Notes, 2(c)

ECE 606

Data Structures – Stacks, Queues, Binary Search Trees

Stack: can think of methods/algorithms associated with it.

- newStack(): creates a new empty stack and returns it.
- stackIsEmpty(S): returns true if the stack S is empty, false otherwise.
- push(i, S): pushes the item i onto the stack S.
- pop(S): pops the topmost item off the stack S and returns it. If S is empty, the behaviour is undefined.

Queue:

- newQueue(): creates a new empty queue and returns it.
- queueIsEmpty(Q): returns true if the queue Q is empty, false otherwise.
- enqueue(i, Q): enqueues the item i onto the queue Q.
- dequeue(Q): dequeues the frontmost item off the stack Q and returns it. If Q is empty, the behaviour is undefined.

Natural question: is one data structure somehow more "expressive" than another.

- We can consider a data structure A to be at least as expressive as a data structure B if, given A, we can realize B.
- A is strictly more expressive if, in addition, given B, we cannot realize A.

E.g., is Stack at least as expressive as Queue? If yes, is it strictly more expressive?

Answer to first question: yes. To second: no. Proofs by construction.

Given Stack, to realize Queue, we need to realize methods associated with Queue using those associated with Stack.

```
newQueue()
    return newStack()

enqueue(i, Q)
    T ← newStack()
    while stackIsEmpty(Q) = false do
        j ← pop(Q)
        push(j, T)
    push(i, Q)
    while stackIsEmpty(T) = false do
        j ← pop(T)
        push(j, Q)

queueIsEmpty(Q)
    return stackIsEmpty(Q)

dequeue(Q)
    return pop(Q)
```

There is a cost: enqeueue slows linearly in the number of items in the queue.

Similarly, we can ask:

- Given Array, can we realize Queue?
- Given Queue, can we realize Array?
- Given Set, can we realize Queue?
- Given Map, can we realize Set?
- For each of the above, if the answer is yes, what is the cost?
- In Java: TreeSet vs. HashSet, what trade-offs does each incur?

Good thing to keep in mind with algorithms, and really, all of engineering and life: there is always a cost. The question is only whether the reward outweighs the cost.

See textbook for Stack & Queue from Array, Linked Lists, Pointers, Rooted Trees.

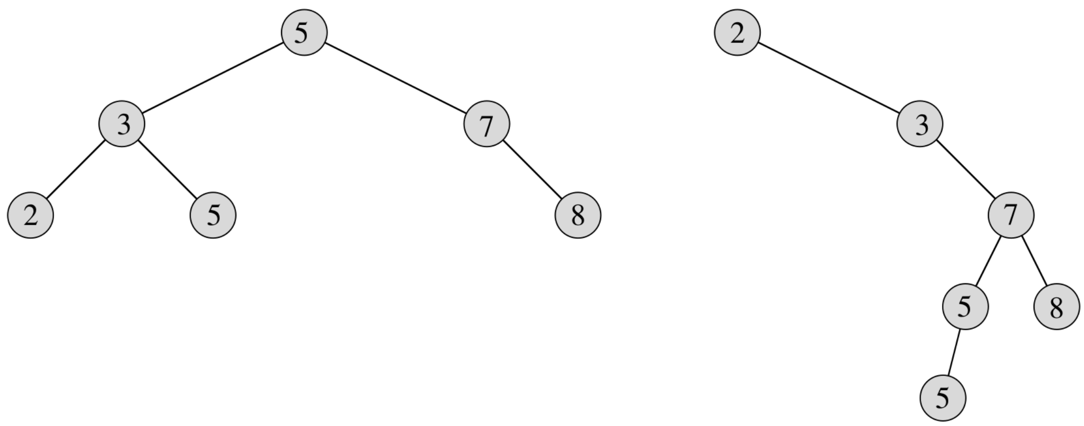
Note: we have not yet discussed what $\Theta(\cdot)$ means. That's part of Lecture (3).

Binary search tree

A binary tree, each of whose nodes contains a key.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$.



Algorithms to insert, delete, search, minimum, successor, ... — see textbook.
We discuss a few here.

Claim 1. *A key can always be inserted as a leaf into a binary search tree.*

TREE-SEARCH(x, k)

if $x =$ NIL or $k = key[x]$
    then return x
if $k < key[x]$
    then return TREE-SEARCH$(left[x], k)$
    else return TREE-SEARCH$(right[x], k)$

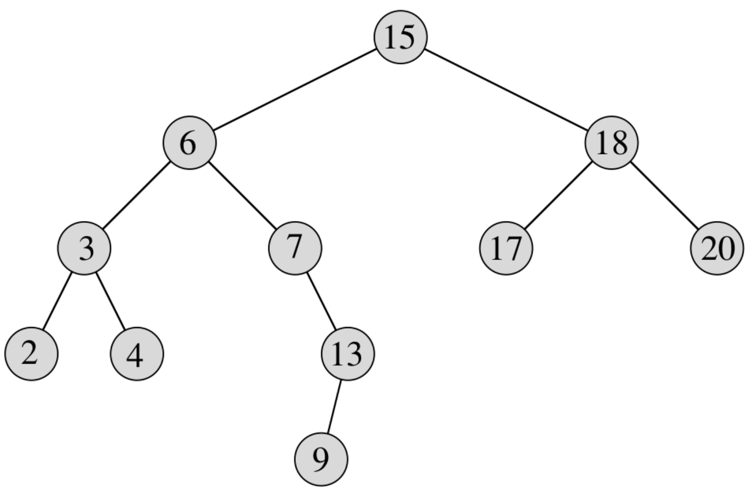
Claim 2. TREE-SEARCH: *(a) is guaranteed to terminate given an input tree and key that are each finite, and, (b) on termination gives the correct output.*

TREE-MINIMUM(x)

while $left[x] \neq$ NIL
    do $x \leftarrow left[x]$
return x

TREE-SUCCESSOR(x)

if $right[x] \neq$ NIL
    then return TREE-MINIMUM$(right[x])$
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$
    do $x \leftarrow y$
        $y \leftarrow p[y]$
return y



Claim 3. *Each of* TREE-MINIMUM *and* TREE-SUCCESSOR*: (a) is guaranteed to terminate, and, (b) output the correct thing upon termination.*