6.0

Last week we proved lower bounds on comparison based sorting, then gave counting sort that when all values are in a ~~not too~~ large interval works in linear time. Then we gave linear time alg. for finding the median and in general for returning the i'th element.

Reminder: Mid term on Oct 25. Will contain all material covered till Mid term.

Grading: HW grading has been very very gentle. It was important for me to encourage you to do your HW on your own, but also to give you good feedback on your ~~answer~~ solutions.
Midterm, exam will not be so gently graded!

Evaluation form: Please fill these.

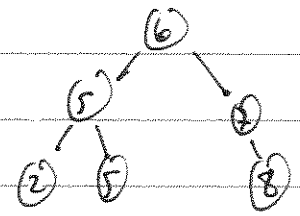
Today: Binary Search Trees

6.1

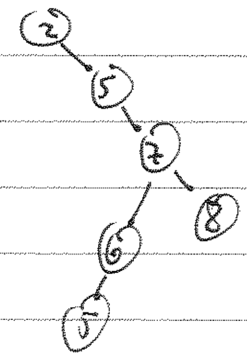
Binary Search Tree is a data structure for storing sets that evolve over time via insertions/deletions, similar to heaps. Thus, BST can be used for priority queue as well.
BST is designed so that searching the tree is similar to binary search and hence its name.
We will see the definition of BST and how to perform the basic operations on it.
Similar to heaps, most operations take time $\Theta(h)$, but unlike heaps, BST are not always balanced.
We shall speak more about heaps vs. BST at the end.

So what is a BST?
Each element has a key according to which we build the BST and satellite data.
The elements are ordered in a binary tree, which can be stored ~~as~~ using pointers from each node to its parent and children.
Thus, each node x contain the attributes x.key, x.left, x.right, x.p.
If a child or a parent is missing (e.g. leaf or root) then that attribute contains NIL.

The main property of BST is that for every node x in a BST we have that if y is in the left subtree of x then $y.key \le x.key$ and if y is in the right subtree of x then $y.key \ge x.key$.
E.g.



and



are BST representing the same data.

6.2

The nice thing about the BST property is that it allows us to print out all the keys in a sorted order in linear time by a simple recursive algorithm.
The alg. is called inordertree walk:

Inorder-tree-walk (x) $\qquad$ x is a node

1. If $x \neq NIL$
2. Inorder-tree-walk (x.left)
3. Print x.key
4. Inorder-tree-walk (x.right)

Running time is $\Theta(n)$ and correctness can be easily proved by induction. Indeed if x has k nodes on the left and n-k-1 on the right then

$$T(n) \leq T(k) + T(n-k-1) + O(1).$$

The solution is $T(n) \leq cn$.

We can also define preorder-tree walk which prints the root before the value in either subtree and postorder tree walk which prints it after the values in its subtree. For example

Preorder-tree-walk (x)

1. If $x \neq NIL$
2. Print x.key
3. Preorder-tree-walk(x.left)
4. Preorder-tree-walk(x.right)

6-3

A basic property of BST (and what makes them so useful) is that searching the tree and finding the max and min can be done efficiently. Moreover, finding successor and predecessor is also done efficiently depending only on h.

Tree-search (x,k) x is the root, k is the key we search

1. If x = NIL or x.key = k
2. return x
3. If k < x.key
4. Tree-search (x.left, k)
5. Else Tree-search (x.right, k)

Running time: Notice that if x.key ≠ k then we go down the tree. Thus, running time is $\Theta(h)$.

Another nice way to do search is:

Iterative-tree-search (x, k)

1. While x ≠ NIL and x.key ≠ k
2. If x.key < k
3. x ← x.left
4. Else x ← x.right
5. Return x

6.4

Min and Max are also easy, we go straight left or straight right.

Tree-Min (x)

1. While x.left $\neq$ NIL
2. $\quad$ x $\leftarrow$ x.left
3. Return x

Similar for Tree-max.
Successor is more interesting. Where can we find x's successor?
If x has a right child then the min value in that subtree is larger than x. Are there other candidates? what about x's parent?
Well, if x was a right child (x.p).right = x then x.p is smaller than x. if x was a left child then x.p.key is larger than x's and all of x's children. Thus, the minimum of the subtree of
~~x.right is x's prede successor.~~
~~But what if x.right = NIL?~~
~~Then we go to the father~~
Well, not quite. What about x.p.p? etc.
Notice that as long as x is a right child of x.p and x.p is a right child of (x.p).p etc. the value can only decrease. If at some point the parent is such that we were in its left subtree then this parent is larger than all values in its subtree including the min of x.right subtree.
OK, so if x.right $\neq$ NIL we know how to find x's successor. But what if x.right = NIL?

6.5

By the argument above, we should look for the first ancestor of x that x lies in its left subtree. Indeed notice that for this y we have that x is the max value in y.left.

```
Tree-successor (x)
1. If x.right ≠ NIL
2.      Return Tree-min (x.right)
3. y ← x.p
4. While y ≠ NIL and x = y.right
5.      x ← y
6.      y ← y.p
7. Return y.
```

Note that if x is the largest (and thus has no successor) we will reach the root and then go to its parent which is NIL.

Conclusion:

<u>Theorem:</u>

We can implement the dynamic-set operations search, Min, Max, successor, predecessor in time $O(h)$ on BST of height h.

6.6

We shall now see how to insert elements and delete elements from a BST. Inserting is relatively easy but deleting is more complicated.
The insertion alg. takes a tree T and it should insert the value v to T. So we assume we have a node z with z.key = v, z.left, z.right, z.p = NIL and inserts z to T.
The idea is to use search to find where to insert z.

```
Tree-insert (T, z)
y = NIL,                                  y will be x's parent
X = T.root
While  x ≠ NIL
        y ← x
        If  z.key < x.key
                x ← x.left
        Else  x ← x.right
z.p = y                                   y now points to x and x is
If  y = NIL                               where z should be, unless
        T.root = z                        tree is empty
Else  If  z.key < y.key
        y.left = z
        Else  y.right = z
```

Running time is again O(h).

6.7

What about deletion?

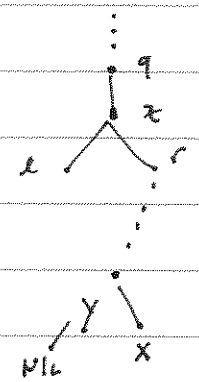
Well, if z is a leaf then no problems.

Also, if z has only one child then we just let $z.p$ point to z's child

If z has two children then we should replace z with its successor y.

But this means that we have to pull y from its location and this should be done carefully:

E.g. if we have



Then pulling y means we should connect x to y's parent. Indeed, y is the smallest so it is $y = y.p.left$ thus $y.p$ is larger than x.

In order to move subtrees around within the BST we define a subroutine called Transplant which replaces one subtree as a child of its parent with another subtree. (replacing u with v)

```
Transplant(T, u, v)
If u.p = NIL
    T.root = v
Else if u = u.p.left
    u.p.left ← v
Else u.p.right ← v
If v ≠ NIL
    v.p = u.p
```

If u is the root then we make v the root.

Otherwise we connect v to u's parent instead of u and make $v.p$ point to u's parent.

6.8

Notice that transplant does not touch the pointer to v from v.p. This will have to be dealt with by the alg. running transplant! Same with v's children.

Run-time is $O(1)$

We can now give the deletion alg.

```
Tree-Delete(T, z)
If   z.left = NIL                              } we first handle the case
        Transplant(T, z, z.right)              } of at most one child.
Else ~~Else If~~ If  z.right = NIL             }
            Transplant(T, z, z.left)           }
     Else   y = Tree-min(z.right)                y is the successor
        If    y.p ≠ z                            y is not z.right
                Transplant(T, y, y.right)      ← we fix y's subtree
                y.right = z.right              ← make y point to z.right
                y.right.p ← y                  ← and viceversa
        Transplant(T, z, y)                    ← Now we put y instead of z
        y.left ← z.left                        ← we make z's left son y's
        y.left.p ← y                           ← and make it point to y
```

Correctness follows from the previous discussion.

Run time is $O(h)$ again. ($O(1)$ Transplant operations and at most one Tree-Min)

6.9

One question is how to control the height of the BST.
Note that if we insert a sorted sequence one by one we will get a tree of depth $O(n)$. This is bad as our algorithms run in time $O(h)$.

The solution is to insert them in random order. One can show that if we pick a random permutation ~~the~~ (on n distinct keys) then with high probability there will not be monotone sequences of length more than $O(\log n)$.

BST vs. Heaps

	BST	max. Heap
~~Max~~		
Max	$O(h)$	$O(1)$
Delete	$O(h)$	$O(\lg n)$
Insert	$O(h)$	$O(\lg n)$

h can be very large. But

Search	$O(h)$	$O(n)$
Sort	$O(n)$	$O(n \lg n)$

And there are versions of BST that are self balancing so we can make h always $O(\lg n)$.

Build	$O(n \lg n)$	$O(n)$