Binary Search Tree: Introduction

Local Search

A Local Search Data structure stores a number of elements each with a key coming from an ordered set. It supports operations:

- `RangeSearch(x, y)`: Returns all elements with keys between x and y
- `NearestNeighbors(z)`: Returns all the element with keys on either side of z

Dynamic data structure

`Insert(x)`: Adds an element with key x

`Delete(x)`: Removes the element with key x

Binary Search Tree: Search Trees

Parts of a tree

- `Root` node
- `Left` subtree has smaller keys
- `Right` subtree has bigger keys

Search tree property

X's key is larger than the key of any descendant of its left child, and smaller than the key of any descendant of its right child.

Binary Search Tree: Basic Operations

Find

Input: Key k, Root R

Output: The node in the tree of R with key k

```
Find(k, R):
if R.key == k:
    return R
else if R.Key > k:
    return Find(k, R.Left)
else if R.Key < k:
    return Find(k, R.Right)
```

```
Find(k, R) (modified):
if R.key == k:
    return R
else if R.Key > k:
    if R.Left != null:
        return Find(k, R.Left)
    return R
else if R.Key < k:
    if R.Right != null:
        return Find(k, R.Right)
    return R
```

Next

Input: Node N

Output: The node in the tree with the next largest key

```
Next(N):
if N.Right != null:
    return LeftDescendant(N.Right)
else:
    return RightAncestor(N)
```

```
LeftDescendant(N):
if N.Left == null:
    return N
else:
    return LeftDescendant(N.Left)
```

```
RightAncestor(N):
if N.Key < N.Parent.Key
    return N.Parent
else:
    return RightAncestor(N.Parent)
```

Range Search

Input: Numbers x, y, root R

Output: A list of nodes with key between x and y

```
RangeSearch(x, y, R):
L <- empty list
N <- Find(x, R)
while N.Key <= y:
    if N.Key >= x:
        L.Append(N)
    N <- Next(N)
return L
```

Insert

Input: Key k and root R

Output: Adds node with key k to the tree

```
Insert(k, R):
P <- Find(k, R)
Add new node with key k as child of P
```

Delete

Input: Node N

Output: Removes node N from the tree

```
Delete(N):
if N.Right == null:
    Remove N, promote N.Left
else:
    X <- Next(N) \\ X.Left = null
    Replace N by X, promote X.Right
```

Binary Search Tree: Balance

- When left and right subtrees have approximately the same size
- Suppose perfectly balanced;
    - Each subtree half the size of its parent
    - After $\log_2 n$ levels, subtree of size 1
    - Operations run in $O(\log(n))$ time

However, insertion and deletion can destroy balance, rotation is needed.

Let Y be a tree such that `Y.Left` is A and `Y.Right` is B, and X be a tree such that `X.Left` is Y, `X.Right` is C and `X.Parent` is P. We have $A \leq Y \leq B \leq X \leq C \leq P$.

```
RotateRight(X):
P <- X.Parent
Y <- X.Left
B <- Y.Right
Y.Parent <- P
P.AppropriateChild <- Y
X.Parent <- Y, Y.Right <- X
B.Parent <- X, X.Left <- B
```

After rotation, Y be a tree such that `Y.Parent` is P, `Y.Left` is A and `Y.Right` is X, and X be a tree such that `X.Left` is B, `X.Right` is C. We still have $A \leq Y \leq B \leq X \leq C \leq P$

Binary Search Tree: AVL Trees

Height

The height of a node is the maximum depth of its subtree.

```
Height(tree): # Number of nodes to its leaf. If N is a leaf, it has
height of 1
if tree == nil:
    return 0
return 1 + Max(Height(tree.left), Height(tree.right))
```

AVL Property

For all nodes N,

$$|\text{N.Left.Height} - \text{N.Right.Height}| \leq 1$$

Theorem

Let N be a node of a binary tree satisfying the AVL property. Let $h = \text{N.Height}$. Then the subtree of N has size at least the Fibonacci Number F_h.

Proof:

- By induction on h.
- If $h = 1$, it has one node.
- Otherwise, it has one subtree of height $h - 1$ and another subtree of height at least $h - 2$. By inductive hypothesis, total number of nodes is at least $F_{h-1} + F_{h-2} = F_h$.

So, node of height h has subtree of size at least $2^{\frac{h}{2}}$ because $F_n \geq 2^{\frac{n}{2}}$ for $n \geq 6$. In other words, if n nodes in the tree, it has height $h \leq 2\log_2 n = O(\log n)$.

Remark: $F_n \geq 2^{\frac{n}{2}}$ can be proved by induction on n.

Binary Search Tree: AVL Tree Implementation

```
AVLInsert(k, R):
Insert(k, R)
N <- Find(k, R)
Rebalance(N)
```

```
Rebalance(N):
P <- N.Parent
if N.Left.Height > N.Right.Height + 1:
    RebalanceRight(N)
if N.Right.Height > N.Left.Height + 1:
    RebalanceLeft(N)
AdjustHeight(N)
if P != null:
    Rebalance(P)
```

```
AdjustHeight(N):
N.Height <- 1 + max(N.Left.Height, N.Right.Height)
```

```
RebalanceRight(N):
M <- N.Left
If M.Right.Height > M.Left.Height:
    RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes
```

```
AVLDelete(N):
Delete(N)
M <- Parent of node replacing N
Rebalance(M)
```

Remark: AVL tree allows $O(\log n)$ time per operation

Binary Search Tree: Split and Merge

Merge

Input: Roots R_1 and R_2 of trees with all keys in R_1's tree smaller than those in R_2's

Output: The root of a new tree with all the elements of both trees

```
MergeWithRoot(R1, R2, T): # Time O(1)
T.Left <- R1
T.Right <- R2
R1.Parent <- T
R2.Parent <- T
return T
```

```
Merge(R1, R2): # O(h)
T <- Find(infinity, R1)
Delete(T)
MergeWithRoot(R1, R2, T)
return T
```

```
AVLTreeMergeWithRoot(R1, R2, T):
if |R1.Height - R2.Height| <= 1:
    MergeWithRoot(R1, R2, T)
    T.Ht <- max(R1.Height, R2.Height) + 1
    return T
else if R1.Height > R2.Height:
    R' <- AVLTreeMergeWithRoot(R1.Right, R2, T)
    R1.Right <- R'
    R'.Parent <- R1
    Rebalance(R1)
    return root
else if R1.Height < R2.Height:
    R' <- AVLTreeMergeWithRoot(R1, R2.Left, T)
    R2.Left <- R'
    R'.Parent <- R2
    Rebalance(R2)
    return root
```

- Each step changes height difference by 1 or 2
- Eventually within 1
- Time complexity $O(|R_1.\text{Height} - R_2.\text{Height}| + 1) = O(\log n)$

Split

Input: Root R of a tree, key x

Output: Two trees, one with elements $\leq x$, one with elements $> x$

```
Split(R, x):
if R == null:
    return (null, null)
if x <= R.Key:
    (R1, R2) <- Split(R.Left, x)
    R3 <- MergeWithRoot(R2, R.Right, R)
    return (R1, R3)
if x > R.Key:
    (R1, R2) <- Split(R.Right, x)
    R3 <- MergeWithRoot(R1, R.Left, R)
    return (R1, R3)
```

Binary Search Tree: Applications

Order statistics

Input: The root of a tree T and a number k

Output: The k^{th} smallest element in T

Remark: Need to know which subtree to look in and how many elements are in the left subtree.

A new field is needed: `N.Size = N.Left.Size + N.Right.Size + 1`*, null node has size* 0

```
RecomputeSize(N):
N.Size <- N.Left.Size + N.Right.Size + 1
```

```
Rotate:
As before
RecomputeSize(Old root)
RecomputeSize(New root)
```

```
OrderStatistic(R, k):
s <- R.Left.Size
if k == s + 1:
    return R
else if k < s + 1:
    return OrderStatistics(R.Left, k)
else if k > s + 1:
    return OrderStatistics(R.Right, k - s - 1)
```

Color flips

Problem: An array of squares in either black or white for each square. Want to be able to flip colors of all squares after index x.

```
NewArray(n):
Create two trees T1, T2 with keys 1...n
Give nodes extra Color field
All in T1 have color White
All in T2 have color Black
```

```
Color(m):
N <- Find(m, T1)
return N.Color
```

```
Flip(x):
(L1, R1) <- Split(T1, x)
(L2, R2) <- Split(T2, x)
Merge(L1, R2) -> T1
Merge(L2, R1) -> T2
```

Binary Search Tree: Splay Tree

Non-uniform inputs

- Search for random elements $O(\log n)$ best possible
- If some items more frequent than others, can do better putting frequent queries near root

```
Splay(N):
Determine proper case
Apply Zig-Zig, Zig-Zag, or Zig as appropriate
if N.Parent != null:
    Splay(N)
```

```
STFind(k, R):
N <- Find(k, R)
Splay(N)
return N
```

```
STInsert(k, R):
Insert(k, R)
STFind(k, R)
```

```
STDelete(N):
Splay(Next(N))
Splay(N)
Delete(N)
```

```
STSplit(R, x):
N <- Find(x, R)
Splay(N)
split off appropriate subtree of N
```

```
STMerge(R1, R2):
N <- Find(infinity, R1)
Splay(N)
N.Right <- R2
```

Performs all operations in $O(\log n)$ in amortized time.

Other property of splay tree

Weighted nodes: If you assign weights so that $\sum_N \mathrm{wt}(N) = 1$, accessing N costs $O(\log(\frac{1}{\mathrm{wt}(N)}))$

Dynamic finger: Cost of accessing node $O(\log(D+1))$ where D is distance between last access and current access

Working set bound: Cost of accessing N is $O(\log(t+1))$ where t is time since N was last accessed