Notation

O = Upper bound

Ω = Lower bound

Θ = Exact bound

Insertion Sort

Space complexity: O(1)

Time complexity: $O(n^2)$

Merge Sort

Space complexity: O(n)

Time complexity: O(n log n)

Divide-and-Conquer by recursion

Quick Sort

Space complexity: O(log n)

Time complexity: O(n log n)

Linked List

Singly, Double, Circular Link List

Implementation:

```
struct node {
int info;            // data
node* next;
};
node* head;      // head pointer pointing to first node
```

Find length:

```
int len = 0;
node *cur = head; //traverse the list using cur
while (cur != NULL) { // cur points to a valid node
len++;
cur = cur->link; //move to the next node
}
```

Search from beginning:

```
node *cur = head;
while (cur != NULL && cur->info != x) { // search x
cur = cur->link;
}
// cur now points to target x or is NULL (x does not exist)
```

Insert at front:

```
// create the node dynamically for storing x
node *p = new node;
p->info = x;
p->link = head;     // step 1
head = p;           // step 2
```

Merge two sort linked list:

Step 1: Create a new dummy node. It will have the value 0 and will point to NULL respectively. This will be the head of the new list. Another pointer to keep track of traversals in the new list.

Step 2: Find the smallest among two nodes pointed by the head pointer of both input lists, and store that data in a new list created.

Step 3: Move the head pointer to the next node of the list whose value is stored in the new list.

Step 4: Repeat the above steps till any one of the head pointers stores NULL. Copy remaining nodes of the list whose head is not NULL in the new list.

Stack

Last In First Out (LIFO) order

Operations: initialize, size, empty, top, push, pop

Queue

First In First Out (FIFO) order

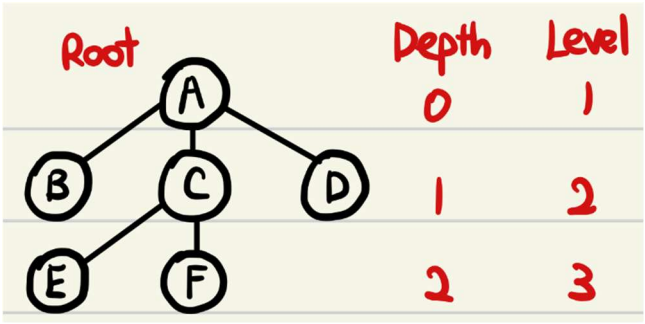
Operations: initialize, size, empty, front, back, push, pop

Double-Ender Queue

Allows insertion and deletion at both ends

push_front, push_back, pop_front, pop_back

Tree



Depth: Length of the unique path from root to node

Level: It's depth + 1

Height: No. of edges from the node to the deepest leaf

Height of a tree = Height of the root

Degree: No. of subtree of the node

Root node: The top node in a tree

Interior nodes: Nodes with at least one child

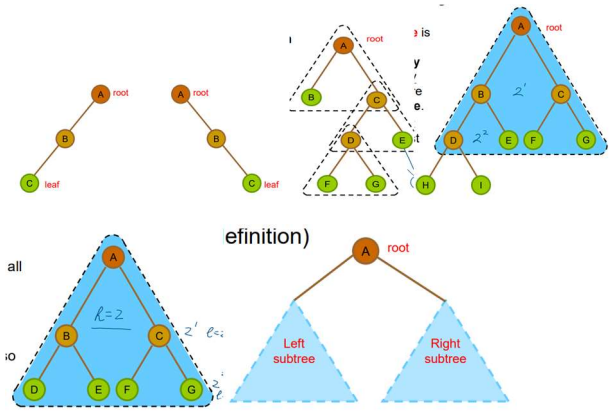
Leaf nodes: Nodes with no child

Ancestors of X: Nodes along the simple path from the root to node X

Descendants of X: Nodes in the subtree of X

Binary Tree

Definition: Each node has at most 2 children (L/R)



Skewed Tree (1)

All nodes are either on the left-hand side or right-hand side

Full Binary Tree (2)

Every node in the tree has either 0 or 2 children.

Complete Binary Tree (3)

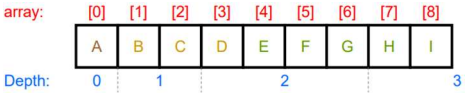
Every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Perfect Binary Tree (4)

All interior nodes have two children, and all leaves have the same depth or same level.

Left / Right Subtree (5)

Array Implementation



Linked List Implementation

```
template<class Type>
struct treeNode {
Type info;
treeNode<Type> *left, *right;
};
```

Find parent node: int((x-1)/2) or int (x/2)

Find child node: int(2x)&(2x+1) or int(2x+1)&(2x+2)

Compute the height: 1 + h(left) + h(right)

Traversal: Level, Preorder(VLR), Inorder(LVR), Postorder(LRV)

Binary Search Tree: Left subtree < Node < Right subtree

Left and right subtree each must also be a binary search tree

Search in BST: Compare k with root, root == k answer is root, root > k go left, root < k go right, until meet leaf node

Time complexity: O($\log_2 n$)

Insert: Create a new node, compare value and key, until NULL

Delete: Case 1(Degree 0 Node): Delete it, reset the reference

Case 2(Degree 1 Node): Adjust the pointer of parent to point to the grandson, delete it

Case 3(Degree 2 Node): Replace the deleted node with its biggest node in left subtree or smallest node in right subtree, if the inorder successor or predecessor has a child, delete it in turn with the same steps in case 2.

Output of in-order traversal of a BST = Ascending order

Heap

Definition: Max heap: is a tree in which the key value in each node is no smaller than the key values in its children (if any), min heap vice versa.

Implement: With node and tree structure or array

Save the tree in an array with starting index 1

Index of each left/right child = 2* index of its parent/+1

Insert: Push back, percolate up

Remove: Replace with the bottom rightmost, percolate down

Heap sort: Swap the top element with last element in the array, Heap_ size – 1, Heapify(H,1), until Heap_size = 1

Hash

Hash function: well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small range of index (positions)

Ideal properties: low cost, variable range, uniformity

Hash Collision resolution: Chaining, Open addressing (Linear Probing) Chaining vs Open Addressing: Need outside storage, deletion easier, wastage of space

Pattern matching:

Algorithm 1 Use of hashing for substring search function RabinKarp(String S[1, . . . , n], String sub[1, . . . , m])

```
1: hsub = hash(sub[1, ..., m])
2: hs = hash(S[1, ..., n])
3: for i = 1 to n − m + 1 do
4:     if hs == hsub then
5:         if S[i, ..., i + m − 1] == sub then        ▷ String comparison
6:             return i
7:         hs = hash(S[i+1, ..., i+m)
8: return not found
```

Bloom-Filter: A space-efficient data structure that is used to test whether an element is a member of a set

DFS: Start from a vertex, travel to a new vertex going as deep as possible, if stuck, go back and repeat the process

BFS: Start from a vertex, travel to new vertex in the first layer (which is connected with the start vertex), if there is no new vertex in the first layer, go to nodes in the second layer (which is connected with the nodes in the first layer)

Topological Sort: find a vertex v with 0 in-degree, output v, delete v and the corresponding outgoing edges from G