哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

# 数据结构
# Data Structures

## **Chapter 12** ADT Search: A Review

Prof. Yitian Shao
School of Computer Science and Technology

# A Review of ADT Search

- Unsorted Array (Vector)
- Linked List

} O(N)

- Stack
- Queue

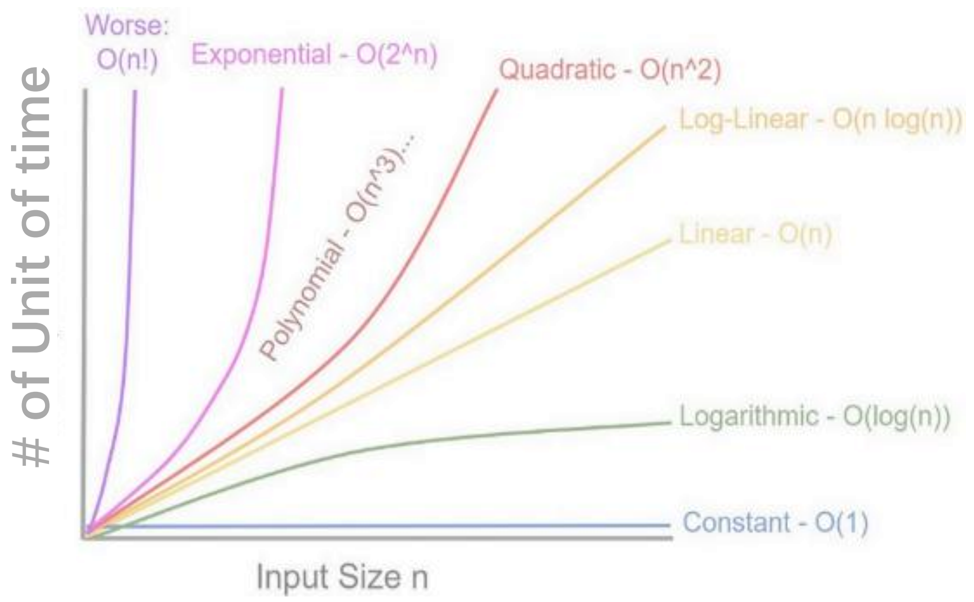
Priority Queue

} Can only access the top/front/back: O(1)

- String O(N)
- Sorted Array (Vector)
- Binary Search Tree

} O(logN)

- Graph O(E+N) BFS/DFS (E edges + N nodes) O(ElogN) Dijkstra’s algorithm
- Hash Map O(1)



# Array (Vector)

- Vector is a collection of elements, a dynamically-resizable array

**Element access**

| | |
|---|---|
| **at(i)** | access element i with bounds checking |
| **operator[]** | access specified element |

**Capacity**

| | |
|---|---|
| **empty()** | checks whether the container is empty |
| **size()** | returns the number of elements |

**Modifiers**

| | |
|---|---|
| **insert(i, value)** | inserts elements at position i |
| **erase(i)** | erases elements at position i |
| **push_back(value)** | adds an element to the end |

# In-Class Exercise: Vector

## 2951. Find the Peaks

Easy | Topics | Companies | Hint

You are given a **0-indexed** array `mountain`. Your task is to find all the **peaks** in the `mountain` array.

Return *an array that consists of* indices *of* ***peaks*** *in the given array in* ***any order***.

**Notes:**

- A **peak** is defined as an element that is **strictly greater** than its neighboring elements.
- The first and last elements of the array are **not** a peak.

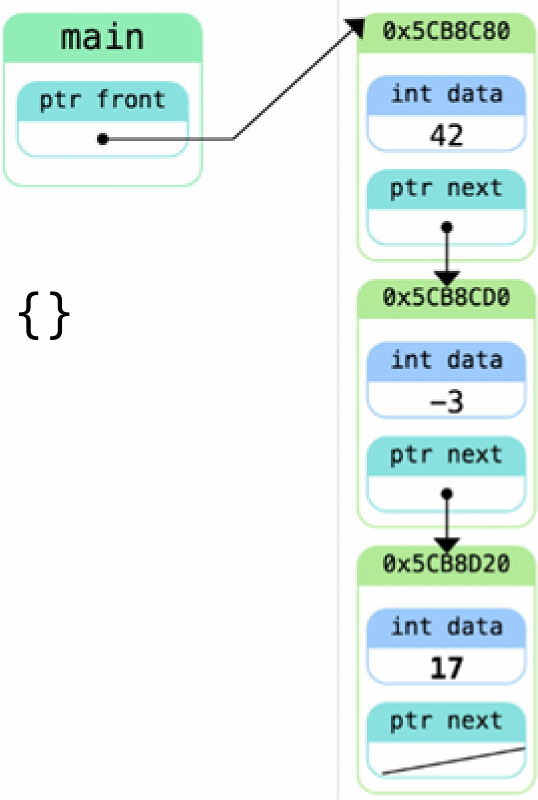
**Further question: How to find the highest peak?**

# Linked List

- Linked List stores elements chained together in a sequence

```
struct ListNode {
     int val;           //value of the element data
     ListNode* next;    //pointer to the next element
     ListNode(int x) : val(x), next(nullptr) {}
     ListNode(int x, ListNode *next) : val(x), next(next) {}
};

int main(){
    ListNode* front = new ListNode(42);
    front->next = new ListNode(-3);
    front->next->next = new ListNode(17);
}
```



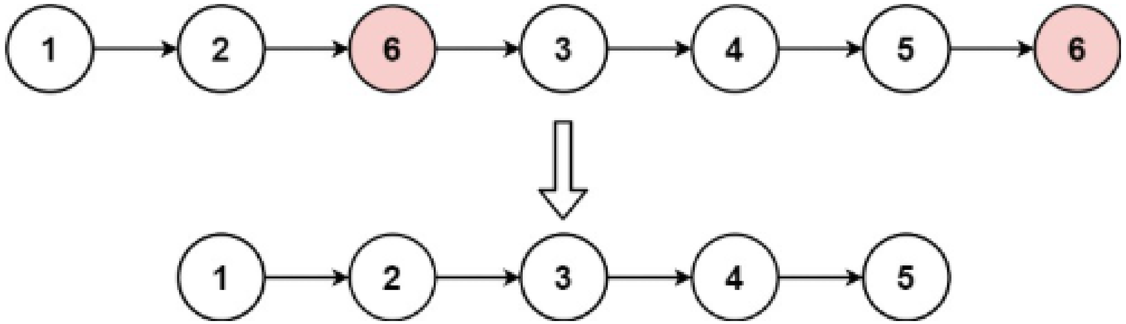
# In-Class Exercise: Listed List

## 203. Remove Linked List Elements

Easy Topics Companies

Given the `head` of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

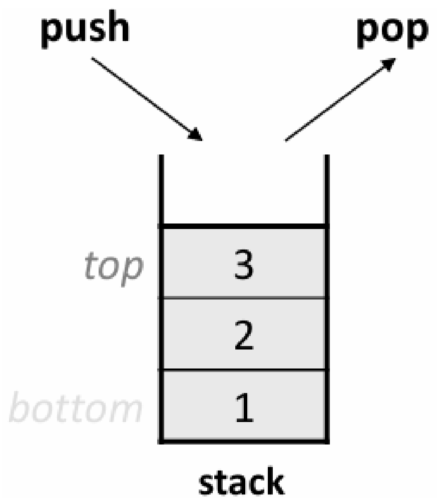
**Example 1:**

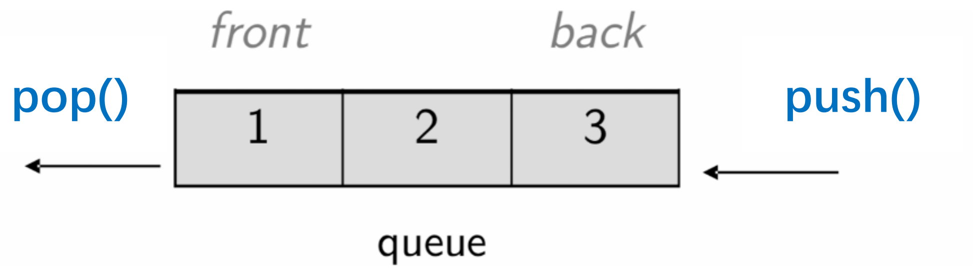



> **Input:** head = [1,2,6,3,4,5,6], val = 6
> **Output:** [1,2,3,4,5]

# Stack and Queue

- Only need to access the “last” element added to the storage
- Stack: Last In First Out (LIFO)
- Queue: First In First Out (FIFO)



# In-class Exercise: Stack

## 1047. Remove All Adjacent Duplicates In String

Easy Topics Companies Hint

You are given a string `s` consisting of lowercase English letters. A **duplicate removal** consists of choosing two **adjacent** and **equal** letters and removing them.

We repeatedly make **duplicate removals** on `s` until we no longer can.

Return *the final string after all such duplicate removals have been made*. It can be proven that the answer is **unique**.

**Example 1:**

```
Input: s = "abbaca"
Output: "ca"
Explanation:
For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and this is the only
possible move.  The result of this move is that the string is "aaca", of which only "aa" is possible, so the final
string is "ca".
```

**Example 2:**

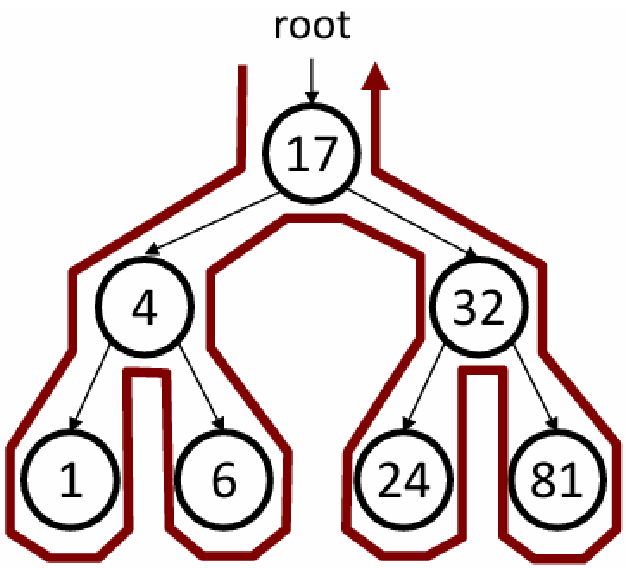
```
Input: s = "azxxzy"
Output: "ay"
```

# Tree - Binary Search Tree

- pre-order: root prioritized (Root – Left - Right)
  **17→4→1→6→32→24→81**

- in-order: leftmost prioritized (Left - Root - Right)
  **1→4→6→17→24→32→81**

- post-order: child prioritized (Left - Right - Root)
  **1→6→4→24→81→32→17**




```
struct TreeNode {
     int data;
     TreeNode* left;
     TreeNode* right;
 };
```

# Tree - Binary Search Tree

```
// Returns whether this BST contains the given integer.
bool contain(TreeNode* node, int value) {
    if (node == nullptr) {
      return false;  // base case: not found here
    }
    else if (node->data == value) {
      return true;  // base case: found here
    }
    else if (node->data > value) {
      return contain(node->left, value);
    }
    else {  // root->data < value
      return contain(node->right, value);
    }
}
```

# In-class Exercise: Binary Search Tree

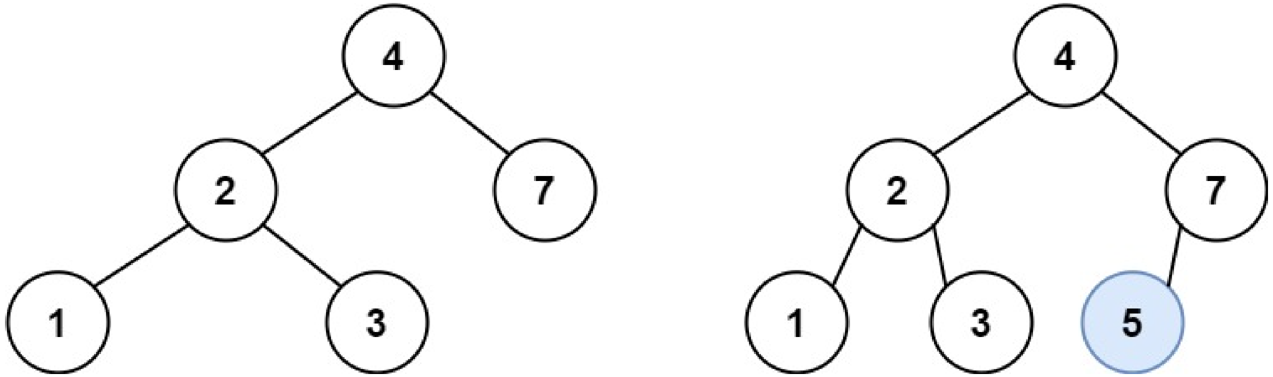
## 701. Insert into a Binary Search Tree

Medium | Topics | Companies

You are given the `root` node of a binary search tree (BST) and a `value` to insert into the tree. Return *the root node of the BST after the insertion*. It is **guaranteed** that the new value does not exist in the original BST.

**Notice** that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return **any of them**.

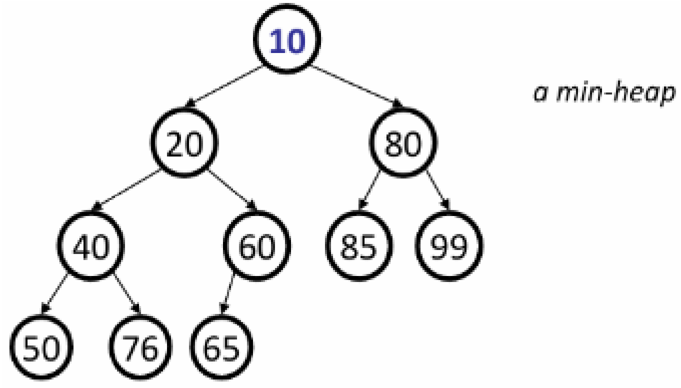
**Example 1:**




```
Input: root = [4,2,7,1,3], val = 5
Output: [4,2,7,1,3,5]
```

# Tree - Heap

- Heap is **complete binary tree** with vertical ordering:
- **Min-heap**: all children must be ⩾ parent's value
- **Max-heap**: all children must be ⩽ parent's value
- Can create a priority queue (prioritize minimum/maximum values)



# In-class Exercise: Priority-Queue

## 1845. Seat Reservation Manager

Medium | Topics | Companies | Hint

Design a system that manages the reservation state of `n` seats that are numbered from `1` to `n`.

Implement the `SeatManager` class:

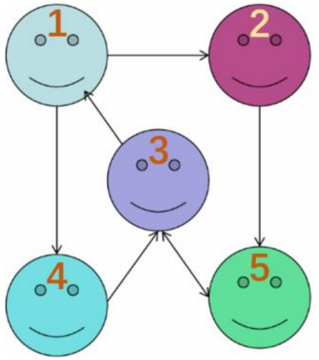
- `SeatManager(int n)` Initializes a `SeatManager` object that will manage `n` seats numbered from `1` to `n`. All seats are initially available.
- `int reserve()` Fetches the **smallest-numbered** unreserved seat, reserves it, and returns its number.
- `void unreserve(int seatNumber)` Unreserves the seat with the given `seatNumber`.

```
priority_queue<int, vector<int>, greater<int>> minHeap; // min heap in C++
```

# Graph DFS and BFS

- Graph consists of nodes and edges, with edges represent the relationships and nodes the items

```
vector<vector<int>> adjacencyList = {{2,4},{5},{5,1},{3},{3}};
vector<vector<int>> edges = {{1,2},{1,4},{2,5},{5,3},{3,5},{4,3},{3,1}};
vector<vector<int>> adjacencyMaxtrix = {{0,1,0,1,0},
                                        {0,0,0,0,1},
                                        {1,0,0,0,1},
                                        {0,0,1,0,0},
                                        {0,0,1,0,0}};
```



# In-class Exercise: Graph DFS and BFS

## 841. Keys and Rooms

Medium | Topics | Companies

There are `n` rooms labeled from `0` to `n - 1` and all the rooms are locked except for room `0`. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room `i`, return `true` *if you can visit* ***all*** *the rooms, or* `false` *otherwise*.

**Example 2:**

```
Input: rooms = [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can not enter room number 2 since the only key that unlocks it is in that room.
```

# In-class Exercise: DFS Solution

```
bool canVisitAllRooms(vector<vector<int>>& rooms)
{
    vector<int> vis(rooms.size(), 0);

    dfs(0, rooms, vis);

    for(int v:vis)
        if(v==0) return false;

    return true;
}
```

```
void dfs(int i, vector<vector<int>>& rooms, vector<int>& visited)
{
    visited[i] = 1;
    for(int key : rooms[i]){
        if(visited[key] == 0){
            dfs(key, rooms, visited);
        }
    }
    return;
}
```

# In-class Exercise: BFS Solution

```
bool canVisitAllRooms(vector<vector<int>>& rooms)
{
    vector<int> vis(rooms.size(), 0);

    bfs(0, rooms, vis);

    for(int v:vis)
        if(v==0) return false;

    return true;
}
```

```
void bfs(int i, vector<vector<int>>& rooms,
                                  vector<int>& visited)
{
    queue<int> q;
    q.push(i);
    while(!q.empty()){
        int curr= q.front();
        q.pop();
        for(int key : rooms[curr]){
           if(visited[key] == 0){
              q.push(key);
              visited[key] = 1;
           }
        }
    }
    return;
}
```