CS2040C

Sorting

$O(n^2)$ sorting algorithms

- Bubble sort (stable): well we know how it works already
    - Can be used if the list is almost sorted, since can be terminated early at $O(n)$
- Selection sort (not stable): swap with the smallest element after it
- Insertion sort (stable): shift left until good
    - Can be used when n is small, as it is stable and has a small constant despite being $O(n^2)$

$O(n \log n)$ sorting algorithms

- Merge sort (stable): D&C, iterative merge step
    - Can be used when n is large and the sort must be stable
    - Slower than random quick sort on average
- Quick sort (not stable): select pivot, shift stuff so that left < pivot and right > pivot
    - Pre-determined pivot: $O(n^2)$ worst case — not good
    - Randomised pivot (random quick sort): $O(n \log n)$ average case — good
    - Randomised version can be used when sorting doesn't need to be stable
    - Randomised version much faster than merge sort on average

$O(n)$ sorting algorithms

- Very specific constraints
- Counting sort: we know it already
    - Must be integers of small range (<1m)
    - Actual complexity: $O(n+k)$ where k is the range
- Radix sort: sort as strings (pad zeros if needed), sort by pushing to queue on every digit from the least significant then concatenate
    - Actual complexity: $O(w(n+k))$ where w is the length of strings and k is the base (base-10 for decimal numbers, base-26 for lowercase English alphabet, etc.)
    - Can be used if w is significantly smaller than $\log n$.

Comparisons in specific use cases

	Random	Sorted (non-↓)	Sorted (non-↑)	Nearly (non-↓)	Nearly (non-↑)	Duplicates
Bubble	$O(n^2)$	$O(n)$	$O(n^2)$	←	←	←
Selection	$O(n^2)$	←	←	←	←	←
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	←	←	←	←	←
Quick	$O(n \log n)$	$O(n^2)$	←	←	←	←
Rand quick	$O(n \log n)$	←	←	←	←	←
Counting	$O(n)$*	←	←	←	←	←
Radix	$O(n)$*	←	←	←	←	←

*: see specific notes for counting and radix sort algorithms above.

Linked List

	SLL	DLL	Stack	Queue	Deque
search	$O(n)$	$O(n)$	🚫	🚫	🚫
peek front	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
peek back	$O(1)$	$O(1)$	🚫	$O(1)$	$O(1)$
insert front	$O(1)$	$O(1)$	$O(1)$	🚫	$O(1)$
insert back	$O(1)$	$O(1)$	🚫	$O(1)$	$O(1)$
insert middle	$O(n)$	$O(n)$	🚫	🚫	🚫
remove front	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
remove back	$O(n)$	$O(1)$	🚫	🚫	$O(1)$
remove middle	$O(n)$	$O(n)$	🚫	🚫	🚫

Binary Max Heap

For **priority queues** with $O(\log n)$ enqueue and dequeue operations.

Definition

- Complete binary tree
- Max → min from top down

Main content

- PQ can't be implemented as LL/array as it is slow ($O(n)$ for either enqueue or dequeue)
- Store binary tree as compact array
  - Parent: $x/2$
  - Left child: $2x$
  - Right child: $2x + 1$
- Operations:
  - ShiftUp(v): swap v with parent p of v if need to, then ShiftUp(p) until finish or root
    - $O(\log n)$ since maximum height is $\log n$
  - ShiftDown(v): swap v with largest child of v if need to, until finish or no more child
    - $O(\log n)$ for same reason

Insert

Insert at index $n + 1$ (last position in the tree), then ShiftUp(new vertex). $O(\log n)$

Extract max

Retrieve and remove root, put last position to root, then ShiftDown(root). $O(\log n)$

Create from array

- $O(n \log n)$ version: keep inserting for each value
- $O(n)$ version: Robert W. Floyd (1964)
  - Compact array = complete binary tree where half of all vertices are binary max heap by default

- For all non-leaf vertices from lowest position to root, ShiftDown()

Update/delete key given index

Update `A[i] = newValue` then call both shifting operations on it (only one of them will actually be triggered). Similar for delete key. $O(\log n)$.

Update/delete key *without* index

Still can do in $O(\log n)$ with a hash table to look up key in $O(1)$. A bit more complicated though.

Sorting algorithms

Call extract max n times → $O(n \log n)$ sorting algorithm

Call extract max k times → $O(k \log n)$ *partial* sorting algorithm to take only the top k largest elements

---

Hash Tables

Used for `std::unordered_map` and `std::unordered_set` to look up a key in $O(1)$.

Key operations: search(v), insert(v), remove(v) — all have to be $O(1)$.

Terminologies

- Symbols: n number of keys, m hash table size
  - m is usually a large prime not near a power of 2 to make distribution "more" uniform
- **Load factor** $\alpha = n/m$, α should be smaller than 0.5
- Expected number of probes = expected cost of an operation (in **open addressing**):

$$C_{\text{expected}} \leq \frac{1}{1-\alpha}$$

Open addressing

Linear probing

- $h(v) = v \bmod m$ (`v % m`)
- If already occupied, scan forwards one index at a time for an empty slot: $h(v)+1$, $h(v)+2$, etc.
- Remove: set to `DELETED` instead of `EMPTY` so that prob sequence is intact (search() won't break)
- Can easily cause large primary clusters (sequence of successive occupied slots) → slow

Quadratic probing

- $h(v) = v \bmod m$ again
- $h(v)+1^2$, $h(v)+2^2$, $h(v)+3^2$, etc.
- **NOT** $h(v)+1$, $h(v)+1+4$, $h(v)+1+4+9$, etc.
- Have to ensure $\alpha < 0.5$ to avoid infinite loop

Double hashing

- $h(v) = v \bmod m$
- Second hash function $h_2(v)$, can be $v \bmod m_2$ (VisuAlgo) or anything…
  - Should satisfy $h_2(v) > 1$
- Prob: $h(v)+h_2(v)$, $h(v)+2h_2(v)$, $h(v)+3h_2(v)$, etc.

Separate chaining

Each slot in the hash table is a DLL. Simply insert to the back of the DLL when we insert new key.

Search and remove are $O(1 + \alpha)$.

Likely used in `std::unordered_map` and `std::unordered_set`.

When α gets too fat

Can rehash: build a new hash table of size $2m$, then recompute the new hash values completely for each value in the old hash table.

Binary Search Tree

Used for `std::set`, `std::map`, can be used for priority queues as well.

Search, insert and remove operations in $O(\log n)$. Compared to hash table (unordered keys), in BST keys must be ordered (opening the way for more operations: max(), min(), predecessor(), successor(), etc.)

Definitions

- Binary search tree: A binary tree sorted from leftmost vertex to the rightmost one
- AVL tree: self-balancing BST where height is always $O(\log n)$
- Rank of a key v is where it stays (1-based index???) if all keys are sorted in ascending order

Operations

Search(v)

- Search from root, compare with v and continue searching in left/right children accordingly
- min() and max() can be found by keep searching left/right children
- $O(h)$ (which is $O(n)$ in bare BST and $O(\log n)$ in AVL)

Successor(v)

- Search for v (of course)
- If have right subtree, find the min in the right subtree
- If not have right subtree, traverse the ancestors until we find a right turn, from there we get the max in that subtree
- Similar for predecessor(v)
- Still $O(h)$

Traversal

- In-order: visit(left), print(root), visit(right)
- Pre-order: print(root), visit(left), visit(right)
- Post-order: visit(left), visit(right), print(root)
- All three are $O(n)$

Insert(v)

- Search for v. Should be not found (since all values in BST are unique — if not unique needs to add secondary data to ensure uniqueness)
- Insert v at that position
- $O(h)$

Remove(v)

- Search for v, already $O(h)$
- If leaf: remove the leaf $O(1)$
- If not leaf and has only one child: connect the child with the parent $O(1)$
- Otherwise: replace v with its successor, then remove the successor in the subtree $O(h)$

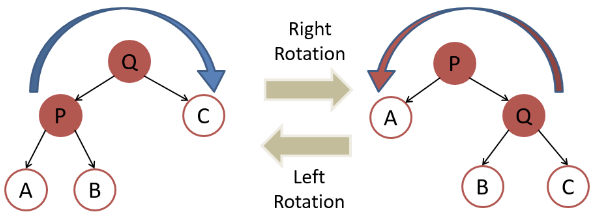
Create

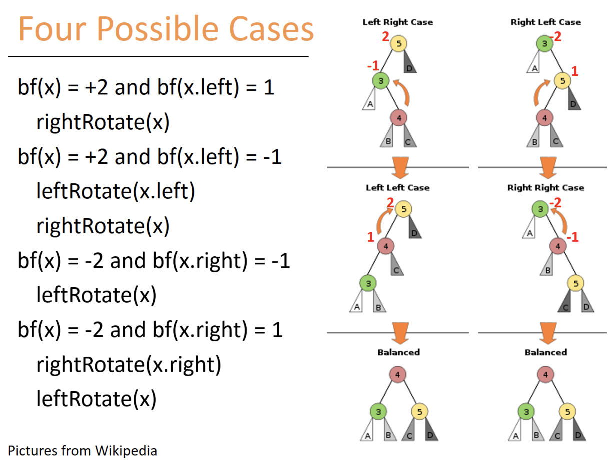
- Insert for each element

AVL

- Adelson-Velskii & Landis 🇷🇺 1962 in case prof somehow asks history questions
- Define `v.height`: number of edges from v to deepest leaf
  - `root.height` is h and `leaf.height` is 0
  - `v.height = max(v.left.height, v.right.height) + 1`
  - Compute on insertion/deletion to cache value
- $h < 2\log n$, so all $O(h)$ operations are $O(\log n)$
- For all vertices v, balance factor is `v.left.height - v.right.height`
  - Height balanced: must be between -1 and $+1$ inclusive.

Rotations





Insert(v)

- Insert as a normal BST
- From insertion point to the root, update `height` and balance factor `bf`
  - Do rotation if necessary

Remove(v)

- Remove as a normal BST

- From deletion point to the root, update `height` and `bf`, do rotation if necessary

Union-Find Disjoint Sets

Find set, check same set and union two sets must be constant time (not exactly $O(1)$ but it is $O(\alpha(1)) \approx O(1)$, with $\alpha(x)$ being the inverse Ackermann function.

Data storage

`p[i]` is the parent of `i` and `p[i] = i` if `i` is the root of the tree. By default, `p[i] = i` since all values are disjointed.

`rank[i]` is the upper bound of the height of subtree rooted at vertex `i`. It's *not necessarily the true height* of the subtree and only used as a guiding heuristic. By default `rank[i] = 0`.

Operations

Initialise

Set `p[i] = i` and `rank[i] = 0`. Obviously $O(n)$.

Find set (i)

Recursively get `p[i]` until `p[i] = i` (root). Then assign `p[i]` to this root (compress the tree). $O(h)$, but thanks to tree compression, it's expectedly $O(1)$.

Check same set

Find the root of both and check for equality. $O(1)$

Union set

Find roots r_i and r_j of i and j. If rank of r_i is smaller than rank of r_j, mark parent of r_i to be r_j, and vice versa. If the two ranks are equal, can mark parent of either to be the other, then increase the rank of the new root by one.

Graphs

- Graph, (directed/undirected) edge, edge weight, vertex, path, connected graph, cycle, DAG, trees, complete graph, bipartite, AM, AL, EL, DFS, BFS: ok
- **Simple graph:** No self-loop edges ($u \to u$), no parallel edges (two $u \to v$).
- **Simple path:** No repeated vertices along the path
- **Adjacency:** two edges sharing a vertex or two vertices sharing one edge
  - Directed: v is adjacent to $u \Leftrightarrow u \to v$ exists
- **Degree of vertex**
  - Undirected: Number of edges from/to that vertex
  - Directed: Differentiate with *in-degree* and *out-degree*
- **Strongly Connected Component:** (only directed) Each vertex can visit all other vertices in the subgraph
- **Articulation points** also **cut vertices**: vertices if removed will disconnect the graph
- **Bridges:** edges if removed will disconnect the graph
- **Topological sort:** Sort vertices in order in directed graph

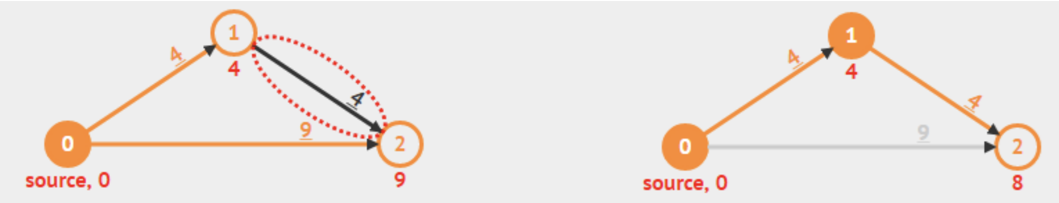
Single-Source Shortest Paths

Given a graph and a source vertex s, find d_i being the minimum distance from s to i for all vertices i in the graph.

Output is ill-defined when there is a negative weight cycle.

Algorithms

Relax operation



What happens when `relax(1, 2)` is called: since $d_1 + w < d_2$, d_2 is updated to be $d_1 + w$ (the SSSP to vertex 2 now includes 1)

Bellman-Ford

- $O(V \times E)$ — slow, although it can solve all kinds of valid SSSP variants
- After each iteration of i, all direct neighbours of s has the correct d values. Hence after $V - 1$ iterations, all $(V - 1)$-th neighbours of s aka the entire graph has the correct d values.
- If no relaxation occurs, outermost loop can be terminated → can be $O(k \times E) < O(V \times E)$

BFS for constant-weight graphs

Okay… I guess. $O(V + E)$

Doesn't work for non-constant-weight graphs.

DFS for trees

$O(V + E) = O(V)$, doesn't work for non-trees.

DP for DAG

- Get the topological sort
- Relax edges according to the topological order
  - Similar to Bellman-Ford, but we only do the inner loop once with a very specifically ordered edge list
- $O(V + E)$

Dijkstra

- **Does not work for negative edges.**
- Maintain a set S of solved vertices, initially $S = \{s\}$.
- While S is not the entire graph:
  - u is the vertex with the minimum shortest path estimate
  - Relax all outgoing edges of u
  - Add u to S
- Time complexity:
  - $O(V \log V)$ when handling vertices inside the PQ
  - $O(E \log V)$ when relaxing the neighbours and updating values inside the PQ
  - $O((V + E) \log V)$ in total

Modified Dijkstra

We do not update the d value inside the PQ when relaxing, but add new values instead. **Works for negative edges, but may not terminate.**

No negative edges: $O((V + E) \log V)$, but exponential time for negative edges.

Code snippets

```
void bellman_ford(int s) {
    dist[s] = 0;
    for (int i = 0; i < n - 1; i++) {
        bool updated = false;
        for (int u = 0; u < n; u++)
            for (auto& [v, w] : adj[u])
                if (__sssp_relax(u, v, w, dist, parent)) updated = true;
        if (!updated) break;
    }
    for (int u = 0; u < n; u++)
        for (auto& [v, w] : adj[u])
            if (dist[v] > dist[u] + w) {
                cerr << "Negative cycle detected\n";
                return;
            }
}
void original_dijkstra(int s) {
    dist[s] = 0;
    set<pair<T, int>> pq;
    for (int i = 0; i < n; i++) pq.insert({dist[i], i});
    while (!pq.empty()) {
        auto [d, u] = *pq.begin();
        pq.erase(pq.begin());
        for (auto& [v, w] : adj[u]) {
            if (dist[u] + w >= dist[v]) continue;
            pq.erase(pq.find({dist[v], v}));
            dist[v] = dist[u] + w;
            parent[v] = u;
            pq.insert({dist[v], v});
        }
    }
}
void dijkstra(int s) {
    dist[s] = 0;
    priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int>>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (d > dist[u]) continue;
        for (auto& [v, w] : adj[u]) {
            if (dist[u] + w >= dist[v]) continue;
            dist[v] = dist[u] + w;
            parent[v] = u;
            pq.push({dist[v], v});
        }
    }
}
```

Comparisons

	Bellman-Ford	Dijkstra	Mod. Dijkstra
Neg cycle	WA	WA	Not terminated
Neg edge	AC	WA/AC	AC (long)
Dijkstra killer	AC	WA	AC (long)
BFS killer	AC	AC	AC

Minimum Spanning Tree

Applicable to a connected undirected weighted graph. Find the spanning tree with the lowest total weight.

Properties of MST

If an algorithm satisfies both of these, it is a valid MST algorithm.

- For any cycles in the graph, the maximum edge is never in the MST
- Divide the graph to two subgraphs, the minimum edge between the two is always in the MST

Kruskal

- $O(E \log V)$ greedy algorithm
- Sort all edges by lowest weight first
- Loop through the sorted edges, select an edge to the current spanning tree (initially empty) if it doesn't create a loop
    - Using UFDS
- If the spanning tree reaches $V - 1$ edges, we can terminate.

Prim (Jarnik-Prim)

- Also $O(E \log V)$ greedy algorithm
- Starts from source s (can be any source)
    - If there are equal edge weights, the choice of s may change the MST (although total weight is still unique)
- Enqueue all edges incident to s to a PQ sorted by lowest weight first
- While PQ is non-empty and tree doesn't have $V - 1$ edges yet:
    - If the dequeued edge doesn't form a cycle, add it to the tree and enqueue edges connected to it
    - Otherwise discard the dequeued edge

Code snippets

```
T kruskal() {
    UnionFind uf(n);
    sort(edges.begin(), edges.end());
    T mst_cost = 0;
    int num_edges = 0;
    for (auto& [w, u, v] : edges) {
        if (uf.is_same_set(u, v)) continue;
        uf.union_set(u, v);
        mst_cost += w;
        if (++num_edges == n - 1) break;
    }
    return mst_cost;
}
void __prim_process(int u, vector<bool>& in_mst, priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int>>>& pq) {
    in_mst[u] = true;
    for (auto& [v, w] : adj[u])
        if (!in_mst[v]) pq.push({w, v});
}
T prim() {
    vector<bool> in_mst(n, false);
    priority_queue<pair<T, int>, vector<pair<T, int>>, greater<pair<T, int>>> pq;
    __prim_process(0, in_mst, pq);
    T mst_cost = 0;
    int num_edges = 0;
    while (!pq.empty()) {
        auto [w, u] = pq.top();
        pq.pop();
        if (in_mst[u]) continue;
        mst_cost += w;
        __prim_process(u, in_mst, pq);
        if (++num_edges == n - 1) break;
    }
    return mst_cost;
}
```