AVL

Rec 5

Data Summarisation

Given a set of students (names) with heights and grades, implement query to calculate the average grade of all students tall than (student X).

`insert(name, height, grade)`

`findAverageGrade(name)`

- use either 2 BSTs (one name tree, another height tree) or one hash table storing the name and height with another height tree
- tree ordered by heights
- root node stores the total no. of students (`root.weight`) & their combined grade (`root.gradeSum`)

Decumulation

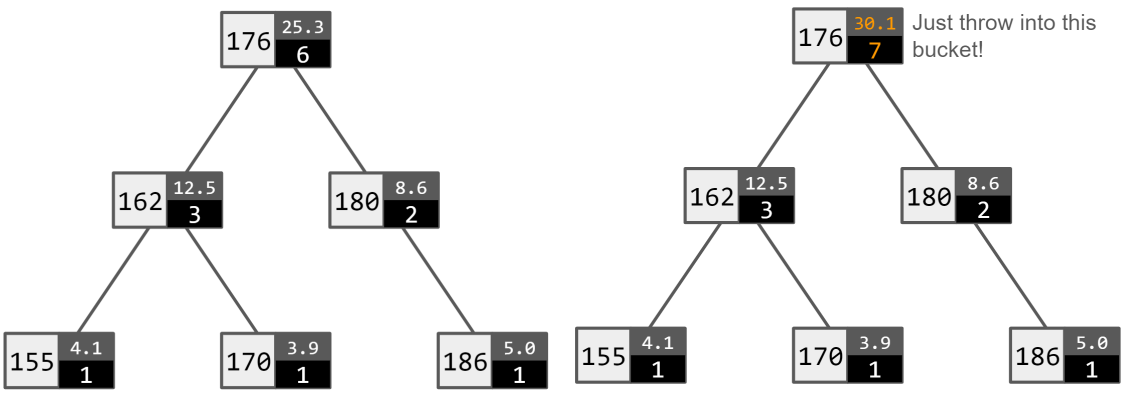
- deduct from the root's value the value of the query student X & those values of students who are shorter than X
- start from the root (initialise `tallerGradeSum = root.gradeSum` , `tallerWeight = root.weight`), as we traverse down the tree to X, deduct `current` value and add `current.rightSubtree` values until current = X
    - if X is smaller than the current node, add current node's values also
- return `tallerGradeSum = root.gradeSum` , `tallerWeight = root.weight`

Accumulation

- first find node X and traverse upwards to the root, adding the right subtree values

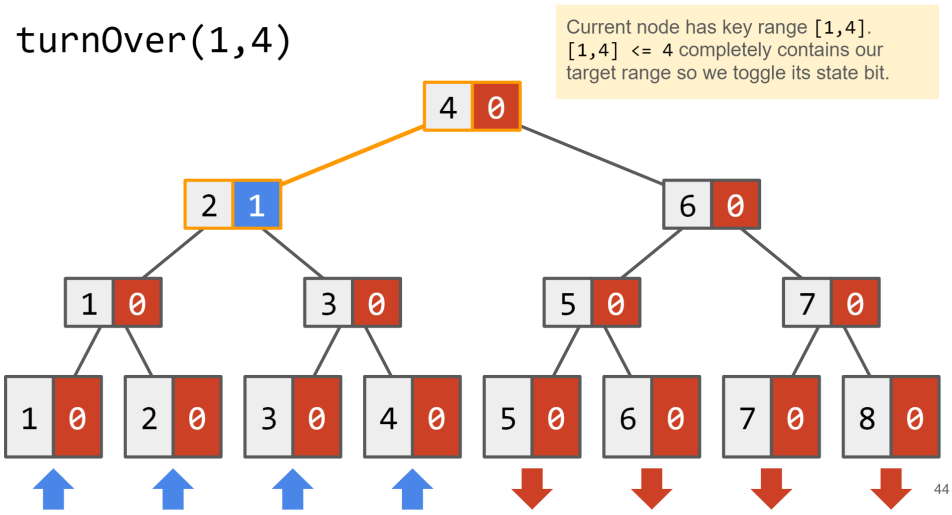
Duplicates

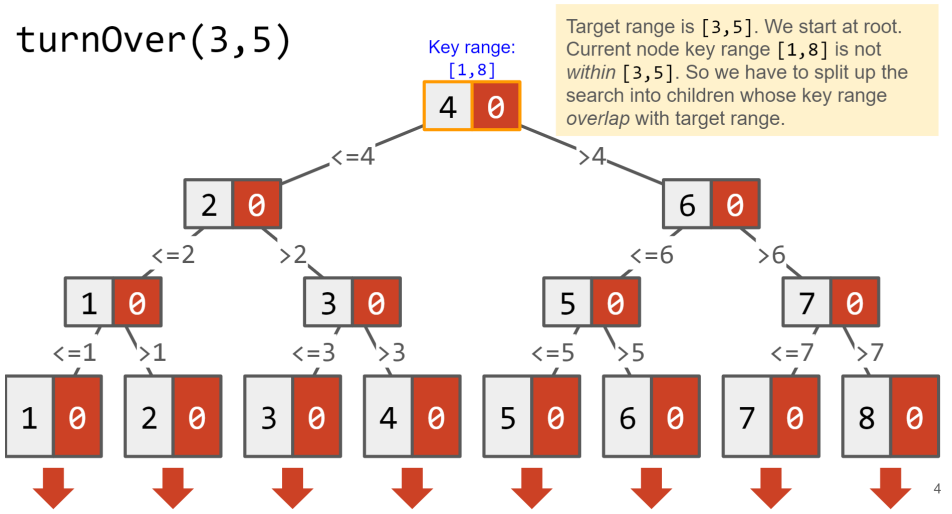
- add to the same node

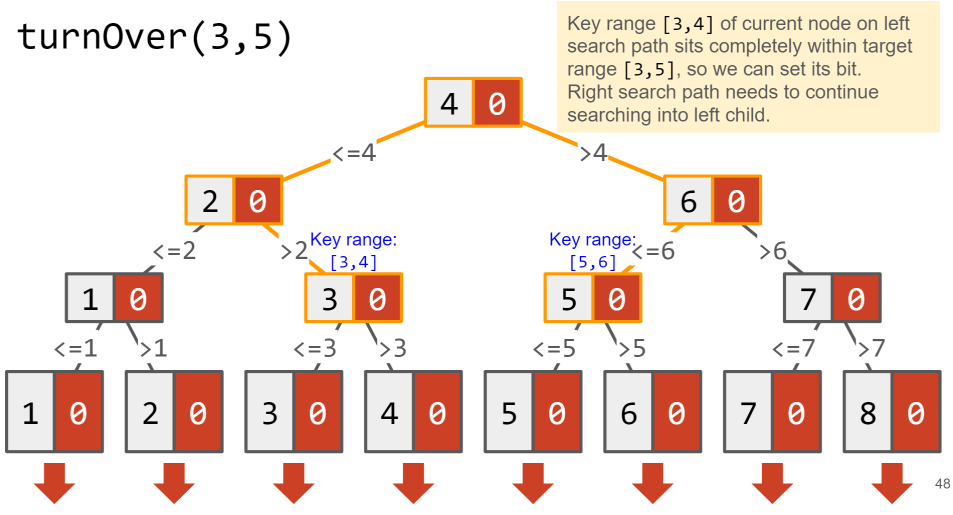


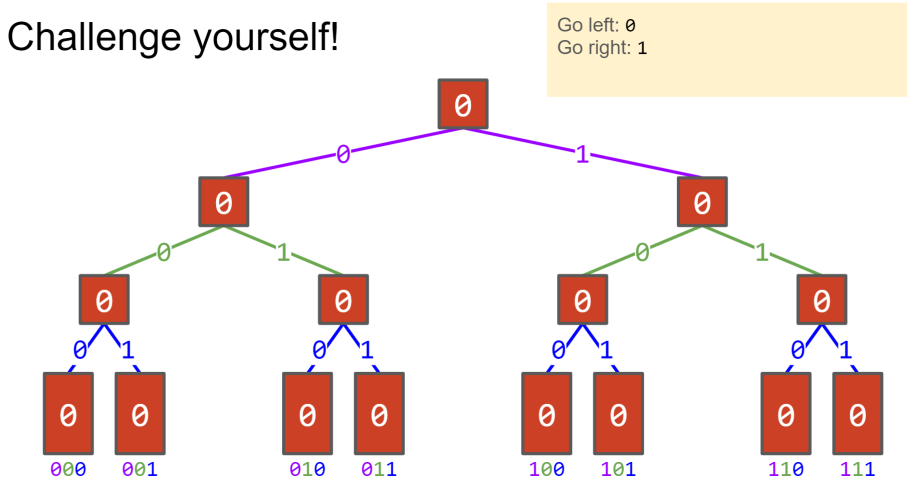
Interval Tree

- toggle the state of a range of (leaf) nodes
- each node contains the mid-value, left subtree represents <= mid, right represents > mid
- each node stores the state of the leaves









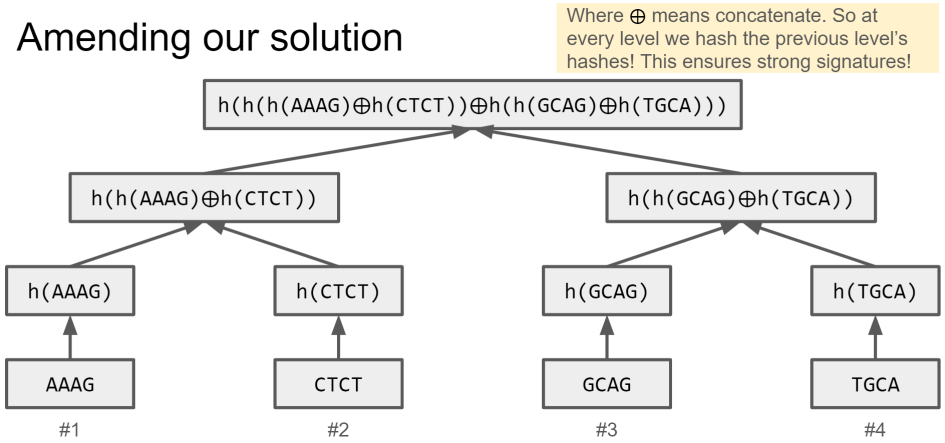
Determining if the final state is same as the initial state

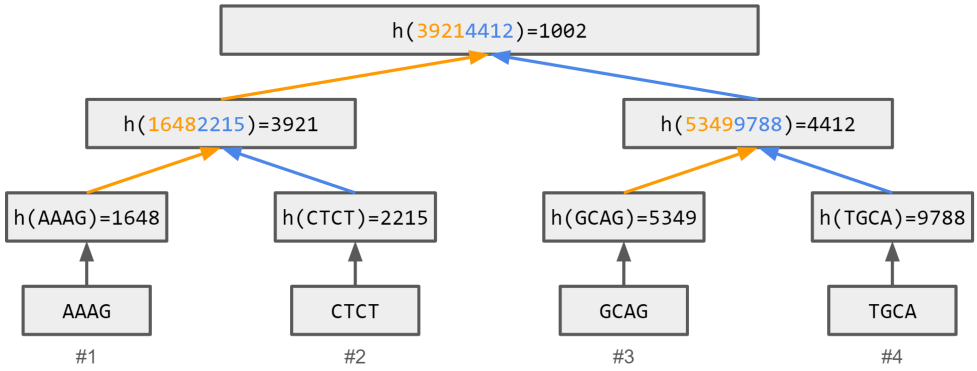
3 equivalent methods:

1. Count the number of 1's in the root-to-leaf path, even number means card is same as initial state, odd means otherwise
2. XOR all the bits encountered in the root-to-leaf path, result of 0 means card is same as initial state, 1 means otherwise
3. Initialize a variable direction d with initial face direction of cards; traverse the tree from root to leaf, every time we encounter a 1, we toggle the direction of d; at the end of the traversal, d will be the card's final face direction

Merkle Tree

- for effective comparison of arrays of items
- a bBST where leaf nodes store hash of its data and each non-leaf node takes and concatenates the values of children and hash,which is stored as its own value.
- same value at root indicates that all items (leaf nodes) are identical (provided a good hash function with strong uniqueness is used).





Hash Tables

Rec 6

imperfect hash function → Collisions

- If a deleted photo and a non-deleted photo is hashed to the same value, we would be led to believe that the deleted photo exists locally.
- If tried to resolve using chaining → worsen time complexity: no longer O(1), could be O(n)

> 1. Let k be some integer (which may depend on n and m)
> 2. Repeat:
>   1. Randomly pick a hash function h that maps a photograph to an integer in [1, k]
>   2. For each photo $\ell_i : i \in [1, m]$ on Alice's local computer
>   - Compute its hash value $h(\ell_i)$
>   - Save $h(\ell_i)$ to a local file H_ℓ
>   3. For each photo $r_i : i \in [1, n]$ on the remote server
>   1. Compute its hash value $h(r_i)$
>   2. Save $h(r_i)$ to a remote file H_r
>   4. Download H_r to Alice's local computer
>   - If $|H_r| - |H_\ell| = n - m$,
>   1. Download the photos r_i whose hash value $h(r_i)$ is in H_r but not in H_ℓ
>   2. Terminate the repeat loop
>   - Else, continue the loop to look for a better hash function

- $|H_r| = n$ and $|H_l| = m$: perfect function - uniquely identify all photos
- $|H_r| - |H_\ell| = n - m$: uniquely identify all deleted photos
  - After having δ photos deleted locally, the number of hash values also decrease by δ allows for collision between non-deleted photos
- $|H_r| - |H_\ell| < n - m$: collision between deleted photos or deleted&non-deleted photos

Next, it is known that a continuous subsequence of δ photos are deleted

- Find the index j of the first item that is available remotely but missing locally. Thereafter the recover the missing block from remote by downloading images [j, $j + \delta$).

Solution 1: Binary Search for 1st deleted photo

- Whenever we get an inconsistent hash values, we choose to search on LHS
- When the search converged, we'll end up with the leftmost deleted photo (i.e. 1st deleted photo in sequence)

> 1. Initialize $a = 1$ and $b = m$
> 2. While $a \neq b$ do:
>   1. $x = \lfloor (a + b)/2 \rfloor$ (i.e. median index)
>   2. Compute hash $h(r_x)$ for *remote* photo r_x
>   - If $h(r_x)$ matches *local* photo hash $h(l_x)$, update $a = x + 1$ (i.e. continue searching in the RHS range $[x + 1, b]$)
>   - Else, update $b = x - 1$ (i.e. continue searching in the range $[a, x - 1]$
>   3. Return a (when $a = b$)
> 3. The photos to download is therefore $[r_a, r_a + \delta]$

Solution 2: Merkle Tree

> 1. Hash the first $n/2$ photos on the client and server and compare
>   1. If the hashes equate, recurse on the second $n/2$ half
>   2. Else, recurse on the first half
> 2. Repeat until you find a missing photo (1st deleted photo)
>
> - This follows a *root-to-leaf* path in the Merkle tree, building the relevant parts of the tree *on the fly* (thus only $O(1)$ space)

2SUM Problem

Solution 1: Lower & Upper pointers ⇒ T(n) = O(n log n + n); S(n) = 1 for in-place quick sort

1. Sort the array
2. a + b > c: shift upper pointer (initialised at index 1)
3. a + b < c: shift lower pointer (initialised at index 1)
4. a + b = c: found

Solution 2: Hash Table ⇒ T(n) = O(n); S(n) = O(n) -> can return num of item instead of just the value

- Create hash table H
- Go through each item a_i in array
  - Check if $(x - a_i)$ exists in H, if it does we found a pair and we are done
  - Insert into H the key a_i (price) and value `i` (item number)

3SUM Problem

Solution 1: Converging Pointers ⇒ T(n) = O(n log n + n^2); S(n) = 1 for in-place quick sort

- Sort the array
- Go through each item a_i in the sorted array

- Check if we can find a pair using `2SUM(x−a_i)` in the subarray after a_i using converging pointers

Solution 2: Hash Table ⇒ T(n) = O(n^2); S(n) = O(n) -> can return num of item instead of just the value

- Build the same hash table as before
- Go through each item a_i in the array
- Use the previous 2SUM hash table solution with the target sum as $x - a$

$N - SUM(x, arr) = (N - 1) - SUM(x - a_i, arr \backslash a_i)$ for all a_i in the array *arr*

Graphs

Rec 7: SSSP

Graph modelling

- Representation of vertices
  - location, direction and/or other requirements
- Representation of edges
  - directed/undirected
  - weighted/unweighted
- Representation of Graph
  - Adjacency list
  - Adjacency matrix
  - Edge List (e.g. in the case where most of the vertices are connected, can use a list to store unconnected vertices instead so as to reduce S(n))
- Cyclic/Acyclic

SSSP

- unweighted/uniformly-weighted: BFS
- weighted acyclic: Topo-sort, Dijkstra
- weighted possible cycle: Bellman-Ford

Recover paths

- When a vertex is being visited, store the *previous (parent)* vertex from whence it came.

Solution Techniques:

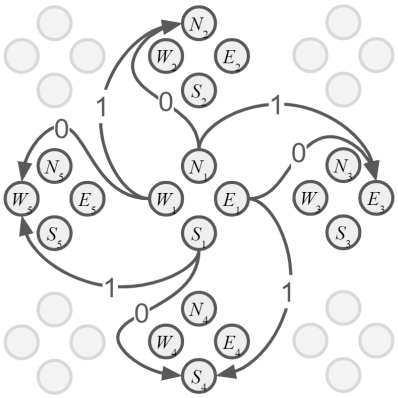
1. Create an edge connect to the same vertex
2. Super-node: connect starting vertices of the same "weightage" to a dummy/super node with weight 0 ⇒ if bfs, deduct off the dummy edge in the final step after backtracking the path
3. reverse the direction for directed graphs: let the destination vertex be the new source vertex
4. stacking of graphs:
   - make a duplicate graph and set the destination as the corresponding vertex in duplicate graph, connect the vertices from original graph to their corresponding vertices in the

duplicate graph with weight 0 for weighted.
    - (if necessary) connect the neighbour nodes in duplicate graphs
    - for unweighted/uniformly weighted, deleted the extra length/distance at the end
5. Find Longest path:
    - sum: negate the values ⇒ run SSSP
    - product:
        - if all positive values ⇒ run modified SSSP by changing the sum to product;
        - if there's potential non-positive values, log the values and run normal SSSP
6. Powers of adjacency matrix
    - represent jumps (more than one step taken)
    - each matrix entry represents the length of the walk between the two nodes
    - matrix multiplication = O(n^3)

Question 1

- cannot turn left
- minimise right turns

Problem states: current location & direction



> Instead of letting 1 vertex to representation each intersection, use 4 vertices for each intersection
>
> - Each intersection has 4 directions, as represented by the 4 vertices
> - Connect each vertex with outgoing edges to the *only two* valid states in the next step:
>   - going straight and
>   - turning right.
> - In addition, we assign edge weights of 0 for going straight and 1 for turning right.
> - Run SSSP (Dijkstra)

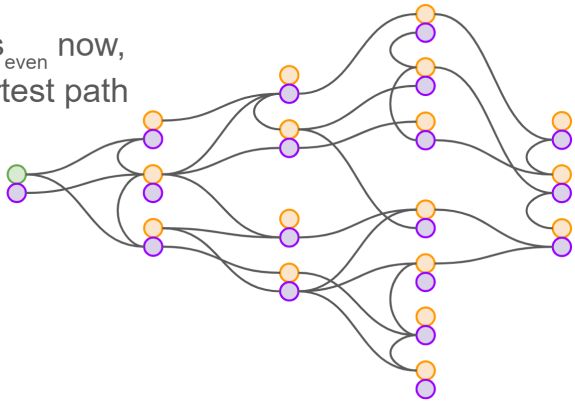
Question 2

- can only drive on even roads on even days and likewise for odd roads&days

(a) Suppose Manon will have to leave from Paris on an *even* day and she will *not* be spending overnight in any city.

Problem States: current location (city) & current day parity (odd/even)

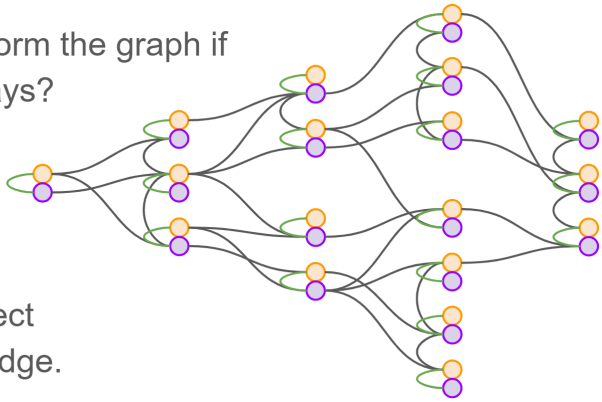
Running BFS from $\text{paris}_{\text{even}}$ now, how do we find the shortest path to Moscow?



- Split each city vertex into arrivals on odd/even days respectively.
  - E.g. split vertex v in v_{even} and v_{odd}
- For *even* roads from u to v, we connect between u_{even} and v_{odd}. For odd roads, the opposite is done.
- Run BFS
- Determine the shorter path between the shortest paths to vertices $Moscow_{even}$ and $Moscow_{odd}$.

(b) Suppose Manon can now choose to stay overnight.

How should we transform the graph if we allow overnight stays?



Answer:

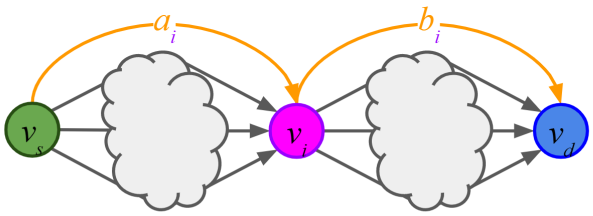
For every city v, connect v_{odd} and v_{even} with an edge.

To determine which is the better day to leave:

- Connect $Paris_{even}$ and $Paris_{odd}$ to a *dummy source* vertex from which we will run BFS.
- Deduct off the dummy edge in the final step after backtracking path.

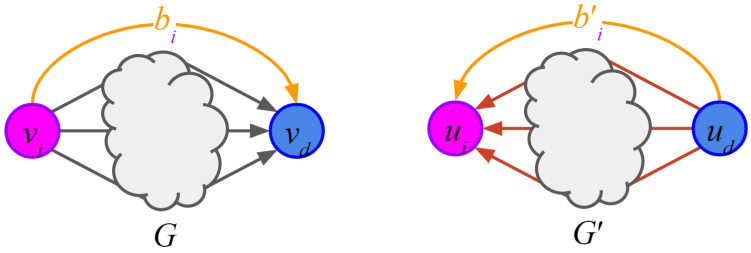
Question 3

Solution 1



a_i: Shortest distance from source vertex v_s to mandatory vertex v_i
b_i: Shortest distance from mandatory vertex v_i to destination vertex v_d

Objective: We want to find $a_i + b_i$ that is the *minimum* across all mandatory vertices $i \in [1, k]$.



- Suppose we reverse all edges in G and create a new graph G'.
- Then by construction, the shortest distance from v_i to v_d in graph G must be the same as the shortest distance from u_d to u_i in graph G'
- Thus $b_i = b'_i$

1. Run SSSP from v_s to obtain shortest paths $a_1, \ldots, a_k$. ⇒ BFS: O(V+E)
2. Reverse all edges in graph G ⇒ Construct new graph w adjacency list: O(V+E)
3. Run SSSP from destination vertex v_d to obtain shortest paths (in reverse) $b_1, \ldots, b_k$. ⇒ BFS: O(V+E)
4. Find the minimum of $a_i + b_i$. ⇒ O(V)

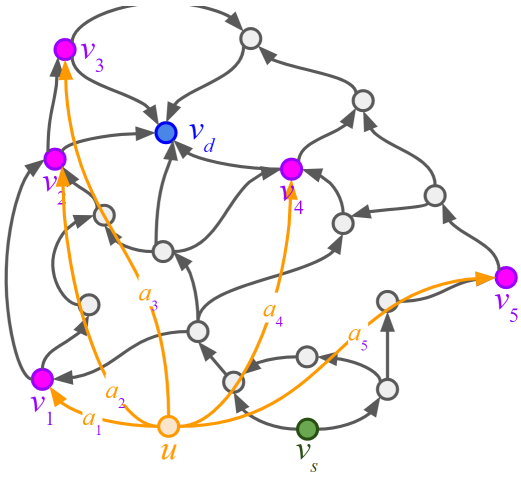
T(n) = O(V+E)

Solution 2

Create a dummy vertex u and connect it to all mandatory vertices via outgoing edges.

In addition, we transform the graph into a *weighted* one:

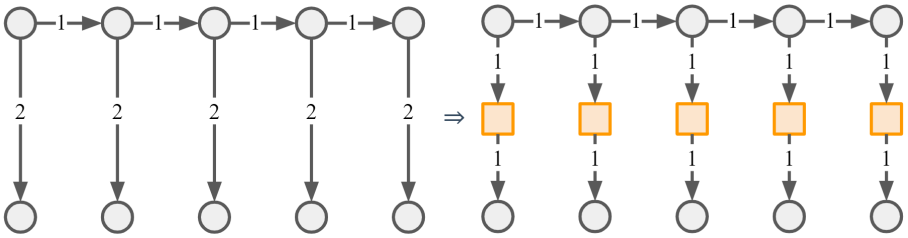
- Assign weight a_i to edge $u \rightarrow v_i$
- Assign weight 1 to all other edges



- Run SSSP from v_s to obtain shortest paths $a_1, \ldots, a_k$. ⇒ BFS: O(V+E)
- Run SSSP from u

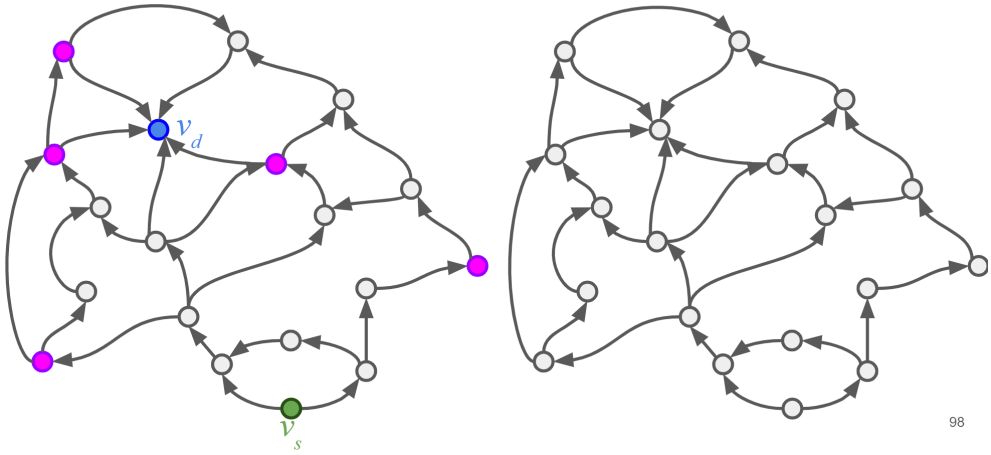
Solution 3

Suppose only BFS is allowed to search for the shortest path, how can we transform the graph?

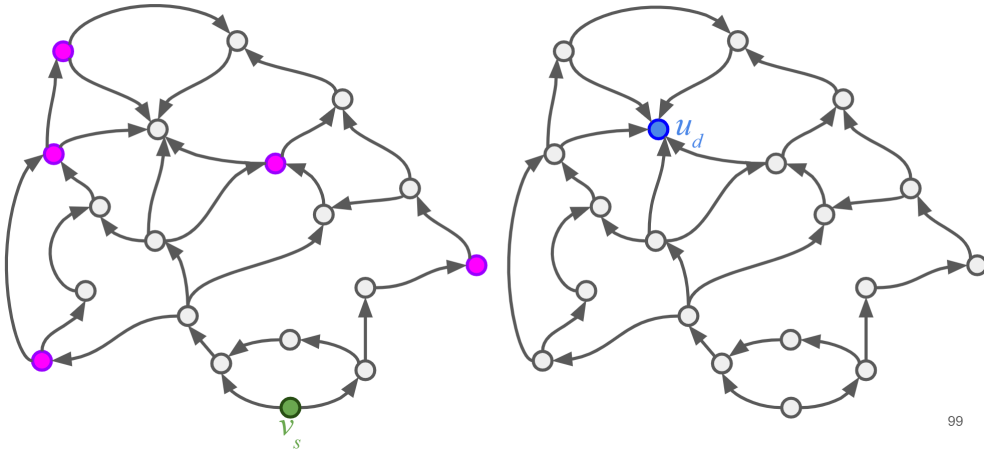


> Insert a dummy vertex between edges of weight 2 so as to split it into 2 edges of weight 1 each.
>
> This of course incurs more space..

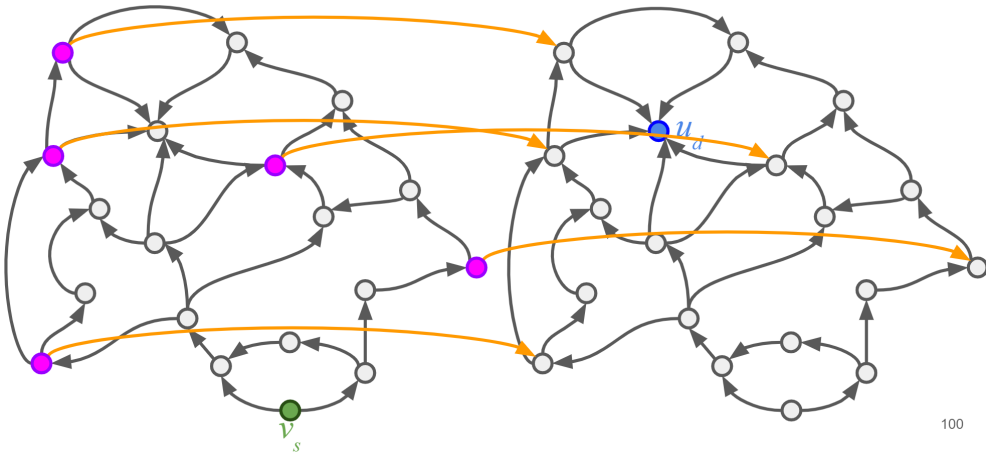
Step 1: Make a duplicate graph G'



Step 2: Set vertex in G' as our new destination



Step 3: Connect outgoing edges to second graph



- Run BFS
  - The SSSP exploration will now be forced to visit the mandatory vertices because they are now the only “bridges” to the destination vertex in graph G'.
- delete off the length due to the "bridge" at the end

Rec 8

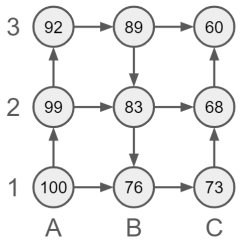
Question 1

Problem 1: Description

- Each piece is associated with reward points
- A valid move is one in which the piece is moving into a square of < points than the current square
- Goal: find the sequence of valid moves that give the most points

	A	B	C	D	E	F	G	H
8	32	44	92	32	14	10	8	5
7	20	54	61	71	18	16	12	32
6	46	80	88	90	9	18	90	7
5	61	83	30	29	22	20	25	10
4	97	50	51	57	30	80	78	15
3	92	89	60	54	88	47	82	33
2	99	83	68	50	23	92	9	68
1	100	76	73	99	55	74	73	22

Solution 1: Model reward points as Vertex Weights

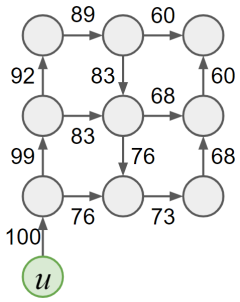


The most rewarding path to a vertex v is the *maximum* of all the most rewarding paths to u where $u \to v$, plus v's weight.

This means if we can solve all the subproblems *in sequence*, ending with the destination vertex, we would be able to compute the most rewarding path to it in a single pass.

- Topo order: u will appear before v in the topological sort.
  - by the time we get to v, we are guaranteed to have obtained the best paths for all the u that can directly go to v.

Solution 2: Model reward points as Edge Weights with a super-node



- Negate all the edges (will be acyclic)
- Run SSSP with vertex u as the source

Solution 3: Dynamic Programming

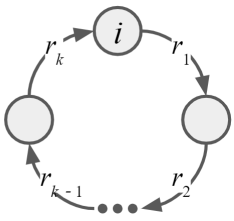
> DP[v] = max(DP[u_1] + w(u_1, v), DP[u_2] + w(u_2, v), ...)

Question 2:

- There are currently n currencies being traded
- You have a matrix R containing all exchange rates where R[i , j] is the exchange rate from currency i to j
- Exchange rates are not symmetric: R[i , j] ≠ R[j , i]
- No arbitrage possible: if you start with one currency and then convert it to another, and so on, and then back to the first currency, you will end up with *no more* money then what you started with
    - If there exists a cycle from i back to itself such that the total product of edge weights within that cycle is more than 1, then we can infinitely get more money than what we started with.

$$i \times r_1 \times r_2 \times \ldots \times r_k \leqslant i$$
$$r_1 \times r_2 \times \ldots \times r_k \leqslant 1$$

Thus graphically, this means that we won't have a cycle where the product of all its edge weights exceed 1.



Graph Modelling

- Vertex: currency
- Edge(i, j): exchange rate

Solution 1: modify SSSP (longest path)

> Assuming we are relaxing edge (u , v , w)
>
> - "Product instead of sum": Compare $D[u] \times w$ instead of $D[u] + w$
> - "Maximization instead of minimization": Update when new estimate is *greater* than previous instead of lower
>   - if $D[u] \times w > D[v]$ then $D[v] = D[u] \times w$

Solution 2: modify graph

maximimize length = maximimize $r_1 \times r_2 \times \ldots \times r_k$

maximimize log length = maximimize $\log(r_1 \times r_2 \times \ldots \times r_k)$

= maximimize $\log r_1 + \log r_2 + \ldots + \log r_k$

Our new definition for length!

So now the graph becomes:

- negate the log edge weight and run SSSP

Answer: Shortest distance $\delta = -\log r_1 - \log r_2 - \ldots - \log r_k$ now no longer represents the max currency conversion product $L = r_1 \times r_2 \times \ldots \times r_k$.

How do we *recover* L? Here's how:

$$\begin{aligned}
\delta &= -\log_a r_1 - \log_a r_2 - \cdots - \log_a r_k \\
-\delta &= \log_a r_1 + \log_a r_2 + \cdots + \log_a r_k \\
a^{-\delta} &= a^{\log_a r_1 + \log_a r_2 + \cdots + \log_a r_k} \\
&= a^{\log_a r_1} a^{\log_a r_2} \ldots a^{\log_a r_k} \\
&= r_1 \times r_2 \times \cdots \times r_k \\
&= L
\end{aligned}$$

Question 3: Min Edit Distance

Operation	Behaviour	Cost
`insert(i, c)`	Inserts a character `c` at position `i` in the string.	$ins
`delete(i)`	Removes character at position `i` in the string.	$del
`replace(i, c)`	Substitutes a character at position `i` in the string with character `c`.	$rep

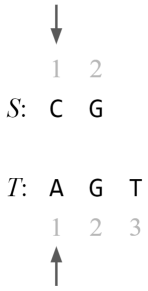
- cost of replace <= cost of insert + delete (to justify its existence)

Graph Modelling (DAG)

- Vertex: string state ⇒ alignment of characters
  - A unique alignment correspond to
    - Multiple edit sequences (permutations of each other)
    - A single edit cost
  - A unique alignment is represented by a *unique path* from source string S to target string T
- Directed Edge: state transition - an outgoing edge from S to T denotes a single character edit which transforms S to T.
- Weight: cost of transition
- Path from initial to goal = valid character edits

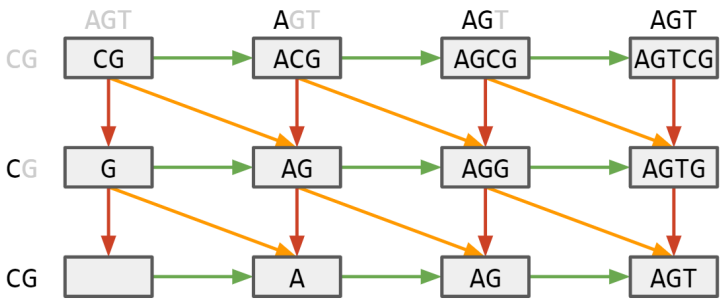
    - The longest edit sequence length is $m + n$, which correspond to deleting every character from S and inserting every character in T.
    - shortest path between two strings ⇒ the alignment between them with the cheapest cost
- 2D array
    - Current row `i` represent current character in S
    - Current column `j` represent current character in T
    - `N(i, j)` to be the node at row `i` column `j`.
    - progress from left to right `(j → j+1)`, insert the character `T[j]` at position `j`.
    - progress top to bottom `(i → i+1)`, delete the character in `S[i]`.
    - progress diagonally, replace `S[i]` with `T[j]`

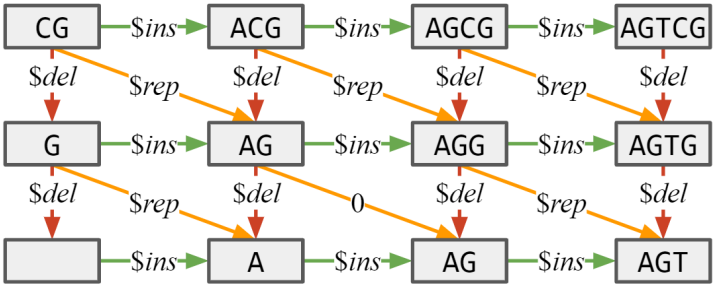
Alignment (acyclic)



- insert a character from the *target* string T
- delete a character from the *source* string S
- We have two pointers, each pointing to the current character in S and T respectively, starting at their first characters.
    - Everything *to the left* of the current characters represent prior sub-sequences that has already been fixed/matched.
- We want to *sequentially* decide on a *fixture* for the current character (to a position in the *other* string)
    - Whenever we insert, pointer for T advances
    - Whenever we delete, pointer for S advances
    - Whenever we replace, *both* pointers advance
- When both pointers have been fully exhausted, we obtain an unique alignment
    - fix each of the m characters in S to a position in T
    - fix each of the n characters in T to a position in S ⇒ total (m+n) fixtures
    - a match/replace entails 2 fixtures (one from each string), while insert and delete entail 1 fixture each.

Every path from S to T entails fixing a total of $m + n$ character positions to complete an alignment!





$$\text{MED}(N(i,j)) = \min \begin{cases} \text{MED}(N(i-1,j)) + \$del \\ \text{MED}(N(i,j-1)) + \$ins \\ \text{MED}(N(i-1,j-1)) + \$rep & \text{if not correspondence} \\ \text{MED}(N(i-1,j-1)) & \text{if correspondence} \end{cases}$$

> When we replace a character with itself, then the cost should be zero because this is a correspondence
>
> - no. of vertices = mn
> - no. of edges = $3(mn-m-n+1)+m-1+n-1 = O(mn)$
> - `DP[i][j] = min(DP[i][j-1] + $ins, DP[i-1][j-1] + $rep, DP[i-1][j] + $del, DP[i-1][j-1])`
>
> DAG: Relax vertices in topo order
>
> - Time to obtain topological order: $O(V+E) = O(mn)$
> - Time to relax all vertices: $O(V) = O(mn)$
> - $T(n) = O(mn)$

Union-Find Disjoint Set & Heaps

Heap search is O(n)

- optimise by using a hash table to map keys to their node pointers
- T(n) = O(1); S(n) = O(n)

Rec 9

Question 1

- a list of n tasks are stored in array W
- `W[i] = 0` if task `i` is not yet completed
- `W[i] = 1` if task `i` is completed

Operation	Behaviour
`lookup(i)`	Returns the value of `W[i]` - O(1): just maintain the binary array W
`mark(i)`	Marks task `i` as completed, i.e., sets `W[i]←1`
`nextTask(i)`	Returns the next task from `i` onwards that is not yet completed, i.e., the next index `j ≥ i` where `W[j]=0`.

(a) Design a data structure where the `mark` and `nextTask` operations complete in $O(\log n)$ time, worst-case.

Solution 1: AVL tree to store incomplete tasks

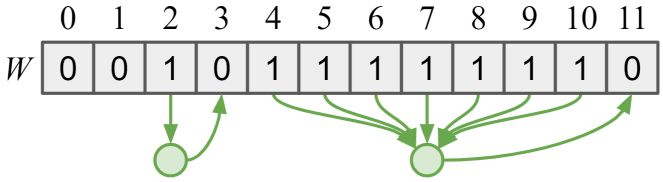
- Key `i` in the tree means task `i` is *not yet* completed
- Initially all the tasks are in the tree: store in order to minimise initial penalty to O(n) instead of O(n log n)
- `mark(i)` = `delete(i)` : O(log n)
- `nextTask(i)` = `successor(i)` : O(log n)

Solution 2: AVL tree to track the disjoint interval of completed tasks

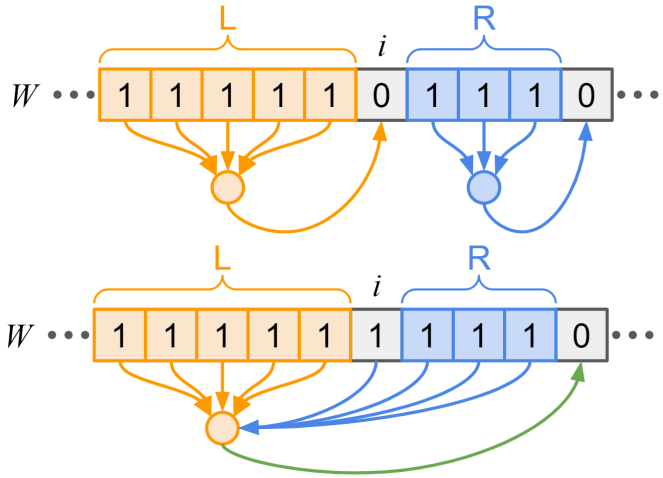
- Key value pair `(i, j)` in the tree means all the tasks from `i` to `j` inclusive has been completed
- If task `k` is within interval `[i, j]` then `nextTask(k) = j+1`

(b) Design a data structure where the `mark` operation runs in $O(\log n)$ *amortized* time and `nextTask` operation runs in $O(1)$ time, worst-case.

Solution 1: Indirection nodes (weighted union idea)

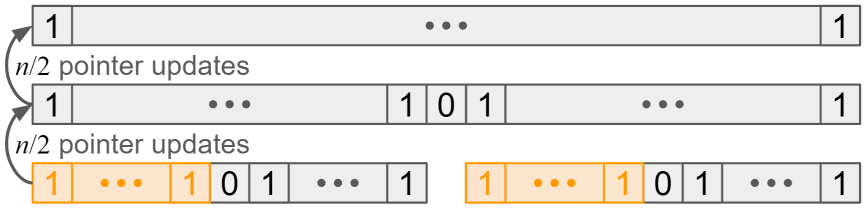


Every contiguous subsequence of *completed* tasks is connected to an indirection node which points to the next *uncompleted* task.



> If we were to complete task `i`,
>
> - We always update the pointers on the *smaller* of the two parts to point them to indirection node of the larger part.
>   - always pick the smaller problem to resolve
> - Remember to update the indirection node pointer if necessary.
> - CAUTION: should not point one indirection node to another, could result in n indirection nodes pointing to one another ⇒ O(n)

- Best case - mark each task in sequence, each time only updating/creating 1 pointer: O(n)
- Worst case - O(n log n) ⇒ amortized cost = O(log n)



Realise going from one level to another will *always* entail $n/2$ pointer total updates in the worst case. Since we are partitioning the subsequences into halves at every level, we would get $O(\log n)$ levels deep. This means marking all n tasks incurs $O(n \log n)$ pointer updates in the worst case.

(c) Design a data structure where the `mark` operation runs in worst-case O(1) time and `nextTask` operation runs in O(α(n)) amortized time(where α(n) refers to the time complexity of the inverse Ackermann function, i.e., the amortized time for executing n operations on a union-find data structure).

Solution: Union-Find Disjoint Sets (UFDS)

- One set for each contiguous subsequence of completed tasks
- The root of each up-tree serves as an indirection node, i.e. point to the next uncompleted task

`mark`

- Best case - Completing task `i` does not extend any existing sets. O(1)
  - i.e. `i` is an island
- Worst case - Completing task `i` merge/unions 2 existing sets: O(αn) amortised time w/o optimisation
  - i.e. either `i` is sandwiched by 2 adjacent sub-sequences
  - Main contributor of time is traversing up to the roots of `i` and `j` respectively.

- if we execute the union operations on the roots directly without having to look for them, then we do not need to spend the O(α(n)) amortised time.
- Here we are extending *sequences* instead of random sets.
- Only the boundaries of these sequences increases
- We simply need to maintain that the *inclusive ends* of a completed sequence point to the root node directly. ⇒ O(1)

> Maintain the following whenever a union operation occurs:
>
> - For each set, we maintain the next available task, the maximum index in the interval, and the minimum index in the interval
> - For each index in the main binary array, if it is the endpoint of an interval, maintain a pointer to the root of the up-tree for the associated set
>
> When a union occurs, we can maintain these by looking at the next available task for each interval and choosing the later one, by updating the minimum and maximum, and by setting the pointers for the minimum and maximum array elements to the root.

`nextTask(i)`

- traverse to the root of the up-tree: O(α(n)) amortized time

Question 2: Leftist Heap

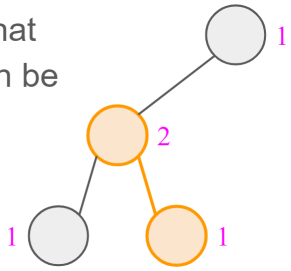
Term	Definition
Right Spine	The sequence of nodes traversed in a tree if you start at a node and always go right until you find a node with no right child (which may not be a leaf).
`u.rightRank`	The number of nodes along the right spine of a node u.
LEFTIST property	The property a tree satisfies if, for every node, `L.rightRank` ≥ `R.rightRank`, where L and R are the left and right child respectively. - `rightRank` of a non-existent child = 0. - When a node have only one child, it must be a left child - A node with no right child will always have a `rightRank` of 1 - The `rightRank` of a node depends solely on its right child and not its left
LEFTIST (Max) Heap	A tree that satisfies both the (Max) Heap order property and the LEFTIST property. - a max heap (complete binary tree) is also a LEFTIST heap, but not the converse

Operation	Behaviour
`insert(u)`	Insert node u into the heap.
`merge(t2)`	Returns the tree as a result of merging the heap with t 2.
`extractMax()`	Removes the node with the maximum key from the heap and return it.

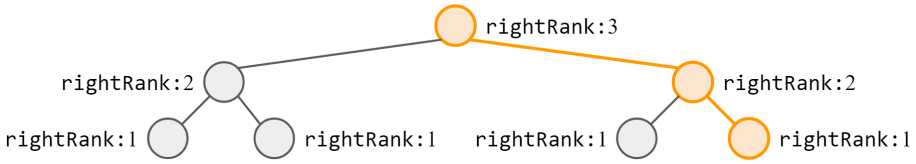
- Max height = n
  - "linear" tree with only left child
- Each node also has to store its `rightRank`

- `rightRank` of the root may not always be the max

  Answer: No. This also shows that the `rightRank` of a left child can be more than that of its parent.



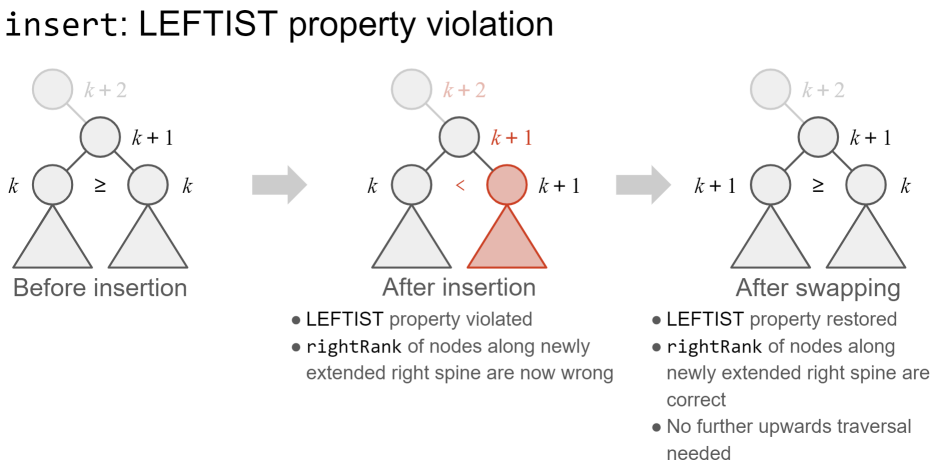
- If node v, `v.rightRank` $= k$ where k > 1
  - `v.left.rightRank >= k - 1`
  - `v.right.rightRank = k - 1`
  - There must be,
    - 2 nodes 1 hop away from v with `rightRank` at least $k - 1$
    - 4 nodes 2 hops away from v with `rightRank` at least $k - 2$
    - 8 nodes 3 hops away from v with `rightRank` at least $k - 3$,
    - and so on, with the number of nodes doubling at each level for k hops (to reach the leaves)



- Max `rightRank` = log (n + 1) or $\lceil log(n) \rceil$
  - complete binary tree

`insert`

- insert for Max Heap (insert at the leftmost left and bubble up) will also work and never violate LEFTIST property
  - not efficient: O(n) if need to bubble up the entire tree
- insert as a left child will never violate LEFTIST property

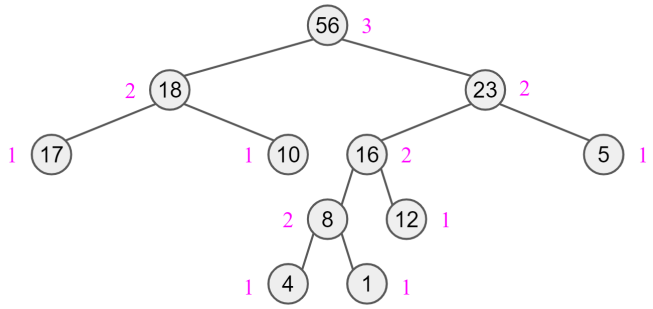


- insert along the right spine (length O(log n))

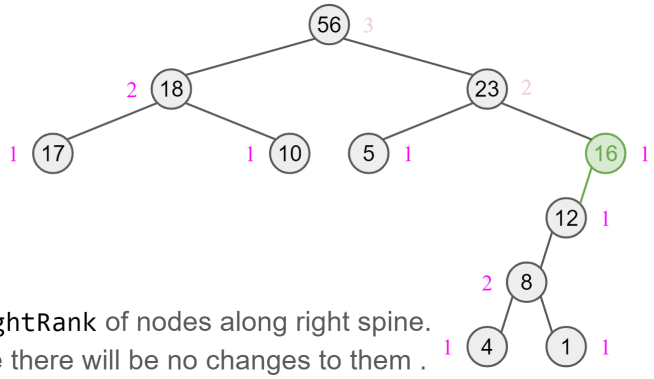
> We want to bubble down along the right spine:
>
> - Compare the root of our tree to the new node
> - If the new node has a larger priority, then swap the two

- Then, recursively insert the node (either the new node, or the old root if swapped) into the right subtree
- Base case: when you reach a node with no right child, and can just insert it
- Finally, update the `rightRank` maintained in each node, and swap the two children in order to maintain the LEFTIST property

```
// Algo 1: with swapping
root.insert(Node newNode)
    if (root.key < newNode.key): // maintain root to be the one with greater key
      swap(root, newNode)

    if (root.right == null): // Base case
      newNode.rightRank = 1
      root.right = newNode // Extend right spine
    else:
      root.right = root.right.insert(newNode) // recurse

    if (root.right.rightRank > root.left.rightRank)
      // swap the left and right children to maintain the rightRank
      swap(root.left, root.right)

    root.rank = root.right.rightRank + 1 // update rightRank

    return root

// Algo 2: w/o swapping
// When the current node being bubbled down is larger than the root node of the current subtree, just assign the entire subtree as the left child of current node and make current node the root of the new subtree.

root.insert(Node newNode)
    if (root.key < newNode.key):
      newNode.left = root
      root = newNode

    else:
      if (root.right == null): // base case
        newNode.rightRank = 1
        root.right = newNode // extend right spine

      else:
        root.right = root.right.insert(newNode) // perform insert (recursive call)

    root.rank = root.right.rightRank + 1

    return root
```



Swapping the left and right child of node 23.
We are done!



Update `rightRank` of nodes along right spine.
In this case there will be no changes to them .

`merge` + `extractMax()`

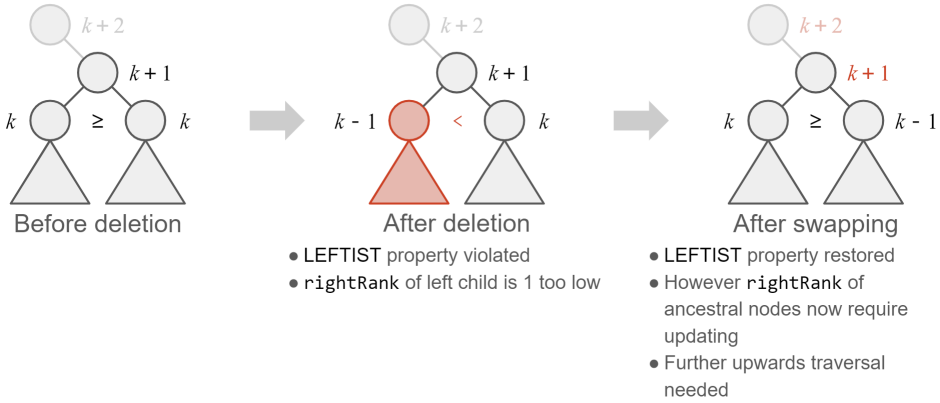
> Key idea:
>
> - Merge only along the right spine
> - Recurse down the tree:
> - Starting with the two roots of each tree respectively
> - Compare the keys between two roots
> - Whichever root is smaller, merge it with the *right child* of the other
>
> T(n) = O(log n) if each tree has size O(n)
>
> - at every step we are going down the right spine of one of the two trees by one level, the number of steps we take in total is at most 2 times the maximum `rightRank` in the trees.
>
> `insert` is a special case of `merge` where the tree to be merge is a single node
>
> `extractMax()` : remove the root and `merge(root.left, root.right)`

```
t1.merge(t2)
    if (t1.key < t2.key): // Maintain that t1 is larger (recursive call)
      return t2.merge(t1)

    if (t1.right == null): // base case
      t1.right = t2

    else:
      t1.right = merge(t1.right, t2)

    if (t1.right.rightRank > t1.left.rightRank)
    // If LEFTIST property violated, swap children
      swap(t1.left, t1.right)

    t1.rightRank = t1.right.rightRank + 1 // update rightRank
```

```
16
17        return rootOne
18
```

`delete(u)`

- could potentially violate the LEFTIST property
- need to traverse upwards, along the way, update the `rightRank` and swap any pair of siblings where the property is violated.
    - need to store the parent
    - at most one swap, but may need to traverse up the ancestral path to update (decrement) `rightRank`
    - unnecessary to traverse all the way up to the root to update the `rightRank`, `rightRank` of a node `u` only depends on the `rightRank` of its descendant along its right spine ⇒ stop at the 1st instance when current node is a left child
- T(n) = O(log n)



`updateKey(u, k)`

- `delete(u)` and `insert(v)` with key `k`

`search(u)`

- search all nodes for min/max heaps: T(n) = O(n)
- optimise by using a hash table to map keys to their node pointers
- T(n) = O(1); S(n) = O(n)

Combination of Concepts - Rec 10

Suppose further that distance between gas stations is always an integer number of miles apart. You want to determine the *best guarantee* for the furthest gas station (i.e. minimise max distance between petrol stations) from a start to end point

Definition

- Width: the edge with the heaviest weight
- Minimax distance: The *minimum possible* width of a path connecting two vertices.
- Best path: The path whose width is the minimax distance. i.e. the minimax path itself.

Graph Modelling

- Each city is identified by an *integer*
- The map is provided as an *adjacency list* :

- Each entry in the AL (i.e. a city): A list of roads running out of the city - outgoing edges
- Each road entry: A pair of
  - neighbour: the city to which the road connect to
  - weight: the maximum distance between adjacent gas stations along that road)
- All the roads are two-way

Solution comprises 2 stages:

Pre-processing stage:

- Initialize the service by *pre-processing* the given road map and storing it in a data structure (DS) for querying.

Query stage:

- Service user requests by *querying* the DS for the maximum distance between adjacent gas stations along the *best* routes recommended to them (i.e the minimax distances).

Solution 1: Binary search on query-time trimmed graph - suitable for small k

- No pre-processing required

> Query Stage:
>
> Binary search for the *lowest* $w \in [1, k]$ such that destination D is reachable from source S in the *trimmed* graph where all edges with weight > w are *removed*. ⇒ O(log k)
>
> - When implementing `reachableInTrimmedGraph(S, D, w)`, do not need to construct a new graph.
> - run modified DFS/BFS from S which ignores all edges with weight > w ⇒ O(V+E)
> - return true once we encounter D, else at the end of DFS/BFS, we return false.
>
> T(n) = O((V+E)log k))
>
> S(n) = O(V) for storing recursion-stack/queue for DFS/BFS

```
1   Query(S, D)
2   lo ← 1
3   hi ← k
4   while lo < hi do
5       w ← ⌊(lo+hi)/2⌋
6       if reachableInTrimmedGraph(S, D, w) then
7           hi ← w          // Search LHS
8       else
9           lo ← w+1        // Search RHS
10      end
11  end
12  return hi
```

Solution 2: Binary Search on pre-processed trimmed graphs - suitable for small k

> Pre-process: T(n) = O(k(V+E)); S(n) = O(kV)
>
> - Pre-process all component tables $C_i, i \in [1, k]$, using modified DFS/BFS where C_i corresponds to the component table in the graph trimmed of weights > i

Query: T(n) = O(log k)

- `reachableInTrimmedGraph(S, D, w)` can be achieved in O(1) by checking $C_w[S] = C_w[D]$
- Binary search for the lowest $i \in [1, k]$ such that $C_i[S] = C_i[D]$, where C_i is the vertex component table obtained from pre-processing step.

Solution 3: Modified SSSP Dijkstra

Query:

- define path length to be the max edge weight along the path

```
1   ModifiedRelax(u, v, w)
2       check ← max(est[u],  w)
3       // instead of est[u] + w in original relax
4       // can use even if there are negative edges
5       if (check < est[v]) then
6           est[v] ← check
7           end
```

- Since extending a path with an edge in this graph can only increase its maximum weight, we can safely run Dijkstra's on source S with the `modifiedRelax` subroutine.

T(n) = O((V+E) log V)

S(n) = O(V): visited array + PQ

Solution 4: Modified ASPSP Dijkstra - suitable for sparse graph, E = O(V)

Pre-process: T(n) = O(V(V+E) log V); S(n) = O(V^2)

- SSSP on all nodes
- store results in 2D array

Query: just lookup `result[S][D]` T(n) = O(1)

Solution 5: Modified APSP Flyod-Warshall - suitable for dense graphs, E = O(V^2)

Pre-process: T(n) = O(V^3); S(n) = O(V^2)

- compute APSP by running Floyd-Warshall's

Query: lookup `result[S][D]` T(n) = O(1

Solution 6: Query-time MST

Query:

- Obtain MST using Prim's or Kruskal's
- Run BFS/DFS on MST to obtain minimax distance from S to D by keeping track of the maximum weighted edge along the way

T(n) = O(E log V)

S(n) = O(V) : BFS stack/DFS queue

Solution 7: Pre-processed MST

Pre-process: T(n) = O(E log V)

- obtain MST using Prim/Kruskal's ⇒ E = O(V)

Query: T(n) = O(V + E) = O(V)

- Run BFS/DFS on MST to obtain path width from S to D, keeping track of the maximum weighted edge along the way

Solution 8: Pre-processed MST + Binary Search + Lowest Common Ancestor (LCA)

Pre-process: T(n) = O(E log V + V^2); S(n) = O(V log V)

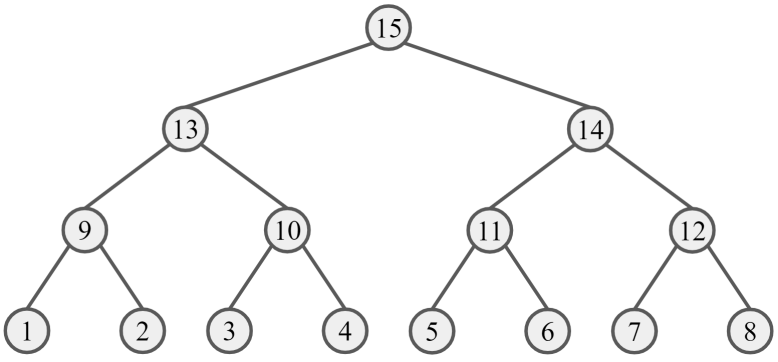
- Obtain MST T
- For each vertex in T, add O (log V) ancestral skip pointer to it and precompute lineage widths to them

Query: T(n) = O(log^2 V); S(n) = O(1)

- Obtain $h_S = \text{findLCA}(S, D)$, the number of hops up from S to their LCA
- Obtain $h_D = \text{findLCA}(D, S)$, the number of hops up from D to their LCA
- Obtain $w_S = \text{getWidth}(S, h_S)$, the width of lineage from S to the LCA
- Obtain $w_D = \text{getWidth}(D, h_D)$, the width of lineage from D to the LCA
- The minimax distance is $\max(w_S, w_D)$

LCA:

- The LCA of two nodes u and v in a tree is the deepest node (nearest to u and v) in the tree with both u and v as its descendants.
- LCA is unique and it exist for *any two* nodes in a tree.
- highest possible LCA = root node

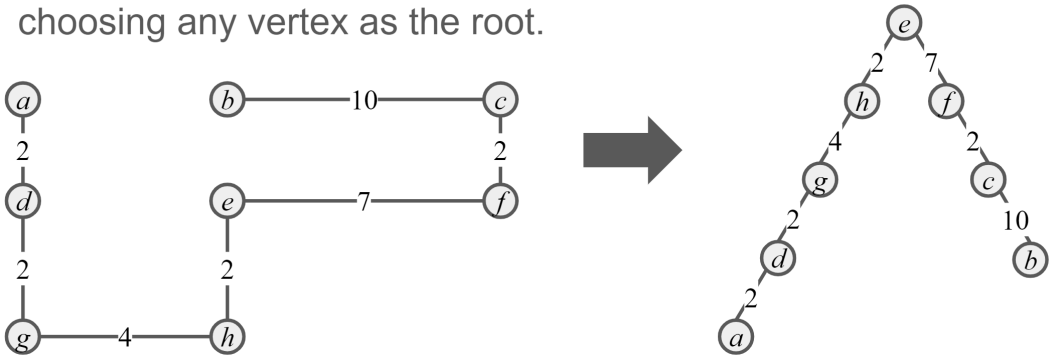


$\text{LCA}(1, 2) = 9$
$\text{LCA}(1, 9) = 9$
$\text{LCA}(2, 10) = 13$
$\text{LCA}(9, 5) = 15$
$\text{LCA}(1, 15) = 15$

An important property/advantage of trees over undirected graphs

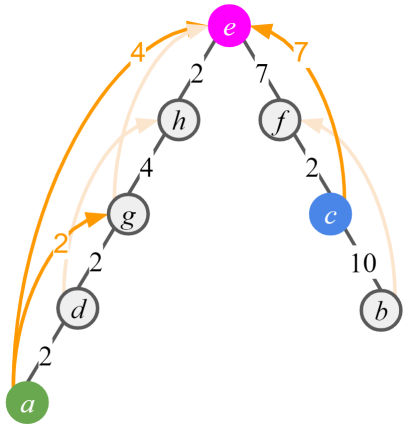
- Every node in the tree has a unique path to every other node.
- We can therefore pre-process the width along query paths

We can draw the MST as a rooted tree by choosing any vertex as the root.



Pre-process

- `width(u, v)` : width of the path from descendant u upwards to ancestor v in the tree
- pre-compute the widths of vertex-to-ancestor paths (refer to as lineage)
  - Optimisation with skip pointers instead of pointing from one to every other (S(n) = O(V^2)):
    - Each node in the tree stores the pointers to ancestors $2 = 2^1$ hops up, $4 = 2^2$ hops up, $8 = 2^3$ hops up and so on
    - `u.ancestor(2^k)` : the ancestor 2^k hops up, k>= 0, when k = 0 → parent
    - Each node also stores the pre-processed widths along the lineages to these ancestors
    - `u.lineageWidth(2^k)` : access the pre-processed width of the lineage to the ancestor 2^k hops up, k = 0 -> edge weight to parent
    - Each node stores O(log V) path widths and pointers ⇒ S(n) = O(V log V)



Query:

- Find LCA

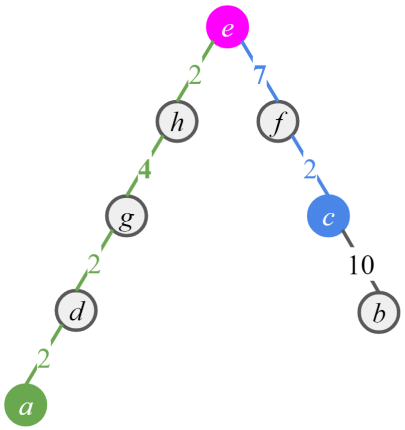
```
1   getAncestor(u, hops) // T(n) = O(log V)
2       currNode ← u
3       hopsRemaining ← hops
4
5       while hopsRemaining > 0 do
6           kMost ← ⌊log2 hopsRemaining⌋
7           currNode ← currNode.ancestor(2^(kMost))
8           hopsRemaining ← hopsRemaining − 2^(kMost)
9       end
10
11      return currNode
12
13  isAncestor(currNode, v) // O(log V)
```

```
    checkHops ← currNode.depth−v.depth

    if checkHops < 0 then
        return false
    end

    vAncestor ← getAncestor(v, checkHops)

    return currNode=vAncestor

findLCA(u, v) // returns the num of hops to get to the LCA; O(log^2 V)
    lo ← 1
    hi ← u.depth
    while lo < hi do
        mid ← lo+⌊(lo+hi)/2⌋
        ancestor ← getAncestor(u, mid)
        if isAncestor(ancestor, v) then // O(log V) per isAncestor call
            hi ← mid−1      // Search LHS
        else
            lo ← mid+1      // Search RHS
        end
    end
    return hops
```

- Use pre-computed lineage widths to obtain `width(S, L)` and `width(D, L)`
  - `getWidth(S, hops)` : hops refer to the number of edge-hops to go from S to the LCA
  - Start at current node S with the current maximum width = 0
  - Skip *as high up* to LCA as possible using the pre-processed exponentially-incremented pointers to ancestors (by updating current node to the ancestor node hopped to)
  - For each hop, update the maximum width if the width due to the hop is greater than current maximum
  - When we have exhausted all the hops (i.e. currently at LCA node), we can read off the maximum recorded width as the vertex-to-LCA width

```
getWidth(u, hops) // T(n) = O(log V)
    return getWidthHelper(u,hops,0)

getWidthHelper(currNode, hopsRemaining, currMax)
    if hopsRemaining=0 then
        return currMax
    else
        kMost ← ⌊log2 hopsRemaining⌋
        return getWidthHelper(
            currNode.ancestor(2^(kMost)),
            hopsRemaining−2^(kMost),
            max(currNode.lineageWidth(2^(kMost)), currMax))
    end
```

- minimax distance = `max(width(S, L), width(D, L))` where `L = LCA(S, D)`



So if we were to query for the maximum edge between a and c, the query problem now becomes finding:

$$\max(\text{width}(a, e), \text{width}(c, e))$$

$$\max(4, 7) = 4$$

Solution	Time Complexity		Space Complexity		
	Preprocess	Query	Preprocess	Query	DS
Binary search on query-time trimmed graph	0	$O((V+E)\log k)$	0	$O(V)$	N/A
Binary search on preprocessed trimmed graphs	$O(k(V+E))$	$O(\log k)$	$O(kV)$	$O(1)$	$O(kV)$
Modified relax SSSP with Dijkstra's	0	$O((V+E)\log V)$	0	$O(V)$	N/A
Modified relax APSP with Dijkstra's	$O(V(V+E)\log V)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Modified relax APSP with Floyd-Warshall's	$O(V^3)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Query-time MST†	0	$O(E\log V)$	0	$O(V)$	N/A
Preprocessed MST†	$O(E\log V)$	$O(V)$	$O(V)$	$O(V)$	$O(V)$
Preprocessed MST† + Binary search + LCA	$O(E\log V+V^2)$	$O(\log^2 V)$	$O(V\log V)$	$O(1)$	$O(V\log V)$

†: Implemented using Prim's with binary heap