**CS2040 Summary (18.19 Sem 2 Syllabus)**

**AP & GP Formula**

**AP**

Tn = a + (n-1)d | Sn = |

Tn = arn-1| Sn = | Sn = |

**Sorting**

**Sorting Algorithms (Summary)**

1. **Selection Sort**

* Select the largest element in the curr subarray (by finding it) and swap it with the last item in the currarray

1. **Bubble Sort**

* “Bubble” the largest element to the end of the curr subarray by keep swapping it with its next element

1. **Insertion Sort**

* Arrange poker cards: Take the first element you have now and insert it into the right position in the curr subarray in front of it.

1. **Merge Sort**

* Keep dividing the array into part by part until you can’t divide it anymore and gradually merge the parts together
* Divide the array into 2 parts, call merge sort on the left part and right part, and then merge them together

1. **Quick Sort**

* Choose a pivot, partition the array by putting the elements smaller than the pivot to its left and elements greater than the pivot to its right, call quicksort again on the 2 subparts recursively.

1. **Radix Sort**

* For number-wise radix sort, you create 10 groups (digit 0~9), put the numbers into one of the groups according to their digit at the curr k position, repeat this until you hit the first digit

**Terminologies**

1. **In-place sorting**

* Doesn’t require extra space(or aka memory) when sorting (however a constant amount of spaces for **variables** are allowed.

1. **Stable sorting**

* Doesn’t change the relative order of the elements (based on the previous key used to sort the elements) in the curr sorting process

**A screenshot of a cell phone

Description automatically generated**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Hashing**

1. Hash table is a data structure (a table) which utilizes hashing function to efficiently map a key to where the value is located for efficient look-up and retrievals.
2. **No duplicate keys** are allowed! **KEY OVERWRITE (SAME KEY VALUES) WILL OVERWRITE THE CURRENT VALUE AT THAT KEY POSITION**

* The reason is quite simple: what should hashmap.get(key) return if there are multiple duplicate keys and each key maps to different value?

1. **Hash Functions**

* **Uniform Hash Function**
  + Used when your keys (integers) are uniformly distributed among a range
  + If k integers are **uniformly** distributed among 0 and X – 1, we can then map the values to a hash table of size m (m < X) using the hash function below
  + Hash(k) = , k [ 0, X )
* **Division Method**
  + Hash(k) = k % m
  + Map an integer into a table of size m
  + It’s like the direct-mapped cache technique in cs2100
  + How to pick **m**?
    - Pick a m to be a **prime number** that is close to a power of two
    - Don’t use powers of two or powers of ten!
* **Multiplication method**
  + Hash(k) = , m = table size
  + A = a real number between 0 and 1
  + A good choice for A is the **golden ratio,** which is ((sqrt(5) – 1) / 2 = 0.618033
  + Basically what we are doing here is extract the fractional part of kA and multiply it with m
* **Hashing of Strings**
  + Hash(s)

sum = 0;

for each character c in s

sum = sum \* **31** + c

return sum % m

* + This is to make the position of the characters affect the hash value by “shifting” the sum every time (multiply the sum by a **prime number**)

1. **Collision Resolution Techniques (x4)**

|  |  |  |
| --- | --- | --- |
| **Technique** | **Description** | **Remark** |
| Separate Chaining | Basically it’s to store the keys that map to the same hash value(index of the table) in a list. | * Now insert, delete, and find are now **using the list** that corresponds to the index (hash value) in the table   linked lists in this case.   * + e.g. insert(key, data) -> insert the key and data pair into the list which reference is stored at a[hash(key)] * The load factor measures the **average length** of the linked lists. * **Average Running Time** * Find 1 + (successful), 1 + (unsuccessful search) * Insert 1 + * Delete 1 + (successful), 1 + (unsuccessful) * If load factor is bounded, then the complexity are O(1) * If load factor exceeds the bound, then need to reconstruct the hash table with a new size m and rehash all the keys |
| Linear Probing | When collision occurs, go to the next closest empty slot to put the key | * **Important:** * **Deletion**: cannot simply delete or else it will affect find() ! * Use **lazy deletion** instead: mark the slot being deleted. * Then when inserting, insert at the nearest deleted slot so that we can find that key faster * Problem: causes **primary clustering** (consecutive slots being occupied due to collisions) * **Original Prob sequence (causes primary clustering)**   Hash(key)  (Hash(key) + 1) % m  (Hash(key) + 2) % m … and so on   * **Modified Prob sequence (to introduce gaps)**   Hash(key)  (Hash(key) + 1 \* d) % m  (Hash(key) + 2 \* d) % m … and so on   * **Condition:** d is a constant integer and is **coprime to m**   (or else will end up in an infinite loop when searching for slots to insert!) |
| Quadratic Probing | Similar to linear probing, but uses a different prob sequence | * **Prob sequence**   Hash(key)  (Hash(key) + 1) % m  (Hash(key) + 4) % m  (Hash(key) + 9) % m … and so on   * **Condition:** If  **< 0.5** and m is **prime**, then we can always find an empty slot (if any), meaning that the table needs to be less than half full * If > 0.5, then the process to find the empty slot using quadratic probing may **never terminate** even if there are empty slots * **Problem**: Secondary Clustering (cluster along the way of the prob sequence) * Occurs if a lot of keys have the same initial positions |
| Double Hashing | Use a **secondary hash function, hash2** together with the primary one to hash when collision occurs | * **Probe sequence**   Hash(key)  (Hash(key) + **1** \* hash2(key)) % m  (Hash(key) + **2** \* hash2(key)) % m // Only use it if collision still occurs for the  // previous one.  (Hash(key) + **3** \* hash2(key)) % m … and so on   * Can help to overcome primary and secondary clustering as two keys may not have the same hash2 value, hence the probe sequence is different. E.g. although a and b have the same initial position, their hash2 may be different, so the prove sequence of both of them are different, so further collisions can be avoided, and hence primary or secondary clustering are avoided. * **Condition:** * Let hash1(k) = k mod a, hash2(k) = k mod b, * **b must be a prime number, < than a** and **coprime with a** (or else may end up with infinite loop!) * **hash2(k) % m** **must never** evaluate to 0 (or else may end up with infinite loop, which is hentak kaki!) (m is the table size) * A way to overcome the problem where hash2(k) = k % n evaluates to 0 is e.g. **hash2(k) = n – k % n.** Since k % n never evaluates to n, hash2(k) will never be 0. However, you still got to make sure that that hash2new(k) % m never equals to 0!!!! (% here is mod) * If hash2(k) % m can evaluate to zero, then the probe sequence may always end up at the initially collided position, hence infinite loop. |

* In other technique other than separate chaining, load factor measures **how full the table is**

1. **Criteria for good hash functions**

* **Minimize** clustering
* **Always** find an empty slot if it exists
* Give **different** probe sequence when 2 initial probes are the same (i.e. no secondary clustering)
* **Fast**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Binary Heap**

1. **Backbone Data Structure:** Array

* The first element of the array is set to be a **dummy** to simplify traversal operations, e.g. leftChildIndex(x) which returns the index of the left child of x (if any).

1. **Complete Binary Tree**

* A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far as left as possible.
* If every level is filled, then it is a **perfect/full binary tree**

1. Height of a binary tree with n items = **log n**

* Height = **max edges** from the root to the deepest leaf

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Operations** | **RunTime** | **Remark** |
| Binary Heap | Shift up | O(log n) | * Implement a **priority queue** * Basically a binary tree (not BST!) of height log n * Uses a compact array to store the items, with index 0 being null. * But **actually**, you can use a bBST to implement priority queue too, depending on the need. * **Navigation** * Parent(i) = floor(i / 2), except for i = 1 (root) * Left(i) = 2 \* I, no left child when: left(i) > heapsize * Right(i) = 2 \* i + 1, no right child when: right(i) > heapsize * **Some formulas (more exact)** * **Height** of heap of size N: * **Number of nodes** at height h of a **full** binary tree = * Cost to run shiftdown(i) = O(h) * Note that the indices of the level in heap is **reversed!** (i.e. from bottom to top, lowest to highest). Lowest level is h = 0. |
| Shift down | O(log n) |
| Insert | O(log n) |
| Delete | O(log n) |
| Extract Min / Max | O(log n) |
| Fast heap Create | O(n) |
| Slow heap Create | O(n log n) |
| Heap Sort | O(n log n) |

1. **Explanation on some of the operations**

* **Insert**: insert at the bottommost level of the heap, as left as possible, then call **shift** **up** on the inserted node
* **ExtractMax:** extract the root, replace the root with the last index node, and **shift** the last index node **down**
* **Shift up:** keep shifting the node upwards until it is smaller than its parent
* **Shift Down:** compare left, compare right, swap accordingly (if smaller than any of its children, max heap), and update the curr position of the shifted node (if shifted)
* **Create Heap Fast O(N):** fill up the heap array with all the element first, then starting from the second bottommost level of the heap, call shiftdown on every node (from right to left) in that level and repeat the same thing for the higher levels until you hit the root
  + The algorithm given in the lecture calls shiftdown from **right to left**, but actually it doesn’t matter even if we run from left to right, as long we run shiftdown bottom up then the heap created will still be valid. (refer cs2010 17.18 Sem 1 WQ1 Q4.)
* **Create Heap Slow O(NlogN)**: Insert the element one by one into the curr heap
* **Heap Sort:** Build a max heap out of the array we have (O(n), then call extraMax for n times (O(N log N)).

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Binary Search Tree (MAY NOT BE BALANCED)**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Complexity** | **Remark** |
| Insert | O(h) | Starting from the root, search until you find a suitable position to insert the node |
| Delete | O(h) | Find the successor of the node (if any) to replace the node and delete the successor from the tree |
| Search | O(h) |  |
| FindMax / FindMin | O(h) |  |
| Successor / Predecessor Searching | O(h) |  |
| In-order Traversal | O(N) |  |
| Selecting | O(h) |  |
| Ranking | O(h) |  |

**AVL Tree (A balanced BST)**

1. **h >= log2N** (For a general BST)

* For a height balance BST, **h < 2log2N**

1. **Height:** #edges on the path of this vertex to the deepest leaf.

* Height = -1 (empty tree)
* Height = max(x.left.height, x.right.height) + 1 (all other cases)

1. **Size:** #nodes in the curr subtree (including the node itself)

* Size = 0 (empty tree)
* Size = x.left.size + x.right.size + 1 (all other cases)

1. The invariant property in AVL tree: **Height-Balanced**

* A vertex x is height-balanced if **|x.left.height – x.right.height| <= 1**
* A BST is height-balanced if **every vertex** inside it is height-balanced.

1. Balance factor = **|x.left.height – x.right.height|**
2. **Smallest** AVL tree that deleting one of the two vertices will cause 2 separate groups of rebalancing operations have **12 vertices.**
3. To create an avl tree such that at every insertion there is **no** rotations, sort the list of vertices to be inserted first. Then, insert the median of this entire list into the avl tree, and recursively insert the median of the left sublist and insert the median of the right sublist **alternatively.**
4. The **hack** in creating an AVL tree where deleting one vertex can lead N deletions is to create the tree using the previous tree which cause N-1 deletion **as one of its subtree**

* E.g. in the example given in the slide, the left subtree causes 1 rotation, so to build an avl tree which cause 2 rotations, simply use this current tree as the left subtree of the new subtree and fill up the right subtree such that when you delete one vertex from the previous tree (now is the left subtree), it will cause the entire left subtree’s height to differ more than 1 than the right subtree, and now it will cause a **chain effect** in rotation.

**Modified Operations**

|  |  |
| --- | --- |
| **Operations** | **Remark** |
| Insert | * Starting from the root, search until you find a suitable position to insert the node. * After you have inserted the node, **update the heights** of the node along the way when you backtracked back to the root. * If at any point during backtracking the balance factor of a node is out (+2 or -2), **perform rotation** to fix the balance factor. * Insertion will trigger at most one of the 4 possible cases * **O(1) time (Only rotate one or two nodes max**) |
| Delete | * Starting from the deletion point (the deleted **successor**), backtrack all the way till root and update the height along the way. * Same thing, perform rotation if balance factor of a node is out. * Deletion can trigger more than one of the 4 possible cases (up to h = log n times) * **O(log(n))** time (**have to rotate for each level**) |

A close up of text on a white background

Description automatically generated

A map with text

Description automatically generatedA map with text

Description automatically generated

A close up of a map

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Graph**

1. A complete graph has edges.
2. Tree is the **smallest connected** graph.
3. **3 graph data structures**

* Adjacency matrix
* Adjacency list
* Edge List

1. The only **connected undirectexd** graph that **doesn’t have a cycl**e is a **tree** (which has V – 1 edges). If the connected undirected graph has >= V edges, then it has at least one simple cycle with 3 or more vertices.

**Graph Traversal**

|  |  |
| --- | --- |
| **Algorithm** | **Remark** |
| **BFS**  O(V + E),  O(V2 time with **ADJANCENCY MATRIX**) | * Starting from a vertex S, **visit all its adjacent neighbors** within **1-edge distance**, and repeat this again but starting from the neighbors.   A close up of text on a white background  Description automatically generated |
| **DFS**  Complexity: O(V + E) | * Starting from a vertex S, I go to one neighbor, and I then **go as deep as possible**. After I have done with that neighbor, I do this again for all the neighbors (1-edge distance) of S.   A close up of text on a white background  Description automatically generated |

**\***Note that to achieve O(V + E) for BFS and DFS, an adjacency **list** must be used instead of adjacency matrix!

**Path Reconstruction**

**Iterative Version (Reversed Path)**

**A close up of a logo

Description automatically generated**

**Recursive Version (Normal path)**

**A screenshot of text

Description automatically generated**

**Applications of Graph Traversal**

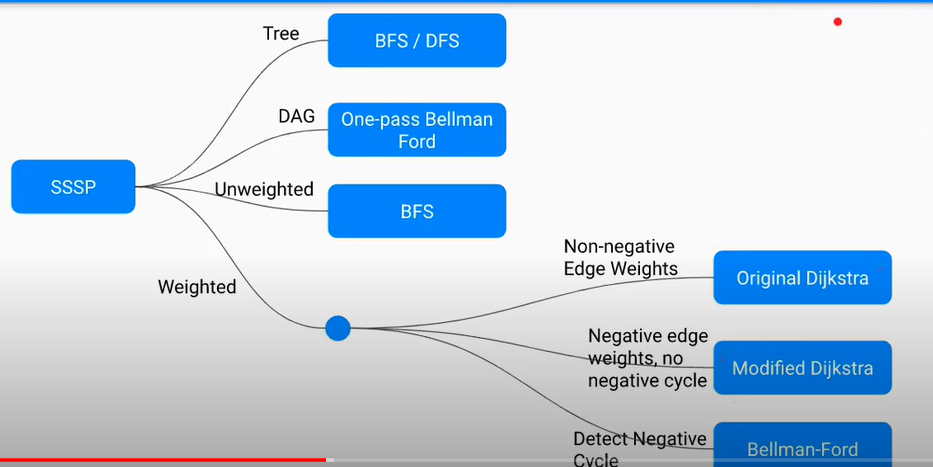
|  |  |
| --- | --- |
| **Application** | **Remarks** |
| Reachability Test | * To check whether v is reachable from s, simply perform BFS from s and check if visited[v] is 1 |
| Identifying Components | * Maintain a counter and run BFS/DFS on every vertex **only if they are not visited before by any of the vertex**. * If they are not visited before, counter++ * Complexity: O(V + E) |
| Topological Sort | * **Idea for Khan’s Algorithm (BFS):** Keep selecting the vertex with in-degree 0 and append it to the back of the topological order. * **Idea for Khan’s Algorithm (DFS):** Perform DFS on s, then append s to the back of the topological order. Repeat this for all vertices that are not visited by s (i.e. in another component). **After every vertex in all components have been visited**, **reverse** the entire topological order.   A close up of text on a white background  Description automatically generated  A close up of text on a white background  Description automatically generated |

**SSSP Problem**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **SSSP Problem Solved** | **Remark** |
| **Modified BFS**  O(V + E) | **Unweighted** graph/ graph with **constant weight** edges | * The graph can be directed / undirected * **Data structure:** Adjacency List * Key idea: same as normal BFS   + just replace the visited array with distance array   + increase the distance by 1 when move to the next immediate vertex. |
| **Bellman Ford’s**  O(VE) | 1. General weighted graph  2. Detect negative weight cycle  **TRIES ALL SOLUTIONS, PICKS THE BEST** | * The graph can be directed / undirected * **Data structure:** Adjacency list / edge list   + Edge list can be used here cuz we are not really doing any traversal here: just keep relaxing the edges * To detect negative weight cycle, simply run Bellman Ford again after executing it for V – 1 times. If there’s any edge that can be relaxed, then there exists a negative weight cycle. * **Key Idea:**    + Do initialization   + Perform the next step for V – 1 times   + For every edge, if the edge can be relaxed (i.e. if D[v] > D[u] + w(u, v)), relax it and set p[v] = u * Bellman Ford in one sentence: relax all the edges for v -1 times * **Optimization:** Stop when no more relaxation after a pass (set a flag) * Show performance on **small graphs** (note the graph can have negative edge, but cannot have negative weight cycle, will give wrong answer is sssp is undefined for graph with negative weight cycle) |
| Original Dijkstra’s  O((V+E)logv) | **Graph with no negative edge** | * Each vertex is only processed **once**. * **Key idea:**   + Enqueue all vertices as integer pair (INF, v) except the source which will be enqueue as (0, s) into the PQ   + Loop (Keep doing this until the PQ is empty)   + Remove vertex u with the **minimum d** from PQ, **add** u into Solved, and **relax** all the outgoing edges from u.     - Update D[v] and (d, v) in the PQ once an edge(u, v) is relaxed.     - Use BST to implement this PQ * **Original** Dijkstra can be used to detect whether the graph contains **negative weight EDGE** by reporting true when it tries to update the shortest distance of a vertex in the PQ but can’t find it (i.e. the vertex has been added into the Solved set, but now we are updating it again, which means that there exists a negative edge. Refer to lemma 2 of Dijkstra proof. ( because every vertex in a positive weight graph will only be processed once, if it is processed more than one time, that means that there is negative weight edge) * May need to combined with the approach is to check whether at any point D[u] is lesser than D[u] < D[u] + w(u, v) * However, Dijkstra cannot be used to differentiate between negative weight edge and negative weight cycle because in both cases same thing will happen, i.e. updating a vertex that has been previously added into the Solved set. |
| Modified Dijkstra’s  O((V+E)logv) | Graph with **no negative weight cycle** | * Allow a vertex to be processed multiple times (due to the presence of negative edge) * Run **much faster** than Bellman’s Ford in general case * Can be trapped if there is negative weight cycle! * **Key Idea:**   + Enqueue the source vertex-distance pair (0, s) into a normal PQ   + Loop (Keep doing this until the PQ is empty)   + Remove vertex u with the **minimum D** from the PQ. Let this D be d.   + Check if D[u] == d (i.e. the up-to-date copy)     - If yes, process it by relaxing all its outgoing edges.       * If an edge(u, v) is relaxed, update D[v] and enqueue a new (d, v) into the pq     - If no, discard the pair |

**Special Cases**

|  |  |
| --- | --- |
| **Cases** | **Algorithm** |
| Tree | BFS / DFS  **Complexity:** O(V)   * as O(V + E) = O(V + V – 1) = O(V) |
| Unweighted Graph | BFS only  **Complexity:** O(V + E)   * Note that DFS can’t be used here! |
| DAG | **One-pass** Bellman’s Ford  **Complexity:** O(V+E) + O(E) = O(V + E)   * Condition: Process the edges in **topological order,** i.e. you relax all the outgoing edges of the vertices listed in the topological order, starting from the leftmost vertex. |
| Graph with no negative weight **edge** | Original Dijkstra’s  **Complexity:** O((V+E)log V) |
| Graph with no negative weight **cycle** | Modified Dijkstra’s  **Complexity:** O((V+E)log V) |
| Shortest path cost of multiple sources to the **same destination T** | **Flip all the edges in the graph** (if it is directed) and perform a SSSP algorithm from T. Then the shortest path from a vertex v to T D[v->T] is simply D[T->v] in the **transformed** graph. |



**Pseudocodes**

**Initialization & Relaxation**

Initialization

* set all D[v] to INF
* set all p[v] to -1
* set D[s] to be 0

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Modified BFS**

**A screenshot of a cell phone

Description automatically generated**

**Bellman Ford’s**

**A screenshot of a cell phone

Description automatically generated**

**Modified Dijkstra**

**A screenshot of a cell phone

Description automatically generated**

**APSP**

|  |  |
| --- | --- |
| **Algorithm** | **Remark** |
| Floyd Warshall’s  O(V3) | * Used to solve APSP problem * **Key Idea** * Use a 2d matrix for SP cost D[u][v] * Initialization   + At start, D[i][i] = 0, D[i][j] = weight of edge(i, j) exist, otherwise set to INF * In the nested loops   + If D[i][k] + D[k][j] < D[i][j]   D[i][j] = D[i][k] + D[k][j]   * Once the algo stops, D[i][j] contains the SP cost form i to j |

Note that there is a special case for APSP problem: when the graph is a **DAG**.

When the graph is a DAG, simply call one-pass bellman ford for all the vertices according to its topological order. Toposort is O(V+E), calling one-pass bellman ford one time is O(E) time, so calling it for V number of times (cuz we call it for all vertices in the topological order is O(VE) time. Thus, the overall time complexity is O(VE + V + E), which in general is faster than Floyd Warshall’s O(V3) time (Floyd Warshall is always O(V3)))

**Pseudocode (Floyd Warshall)**

**A screenshot of a cell phone

Description automatically generated**

**Applications of Floyd Warshall’s**

|  |  |
| --- | --- |
| **Application** | **Remark** |
| Print Actual SP | * Use a 2D predecessor array. P[i][j] stores the predecessor of j on the shortest path from I to j. * **Key Idea** * Initially, p[i][j] = i * In the loop   + If D[i][k] + D[k][j] < D[i][j],   D[i][j] = D[i][k] + D[k][j];  **P[i][j] = P[k][j]** // Update the predecessor of j |
| Transitive Closure | * To determine if j is reachable from i * Modify the matrix D to contain only 0 and 1   A screenshot of a cell phone  Description automatically generated  // Bitwise operator |
| Detecting +ve / -ve cycle |  |