Interview Questions: Linked Lists and Vector Data Structures

This document provides a comprehensive set of interview questions and answers related to Linked Lists (Singly and Doubly) and Vector data structures. The explanations are designed to be simple and accurate, with C++ code examples where applicable.

Section 1: Introduction to Linked Lists

Question 1: What is a Linked List?

Answer: A Linked List is a linear data structure where elements are not stored at contiguous memory locations. Instead, each element (called a **node**) consists of two parts: the **data** and a **pointer (or reference)** to the next node in the sequence. This structure allows for efficient insertion and deletion operations compared to arrays.

Question 2: What are the main components of a Linked List node?

Answer: A typical Linked List node has two main components:

1. **Data:** This part stores the actual value or information of the element.
2. **Next Pointer (or Link):** This part stores the memory address (or reference) of the next node in the sequence. The last node's next pointer typically points to `NULL` (or `nullptr` in C++), indicating the end of the list.

Question 3: What are the advantages of Linked Lists over arrays?

Answer:

- **Dynamic Size:** Linked lists can grow or shrink in size dynamically at runtime, as memory is allocated for nodes only when needed. Arrays have a fixed size.

- **Efficient Insertions and Deletions:** Inserting or deleting an element in a linked list is generally more efficient (O(1) if the position is known, O(n) otherwise) compared to arrays (O(n)), as it only requires updating a few pointers, not shifting elements.
- **No Memory Waste:** Linked lists only allocate memory for the elements they actually store, avoiding the potential for wasted memory due to pre-allocation (as seen in arrays or vectors that reserve capacity).

Question 4: What are the disadvantages of Linked Lists compared to arrays/vectors?

Answer:

- **No Random Access:** Elements in a linked list cannot be accessed directly by index (e.g., `list[i]`). To access an element, you must traverse the list from the beginning, which takes O(n) time.
- **More Memory Overhead:** Each node in a linked list requires extra memory to store the pointer(s) to the next (and previous) node(s), which can be significant for large numbers of small data elements.
- **Poor Cache Performance:** Due to non-contiguous memory allocation, linked list nodes can be scattered throughout memory, leading to poorer cache locality and potentially slower performance compared to arrays/vectors during traversal.

Question 5: What is the time complexity for common operations in a Singly Linked List?

Answer:

- **Insertion at Beginning:** O(1)
- **Deletion at Beginning:** O(1)
- **Insertion at End:** O(n) (unless a tail pointer is maintained, then O(1))
- **Deletion at End:** O(n) (requires traversing to the second-to-last node)
- **Insertion at Specific Position:** O(n)
- **Deletion at Specific Position:** O(n)
- **Searching:** O(n)

- **Traversal:** O(n)
- **Access by Index (Random Access):** O(n)

Section 2: Singly Linked Lists

Question 6: Describe the structure of a Singly Linked List node.

Answer: A Singly Linked List node typically contains two fields:

1. **Data:** Stores the actual value of the element.
2. **Next Pointer:** A pointer (or reference) to the next node in the sequence. The `next` pointer of the last node is `nullptr`.

C++ Node Structure:

```
struct Node {
    int data;      // Data part
    Node* next;    // Pointer to the next node

    Node(int val) : data(val), next(nullptr) {}
};
```

Question 7: How do you traverse a Singly Linked List?

Answer: To traverse a Singly Linked List, you start from the `head` of the list and follow the `next` pointers until you reach a node whose `next` pointer is `nullptr` (the end of the list).

Algorithm:

1. Initialize a `current` pointer to the `head` of the list.
2. While `current` is not `nullptr`: a. Process `current->data`. b. Move `current` to `current->next`.

C++ Example:

```cpp
void traverse(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
```

Question 8: How do you insert a node at the beginning of a Singly Linked List?

Answer: To insert a node at the beginning:

1. Create a new node with the given data.
2. Set the `next` pointer of the new node to the current `head`.
3. Update the `head` of the list to point to the new node.

C++ Example:

```cpp
Node* insertAtBeginning(Node* head, int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    return newNode; // New head
}
```

Question 9: How do you insert a node at the end of a Singly Linked List?

Answer: To insert a node at the end:

1. Create a new node with the given data.
2. If the list is empty, the new node becomes the `head`.
3. Otherwise, traverse the list to find the last node.
4. Set the `next` pointer of the last node to the new node.

C++ Example:

```
Node* insertAtEnd(Node* head, int val) {
    Node* newNode = new Node(val);
    if (head == nullptr) {
        return newNode; // New head
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
    return head;
}
```

Question 10: How do you delete a node from the beginning of a Singly Linked List?

Answer: To delete a node from the beginning:

1. Check if the list is empty. If so, there's nothing to delete.
2. Store the current `head` node in a temporary pointer.
3. Update the `head` to point to the next node (`head = head->next`).
4. Delete the temporary node to free memory.

C++ Example:

```
Node* deleteFromBeginning(Node* head) {
    if (head == nullptr) {
        return nullptr;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
    return head;
}
```

Question 11: How do you delete a node from the end of a Singly Linked List?

Answer: To delete a node from the end:

1. Handle empty list or single-node list as special cases.
2. Traverse the list with two pointers: `current` and `previous`. `current` moves one step ahead of `previous`.

3. When `current` reaches the last node, `previous` will be at the second-to-last node.
4. Set `previous->next` to `nullptr`.
5. Delete `current` (the last node).

C++ Example:

```
Node* deleteFromEnd(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        delete head;
        return nullptr;
    }
    Node* current = head;
    Node* previous = nullptr;
    while (current->next != nullptr) {
        previous = current;
        current = current->next;
    }
    previous->next = nullptr;
    delete current;
    return head;
}
```

Question 12: How do you reverse a Singly Linked List?

Answer: Reversing a Singly Linked List involves changing the direction of the `next` pointers. This can be done iteratively or recursively.

Iterative Approach:

1. Initialize three pointers: `prev = nullptr`, `current = head`, `next = nullptr`.
2. Iterate while `current` is not `nullptr`: a. Store `current->next` in `next`. b. Set `current->next = prev` (reverse the link). c. Move `prev` to `current`. d. Move `current` to `next`.
3. The new `head` will be `prev`.

C++ Example (Iterative):

```
Node* reverseList(Node* head) {
    Node* prev = nullptr;
    Node* current = head;
    Node* next = nullptr;
    while (current != nullptr) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev; // New head
}
```

Question 13: What is a

circular singly linked list?

Answer: A circular singly linked list is a variation of a singly linked list where the `next` pointer of the last node points back to the first node (head), instead of `nullptr`. This forms a continuous loop. It is useful for applications where you need to cycle through elements repeatedly, like a round-robin scheduler or a buffer.

Question 14: How do you detect a loop in a Singly Linked List?

Answer: The most common and efficient way to detect a loop in a Singly Linked List is using **Floyd's Cycle-Finding Algorithm**, also known as the

tortoise and hare algorithm.

Algorithm:

1. Use two pointers, `slow` and `fast`, both starting at the `head`.
2. In each iteration, move `slow` by one step and `fast` by two steps.
3. If there is a loop, the `fast` pointer will eventually catch up to the `slow` pointer, and they will meet at some node within the loop.
4. If the `fast` pointer reaches `nullptr` (or `fast->next` is `nullptr`), there is no loop.

C++ Example:

```cpp
bool hasLoop(Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true; // Loop detected
        }
    }
    return false; // No loop
}
```

Question 15: How do you find the middle element of a Singly Linked List?

Answer: You can find the middle element of a Singly Linked List in a single pass using the two-pointer approach (slow and fast pointers).

Algorithm:

1. Initialize two pointers, `slow` and `fast`, both at the `head`.
2. Move `slow` by one step and `fast` by two steps in each iteration.
3. When `fast` reaches the end of the list, `slow` will be at the middle element.

C++ Example:

```cpp
Node* findMiddle(Node* head) {
    if (head == nullptr) {
        return nullptr;
    }
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow; // Middle element
}
```

Section 3: Doubly Linked Lists

Question 16: What is a Doubly Linked List?

Answer: A Doubly Linked List is a more complex type of linked list where each node has three components:

1. **Data:** The actual value of the element.
2. **Next Pointer:** A pointer to the next node in the sequence.
3. **Previous Pointer:** A pointer to the previous node in the sequence.

This bidirectional linking allows for traversal in both forward and backward directions.

Question 17: What are the advantages of a Doubly Linked List over a Singly Linked List?

Answer:

- **Bidirectional Traversal:** You can traverse the list in both forward and backward directions.
- **Efficient Deletion:** Deleting a node is more efficient if you have a pointer to the node to be deleted, as you don't need to traverse the list to find the previous node.
- **Easier to Implement Certain Operations:** Operations like reversing the list or inserting before a given node are simpler to implement.

Question 18: What are the disadvantages of a Doubly Linked List?

Answer:

- **More Memory Overhead:** Each node requires an extra pointer (the `prev` pointer), which increases the memory usage compared to a Singly Linked List.
- **More Complex Implementation:** The implementation of insertion and deletion operations is more complex as you need to manage two pointers per node.

Question 19: Describe the structure of a Doubly Linked List node.

Answer: A Doubly Linked List node has three fields:

1. **Data:** Stores the value.
2. **Next Pointer:** Points to the next node.
3. **Previous Pointer:** Points to the previous node.

C++ Node Structure:

```cpp
struct Node {
    int data;
    Node* next;
    Node* prev;

    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};
```

Question 20: How do you insert a node at the beginning of a Doubly Linked List?

Answer: To insert a node at the beginning:

1. Create a new node.
2. Set the `next` pointer of the new node to the current `head`.
3. If the list is not empty, set the `prev` pointer of the current `head` to the new node.
4. Update the `head` to point to the new node.

C++ Example:

```cpp
Node* insertAtBeginning(Node* head, int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    if (head != nullptr) {
        head->prev = newNode;
    }
    return newNode; // New head
}
```

Question 21: How do you insert a node at the end of a Doubly Linked List?

Answer: To insert a node at the end:

1. Create a new node.
2. If the list is empty, the new node becomes the `head`.
3. Otherwise, traverse to the last node.
4. Set the `next` pointer of the last node to the new node.
5. Set the `prev` pointer of the new node to the last node.

C++ Example:

```
Node* insertAtEnd(Node* head, int val) {
    Node* newNode = new Node(val);
    if (head == nullptr) {
        return newNode;
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
    newNode->prev = current;
    return head;
}
```

Question 22: How do you delete a node from a Doubly Linked List (given a pointer to the node)?

Answer: To delete a node from a Doubly Linked List (given a pointer `nodeToDelete`):

1. If `nodeToDelete` is the `head`, update the `head` to `nodeToDelete->next`.
2. If `nodeToDelete` has a previous node, update its `next` pointer to point to `nodeToDelete->next`.
3. If `nodeToDelete` has a next node, update its `prev` pointer to point to `nodeToDelete->prev`.
4. Delete `nodeToDelete`.

C++ Example:

```
void deleteNode(Node* nodeToDelete) {
    if (nodeToDelete == nullptr) {
        return;
    }
    if (nodeToDelete->prev != nullptr) {
        nodeToDelete->prev->next = nodeToDelete->next;
    }
    if (nodeToDelete->next != nullptr) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    }
    delete nodeToDelete;
}
```

Question 23: What is a circular doubly linked list?

Answer: A circular doubly linked list is a variation where the `next` pointer of the last node points to the `head`, and the `prev` pointer of the `head` points to the last node. This creates a circular structure that can be traversed in both directions indefinitely.

Section 4: Vector Data Structures

Question 24: What is a vector data structure?

Answer: A vector data structure (often implemented as a dynamic array) is a sequence container that stores elements in a contiguous block of memory. It provides fast random access (O(1)) and can dynamically resize itself to accommodate new elements.

Question 25: How does a vector differ from a linked list?

Answer:

Feature	Vector	Linked List
Memory Allocation	Contiguous	Non-contiguous
Random Access	O(1)	O(n)
Insertion/Deletion (Middle)	O(n)	O(1) (if position is known)
Memory Overhead	May have unused capacity	Pointer overhead per node
Cache Locality	Excellent	Poor

Question 26: What are the advantages of using a vector?

Answer:

- **Fast Random Access:** O(1) access to elements by index.
- **Good Cache Performance:** Contiguous memory improves cache locality.
- **Efficient Additions at End:** Amortized O(1) for `push_back`.
- **Automatic Memory Management:** Handles resizing and memory deallocation automatically.

Question 27: What are the disadvantages of using a vector?

Answer:

- **Inefficient Insertions/Deletions in Middle:** O(n) due to element shifting.
- **Reallocation Overhead:** Can be costly when the vector needs to grow and copy elements to a new memory block.
- **Memory Waste:** May allocate more memory than needed (capacity > size).

Question 28: Explain `size()` vs. `capacity()` in a vector.

Answer:

- **`size()`:** The number of elements currently stored in the vector.
- **`capacity()`:** The total number of elements the vector can hold without needing to reallocate memory.

`size()` is always less than or equal to `capacity()`.

Question 29: What is the time complexity for common operations in a vector?

Answer:

- **Access by Index:** O(1)
- **Insertion at End (`push_back`):** Amortized O(1)
- **Deletion at End (`pop_back`):** O(1)

- **Insertion in Middle:** O(n)
- **Deletion in Middle:** O(n)
- **Searching:** O(n)

Question 30: When would you use `reserve()` with a vector?

Answer: You would use `reserve()` when you know the approximate number of elements you will store in the vector beforehand. This pre-allocates memory, preventing multiple reallocations and improving performance when adding a large number of elements.

Section 5: C++ STL Containers (`std::list`, `std::vector`)

Question 31: What is `std::list` in C++?

Answer: `std::list` is the C++ Standard Template Library's implementation of a doubly linked list. It provides efficient (O(1)) insertion and deletion of elements anywhere in the list, but does not support random access (O(n)).

Question 32: What is `std::vector` in C++?

Answer: `std::vector` is the C++ STL's implementation of a dynamic array. It provides fast random access (O(1)) and efficient insertions/deletions at the end (amortized O(1)), but is inefficient for insertions/deletions in the middle (O(n)).

Question 33: When would you choose `std::list` over `std::vector`?

Answer: You would choose `std::list` over `std::vector` when your primary operations involve frequent insertions and deletions in the middle of the container, and you do not need fast random access to elements.

Question 34: When would you choose `std::vector` over `std::list`?

Answer: You would choose `std::vector` over `std::list` when you need fast random access to elements by index, and most of your insertions and deletions occur at the end of the container.

Question 35: What is an iterator in the context of STL containers?

Answer: An iterator is an object that acts like a generalized pointer, allowing you to traverse and access elements in a container. `std::vector` provides random-access iterators, while `std::list` provides bidirectional iterators.

Question 36: What is iterator invalidation?

Answer: Iterator invalidation occurs when an operation on a container makes one or more of its iterators unusable. For `std::vector`, insertions, deletions, and reallocations can invalidate iterators. For `std::list`, iterators are only invalidated when the element they point to is deleted.

Question 37: How do you sort a `std::vector`? How do you sort a `std::list`?

Answer:

- **`std::vector`:** Use `std::sort(vec.begin(), vec.end());` from the `<algorithm>` header.
- **`std::list`:** Use the member function `myList.sort();`. `std::sort` cannot be used with `std::list` because it requires random-access iterators.

Question 38: What is `std::forward_list` in C++?

Answer: `std::forward_list` (introduced in C++11) is the STL's implementation of a singly linked list. It is more memory-efficient than `std::list` as it only stores a `next` pointer, but it only supports forward traversal.

Question 39: How do `emplace_back()` and `push_back()` differ in `std::vector`?

Answer:

- **`push_back()`:** Takes an already constructed object and copies or moves it into the vector.
- **`emplace_back()`:** Constructs the object *in-place* at the end of the vector, using the provided arguments. This can be more efficient as it avoids creating a temporary object.

Question 40: What is the erase-remove idiom?

Answer: The erase-remove idiom is a common pattern for removing elements from a container that satisfy a certain condition. It involves using `std::remove` or `std::remove_if` to move the elements to be kept to the beginning of the container, and then using the container's `erase` member function to remove the remaining elements.

Example for `std::vector`:

```
vec.erase(std::remove(vec.begin(), vec.end(), value_to_remove), vec.end());
```

Section 6: More Advanced Linked List Problems

Question 41: How do you find the Nth node from the end of a Singly Linked List?

Answer: Use the two-pointer approach:

1. Initialize two pointers, `p1` and `p2`, to the `head`.
2. Move `p1` `N` steps forward.
3. Now, move both `p1` and `p2` one step at a time until `p1` reaches the end of the list.
4. When `p1` is at the end, `p2` will be at the Nth node from the end.

Question 42: How do you merge two sorted Singly Linked Lists?

Answer: You can merge two sorted linked lists recursively or iteratively.

Iterative Approach:

1. Create a dummy `head` node for the merged list.
2. Use a `tail` pointer to build the new list.
3. Compare the current nodes of both lists and append the smaller one to the `tail`.
4. Move the pointer of the list from which the node was taken.
5. Repeat until one list is exhausted.
6. Append the remaining part of the other list.

Question 43: How do you add two numbers represented by Linked Lists?

Answer: Each node represents a digit, and the lists are in reverse order.

1. Initialize a `carry` variable to 0.
2. Create a dummy `head` for the result list.
3. Iterate through both lists simultaneously.
4. Calculate the `sum` of the current digits and the `carry`.
5. Create a new node with `sum % 10` and append it to the result list.
6. Update the `carry` to `sum / 10`.
7. Continue until both lists are traversed and the `carry` is 0.

Question 44: How do you copy a Linked List with a `random` pointer?

Answer: This requires a multi-pass approach, often using a hash map:

1. **Pass 1:** Iterate through the original list and create a copy of each node. Store the mapping between original nodes and copied nodes in a hash map (`std::unordered_map<Node*, Node*>`).

2. **Pass 2:** Iterate through the original list again. For each original node, set the `next` and `random` pointers of its corresponding copied node using the hash map.

Question 45: How do you flatten a multilevel Doubly Linked List?

Answer: A multilevel doubly linked list has a `child` pointer in addition to `next` and `prev`. To flatten it:

1. Use a recursive or iterative approach (with a stack).
2. Traverse the list. If a node has a `child`: a. Find the tail of the child list. b. Connect the tail of the child list to the `next` of the current node. c. Connect the `next` of the current node to the `child`. d. Set the `child` pointer to `nullptr`.

Question 46: How do you check if a Singly Linked List is a palindrome?

Answer:

1. Find the middle of the linked list.
2. Reverse the second half of the list.
3. Compare the first half with the reversed second half.
4. (Optional) Restore the list by reversing the second half again.

Question 47: How do you rotate a Linked List by K positions?

Answer:

1. Find the length of the list.
2. Connect the last node to the `head` to make it circular.
3. Find the new last node (at `length - k` position).
4. The new `head` will be the node after the new last node.
5. Break the circular link by setting the `next` of the new last node to `nullptr`.

Question 48: How do you remove the Nth node from the end of a list?

Answer: Use the two-pointer approach (as in finding the Nth node from the end), but keep track of the previous node of the slow pointer. When the fast pointer reaches the end, the slow pointer is at the node to be deleted. Use the previous pointer to bypass it.

Question 49: How do you swap nodes in pairs in a Linked List?

Answer: You can do this iteratively or recursively. Iteratively, you would process the list in pairs, swapping the `next` pointers of the two nodes in each pair and updating the link from the previous pair.

Question 50: How do you find the intersection point of two Singly Linked Lists?

Answer:

1. Find the lengths of both lists (`lenA` , `lenB`).
2. Find the difference in lengths (`diff`).
3. Move the pointer of the longer list by `diff` steps.
4. Now, traverse both lists simultaneously. The first node where the pointers are equal is the intersection point.

Section 7: More Advanced Vector Problems

Question 51: How would you implement a sparse vector?

Answer: A sparse vector (where most elements are zero) can be implemented efficiently by storing only the non-zero elements. Common approaches include:

- **`std::vector<std::pair<int, T>>`:** A vector of (index, value) pairs.
- **`std::map<int, T>` or `std::unordered_map<int, T>`:** A map where the key is the index and the value is the non-zero element.

Question 52: How do you find the kth largest element in an unsorted vector?

Answer:

- **Sorting:** Sort the vector (O(n log n)) and return the element at index `n - k`.
- **Min-Heap:** Maintain a min-heap of size `k`. Iterate through the vector, and for each element, if it's larger than the top of the heap, pop the top and push the new element. The final top of the heap is the kth largest element (O(n log k)).
- **Quickselect:** Use a partition-based algorithm similar to Quicksort. This has an average time complexity of O(n).

Question 53: How do you find a pair of elements in a vector that sum up to a given value?

Answer:

- **Brute Force:** Use two nested loops (O(n^2)).
- **Sorting:** Sort the vector. Use two pointers, one at the beginning and one at the end, and move them inwards based on whether the sum is too small or too large (O(n log n)).
- **Hash Set:** Iterate through the vector. For each element `x`, check if `target - x` exists in a hash set. If not, add `x` to the set (O(n)).

Question 54: How do you find the majority element in a vector (appears more than n/2 times)?

Answer: Use **Boyer-Moore Voting Algorithm**:

1. Initialize a `candidate` and a `count` to 0.
2. Iterate through the vector. If `count` is 0, set the `candidate` to the current element.
3. If the current element is the same as the `candidate`, increment `count`. Otherwise, decrement `count`.
4. The final `candidate` is the majority element.

This works in O(n) time and O(1) space.

Question 55: How do you find the length of the longest consecutive elements sequence in an unsorted vector?

Answer: Use a hash set for O(1) lookups:

1. Insert all elements into a hash set.
2. Iterate through the vector. For each element `x`, if `x - 1` is not in the hash set, it's the start of a new sequence.
3. From this starting point, count how many consecutive elements (`x + 1`, `x + 2`, ...) are in the hash set.
4. Keep track of the maximum length found.

This has an average time complexity of O(n).

Question 56: How do you rotate a vector by k positions?

Answer: You can use the `std::rotate` algorithm from `<algorithm>`:

```
std::rotate(vec.begin(), vec.begin() + k, vec.end());
```

Alternatively, you can use a three-reversal approach:

1. Reverse the first `k` elements.
2. Reverse the remaining `n - k` elements.
3. Reverse the entire vector.

Question 57: How do you merge overlapping intervals in a vector of intervals?

Answer:

1. Sort the intervals based on their start times.
2. Initialize a `merged` vector with the first interval.
3. Iterate through the remaining intervals. If the current interval overlaps with the last interval in `merged`, merge them by updating the end time of the last interval.

Otherwise, add the current interval to `merged`.

Question 58: How do you find the maximum sum of a contiguous subarray in a vector (Kadane's Algorithm)?

Answer: Use **Kadane's Algorithm**:

1. Initialize `max_so_far` and `current_max` to the first element.
2. Iterate through the vector from the second element.
3. For each element, `current_max = max(element, current_max + element)`.
4. Update `max_so_far = max(max_so_far, current_max)`.
5. Return `max_so_far`.

This works in O(n) time and O(1) space.

Question 59: How do you find the product of all elements in a vector except the element at the current index?

Answer: Use a two-pass approach without using division:

1. **Pass 1 (Prefix Products):** Create a result vector. For each index `i`, `result[i]` will be the product of all elements before `i`.
2. **Pass 2 (Suffix Products):** Iterate from the end of the vector. Maintain a `suffix_product` variable. For each index `i`, multiply `result[i]` by `suffix_product`, and then update `suffix_product`.

This works in O(n) time and O(1) extra space (if the result vector is not counted).

Question 60: How do you find the first missing positive integer in an unsorted vector?

Answer: Use the vector itself as a hash map:

1. Iterate through the vector. For each element `x`, if it's within the range `[1, n]`, try to place it at index `x - 1`.
2. Iterate through the modified vector. The first index `i` where `vec[i] != i + 1` indicates that `i + 1` is the first missing positive.

This works in O(n) time and O(1) space.

Section 8: Comparing `std::vector`, `std::list`, and `std::deque`

Question 61: Summarize the key differences between `std::vector`, `std::list`, and `std::deque`.

Answer:

| Feature | `std::vector` | `std::list` | `std::deque` | |---|---|---|---|**Memory** | Contiguous | Non-contiguous (nodes) | Non-contiguous (blocks) | | **Random Access** | O(1) | O(n) | O(1) | | **Insertion/Deletion (Middle)** | O(n) | O(1) | O(n) | | **Insertion/Deletion (Ends)** | Back: O(1), Front: O(n) | Both: O(1) | Both: O(1) | | **Iterator Invalidation** | High | Low | Moderate |

Question 62: When would you use `std::deque`?

Answer: `std::deque` (double-ended queue) is the best choice when you need a container that behaves like a queue (efficient insertions/deletions at both ends) but also requires fast random access. It's a good compromise between `std::vector` and `std::list`.

Question 63: Why is `std::sort` not applicable to `std::list`?

Answer: `std::sort` requires random-access iterators to perform its efficient sorting algorithm (typically Introsort). `std::list` only provides bidirectional iterators, which do not support the pointer arithmetic needed by `std::sort`. `std::list` has its own `sort()` member function that is optimized for linked lists.

Question 64: How does iterator invalidation differ between `std::vector`, `std::list`, and `std::deque`?

Answer:

- **`std::vector`:** Any insertion or deletion can invalidate all iterators if a reallocation occurs. Otherwise, it invalidates iterators at or after the point of modification.
- **`std::list`:** Iterators are only invalidated if the element they point to is deleted. Insertions do not invalidate any iterators.
- **`std::deque`:** Insertions/deletions at the ends do not invalidate iterators. Insertions/deletions in the middle invalidate all iterators.

Question 65: If you need to store a large number of elements and frequently insert/delete in the middle, which container is best?

Answer: `std::list` is the best choice for this scenario, as it provides O(1) insertion and deletion anywhere in the list, assuming you have an iterator to the position.

Section 9: Advanced Topics and Edge Cases

Question 66: What is a self-organizing list?

Answer: A self-organizing list is a type of linked list that reorders its nodes based on access patterns to improve average access time. Common strategies include:

- **Move-to-Front:** When an element is accessed, it is moved to the front of the list.
- **Transpose:** When an element is accessed, it is swapped with the element before it.

These are useful when some elements are accessed much more frequently than others.

Question 67: What is an unrolled linked list?

Answer: An unrolled linked list is a hybrid data structure that combines the features of a linked list and an array. Each node in the list contains a small array of elements instead of a single element. This improves cache performance by storing multiple elements contiguously, while still allowing for efficient insertions and deletions of entire blocks.

Question 68: How can you implement a queue using two stacks?

Answer:

- Use two stacks, `s1` (for enqueue) and `s2` (for dequeue).
- **Enqueue:** Push the element onto `s1`.
- **Dequeue:** If `s2` is empty, pop all elements from `s1` and push them onto `s2`. Then, pop from `s2`.

This gives amortized O(1) time complexity for both operations.

Question 69: How can you implement a stack using a single queue?

Answer:

- **Push:** Enqueue the new element. Then, dequeue all other elements and enqueue them back into the queue. This keeps the most recently added element at the front.
- **Pop:** Dequeue from the front of the queue.

This is inefficient (O(n) for push) but demonstrates the concept.

Question 70: What is a skip list?

Answer: A skip list is a probabilistic data structure that allows for fast search, insertion, and deletion operations (average O(log n)) in a sorted sequence. It's essentially a linked list with additional layers of

pointers that skip over some nodes, allowing for faster traversal.

Question 71: How do you handle memory management in a custom linked list implementation?

Answer: In a custom linked list implementation in C++, you must handle memory management manually:

- **Allocation:** Use `new Node(...)` to allocate memory for new nodes on the heap.

- **Deallocation:** Use `delete` to free the memory of a node when it is removed from the list. It is crucial to implement a destructor for the linked list class that iterates through all nodes and deletes them to prevent memory leaks when the list object goes out of scope.

Question 72: What is a

sentinel node in a linked list?

Answer: A sentinel node (also known as a dummy node) is a special node that does not hold any data but serves as a placeholder at the beginning (and sometimes end) of a linked list. Its purpose is to simplify the logic for insertion and deletion operations, especially at the boundaries (head or tail), by eliminating the need for special case handling for empty lists or single-node lists.

Question 73: How can you detect and remove a loop in a Singly Linked List?

Answer:

1. **Detection:** Use Floyd's Cycle-Finding Algorithm (tortoise and hare). If `slow` and `fast` pointers meet, a loop exists.
2. **Removal:** a. After `slow` and `fast` meet, move `slow` back to the `head`. b. Keep `fast` at the meeting point. c. Move both `slow` and `fast` one step at a time. The point where they meet again is the start of the loop. d. Traverse from the start of the loop until you reach the node just before it (the node whose `next` pointer points to the loop start). Set that node's `next` pointer to `nullptr`.

Question 74: How do you find the starting node of a loop in a Singly Linked List?

Answer: After detecting a loop using Floyd's Cycle-Finding Algorithm (where `slow` and `fast` meet), reset `slow` to the `head` of the list. Then, move both `slow` and `fast` one step at a time. The node where they meet again is the starting node of the loop.

Question 75: How do you check if two Singly Linked Lists intersect?

Answer:

1. Traverse the first list to its end and get its length.
2. Traverse the second list to its end and get its length.
3. If their end nodes are different, they do not intersect.
4. If they are the same, an intersection exists. To find the intersection point, move the pointer of the longer list forward by the difference in lengths. Then, traverse both lists simultaneously; the first common node is the intersection point.

Question 76: How do you remove duplicates from an unsorted Singly Linked List?

Answer:

1. **Using a Hash Set:** Iterate through the list. For each node, check if its data is already in a hash set. If not, add it to the set and keep the node. If it is, remove the node. This is O(n) on average.
2. **Without Extra Space (Two Pointers):** Use two pointers, `current` and `runner`. `current` iterates through the list. For each `current` node, `runner` iterates from `current`'s next to find and remove duplicates of `current->data`. This is O(n^2).

Question 77: How do you swap nodes in a Linked List without swapping data?

Answer: Swapping nodes means changing the `next` (and `prev` for doubly linked lists) pointers so that the nodes themselves are reordered, rather than just exchanging their data values. This involves careful manipulation of pointers to the nodes before, between, and after the nodes being swapped.

Question 78: How do you merge two sorted Doubly Linked Lists?

Answer: Similar to merging singly linked lists, but you also need to manage the `prev` pointers. You can iteratively compare nodes from both lists, appending the smaller one to the result list and updating both `next` and `prev` pointers accordingly.

Question 79: What is the purpose of `std::vector::shrink_to_fit()` and when should it be used?

Answer: `std::vector::shrink_to_fit()` is a non-binding request to reduce the vector's `capacity()` to match its `size()`. It attempts to deallocate any excess memory that was reserved but is no longer needed. It should be used when:

- **Memory optimization is critical:** In memory-constrained environments or for long-lived vectors that have significantly shrunk.
- **The vector's size has stabilized:** After a period of growth and subsequent reduction in size, and you don't expect further significant additions.

It's a non-binding request because the implementation might choose not to deallocate memory if it deems it inefficient or if the system is under memory pressure. It can be an expensive operation as it might involve a reallocation and copying of elements.

Question 80: How does `std::vector` handle exceptions during element construction or assignment?

Answer: `std::vector` provides strong exception safety for most operations. If an exception occurs during the construction or assignment of an element (e.g., during a reallocation and copy/move of elements to a new memory block), `std::vector` guarantees that:

- **No memory leaks:** All memory that was allocated will be properly deallocated.
- **No data corruption:** The vector will remain in a valid state, often reverting to its state before the failed operation. This is typically achieved by performing operations on a temporary buffer and only swapping with the original vector if successful (copy-and-swap idiom).

However, if the element's move constructor is not `noexcept`, `std::vector` might fall back to using the copy constructor during reallocations to maintain strong exception safety, which can impact performance.

Question 81: Can `std::vector` store objects of different types? If not, how can you achieve similar functionality?

Answer: No, `std::vector` can only store objects of a single, homogeneous type. All elements in a `std::vector<T>` must be of type `T` (or a type implicitly convertible to `T`).

To achieve similar functionality (storing objects of different types), you can use:

1. **Pointers to a common base class (polymorphism):** Store `std::vector<Base*>` or, preferably, `std::vector<std::unique_ptr<Base>>` or `std::vector<std::shared_ptr<Base>>`. This allows you to store objects of different derived classes that inherit from a common base class.
2. **`std::variant` (C++17):** If the set of possible types is known and fixed, `std::vector<std::variant<Type1, Type2, ...>>` can store objects of different types in a type-safe union.
3. **`std::any` (C++17):** If the types are arbitrary and not known beforehand, `std::vector<std::any>` can store objects of any copy-constructible type, but it comes with runtime type checking overhead.

Question 82: What is the role of an allocator in `std::vector`?

Answer: An allocator is an object that `std::vector` (and other standard library containers) uses to manage memory. It encapsulates the details of memory allocation and deallocation. By default, `std::vector` uses `std::allocator`, which uses `new` and `delete` internally.

Advanced users can provide custom allocators to `std::vector` to:

- **Control memory allocation:** Use custom memory pools, stack-based allocation, or specific hardware memory.
- **Integrate with existing memory management systems.**
- **Track memory usage.**

Question 83: Explain the difference between `std::vector::push_back()` and `std::vector::insert()`.

Answer:

- **`push_back(const T& value)` / `push_back(T&& value)`:** Adds an element to the *end* of the vector. This operation has amortized O(1) time complexity, as it only requires reallocation when the capacity is exhausted.
- **`insert(iterator pos, const T& value)` / `insert(iterator pos, T&& value)`:** Inserts an element at a *specified position* (`pos`) within the vector. This operation has O(n) time complexity because all elements from `pos` to the end of the vector must be shifted to make room for the new element.

`push_back()` is generally preferred for adding elements when order doesn't matter or when adding to the end, due to its efficiency.

Question 84: How would you implement a dynamic array that grows by a fixed amount (e.g., 10 elements) instead of doubling its capacity?

Answer: To implement a dynamic array that grows by a fixed amount, you would modify the reallocation logic. Instead of multiplying the `capacity` by 2, you would add a fixed increment (e.g., 10) to the `capacity` when a reallocation is needed.

Conceptual Change in `reallocate()`:

```
void reallocate() {
    int new_capacity = _capacity + FIXED_INCREMENT; // e.g., 10
    if (_capacity == 0) new_capacity = INITIAL_CAPACITY; // Handle initial case
    T* new_arr = new T[new_capacity];
    // ... copy elements ...
    // ... delete old array ...
    arr = new_arr;
    _capacity = new_capacity;
}
```

Consequences: This approach would lead to more frequent reallocations compared to doubling, potentially degrading performance, especially for large vectors, as each reallocation is an O(n) operation. The amortized constant time guarantee of `push_back()` would no longer hold.

Question 85: How can you use `std::vector` to represent a stack and a queue, and what are the performance implications?

Answer:

- **Stack (LIFO - Last In, First Out):**
  - **Push:** `vector::push_back()` (Amortized O(1))
  - **Pop:** `vector::pop_back()` (O(1))
  - **Top:** `vector::back()` (O(1))
  - **Performance:** `std::vector` is an excellent choice for implementing a stack due to efficient operations at the back.
- **Queue (FIFO - First In, First Out):**
  - **Enqueue:** `vector::push_back()` (Amortized O(1))
  - **Dequeue:** `vector::erase(vector::begin())` (O(n))
  - **Front:** `vector::front()` (O(1))
  - **Performance:** `std::vector` is generally a poor choice for implementing a queue if frequent dequeuing from the front is required, because `erase(begin())` involves shifting all remaining elements, leading to O(n) complexity. `std::deque` is a much better choice for queues as it provides O(1) `push_front()` and `pop_front()`.

Question 86: Describe a real-world scenario where `std::vector` would be the ideal data structure.

Answer: A common real-world scenario where `std::vector` is ideal is managing a collection of game objects in a game engine. For example, a list of active enemies, projectiles, or particles. In such a scenario:

- **Dynamic number of objects:** Enemies are spawned and destroyed, so the collection needs to grow and shrink.
- **Frequent iteration:** The game loop needs to iterate through all active objects to update their state or render them.

- **Random access:** Sometimes, a specific enemy might need to be accessed by an ID or index.
- **Additions/Deletions at end:** New projectiles are often added to the end, and destroyed enemies can be efficiently removed by swapping with the last element and then `pop_back()`.
- **Memory locality:** Storing objects contiguously improves cache performance during iteration.

Question 87: How would you use `std::vector` to implement a simple image representation (e.g., grayscale image)?

Answer: A grayscale image can be represented as a 2D grid of pixel intensity values. You can use `std::vector<std::vector<unsigned char>>` to represent this, where the outer vector represents rows and the inner vectors represent columns (pixels in each row). Each `unsigned char` would store the intensity value (0-255).

Example:

```
 #include <iostream>
#include <vector>

int main() {
    const int width = 10;
    const int height = 5;

    // Create a 5x10 grayscale image, initialized to black (0)
    std::vector<std::vector<unsigned char>> image(height, std::vector<unsigned
char>(width, 0));

    // Set some pixels to white (255)
    image[0][0] = 255;
    image[2][5] = 255;
    image[4][9] = 255;

    // Print a simplified representation of the image
    for (int r = 0; r < height; ++r) {
        for (int c = 0; c < width; ++c) {
            std::cout << (image[r][c] > 0 ? '#' : '.') << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Question 88: You have a large `std::vector` of custom objects. What considerations would you have for its performance?

Answer: For a large `std::vector` of custom objects, performance considerations include:

1. **Object Size and Copy/Move Cost:** If the custom objects are large or have expensive copy/move constructors, frequent reallocations will be very costly. Ensure efficient move constructors are available or use `std::unique_ptr` if ownership semantics allow.
2. **`reserve()` Usage:** Pre-allocate memory using `reserve()` to minimize reallocations if the final size is predictable.
3. **`emplace_back()` vs. `push_back()`:** Use `emplace_back()` to construct objects in-place and avoid temporary object creation and copy/move overhead.
4. **Cache Locality:** Contiguous storage is generally good for cache performance during iteration. However, if objects themselves contain pointers to scattered heap memory, this benefit might be reduced.
5. **Algorithm Choice:** Be mindful of algorithms that involve frequent insertions/deletions in the middle (O(n)), as these will be slow.
6. **Destruction Cost:** When the vector is destroyed or cleared, the destructors of all contained objects will be called. For very large numbers of complex objects, this can take time.

Question 89: How would you implement a circular buffer using `std::vector`?

Answer: A circular buffer (or ring buffer) can be implemented using `std::vector` by maintaining a fixed-size vector and two indices: `head` and `tail`. `head` points to the next element to be read, and `tail` points to the next available slot for writing. When either index reaches the end of the vector, it wraps around to the beginning using the modulo operator (`%`).

Key operations:

- **Enqueue:** Add element at `tail`, increment `tail`. If `tail` reaches end, `tail = 0`. Handle overflow (e.g., overwrite oldest element or return error).

- **Dequeue:** Read element at `head`, increment `head`. If `head` reaches end, `head = 0`. Handle underflow (empty buffer).

Example (simplified):

```cpp
#include <iostream>
#include <vector>

template <typename T>
class CircularBuffer {
private:
    std::vector<T> buffer_;
    size_t head_;
    size_t tail_;
    size_t capacity_;
    size_t size_;

public:
    CircularBuffer(size_t capacity) : buffer_(capacity), head_(0), tail_(0),
capacity_(capacity), size_(0) {}

    bool enqueue(const T& item) {
        if (size_ == capacity_) {
            // Buffer is full, handle overflow (e.g., overwrite oldest)
            std::cout << "Buffer full, overwriting oldest element.\n";
            head_ = (head_ + 1) % capacity_;
        } else {
            size_++;
        }
        buffer_[tail_] = item;
        tail_ = (tail_ + 1) % capacity_;
        return true;
    }

    bool dequeue(T& item) {
        if (size_ == 0) {
            std::cout << "Buffer empty.\n";
            return false;
        }
        item = buffer_[head_];
        head_ = (head_ + 1) % capacity_;
        size_--;
        return true;
    }

    size_t size() const { return size_; }
    bool empty() const { return size_ == 0; }
};

int main() {
    CircularBuffer<int> cb(3);

    cb.enqueue(10);
    cb.enqueue(20);
    cb.enqueue(30);
    cb.enqueue(40); // Overwrites 10

    int val;
    while (!cb.empty()) {
        cb.dequeue(val);
        std::cout << "Dequeued: " << val << std::endl;
    }

    return 0;
}
```

Question 90: What is the purpose of `std::vector::get_allocator()`?

Answer: The `std::vector::get_allocator()` member function returns a copy of the allocator object associated with the vector. This can be useful if you are using a custom allocator and need to access its state or perform operations directly through the allocator (e.g., allocating raw memory or constructing objects using the allocator's `construct` method).

For most common use cases with the default `std::allocator`, this function is rarely used directly by application code.

Question 91: How would you implement a dynamic array that supports efficient insertion/deletion at both ends (like a `std::deque`) using `std::vector` as a base?

Answer: While `std::vector` is not ideal for efficient insertions/deletions at the front (due to O(n) shifting), you could conceptually build a `deque`-like structure on top of it by:

1. **Using a `std::vector`** as the underlying storage.
2. **Maintaining a `start_index` and `end_index`** within the vector to define the active range of elements.
3. **For `push_back`:** Add to `end_index`, expand vector if needed.
4. **For `pop_back`:** Decrement `end_index`.
5. **For `push_front`:** This is the tricky part. If there's space before `start_index`, you can decrement `start_index` and insert. If not, you'd have to shift all elements to the right to make space at the beginning, which is O(n). Alternatively, you could reallocate and copy elements to the middle of a new, larger buffer to create space at both ends.
6. **For `pop_front`:** Increment `start_index`.

This approach would still suffer from O(n) complexity for `push_front` and `pop_front` if the `start_index` hits the beginning of the underlying array and a shift is required. This is why `std::deque` uses a more complex block-based memory management to achieve O(1) at both ends.

Question 92: Explain the concept of vectorization in the context of high-performance computing, and how it relates to the contiguous memory of `std::vector`.

Answer: Vectorization (or SIMD - Single Instruction, Multiple Data) is a form of parallel computing where a single instruction is applied to multiple data points simultaneously. Modern CPUs have special vector registers and instructions that can perform operations (like addition, multiplication) on multiple data elements (e.g., 4, 8, or 16 integers or floats) in a single clock cycle.

The contiguous memory layout of `std::vector` is crucial for effective vectorization. Because the elements are stored next to each other in memory, the CPU can efficiently load a block of elements into a vector register and apply a single instruction to all of them. This leads to significant performance improvements in computationally intensive tasks like scientific computing, image processing, and machine learning.

If the data were stored non-contiguously (as in a `std::list`), the CPU would have to perform multiple memory fetches to gather the data, making vectorization impossible or very inefficient.

Question 93: How can you safely modify a `std::vector` while iterating over it?

Answer: Modifying a `std::vector` while iterating over it is dangerous due to iterator invalidation. If you need to remove elements during iteration, the safest approach is to use the erase-remove idiom or a traditional for loop with careful index/iterator management.

Safe Method 1: Erase-Remove Idiom (Best for removing multiple elements)

```
v.erase(std::remove_if(v.begin(), v.end(), condition), v.end());
```

Safe Method 2: Traditional for loop with iterator management

```
for (auto it = v.begin(); it != v.end(); ) {
    if (condition_to_remove(*it)) {
        it = v.erase(it); // erase() returns an iterator to the next valid
element
    } else {
        ++it;
    }
}
```

Unsafe Method (Avoid): A range-based for loop should not be used to erase elements, as it can lead to undefined behavior due to iterator invalidation.

Question 94: What is the difference between `std::vector::at()` and `std::vector::operator[]` in terms of performance?

Answer: In terms of performance, `std::vector::operator[]` is generally slightly faster than `std::vector::at()`. This is because `operator[]` does not perform bounds checking, while `at()` does. The bounds check in `at()` adds a small amount of overhead (typically a single comparison and a potential branch).

However, this performance difference is usually negligible in most applications. The choice between them should be based on safety requirements:

- Use `at()` when you need to ensure the index is valid and want to handle out-of-bounds access gracefully with exceptions.
- Use `operator[]` in performance-critical code where you are certain that the index will always be within the valid range.

Question 95: How would you flatten a `std::vector<std::vector<T>>` into a `std::vector<T>`?

Answer: To flatten a 2D vector into a 1D vector, you can iterate through the outer vector and then through each inner vector, adding each element to the flattened vector.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<std::vector<int>> matrix = {{1, 2}, {3, 4, 5}, {6}};
    std::vector<int> flattened_vector;

    for (const auto& row : matrix) {
        flattened_vector.insert(flattened_vector.end(), row.begin(),
row.end());
    }

    std::cout << "Flattened vector: ";
    for (int x : flattened_vector) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Question 96: What is the purpose of `std::vector::data()` and how does it relate to C-style APIs?

Answer: `std::vector::data()` returns a direct pointer to the first element of the underlying contiguous array used by the vector. This is crucial for interoperability with C-style APIs or libraries that expect a raw pointer to an array of data. It allows you to pass the contents of a `std::vector` to functions that were written to work with C-style arrays, without having to manually copy the data.

Question 97: How can you create a `std::vector` from a C-style array?

Answer: You can create a `std::vector` from a C-style array using the range constructor, which takes two pointers (or iterators) defining the beginning and end of the range to be copied.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    int c_array[] = {10, 20, 30, 40, 50};
    size_t array_size = sizeof(c_array) / sizeof(c_array[0]);

    // Create a vector from the C-style array
    std::vector<int> myVector(c_array, c_array + array_size);

    std::cout << "Vector created from C-style array: ";
    for (int x : myVector) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Question 98: What are the potential issues with storing raw pointers in a `std::vector`?

Answer: Storing raw pointers (`T*`) in a `std::vector` can lead to several issues:

1. **Memory Leaks:** The `std::vector` owns the pointers, but not the objects they point to. When the vector is destroyed, it will destroy the pointers but not call `delete` on the objects they point to, leading to memory leaks.
2. **Dangling Pointers:** If the object pointed to is deleted elsewhere, the pointer in the vector becomes a dangling pointer, leading to undefined behavior if dereferenced.
3. **Ownership Ambiguity:** It becomes unclear who is responsible for deleting the objects. This can lead to double-deletion errors or memory leaks.

Solution: Use smart pointers (`std::unique_ptr` or `std::shared_ptr`) to manage the lifetime of the objects automatically and provide clear ownership semantics.

Question 99: How does `std::vector` handle alignment of its elements?

Answer: `std::vector` correctly handles the alignment of its elements. When it allocates memory, it ensures that the memory is suitably aligned for the type `T` being stored. This is typically handled by the underlying memory allocation mechanism (`new`) and the allocator used by the vector. This ensures that even for types with strict

alignment requirements (e.g., for SIMD operations), the elements are stored correctly in memory.

Question 100: Can you have a `std::vector` of `std::vector`s? What are the performance implications?

Answer: Yes, you can have a `std::vector` of `std::vector`s (`std::vector<std::vector<T>>`), which is a common way to represent a 2D matrix or a jagged array.

Performance Implications:

- **Memory Fragmentation:** Each inner vector is allocated separately on the heap, so the memory for the entire 2D structure is not contiguous. This can lead to poorer cache performance compared to a single, flattened 1D vector representing the 2D data.
- **Overhead:** Each inner vector has its own size, capacity, and pointer overhead.
- **Reallocations:** Reallocating the outer vector is relatively cheap (moving the inner vector objects), but reallocating an inner vector can be expensive if it contains many elements.

For performance-critical applications with dense matrices, a single `std::vector` with manual index calculation (`index = row * num_cols + col`) is often preferred for better memory locality.