CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 0

Harvard SEAS - Fall 2024 *Due: Wed 2024-09-11 (11:59PM)*

The purpose of this problem set is to reactivate your skills in proofs and programming from CS20 and CS32/CS50. For those of you who haven't taken one or both those courses, the problem set can also help you assess whether you have acquired sufficient skills to enter CS1200 in other ways and can fill in any missing gaps through self-study. Even for students with all of the recommended background, this problem set may still require a significant amount of thought and effort, so do not be discouraged if that is the case and do take advantage of the staff support in section and office hours.

For those of you who are wondering whether you should wait and take CS20 before taking CS1200, we encourage you to also complete the CS20 Placement Self-Assessment. Some problems there that are of particular relevance to CS1200 are Problems 2 (counting), 4 & 5 (comparing growth rates), 9 (quantificational logic), and 12 (graph theory).

Written answers must be submitted in pdf format on Gradescope. Although LaTeX is not required, it is strongly encouraged. You may handwrite solutions so long as they are fully legible. The `ps0` directory, which contains your code for problems 1a and 1c, must be submitted separately to an autograder on Gradescope. Please fill out the "GitHub Username Submission" assignment on Gradescope to be added to the GitHub Classroom assignment - in the meantime, you can accept the GitHub Classroom assignment, or pull the starter code from the cs1200 GitHub repository.

1. (Binary Trees)

   In the cs1200 GitHub repository, we have given you a Python implementation of a binary tree data structure, as well as a collection of test trees built using this data structure. We specify a binary tree by giving a pointer to its *root*, which is a special *vertex* (a.k.a. *node*), and giving every vertex pointers to its *children* vertices and its *parent* vertex as well as an identifying *key*:

```
class BinaryTree:
    def __init__(self, root):
        self.root: BTvertex = root

class BTvertex:
    def __init__(self, key):
        self.parent: BTvertex = None
        self.left: BTvertex = None
        self.right: BTvertex = None
        self.key: int = key
        self.size: int = None
```

   In CS50, the concept of a Python `class` was not covered. Here, with `BinaryTree` and `BTvertex`, we are using them in the same way as a `struct` in C. An object `v` of the `BTvertex` class contains five attributes, which we list with the type of the object we expect to be named by each attribute (using the Python type annotation syntax). These attributes can be accessed as `v.parent`, `v.left`, `v.right`, `v.key`, and `v.size`. For example, `v.left.key` is the key associated with `v`'s left child. An object of the `BinaryTree` class contains only one attribute, which is the `BTvertex` object that is the root of our binary tree. You can create a `BinaryTree` object as follows:

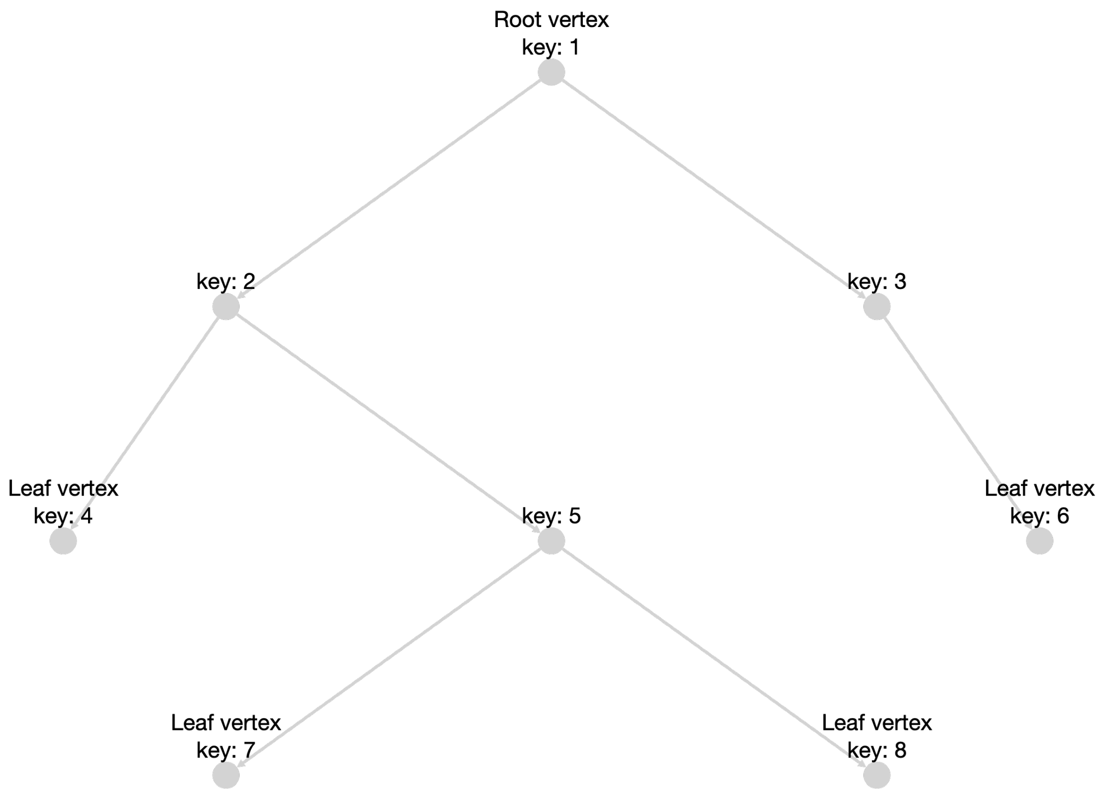
```
root = BTvertex(120)
tree = BinaryTree(root)
tree.root.left = BTvertex(121)
tree.root.right = BTvertex(124)
```

You can then print attributes of the newly created `BinaryTree` object:

```
print(tree.root.key)
>> 120
print(tree.root.left.key)
>> 121
```

Classes are more general than structs because they can also have private attributes and methods that operate on the attributes, allowing for object-oriented programming. However, you won't need that generality in this problem set.

Here is an instance `T` of `BinaryTree`:



A `BinaryTree T` contains only a pointer to its root vertex, `T.root`, which is required to satisfy `T.root.parent==None`. In the above example, the root is the vertex with key 1 (i.e. `T.root.key==1`). A binary tree vertex `v` can have zero, one, or two children, determined by which of `v.left` and `v.right` are equal to `None`. In the above example, the vertex `v` with key 3 has `v.left==None` but `v.right` is the vertex with key 6. A *leaf* is a vertex with zero children, i.e. `v.left==v.right==None`.

A vertex `w` is *descendant* of a vertex `v` if there is a sequence of vertices $\texttt{v}_0, \texttt{v}_1, \ldots, \texttt{v}_k$, $k \in \mathbb{N}$ such that $\texttt{v}_0 = \texttt{v}$, $\texttt{v}_k = \texttt{w}$, and $\texttt{v}_i \in \{\texttt{v}_{i-1}.\texttt{left}, \texttt{v}_{i-1}.\texttt{right}\}$ for $i = 1, \ldots, k$.[1] In the above example, the vertex with key 5 is a descendant of the root (with a path of length 2), but is not a descendant of the vertex with key 3. The sequence $\texttt{v}_0, \texttt{v}_1, \ldots, \texttt{v}_k$ is called a *path* from `v` to `w` and k is the *distance* from `v` to `w`. Taking $k = 0$, we see that `v` is a descendant of itself.

The *vertex set* of a binary tree `T` consists of all of the descendants of `T.root`. The *size* of `T` is its number of vertices. The *height* of `T` is the largest distance from the root to a leaf. The above example has size 8 and height 3.

Given any vertex `v` in a tree, the *subtree* rooted at `v` consists of all of `v`'s descendants. Note that we can remove a subtree and turn it into a new tree `S` by setting `S.root=v` and `v.parent=None`.

For now, the `key` attribute serves to distinguish vertices from each other in our tests and help illustrate what the algorithms are doing. The `BTvertex` class also has a `size` attribute, which is initialized to `None` in all of the test instances; it will be filled in by the program you write in Part 1a.

An instance `T BinaryTree` is *valid* if it satisfies the following constraints:

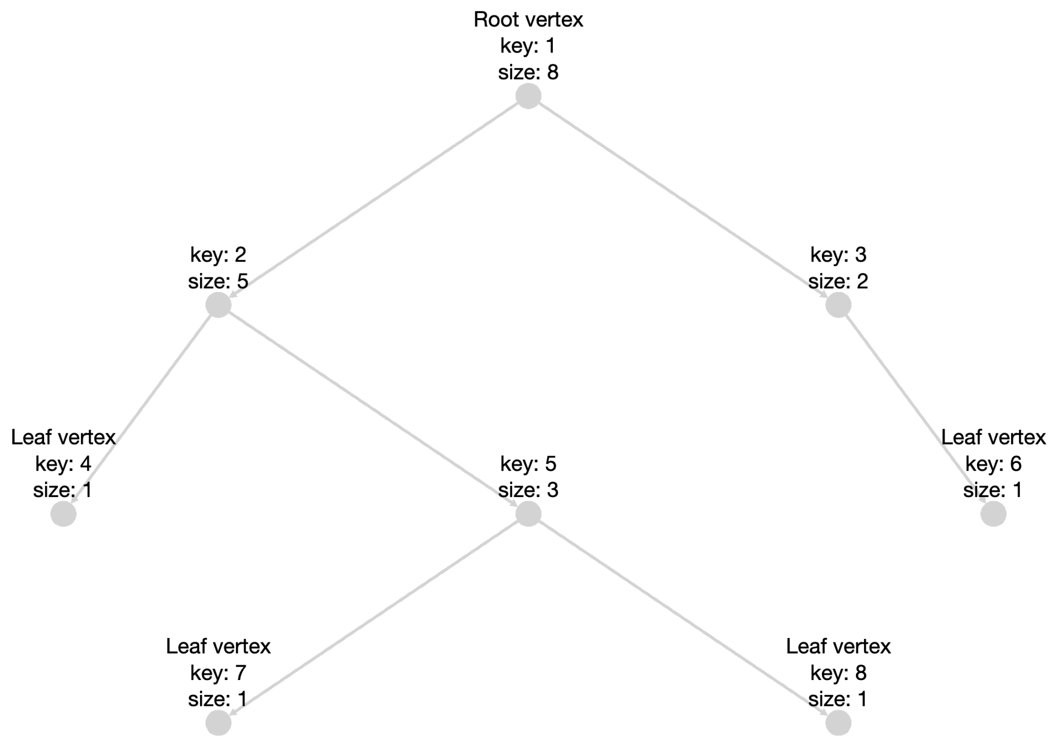
- `T.root.parent==None`
- `T` has finitely many vertices.
- No two vertices `v`, `w` of `T` share a child, i.e. $\{\texttt{v.left}, \texttt{v.right}\} \cap \{\texttt{w.left}, \texttt{w.right}\} = \emptyset$.

All of the test instances we provide are valid, and furthermore have the property that all of the vertices have distinct keys (which is something we often want, but not always).

(a) (recursive programming) Write a recursive program `calculate_sizes` that given a vertex `v` of a binary tree `T`, calculates the sizes of all of the subtrees rooted at descendants of `v`. After running your program on `T.root`, every vertex `v` in `T` should have `v.size` set to the size of the subtree rooted at `v`. (Recall that the size attributes are initialized to `None`.) We call the resulting tree a *size-augmented* tree.

For example, if `T` is the tree shown above, then calling `calculate_sizes(T.root)` should modify `T` to be the following size-augmented tree:

[1] $\mathbb{N}$ denotes the natural numbers $\{0, 1, 2, 3, \ldots\}$. Since we are computer scientists, we start counting at 0.



Your program should run in time $O(n)$ when given the root of a tree with n vertices. In a sentence or two, informally justify why your program has such a runtime.

—

This program has a runtime of $O(n)$ because we are essentially just visiting each node once.

—

(b) (proofs by induction) Let t be a positive integer. Prove that every binary tree `T` of size at least $2t + 1$ has a vertex `v` such that the subtree rooted at `v` has size at least t and at most $2t$. (Hint: use induction on the size n of `T`, starting with base case $n = 2t + 1$. Remember that nodes in binary trees can have 2, 1, or 0 children.)

———

We can prove this by induction. We start with our base case of when the size of `T` is $2t + 1$. In this case, given that we are looking at the *subtree* v of `T`, the root of `T` takes up 1 node. Therefore, among the immediate subtrees, we have that their sizes must sum to $2t + 1 - 1 = 2t$. In the simplest case, when branch of the root of the tree has a size of 0, the other branch must have size $2t$. Now, we must consider how a subtree must exist such that t is the minimum size.

Consider that we have subtrees s_0 and s_1. If we were to split the size evenly, each subtree would have a size of $2t/2 = t$. So, we have (let size be a function that gives the size of a tree):

$$size(s_0) = t$$

$$size(s_1) = t$$

Now, given that we have a finite size, if we were to move nodes over to another subtree, we would have (where $a \leq t$):

$$size(s_0) = t - 1a$$
$$size(s_1) = t + 1a$$

Therefore, it becomes clear that there must exist some subtree where the size is $\geq t$.

Now, we can take the $n+1$ which is $2t+2$. We can show that there must exist a subtree v where the size is $\geq t$ and $\leq 2t$. When the tree only has one chid, the child node is the root of a subtree that is size $2t+1$ which is our base case. In the case where there are two children, the minimum size each could be is $\frac{2t+2}{2} = t+1$. This is between t and $2t$ which satisfies our condition. Therefore, the thereom is proven in the $n+1$ case.

(c) (from proofs to algorithms) Turn your proof from Part 1b into a Python program `FindDescendantOfSize(t,v)` that, given a positive integer `t` and a vertex `v` of a *size-augmented* tree `T` such that `v.size` $\geq$ `2t+1`, finds and returns a descendant `w` of `v` such that `t` $\leq$ `w.size` $\leq$ `2t`. Your algorithm should run in time $O(h)$, where h is the height of the subtree rooted at `v`; explain in words why this is the case.

2. (asymptotic notation) Recall the definitions of asymptotic notation from CS20: Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}^+$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the natural numbers (including 0, since we are computer scientists) and $\mathbb{R}^+$ denotes the positive real numbers.

- We say that $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large n.
- We say that $f = o(g)$ if for *every* constant $c > 0$, $f(n) \leq c \cdot g(n)$ for all sufficiently large n; equivalently, $\lim_{n\to\infty}(f(n)/g(n)) = 0$.

(a) For each of the following pairs of functions, determine whether $f = O(g)$ and whether $f = o(g)$. Justify your answers.

i. $f(n) = 3\log_2^3 n$, $g(n) = n^2 + 1$.

We can show that $f = o(g(n))$ and $f = O(g(n))$ by showing $f = o(g(n))$ because $f = o(g(n)) \implies O(g(n))$. To do this, we can use L'Hopital's rule to show that the asymptotic growth rate of $g(n)$ is greater than $f(n)$:

$$\lim_{n\to\infty} \frac{\frac{d}{dn}(3log_2(n+1))^3}{\frac{d}{dn}n^3}$$

$$\lim_{n\to\infty} \frac{3(3log_2(n+1))^2 * \frac{1}{3n\ln 2}}{3n^2}$$

$$\lim_{n\to\infty} \frac{3log_2(n+1)^2 * \frac{1}{\ln 2}}{3n^3}$$

From here, we can see that, n^3 grows faster than n^2 causing the expression to go to 0, so therefore, $f = o(g(n))$.

---

ii. $f(n) = 4n^3$, $g(n) = |\{S \subseteq [n] : |S| \leq 3\}|$, where $[n] = \{0, 1, 2, \ldots, n-1\}$.

iii. $f(n) = 5^n$, $g(n) = n!$.

For this, we show that $f = o(g(n))$. We can start in the case that $n = 1$. We know that in the $n+1$ case, we have:

$$f(n+1) = 5^{n+1} = 5 * 5^n$$

$$g(n+1) = (n+1)! = (n+1) * n!$$

We can see from this that while $f(n)$ grows by a factor of 5 as n incrases, $g(n)$ grows by a factor of $n+1$. This means that as n grows towards infinity, the growth of $g(n)$ is much greater than that of $f(n)$ no matter regardless of any constant factor that could be multiplied to $f(n)$. Therefore, $f = o(g(n))$.

(b) Prove or disprove: For all functions $f, g : \mathbb{N} \to \mathbb{R}^+$, $f = O(g) \Rightarrow g \neq o(f)$.

3. (reflection: excitement and apprehension) On every problem set in cs1200, there will be a qualitative reflection question, usually about some aspect of your engagement with the course (but sometimes about the course content itself). Your answers to these questions and the in-class sender-receiver exercises will be the basis of your participation grade. For starters, every thoughtful answer with some specifics will receive an R grade; R+ will be reserved for exceptional ones and grades below R for responses that are incomplete or generic.

   (Re)watch the course overview video and (re)read the syllabus. What aspect of cs1200 are you most excited about and what aspect are you most apprehensive about? What efforts can you make as a student to make the most out of your excitement, and to help address your apprehension?

   I am excited to formally learn about algorithms and the sort of thinking that goes into implementing them. Even the process of proving certain concepts related to algorithms is an interesting method of thinking that has applications beyond just CS. I will try my best to keep up with the books especially as that is usually my primary method of engaging with learning material.

   ---

4. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!