CS2010 - Binary Search Trees (BST)

1 BST

1.1 BST Property

`x.left.key` $<$ `x.key` $\leq$ `x.right.key`

Left sub-tree < Parent

Right sub-tree $\leq$ Parent

1.2 Searching

```
search(node, value)
    if (node == null) return null // not found
    if (node == value) return start // found
    if (node < value) return search(node.left) // search left sub tree
    if (node >= value) return search(node.right) // search right sub tree
```

1.3 Insertion

```
insert(node, value)
    if (node == null) return new Vertex(value) // found empty spot to insert
    if (value < node) // look in the left sub-tree to insert
        node.left = insert(node.left, value)
        node.left.parent = node
    else // look in right sub-tree to insert
        node.right = insert(node.right, value)
        node.right.parent = node
    return node
```

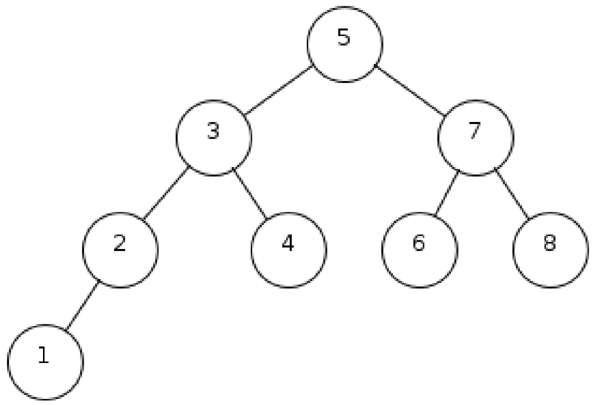
1.4 findMin

```
findMin(node)
    if (node == null) return null // empty tree!!!
    if (T.left == null) return node // nothing less that this node, its the minimum
    else return findMin(node.left) // continue moving down left
```

2 Transversal

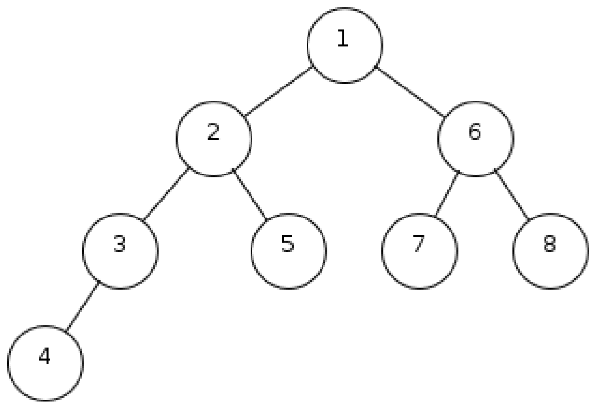
In-order (Sorted order)

```
inorder(node)
    if (node == null) return
    inorder(node.left)
    process(node)
    inorder(node.right)
```



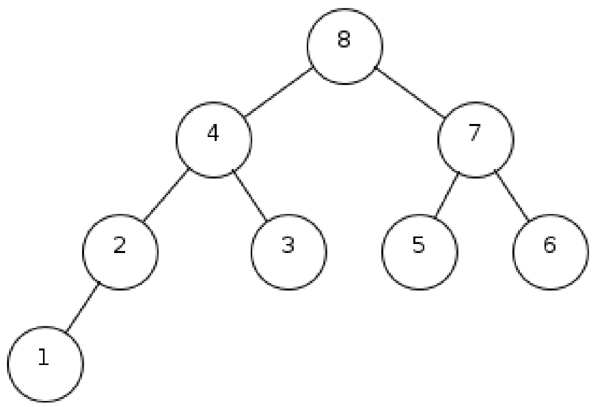
Pre-order

```
preorder(node)
    if (node == null) return
    process(node)
    inorder(node.left)
    inorder(node.right)
```



Post-order

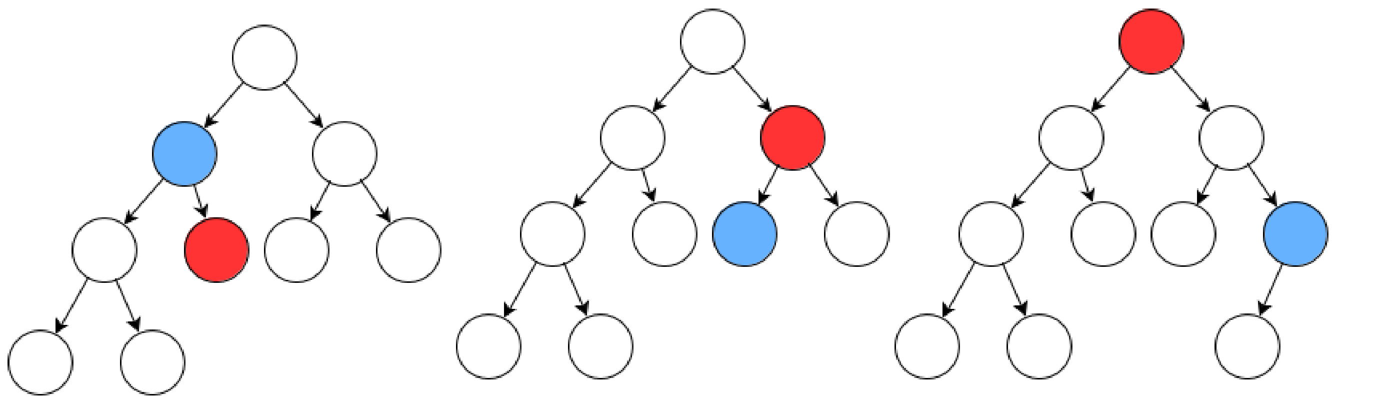
```
postorder(node)
    if (node == null) return
    inorder(node.left)
    inorder(node.right)
    process(node)
```



3 Successor

Either the max of the right sub-tree. Go up until we make a right turn

```
successor(node)
    // if it has a right sub-tree, min of sub-tree is successor
    if (node.right != null) return findMin(node.right)
    else
        parent = node.parent
        curr = node
        while (parent != null && curr == parent.right)
            curr = parent
            parent = curr.parent
        if (parent == null) return null // no right child nor parent (only node left)
        else return parent
```



4 Delete ($O(h)$)

1. find the node to delete
2. if the node is a **leaf** (no children), just remove it
3. if it has only a **1 child**, just bypass node to remove
4. if it has **2 children**, replace node with sucessor. Then remove node

CS2010 - 2 - Balanced BST

1 Height of a Sub-Tree

Max number of edges in tree

- Empty tree: -1
- else: max(left.height, right.height) + 1

2 Height Balanced

`bf(node) = node.left.height - node.right.height`

Tree is height balanced if `|bf(root)|` $\leq$ `1`

3 Rotations

3.1 Left Rotate

```
rotateLeft(node) // node.left != null
    newRoot = node.right
    newRoot.parent = node.parent
    node.parent = newRoot
    node.right = newRoot.left
    if (newRoot.left != null) newRoot.left.parent = node
    newRoot.left = node
    // update height of node and newNode
    return newNode
```

3.2 Possible rotations

3.2.1 Left-Left

```
if (bf(node) == 2 && bf(node.right) == 1)
```

```
    rightRotate(node)
```

3.2.2 Right-Right

```
if (bf(node) == -2 && bf(node.left) == -1)
    leftRotate(node)
```

3.2.3 Right-Left

```
if (bf(node) == -2 && bf(node.right) == 1)
    rightRotate(node.right)
    leftRotate(node)
```

3.2.4 Left-Right

```
if (bf(node) == 2 && bf(node.left) == -1)
    leftRotate(node.left)
    rightRotate(node)
```

4 Insert

Just insert, then walk up tree checking balance factor. When $bf(node) == \pm 2$, rebalance.

CS2010 - 3 - Min/Max Heaps, Priority Queue

1 Complete Binary Tree in compact array

Start from 1 (skip 0)

- `indexOfParent(node) = floor(node/2)`
- `indexOfLeft(node) = 2 * node`
- `indexOfRight(node) = 2 * node + 1`
- No left child if `indexOfLeft > heapsize`
- No right child if `indexOfRight > heapsize`

2 Heap Property

- **Max Heap**: parent $\geq$ children
- **Min Heap**: parent $\leq$ children

3 Insert into Max Heap

Insert at empty leaf then `shiftUp` to ensure heap property

```
insert(node)
    arr[] = v // insert at leaf
    shiftUp(arr.length - 1) // shiftUp newly added node
```

4 Shift Up

```
shiftUp(node)
    while (node.index > 1 && node.parent < node)
        swap(node, node.parent)
        node = node.parent
```

5 Delete

Replace node with last element, then call `shiftDown` on replaced node.

6 Extract Max

```
extractMax()
    max = arr[1] // max is at index 1
    arr[1] = arr[arr.length - 1] // replace max with last element
    shiftDown(1)
    return max
```

7 Shift Down

```
shiftDown(node)
    while (node != null)
     max = node
     if (node.left && max < node.left)
         max = node.left
     if (node.right && max < node.right)
         max = node.right

     if (max != node)
         swap(node, max)
     else
         break
```

8 Build Heap

Just insert all elements into array. Then shift down from root

```
buildHeap(arr)
    heap[0] = null // dummy entry
    for (elem in arr)
        heap[] = elem // insert into heap arr
```

```
// no need to shift down for root
for (node = parent of last element; to root)
    shiftDown(node)
```

Graphs

- DAG: Directed graph with no cycles
- Tree: Connected graph with only 1 unique path between any 2 pairs of vertices. $E = V - 1$
- Bipartite graph: graph with vertices that can be paritioned into 2 sets, where members of 1 set cannot have edges to another in the same set

1 Storage

1.1 Adjacency matrix

- 2D array, each cell containing 1 or edge weight. eg. `adjMatrix[i][j]` refers to edge weight of edge connecting i to j
- Space complexity: $O(V^2)$

1.2 Adjacency list

- `AdjList[i]` stores list if i's neighbours
- Space complexity: O(V+E)

2 Graph transversal

2.1 BFS ($O(V + E)$)

```
visited = new bool[V]
q.enqueue(src)
while (q.size() > 0)
    elem = q.dequeue()
    foreach (neighbour in neighbours(elem)) // O(E) : adj list
        if (!visited[neighbour]) // ensures O(V)
         visited[neighbour] = true
         q.enqueue(neighbour)
```

2.2 DFS ($O(V + E)$)

```
visited = new bool[V]
dfs(src)

dfs(v):
    visited[v] = true
    foreach (neighbour in neighbours(v)) // O(E)
        if (!visited[neighbour]) // ensures O(V)
            dfs(neighbour)
```

2.3 Topological sort

- Linear ordering in **DAG** where each vertex comes before all vertices to which it has outbound edges to
- OR only right arrows on a toposort
- Run DFS, appending to toposort once all edges are processed

```
visited = new bool[V]
dfs(src)

dfs(v):
    visited[v] = true
    foreach (neighbour in neighbours(v)) // O(E)
        if (!visited[neighbour]) // ensures O(V)
            dfs(neighbour)
    toposort[] = v
toposort.reverse() // need to reverse
```

Minimum Spanning Tree (MST)

1 MST

- Spanning Tree of G with min total weight

1.1 Prims ($O(E \log V)$)

1.1.1 Pseudocode

```
visited = new bool[V]
foreach (neighbour in neighbours(src))
    PQ.enqueue(neighbour) // { weight, neighbourIndex }
while (PQ.size() > 0)
    v = PQ.dequeue() // dequeue from PQ (O(log(V))
    if (!visited[v]) // process each edge once (O(E))
        MST.add(v)
        foreach (neighbour in neighbours(v))
            PQ.enqueue(neighbour) // insert into PQ (O(log(V))
```

1.2 Kruskal's ($O(E \log V)$)

1.2.1 Pseudocode

```
processed = new bool[E]
foreach (edge in edgeList) // edge list sorted by edge weight (PQ)
    if (!MST.contains(edge) && adding edge does not form cycle)
        MST.add(edge)
```

Single Source Shortest Path (SSSP)

1 Modified BFS ($O(V + E)$)

Works on unweighted DAG only

2 Bellman Ford ($O(VE)$)

```
// init SSSP
dist = new int[V]
dist.fill(INF) // all distances default to INF
dist[src] = 0

foreach v in V // O(V)
    foreach e in E // O(E)
        // relax
        if (dist[e.to] > dist[e.from] + e.weight):
            dist[e.to] = dist[e.from] + e.weight
```

At the end of run, `dist[v]` is shortest path from `src` to `v`: if no negative edge weight exists.

To check for negative cycle

After Bellman Ford, run

```
foreach e in E // O(E)
    if (dist[e.to] > dist[e.from] + w[u, v])
        negative cycle exists
```

3 Original Dijkstra's ($O((V + E) \log V)$)

Does not work with negative edge weights

```
foreach v in vertices(G)
    PQ.enqueue({ v, INF }) // all dist defaults to INF (sorted by weight)
PQ.set(src, 0) // except src
while (PQ.size() > 0)
    { v, weight } = PQ.dequeue() // get vertex with smallest dist
    relax each neighbour edge of v
```

4 Modified Dijkstra's ($O((V + E) \log V)$)

```
dist = new int[V]
dist.fill(INF)
dist[src] = 0
PQ.enqueue({ dist[src], src })
while (!PQ.empty())
{ d, v } = PQ.dequeue()
if (d == dist[v]) // ensure its the latest known value
    foreach (neighbourEdge to v)
        if (dist[neighbour] > dist[v] + neighbourEdge.weight)
            dist[neighbour] = dist[v] + neighbourEdge.weight)
            PQ.enqueue({ dist[neighbour], neighbour })
```

5 DFS/BFS for Trees ($O(V)$)

Only one path from 1 vertex to another so DFS works?

6 1 pass Bellman Ford for DAG ($O(V + E)$)

Use $O(V + E)$ DFS to get toposort. Then do pass of Bellman Ford

Dynamic Programming/Algorithms on DAG

1 SS(S|L)P on DAG ($O(V+E)$)

Single-source shortest/longest paths on DAG can be found in $O(V+E)$ using topological sort $O(V+E)$ then relax/stretch $O(V+E)$ each outgoing edges of vertices in topological order.

Bottom-up DP.

Since vertices are processed in topological order, there is no way to go from v_0 to v_1, since there is no cycle

1.1 Pseudocode

```
dist = new int[V]
dist.fill(-INF)
dist[src] = 0
G = toposort(G, src)
foreach (v in V(G))
    foreach (adjEdge of v)
        if (dist[adjEdge.dest] < dist[node] + adjEdge.weight)
            dist[adjEdge.dest] = dist[node] + adjEdge.weight
```

2 Longest Increasing Subsequence

2.1 Pseudocode

```
lis(v):
    if (v == N-1) // last node
        return 1
    if (memo[v])
        return memo[v]
```

```
    int ans = 1
    for (int j = v+1; j < N; j++)
        if (v.value < j.value) // implicit edge
            ans = max(ans, lis(j+1))
    memo[v] = ans
    return ans
```

3 Counting Paths in DAG

3.1 Pseudocode (Top Down, recursion with memo starting from dest)

```
numPathsTopDown(v):
    if (v == V-1) // last node (dest)
        return 1
    if (memo[v])
        return memo[v]
    int ans = 0
    foreach (neighbour of v)
        ans += numPaths(neighbour)
    memo[v] = ans
    return ans
```

3.2 Pseudocode (Bottom Up, start from source)

```
numPathsBottomUp(v):
    ways = new int[V]
    ways.fill(0)
    ways[v] = 1
    foreach (node in toposort(G, v)) // propagate info in topo-order, starting from v
        foreach (neighbour of node)
            ways[neighbour] += ways[node]
```

4 Travelling Salesman Problem

```
// where visited is a bitmask keeping track of nodes visited before
```

```
tsp(v, visited):
    if (allVisited(visited))
        return weight(v, 0) // all visited go back to source
    if (memo[v][visited])
        return memo[v][visited]

    memo[v][visited] = INF
    foreach (neighbour of v) // v in V if a complete graph
        if (visited[neighbour])
            memo[v][visited] = min(memo[v][visited],
                weight(v, neighbour) + tsp(neighbour, visited+=neighbour))
    return memo[v][visited]
```

Space complexity: $O(N \times 2^N)$

Time complexity: $O(N^2 \times 2^N)$

All Pairs Shortest Path

1 Floyd Warshall's ($O(V^3)$)

```
for (intermediate in V)
    for (src in V)
        for (dest in V)
            dist[src][dest] = min(
                dist[src][dest],
                dist[src][intermediate] + dist[intermediate][dest])
```

1.1 Print actual SP

Using predecessor matrix. Where `p[i][j]` is last vertex before `j`

```
if (dist[src][intermediate] + dist[intermediate][dest] < dist[src][dest])
    dist[src][dest] = dist[src][intermediate] + dist[intermediate][dest]
    p[src][dest] = intermediate
```

1.2 Transitive Closure Problem

Determine if vertex is connected to another.

```
connected[i][j] = connected[i][j] | connected[i][k] & connected[k][j]
```

1.3 Minimax/Maximin

Minimax: finding minimum of maximum edge weight along all possible paths from one vertex to another.

```
dist[i][j] = min(
        dist[i][j],
        max(dist[i][k], dist[k][j])
    )
```

1.4 Detect Any/-ve Cycle

Set the diagonal to INF, after Floyd Warshall, recheck diagonal. If its -ve, it means theres a -ve cycle. If its not infinity, it means theres a cycle