**OOP**

Classes:

- Blueprint that defines properties and behavior of an object

- Also specifies some common storages, **every instance has one own copy** and not every instance **shares** the same copy.

Each class has a **private & public portion**, of which **functions (methods)** and **variables (attributes / properties)** can be accessed. Each class can only be passed to certain functions for editing attributes, where the functions are pre-defined.

This allows for **encapsulation**, which is a representation of data that does not allow direct access, but rather only using exposed functions.

|  |  |
| --- | --- |
| **Public** | **Private** |
| Publicly accessible to all other classes and functions. | **No visibility** from outside. Cannot be modified using any other means, can only be modified using the functions defined. |
| Methods that can access all attributes. | Can **only be accessed using public methods within class**.  Other classes can access member/attribute **only if they are ‘friend’ classes**. (defined using **friend class List**) |
| These are the only parts of the class that we can access from the outside. | By default, all members of a class are **private** if not specified. |

**Instances** can be created from each class:

- Attributes / Properties belong to the instance and each instance has their own data

- Methods belong to the instance and operate only on its **own data**.

Constructors + Destructors🡪 Used Automatically or deliberately with **delete** or during initialization.

When using classes, we need to have functions that are functions to initialize the object while destructor ‘frees’ it.

Inheritance  
Letting different implementations use the same code and inherit the properties. This uses one **parent class** and other child classes with class child : public parent {} copying the public code of parent. The private members cannot be accessed directly, but the public methods can.

Polymorphism

This is the use of virtual functions (enables overloading) to ensure that the same class, same functions can be used to create different results. Not the same as template T.

**Time Complexities**

Omega 🡪 best case (lower bound)

- T(n) = Omega(f(n)) if there exists a constant c > 0 such that T(n) >= cf(n).

BigO 🡪 Worst case (upper bound)

- Same, but T(n) <= cf(n)

If BigO = Omega, *T*(*n*) = Θ(*f*(*n*))

***f* (*n*) = O(*g*(*n*)) if and only if *g* (*n*) = Ω(*f* (*n*)) is true.**

**Peak Finding:**

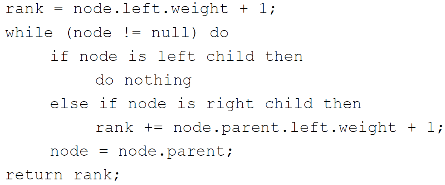
Recurse to the **larger side** for 1D array. O(log n)

In 2D array, **use middle column and find max element within it**.

Let index of mid column be ‘mid’, and max element at m[max][mid].

If max < m[index][mid – 1], recur left half.

If max < m[index[arr + 1], recur right half.

**O(row \* log(columns))** time.

**Quiz 1 Notes**

C++ class can have more than 1 constructor. Just overload w virtual or with diff arguments.

Compiler assigns one constructor automatically.

No need to give implementation details for ADT.

Inheritance is not compatible with templates in C++.

Stable sort refers to maintaining the relative order of the items with equal sort keys. i.e. if keys are repeated, their indexes are maintained. Like in excel when u sort by age, then name u want to maintain the age sort.

**Sorting**

For small arrays (n < 1000), use **Insertion Sort**. For large, use **Quick sort then insertion sort (Hybrid Quick Sort)**. This reduces recursion overhead. In place means no new list created.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sort | Space | Best | Average | Worst | Invariant |
| Selection | In place |  |  |  | Some sorted, rest jumbled |
| Insertion | In place, Stable |  |  |  | Some sorted, rest untouched |
| Bubble | In place, Stable |  |  |  | Largest at end |
| Merge | Stable |  |  |  | Sorted but not merged |
| Quick Sort | In place |  |  |  | List is split into bigger & smaller than pivot. |

**Trees**

In-order: Left, print, right. (Touch twice) **🡪 O(n)**

Pre-order: Print, left, right. (Touch once) **🡪 O(n)**

Post-Order: Left, Right, Print (Touch thrice / Leaving) **🡪 O(n)**

Functions:

**Successor** 🡪 Search for **next largest node**.

To find the lowest node that goes left in the search.

1. Search for key

2. If node has right, return max of right subtree

3. Follow search path from the root, find the last parent that goes left to find x.

**Successors never have 2 children because successors are minimum of the right subtree.**

**Delete** 🡪 Remove node from tree.

- No children 🡪 Remove node

- 1 child 🡪 Remove **and connect child to parent.**

**-** 2 child 🡪 Replace **node with successor**, then remove old node.

Time:

O(h) 🡪 log(n) – 1 <= h <= n

Insert: O(h) = O(log n); Delete: O(h) = O(log n) ; Search:O(h) = O(log n)

Predecessor, Successor O(h) = O(log n); findMax, findMin: O(h) = O(log n)

Height = max(height(left), height(right)) + 1; **height(null) = -1**

Update the heights as we insert / delete nodes recursively.

**BST**

**If tree is balanced, operations take O(log n) time.**

BST is height-balanced if each node is height-balanced.

Height-balanced tree contains at least n > 2­­h/2 node 🡪 O(log n).

**Height is to be checked after each operation on the tree.**

Balancing Through Rotation 🡪 Start from bottom most unbalanced node

**If left heavy:**

- if v.left is right-heavy: left-rotate(v.left), then, right-rotate(v)

**If right heavy:**

- If v.right is left-heavy: right rotate(v.left) then, left-rotate(v)

Right Rotate:

Left Rotate:

New root = node -> right

Node -> right = New root -> left

New root -> left = node

Update Height of node

Update Height of new root

Return new root.

New root = node -> left

Node -> left = New root -> right

New root -> right = node

Update Height of node

Update Height of new root

Return new root.

Insertion 🡪 Needs 0 to 2 rotations to balance.

Deletion 🡪 Needs up to O (log n) rotations to balance.

- Mark the deleted node as ‘deleted’ and leave in tree.

**Augmentation 🡪 Order Statistics**

Let the weight of a node **be the size of the tree rooted at that node**.

w(v) = w(v.left) + w(v.right) + 1

The rank of a node = left.weight + 1

Select(k): Rank(k):

Text, letter

Description automatically generated

Basic Methodology of Augmentation:

- Choose data structure and determine more info needed

- Modify the data structure to maintain this additional info 🡪 Change functions.

- Develop new operations

Remember the update the weight of nodes during operations.

**Augmentation 🡪 Orthogonal Range Searching**

Range List Query:

- Given two numbers a < b, list out elements x such that a <= x <= b

- O(log n + k), where k is the number of output elements.

Range Count:

- Same condition, count all the elements x using rank(b) – rank(a) + 1

- If a or b are not in BST, use their successor or predecessor respectively

For 2D / multi-Dimensional searches:

- Build a BST according to x-coordinates, where each node divides the points into subgroups

- Add a y-tree that is the BST of a subgroup by y-coordinates.

**Query Cost**: O(logdn + k); **buildTree Cost**: O(n logd-1 n); **Space**: O(n logd-1 n)

We can store each (d-1) range tree in each node of a 1D range-tree. Then we can build the (d-1)-dimensional tree recursively.

**Hash Tables**

No duplicate or mutable keys are allowed (keys cannot be modified)

We cannot choose a hash function with no collisions based on **Pigeonhole Principle**, and some keys must collide.

**Simple Uniform Hashing Assumption** 🡪 Every key equally likely to map to every bucket, and keys are mapped independently.

Chaining

Each bucket contains a **linked list of items**. Space = O (m + n)

M = table size, n = list size, **worst case is O(n)** if all items in same bucket.

**Expected search time = 1 + n / m. If m > n, O (1). Else O(n). Insertion = O(1).**

Chaining may fail because keys are not random and have **patterns**.

**Division** 🡪 h(k) = k mod m; Collision when k1 mod m = k2 mod m. **m is a prime number.** Also Division is slow.

If all key values are divisible by common divisor d, then only 1/d space used.

**Multiplication 🡪** h(k) = (Ak) mod 2w >> (w – r) (Takes r bits of second half)

Table size m = 2r, word size in bits = w, A = odd constant.

Faster than Division and works decently.

Open Addressing

No linked lists, all data stored in table, and uses probing.

If there’s collision, find new bucket by going traversing down the table via

(h(F) + f(i)) mod m

**Linear** 🡪 f(i) = i; **Quadratic** 🡪 f(i) = i2; **Double Hash** 🡪 f(i) = i x g(key)

**Double hashing > Quadratic > Linear** because of **minimum collisions!**

When items are deleted, set to **DELETED** instead of NULL, else we cannot search without getting truncated. DELETED have another item put in but does not terminate search.

When table is full, chaining can add items, open addressing cannot.

Performance of open addressing = 1 / (1 – a) where a = n / m

**Advantages** 🡪 Saves space (less empty); Rarely allocate memory; Better cache performance as table is all in one place in memory compared to linked list.

**Disadvantages** 🡪 Open addressing fails when load > 80% and is more sensitive to choose of hash functions. Sensitive to clustering as well.

**Hash table size 🡪 n == m1 then m2 = 2m1; n < m1 / 4, then m2 = m1 / 2**

**Good Hash Function** 🡪 Can reach all slots; Simple Uniform; Preimage Resistance; Collision Resistance; Second-preimage resistance.

**Heap**

Max / Min Priority Queues and stores items in trees. Biggest item at root and smallest at leaves (for max heap). Height = O(log n).

Every level is full, except last, with all nodes as **far left as possible.**

|  |  |
| --- | --- |
| Insert:  1. Add a new leaf to any slot in last level on most left. Most left first!  2. Bubble up until at correct place | BubbleUp:  While v != null, swap if priority is more, else return. |
| IncreaseKey:  Change value then bubbleup | DecreaseKey:  Decrease then bubble down. |
| BubbleDown(n):  - If children > n, swap with **bigger child** until in correct place | Delete:  1. Swap(v, last)  2. Remove the last (which is now v) and size-- 3. BubbleDown(last) |
| Extract Max:  1. Node v = root  2. Delete(root) | Heap vs AVL:  Same cost of operations, simpler and better concurrency. |

We can store trees in arrays, where each level i starts from index 2i – 1.

Left(x) = 2x + 1; Right(x) = left + 1 = 2x + 2; parent(x) = floor ((x – 1) / 2)

Because this technically a tree, we can rotate it.

**Heap Array 🡪 Sorted List**

For (I = n – 1 to >= 0); int val = extractMax(A); A[i] = value **Time**: O(n log n)

This sort is **in-place**, **unstable** and fast. It’s faster than Merge Sort but slower than Quick Sort. **It is also deterministic (quick is probabilistic)**

**Ternary (3-way)** Heapsort is a little faster.

**Unsorted List 🡪 Heap Array**

For (I = n -1 to I >= 0); bubbleDown(I, A) **Time:** O(n)

**Union-Find**

How Quick – Union works is by linking **the parents of nodes together in one tree.**

Weighted union would also merge trees, but it would **first check the size of the trees**, then merge.

|  |  |  |
| --- | --- | --- |
| **Operation (p,q)** | **O** | **Description** |
| QF Find | 1 | Check if p & q have the same component |
| QF Union | n | Check all n for q and switch their components to p |
| QU Find | n | Go up the tree for both p and q, see if grandparents same |
| QU U | n | Set grandparent of p to grandparent of q. (need to find both first) |
| Weighted U | logn | QU U except the bigger one is the grandparent |
| Path Compress | logn | Make every node in the path point to grandparent |
| WU + PC | 1\*\* | Inverse Ackermann Function speed (ALMOST but not linear time) |

**Graphs**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Diameter | Max Dist of graph, following shortest path |
| Connected | All vertexes connected in a single component |
| Degree | Maximum no. of adjacent edges |

|  |  |  |
| --- | --- | --- |
| **Operation** | **Adjacency List (List of LL)** | **Adjacency Matrix (Social)** |
| P & Q neighbours? | Slow | Fast |
| Any / All neighbours of P | Fast | Slow |
| Space Complexity | O(V+E) | O(V2) |
| Cycle | O(V) | O(V2) |
| Clique (Connected Graph) | O(V+E) = O(V2) | O(V2) |
| Graph dense? Dense = θ (V2) | Use when not dense | Use when dense |

**SSSP**

Undirected / Constant Weight graphs – use BFS & DFS (Both Time O(E+V), space O(V) )

BFS – Adds unvisited neighbours to the queue, then call itself

DFS – Adds unvisited neighbours to the stack, then call itself

Bellman-Ford O(EV) – Works on negative weights, but not negative weight cycles. Start with all distances at infinity, then visit every vertex and relax every edge. Terminate early if a round passes without any relaxation. If extra round (V+1) relaxes, means there is a negative cycle.

Dijkstra O(E logV) – Doesn’t work with negative weights! Make a Min Heap of all unvisited nodes & their distances at infinity. While relaxing edges, update the distances in the heap. Move to the minimum unvisited node in the Min Heap and repeat.

Topological Sort O(E + V) – Not unique. Basically DFS, but once there are no unvisited neighbours, the value is pushed into a stack. If a branch terminates, arbitrarily select the next unvisited node. Once all nodes visited, just keep popping the stack.

**MST**

An acyclic subgraph that contains all nodes but with minimum total edge weight. Tree have |E| = |V| - 1.

MST Properties:

1. No cycles
2. If an MST is cut, it forms 2 MSTs
3. For every cycle in the graph, the maximum weight is not in the MST
4. For every separate component in the graph, the minimum weight between them is in the MST

Prim’s Algorithm – Like an empire conquering cities. Searches all the edges connecting to the current tree (starts with just 1), then adds the shortest edge to an unvisited node.

Kruskal’s Algorithm – Like a bridge operator. Sort all the edges in a min list. If the minimum edge does NOT create a cycle (i.e. 2 vertexes in same tree), then save the edge and its vertices.

Boruvka’s Algorithm – Like an air strike. Go to every node and save the shortest, “obvious” edges. There will be different trees because some edges are saved twice. Then run Prim’s algorithm on each separate tree (selecting the shortest outgoing edge of the entire tree) until everything is connected.

**Computational Geometry – Convex Hulls**

Convex Hulls are just the enclosed polygon around a set of points such that there are no concavities in the perimeter.

Jarvis’ March O(fn) (f – number of faces in the final polygon) – Gift Wrapping. Start on the leftmost node, then from the downward angle turn anticlockwise until you see your first node, and move there, etc.

Graham Scan O(n log n) 2D only – Air Strike then March. On the leftmost node, list all the nodes by their angle then construct a polygon from there. After that, “march” around the perimeter, then whenever there is a concavity, walk backwards and “fill the hole” by connecting directly to the one after.

Divide and Conquer O(n log n) in 3D – Form a couple of convex hulls by sorting according to direction. Then between two convex hulls, merge by “walking” to the edges and merge.

Incremental method O(n log n) in 3D – Like Prim’s Algorithm. For each adjacent node to the massive starting blob, update the convex edges to it.