

CS2040C Data Structures and Algorithms
AY23/24, Y1S2
Notes

Sim Ray En Ryan

December 3, 2025

Contents

| | | |
|----------|---|----------|
| 1 | Pointers and C++ Basics | 4 |
| 1.1 | C++ Fundamentals | 4 |
| 1.2 | Passing of Parameters | 4 |
| 2 | Linked Lists and ADTs | 4 |
| 2.1 | Linked Lists | 4 |
| 2.1.1 | Linked List Operations | 4 |
| 2.2 | Abstract Data Types (ADTs) | 5 |
| 2.2.1 | Stack (LIFO) | 5 |
| 2.2.2 | Queue (FIFO) | 5 |
| 3 | Searching and Sorting | 5 |
| 3.1 | Sorting Terminology | 5 |
| 3.2 | Sorting Algorithms and Complexities | 5 |
| 3.2.1 | Comparison-Based Sorts | 6 |
| 4 | Trees and Traversal | 6 |
| 4.1 | Tree Properties | 6 |
| 4.2 | Traversal Orders | 6 |
| 4.3 | AVL Tree | 6 |
| 4.3.1 | AVL Rotations | 7 |
| 5 | Augmented Trees | 7 |
| 5.1 | Augmenting with Rank/Weight | 7 |
| 5.1.1 | Operations using Rank | 7 |
| 5.1.2 | Range Query | 8 |
| 6 | Hash Tables | 8 |
| 6.1 | Hashing | 8 |
| 6.2 | Collision Resolution | 8 |
| 6.2.1 | Chaining | 8 |
| 6.2.2 | Open Addressing | 9 |
| 6.2.3 | Open Addressing vs. Chaining | 9 |
| 6.3 | Hash Table Size and Resizing | 9 |
| 7 | Heap and Union Find | 9 |
| 7.1 | Binary Heap | 9 |
| 7.1.1 | Heap Operations ($O(\log n)$) | 10 |
| 7.1.2 | Heap Sort | 10 |
| 7.2 | Union Find (Disjoint Set Union) | 10 |
| 7.2.1 | Union Find Implementations | 10 |

| | | |
|-----------|--|-----------|
| 8 | Graph Representation and Search | 11 |
| 8.1 | Graph Definitions | 11 |
| 8.2 | Graph Representation | 12 |
| 8.2.1 | Adjacency List (Use if sparse) | 12 |
| 8.2.2 | Adjacency Matrix (Use if dense) | 12 |
| 8.3 | Graph Search | 12 |
| 8.3.1 | Breadth-First Search (BFS) (Level Order) | 12 |
| 8.3.2 | Depth-First Search (DFS) | 12 |
| 9 | Shortest Path and Topological Sort | 13 |
| 9.1 | Single-Source Shortest Path (SSSP) | 13 |
| 9.1.1 | Dijkstra's Algorithm | 13 |
| 9.1.2 | Bellman-Ford Algorithm | 13 |
| 9.2 | Topological Sort | 13 |
| 10 | Minimum Spanning Trees (MST) | 14 |
| 10.1 | Spanning Tree Properties | 14 |
| 10.2 | MST Algorithms | 14 |
| 10.2.1 | Prim's Algorithm | 14 |
| 10.2.2 | Kruskal's Algorithm | 14 |
| 10.2.3 | Boruvka's Algorithm | 14 |
| 10.3 | MST Variants | 15 |
| 11 | Computational Geometry and Convex Hull | 15 |
| 11.1 | Computational Geometry | 15 |

1 Pointers and C++ Basics

1.1 C++ Fundamentals

- `using namespace std` shortens common keywords like `cin`, `cout`, and `endl`.
- **Function Overloading:** Possible if functions with the same name have different signatures (i.e., different input parameters).

1.2 Passing of Parameters

- **Pass by Value:** Only the value is passed. Any changes made inside the function remain local to that function.
- **Pass by Reference:**
 - Indicated by `type&` in the function parameter list.
 - Passes a reference to the input variable. Changes affect the original variable.
- **Pass by Pointer:**
 - The address is passed using `&x`.
 - Indicated by `type*` in the function parameter list.
 - Dereferenced by `*x` within the function to access the value.

2 Linked Lists and ADTs

2.1 Linked Lists

- Arrays are inefficient as space must be declared beforehand.
- **Linked Lists** are **dynamic** and use only the required amount of memory.
- Insertions are generally fast ($O(1)$) at the head.
- A basic linked list typically involves:
 - `class ListNode` with `int value` and `ListNode *next`.
 - `class List` with `int size` and `ListNode *head`.

2.1.1 Linked List Operations

- **insert(int):** Create new `ListNode`, make it point to the current head, then make it the new head. Increase size.

- **removeHead(int)**: Create a temporary pointer to the head, set the current head to where it points, **delete the temporary pointer** (to prevent memory leaks), and decrease size.
- Other operations include `removeAtTail()`, `removePost(int)`, and `removeItem(int)`.

2.2 Abstract Data Types (ADTs)

2.2.1 Stack (LIFO)

- **Last In, First Out (LIFO)**.
- Interface operations: `push(x)`, `pop()`, `empty()`.

2.2.2 Queue (FIFO)

- **First In, First Out (FIFO)**.
- Interface operations: `Enqueue`, `Dequeue`.

3 Searching and Sorting

3.1 Sorting Terminology

- **Comparison-based** sorting relies on comparisons between elements.
- **Recursion** involves a function calling itself.
- **Stability**: A sorting algorithm is stable if elements with equal keys maintain their relative order.
- **In-place**: An algorithm that requires a constant amount of extra space, $O(1)$, relative to the input size.

3.2 Sorting Algorithms and Complexities

Time complexity is given for Best (Ω), Average (θ), and Worst (O) cases.

| Algorithm | Time (Ω) | Time (θ) | Time (O) | Space | Stable |
|----------------|-------------------|-------------------|---------------|--------|--------|
| Bubble Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | Yes |
| Merge Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(N)$ | Yes |
| Quick Sort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(1)$ | No |
| Counting Sort | | | | | |
| Radix Sort | | | | | |

3.2.1 Comparison-Based Sorts

- **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order. The largest element(s) are moved to the last positions.
- **Selection Sort:** Finds the minimum element and places it at the beginning of the unsorted part. Elements are sorted at the front or back without gaps.
- **Insertion Sort:** Builds the final sorted array one item at a time by repeatedly inserting the next element into the already sorted portion. Some elements are sorted with gaps, while others remain completely unchanged from the original.
- **Merge Sort:** Divides the array into two halves, recursively sorts them, and then merges the sorted halves. Equal sizes of subarrays are sorted locally but not globally.
- **Quick Sort:** Selects a pivot element and partitions the other elements into two subarrays according to whether they are smaller or larger than the pivot. The pivot is placed in its correct sorted position. Not stable unless duplicate packing is used.

4 Trees and Traversal

4.1 Tree Properties

- Trees can be **Balanced** or **Unbalanced**.
- Common operations: **Search**, **Insert**, **Remove**.

4.2 Traversal Orders

- **Pre-Order:** Root \rightarrow Left \rightarrow Right.
- **In-Order:** Left \rightarrow Root \rightarrow Right.
- **Post-Order:** Left \rightarrow Right \rightarrow Root.

4.3 AVL Tree

An AVL tree is a self-balancing Binary Search Tree (BST).

- Height of a null node is -1 .
- $\text{height} = \max(\text{left.height}, \text{right.height}) + 1$.
- **Balance Factor (BF):** $\text{left.height} - \text{right.height}$.
- A node is **height-balanced** if $-1 \leq \text{BF} \leq 1$.
- **Left-heavy** if $\text{BF} > 0$, **Right-heavy** if $\text{BF} < 0$.

- An AVL tree with h height has at least $n > 2^{h/2}$ nodes.
- **Insertion:** Requires at most two rotations (single or double rotation).
- **Deletion:** May require $O(\log n)$ rotations. Nodes can be marked as deleted instead of being physically removed.

4.3.1 AVL Rotations

- If $v.BF > 1$ (Left-heavy):
 - If $v.left.BF < 0$: Perform **Left Rotate**($v.left$) (Left-Right case).
 - Perform **Right Rotate**(v) (Left-Left case).
- If $v.BF < -1$ (Right-heavy):
 - If $v.right.BF > 0$: Perform **Right Rotate**($v.right$) (Right-Left case).
 - Perform **Left Rotate**(v) (Right-Right case).

5 Augmented Trees

5.1 Augmenting with Rank/Weight

Each node v stores a **Rank** or **Weight** $\omega(v)$, representing the size of its subtree.

$$\omega(v) = \omega(v.left) + \omega(v.right) + 1$$

5.1.1 Operations using Rank

- **Select(k):** Returns the node with the k -th smallest key.
 1. Calculate $rank = \omega(v.left) + 1$.
 2. If $k = rank$, return v .
 3. If $k < rank$, recurse on $v.left$ with k .
 4. If $k > rank$, recurse on $v.right$ with $k - rank$.
- **Rank(v):** Returns the rank of node v in the overall tree.
 1. Start with $rank = \omega(v.left) + 1$.
 2. While traversing up to the root: if v is a right child, $rank+ = \omega(v.parent.left) + 1$.
- **Updating Weight:** During a rotation, only the weights of the two involved nodes (v and $v.left$ or $v.right$) need to be updated.

5.1.2 Range Query

- **Range Count (RC):** $RC(a, b)$ counts nodes with keys between a and b using the Rank operation.
- **Range List Query (RLQ):** Listing all nodes in a range $[k_1, k_2]$ takes $O(n)$ in the worst case as it traverses the tree:

```
RLQ(Node<T>* node, int k1, int k2) {  
    if (node == nullptr) return;  
    RLQ(node->left, k1, k2);  
    if (k1 <= node->data <= k2) cout << node->data;  
    RLQ(node->right, k1, k2);  
}
```

6 Hash Tables

6.1 Hashing

- **Hash Functions** are required to be:
 - **Fast** ($O(1)$).
 - Capable of mapping a large input space (non-integer or large integer) to a small integer (the bucket index).
 - Designed for **low collisions**.
- **Division Method:** $h(k) = k \pmod{m}$ where m is the size of the hash table. m should ideally be a prime number. Division can be slow.
- **Multiplication Method:** Generally faster than division. $h(k) = (Ak \pmod{2^w}) \gg (w - r)$, where $m = 2^r$ and A is an odd integer constant.

6.2 Collision Resolution

It is impossible to prevent collisions (Pigeonhole Principle).

6.2.1 Chaining

- Each bucket stores a **linked list** of keys that hash to that index.
- **Worst Case:** All keys hash to the same bucket, resulting in $O(n)$ search time.
- **Expected Search Time:** $O(1 + n/m)$, where n is the number of items and m is the table size. If $m > n$ (low load), expected time is $O(1)$.
- Can still add new items even when $m < n$.

6.2.2 Open Addressing

Keys are stored in the table itself. Requires setting deleted slots to **DELETED** (not **null**) for correct searching.

- **Search:** Bucket $h(\text{key}, i)$. Iterate $i = 0, 1, 2, \dots$ until empty/deleted, or found. Stops if $i > m$.
- **Insert:** Same probing, but set key if slot is **null** or **DELETED**.
- **Linear Probing:** $h(\text{key}, i) = (h(\text{key}) + i) \pmod{m}$. Guarantees every bucket can be reached. Highly sensitive to **primary clustering**.
- **Quadratic Probing:** $h(\text{key}, i) = (h(\text{key}) + i^2) \pmod{m}$. Reduces primary clustering, but still sensitive to secondary clustering.
- **Double Hashing:** $h(\text{key}, i) = (h(\text{key}) + i \times g(\text{key})) \pmod{m}$. Reduces clustering effects.

6.2.3 Open Addressing vs. Chaining

- **Open Addressing:** Space is better (no list node allocations). Better cache performance. More sensitive to clustering. Performance degrades rapidly as load factor approaches 1.
- **Load Factor** (n/m): Expected probes for Open Addressing is $O(1/(1 - \text{load}))$. E.g., at 90% load, expected probes is 10.

6.3 Hash Table Size and Resizing

- If table size $M < 2n$, too many collisions (high load). If $M > 10n$, space is wasted.
- N (number of items) is often not known in advance, requiring a **Variable** M (resizing).
- **Resizing Cost:** $O(m_1)$ to scan the old table + $O(m_2)$ to create the new table + $O(n)$ to transfer items (each $O(1)$ transfer). Total time is dominated by $O(n)$ rehashing.

7 Heap and Union Find

7.1 Binary Heap

- A **Complete Binary Tree** that satisfies the **Heap Ordering** property: $\text{priority}(\text{parent}) \geq \text{priority}(\text{child})$.
- The largest element is always at the root. No rotations are needed to balance, only bubbling.

- **Complete Binary Tree:** Every level is full, except possibly the last, which is filled from left to right. Height is $O(\log n)$.
- **Storage:** Can be efficiently stored in an array:
 - $\text{parent}(x) = \lfloor (x - 1)/2 \rfloor$
 - $\text{left}(x) = 2x + 1$
 - $\text{right}(x) = 2x + 2$

7.1.1 Heap Operations ($O(\log n)$)

- **Insert:** Add at the next empty spot, then **bubble up**.
- **Extract Max:** Replace root with the last element, remove the last element, then **bubble down**.
- **Increase Key:** Change priority, then **bubble up**.
- **Decrease Key:** Change priority, then **bubble down** to the larger child.
- **Delete:** Swap the node with the last element, remove the last element, then **bubble down**.

7.1.2 Heap Sort

- **Heap Sort:** Repeatedly calls `extractMax()`. Time complexity $O(n \log n)$. It is **In-Place** but **Unstable**.
- **Heapify v2:** Builds the heap in $O(n)$ time by calling `BubbleDown` on elements starting from the last non-leaf node up to the root.
- Using a Ternary Tree instead of a Binary Tree makes heap sort slightly faster.

7.2 Union Find (Disjoint Set Union)

Used to maintain a collection of disjoint sets.

- **Union(x, y):** Connects the sets containing x and y .
- **Find(x):** Returns the component identifier (root) of x .
- The connectivity property is **Transitive**.

7.2.1 Union Find Implementations

| Version | Find | Union | Data Structure |
|------------------------------|------------------------|------------------------|--------------------------------------|
| Quick Find (V1) | $O(1)$ | $O(n)$ | Array (component ID) |
| Quick Union (V2) | $O(n)$ | $O(n)$ | Array (parent pointer) |
| Weighted Union (V3) | $O(\log n)$ | $O(\log n)$ | Array (parent pointer + size/height) |
| Path Compression (V4) | $\approx O(\alpha(n))$ | $\approx O(\alpha(n))$ | Array (parent pointer) |

Where $\alpha(n)$ is the inverse Ackermann function, which grows extremely slowly (i.e., $\alpha(n) \leq 5$ for all practical n).

- **Quick Find (V1)**: Each object stores a component identifier. Union requires updating the ID for every object in the merged set ($O(n)$).
- **Quick Union (V2)**: Each object stores a parent identifier, forming a forest of trees. $\text{Union}(x, y)$ sets the root of y to point to the root of x .
- **Weighted Union (V3)**: When merging two trees, the root of the smaller tree (by size or height) is attached to the root of the larger tree. Limits maximum depth to $O(\log n)$.
- **Path Compression (V4)**: During the FindRoot operation, every traversed node's parent is set directly to the root.

```
findRoot(int p) {
    root = p;
    while (parent[root] != root) root = parent[root];
    // Path Compression:
    while (parent[p] != root) {
        temp = parent[p];
        parent[p] = root;
        p = temp;
    }
    return root;
}
```

8 Graph Representation and Search

8.1 Graph Definitions

A graph $G = (V, E)$ consists of **Nodes/Vertices** (V) and **Edges/Arcs** (E).

- **Degree**: The number of adjacent edges to a vertex. The **Degree of Graph** is $\max(\text{degree}(V))$.
- **Diameter**: The maximum distance between any two nodes, using a shortest path.
- **Connectedness**: Every pair of nodes is connected by a path. If not, the graph has multiple **components**.

- **Digraph:** A graph with **Directed Edges** where order matters (e.g., $e = (v, w)$). Uses **In-degree** and **Out-degree**.

8.2 Graph Representation

- **Dense Graph:** $|E| = \Theta(|V|^2)$.
- **Sparse Graph:** $|E| = O(|V|)$.

8.2.1 Adjacency List (Use if sparse)

- Nodes stored in an array. Edges stored in a linked list per node.
- **Space:** $O(|V| + |E|)$.
- **Tradeoff:** Slow to check if v and w are neighbors ($O(|V|)$ in worst case), but fast to enumerate all neighbors ($O(\text{degree}(v))$).

8.2.2 Adjacency Matrix (Use if dense)

- A $|V| \times |V|$ matrix A . A^n gives the number of paths of length n .
- **Space:** $O(|V|^2)$.
- **Tradeoff:** Fast to check if v and w are neighbors ($O(1)$), but slow to enumerate all neighbors ($O(|V|)$).

8.3 Graph Search

Both BFS and DFS run in $O(|V| + |E|)$ time.

8.3.1 Breadth-First Search (BFS) (Level Order)

- Uses a **Queue** (FIFO).
- Finds the **minimum hops** (shortest path in an unweighted graph).
- The parent pointer stores the shortest path back to the starting node, creating a shortest path tree.

8.3.2 Depth-First Search (DFS)

- Uses a **Stack** (LIFO) or recursion.
- Follows a path as far as possible before backtracking.

9 Shortest Path and Topological Sort

9.1 Single-Source Shortest Path (SSSP)

The Relaxation step: $\text{Dist}[v] = \min(\text{Dist}[v], \text{Dist}[u] + \text{weight}(u, v))$.

9.1.1 Dijkstra's Algorithm

- Works only with **non-negative weights**.
- Uses a **Priority Queue (PQ)** to repeatedly extract the vertex with the shortest estimated distance.
- **Time Complexity:** $O(|E| \log |V|)$ (using a balanced BST-based PQ).
- **Invariant:** Every dequeued vertex has a correct shortest path estimate.

Initialize all distances to infinity, start to 0.

Put all vertices into PQ.

While PQ is not empty:

 Extract vertex v (shortest estimated distance).

 Relax all neighbors w of v still in PQ:

 If $(\text{distTo}[w] > \text{distTo}[v] + \text{weight})$:

$\text{distTo}[w] = \text{distTo}[v] + \text{weight}$

$\text{pq.decreaseKey}(w, \text{distTo}[w])$

9.1.2 Bellman-Ford Algorithm

- Works with **negative weights** (but not negative cycles).
- **Time Complexity:** $O(|V||E|)$.
- **Detecting Negative Cycle:** Run the relaxation step for $|V| + 1$ iterations. If any distance estimate changes in the last iteration, a negative weight cycle exists.
- If all edges have the same weight, use BFS ($O(|V| + |E|)$).

9.2 Topological Sort

Applies only to a **Directed Acyclic Graph (DAG)**. It yields a total ordering of vertices such that for every directed edge $u \rightarrow v$, u comes before v in the ordering.

- **DFS-based (Post-Order):** Perform DFS, and prepend each vertex to the sorted list when it is "processed" (after recursively visiting all neighbors).
- **Alternative (Kahn's Algorithm):**
 1. Compute the in-degree for all nodes.

2. Add all nodes with in-degree 0 to a queue.
 3. While the queue is not empty, dequeue a node v , add it to the sorted list, and reduce the in-degree of all its neighbors. If a neighbor's in-degree drops to 0, enqueue it.
- **Time Complexity:** $O(|V| + |E|)$.

10 Minimum Spanning Trees (MST)

10.1 Spanning Tree Properties

- A **Spanning Tree** is an acyclic subset of edges that connects all nodes.
- A **Minimum Spanning Tree (MST)** has the minimum total edge weight.
- **Cycle Property:** For every cycle, the maximum weight edge is **not** in the MST. The minimum weight edge may or may not be in the MST.
- **Cut Property:** The minimum weight edge across any cut (partition of vertices) is in the MST.
- An MST contains $|V| - 1$ edges.
- A connected graph is **Strongly Connected** if there is a path from v to w and w to v for every v and w .

10.2 MST Algorithms

10.2.1 Prim's Algorithm

- **Greedy Approach:** Grows the MST from a starting node.
- Uses a PQ to store edges, prioritizing the shortest edge connecting a node in the MST to a node outside the MST.
- **Time Complexity:** $O(|E| \log |V|)$.

10.2.2 Kruskal's Algorithm

- **Greedy Approach:** Sorts all edges by weight and adds the smallest edge if it does not form a cycle (checked using Union-Find).
- **Time Complexity:** $O(|E| \log |E|) = O(|E| \log |V|)$ (dominated by the sorting step).

10.2.3 Boruvka's Algorithm

- For every connected component, add the minimum outgoing edge.

- **Time Complexity:** $O(|E| \log |V|)$.

10.3 MST Variants

- **All edges have same weight:** Can use DFS or BFS.
- **Maximum Spanning Tree:** Multiply all edge weights by -1 and find the MST (which will correspond to the maximum spanning tree in the original graph).

11 Computational Geometry and Convex Hull

11.1 Computational Geometry

- Includes algorithms for dealing with geometric problems (e.g., finding the **Convex Hull**).