Section A

1. (a) Analyze the performance of the following operations of the List data structure as specified by the Standard Template Library of C++: push_front(e), push_back(e), pop_front( ), and pop_back( ).Under what circumstances would you prefer a std::list over a std::vector?

- **Performance of std::list operations:**
  - *push_front(e)*: $O(1)$ constant time. A new node is created and linked to the beginning of the list.
  - *push_back(e)*: $O(1)$ constant time. A new node is created and linked to the end of the list.
  - *pop_front()*: $O(1)$ constant time. The first node is unlinked and deleted.
  - *pop_back()*: $O(1)$ constant time. The last node is unlinked and deleted.
- **Circumstances to prefer std::list over std::vector:**
  - Frequent insertions or deletions at the beginning or middle of the sequence. *std::vector* would require shifting elements, leading to $O(N)$ time complexity for these operations.
  - When the exact size of the sequence is not known beforehand or is highly dynamic, as *std::list* does not incur reallocation costs like *std::vector*.
  - When memory fragmentation is not a major concern, as *std::list* nodes are not contiguous in memory.

1. (b) Explain Kruskal's Algorithm for finding the Minimum Spanning Tree (MST) in an undirected weighted graph.Discuss how the Union-Find data structure is used by Kruskal's algorithm to efficiently detect cycles during edge addition.

- **Kruskal's Algorithm for MST:**
  - Sort all edges in the graph in non-decreasing order of their weights.
  - Initialize an empty set for the MST.
  - Iterate through the sorted edges. For each edge (u, v) with weight w:
    - If adding the edge (u, v) to the MST does not form a cycle, add it to the MST.
    - If the MST has $V - 1$ edges (where V is the number of vertices), stop.
- **Role of Union-Find in Kruskal's Algorithm:**
  - The Union-Find data structure is used to efficiently keep track of the connected components of the graph and detect cycles.
  - Initially, each vertex is in its own disjoint set.
  - When considering an edge (u, v), Kruskal's algorithm performs a *Find* operation on both u and v to determine their representative elements (i.e., which connected component they belong to).
  - If *Find(u)* and *Find(v)* return different representatives, it means u and v are in different connected components. Adding the edge (u, v) will not form a cycle, so the edge is added to the

MST, and a *Union* operation is performed on the sets containing u and v to merge them.

- If *Find(u)* and *Find(v)* return the same representative, it means u and v are already in the same connected component. Adding the edge (u, v) would create a cycle, so the edge is discarded. This cycle detection is efficient, typically with nearly constant amortized time per operation when path compression and union by rank/size are used.

1. (c) Consider the following array: A =.Compare the performance of Quicksort and Randomized Quicksort for sorting the given array, A. Explain the impact of pivot selection on the partitioning process for both the algorithms and how it influences the overall time complexity.

- **Array A:**
- **Comparison of Quicksort and Randomized Quicksort performance for array A:**
  - **Quicksort (deterministic pivot selection, e.g., first element):** If the first element (which is 1) is chosen as the pivot, the array will be partitioned into: *[1]* and *[5, 2, 7, 3, 8, 4, 6]*. This is a highly unbalanced partition. In the next step, for *[5, 2, 7, 3, 8, 4, 6]*, if 5 is chosen as the pivot, it will again lead to unbalanced partitions. This scenario exemplifies the worst-case performance for Quicksort when the input array is already sorted or nearly sorted, leading to $O(N^2)$ time complexity.
  - **Randomized Quicksort:** In Randomized Quicksort, the pivot is chosen randomly from the array. This significantly reduces the

probability of consistently hitting the worst-case pivot (smallest or largest element) in successive partitions. For array A, a random pivot selection would likely lead to more balanced partitions on average. For example, if 4 or 5 is randomly chosen as a pivot, the partitions would be more even, leading to performance closer to the average case $O(N \\log N)$ time complexity.

- **Impact of pivot selection on partitioning process and overall time complexity:**
  - **Impact on Partitioning:** The chosen pivot determines how the array is divided into two sub-arrays: elements less than the pivot and elements greater than the pivot.
    - **Good Pivot (close to median):** Leads to balanced partitions, where both sub-arrays are approximately half the size of the original array. This results in a balanced recursion tree.
    - **Bad Pivot (smallest or largest element):** Leads to highly unbalanced partitions, where one sub-array is empty or contains only one element, and the other contains almost all remaining elements. This results in a skewed recursion tree.
  - **Influence on Overall Time Complexity:**
    - **Balanced Partitions:** When partitions are balanced, the recursion depth is $O(\\log N)$, and at each level, $O(N)$ work is done (for partitioning). This yields an overall time

complexity of $O(N \\log N)$, which is the best-case and average-case performance for Quicksort.

- **Unbalanced Partitions:** When partitions are consistently unbalanced, the recursion depth can be $O(N)$. In this scenario, the total work becomes $O(N^2)$, which is the worst-case performance for Quicksort.
- Randomized pivot selection in Randomized Quicksort makes the worst-case scenario highly improbable, ensuring that the average-case $O(N \\log N)$ performance is achieved with high probability, regardless of the initial arrangement of elements in the array.

1. (d) Given a residual graph, G with a flow, f and v(f) be the value of flow. The function bottleneck(p,f) for an augmenting path P is the minimum residual capacity with respect to the flow f. Argue that the bottleneck(P,f) always greater than zero.If the v(f') is equal to the v(f) + bottleneck(P,f), Prove that v(f') is greater than v(f) and the capacity condition holds for f'.

- **Argument that bottleneck(P,f) is always greater than zero:**
  - An augmenting path P in a residual graph G_f is a path from the source s to the sink t consisting only of edges with positive residual capacity.
  - By definition, the residual capacity $c\_f(u,v)$ for any edge (u,v) in an augmenting path P must be strictly greater than zero.

- The *bottleneck(P,f)* is defined as the minimum residual capacity along this path. Since all edges on the path have positive residual capacity, their minimum must also be positive.
- Therefore, *bottleneck(P,f)* is always greater than zero.

- **Proof that v(f') > v(f) and capacity condition holds for f':**
  - Let f be an existing flow and P be an augmenting path with bottleneck capacity $\\delta = \\text{bottleneck}(P,f)$.
  - We define a new flow f' as $f'(u, v) = f(u, v) + \\delta$ for forward edges $(u, v) \\in P$, and $f'(u, v) = f(u, v) - \\delta$ for backward edges $(u, v) \\in P$. For edges not on P, $f'(u,v) = f(u,v)$.
  - **Proof that** $v(f')\ v(f)$**:**
    - The value of a flow is defined as the net flow out of the source s: $v(f) = \\sum\_{(s, u) \\in E} f(s, u) - \\sum\_{(u, s) \\in E} f(u, s)$.
    - When we augment the flow by $\\delta$ along path P, the flow on all forward edges in P increases by $\\delta$, and the flow on all backward edges in P decreases by $\\delta$.
    - Specifically, the flow out of the source s along the first edge of the augmenting path P increases by $\\delta$. Since $\\delta \> 0$, the total flow out of s increases, and thus $v(f') = v(f) + \\delta$.
    - Since $\\delta \> 0$, it follows that $v(f')\ v(f)$.
  - **Proof that capacity condition holds for f':**

- The capacity condition states that for every edge $(u, v) \\in E$, $0 \\le f'(u, v) \\le c(u, v)$.
- Consider an edge (u, v) in the original graph G:
    - **Case 1: (u, v) is a forward edge in P.**
        - In the residual graph, $c\_f(u, v) = c(u, v) - f(u, v) \\ge \\delta$.
        - So, $f(u, v) \\le c(u, v) - \\delta$.
        - The new flow is $f'(u, v) = f(u, v) + \\delta$.
        - Since $f(u, v) \\ge 0$, $f'(u, v) \\ge \\delta \> 0$.
        - Also, $f'(u, v) = f(u, v) + \\delta \\le (c(u, v) - \\delta) + \\delta = c(u, v)$.
        - Thus, $0 \\le f'(u, v) \\le c(u, v)$ holds.
    - **Case 2: (u, v) is a backward edge in P.** This means (v, u) is a forward edge in the original graph, and its flow is being reduced.
        - In the residual graph, $c\_f(u, v) = f(v, u) \\ge \\delta$.
        - This implies $f(v, u) \\ge \\delta$.
        - The new flow for (v, u) is $f'(v, u) = f(v, u) - \\delta$.
        - Since $f(v, u) \\ge \\delta$, $f'(v, u) \\ge 0$.
        - Also, $f'(v, u) = f(v, u) - \\delta \\le f(v, u) \\le c(v, u)$ (since $f(v, u)$ must satisfy capacity constraints in the first place).

- Thus, \$0 \\\\le f'(v, u) \\\\le c(v, u)\$ holds.
- **Case 3:** (u, v) **is not on** P.
  - $f'(u, v) = f(u, v)$, and since f satisfies the capacity condition, f' also satisfies it.
- Therefore, the capacity condition holds for f'.

1. (e) Given the sequence of probes generated by quadratic probing for keys, K p(i) = h(K) + $(-1)^{i-1}((i + 1)/2)^2$ for i = 1, 2, ..., TSize – 1 i.e. h(K), h(K)+1, h(K)-1, h(K)+4, h(K)-4,....What happens if the table size, TSize is not prime?.Can quadratic probing guarantee the use of all positions in the table for a non-prime table size?

- **What happens if the table size, TSize is not prime?**
  - If the table size *TSize* is not prime, and especially if it is a composite number, quadratic probing can lead to a situation where only a limited subset of table positions are probed.
  - Specifically, if *TSize* is a power of 2, or has many small factors, the sequence of probes *h(K) + i^2 (mod TSize)* or *h(K) - i^2 (mod TSize)* will repeat values and will not explore all available slots. This significantly increases the chance of secondary clustering, where multiple keys hash to the same initial slot and then follow the same probe sequence.
- **Can quadratic probing guarantee the use of all positions in the table for a non-prime table size?**
  - No, quadratic probing cannot guarantee the use of all positions in the table for a non-prime table size.

  - Quadratic probing guarantees that at least half of the table will be probed if the table size *TSize* is a prime number. If *TSize* is not prime, particularly if it's a composite number with common factors with the probe sequence increments ($1^2, 2^2, 3^2, \\dots$), the number of distinct probe locations can be significantly less than *TSize*. This can lead to the hash table filling up even when there are empty slots available, as the algorithm might enter a cycle of previously probed locations.

1. (f) Given that in a vh-tree, the shortest path to a leaf consists only of vertical links, and the longest path to another leaf may use both vertical and horizontal links alternately, derive the relationship between the longest path, path_longest and the shortest path, path_shortest. Given a vh-tree with height h, derive the minimum number of nodes n_min that the tree can have.

- **Relationship between path_longest and path_shortest:**
  - Let h be the height of the vh-tree.
  - The shortest path to a leaf consists only of vertical links. Since the height is h, this means there are h vertical links.
  - Therefore, $path\_shortest = h$.
  - The longest path to a leaf uses both vertical and horizontal links alternately. To maximize the path length, we would want to alternate between vertical and horizontal links as much as possible.

- Starting from the root, a vertical link takes us to a child. From that child, a horizontal link takes us to a sibling. From that sibling, another vertical link takes us down, and so on.
- Each vertical link takes us down one level. Each horizontal link stays at the same level.
- Consider a path where we make a vertical move, then a horizontal move, then a vertical move, then a horizontal move, and so on.
- If the height of the tree is h, there will be h vertical links.
- Between any two vertical links (or after the first vertical link and before the last vertical link), there can be a horizontal link.
- To maximize the path, we can have $h - 1$ horizontal links (one at each level from 1 to $h - 1$, immediately after a vertical descent, before the next vertical descent).
- So, $path\_longest = h$ (for vertical links) $+(h - 1)$ (for horizontal links) $= 2h - 1$.
- Therefore, the relationship is \$path\_longest = 2 \\\\times path\_shortest - 1\$.

- **Minimum number of nodes n_min for a vh-tree with height h:**
  - To minimize the number of nodes, we want each node to have the minimum possible number of children and siblings.
  - A vh-tree is a binary tree where each node can have a vertical child and a horizontal sibling.
  - For a tree of height h, there must be at least one path of length h consisting only of vertical links. This means there are $h + 1$ nodes along this path (root + h vertical links).

- To minimize the total number of nodes, we would construct a "stick-like" tree where each node has only one vertical child (to maintain height h) and no horizontal siblings except possibly for the root's first child if it leads to the longest path, but for minimum nodes, we aim for minimal branching.
- Consider the minimum structure required to achieve height h. This would be a single path of h vertical links.
- The root is at level 0. Its vertical child is at level 1, and so on, until the leaf at level h.
- This minimum structure would have $h + 1$ nodes (root, and h subsequent nodes along the single vertical path).
- Thus, $n\_min = h + 1$.

1. (g) What is inverted index (or inverted file) with respect to search engines.

- **Inverted Index (or Inverted File) in Search Engines:**
  - An inverted index is a data structure used by search engines to map content (like words or phrases) to their locations within a document or a set of documents.
  - Instead of mapping documents to keywords (which is a forward index), an inverted index maps keywords to documents (and often specific locations within those documents).
  - **Structure:** It typically consists of two main parts:
    - **Vocabulary (or Lexicon):** A sorted list of all unique words (terms) that appear in the documents.

        - **Posting List (or Postings):** For each term in the vocabulary, there is a list of all documents (document IDs) in which that term appears, along with the positions (and sometimes frequency) of the term within those documents.
    - **How it works:** When a user enters a search query, the search engine looks up each term in the query in the inverted index. It then retrieves the posting lists for those terms and performs operations (like intersection or union) on these lists to find documents that contain all or some of the query terms. The positional information allows for phrase searching.
    - **Advantage:** It allows for very fast retrieval of documents containing specific keywords, as it directly provides the mapping from term to document. This is crucial for the efficiency of modern search engines.

Section B

2. (a) Given a text string T="ABABBDABACDABABCA BAB" and a pattern string P="ABABCABAB", perform the Knuth-Morris-Pratt (KMP) string matching algorithm to find first occurrence of P in T. Show the steps, including the construction of the failure table.Compare the Brute Force and KMP algorithms in terms of time complexity.

- **Text string T:** "ABABBDABACDABABCA BAB"
- **Pattern string P:** "ABABCABAB"

- **Construction of the Failure Table (LPS Array) for P="ABABCABAB":** The failure table (also known as the longest proper prefix suffix or LPS array) stores, for each position *i* in the pattern, the length of the longest proper prefix of *P[0...i]* that is also a suffix of *P[0...i]*.

Character	A	B	A	B	C	A	B	A	B
Index	0	1	2	3	4	5	6	7	8
LPS Array	0	0	1	2	0	1	2	3	4

  - *LPS[0] = 0* (always)
  - *P[1] = 'B'*, *P[0] = 'A'*. No match. *LPS[1] = 0*.
  - *P[2] = 'A'*, *P[0] = 'A'*. Match. *LPS[2] = 1*.
  - *P[3] = 'B'*, *P[1] = 'B'*. Match. *LPS[3] = LPS[2] + 1 = 2*. (Prefix "AB", suffix "AB")
  - *P[4] = 'C'*, *P[2] = 'A'*. No match. *LPS[4] = 0*.
  - *P[5] = 'A'*, *P[0] = 'A'*. Match. *LPS[5] = 1*.
  - *P[6] = 'B'*, *P[1] = 'B'*. Match. *LPS[6] = LPS[5] + 1 = 2*. (Prefix "AB", suffix "AB")
  - *P[7] = 'A'*, *P[2] = 'A'*. Match. *LPS[7] = LPS[6] + 1 = 3*. (Prefix "ABA", suffix "ABA")
  - *P[8] = 'B'*, *P[3] = 'B'*. Match. *LPS[8] = LPS[7] + 1 = 4*. (Prefix "ABAB", suffix "ABAB")

- **KMP String Matching Algorithm Steps:**

Initialize *i = 0* (for text) and *j = 0* (for pattern).

Step	Text (T)	Pattern (P)	*i*	*j*	Match/Mismatch	Action/ LPS value
1	**A**BABBDA...	**A**BABCABAB	0	0	Match	*i*++, *j*++
2	A**B**ABBDA...	A**B**ABCABAB	1	1	Match	*i*++, *j*++
3	AB**A**BBD...	AB**A**BCABAB	2	2	Match	*i*++, *j*++
4	ABA**B**BD...	ABA**B**CABAB	3	3	Match	*i*++, *j*++
5	ABAB**B**D...	ABAB**C**ABAB	4	4	Mismatch	*j = LPS[j-1] = LPS[3] = 2*
6	ABAB**B**D...	AB**A**BCABAB	4	2	Mismatch	*j = LPS[j-1] = LPS[1] = 0*
7	ABAB**B**D...	**A**BABCABAB	4	0	Mismatch	*i*++ (no *j* change as *j* is

Step	Text (T)	Pattern (P)	*i*	*j*	Match/Mismatch	Action/ LPS value
						already 0)
8	ABABB**D**A...	A**B**ABCA BAB	5	0	Mismatch	*i*++ (no *j* change as *j* is already 0)
9	ABABBD**A**B...	**A**BABCA BAB	6	0	Match	*i*++, *j*++
10	ABABBDA**B**A...	A**B**ABCA BAB	7	1	Match	*i*++, *j*++
11	ABABBDAB**A**C...	AB**A**BCA BAB	8	2	Match	*i*++, *j*++
12	ABABBDABA**C**D...	ABA**B**CA BAB	9	3	Mismatch	*j = LPS[j-1] = LPS[2] = 1*
13	ABABBDABA**C**D...	A**B**ABCA BAB	9	1	Mismatch	*j = LPS[j-1] =*

Step	Text (T)	Pattern (P)	*i*	*j*	Match/Mismatch	Action/ LPS value
						LPS[0] = 0
14	ABABBDABA**C**D...	**A**BABCABAB	9	0	Mismatch	*i*++ (no *j* change as *j* is already 0)
15	ABABBDABAC**D**AB...	A**B**ABCABAB	10	0	Mismatch	*i*++ (no *j* change as *j* is already 0)
16	ABABBDABACD**A**BAB...	**A**BABCABAB	11	0	Match	*i*++, *j*++
17	ABABBDABACDA**B**AB...	A**B**ABCABAB	12	1	Match	*i*++, *j*++
18	ABABBDABACDAB**A**B...	AB**A**BCABAB	13	2	Match	*i*++, *j*++
19	ABABBDABACDABA**B**CA...	ABA**B**CABAB	14	3	Match	*i*++, *j*++

Step	Text (T)	Pattern (P)	*i*	*j*	Match/Mismatch	Action/ LPS value
20	ABABBDABACDABABB**C**A...	ABAB**C**ABAB	15	4	Match	*i*++, *j*++
21	ABABBDABACDABABC**A**B	ABABC**A**BAB	16	5	Match	*i*++, *j*++
22	ABABBDABACDABABCD**A**B	ABABCA**B**AB	17	6	Match	*i*++, *j*++
23	ABABBDABACDABABCDA**B**	ABABCAB**A**B	18	7	Match	*i*++, *j*++
24	ABABBDABACDABABCDAB**A**B	ABABCABA**B**	19	8	Match	*i*++, *j*++
25	ABABBDABACDABABCDABA**B**	ABABCABAB** (end of P)**	20	9	Match (Pattern found!)	*j == length(P)*

The pattern "ABABCABAB" is found starting at index 11 in the text T (i.e., T[11...19]).

- **Comparison of Brute Force and KMP algorithms in terms of time complexity:**
  - **Brute Force Algorithm:**
    - **Time Complexity:** In the worst-case scenario (e.g., text *AAAAA...A* and pattern *AAAB*), the brute force algorithm

can take $O(MN)$ time, where M is the length of the text and N is the length of the pattern. This is because for each character in the text, it might perform up to N comparisons.

- **KMP Algorithm:**
  - **Time Complexity:** The KMP algorithm has a time complexity of $O(M + N)$, where M is the length of the text and N is the length of the pattern.
    - Building the LPS (failure) table takes $O(N)$ time.
    - The searching phase takes $O(M)$ time because *i* (text pointer) always advances, and *j* (pattern pointer) only moves forward or jumps back using the LPS table, but it never goes backward past previously matched characters. In total, the number of comparisons is at most $2M$.
- **Advantage of KMP:** KMP is significantly more efficient than Brute Force, especially for larger texts and patterns with repeating sub-patterns, as it avoids redundant comparisons by intelligently skipping characters based on the precomputed failure table.

2. (b) Consider the following weighted directed graph with 4 nodes (A, B, C, D) with the following edges and weights:.

- A → B (weight 3)
- A → C (weight -2)

- B → D (weight 2)
- C → B (weight -1)
- D → A (weight 1) Use the Bellman-Ford algorithm to find the shortest path from vertex A to D.Can we find the shortest path in a graph, if it contains a negative cycle? Support your answer with proper arguments.
- **Graph Edges and Weights:**
  - A $\\xrightarrow{3}$ B
  - A $\\xrightarrow{-2}$ C
  - B $\\xrightarrow{2}$ D
  - C $\\xrightarrow{-1}$ B
  - D $\\xrightarrow{1}$ A

- **Bellman-Ford Algorithm to find the shortest path from vertex A to D:**

  Initialize distances: $dist[A] = 0$ $dist[B] = \\infty$ $dist[C] = \\infty$ $dist[D] = \\infty$

  Number of vertices $V = 4$. We need to relax edges $V - 1 = 3$ times.

  Pass 1 (V=3 relaxations):

  - Relax A $\\xrightarrow{3}$ B: $dist[B] = \\min(\\infty, dist[A] + 3) = \\min(\\infty, 0 + 3) = 3$
  - Relax A $\\xrightarrow{-2}$ C: $dist[C] = \\min(\\infty, dist[A] + (-2)) = \\min(\\infty, 0 - 2) = -2$

- Relax B $\\xrightarrow{2}$ D: $dist[D] = \\min(\\infty, dist[B] + 2) = \\min(\\infty, 3 + 2) = 5$
- Relax C $\\xrightarrow{-1}$ B: $dist[B] = \\min(3, dist[C] + (-1)) = \\min(3, -2 - 1) = -3$ (Update!)
- Relax D $\\xrightarrow{1}$ A: $dist[A] = \\min(0, dist[D] + 1) = \\min(0, 5 + 1) = 0$ (No change, as 0 is already minimum)

Distances after Pass 1: $dist[A] = 0$ $dist[B] = -3$ $dist[C] = -2$ $dist[D] = 5$

Pass 2:

- Relax A $\\xrightarrow{3}$ B: $dist[B] = \\min(-3, dist[A] + 3) = \\min(-3, 0 + 3) = -3$ (No change)
- Relax A $\\xrightarrow{-2}$ C: $dist[C] = \\min(-2, dist[A] + (-2)) = \\min(-2, 0 - 2) = -2$ (No change)
- Relax B $\\xrightarrow{2}$ D: $dist[D] = \\min(5, dist[B] + 2) = \\min(5, -3 + 2) = -1$ (Update!)
- Relax C $\\xrightarrow{-1}$ B: $dist[B] = \\min(-3, dist[C] + (-1)) = \\min(-3, -2 - 1) = -3$ (No change)
- Relax D $\\xrightarrow{1}$ A: $dist[A] = \\min(0, dist[D] + 1) = \\min(0, -1 + 1) = 0$ (No change)

Distances after Pass 2: $dist[A] = 0$ $dist[B] = -3$ $dist[C] = -2$ $dist[D] = -1$

Pass 3:

- Relax A $\\xrightarrow{3}$ B: $dist[B] = \\min(-3, dist[A] + 3) = \\min(-3, 0 + 3) = -3$ (No change)

- Relax A $\\xrightarrow{-2}$ C: $dist[C] = \\min(-2, dist[A] + (-2)) = \\min(-2, 0 - 2) = -2$ (No change)
- Relax B $\\xrightarrow{2}$ D: $dist[D] = \\min(-1, dist[B] + 2) = \\min(-1, -3 + 2) = -1$ (No change)
- Relax C $\\xrightarrow{-1}$ B: $dist[B] = \\min(-3, dist[C] + (-1)) = \\min(-3, -2 - 1) = -3$ (No change)
- Relax D $\\xrightarrow{1}$ A: $dist[A] = \\min(0, dist[D] + 1) = \\min(0, -1 + 1) = 0$ (No change)

Distances after Pass 3: $dist[A] = 0$ $dist[B] = -3$ $dist[C] = -2$ $dist[D] = -1$

Since no distances were updated in Pass 3, the algorithm terminates. (Alternatively, run one more pass to check for negative cycles).

The shortest path distance from A to D is $dist[D] = -1$. The shortest path is A $\\to$ C $\\to$ B $\\to$ D, with a total weight of $-2 + (-1) + 2 = -1$.

- **Can we find the shortest path in a graph, if it contains a negative cycle? Support your answer with proper arguments.**
  - No, we cannot find a meaningful "shortest path" in a graph if it contains a negative cycle reachable from the source vertex.
  - **Argument:**
    - If there is a negative cycle reachable from the source, say a cycle C with a total weight of $W\_C \< 0$.
    - If we traverse this cycle once, the path length decreases by W_C.

    - If we traverse it twice, the path length decreases by $2W\_C$, and so on.
    - We can traverse the negative cycle an infinite number of times, making the path length infinitely small (i.e., approaching $-\\infty$).
    - Therefore, the concept of a "shortest path" becomes undefined, as there is no finite minimum value for the path length.
  - **Bellman-Ford's behavior:** The Bellman-Ford algorithm can detect the presence of a negative cycle. After $V - 1$ relaxations, if a V-th relaxation pass yields any further distance updates, it indicates the presence of a negative cycle. The algorithm will then report that a negative cycle exists and that shortest paths are not well-defined. It does not provide a finite shortest path in such cases.

3. (a) Consider a hash table of size 8, with the following hash function: H(k) = (k mod8).Insert the following set of keys : {14, 30, 22, 10, 50, 60, 90} using the given hash function and for collision resolution use linear probing.Discuss the advantages and disadvantages of linear probing and quadratic probing in terms of collision handling and performance.

- **Hash Table of size 8, H(k) = (k mod 8), using Linear Probing:**

  Initial Hash Table (size 8): *[_, _, _, _, _, _, _, _]* (Indices 0 to 7)

  a. **Insert 14:**

- H(14) = 14 mod 8 = 6
- Table[6] is empty.
- *[_, _, _, _, _, _, 14, _]*

b. **Insert 30:**

- H(30) = 30 mod 8 = 6
- Table[6] is occupied (14).
- Linear probe: Check (6+1) mod 8 = 7. Table[7] is empty.
- *[_, _, _, _, _, _, 14, 30]*

c. **Insert 22:**

- H(22) = 22 mod 8 = 6
- Table[6] is occupied (14).
- Linear probe: Check (6+1) mod 8 = 7. Table[7] is occupied (30).
- Linear probe: Check (7+1) mod 8 = 0. Table[0] is empty.
- *[22, _, _, _, _, _, 14, 30]*

d. **Insert 10:**

- H(10) = 10 mod 8 = 2
- Table[2] is empty.
- *[22, _, 10, _, _, _, 14, 30]*

e. **Insert 50:**

- H(50) = 50 mod 8 = 2
- Table[2] is occupied (10).
- Linear probe: Check (2+1) mod 8 = 3. Table[3] is empty.
- *[22, _, 10, 50, _, _, 14, 30]*

f. **Insert 60:**

- H(60) = 60 mod 8 = 4
- Table[4] is empty.
- *[22, _, 10, 50, 60, _, 14, 30]*

g. **Insert 90:**

- H(90) = 90 mod 8 = 2
- Table[2] is occupied (10).
- Linear probe: Check (2+1) mod 8 = 3. Table[3] is occupied (50).
- Linear probe: Check (3+1) mod 8 = 4. Table[4] is occupied (60).
- Linear probe: Check (4+1) mod 8 = 5. Table[5] is empty.
- *[22, _, 10, 50, 60, 90, 14, 30]*

Final Hash Table: Index: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 Value: 22 | _ | 10 | 50 | 60 | 90 | 14 | 30

- **Advantages and Disadvantages of Linear Probing and Quadratic Probing:**

Linear Probing:

- **Advantages:**
  - **Simple to implement:** The probing sequence is straightforward: $h(k), (h(k) + 1)$.
  - **Good Cache Performance (Locality of Reference):** Subsequent probes access memory locations that are

close to each other, often improving cache hit rates. This can lead to faster access times in practice.

- **Disadvantages:**
  - **Primary Clustering:** This is the main drawback. Long runs of occupied slots are formed. When a collision occurs, a new key probes sequentially until an empty slot is found. This tends to make existing clusters grow larger and larger, increasing the average search time.
  - **High Collision Rate:** As clusters grow, more and more insertions will inevitably hash into these clusters, leading to more collisions and longer probe sequences.

Quadratic Probing:

- **Advantages:**
  - **Reduces Primary Clustering:** By using a quadratic increment ($h(k) \\pm i^2$), it spreads out the probe sequence, avoiding the formation of long contiguous clusters. This helps in reducing the impact of primary clustering.
  - **Better Performance than Linear Probing (in many cases):** Due to reduced clustering, it generally performs better than linear probing, especially as the load factor increases.
- **Disadvantages:**
  - **Secondary Clustering:** While it avoids primary clustering, it can suffer from secondary clustering. If two distinct keys hash to the same initial slot $h(k)$, they will

follow the *exact same* quadratic probe sequence. This can lead to performance degradation if many keys map to the same initial hash value.

- **Doesn't Guarantee Probing All Slots:** If the table size is not a prime number, or not specifically a prime number of the form $4k + 3$, quadratic probing might not probe all available slots in the table. This means the table could appear full even if empty slots exist, leading to insertion failures or infinite loops if not handled carefully.
- **Worse Cache Performance (Locality of Reference):** The jumps in the probe sequence are larger and less predictable than linear probing, which can lead to poorer cache performance due to less spatial locality.

3. (b) Does the splitting of a node at the time of insertion always increase the height of the tree?.Construct a B-tree of order 5 which is initially empty. Insert the following keys into the B-tree: 7,15,11,25,12,10,22,30,20,5,27,30,18. After each insertion, show the structure of the B-tree. What is the height of the final tree?

- **Does the splitting of a node at the time of insertion always increase the height of the tree?**
  - No, the splitting of a node at the time of insertion does not always increase the height of the B-tree.
  - A split increases the height of the tree only when the root node itself splits. When an internal (non-root) node splits, a new median key is propagated up to its parent. If the parent has

space, it absorbs the key, and the height remains unchanged. If the parent is also full, it might split, and this process can propagate upwards. Only when this propagation reaches the root and causes the root to split does the height of the entire tree increase.

- **Construction of a B-tree of order 5 (maximum 4 keys per node):** A B-tree of order 5 means each node can have a maximum of $m = 5$ children and a maximum of $m - 1 = 4$ keys. The minimum number of keys (except for the root) is $\\lceil m/2 \\rceil - 1 = \\lceil 5/2 \\rceil - 1 = 3 - 1 = 2$.

  Keys to insert: 7, 15, 11, 25, 12, 10, 22, 30, 20, 5, 27, 30, 18

  h. **Insert 7:**
     - [7]

  i. **Insert 15:**
     - [7, 15]

  j. **Insert 11:**
     - [7, 11, 15]

  k. **Insert 25:**
     - [7, 11, 15, 25]

  l. **Insert 12:** (Node is full [7, 11, 15, 25], insert 12. Sorted: [7, 11, 12, 15, 25]. Median is 12. 12 moves up, others split)
     - Root: [12]
     - Children: [7, 11] and [15, 25]

m. **Insert 10:** (Goes into [7, 11])
   - Root: [12]
   - Children: [7, 10, 11] and [15, 25]

n. **Insert 22:** (Goes into [15, 25])
   - Root: [12]
   - Children: [7, 10, 11] and [15, 22, 25]

o. **Insert 30:** (Goes into [15, 22, 25])
   - Root: [12]
   - Children: [7, 10, 11] and [15, 22, 25, 30]

p. **Insert 20:** (Goes into [15, 22, 25, 30]. Node is full. Sorted: [15, 20, 22, 25, 30]. Median is 22. 22 moves up to parent [12])
   - Root: [12, 22]
   - Children: [7, 10, 11], [15, 20], and [25, 30]

q. **Insert 5:** (Goes into [7, 10, 11])
   - Root: [12, 22]
   - Children: [5, 7, 10, 11], [15, 20], and [25, 30]

r. **Insert 27:** (Goes into [25, 30])
   - Root: [12, 22]
   - Children: [5, 7, 10, 11], [15, 20], and [25, 27, 30]

s. **Insert 30:** (This key 30 already exists. B-trees typically store unique keys. Assuming it's a duplicate and is ignored or it could be handled differently based on exact requirements. Assuming it is ignored for this specific construction to maintain tree properties.)

  - Tree remains unchanged. (If duplicates are allowed, it would depend on the rule, e.g., appended to the right child or a count increased. For simplicity, we assume unique keys or ignore duplicate insertion.)

t. **Insert 18:** (Goes into [15, 20])

  - Root: [12, 22]
  - Children: [5, 7, 10, 11], [15, 18, 20], and [25, 27, 30]

Final B-tree Structure:

Root:

[12, 22]
/ **|** \

Left Child (keys < 12): *[5, 7, 10, 11]* Middle Child (keys between 12 and 22): *[15, 18, 20]* Right Child (keys > 22): *[25, 27, 30]*

- **Height of the final tree:**
  - The height of the final tree is 1 (counting edges from root to leaf). The root is at level 0, and the leaf nodes are at level 1.

4. (a) Given the following set of words: ["cat", "bat", "rat", "at", "batman", "sat", "saturday", "rattle", "cattle", "ate", "battle", "satin"], Construct a standard trie to store these words.Also show the Compressed trie for the standard trie created above. What is the advantage of using a compressed trie?

- **Set of Words:** ["cat", "bat", "rat", "at", "batman", "sat", "saturday", "rattle", "cattle", "ate", "battle", "satin"]
- **Standard Trie Construction:** (Due to the constraint "Do not make diagrams or structures in future responses" from 2025-07-19, I will describe the structure textually).

  The trie will start with an empty root node. Each edge represents a character, and nodes mark the end of a word.

  - **Root (empty string):**
    - 'c' -> node1 (end of "c")
      - 'a' -> node2 (end of "ca")
        - 't' -> node3 (end of "cat" - WORD END)
    - 'b' -> node4 (end of "b")
      - 'a' -> node5 (end of "ba")
        - 't' -> node6 (end of "bat" - WORD END)
          - 'm' -> node7 (end of "batm")
            - 'a' -> node8 (end of "batma")
              - 'n' -> node9 (end of "batman" - WORD END)
          - 't' -> node10 (end of "batt")
            - 'l' -> node11 (end of "battl")
              - 'e' -> node12 (end of "battle" - WORD END)
    - 'r' -> node13 (end of "r")
      - 'a' -> node14 (end of "ra")

    - 't' -> node15 (end of "rat" - WORD END)
      - 't' -> node16 (end of "ratt")
        - 'l' -> node17 (end of "rattl")
          - 'e' -> node18 (end of "rattle" - WORD END)
- 'a' -> node19 (end of "a")
  - 't' -> node20 (end of "at" - WORD END)
    - 'e' -> node21 (end of "ate" - WORD END)
- 's' -> node22 (end of "s")
  - 'a' -> node23 (end of "sa")
    - 't' -> node24 (end of "sat" - WORD END)
      - 'u' -> node25 (end of "satu")
        - 'r' -> node26 (end of "satur")
          - 'd' -> node27 (end of "saturd")
            - 'a' -> node28 (end of "saturda")

'y' -> node29 (end of "saturday" - WORD END)

      - 'i' -> node30 (end of "sati")
        - 'n' -> node31 (end of "satin" - WORD END)
- 'c' -> node1 (already exists from "cat")
  - 'a' -> node2 (already exists)
    - 't' -> node3 (already exists)

            - 't' -> node32 (end of "catt") (New branch for "cattle")
                - 'l' -> node33 (end of "cattl")
                    - 'e' -> node34 (end of "cattle" - WORD END)

This textual representation shows the paths and where words end. Each character from the word corresponds to a node (or a series of nodes if there are no branches).

- **Compressed Trie Construction:** A compressed trie (or Patricia trie/radix tree) collapses chains of single-child nodes into a single edge labeled with a string.
    - **Root:**
        - 'at' -> node1 (end of "at" - WORD END)
            - 'e' -> node2 (end of "ate" - WORD END)
        - 'c' -> node3
            - 'at' -> node4 (end of "cat" - WORD END)
            - 'attle' -> node5 (end of "cattle" - WORD END)
        - 'b' -> node6
            - 'at' -> node7 (end of "bat" - WORD END)
                - 'man' -> node8 (end of "batman" - WORD END)
            - 'attle' -> node9 (end of "battle" - WORD END)
        - 'r' -> node10
            - 'at' -> node11 (end of "rat" - WORD END)

            - 'attle' -> node12 (end of "rattle" - WORD END)
        - 's' -> node13
            - 'at' -> node14 (end of "sat" - WORD END)
                - 'urday' -> node15 (end of "saturday" - WORD END)
            - 'atin' -> node16 (end of "satin" - WORD END)

- **Advantage of using a Compressed Trie:**
    - **Space Efficiency:** The primary advantage is significant space savings. In a standard trie, if there are long sequences of characters where a node has only one child, each character takes up a separate node. A compressed trie collapses these chains into a single edge labeled with the entire string segment, thereby reducing the number of nodes and pointers needed. This is particularly beneficial for sparse tries (tries with many long, unbranching paths).
    - **Faster Traversal (for certain operations):** While asymptotic time complexities for search and insertion remain similar to a standard trie (dependent on string length, not number of nodes traversed), in practice, fewer node traversals might be needed for a compressed trie because multiple characters are "processed" at once along a compressed edge.

4. (b) Consider the array implementation of the Union-Find data structure for set S of size n, where unions keep the name of the larger set. Prove the following :.

- (i) Find operation takes O(1) time,.
- (ii) MakeUnionFind(S) takes O(n) time, and.
- (iii) Sequence of k Union operations takes at most O(k log k) time.. Explain the role of the “name of the larger set” optimization in the Union-Find data structure. How does this optimization reduce the height of the tree created during the union process?
- **Array Implementation of Union-Find (using parent array and size array):**
  - Each element i in the set S has a *parent[i]* entry, which stores the parent of i. If i is a root of a set, *parent[i]* typically stores a negative value indicating the size of the set (e.g., *-size*).
  - *Find(i)*: To find the representative of element i, traverse up the parent pointers until a root (negative *parent* value) is found.
  - *Union(i, j)*: Find the roots of i and j. If they are different, make the root of the smaller set point to the root of the larger set. Update the size of the larger set.
- **(i) Find operation takes O(1) time:**
  - This statement is **incorrect** in a general array implementation of Union-Find, even with the "union by size" (or "union by rank") optimization. A single *Find* operation without path compression can take $O(\\log N)$ time in the worst case (if the tree becomes a path).
  - If path compression is also applied alongside union by size/rank, then the amortized time complexity for *Find* is nearly

constant, specifically $O(\\alpha(N))$, where $\\alpha$ is the inverse Ackermann function, which grows extremely slowly and is practically considered constant for all realistic input sizes.

- However, if the question implies a *direct* lookup (e.g., if each element directly stores its representative, which would be a flattened structure), then *Find* would be $O(1)$. But for a typical array-based Union-Find using parent pointers, it is not strictly $O(1)$ without a very strong form of path compression that flattens completely on every find, which would negate the tree structure.
- Assuming the intent is the theoretical best performance with full optimizations: with both path compression and union by size/rank, the amortized time is $O(\\alpha(N))$, which is practically $O(1)$. Without path compression, it's $O(\\log N)$. Given the context of the other parts, it's likely implying the optimized version.

- **(ii) MakeUnionFind(S) takes O(n) time:**
  - To initialize the Union-Find data structure for a set S of size n, we need to create n disjoint sets, one for each element.
  - For each element $i \\in S$, we initialize its parent to itself (or *parent[i] = -1* or *-size* for a set of size 1).
  - This involves iterating through all n elements once and performing a constant number of operations (array assignments) for each element.
  - Therefore, the total time complexity for *MakeUnionFind(S)* is $O(N)$.

- **(iii) Sequence of k Union operations takes at most O(k log k) time:**
  - This statement is also **incorrect** for the standard optimized Union-Find. With both path compression and union by size (or rank), a sequence of M operations (including *Union* and *Find*) on N elements takes \$O(M \\\\alpha(N))\$ time.
  - If only "union by size" is used (without path compression), then a sequence of k union operations (and implicit find operations within them) can take \$O(k \\\\log N)\$ time, where N is the total number of elements.
  - The bound \$O(k \\\\log k)\$ is not a typical bound for Union-Find operations. If it means k is the number of distinct elements involved in the unions, then it still fits the \$O(k \\\\log N)\$ or \$O(k \\\\alpha(N))\$ bounds.
  - For k unions, the total number of *Find* operations could be up to $2k$. Each *Find* operation without path compression could take \$O(\\\\log N)\$ time. So, \$O(k \\\\log N)\$ would be the bound with union by size only.
  - It's possible there's a specific scenario or a different variant implied for \$O(k \\\\log k)\$, but generally, for Union-Find with union by size/rank, the total time for k union operations (and associated finds) is \$O(k \\\\log N)\$ or \$O(k \\\\alpha(N))\$ (amortized).
- **Role of the "name of the larger set" optimization (Union by Size/Rank):**

- This optimization (often called "union by size" or "union by rank") is crucial for keeping the trees representing the disjoint sets relatively flat and bounded in height.
- When performing a *Union(u, v)* operation, after finding the roots $root\_u$ and $root\_v$ of the sets containing u and v respectively:
    - If the size (number of elements) of the set rooted at $root\_u$ is smaller than the size of the set rooted at $root\_v$, then $root\_u$ is made a child of $root\_v$. The size of $root\_v$'s set is updated by adding the size of $root\_u$'s set.
    - Conversely, if $root\_v$'s set is smaller, $root\_v$ is made a child of $root\_u$.
    - If sizes are equal, either can be chosen as the parent, and the size of the combined set is doubled.

- **How this optimization reduces the height of the tree created during the union process:**
    - By always attaching the root of the smaller tree to the root of the larger tree, the "union by size" heuristic ensures that the height of the resulting tree increases only if the two trees being united have the same size. In this case, the height increases by exactly one.
    - Crucially, if a tree of height h is formed using union by size, it must contain at least 2^h nodes. This is because every time the height increases, the size of the combined tree at least doubles.

- Since the total number of elements is N, the maximum height of any tree in the Union-Find structure cannot exceed $\\log\_2 N$.
- This logarithmic height bound is essential for achieving efficient *Find* operations ($O(\\log N)$ worst-case without path compression, and $O(\\alpha(N))$ amortized with path compression), preventing the trees from degenerating into long, skinny paths that would lead to $O(N)$ find times.

5. (a) Give pseudo-code for the following operations of Sequence Abstract Data Type using an array implementation :.

- (i) Insert from the back,.
- (ii) Remove from the end,.
- (iii) To access an element at any index. What is the running time for each of these operations?
- **Array Implementation of Sequence ADT:** Let's assume a dynamic array (resizable array) for the implementation, often called *ArrayList* or *Vector*. *array*: The underlying array. *size*: Current number of elements in the sequence. *capacity*: Total allocated size of the underlying array.
- **(i) Insert from the back (push_back(e))**

```
Algorithm push_back(element e):
    // Check if array is full and needs resizing
    if size == capacity:
        // Double the capacity (or increase by a constant factor)
        new_capacity = capacity * 2 // Or capacity + k
        new_array = create_new_array_of_size(new_capacity)
        copy_elements(array, new_array, size) // Copy existing
elements
        array = new_array
        capacity = new_capacity

    // Insert element at the end
    array[size] = e
    size = size + 1
```

- **Running Time:**
    - **Best Case:** $O(1)$ (constant time) when there is enough space in the array and no resizing is needed.
    - **Worst Case:** $O(size)$ (linear time) when the array is full and a resizing operation is required. This involves allocating a new larger array and copying all existing *size* elements.
    - **Amortized Analysis:** Over a sequence of N insertions, the total time for resizing operations averages out, resulting in an amortized time complexity of $O(1)$ per *push_back* operation, assuming the array capacity grows by a constant multiplicative factor.

- **(ii) Remove from the end (pop_back())**

```
Algorithm pop_back():
    if size == 0:
        throw Exception("Sequence is empty, cannot remove from
end.")

    // Element to be removed (optional: return it)
    // element_removed = array[size - 1]

    size = size - 1

    // Optional: Shrink array if it becomes too empty (e.g., < 25% full)
    // This is usually done to save space, not strictly for correctness.
    // if size > 0 and size < capacity / 4:
    //     new_capacity = capacity / 2
    //     new_array = create_new_array_of_size(new_capacity)
    //     copy_elements(array, new_array, size)
    //     array = new_array
    //     capacity = new_capacity

    // return element_removed (if designed to return)
```

  - **Running Time:** $O(1)$ (constant time). This operation simply decrements the *size* counter. Shrinking the array (if implemented) would incur $O(size)$ costs in its worst case, but amortized analysis for shrinking also leads to $O(1)$.

- **(iii) To access an element at any index (at(index) or operator[])**

```
Algorithm at(index i):
    if i < 0 or i >= size:
        throw Exception("Index out of bounds.")

    return array[i]
```

- **Running Time:** $O(1)$ (constant time). Accessing an element in an array by its index is a direct memory lookup, which is a constant time operation.

5. (b) Do we need to restrict the number of levels in a skip list? Justify your answer.Consider a skip list S with n elements and the insertion policy that restricts the top-level h to a fixed value h=max{10, 2[logn]}, where n is the current number of entries in the skip list.How does the given policy impact the structure of the skip list during insertion?How does the condition [logn] < [log(n+1)] help allow an insertion to increase the height of the skip list by atmost one level?

- **Do we need to restrict the number of levels in a skip list? Justify your answer.**
  - Yes, we generally need to restrict the maximum number of levels (or height) in a skip list.
  - **Justification:**
    - **Memory Efficiency:** Without a restriction, some elements might be promoted to extremely high levels due to the probabilistic nature of level assignment. While rare, this

can lead to a few elements consuming excessive memory for their linked lists, potentially wasting space.

- **Performance Guarantees (Worst Case):** Although the average-case performance of a skip list (with proper probabilistic level assignment) is $O(\\log N)$ for search, insertion, and deletion, an unbounded height means that in a very rare, worst-case scenario (due to extremely unlikely coin flips), a skip list could become very tall, degenerating into a linked list and leading to $O(N)$ operations. Restricting the maximum height (e.g., to $c \\log N$ for some constant c) provides a probabilistic guarantee that operations will remain efficient and prevents such pathological cases.
- **Implementation Simplicity:** A fixed maximum height simplifies the implementation of operations, especially if using arrays of pointers for nodes at different levels.

- **Consider a skip list S with n elements and the insertion policy that restricts the top-level h to a fixed value h=max{10, 2[logn]}, where n is the current number of entries in the skip list.**
  - **How does the given policy impact the structure of the skip list during insertion?**
    - **Dynamic Height Adjustment:** This policy makes the maximum allowed height of the skip list dynamic, adjusting based on the current number of elements n.
    - **Minimum Height Guarantee:** The *max{10, ...}* part ensures that even for very small n, the skip list maintains

a minimum height of 10. This prevents the skip list from becoming too shallow for small datasets, potentially improving performance even if $\\log N$ is small.

- **Logarithmic Growth for Larger N:** For larger n, the $2 \\lceil \\log n \\rceil$ term dominates. This means the maximum height grows logarithmically with the number of elements, which is consistent with the desired $O(\\log N)$ performance characteristics of a skip list.
- **Prevention of Excessive Height:** The policy directly limits the maximum height any newly inserted element can reach. When assigning a random level to a new element, if the randomly generated level exceeds this *h*, the element's level is capped at *h*. This prevents any single element from reaching an "unreasonably" high level, thus maintaining the overall logarithmic structure and preventing extreme memory consumption or degenerate performance for a few nodes. It ensures that the overall height of the skip list remains within a controlled, theoretically sound bound.

- **How does the condition [logn] < [log(n+1)] help allow an insertion to increase the height of the skip list by atmost one level?**
  - The condition $\\lceil \\log n \\rceil \< \\lceil \\log(n+1) \\rceil$ reflects a point where the ceiling of the logarithm increases.
  - Let $H\_{max}(n) = \\text{max}{10, 2\\lceil\\log n\\rceil}$ be the maximum allowed height for a skip list with n elements.

- When an element is inserted, n becomes $n + 1$. The new maximum allowed height is $H\_{max}(n+1) = \\text{max}{10, 2\\lceil\\log(n+1)\\rceil}$.
- If $\\lceil \\log n \\rceil = \\lceil \\log(n+1) \\rceil$, then $H\_max(n + 1) = H\_max(n)$. In this case, the maximum allowed height does not increase. Any new node can only be inserted up to the existing maximum height.
- If $\\lceil \\log n \\rceil \< \\lceil \\log(n+1) \\rceil$, it means $n + 1$ has crossed a power of 2 (or some other boundary where the ceiling of the logarithm increases). In this specific scenario, $2\\lceil\\log(n+1)\\rceil$ will be exactly 2 greater than $2\\lceil\\log n\\rceil$.
- For example, if $n = 3$, $\\lceil \\log 3 \\rceil = 2$, so $H\_{max}(3) = \\text{max}{10, 4} = 10$. If $n = 4$, $\\lceil \\log 4 \\rceil = 2$, $H\_{max}(4) = \\text{max}{10, 4} = 10$.
- If $n = 7$, $\\lceil \\log 7 \\rceil = 3$, $H\_{max}(7) = \\text{max}{10, 6} = 10$. If $n = 8$, $\\lceil \\log 8 \\rceil = 3$, $H\_{max}(8) = \\text{max}{10, 6} = 10$.
- The property of the logarithm (and its ceiling) is that $\\lceil \\log (n+1) \\rceil$ can be at most $\\lceil \\log n \\rceil + 1$.
- Therefore, $2\\lceil\\log(n+1)\\rceil$ can be at most $2(\\lceil\\log n\\rceil + 1) = 2\\lceil\\log n\\rceil + 2$.
- This means the calculated maximum height $2\\lceil\\log n\\rceil$ can increase by at most 2 when n increments to $n + 1$.
- However, the *actual* height of the skip list is defined by the tallest node inserted. An insertion itself only adds *one* new

node. The new node's level is determined probabilistically. Even if the *maximum allowed* height from the formula increases by 2, a single new insertion can only physically extend the list by one level at a time (if its random level is higher than the current tallest node, and within the new max allowed). It doesn't magically create multiple new levels.

- The condition $\\lceil \\log n \\rceil \< \\lceil \\log(n+1) \\rceil$ specifically indicates when the *upper bound for the calculated max height* for the *entire list* increases. An insertion can only ever increase the *actual* height of the skip list by at most one if its randomly determined level is higher than the current maximum level of any node in the skip list and is less than or equal to the *new* max allowed height. The formula simply defines the cap, not how much the height grows per insertion.

6. (a) Consider the following unsorted array of integers: A =.Find the i-th smallest element where i = 7 from the array A using the Randomized Select algorithm, where the first element of the resulting array is selected as the pivot.When do we say a partition is helpful and show that the probability of helpful partition is atleast ½.

- **Array A:**
- **Find the 7th smallest element using Randomized Select (first element as pivot):**
    - **Goal:** Find the 7th smallest element ($i = 7$).
    - Array *A = [8, 3, 2, 7, 1, 5, 4, 6, 9]*

Step 1: Initial call *RandomizedSelect(A, 1, 9, 7)*

- Choose pivot: First element is *8*.
- Partition *A* around *8*:
    - Elements less than 8: *[3, 2, 7, 1, 5, 4, 6]*
    - Elements equal to 8: *[8]*
    - Elements greater than 8: *[9]*
    - After partitioning (relative order of sub-arrays doesn't strictly matter for the algorithm, but elements are placed correctly): *[3, 2, 7, 1, 5, 4, 6, 8, 9]*
- The pivot *8* is at index 8 (considering 1-based indexing for position after sorting the partitioned array). Let's say its rank (position) is $k = 8$.
- We are looking for the 7th smallest element ($i = 7$).
- Since $i=7 \< k=8$, the 7th smallest element must be in the left partition (elements less than 8).
- New array for recursive call: *[3, 2, 7, 1, 5, 4, 6]*
- Adjust i: Still looking for the 7th smallest among these.

Step 2: Recursive call *RandomizedSelect([3, 2, 7, 1, 5, 4, 6], 1, 7, 7)*

- Choose pivot: First element is *3*.
- Partition around *3*:
    - Elements less than 3: *[2, 1]*
    - Elements equal to 3: *[3]*
    - Elements greater than 3: *[7, 5, 4, 6]*
    - After partitioning: *[2, 1, 3, 7, 5, 4, 6]*
- The pivot *3* is at index 3. Its rank is $k = 3$.

- We are looking for the 7th smallest ($i = 7$).
- Since $i = 7\ k = 3$, the 7th smallest element must be in the right partition (elements greater than 3). We need to adjust i.
- New i: $i - k = 7 - 3 = 4$. So, we are looking for the 4th smallest element in the right partition.
- New array for recursive call: *[7, 5, 4, 6]*

Step 3: Recursive call *RandomizedSelect([7, 5, 4, 6], 1, 4, 4)*

- Choose pivot: First element is *7*.
- Partition around *7*:
  - Elements less than 7: *[5, 4, 6]*
  - Elements equal to 7: *[7]*
  - Elements greater than 7: *[]*
  - After partitioning: *[5, 4, 6, 7]*
- The pivot *7* is at index 4. Its rank is $k = 4$.
- We are looking for the 4th smallest ($i = 4$).
- Since $i = k = 4$, the pivot *7* is the 4th smallest element in this sub-array.
- Therefore, *7* is the 7th smallest element in the original array A.

The 7th smallest element is 7.

- **When do we say a partition is helpful and show that the probability of helpful partition is atleast ½.**
  - **When is a partition helpful?** A partition is considered "helpful" or "good" in randomized algorithms like Quicksort or Randomized Select if the pivot element selected results in two

sub-arrays (partitions) that are reasonably balanced in size. Specifically, a partition is helpful if both sub-arrays are at most a constant fraction of the original array's size. For example, a partition is helpful if the pivot falls within the middle 50% of the sorted range of elements. That is, if the pivot's rank (position in a sorted array) is between $N/4$ and $3N/4$, where N is the size of the array being partitioned.

- **Probability of a helpful partition is at least ½:** Let the array to be partitioned have N elements. When we choose a pivot uniformly at random from these N elements, each element has a $1/N$ probability of being chosen. Consider the elements sorted. The "middle 50%" of the elements are those whose ranks are from $\\lfloor N/4 \\rfloor + 1$ to $\\lceil 3N/4 \\rceil$. The number of elements in this range is approximately $\\lceil 3N/4 \\rceil - (\\lfloor N/4 \\rfloor + 1) + 1 = \\lceil 3N/4 \\rceil - \\lfloor N/4 \\rfloor$. For sufficiently large N:
  - If N is a multiple of 4, the number of elements in the middle 50% is $(3N/4) - (N/4) = N/2$.
  - If N is not a multiple of 4, it's approximately $N/2$. For instance, if $N = 5$, the ranks are 1, 2, 3, 4, 5. Middle 50% is elements at rank 2, 3, 4 (3 elements). $3/5\ 1/2$. If $N = 4$, middle 50% is ranks 2, 3 (2 elements). $2/4 = 1/2$. The number of elements that result in a helpful partition (i.e., cause the pivot to fall into the middle 50% of ranks) is at least $N/2$. Since the pivot is chosen uniformly at random, the probability of selecting an element from this "middle

50%" range is: P(\\text{helpful partition}) = \\frac{\\text{Number of elements in middle 50%}}{\\text{Total number of elements}} = \\frac{\\ge N/2}{N} \\ge \\frac{1}{2} Thus, the probability that a randomly chosen pivot yields a helpful partition is at least $1/2$. This forms the basis for the average-case $O(N \\log N)$ performance of randomized Quicksort and $O(N)$ expected performance of Randomized Select.

6. (b) Consider the following directed graph, G with A, B, C, D, S, T as the nodes representing a flow network :.

- S → A (capacity 10)
- S → C (capacity 10)
- A → B (capacity 4)
- A → C (capacity 2)
- A → D (capacity 8)
- B → T (capacity 10)
- C → D (capacity 9)
- D → B (capacity 6)
- D → T (capacity 10) Apply the Ford-Fulkerson algorithm to find the maximum flow from source S to sink T. How does the Ford Fulkerson algorithm ensure that the flow is maximized?

- **Ford-Fulkerson Algorithm for Maximum Flow (S to T):**

Initial flow $f = 0$. Residual capacity $c\_f(u, v) = c(u, v)$.

Iteration 1:

- Find an augmenting path from S to T in the residual graph.
- Path: S $\\to$ A $\\to$ B $\\to$ T
  - Residual capacities: $c\_f(S, A) = 10$, $c\_f(A, B) = 4$, $c\_f(B, T) = 10$
  - Bottleneck capacity: $\\min(10, 4, 10) = 4$
- Augment flow by 4 along S $\\to$ A $\\to$ B $\\to$ T.
  - $f(S, A) = 4$, $c\_f(S, A) = 6$, $c\_f(A, S) = 4$
  - $f(A, B) = 4$, $c\_f(A, B) = 0$, $c\_f(B, A) = 4$
  - $f(B, T) = 4$, $c\_f(B, T) = 6$, $c\_f(T, B) = 4$
- Current flow value: $v(f) = 4$.

Iteration 2:

- Path: S $\\to$ C $\\to$ D $\\to$ T
  - Residual capacities: $c\_f(S, C) = 10$, $c\_f(C, D) = 9$, $c\_f(D, T) = 10$
  - Bottleneck capacity: $\\min(10, 9, 10) = 9$
- Augment flow by 9 along S $\\to$ C $\\to$ D $\\to$ T.
  - $f(S, C) = 9$, $c\_f(S, C) = 1$, $c\_f(C, S) = 9$
  - $f(C, D) = 9$, $c\_f(C, D) = 0$, $c\_f(D, C) = 9$
  - $f(D, T) = 9$, $c\_f(D, T) = 1$, $c\_f(T, D) = 9$
- Current flow value: $v(f) = 4 + 9 = 13$.

Iteration 3:

- Path: S $\\to$ A $\\to$ D $\\to$ T
  - Residual capacities: $c\_f(S,A) = 6$, $c\_f(A,D) = 8$, $c\_f(D,T) = 1$
  - Bottleneck capacity: $\\min(6, 8, 1) = 1$
- Augment flow by 1 along S $\\to$ A $\\to$ D $\\to$ T.
  - $f(S,A)$ becomes $4 + 1 = 5$, $c\_f(S,A) = 5$, $c\_f(A,S) = 5$
  - $f(A,D)$ becomes $0 + 1 = 1$, $c\_f(A,D) = 7$, $c\_f(D,A) = 1$
  - $f(D,T)$ becomes $9 + 1 = 10$, $c\_f(D,T) = 0$, $c\_f(T,D) = 10$
- Current flow value: $v(f) = 13 + 1 = 14$.

Iteration 4:

- Path: S $\\to$ C $\\to$ B $\\to$ T
  - Residual capacities: $c\_f(S,C) = 1$, $c\_f(C,B) = 0$ (no direct edge $C \\to B$ in original graph, assuming no reverse edge, but wait, there is $D \\to B$ and $C \\to D$. Let's check original graph. Ok, it's $C \\to B$ cap 0. Oh, wait, the problem definition is important: C $\\to$ B (weight -1). Capacities are always positive. So let's assume capacity is 1 for $C \\to B$ if not specified. If it is 0, then this path is not possible. If it refers to weight in some other context, let's assume it has some capacity, say 1 for now if not given, as this is a flow problem and capacities are non-negative.

- Let's re-read the capacities: "A -> B (capacity 4), A -> C (capacity 2), A -> D (capacity 8), B -> T (capacity 10), C -> D (capacity 9), D -> B (capacity 6), D -> T (capacity 10)".
- The edge $C \\to B$ has a "weight -1" in problem 2.b, which describes Bellman-Ford on a weighted directed graph. Problem 6.b is a *flow network*. In flow networks, capacities are non-negative. So, I will assume $C \\to B$ has a capacity of 0 or it's not present for flow purposes, as negative capacities are not standard for max flow. Let's list valid capacities from problem 6.b.
- Valid capacities:
  - S $\\to$ A: 10
  - S $\\to$ C: 10
  - A $\\to$ B: 4
  - A $\\to$ C: 2
  - A $\\to$ D: 8
  - B $\\to$ T: 10
  - C $\\to$ D: 9
  - D $\\to$ B: 6
  - D $\\to$ T: 10
- The edge $C \\to B$ with "weight -1" is from a different problem (2.b). It is not part of this flow network. Let's ignore it for 6.b.

Let's re-evaluate residual graph after Iteration 3. Flows:

- $f(S,A) = 5$
- $f(S,C) = 9$
- $f(A,B) = 4$
- $f(A,D) = 1$
- $f(B,T) = 4$
- $f(C,D) = 9$
- $f(D,T) = 10$ (saturated)

Residual capacities:

- $c\_f(S,A) = 5$, $c\_f(A,S) = 5$
- $c\_f(S,C) = 1$, $c\_f(C,S) = 9$
- $c\_f(A,B) = 0$, $c\_f(B,A) = 4$ (A to B is saturated)
- $c\_f(A,C) = 2$ (still available, no flow yet)
- $c\_f(A,D) = 7$, $c\_f(D,A) = 1$
- $c\_f(B,T) = 6$, $c\_f(T,B) = 4$
- $c\_f(C,D) = 0$, $c\_f(D,C) = 9$ (C to D is saturated)
- $c\_f(D,B) = 6$ (still available, no flow yet)
- $c\_f(D,T) = 0$, $c\_f(T,D) = 10$ (D to T is saturated)

Iteration 4 (attempting to find new path):

- Can we find a path from S to T?
  - From S: Can go to A ($c\_f(S,A) = 5$) or C ($c\_f(S,C) = 1$).
  - If S $\\to$ A:
    - From A: Cannot go to B ($c\_f(A,B) = 0$). Can go to C ($c\_f(A,C) = 2$). Can go to D ($c\_f(A,D) = 7$).

- S $\\to$ A $\\to$ C: Path is S $\\to$ A $\\to$ C. Capacity on this segment is $\\min(5, 2) = 2$.
  - From C: Can't go to D ($c\_f(C, D) = 0$). Can go back to S (not useful for augmenting path).
  - Is there a path from C to T? No direct edge.
  - What about $C \\to D$? $c\_f(C, D) = 0$. No.
- S $\\to$ A $\\to$ D: Path is S $\\to$ A $\\to$ D. $c\_f(A, D) = 7$.
  - From D: Cannot go to T ($c\_f(D, T) = 0$). Can go to B ($c\_f(D, B) = 6$). Can go back to C ($c\_f(D, C) = 9$). Can go back to A ($c\_f(D, A) = 1$).
  - If D $\\to$ B, then S $\\to$ A $\\to$ D $\\to$ B $\\to$ T.
    - Capacities: $c\_f(S, A) = 5$, $c\_f(A, D) = 7$, $c\_f(D, B) = 6$, $c\_f(B, T) = 6$.
    - Bottleneck: $\\min(5, 7, 6, 6) = 5$.
    - This is a valid path!

Iteration 4 (Actual):

- Path: S $\\to$ A $\\to$ D $\\to$ B $\\to$ T
  - Residual capacities: $c\_f(S, A) = 5$, $c\_f(A, D) = 7$, $c\_f(D, B) = 6$, $c\_f(B, T) = 6$
  - Bottleneck capacity: $\\min(5, 7, 6, 6) = 5$
- Augment flow by 5 along S $\\to$ A $\\to$ D $\\to$ B $\\to$ T.

- $f(S, A)$ becomes $5 + 5 = 10$, $c\_f(S, A) = 0$, $c\_f(A, S) = 10$ (S to A saturated)
- $f(A, D)$ becomes $1 + 5 = 6$, $c\_f(A, D) = 2$, $c\_f(D, A) = 6$
- $f(D, B)$ becomes $0 + 5 = 5$, $c\_f(D, B) = 1$, $c\_f(B, D) = 5$
- $f(B, T)$ becomes $4 + 5 = 9$, $c\_f(B, T) = 1$, $c\_f(T, B) = 9$

- Current flow value: $v(f) = 14 + 5 = 19$.

Iteration 5:

- Check for augmenting path.
  - From S: Can go to C ($c\_f(S, C) = 1$). Cannot go to A ($c\_f(S, A) = 0$).
  - If S $\\to$ C:
    - From C: Cannot go to D ($c\_f(C, D) = 0$).
    - Can we use a back edge? Yes. A $\\to$ C (capacity 2).
    - There's $c\_f(A, C) = 2$. If we could flow $S \\to \\dots \\to A \\to C \\to \\dots \\to T$.
    - Consider path S $\\to$ C (remaining capacity 1).
    - From C, no forward edges to T.
    - Consider residual graph state.
      - $c\_f(S, A) = 0$, $c\_f(S, C) = 1$.
      - $c\_f(A, B) = 0$, $c\_f(A, C) = 2$, $c\_f(A, D) = 2$.
      - $c\_f(B, T) = 1$.
      - $c\_f(C, D) = 0$.
      - $c\_f(D, B) = 1$, $c\_f(D, T) = 0$.

- Path: S $\\to$ C $\\to$ D (using back edge from D $\\to$ C if needed) or C $\\to$ something.
- The only remaining path from S seems to be S $\\to$ C. From C, the only outgoing edge (with capacity) is to D, which is saturated ($c\_f(C, D) = 0$).
- However, we have back edges. Can we use $D \\to C$ (capacity 9)?
- If S $\\to$ C (cap 1), then from C:
    - No direct path to T.
    - What about $C \\to D$? No. $c\_f(C, D) = 0$.
    - What about $A \\to C$? Yes, $c\_f(A, C) = 2$. Is there a path like $S \\to A \\to C \\to \\dots \\to T$?
        - $c\_f(S, A) = 0$. So $S \\to A$ is saturated. No.
- Let's check paths from source S that can reach T.
    - S $\\to$ A (saturated).
    - S $\\to$ C (residual capacity 1).
        - From C, can we reach D? $c\_f(C, D) = 0$. No.
        - What about a path like S $\\to$ C $\\xrightarrow{\\text{back edge}} D \\xrightarrow{\\text{forward}} B \\xrightarrow{\\text{forward}} T$?

        - S $\\to$ C ($c\_f = 1$) $\\to$ D (using $f(C,D) = 9$ means $D \\to C$ residual capacity is 9)
        - So, $S \\to C \\leftarrow D \\to B \\to T$ is not an augmenting path *from* S *to* T. An augmenting path must use forward edges or reverse edges in the residual graph.
        - The edges in the residual graph are:
            - Forward edges for $c\_f(u,v)\ 0$: (S,C) cap 1; (A,C) cap 2; (A,D) cap 2; (B,T) cap 1; (D,B) cap 1.
            - Backward edges for $f(u,v)\ 0$: (A,S) cap 10; (C,S) cap 9; (B,A) cap 4; (D,A) cap 6; (B,D) cap 5; (T,B) cap 9; (D,C) cap 9; (T,D) cap 10.

- Is there an S-T path in this residual graph?
    - From S, only to C. Path S $\\to$ C (cap 1).
    - From C: Cannot go to D, as $c\_f(C,D) = 0$.
    - Can go to S via $C \\leftarrow D$ using residual edge (D,C) capacity 9? This is a backward edge.
    - S $\\to$ C (cap 1). From C, we have to find a path to T. No direct edge from C to T.
    - If we go $S \\to C \\to \\dots$:

- $S \\to C$ (cap 1). From C, we can only go "backwards" to S ($C \\to S$ has residual cap 9 if $S \\to C$ had flow, but this $S \\to C$ current flow is 9, so $C \\to S$ has 9 capacity in residual graph. That means $c\_f(C,S) = 9$, so an edge $S \\to C$ has flow 9, so we can send 1 more from $S \\to C$ to make it 10)
- Wait, $f(S,C) = 9$. So $c\_f(S,C) = 10 - 9 = 1$. $c\_f(C,S) = 9$.
- Now, look from C. $c\_f(C,D) = 0$.
- No other outgoing edges with positive residual capacity from C.
- This implies there are no more augmenting paths from S to T.

The Ford-Fulkerson algorithm stops when no more augmenting paths can be found.

Final flow distribution:

- $S \\to A: 10$ (saturated)
- $S \\to C: 9$
- $A \\to B: 4$
- $A \\to C: 0$ (No flow)
- $A \\to D: 6$
- $B \\to T: 9$
- $C \\to D: 9$

- $D \\to B: 5$
- $D \\to T: 10$ (saturated)

Max flow value = sum of flows out of S = $f(S, A) + f(S, C) = 10 + 9 = 19$. (Also sum of flows into T = $f(B, T) + f(D, T) = 9 + 10 = 19$).

- **How does the Ford-Fulkerson algorithm ensure that the flow is maximized?**
  - The Ford-Fulkerson algorithm ensures that the flow is maximized based on the **Max-Flow Min-Cut Theorem**.
  - **The Theorem states:** The maximum amount of flow through a network from a source s to a sink t is equal to the minimum capacity of an $s - t$ cut in the network.
  - **How Ford-Fulkerson achieves this:**
    1. **Augmenting Paths:** The algorithm repeatedly finds an augmenting path from the source to the sink in the residual graph and increases the flow along this path by its bottleneck capacity. This process continues as long as an augmenting path exists.
    2. **Residual Graph:** The residual graph G_f (for flow f) represents the remaining capacity of edges. A forward edge (u, v) in G_f has capacity $c(u, v) - f(u, v)$. A backward edge (v, u) in G_f has capacity $f(u, v)$. This allows the algorithm to "push back" flow, effectively rerouting it if a better path becomes available.

iii. **Termination Condition:** The algorithm terminates when no more augmenting paths can be found from the source s to the sink t in the residual graph.

iv. **Min-Cut Connection:** When the algorithm terminates, it means that the source s and sink t are disconnected in the residual graph. This implies that all paths from s to t are saturated (i.e., they contain at least one edge with zero residual capacity). The set of these saturated edges forms an $s - t$ cut. Due to the construction of the residual graph, the capacity of this cut (sum of capacities of edges going from the s-side to the t-side) will be equal to the total flow. Since no more flow can be pushed, this cut is a minimum cut, and by the Max-Flow Min-Cut Theorem, the current flow is indeed the maximum possible flow.

7. (a) Let V be a vector implemented by means of an extendable array A which is extended with [N/4] additional cells. Capacity of V increases from N to N + [N/4]. Initially V is empty and A has size N = 1. Show that the total time to perform a series of n push operations in V is O(n).If we extend the array from N to N + k where k is a constant, can the sequence of N push operations in V still be done in O(n) time?

- **Total time to perform a series of n push operations in V (Capacity increases from N to N + [N/4]):**

  This is a variation of amortized analysis. When the array is extended by a constant fraction ($1/4$ of its current size), the amortized cost per operation is still constant.

Let's trace the costs:

- Initial size: $N\_0 = 1$.
- Insertion 1: Cost 1 (insert). Size becomes 1.
- Insertion 2: Array is full (size 1, capacity 1). Needs resize. Old capacity $N = 1$. New capacity $1 + \\lfloor 1/4 \\rfloor = 1+0=1$. This is problematic as the capacity isn't growing effectively for small N. Let's assume $N + \\max(1, \\lfloor N/4 \\rfloor)$ to ensure growth, or that N starts at a reasonable size (e.g., 4) for $\\lfloor N/4 \\rfloor$ to be at least 1. If $\\lfloor N/4 \\rfloor$ can be 0, the array won't grow.
- **Clarification of *[N/4]*:** Assuming *[N/4]* means integer division $\\lfloor N/4 \\rfloor$. For $N = 1,2,3$, $\\lfloor N/4 \\rfloor = 0$. This would mean the array never grows if initial capacity is 1, and the first 3 insertions would immediately fail to find space if capacity just increases by 0.

Let's assume the problem implicitly means the capacity grows such that it always adds at least 1, or that it means $N + N/4$ where $N/4$ is always at least 1. A more standard analysis uses doubling. If it's literally $N + \\lfloor N/4 \\rfloor$, and initial $N = 1$, this scheme has issues.

Let's assume a practical interpretation where the capacity increases by a factor of 1.25 (i.e., $N \\to N + N/4 = 1.25N$): When a resize occurs, it costs $O(N\_old)$ to copy elements. The capacities will be: $C\_0 = 1$. $C\_1 = C\_0 \\times 1.25 \\approx 1$. $C\_2 \\approx 1.25 \\times 1.25 \\approx 1.56$. $C_3 \\approx 1.56 \\times

1.25 \\approx 1.95$. This geometric growth ensures that total cost is $O(n)$.

Let the capacity be $C\_0, C\_1, C\_2, \\ldots, C\_k$ such that $C\_{j+1} = C\_j + \\lfloor C\_j/4 \\rfloor$. The total cost of n insertions is the sum of insertion costs plus the sum of copying costs during resizes. Each insertion (not involving a resize) costs 1. So, n insertions give a base cost of n. Consider the resize costs: when capacity grows from C_j to $C\_j + 1$, it costs C_j to copy elements. $C\_{j+1} = C\_j + \\lfloor C\_j/4 \\rfloor \\ge C\_j + C\_j/4 - 1 = (5/4)C\_j - 1$. So $C\_j + 1$ is at least $(5/4)C\_j$ for large C_j. This means $C\_j \\le (4/5)C\_{j+1}$. The total cost for copying is $\\sum\_{j=0}^{k-1} C\_j$, where $C\_k \\approx n$. $C\_k \\approx (5/4)^k C\_0$. So $n \\approx (5/4)^k C\_0$, which implies $k \\approx \\log\_{5/4} (n/C\_0) = O(\\log n)$. The sum of costs C_j is a geometric series: $C\_0 + C\_1 + \\dots + C\_{k-1}$. $C\_0 + (4/5)C\_1 + (4/5)^2 C\_2 + \\dots$. The sum $\\sum C\_j$ where $C\_j \\ge (4/5) C\_{j+1}$ is bounded by C_k + (4/5)C_k + (4/5)^2C_k + \\dots = C_k \\sum_{p=0}^\\infty (4/5)^p = C_k \\frac{1}{1 - 4/5} = C_k \\times 5 = O(n). Therefore, the total time to perform a series of n push operations is $O(n)$ (amortized $O(1)$ per operation), provided the base increase for small N is handled correctly (e.g. $N + \\max(1, \\lfloor N/4 \\rfloor)$).

- **If we extend the array from N to N + k where k is a constant, can the sequence of N push operations in V still be done in O(n) time?**

- No, if the array is extended by a constant *k* (additive increase) instead of a multiplicative factor, a sequence of N push operations **cannot** be done in $O(N)$ total time.
- **Argument:**
  - Let the initial capacity be C_0.
  - When the array fills up, it is extended by *k* cells. The new capacity becomes $C\_old + k$.
  - To perform N push operations, the array will need to be resized approximately N/k times.
  - The costs for copying elements during resizing will be:
    - C_0 for the first resize.
    - $C\_0 + k$ for the second resize.
    - $C\_0 + 2k$ for the third resize.
    - ...
    - $C\_0 + ((N/k) - 1)k$ for the last resize.
  - The total copying cost will be approximately $\\sum\_{i=0}^{N/k - 1} (C\_0 + i \\cdot k)$.
  - This sum is roughly equivalent to $\\int\_0^{N/k} (C\_0 + xk) dx = [C\_0 x + k x^2/2]\_0^{N/k} = C\_0 (N/k) + k (N/k)^2 / 2 = O(N) + O(N^2/k) = O(N^2)$.
- Since k is a constant, the dominant term is $O(N^2)$.
- Thus, additive capacity increase leads to $O(N^2)$ total time for N operations, not $O(N)$. This is why multiplicative (geometric) resizing is preferred for dynamic arrays.

7. (b) Explain the deletion process in B-trees.Consider a B-tree of order 4 (maximum 3 keys per node) with the following initial structure:.(Image of B-tree structure: a root node with three children:, , and is provided).Delete the keys 30 and 50 from the B-tree. Show all the necessary steps involved in the deletion process.

- **Explanation of the Deletion Process in B-trees:**

  Deletion in a B-tree is more complex than insertion because it must maintain the B-tree properties, especially the minimum degree requirement (minimum number of keys and children in non-root nodes). The general steps are:

  u. **Locate the Key:** Traverse the tree to find the node containing the key to be deleted.

  v. **Case 1: Key is in a Leaf Node:**

    - If the leaf node has more than the minimum number of keys (i.e., it can afford to lose a key without violating the minimum degree requirement), simply remove the key.
    - If the leaf node has exactly the minimum number of keys:
      - **Borrow from a Sibling:** Try to borrow a key from an immediate left or right sibling. If a sibling has more than the minimum keys, steal a key from it and move the appropriate key from the parent down to the current node, and the key from the sibling up to the parent. This ensures both nodes maintain the minimum.

- **Merge with a Sibling:** If no sibling can spare a key (i.e., all siblings have only the minimum number of keys), merge the current node with a sibling. This involves moving the separator key from the parent node down into the newly merged node. The parent then loses a key and a child. This merging process might propagate upwards if the parent also falls below its minimum key count.

w. **Case 2: Key is in an Internal Node:**

- If the key k is in an internal node, find its inorder predecessor (largest key in the left subtree) or inorder successor (smallest key in the right subtree). Let's call it k'.
- **Swap and Delete from Leaf:** Replace k with k'. Now the problem is reduced to deleting k' from a leaf node (where k' was found). Proceed with deletion from the leaf node (Case 1). This ensures that the structure and sorted order are maintained. If the leaf node from which k' was taken becomes underflowed, apply borrowing or merging as in Case 1.

Propagation: If a merge operation causes the parent node to underflow, the deletion process (borrowing or merging) is recursively applied to the parent. This can continue up to the root. If the root node merges with its only child and loses its last key, the child becomes the new root, and the height of the tree decreases.

- **Consider a B-tree of order 4 (maximum 3 keys per node) with the following initial structure:** Order 4 means max keys = 3, min keys (except root) = $\\lceil 4/2 \\rceil - 1 = 2 - 1 = 1$. (Wait, order *m* means *m-1* keys and *m* children. So *min_keys* is $\\lceil m/2 \\rceil - 1$. For order 4, min keys is $4/2 - 1 = 2 - 1 = 1$. Max keys is 3. Yes, this is correct for a common definition of B-tree order.)

  Initial Structure (textual representation as per user's constraint):

```
      [20]
     / | \
    /  |  \
[10]  [30, 40]  [60, 70]
```

  Children of [10]: (none shown, implies leaf children) Children of [30, 40]: (none shown, implies leaf children) Children of [60, 70]: (none shown, implies leaf children)

  Let's assume the provided children are leaf nodes themselves for simplicity, as no further sub-trees are shown. If they are internal nodes, their children are implicit and empty leaves.

  Delete Key 30:

  x. **Locate 30:** Key 30 is in the internal node *[30, 40]*.
  y. **Internal Node Deletion:** Find its inorder successor (smallest key in the right child subtree). Assuming the children are leaf nodes, the inorder successor of 30 would be 40 (the next key in the same node). However, the standard approach for internal

node deletion is to swap with an inorder predecessor/successor from a leaf node to simplify handling underflow at the leaves.

  - If *[30, 40]* is a leaf node, then it simply becomes *[40]*. This node *[40]* now has 1 key, which is the minimum (order 4, min keys = 1). So, no underflow.
  - Let's assume the standard case where the key to be deleted from an internal node is replaced by its inorder predecessor or successor, and then that predecessor/successor is deleted from its leaf node.
  - The inorder successor of 30 is 40 (the next key in the same node, or if 30 was at end of node, would be first key of right child). Let's assume the node structure is $D = [k\_1, k\_2, k\_3]$. So 30 is k_1. Its successor for internal node replacement logic is from its right child. But since these are not internal nodes with children, let's treat it as a leaf.

- **Re-interpreting structure:** The provided structure *[20]* with children *[10]*, *[30, 40]*, *[60, 70]* implies that *[10]*, *[30, 40]*, *[60, 70]* are internal nodes that would have *their own* children. Since no image of children of these nodes is provided, I will assume the typical B-tree behavior where keys from an internal node are replaced by keys from leaf nodes.

Let's assume *[10]*, *[30,40]*, *[60,70]* are children of the root and are effectively leaf nodes for the purpose of deletion, meaning they don't have further children containing values. This is not strictly a B-tree definition, as internal nodes typically only contain separators and have children. Let's use the interpretation that *[10]*, *[30, 40]*, *[60, 70]* are actual *nodes* at level 1, and they are not necessarily leaves. Let's assume there are implicit leaf children for each of these nodes.

Correct Approach (Standard B-tree deletion): Assume the provided image implies that *[10]*, *[30, 40]*, *[60, 70]* are internal nodes at level 1, and their keys represent ranges for their child pointers. Since keys 30 and 50 are to be deleted, and 50 is not in the given structure, let's assume it's in a leaf node that would be a child of *[30,40]* or *[60,70]*.

Let's adjust the interpretation of the structure based on standard B-tree definitions. The nodes *[10]*, *[30, 40]*, *[60, 70]* are internal nodes. Their keys are separators. Root: *[20]* Children:

- Node L1 (left of 20): *[10]* (This node should have children representing values $\<10$, between 10 and 20).
- Node M1 (between 20 and 60): *[30, 40]* (This node should have children for values $\<30$, between $30,40$, and 40).
- Node R1 (right of 60): *[60, 70]* (This node should have children for values $\<60$, between $60,70$, and 70).

Since the problem states "Delete the keys 30 and 50 from the B-tree", and the current keys shown are 10, 20, 30, 40, 60, 70. Key 50 is *not* in the tree structure shown. This indicates that the problem implicitly expects the user to understand that 50 *would have been* in a leaf child of either *[30,40]* or *[60,70]*.

Given the input structure (which is a bit ambiguous without leaf nodes shown), let's make an assumption to proceed: Let's assume *[10]*, *[30, 40]*, *[60, 70]* are indeed the leaf nodes themselves, and the root *[20]* is the only internal node. This is a common simplification in textbook problems when full leaf-level detail is too verbose. In this case, a B-tree of order 4 means min 1 key per non-root node.

Adjusted Initial Structure (assuming *[10]*, *[30,40]*, *[60,70]* are leaf nodes):

```
     [20]
    /  |  \
   /   |   \
[10]  [30, 40]  [60, 70]
```

Delete Key 30:

z. **Locate 30:** Key 30 is in the leaf node *[30, 40]*.

aa. **Check Minimum:** This node *[30, 40]* has 2 keys. The minimum number of keys is 1. It can afford to lose a key.

bb. **Remove Key:** Remove 30 from *[30, 40]*.

cc. **Result:** The node becomes *[40]*.

```
     [20]
    /  |  \
   /   |   \
[10]  [40]   [60, 70]
```

Delete Key 50:

dd. **Locate 50:** Key 50 is not in the shown structure. In a B-tree, 50 would logically be in a leaf node pointed to by the *[40]* (if it was an internal node) or the *[60,70]* node. Given the simplified structure, and 50 is between 40 and 60.

- If *[40]* is a leaf: 50 is not there.
- If *[60,70]* is a leaf: 50 is not there.
- This means 50 must have been in an "implicit" leaf node that would be a child of *[20]* in the range *(40, 60)*. But there is no child for *(40,60)*.

This indicates a strong ambiguity in the problem's provided initial structure for a B-tree, particularly concerning where 50 would reside and the nature of the nodes [10], [30,40], [60,70].

Let's assume a more standard B-tree structure that *could* contain 50: Root: [20] Children (pointers):

- P1 (keys < 20): Points to node [10] (which is an internal node or leaf).
- P2 (keys > 20): Points to node [60, 70] (which is an internal node or leaf).

This doesn't fit the image which shows 3 children. The image shows: *[20]* has three children: Node1 (for values $\<20$), Node2 (for values between 20 and 60 if 20 and 60 are keys in root), Node3 (for values 60). So, it means the structure is like: Root: *[20, 60]* Children:

- *[10]* (values < 20)
- *[30, 40]* (values between 20 and 60)
- *[70]* (values > 60) But the image explicitly states root *[20]* and children *[10]*, *[30,40]*, *[60,70]*. This implies 20 is the only key in the root, and it has 3 children, which is impossible for a

standard B-tree of order 4 (max 3 keys, max 4 children; if root has 1 key it has 2 children).

Given the constraint that I cannot make diagrams and the image provided in the prompt is not visible to me directly, I must work with the textual description *a root node with three children:, , and is provided* where the actual values for children were implicitly given as *[10]*, *[30,40]*, *[60,70]* earlier in the example text.

Therefore, I am forced to assume the B-tree structure provided is:

- Root: *[20]*
- Its children are the nodes: *Node_A = [10]*, *Node_B = [30, 40]*, *Node_C = [60, 70]*.
- And these *Node_A*, *Node_B*, *Node_C* are *leaf nodes*.
- This is an unusual B-tree structure for order 4 (a key in the root *20* should separate two children. Having 3 children for 1 key is impossible in a standard B-tree of order 4. An order 4 B-tree root with 1 key can have 2 children; with 2 keys can have 3 children; with 3 keys can have 4 children).
- **Proceeding under the assumption of the literal interpretation of the problematic structure (where *[20]* has 3 children *[10]*, *[30,40]*, *[60,70]*, and these are leaves):**

Initial B-tree (Order 4, Max 3 keys, Min 1 key per non-root node):

```
 [20]
/ | \
   /  |  \
[10]  [30, 40]  [60, 70]
```

Deletion of 30:

ee. Locate 30: Found in node *[30, 40]*.

ff. This node has 2 keys (30, 40). Minimum keys required is 1. It can afford to lose a key.

gg. Remove 30.

hh. Resulting tree after deleting 30:

```
        [20]
       / | \
      /  |  \
  [10]  [40]   [60, 70]
```

Deletion of 50:

ii. Locate 50: Key 50 is not in any of the existing nodes *[10]*, *[40]*, *[60, 70]*, nor is it the root *[20]*.

jj. In a correctly structured B-tree of this type, 50 would belong in a leaf node that is a child of the *[40]* node's pointer that handles values between 40 and 60. Given that *[40]* is a leaf, this key *50* cannot be in the tree described.

Conclusion for Key 50: Based on the provided ambiguous tree structure, key 50 cannot be found or deleted as it does not exist in the defined nodes. This suggests an issue with the problem's example tree or the keys to delete.

If I am forced to assume a standard B-tree structure where 50 *could* exist: Let's assume the root is *[20]*, its left child contains keys $\<20$ and its right child keys 20. Let's also assume the other nodes are children of the root and contain appropriate keys. If 50 were to be present, it would logically be in a node (perhaps a leaf node) in the subtree corresponding to values greater than 20. For example, if the node *[30, 40]* was actually *[30]* and had a right child containing *[40, 50]* etc. this would be too complex without a proper image.

Given the limitations, I will stick to the literal interpretation of the structure for 30, and state the issue for 50.

Final Steps (assuming 50 is not in the given initial structure): The tree after deleting 30:

```
     [20]
    /  |  \
   /   |   \
[10]  [40]   [60, 70]
```

Attempting to delete 50 from this structure is not possible as 50 is not present.

If the question implies that the nodes listed were placeholders for an internal node where 20 has values <20 in [10], between 20 and 60 in [30,40] and >60 in [60,70] this would require the root to be [20, 60] not just [20].

Due to the inherent contradiction in the given B-tree structure description for order 4 and having a single key root *[20]* with

three children *[10]*, *[30,40]*, *[60,70]*, which violates B-tree properties, I can only provide the steps based on the most direct interpretation (where the children are leaves and 50 is absent).