms without understanding fundamental data structures, proofs, and big-O notation! So expect to pull out your old textbooks if you're rusty on those, or look in the introduction of our textbook for some suggestions, or just email to ask me for some recommendations if you aren't sure where to look.

Academic integrity policy: In this class, while you're welcome to use any reference you'd like, you are responsible for both citing your sources AND re-writing the solution in your own words - including any usage of ChatGPT or similar tools. So feel free to use the internet, but I'd recommend reading, thinking about it, adding a citation for what you read to your homework, and then putting all that away and writing it from memory, so you'll understand the answer properly. Please believe me when I say that ANY verbatim copying will get you a 0 on the homework, as well as being reported to the department. Any second offense will get you a 0 in the class, as well as be forwarded to the college for further disciplinary measures.

Required Problems

1. Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but make sure you understand how you would prove these and what facts/identities are needed to simplify.

$$\begin{array}{ccccc} 1 & n & 3n^2-6n+12 & \lg n & \lg\sqrt{n} \\ \cos n+2 & n^{\lg n} & \lg 2^n & \sum_{i=1}^{n}\sum_{j=1}^{i}1 & 2^{2\lg n} \\ \sqrt{2^{\lg n}} & n! & n\lg n & \sum_{i=1}^{n} i^2 & \lg(\sqrt{2^n}) \end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2, n, $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$. *[Hint: When considering two functions $f(\cdot)$ and $g(\cdot)$ it is sometime useful to consider the functions $\ln f(\cdot)$ and $\ln g(\cdot)$.]*

Solution: The final sorted list is as follows:

$$\cos n + 2 \equiv 1 << \log\sqrt{n} \equiv \lg n << \sqrt{2^{\lg n}} << n \equiv \log(\sqrt{2^n}) \equiv \log 2^n << n\lg n << 3n^2 - 6n + 12 \equiv \sum_{i=1}^{n}\sum_{j=1}^{i} 1 \equiv 2^{2\lg n} << \sum_{i=1}^{n} i^2 << n^{\lg n} << n!$$

■

[Hint: Hopefully this will become obvious, but my goal here is to make you remember: big-O, logarithms, summations, and ignoring constants! Go back and check your discrete math book and that chapter in your data structures book that talked about big-O.]

2. (a) Your fellow student presents the following idea: in order to implement "find the maximum" in constant time, why not use a stack or a queue, but keep track of the maximum value inserted so far, then return that value for "find the maximum". Does this seem like a good idea, or do you see issues?

   Solution: The obvious idea is to keep track of the max, and when inserting, you can easily update in $O(1)$ if a larger value is added. Returning this max is then easy. However, if that item is deleted, there is no way to find the next larger element in the stack or queue without taking $O(n)$ time to search all elements present. In addition to the space, this would also require an auxiliary queue or stack to store the data as you search it, so things are not deleted when searching. So, this idea has definite issues! ■

   (b) Suppose you have been asked to implement a stack, but the only data structure you have available is a queue (although you may use as many queues as you would like). Give pseudocode for implementing stack's push and pop function, which use only the queue(s) you will use to maintain the data. What is the asymptotic runtime of each of your functions, assuming the queue push and pop functions run in $O(1)$ time?

   Solution: One approach is to maintain two queues (Q1 and Q2) at all times, with the invariant that the stack order "top" is the first element in Q2 at all times. When things are added implement push and pop as follows, assuming a standard set of functions for queues which are all $O(1)$ time: enqueue (which adds an element to the "back" of the queue), dequeue (which returns the element at the "front"), and empty, which returns true if the queue has no elements in it.

```
Push(x):
  Q1.enqueue(x)
  while Q2 not empty
      Q1.enqueue(Q2.dequeue)
  Swap(Q1,Q2)
```

   The runtime is $O(n)$, because we must move elements from Q1 to Q2 which involves an $O(1)$ dequeue operation for every element in the queue.

```
Pop():
  if Q2 is empty
      Error ("Stack is empty)
  return Q2.dequeue()
```

   The runtime here is $O(1)$, as we make a single call to empty and a single call to dequeue (as well as possibly raising the error). ■

3. Dr. Chambers recently returned from Germany with a new favorite 24-node binary tree, in which every node is labeled with a unique letter from the German alphabet. (Note that this is pretty similar to English, but adds interesting characters like the umlaut and ß.) She gives you the following traversals:

   - Preorder: B K Ü E H L Z I Ö R C ß T S O A Ä D F M N U G

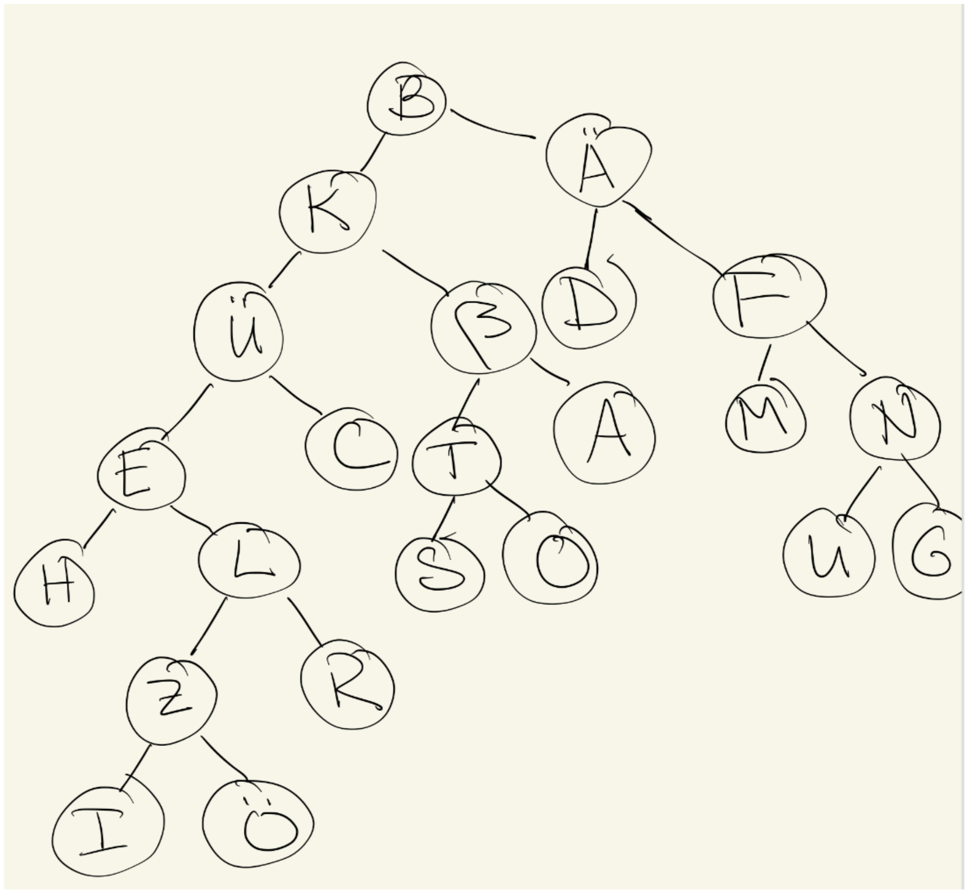
- Postorder: H I Ö Z R L E C Ü S O T A ß K D M U G N F Ä B

(a) List the nodes in an in order traversal of the tree.

Solution: H E I Z Ö L R Ü C K S T O ßA B D Ä M F U N G ■

(b) Draw the tree.

Solution: With apologies for a hand drawn version:



■

4. A binomial tree of order k is defined recursively as follows:

   - A binomial tree of order 0 is a single node.
   - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k - 1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

   Prove the following claims:

   (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

Solution: We prove this by induction on the order of the tree.

Base case: a tree of order 0. This has 1 node, and $2^0 = 1$, so the statement holds for trees of order 0.

Inductive Hypothesis: Assume that for a binomial tree of order $k-1$, there are exactly 2^{k-1} nodes.

Inductive Step: Consider a binomial tree of order k. Based on our recursive construction above, we know this tree is formed from two trees of order $k-1$, where the root of one of the two trees is connected as a new child of the other; no other nodes are added in the construction. This means that every node in a tree of order k came from one of the two smaller trees of order $k-1$, so the total number of nodes is $2^{k-1} + 2^{k-1} = 2 \cdot 2^{k-1} = 2^k$, so the identity also holds for trees of order k. ■

(b) For all positive integers k, attaching a leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

Solution: Again, we use induction on the order of the tree:

Base case: Consider a tree of order 1 as our base case. This can be build via our recursive construction by combining to trees of order 0, with the root of one tree put as as child of the other. Since both are single nodes, this yields a tree with two nodes, as shown in the figure. This is equivalent to taking one tree of order 0 and adding a child to the root, so the claim holds for $k = 1$.

Inductive hypothesis: Assume that a binomial tree of order $k-1$ can be constructed by taking a tree of order $k-2$ and adding a leaf under every node.

Inductive step: Now consider a binomial tree of order k. By our recursive construction, we know this is build from two trees of order $k-1$, where where the root of one of the two trees is connected as a new child of the other.

Consider these smaller trees of order $k-1$, which I'll call T_1 and T_2. By our inductive hypothesis, both T_1 and T_2 could have been constructed by taking trees of order $k-2$, and adding a leaf to every node.

Now, consider removing those leaves for a moment, leaving two trees of order $k-2$. By our recursive definition, this is the way to build a tree of order $k-1$. Now, if we re-attach the leaves to every node, we are back to our order k tree again. (Note that we haven't changed anything here, just we are considering combining the two order $k-2$ trees before adding the leaves to each node.) But, this means that we can construct our order k tree by taking a tree of order $k-1$, and adding a leaf to every node, proving the claim.

■

(c) Prove that for all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d. (Hence the name!)

Solution: A critical step here is to make sure you know what you are inducting on, since we have two variables! We're going to follow the pattern, and look at induction on the order of the binomial tree. An important fact to note that follows immediately from the construction is that a tree of order k has nodes at depth 0 to k, because the recursive construction starts with a tree of order 0 (and depth 0, since it's a single node) adds one to the depth at every level by combining two smaller trees so the depth only increases by 1 total each time.

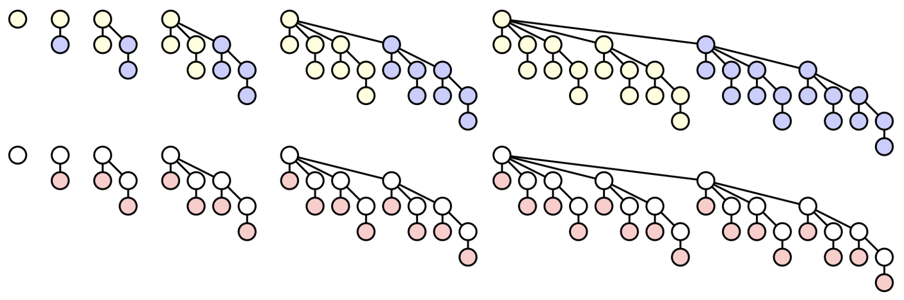
Base case: A binomial tree of order 0 is a single node. It is at depth 0, and $\binom{0}{0} = 0!/0! = 1$. We can also check a tree of order 1: it has a single node at level 1 (the root) and a single leaf at depth 0. In each level respectively, we should have $\binom{1}{0} = 1$ and $\binom{1}{1} = 1$ nodes, so the desired equality is true.

Induction hypothesis: Assume that a binomial tree of order $k-1$ has $\binom{k-1}{d}$ nodes at every depth $d \in \{0, k-1\}$.

Inductive step: Consider a binomial tree of order k, and again use the recursive construction given for these trees. A tree of order k is constructed from two trees of order $k-1$. Call these two smaller trees T_1 and T_2, where the root of T_2 has been attached as a child of the root of T_1. Note that every node in T_2 is now one level deeper in the larger tree, since a single root was added on top of all of them.

Now consider some level d of the larger order k tree. Every node on this particular level came from one of the two smaller trees, since every node in the tree is present in T_1 or T_2. In this case, since it's on level d of the larger tree, it came from either level d of T_1 (since those nodes have not changed depth) or came from level $d-1$ of tree T_2. By the inductive hypothesis, we know that there are $\binom{k-1}{d}$ nodes at depth d in T_1, and $\binom{k-1}{d-1}$ nodes at depth $d-1$ in T_2. Therefore, we have $\binom{k-1}{d} + \binom{k-1}{d-1}$ nodes total at depth d in the order k tree.

The final step in this proof is showing that this sum is equal to $\binom{k}{d}$; this can be proven by evaluating the factorials and canceling, by a combinatorial proof, or by citing Pascal's identity from your discrete math reference. ■


Binomial trees of order 0 through 5.
Top row: the recursive definition. Bottom row: the property claimed in part (b).