# Array methods

| Method | Description |
|---|---|
| `arr.size()` | Returns the number of elements |
| `arr.empty()` | Checks if array is empty |
| `arr.front()` | First element |
| `arr.back()` | Last element |
| `arr.at(i)` | Access with bounds checking (throws exception if out of bounds) |
| `arr[i]` | Direct access (no bounds check, like raw arrays) |
| `arr.fill(val)` | Fill entire array with `val` |
| `arr.begin()/end()` | Iterators for loops or STL algorithms |
| `arr.data()` | Pointer to underlying array (C-style) |
| `std::sort(begin, end)` | Sort elements |
| `std::reverse(begin, end)` | Reverse order |
| `std::find(begin, end, val)` | Find first occurrence of `val` |
| `std::count(begin, end, val)` | Count how many times `val` appears |
| `std::accumulate(begin, end, 0)` | Sum elements (needs `<numeric>`) |
| `std::max_element(begin, end)` | Get iterator to max element |
| `std::min_element(begin, end)` | Get iterator to min element |
| `std::binary_search(begin, end, val)` | `check if val exists` |

# Vector methods

| **Method** | **Description** |
|---|---|
| `push_back(val)` | Adds an element to the end of the vector |
| `pop_back()` | Removes the last element |
| `size()` | Returns the number of elements in the vector |
| `capacity()` | Returns the total capacity before reallocation is needed |
| `empty()` | Returns `true` if the vector is empty |
| `clear()` | Removes all elements |
| `at(index)` | Access element with bounds checking |
| `operator[]` | Access element without bounds checking (faster, but risky) |
| `insert(pos, val)` | Inserts `val` before position `pos` |
| `erase(pos)` | Removes the element at position `pos` |
| `erase(start, end)` | Removes elements in the range `[start, end)` |
| `resize(n)` | Resizes the vector to contain `n` elements |
| `begin()/end()` | Iterators to the beginning and end (for loops, algorithms, etc.) |
| `front()/back()` | Access first or last element |
| `swap(v2)` | Swaps contents with another vector |
| `assign(n, val)` | Assigns `n` copies of `val` to the vector |
| `emplace_back(val)` | Constructs element in-place at the end (faster than `push_back`) |

# String methods

| Method | Description |
|---|---|
| `s.length()` / `s.size()` | Get string length |
| `s.empty()` | Check if string is empty |
| `s.clear()` | Clear contents of string |
| `s.push_back(c)` | Add character at the end |
| `s.pop_back()` | Remove last character |
| `s.substr(pos, len)` | Extract substring from position `pos` of length `len` |
| `s.find(sub)` | Find index of first occurrence of `sub`, returns `npos` if not found |
| `s.rfind(sub)` | Find last occurrence of `sub` |
| `s.find_first_of(chars)` | Find first occurrence of any char in `chars` |
| `s.find_last_of(chars)` | Find last occurrence of any char in `chars` |
| `s.replace(pos, len, str)` | Replace substring with new string |
| `s.insert(pos, str)` | Insert string at position `pos` |
| `s.erase(pos, len)` | Erase substring from `pos` of length `len` |
| `s.compare(str)` | Compare two strings (`0` = equal, `<0`, `>0`) |
| `std::to_string(num)` | Convert number to string |
| `stoi(s)`, `stol(s)`, etc. | Convert string to int/long/etc. |
| `s.begin()/end()` | Iterators (for loops, STL use) |
| `reverse(s.begin(), s.end())` | Reverse string using STL |
| `transform(...)` | Change case with `toupper`, `tolower`, etc. |

| Problem | **`Useful STL`** |
|---|---|
| Reverse a string | `reverse(s.begin(), s.end())` |
| Check palindrome | `Compare s with reversed copy` |
| Sort characters | `sort(s.begin(), s.end())` |
| Remove duplicates | `unique(s.begin(), s.end()) + erase` |
| Count frequency | `unordered_map<char,int> freq` |

# All types of linked list methods

## Singly Linked List – Super Useful Methods

Each node has `data` and `next`.

| Method | Description |
|---|---|
| `insertAtHead(val)` | Insert node at the beginning |
| `insertAtTail(val)` | Insert node at the end |
| `insertAtPosition(pos, val)` | Insert node at position `pos` |
| `deleteHead()` | Remove first node |
| `deleteTail()` | Remove last node |
| `deleteAtPosition(pos)` | Remove node at position `pos` |
| `search(val)` | Check if value exists |
| `reverse()` | Reverse the list (iterative or recursive) |
| `findMiddle()` | Find middle node using slow/fast pointers |
| `detectCycle()` | Floyd’s Cycle Detection Algorithm (Tortoise & Hare) |
| `removeCycle()` | If cycle detected, remove it |
| `length()` | Count nodes |
| `display()` | Print the list |

## ◆ Doubly Linked List – Extra Power

Each node has `data`, `prev`, and `next`.

| Additional Useful Methods | Description |
|---|---|
| `insertBefore(node, val)` | Insert before a given node |
| `insertAfter(node, val)` | Insert after a given node |
| `deleteNode(node)` | Delete a specific node in O(1) if you have a pointer to it |
| `reverse()` | Reverse the list (just swap `next` and `prev` pointers) |
| `traverseForward()` | Iterate from head to tail |
| `traverseBackward()` | Iterate from tail to head |

## ◆ Circular Linked List – Trickier Stuff

Last node points back to the head.

| Methods | Description |
|---|---|
| `insertAtEnd(val)` | Insert node so last node points to new node, and it points to head |
| `insertAtHead(val)` | Insert node and update tail's next to new head |
| `deleteNode(val)` | Delete node and maintain circularity |
| `display()` | Print list, stopping when you reach head again |

---

## ◆ STL `list` – Built-in Doubly Linked List (`#include <list>`)

| Method | Description |
|---|---|
| `list.push_front(val)` | Add at head |
| `list.push_back(val)` | Add at tail |
| `list.pop_front()` | Remove head |
| `list.pop_back()` | Remove tail |
| `list.insert(it, val)` | Insert at iterator position |
| `list.erase(it)` | Delete at iterator |
| `list.reverse()` | Reverse list |
| `list.sort()` | Sort list |
| `list.remove(val)` | Remove all occurrences of `val` |
| `list.clear()` | Empty the list |
| `list.size()` | Number of elements |
| `list.begin()/end()` | Iterators to start/end |

## 🔥 Bonus Techniques:

- **Slow/Fast Pointers** → Detect cycles, find middle, etc.
- **Dummy Head Node** → Simplifies insertion/deletion at head.
- **Merge Two Lists** → Merging sorted linked lists.
- **Recursive Reverse** → Mind-bender but elegant.
- **K-group Reversal** → Advanced but useful in coding interviews.

# Hashing methods

## ◆ C++ Hashing Tools (from `<unordered_map>`, `<unordered_set>`)

| Tool | What It Is |
|---|---|
| `unordered_map<Key,Val>` | Hash table for key-value pairs |
| `unordered_set<Key>` | Hash table for unique keys |
| `hash<T>()` | Built-in hash function for types like `int`, `string`, etc. |

---

## ◆ Super Useful Methods – `unordered_map`

| Method | Description |
|---|---|
| `umap[key] = val` | Inserts or updates key with value |
| `umap.at(key)` | Access value at key with bounds checking |
| `umap.find(key)` | Returns iterator to key or `umap.end()` if not found |
| `umap.count(key)` | Returns 1 if key exists, 0 otherwise |
| `umap.erase(key)` | Removes key (if it exists) |
| `umap.clear()` | Removes all elements |
| `umap.empty()` | Check if map is empty |
| `umap.size()` | Number of elements |
| `for (auto& [k, v] : umap)` | Range-based loop (C++17+) |

---

## ◆ Super Useful Methods – `unordered_set`

| Method | Description |
|---|---|
| `uset.insert(key)` | Insert key |

| Method | Description |
|---|---|
| `uset.find(key)` | Check existence |
| `uset.count(key)` | 1 if exists, 0 otherwise |
| `uset.erase(key)` | Remove key |
| `uset.size()` | Number of elements |
| `uset.clear()` | Clear the set |

---

# ◆ Hashing Tricks

### 1. Frequency Count (Hash Map)

```
cpp
CopyEdit
unordered_map<int, int> freq;
for (int x : nums) freq[x]++;
```

### 2. Check for Duplicates (Hash Set)

```
cpp
CopyEdit
unordered_set<int> seen;
for (int x : nums) {
    if (seen.count(x)) { /* duplicate found */ }
    seen.insert(x);
}
```

---

# ◆ Common Use Cases

- **Hash Map** → frequency count, memoization (DP), grouping
- **Hash Set** → remove duplicates, quick existence check
- **Custom Hashing** → use composite keys (e.g., tuples, pairs)

---

# 🔥 Bonus Tips:

- Use `reserve(n)` to avoid rehashing if inserting lots of elements.
- Hash collisions are rare but avoid using `unordered_map` with floats/doubles as keys — not precise.

# Stack and Queue methods

## C++ Stack Methods (`#include <stack>`)

| Method | Description |
|---|---|
| `s.push(val)` | Push value onto the top of the stack |
| `s.pop()` | Remove the top element |
| `s.top()` | Access the top element |
| `s.empty()` | Check if the stack is empty |
| `s.size()` | Get number of elements in the stack |

💡 Use Cases:

- **Undo/Redo**, **DFS**, **Balanced Parentheses**, **Backtracking**, **Reverse Data**

---

## ◆ C++ Queue Methods (`#include <queue>`)

| Method | Description |
|---|---|
| `q.push(val)` | Add value to the back |
| `q.pop()` | Remove from the front |
| `q.front()` | Access the front element |
| `q.back()` | Access the last element |
| `q.empty()` | Check if queue is empty |
| `q.size()` | Get number of elements |

💡 Use Cases:

- **BFS**, **Task Scheduling**, **Order Processing**, **Producer/Consumer**

---

## ◆ Deque (Double-Ended Queue) — Bonus Power (`#include <deque>`)

| Method | Description |
|---|---|
| `dq.push_front(val)` | Add to front |
| `dq.push_back(val)` | Add to back |
| `dq.pop_front()` | Remove from front |
| `dq.pop_back()` | Remove from back |
| `dq.front()` / `dq.back()` | Access ends |
| `dq.empty()` / `dq.size()` | Self-explanatory |

### 💡 Use Cases:

- **Sliding Window**, **Monotonic Queue**, **Palindromes**, **Advanced Scheduling**

---

## ◆ Priority Queue (a.k.a. Heap) — For Sorted Access (`#include <queue>`)

| Method | Description |
|---|---|
| `pq.push(val)` | Insert into heap |
| `pq.pop()` | Remove largest element (max-heap by default) |
| `pq.top()` | Access largest element |
| `pq.empty()` / `pq.size()` | Basics |

### 💡 Min-Heap Tip:

```
cpp
CopyEdit
priority_queue<int, vector<int>, greater<int>> minHeap;
```

### 💡 Use Cases:

- **Dijkstra's Algorithm**, **Top K Elements**, **Median Maintenance**, **Greedy Algos**

---

## ◆ Custom Stack/Queue Methods (From Scratch)

| Useful Methods to Implement | Stack | Queue |
|---|---|---|
| `push(val)` | Add to top | Add to rear |
| `pop()` | Remove top | Remove front |
| `peek()` / `top()` | See top | See front |
| `isEmpty()` | Check empty | Check empty |
| `size()` | Count elements | Count elements |
| `reverse()` | Reverse stack with aux stack | Reverse queue with stack |

---

## 🔥 Advanced Stack/Queue Tricks:

- **Implement Queue with 2 Stacks** and vice versa.
- **Monotonic Stack/Queue** → For next greater/smaller problems.
- **Two Queues to Implement Stack**.
- **Stack with Min/Max tracking**.

# Binary Tree methods

## ◆ Basic Binary Tree Methods

| Method | Description |
|---|---|
| `insert(val)` | Insert a node into the tree (BST or general tree logic) |
| `search(val)` | Search for a value |
| `delete(val)` | Delete a node (BST-specific with 3 cases) |
| `traverseInOrder()` | Left → Root → Right |
| `traversePreOrder()` | Root → Left → Right |
| `traversePostOrder()` | Left → Right → Root |
| `traverseLevelOrder()` | BFS using queue (level-by-level) |
| `height()` | Max depth of tree |
| `countNodes()` | Total number of nodes |
| `isBalanced()` | Check if balanced (height difference ≤ 1 at all nodes) |
| `isSymmetric()` | Check if tree is a mirror of itself |
| `maxValue()` / `minValue()` | Get max/min value (BST: rightmost/leftmost node) |
| `lowestCommonAncestor(n1,n2)` | Find LCA of two nodes |

## ◆ Advanced / Super Useful Utilities

| Method | Description |
|---|---|
| `diameter()` | Longest path between any two nodes |
| `invert()` / `mirror()` | Flip tree left ↔ right |
| `sumTree()` | Sum of all nodes |
| `isSubtree(Tree t2)` | Check if t2 is subtree of t1 |
| `flattenToLinkedList()` | Convert to linked list in-place (preorder) |
| `buildFromInPost(in[], post[])` | Build tree from inorder & postorder arrays |
| `printBoundary()` | Print boundary of tree (left + leaves + right) |

# Binary search Tree methods

## Core BST Methods

| Method | Description |
|---|---|
| `insert(root, val)` | Insert `val` in correct position, maintaining BST property |
| `deleteNode(root, val)` | Delete `val` from BST (handle 0, 1, 2 children cases) |
| `search(root, val)` | Find if value exists in BST (returns node or nullptr) |
| `inorder(root)` | Traversal — yields sorted order of values |
| `preorder(root)` | Traversal — useful for copying trees |
| `postorder(root)` | Traversal — useful for deleting tree nodes |
| `levelOrder(root)` | BFS traversal level-by-level |

---

## 🔥 Utility BST Methods

| Method | Description |
|---|---|
| `minValueNode(root)` | Get node with minimum value (leftmost node) |
| `maxValueNode(root)` | Get node with maximum value (rightmost node) |
| `height(root)` | Get tree height (depth) |
| `isBST(root, min, max)` | Check if tree is a valid BST |
| `findKthSmallest(root, k)` | Find the kth smallest value (inorder + counter) |
| `findKthLargest(root, k)` | Reverse inorder + counter |
| `floor(root, key)` | Greatest value ≤ key |
| `ceil(root, key)` | Smallest value ≥ key |
| `rangeSumBST(root, L, R)` | Sum of all values in range [L, R] |

---

## 🔥 Advanced BST Methods

| Method | Description |
|---|---|
| `lowestCommonAncestor(root, p, q)` | Find lowest common ancestor of nodes p and q |
| `trimBST(root, L, R)` | Trim BST so all elements fall within [L, R] |
| `convertToDLL(root)` | Convert BST to Doubly Linked List (inorder traversal) |
| `balanceBST(root)` | Balance an unbalanced BST (build from sorted inorder array) |
| `mergeBSTs(root1, root2)` | Merge two BSTs into one |
| `serializeBST(root)` | Store BST to string (preorder or inorder) |
| `deserializeBST(data)` | Rebuild BST from stored data |

## ◆ Common Interview Patterns w/ BST

- **Successor/Predecessor** (inorder successor/predecessor)
- **Validate BST** (check using min/max or inorder order)
- **Path Sum in BST**
- **BST to Balanced BST** (AVL/Red-Black → or via sorted array)
- **BST Iterator** → simulate in-order traversal with `next()`, `hasNext()` methods

## Heap methods

### ◆ STL Heaps in C++: `priority_queue`

| Method | Description |
|---|---|
| `pq.push(val)` | Add value to the heap |
| `pq.pop()` | Remove the top (max by default) |
| `pq.top()` | Get the top element |
| `pq.empty()` / `pq.size()` | Self-explanatory |

By default, it's a **Max-Heap**.

---

### ◆ Min-Heap in STL (Yes, you can!)

### ◆ Heap from Scratch – Core Methods (Array-based Heap)

| Method | Description |
|---|---|
| `insert(val)` | Add element and heapify up |
| `extractMin()/extractMax()` | Remove min/max and heapify down |
| `peekMin()/peekMax()` | Get min/max element without removal |
| `heapify(arr[], n)` | Build heap from array (O(n)) |
| `heapifyUp(index)` | Restore heap upwards |
| `heapifyDown(index)` | Restore heap downwards |

### ◆ Super Useful Heap Tricks

| Trick | Use Case |
|---|---|
| `make_heap(begin, end)` | Turn array/vector into a heap |
| `push_heap(begin, end)` | Push new element, re-heap |
| `pop_heap(begin, end)` | Move top to end, re-heap (you pop manually after this) |
| `sort_heap(begin, end)` | Heap sort (descending for max-heap) |

---

## 🔥 Must-Know Heap Applications

| Problem | Heap Use |
|---|---|
| Top K Elements | Min-Heap of size K |
| Kth Largest/Smallest Element | Min/Max Heap with size K |
| Merge K Sorted Lists | Min-Heap with next elements |
| Dijkstra’s Shortest Path | Min-Heap for choosing min distance |
| Median Stream | Two Heaps: Max-Heap (lower half), Min-Heap (upper) |
| Huffman Coding Tree | Min-Heap for frequencies |