Tree

Definition: A tree is a collection of nodes. If non-empty, the tree has one root node r and zero or more subtrees whose respective root nodes are connected to r by a directed edge.
Preorder Traversal: process a node before traversing its children.
Postorder Traversal: process a node after traversing its children.

Node

Depth/Level: the path length from root node to this node. Root node is at level 0.
Height: the largest path length from this node to a descendant leaf. The height of a tree is the height of its root node.

Edge

Theorem 1: A tree of n nodes has $n - 1$ edges since each node except r has a parent node (denoted by an edge)

Path

Path: A path p from node n_1 to n_k is the sequence of nodes $n_1, n_2, \cdots, n_k$ such that n_i is the parent of n_{i+1}

Binary Tree

Definition: a tree whose nodes can have at most two children

Theorem 1: At most 2^X nodes can be stored at level X
Theorem 2: At most $2^X - 1$ nodes can be stored in X levels
Theorem 3: At least $\log(N + 1)$ levels required to store N nodes

Inorder Traversal: traverse the left subtree, process or the root node, and then traverse the right subtree.

BST (Binary Search Tree)

Definition: a binary tree whose nodes are sorted with respect to inorder traversal.

insert(): a smaller value goes left, a larger value goes right.
delete():

- Situation 1: X-L = null && X-R = null. Then set a null be child of X-P in place of X
- Situation 2: X-L $\neq$ null && X-R = null. Then two sub-situations:
  - Sub-situation 1: X-L-R $\neq$ null. Then:
    1 set X-L-RM be child of X-P in place of X, (X-L-RM-P, X-L-RM and X have wrong children.)
    2 set X-L-RM-L be right child of X-L-RM-P. (X-L-RM and X have wrong children.)
    3 set X-L and X-R be left and right child of X-L-RM. (X has wrong children.)
    4 set two nulls be left and right child of X
  - Sub-situation 2: X-L-R = null. Then:
    1 set X-L be child of X-P in place of X. (X-L and X have wrong children.)
    2 set X-R be right child of X-L. (X has wrong children.)
    3 set two nulls be left and right child of X
- Situation 3: X-R $\neq$ null && X-L = null. Then two sub-situations:
  - Sub-situation 1: X-R-L $\neq$ null. Then:
    1 set X-R-LM be child of X-P in place of X. (X-R-LM-P, X-R-LM and X have wrong children.)
    2 set X-R-LM-R be left child of X-R-LM-P, (X-R-LM and X have wrong children.)
    3 set X-L and X-R be left and right child of X-R-LM. (X has wrong children.)

4 set two nulls be left and right child of X

Sub-situation 2: X-R-L = null. Then:

1 set X-R be child of X-P in place of X, (X-R and X have wrong children.)

2 set X-L be left child of X-R. (X has wrong children.)

3 set two nulls be left and right child of X

Situation 4: X-R $\neq$ null && X-L $\neq$ null. Then follow either situation 2 or 3. Situation 3 is used in the lecture notes.

CBBST (Completely Balanced BST)

Definition: a BST whose levels all have the largest number of nodes.

Theorem 1: The number of nodes of the tree is $2^{H+1} - 1$

Theorem 2: The height of the tree is $\log(N+1) - 1$

Question 1: Tw to search a node, in terms of number of probes? $Tw = H + 1$

Question 2: Ta to search a node, in terms of number of probes? $Ta \approx H \approx \log N$ when $2^H >> 1$

AVL (Adelson-Velskii and Landis)

Definition: a BST where the height of every left and right subtrees of a node can differ by at most 1

Theorem: Smallest number of nodes in a tree of height H is $N_H = N_{H-1} + N_{H-2} + 1$, $N_0 = 1$, $N_1 = 2$

insert(): consider the path from the violating node r to nodes at the next two levels:

Left Left: rotateRight(r)

```
rotateRight ( theParent& ) {
    leftChild = theParent->left
    theParent->left = leftChild->right
    leftChild->right = theParent
    theParent = leftChild
}
```

Right Right: rotateLeft(r)

```
left_rotate( theParent& ) {
    rightChild = theParent->right
    theParent->right = rightChild->left
    rightChild->left = theParent
    theParent = rightChild
}
```

Left Right: rotateLeft(r->left), rotateRight(r)

Right Left: rotateRight(r->right), rotateLeft(r)

delete(): the same as a BST deletion but may cause an imbalance. This may need h/2 rotations to rebalance it.

Hash

Open Hashing

Use a data structure such as a linked list to store all keys that hash to the same location.

K: the number of keys

L: the number of locations

λ: the size of a linked list. $\lambda = K / L$ if the keys are uniformly spread out

AST-F: average search time if found. AST-F = O(1) +$\lambda/2$
AST-NF: average search time if not found. AST-NF = O(1) +λ
Principle 1: Keep $\lambda \leq 1.0$
Principle 2: Keep L prime to ensure good distribution

Closed Hashing

h(): the function to find an unused location, h(X) = (hash(X) + f(i)) % L
X: the key
hash(): the hash function to map a key to an original location
f(): the function to calculate the location offset. f(0) = 0
i: the number of probes, starting at 0 and increment by one every time
L: the number of locations
Linear Probing: f(i) = i. It will always find an empty location if one exists, but time to find it can be very bad.
Quadratic Probing: f(i) = i^2. If table size is prime, then a new element can always be inserted if the table is at least half empty. So we must keep $\lambda \leq 0.5$ to ensure successful insertion.
Double Hashing: f(i) = i × h2(X), where h2 is the secondary hash function

Heap

Definition: a binary tree that is completely filled except for the bottom level.
- Theorem 1: $2^H \leq N \leq 2^{H+1} - 1$
- Theorem 2: $\log N \leq H \leq \log(N + 1)$
- Theorem 3: H = O(log N)

Probe: for any element arr[i]:
- Left Child: arr[2i]
- Right Child: arr[2i+1]
- Parent: arr[i/2]

Priority Queue

Definition: A special heap whose parent nodes are less than or equal to their children. The heap itself is implemented by an array.

insert(): insert the new element into the bottom of the heap and percolate it up until heap order property is satisfied
- Tw: O(log n)
- Ta: O(1)

delete_min(): place the lastest-inserted element at i = 1 and trickle it down until heap order property is satisfied. When trickling down, we must make sure that we swap with the smaller of the two children.
- Tw: O(log n)

build_heap(): make n successive insert()
- Tw : O(n) (Improved)
- Ta: O(n)