Chapter 4 to 7 Review

Trees

- A tree is a connected graph with no cycle
- **Rooted tree**: a connected graph with no cycle and a particular node called a root
  - **Parent node, child node, sibling node, ancestors, decendants, leaf nodes**
  - Depth of vertex - number of edges from the node to the tree's root node. A root node will have a depth of `0`.
  - Height of vertex - number of edges on the *longest path* from the node to a leaf. A leaf node will have a height of `0`.

Insert (Recursive)

```C++
if (root == nullptr) {
        return new TreeNode(key);
    }

    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }

    return root;
```

Delete (Recursive)

```C++
if (root == nullptr) {
        return root;
    }

    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        TreeNode* temp = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);
    }
```

Find Min (Recursive)

```C++
TreeNode* current = node;

    while (current->left != nullptr) {
        current = current->left;
    }

    return current;
```

A **binary tree** is a data structure in which each node has at most two children, referred to as the left child and the right child. Every node, except for the leaves, has exactly one parent. The left and right children of a node are distinct, and the difference in the number of nodes between the left and right subtrees of any node is at most one.

A **complete binary tree** is a special type of binary tree in which all levels are completely filled, except possibly for the last level, which is filled from left to right. In other words, all nodes are as left as possible in each level.

Pre-order Traversal:

1. Visit the root node.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

In-order Traversal:

1. Traverse the left subtree in in-order.
2. Visit the root node.
3. Traverse the right subtree in in-order.

Post-order Traversal:

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root node.

Level-order Traversal (Breadth-First Traversal):

1. Start at the root node.
2. Traverse each level of the tree from left to right.

Binary Search Tree

- There is an order relation ≤ defined for the vertices of B
- For any vertex v, and and descendant u of v.left u ≤ v
- For any vertex w, and and descendant u of w.right u ≤ w
- also known as a *totally ordered tree*

Operation	Description
Insert	Start at the root. Compare the key to be inserted with the key of the current node. If the key is less, move to the left subtree; if greater, move to the right subtree. Repeat this process until reaching a null position, then insert the new node with the given key at this position.
Delete	Search for the node to be deleted starting at the root. Identify different cases: if the node has no children (a leaf node), remove it; if the node has one child, replace the node with its child; if the node has two children, find the in-order successor (or predecessor), replace the node's key with the successor's (or predecessor's) key, and then recursively delete the successor (or predecessor).

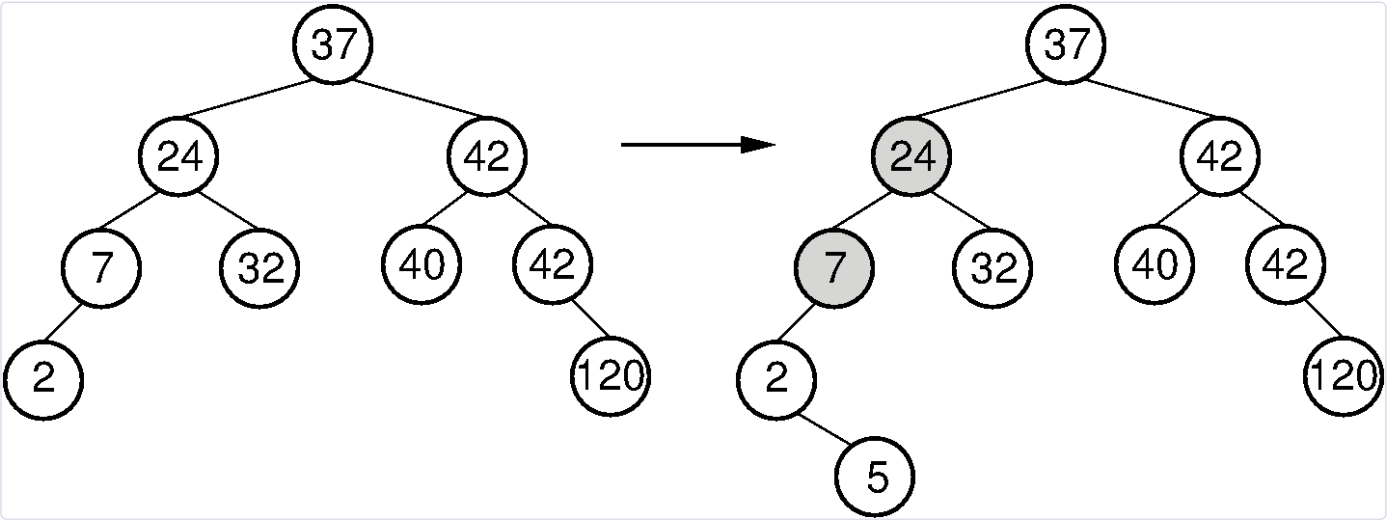
Operation	Description
Search	Start at the root. Compare the key to be searched with the key of the current node. If the key is equal to the current node's key, you have found the node. If the key is less, move to the left subtree; if greater, move to the right subtree. Repeat this process until you find the node with the matching key or reach a null position, indicating the key is not in the tree.

More info on binary search trees can be found in Trees 2 🌲

AVL Tree

An **AVL tree** is a binary search tree with the balance condition, for every node in the tree, height of left and right subtree can differ at most by 1. An avl tree is maintained by fixing the nodes that violate this condition after each insertion or deletion. Fixing is done through either single or double rotation.

Check from the parent of the newly inserted node upwards to the root to ensure that the AVL tree is not being violated by a newly inserted node.



B-tree

A **b-tree** is a m-ary search tree with the following balance restrictions

- Data items are stored at the leaves
- Non leaf nodes store up to `M-1` keys to guide the searching, keys are sorted within a node, Key i represents the smallest key in the subtree (i+1)
- The root can either be a lead, or have between 2 to M children
- All non-leaf nodes except the root have between `ceil(M/2)` and `M` children
- All leaves are stored at the same and have between `ceil(l/2)` and `l` data items for some `l`
- In other words, each node except for the root is at least half full

---

Hash Table

Hash table is a vector of lists, conflicts are resolved with the list. Multiple items can be stored in one list.

- If collision occurs try another cell to look
  - Try cells $h_0(x), h_1(x), \ldots$ in succession until a free cell is found
  - This is called **linear probing**
- A partially ordered tree (POT) is a tree T such that

- There is an order relation ≤ defined for the vertices of T
- For any vertex p and any child c of p, p ≤ c
- Partially ordered complete binary tree
  - Allowing vector representation of the tree

For more on hash tables view... Hash Table and Hash Table 2

Priority Queue

Each job within a computer takes turns using the cpu. If a job comes earlier than another job it needs to be executed earlier. If a job has been scheduled, it should not be scheduled a second time if there exists a job that hasn't been scheduled once. The queue is the answer.

Vector Representation

- sorting the tree in level order continuously
- may start from index 0 or index 1
  - Different starting affects the trees navigation

Heap operations

- DeleteMIN decreased the heap size by one
  - Move the last element to the root and percolate down from the root

Percolate Up (Heapify Up):

This operation is typically used during the insertion of a new element into the priority queue.

1. **Insert Element at the Bottom:**
   - Insert the new element at the bottom of the heap (last position).
2. **Compare with Parent and Swap:**
   - Compare the value of the new element with its parent.
   - If the value of the new element is greater (for a max-heap) or smaller (for a min-heap) than its parent, swap the element with its parent.
   - Repeat this process until the heap order property is restored.

Percolate Down (Heapify Down):

This operation is typically used during extraction or removal of the top element from the priority queue.

1. **Replace Top with Bottom:**
   - Replace the top element (the root) with the last element at the bottom of the heap.
2. **Compare with Children and Swap:**
   - Compare the value of the new root with its children.
   - If the value is smaller (for a max-heap) or greater (for a min-heap) than at least one of its children, swap the element with the smaller (for a max-heap) or larger (for a min-heap) child.
   - Repeat this process until the heap order property is restored.

Building the heap

- Inserting one by one resulting in O(N log N) heap building algorithms

- A better O(N) heap building algorithm builds head from bottom up by percolate elements backward (from N/2 to 1).