Programming 2 Notes - Binary Tree

What is a Binary tree?

Root node
leaf nodes
left & right child
left & right Subtree
parent node
path to a node (from root)
Height of tree = largest path
Depth of tree = height
Size = number of nodes

What model a bunch of nodes a binary tree?

→ Each node has 0, 1, 2 child nodes
→ If >1 node : 1 node has no parent (the root)
→ All other nodes have exactly 1 parent

What is a Binary Search Tree?

A BST is:
- A binary tree
- The key in each node is unique
- Each node contains a key which is
  - → greater than keys in left subtree
  - → smaller than keys in right subtree

```
(class) BSTNode <K, V> {
    K key;
    V value;
    BSTNode <T> left;
    BSTNode <T> right;
}
```

Inorder traversal

```
String toString(TreeNode root){
    if (root == null) { return ""; }
    else { toString(root.left) + root.value + toString(root.right)
}
```

A BST is a binary Tree in Symetric order.

A binary tree is either:- empty
- 2 disjoint binary trees

Symetric order: Each node has a key and every other node's keys:
- Larger than all keys in its left subtree
- Smaller than all keys in its right subtree

Get

```
public Value get(Key key){
    Node x = root;
    while (x != null){
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val
    }
    return null;
}
```

Cost: Number of compares is equal to 1 + depth of node

Put:

Search for key then 2 cases:
- Key in tree => reset value;
- Key not in tree => add new node

Programming 2 Nodes BST.

```
public void put(Key key, Value val){
    root = put(root, key, val);
}
private Node put(Node x, Key key, Value val){
    if (x==null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp<0) x.left = put(x.left, key, val);
    else if (cmp>0) x.right = put(x.right, key, val);
    else if (cmp==0) x.val = val;

    return x;
}
```

Cost: Number of compares is equal to 1 + depth of node.

Proposition: If N distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

Proof: 1-1 correspondence with quicksort partitioning.

If N distinct keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.
BUT worst case height is N $\Rightarrow$ exponentially small chance when keys are inserted in random order.

	Guarantee		Average	
Sequential search (unordered list)	search	insert	search hit	insert
	N	N	$N/2$	N
binary search	$\lg N$	N	$\lg N$	$N/2$
BST	N	N	$1.39 \lg N$	$1.39 \lg N$

Delete

To delete a node with key k, search for node t containing key k.

Case 0: [0 children] Delete t by setting parent link to null
Case 1: [1 child] Delete t by replacing parent link.
Case 2: 2 children -Find successor x of t.
-delete the minimum in t's right subtree
-Put x in t's spot.

```
private node delete (Node x, Key key) {
    if (x==null) return null;
    int cmp = key.compareTo (x.key);
    if (cmp<0) x.left = delete (x.left, key);
    if (cmp>0) x.right = delete (x.right, key);
    else { if (x.right==null) return x.left;
        node t=x;
        x= min (t.right);
        x.right = deleteMin (t.right);
        x.left = t.left;
    }
    x.N = size (x.left) + size (x.right) + 1
    return x;
}
```

Programming Notes - Binary Trees

What is a binary tree?

- Each node has 0, 1, 2 child node
- If >1 node, 1 node has no parent (the root)
- All other node have exactly 1 parent

What is binary search tree?

- A binary tree
- the key in each node is unique
- each node contains a key which is:
  - greater than keys in left subtree
  - smaller than keys in right subtree

```
class BSTNode<K, V> {
  K key;
  V value;
  BSTNode<T> left;
  BSTNode<T> right;
}
```

A BST is a binary tree in symetric order.

Get: return value corresponding to given key or null if no such key

```
while (x != null) {
  compare to x, if less, go left, if greater go right
  if = return
}
```

Cost: number of compared is equal to 1+ depth of node

Put: Associate value with key

Search for key, the 2 cases: key in tree ⇒ replace
key not in tree ⇒ add new node

```
If (x is null create root).
Compare to x, if less go left, if greater go right
else reset value of x
```

Cost: number of compared is equal to 1 + depth of node

Remark: Tree shape depend on order of insertion

Proposition: If N distinct keys are inserted into a BST in RANDOM order, the expected number of compares for a search/insert is
$\sim 2 \ln N$.

Proof: 1-1 corresponds with quicksort partitions

Proposition: If N distinct keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

BUT worst case height is N. (exceptionally small chance that keys are inserted in random order!)

Implementation	Guarantee search	Guarantee insert	average case search hit	average case insert	Ordered ops?	Operations on keys
Sequential search (unordered list)	N	N	N/2	N	no	equals()
Binary search (ordered array)	lg N	N	lg N	N/2	yes	compareTo()
BST	N	N	1.39 lg N	1.39 lg N	?	compareTo()

Floor → Largest key ≤ to a given key
Ceiling → Smallest key ≥ to a given key

3 cases (= root, left the root, greater than root)
Use compare, if compare = 0 return x, otherwise if compare < 0, get floor of left
Otherwise get floor of right.

```
if (x == null) return null;
int cmp = key.compareTo(x.key);
if (cmp == 0) return x;
if (cmp < 0) return floor(x.left, key);
Node t = floor(x.right, key)
if (t != null) return t;
else return x;
```

Programming Assn BST.

Idea: In each node we store the number of nodes in the subtree rooted at that node.

Remark: This facilitates efficient implementation of rank() and select().

Rank: How many keys < k, recursive algorithm - 4 cases

```
private int rank ( Key key, Node x) {
  if(x==null) return 0;
  int cmp = key.compareTo(x.key);
  if (cmp < 0) return rank (key, x.left);
  else if cmp (>0) return 1 + size(x.left) + rank (key, x.right);
  else if (cmp==0) return size (x.left);
```

Select: key of given rank.

```
if (x==null) return null;
int t = size (x.left);
if (t > k) return select (x.left, k);
else if (t < k) return select ( x.right, k - t - 1);
else if (t == k) return x;
```

	Sequential Search.	binary Search.	BST.
search	N	lgN	H
insert	1	N	H
min/max	N	1	H
floor/ceiling	N	lgN	H
rank	N	lgN	H
select	N	1	H
ordered iteration	NlogN	N	N

- h = height of BST
- proportional to log N if keys inserted in random order

4

Deletion

To delete min, go left until finding a node with a null left link. Replace that node by its right link, update subtree counts

To delete a node with key k, search for node t containing key k.

Case 0: [0 children] delete t by setting parent link to null.

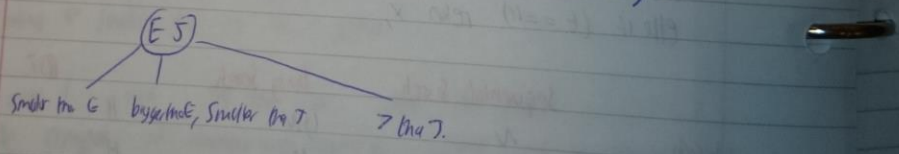
Case 1: [1 child] Delete t by replacing parent link.

Case 2: [2 children]: Find successor x of t.
Delete the min in t's right subtree
Put x in t's spot

Running cost will be $\sqrt{N}$

2-3 tree

2-node : one key, two children
3-node: two keys, three children
Symmetric order: Inorder traversal yields keys in ascending order.
Perfect balance: Every path from root to null link has same length



When inserting, if 4 node created, middle value U sent to parent
Split node into two nodes, continue whole way to root

Tree height: worst case: $\log N$ (all 2 nodes)
Best case: $\log_3 N \approx 0.631 \log N$ (all 3 nodes)
worst case after N inserts
Search Insert delete average case
2-3 tree: $c \log N$ $c \log N$ $c \log N$ $c \log N$ $c \log N$ $c \log N$

CIK ...

5.

Programming Notes BST.

2-3 trees by red-black trees

1. Represent 2-3 tree as a BST
2. Use "internal" left leaning links as "glue" for 3-nodes

Equivalent def":
A BST such that:
- No node has two red links connected to it.
- Every path from root to null link has same number of black links
- Red links lean left

Get a one-to-one sense of BST and one must of the appropriate

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in node

Should we use red or black link when linking to left of a 2-node? Red link

What about the one (right of a 2-node, into a 3-node?
- Red link because:
  - Never create new null in a 2-3 tree except when splitting a 4-node
  - Every path to null must have the same number of black links

Problem: Red links must lean left by definition
Fix by swapping red link node by left rotation