CS120: Intro. to Algorithms and their Limitations	Hesterberg & Anshu

Problem Set 0

Harvard SEAS - Fall 2023 — *Due: Wed 2023-09-13 (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 CS120 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 CS120, we encourage you to also complete the CS20 Placement Self-Assessment. Some problems there that are of particular relevance to CS120 and are complementary to what is covered below are Problems 2 (counting), 4 (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. Be sure to pull the starter code from the cs120 GitHub repository.

1. (Binary Trees) In the cs120 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:

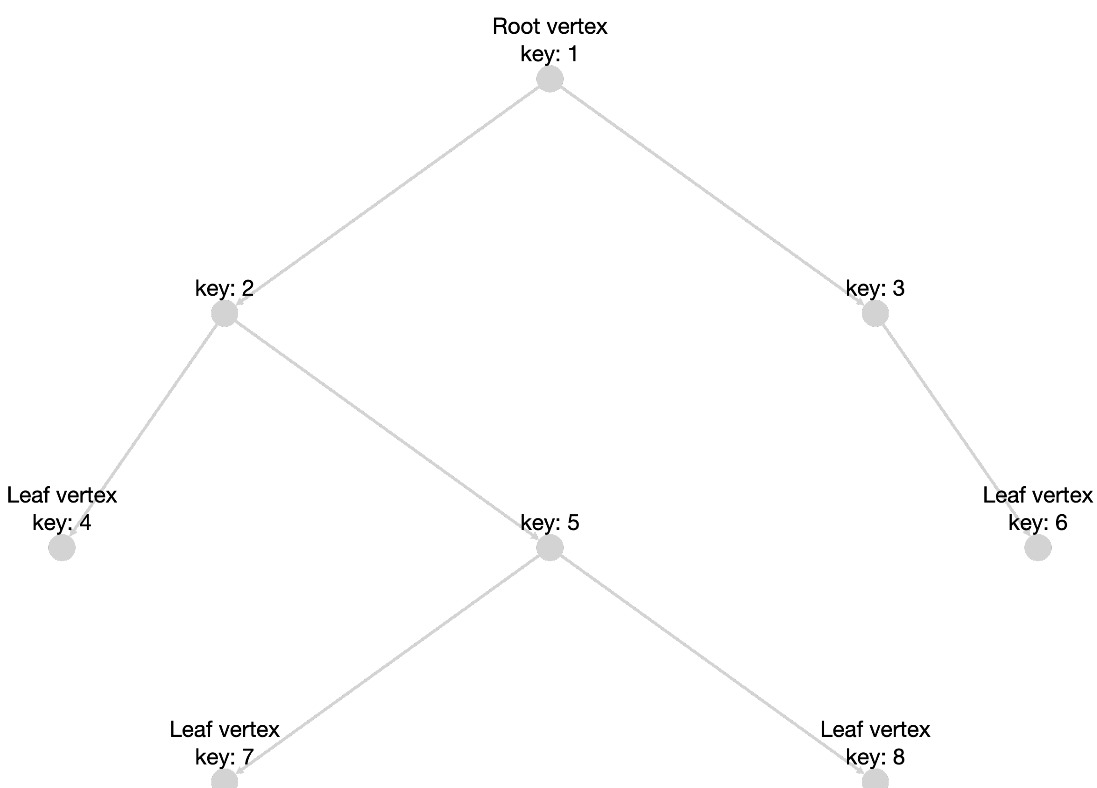
```
root = BTvertex(120)
tree = BinaryTree(root)
tree.root.left = BTvertex(121)
tree.root.right = BTvertex(124)
```

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

The *vertex set* of a binary tree `T` consists of all of the descendents 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 descendents. 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:

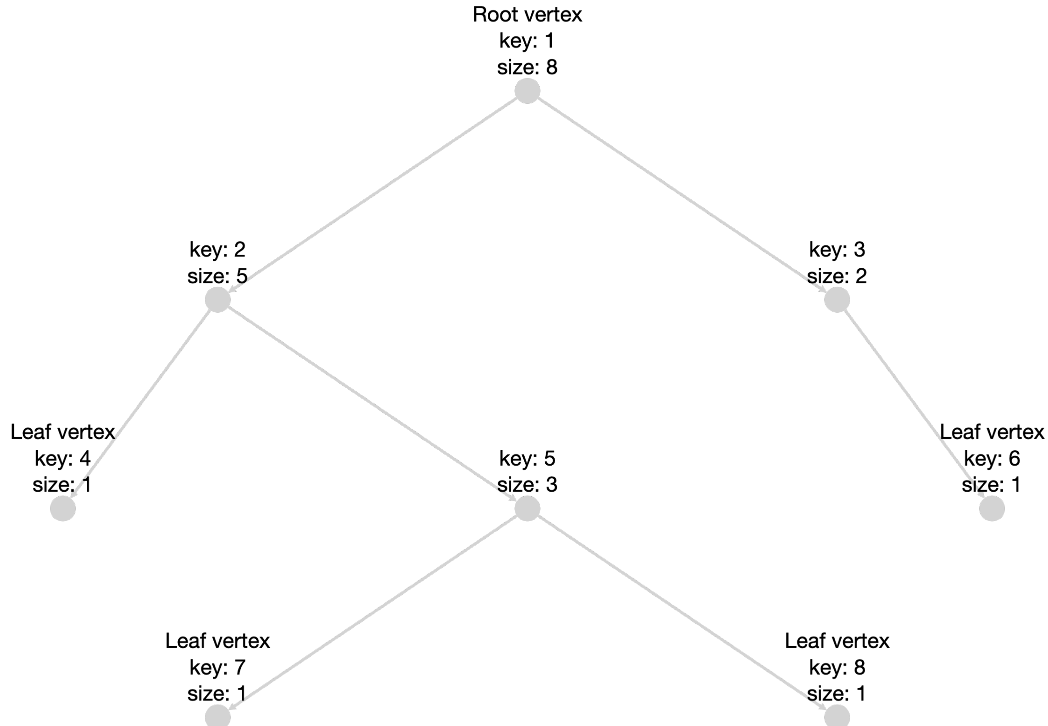
- `T.root.parent==None`
- `T` has finitely many vertices.
- No two vertices `v`, `w` of `T` share a child, i.e. $\{\mathtt{v.left}, \mathtt{v.right}\} \cap \{\mathtt{w.left}, \mathtt{w.right}\} = \emptyset$.

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 descendents 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:

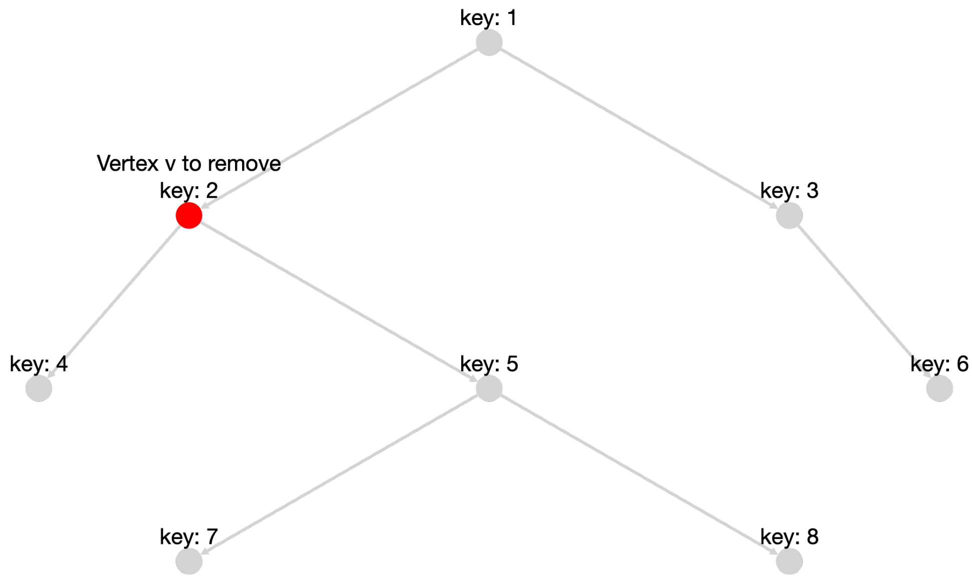
[1] $\mathbb{N}$ denotes the natural numbers $\{0, 1, 2, 3, \ldots\}$. Since we are computer scientists, we start counting at 0.



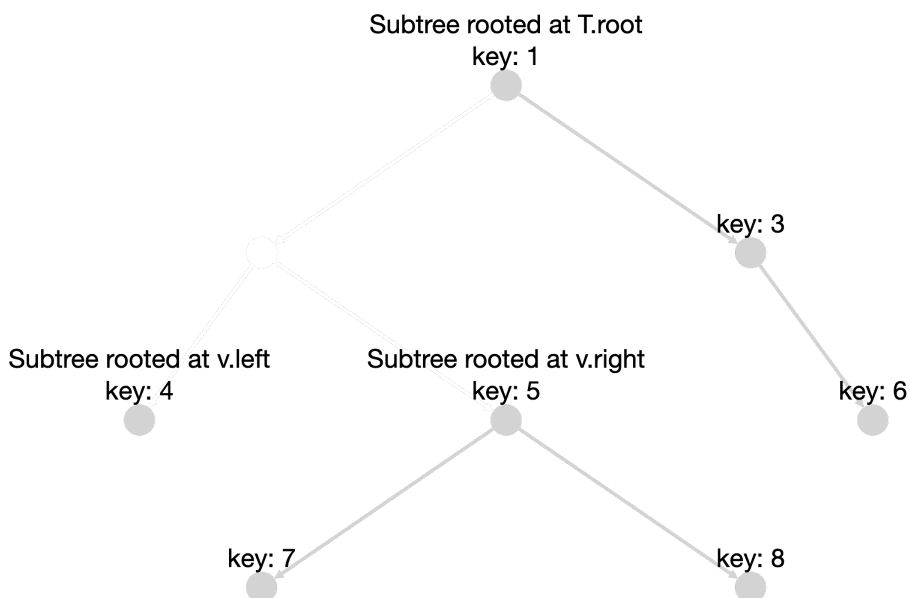
Your program should run in time $O(n)$ when given the root of a tree with n vertices. In a sentence or two, informally justify why your program has such a runtime.

(b) (proof warmup) Removing a vertex `v` from a tree `T` yields up to three disjoint trees: the subtree rooted at `v.left` (unless `v.left==None`), the subtree rooted at `v.right` (unless `v.right==None`), and a tree rooted at `T.root` consisting of all non-descendants of `v` (unless `T.root==v`). For example:

Before:



After:



Suppose that for a vertex `v`, the largest subtree created by removing `v` contains a neighbor (i.e. child or parent) $\mathtt{v}^*$ of `v` and a total of $\phi(\mathtt{v})$ vertices (including $\mathtt{v}^*$). If we remove $\mathtt{v}^*$ and not `v`, we create at most two subtrees which don't contain `v`: prove that these subtrees each contain fewer than $\phi(\mathtt{v})$ vertices.

(c) (proofs by contradiction) Prove that in every tree `T` of size n, there exists a vertex `v` such that removing `v` from `T` results in disjoint trees that all have size at most $n/2$.

You may prove this however you like, but a recommended approach is to extend Part 1b and show that if the largest subtree created by removing a vertex `v` contains a neighbor `v`* of `v` and contains ϕ(`v`) $> n/2$ vertices, then if we remove `v`* and not `v`, the subtree *containing* `v` contains at most $n/2$ vertices. Then choose `v` to be the vertex for which ϕ(`v`), the size of the largest tree created by removing `v`, is smallest.

(d) (from proofs to algorithms) Turn your proof from Part 1c into a Python program that given a root vertex `r` of a *size-augmented* tree `T` with n vertices finds a vertex `v` with ϕ(`v`) $\leq n/2$. Your program should run in time $O(h)$ on all size-augmented trees of height h; again informally justify why your program has such a runtime. (Hint: try to repeatedly reduce the potential function by moving to children. Why don't we need to try moving to parents as in the previous proof?)

2. (matchings and induction) Later in the course, we will study matching algorithms that are used in practice to match kidney donors to patients. The challenge in general is that some donors are incompatible with some patients (i.e. the patient's body is likely to reject the donated kidney). Suppose we are very lucky and have n donors and n patients where each donor d is incompatible with exactly one patient, denoted $\mathit{incomp}(d)$, and each patient p is incompatible with exactly one donor $\mathit{incomp}(p)$. (Incompatibility is symmetric, so $\mathit{incomp}(d) = p$ iff $\mathit{incomp}(p) = d$.) Let $f(n)$ be the number of ways, under these circumstances, of matching donors to patients so that each donor donates exactly one kidney to a compatible patient and each patient receives exactly one kidney from a compatible donor.

(a) Show that for all $n \geq 3$,

$$f(n) > (n-1) \cdot f(n-2).$$

Hint: let d be one of the donors, and consider all possible patients p with whom d could be matched. Then consider the case where $\mathit{incomp}(p)$ is matched with $\mathit{incomp}(d)$.

(b) In fact, show that for all $n \geq 3$, we have

$$f(n) = (n-1) \cdot (f(n-1) + f(n-2)).$$

This will require you to include the remaining case where $\mathit{incomp}(p)$ is *not* matched with $\mathit{incomp}(d)$, unlike the previous exercise.

(c) Prove by strong induction that for all $n \geq 2$,

$$\frac{n!}{3} \leq f(n) \leq \frac{n!}{2}.$$