10B—Constraints and Degrees of Freedom

As Engineers, this should be really comfortable for you.

Constraints and Degrees of Freedom

- We all know what constraints and degrees of freedom are in a mechanics problem, but what are they in a data structure/ algorithm?
- A constraint is an implied condition that must be satisfied in a data structure/algorithm.
- A degree of freedom is an option in a data structure/algorithm, i.e. where there is a *choice*.
- # constraints + # degrees of freedom = (generally) constant in a stable system.

A simple data structure

- The simplest data structure that we can come up with is a (reference to a) singleton object.
- No choices (DFs), no constraints.

A slightly more complex structure

- An object, a reference to another object, and the second object makes up a data structure.
- We can create an algorithm to traverse this data structure: it starts by pointing to the first object, then it follows the reference (pointer, link) to the second object.
- DFs: none
- Constraints: The first object has 1 pointer, the second has 0 pointers.

A linked list

- Now, suppose that every object that we have can have one (or none) pointers.
- This allows us to arrange our data in a linked list.
- This structure has one degree of freedom: its length.
- The constraint is that an object has one or zero pointers.
- This is getting a bit more useful!
- BTW, we might need additional constraints, for example, to prevent a loop, we say that a link (pointer) cannot reference any object from which we can reach this object.

A tree

- Now, let’s suppose that we can have any number of pointers in a node.
- And the depth of the tree (like the length of a linked list) is also free to take any value.
- Now, we’ve got an awesomely flexible data structure (many degrees of freedom, no constraints).
- But is it really useful? If we wanted to search for an object in this tree, is it going to be any quicker than searching for an object in a linked list?

A tree

- Now, let’s suppose that we can have any number of pointers in a node.
- And the depth of the tree (like the length of a linked list) is also free to take any value.
- Now, we’ve got an awesomely flexible data structure (many degrees of freedom, no constraints).
- But is it really useful? If we wanted to search for an object in this tree, is it going to be any quicker than searching for an object in a linked list?
- No.

A more useful tree

- Let’s add some constraints to this tree.
- Constraint #1: an object may only point to zero, one or two objects (we call this a binary tree).
- Constraint #2: an object has some “key” *k* and (possibly) some ”value” *v*.
- Constraint #3: let’s say an object *Y* has two pointers (“child pointers”) to objects *X* and *Z*. $k_x <= k_y <= k_z$.
- Now, when we traverse the tree, searching for an element *Q*, and we are at node *Y*, there are three possibilities (and only three):
    - $k_q == k_y$ (we’re done)
    - $k_q < k_y$ (we only need to continue searching in the tree whose root is *X*)
    - $k_q > k_y$ (we only need to continue searching in the tree whose root is *Z*)
- This is the perfect use case for the *compareTo* method in Java.

And the consequence is…

- Instead of having to search every one of *N* elements (nodes) in the tree, we only have to search *h* elements where *h* is the depth (also called height) of the tree.
- What is *h*?

And the consequence is…

- Instead of having to search every one of *N* elements (nodes) in the tree, we only have to search *h* elements where *h* is the depth (also called height) of the tree.
- What is *h*?
- *h = lg N*
- So, by storing our *N* elements into a (binary) tree--with the constraints mentioned before—we have reduced our search time from *N* to *lg N*. That's *huge*!!!
- Essentially, we've used a *property* of the nodes (their key) to make good decisions on how to traverse the tree.

OK, now you know all that, you can go home.

JK