1 Binary Search Trees (BST)

A Binary Search Tree is a binary tree that satisfies the BST order property.

1.1 BST Algorithms and Complexity

For an unbalanced BST of size n, the height can be $\Theta(n)$. Balanced BSTs have height $\Theta(\log n)$.

Table 1: BST Runtime Analysis (Size n)

Operation	**Unbalanced**	**Bal / Avg**
Insert an item	$O(n)$	$O(\log n)$
Insert all n items	$O(n^2)$	$O(n \log n)$
Search	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(\log n)$

1.1.1 BST Insert

```
algorithm insert(root:node, x:item) -> node
if root is null then
    create new node n with item x
    return n
end if
if x < root.item then
    root.left <- insert(root.left, x)
else if x > root.item then
    root.right <- insert(root.right, x)
end if
return root
end algorithm
```

1.1.2 BST Search

```
algorithm search(root:node, x:item) -> node
if root is null then return null
if x = root.item then return root
if x < root.item then
    return search(root.left, x)
return search(root.right, x)
end algorithm
```

2 Self-Balancing Trees

2.1 Red-Black Trees (RBT)

A Red-Black Tree is a self-balancing BST with the following properties:

1. Every node is red or black.
2. The root is black.
3. Every null link is black.
4. Red nodes have black children.
5. All root-to-null paths contain the same number of black nodes.

2.1.1 LLRBT Helper Operations

Color Flip

```
function flipColors(h):
    h.color = RED
    h.left.color = BLACK
    h.right.color = BLACK
```

Left Rotation

```
function rotateLeft(h):
    x = h.right
    h.right = x.left
    x.left = h
    x.color = h.color
    h.color = RED
    return x
```

Right Rotation

```
function rotateRight(h):
    x = h.left
    h.left = x.right
    x.right = h
    x.color = h.color
    h.color = RED
    return x
```

2.1.2 LLRBT Insert

```
function insertHelper(h, key, value):
    if h == null: return new Node(key, value, RED)
    ... BST insert logic ...
    return fixUp(h)

function fixUp(h):
    if isRed(h.right) and not isRed(h.left):
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left):
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right):
        flipColors(h)
    return h

function insert(key, value):
    root = insertHelper(root, key, value)
    root.color = BLACK
    return root
```

3 B-Trees

B-Trees are multi-way balanced search trees optimized for disk access.

4 Graph Structures and Representations

A graph G consists of vertices V and edges E.

4.1 Graph Representation Complexity

Table 2: Graph Representation Complexities

Rep	Space	Add Edge	Adj Chk
Edge List	$O(E)$	$O(1)$	$O(E)$
Adj Mtrx	$O(V^2)$	$O(1)$	$O(1)$
Adj List	$O(V+E)$	$O(1)$	$O(\deg u)$

4.1.1 Breadth-First Search (BFS)

```
algorithm BFS(G, v):
    enqueue v; mark visited; dist[v] = 0
    while queue not empty:
        u = dequeue
        for each w adjacent to u:
            if not visited:
                enqueue w; mark visited
                edgeTo[w] = u; dist[w] = dist[u] + 1
    return (edgeTo, dist)
```

4.1.2 Depth-First Search (DFS)

```
algorithm DFS(G, v):
    mark v
    for each w adjacent to v:
        if w unmarked: DFS(G, w)
```

4.1.3 Finding Connected Components

```
algorithm FindAllConnectedComponents(G):
    m = 1; C[v] = 0 for all v
    for each v in V:
        if C[v] == 0:
            BFSMark(G, v, m, C)
            m++
    return C
```

4.1.4 Topological Sort

```
algorithm TopologicalSort(G):
    compute indegree count
    S = nodes with indegree 0
    while S not empty:
        remove v from S; append to T
        for edges (v,u):
            decrement count[u]
            if count[u] == 0: add u to S
    return T
```

4.1.5 Strongly Connected Components (Kosaraju)

```
Phase 1:
    run DFS, pushing nodes to stack L on finish
Phase 2:
    unmark all
    while L not empty:
        v = pop L
        if unmarked: DFS on Grev marking SCC
```

4.1.6 Transitive Closure (Floyd-Warshall)

```
for k in 0..n-1:
  for i in 0..n-1:
    for j in 0..n-1:
      R[i][j] = R[i][j] or (R[i][k] and R[k][j])
```

5 Shortest Path Algorithms

5.1 Dijkstra

```
algorithm Dijkstra(G, s):
    initialize dist[]; dist[s]=0
    push all nodes into min-heap
    while heap not empty:
        u = extractMin
        for each w adjacent to u:
            if dist[u] + wght(u,w) < dist[w]:
                update dist[w] and decrease-key
    return dist
```

5.2 Bellman-Ford

```
for i = 1 to V-1:
    relax all edges
for each edge:
    if still relaxable: negative cycle
```

5.3 DAG Shortest Paths

```
compute topological order
dist[s] = 0
for each u in topo order:
    relax all outgoing edges
```

5.4 All-Pairs Shortest Paths (Floyd-Warshall)

```
for k in 0..n-1:
  for i in 0..n-1:
    for j in 0..n-1:
      D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

6 Minimum Spanning Tree Algorithms

6.1 Prim's Algorithm

```
initialize costToAdd[], visited[], priority queue
while queue not empty:
    u = extractMin; mark visited
    for edges (u,v): update improvement
```

6.2 Kruskal's Algorithm

```
sort edges by weight
for each edge (u,v):
    if find(u) != find(v):
        union(u,v); add edge to MST
```

6.3 Union-Find Structure

```
function find(p):
    if p == id[p]: return p
    id[p] = find(id[p]); return id[p]

function union(p,q):
    rootP = find(p); rootQ = find(q)
    attach smaller tree to larger
```