Linked List Assignment Interview Notes

Vector Overview

Generic Overview

A vector is a dynamic array that allows storing and accessing elements of the same data type. It is a container class provided by the Standard Template Library (STL) in C++. Vectors provide several advantages over traditional arrays, such as automatic memory management, resizable capacity, and built-in functions for efficient manipulation.

Internally, a vector is implemented as a contiguous block of memory that stores the elements. It dynamically manages its size, expanding or shrinking as needed. Elements are accessed using zero-based indexing, similar to arrays. The vector class provides various member functions and operators for performing common operations like adding elements at the end, inserting elements at specific positions, removing elements, and accessing elements by index. Vectors offer efficient random access, constant time complexity for accessing elements, and amortized constant time complexity for adding or removing elements at the end.

How Are Vectors Different From Arrays?

Vectors and arrays are both used to store collections of elements in C++, but vary in some areas:

- Arrays are fixed-size collections with a static size determined at compile-time, while vectors are dynamic arrays with a flexible and resizable size.
- Arrays require manual memory management and cannot be easily resized, whereas vectors handle memory allocation and deallocation automatically, allowing for dynamic resizing.
- Arrays provide direct and efficient access to elements using zero-based indexing, while vectors offer convenient member functions like 'push_back()' and 'pop_back()' for adding and removing elements.
- Arrays have a fixed memory layout, resulting in slightly faster element access, while vectors provide more convenience and flexibility.
- Arrays cannot be easily copied or assigned to another array, while vectors can be easily copied of assigned to other vectors.

In summary, arrays are suitable when the size is known and fixed, and direct memory control is needed. Vectors, on the other hand, are preferred when the size needs to be dynamic, convenient operations are required, and automatic memory management is desired.

What Is The Algorithm For Adding And Removing Elements From A Vector?

To add elements to a vector in C++, we use:

- To add an element at the end of the vector, we use the 'push_back()' function, which appends the element to the end of the vector.
- To insert an element at a specific position within the vector, we use 'insert()' function. This function takes an iterator pointing to the position and the element to be inserted.

To remove elements from a vector in C++, we use:

- To remove the last element of the vector, we use the 'pop_back()' function, which removes the element from the end of the vector.
- If you want to remove an element from a specific position within the vector, we use the 'erase()' function. It takes an iterator pointing to the position and removes the element at that position.

These operations provide a convenient and efficient way to add and remove elements from a vector, maintaining the integrity and resizing the vector dynamically as needed.

What Are The Benefits Of Using A Vector Over An Array?

The benefits of using a vector over an array are:

- Dynamic Size: Vectors provide dynamic resizing, allowing you to add or remove elements at runtime. Unlike arrays, which have a fixed size, vectors automatically handle memory allocation.
- Convenience: Vectors come with built-in functions like 'push_back()', 'pop_back()', and 'insert()', making it easier to add, remove, or insert elements without manual management. Vectors also provide member functions like 'size()' and 'empty()' to retrieve information about the number of elements and check if the vector is empty.
- Bounds Checking: Vectors perform bounds checking on element access, ensuring that you do not go out of bounds. This helps prevent accessing memory beyond the allocated size and reduces the risk of errors and crashes that can occur with arrays.
- Iteration Support: Vectors can be easily iterated using range-based for loops or iterators, providing a convenient way to traverse and manipulate elements.
- Copying and Assignment: Vectors can be easily copied and assigned to other vectors using the copy constructor and assignment operator, allowing for efficient and convenient data handling.

Overall, vectors offer the flexibility of dynamic resizing, convenient functions, bounds checking, iteration support, and ease of copying, making them a preferred choice over arrays in many scenarios.

What Is The Basic Algorithm For Iterating Through A Vector?

The simple algorithm for iterating through the elements of a vector is to use a for-loop. Starting the index at 0, and going up to the size of the vector, we can use built in functions like 'at()' to indicate which index we are wanting to examine / manipulate. Using this simple algorithm, we check for certain conditions within our vector, look for specific elements, and many other things as well.

Linked List Overview

Generic Overview

A linked list is a data structure commonly used in object-oriented programming that consists of nodes connected through pointers. Each node contains a value and a pointer to the next node in the list, except for the last node which points to 'nullptr'. The linked list does not have a specific 'head' or 'tail' node, but rather, the first node in the list is often referred to as the 'head' and the last node is commonly known as the 'tail'. The nodes in between the 'head' and 'tail' are linked in a sequential manner. The 'head' node serves as the starting point for traversing the list, and each node points to the next node, allowing efficient insertion and removal operations.

Difference Between An Array And Vector

Linked lists differ from arrays and vectors in several ways. One significant distinction is that the nodes in a linked list, which correspond to elements in an array or vector, contain pointers that indicate the next node in the list. In contrast, elements in arrays and vectors do not have pointers pointing to the next element. Unlike arrays and vectors, linked lists cannot be accessed using an index value. However, similar to vectors, linked lists have the potential to dynamically resize after compilation. It's important to note that the mechanism of size change in a linked list is different from that of vectors. Some other key differences between linked lists and vectors and arrays are:

- Memory Allocation: Linked lists dynamically allocate memory for each node as it is needed. This allows for efficient memory usage, as nodes can be allocated and deallocated independently. In contrast, arrays and vectors allocate a fixed block of memory upfront, regardless of the number of elements, which can clead to potential wastage of insufficient memory.
- Insertion and Deletion: Linked lists excel in insertion and deletion operations, especially in the middle of the list. Inserting or removing a node in a linked list only requires updating the pointers, whereas in arrays and vectors, these operations may involve shifting elements, resulting in less efficiency.
- Random Access: Arrays and vectors support direct and efficient random access using index-based access. You can access any element in constant time with their respective indices. Linked lists, on the other hand, do not provide direct index-based access, requiring traversal from the head to the desired node, resulting in linear time complexity.
- Memory Overhead: Linked lists have a higher memory overhead compared to arrays and vectors due to the additional memory required for storing the pointers. Each node in a linked list needs to store a pointer to the next node, increasing the overall memory usage.

- Iteration efficiency: Arrays and vectors offer efficient iteration using loops with index-based access. Since the elements are stored in contiguous memory locations, sequential access is faster. Linked lists , on the other hand, require traversing each node sequentially, resulting in slower iteration.
- Memory Fragmention: Linked lists are less prone to memory fragmentation compared to arrays and vectors. As nodes are dynamically allocated, they cam be scattered in memory, reducing the likelihood of large blocks of unallocated space.

What Is A Node In A Linked List?

A node in a linked list is analogous to an element in an array or vector. In a linked list, a node contains data and a reference (or pointer) to the next node in the list.

- Data Component: Represents the value or information stored in the node
- Reference Component: Points to the next node in the linked list with the use of a pointer

A node is analogous to a link in a chain. Each addition of a node will elongate the total length of the chain.

Where Are Elements Accessed And Modified?

The main examples of accessing and modifying elements in a linked list are the following:

- Accessing the Value of a Node: We use the 'data' member to access the value stored in a node of a linked list.
- Modifying the Value of a Node: To modify the value of a node, we use the syntax: 'nodePtr->data = newValue', or some variation like this.
- Traversing the Linked List: To traverse through the linked list, we use a while loop to do so.
- Inserting a New Node: We use the 'Insert' functions defined in the program to insert nodes into the list.
- Removing a Node From the List: We use the 'Remove' functions defined in the program to remove nodes from the list.

Accessing & Modifying Elements

We access and modify elements in a linked list with function calls and the like below:

```
\\ Accessing the values of a node and traversing a linked list
\\ like in the Contains function below:
bool LinkedList::Contains(int data){
    shared_ptr<node> currPtr = top_ptr_;
    bool ret = false;
    while (currPtr != nullptr) {
    if (currPtr->data == data) {
        ret = true;
    }
    else {}
    currPtr = currPtr->next;
    }
    return ret;
}

\\ Modifying the value of a node like in the AppendData function below:
void LinkedList::AppendData(int data){
    shared_ptr<node> nxtPtr(new node);
    nxtPtr->data = data;
    nxtPtr->next = nullptr;
    if (top_ptr_ == nullptr) {
    SetTop(nxtPtr);
    }
    else {
    shared_ptr<node> currPtr = top_ptr_;
    while (currPtr->next != nullptr) {
        currPtr = currPtr->next;
    }
```

```
        currPtr->next = nxtPtr;
        }
    }

    \\ Inserting a new node into the linked list like in the Insert function below:
    void LinkedList::Insert(int offset, shared_ptr<node> new_node){
        shared_ptr<node> currPtr = top_ptr_;
        vector<shared_ptr<node>> ptrs;
        while (currPtr != nullptr) {
        ptrs.push_back(currPtr);
        currPtr = currPtr->next;
        }
        new_node->data;
        new_node->next = nullptr;
        if (offset == 0) {
        new_node->next = top_ptr_;
        SetTop(new_node);
        }
        else {
        currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
            new_node->next = currPtr->next;
            currPtr->next = new_node;
            }
            else {}
            currPtr = currPtr->next;
            }
        }
    }

    \\ Removing a node from the linked list like in the Remove function below:
    void LinkedList::Remove(int offset){
        shared_ptr<node> currPtr = top_ptr_;
        vector<shared_ptr<node>> ptrs;
        while (currPtr != nullptr) {
        ptrs.push_back(currPtr);
        currPtr = currPtr->next;
        }
        if (offset == 0) {
        SetTop(top_ptr_->next);
        }
        else {
        currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
            currPtr->next = currPtr->next->next;
            break;
            }
            else {}
            currPtr = currPtr->next;
            }
        }
    }
```

Initializing A Linked List

When the linked list is constructed, we initialize the data member 'next' to be null. To initialize the list with values, we call the 'InitNode' function that is defined in the CPP file from the project. This algorithm can be seen below:

Value Initialization

```
/*  InitNode - This function takes in an integer data type and assigns that value to a "node"
data type's "data" member
*   Input:
*     data - This is an integer data type that is later assigned to a "node" data type's "data"
member
*   Algorithm:
*     shared_ptr<node> ret(new node); - This line creates a smart pointer of data type "node"
named "ret"
*   Output:
*     ret - After setting "ret"'s data member to the input parameter "data"
*     The "next" data member of the "node" object is initially set to nullptr, indicating the
absence of a next node
*/
shared_ptr<node> LinkedList::InitNode(int data){
    shared_ptr<node> ret(new node);
    ret->data = data;
    return ret;
}
```

Adding Elements To A Linked List

Below is the algorithm for adding elements to a linked list found in this class.

Insert Algorithm

```
/*  Insert - This function inserts a new node into a linked list at the specified offset
*   Input:
*     offset - An integer that is the offset at which the new node should be inserted
*     new_node - A smart pointer of object node that is to be inserted into the linked list
*   Algorithm:
*     First we create a new smart pointer called "currPtr" and set it equal to the top of
the linked list
*     After that we create a vector called "ptrs" to store the smart pointers to each node
in the linked list
*     Then we traverse the linked list and add each node's smart pointer to the "ptrs" vector
*     After that, we check if the offset is zero:
*       If it is zero, set the next pointer of the "new_node" node to the current top_ptr_
*       Set the top_ptr_ to the "new_node" node, making it the new first node in the
linked list
*     If the offset is not zero:
*       Reset "currPtr" to the top_ptr_ and iterate through the "ptrs" vector
*       When the current index matches the offset - 1, set the next pointer of "new_node" to
currPtr->next
*       Update the next pointer of "currPtr" to "insertPtr", inserting it at the desired
offset
*   Output:
*     This function does not return a value
*/
void LinkedList::Insert(int offset, shared_ptr<node> new_node){
    shared_ptr<node> currPtr = top_ptr_;
    vector<shared_ptr<node>> ptrs;
    while (currPtr != nullptr) {
    ptrs.push_back(currPtr);
    currPtr = currPtr->next;
    }
    new_node->data;
    new_node->next = nullptr;
    if (offset == 0) {
    new_node->next = top_ptr_;
    SetTop(new_node);
```

```
        }
        else {
        currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
            new_node->next = currPtr->next;
            currPtr->next = new_node;
            }
            else {}
            currPtr = currPtr->next;
            }
        }
    }
```

Removing Elements From A Linked List

Below is the algorithm for removing elements from a linked list found in this class.

Remove Algorithm

```
    /*  Remove - This function removes a node from a linked list at the specified offset
    *   Input:
    *     offset - An integer that is the offset of the node to be removed from our linked list
    *   Algorithm:
    *     First we create a new smart pointer called "currPtr" and set it equal to the top of the
    linked list
    *     Then we create a vector called "ptrs" to store the smart pointers to each node in the
    linked list
    *     After that we iterate through the linked list and add each node's smart pointer to the
    "ptrs" vector
    *     Then we heck if the offset is zero.
    *       If it is zero, set the next pointer of the top node to the node after it
    *     Otherwise, reset "currPtr" to the top node and iterate through the "ptrs" vector
    *       When the current index matches the offset - 1, set the next pointer of "currPtr" to
    the node after it
    *       Break out of the for loop to avoid segmentation faults
    *     Output:
    *       This function does not return a value.
    */
    void LinkedList::Remove(int offset){
        shared_ptr<node> currPtr = top_ptr_;
        vector<shared_ptr<node>> ptrs;
        while (currPtr != nullptr) {
        ptrs.push_back(currPtr);
        currPtr = currPtr->next;
        }
        if (offset == 0) {
        SetTop(top_ptr_->next);
        }
        else {
        currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
            currPtr->next = currPtr->next->next;
            break;
            }
            else {}
            currPtr = currPtr->next;
            }
        }
    }
```

Finding Elements In A Linked List

Below is the algorithm for finding elements in a linked list found in this class:

Search Algorithm

```
/*  Contains - This function determines if an element is present inside a linked list
*   Input:
*     data - This is an integer that is being searched for in our linked list
*   Algorithm:
*     First, create a smart pointer called "currPtr" and initialize it to the head of
our linked list
*     Initialize the return value "ret" to false, so that if a value is not found we do
not need to change it to false
*     Traverse through the linked list with the use of a while loop and check if the
current nodes "data" member matches
*     the input parameter "data"
*       If the two are a match, change the value of "ret" to true, otherwise, leave it
as false
*   Output:
*     ret - Boolean value indicating if a value was found in the linked list
*/
bool LinkedList::Contains(int data){
    shared_ptr<node> currPtr = top_ptr_;
    bool ret = false;
    while (currPtr != nullptr) {
    if (currPtr->data == data) {
        ret = true;
    }
    else {}
    currPtr = currPtr->next;
    }
    return ret;
}
```

Returning The Size Of A Linked List

Below is the algorithm for determining the size of a linked list found in this class:

Size Algorithm

```
/*  Size - This function counts how many nodes are present in our linked list
*   Algorithm:
*     We first create a smart pointer called "currPtr" and set it equal to our "top_ptr_"
in our linked list
*     We initialize the return value "ret" to zero
*     Then we enter a while loop that increments "ret" by one until it runs into the tail
of the list
*     Advance the current pointer object using "currPtr = currPtr->next;"
*   Output:
*     ret - Integer value that represents the number of nodes that are present in our
linked list
*/
int LinkedList::Size(){
    shared_ptr<node> currPtr = top_ptr_;
    int ret = 0;
    while (currPtr != nullptr) {
    ret++;
    currPtr = currPtr->next;
    }
```

```
    return ret;
}
```

Reversing A Linked List

The easiest way to reverse a linked list is to have a doubly linked list. With a singly linked list, the simplest way to reverse the linked list is to create three separate nodes: 'previous', 'current', and 'next'. Once these nodes have been created, we traverse through the linked list and update their values accordingly. This algorithm can be seen below:

Reverse Algorithm

```
void LinkedList::Reverse() {
    if (top_ptr_ == nullptr || top_ptr_->next == nullptr) {
        // Empty list or single node, no need to reverse
        return;
    }

    shared_ptr<node> previous = nullptr;
    shared_ptr<node> current = top_ptr_;
    shared_ptr<node> next = nullptr;

    while (current != nullptr) {
        // Find the next node
        next = current->next;

        // Reverse the connection of the current node
        current->next = previous;

        // Update the previous and current nodes
        previous = current;
        current = next;
    }
    // Update the top of the linked list
    top_ptr_ = previous;
}
```

Concatenating A Linked List

Concatenating a linked list pertains to merging two linked lists. In the simplest terms, to do this, we simply set the tail of a linked list to the head of the other linked list. Algorithmically, this can be seen below:

Concatenation Algorithm

```
void LinkedList::Concatenate(LinkedList& secondList) {
    // Handle the base case where the top node is null of the initial list
    if (top_ptr_ == nullptr) {
        top_ptr_ = secondList.GetTop();
    } else {
        shared_ptr<node> current = top_ptr_;
        // Find the tail of the current linked list
        while (current->next != nullptr) {
            current = current->next;
        }
        // Setting the tail of the current list to the top node of the second list
        current->next = secondList.GetTop();
    }
    // Clear the second linked list
    secondList.SetTop(nullptr);
}
```

Ascending Order In A Linked List

To sort the data in a linked list in ascending order, we do something similar to what we do in a reverse algorithm in linked lists. We first have to create two separate nodes: 'current' and 'previous', and update them accordingly with an if statement. This is done in the presence of a while loop that is nested in a do-while loop. Algorithmically, this can be seen below.

Ascending Order Algorithm

```
void LinkedList::Sort() {
    // Handle the base case of the
    if (top_ptr_ == nullptr || top_ptr_->next == nullptr) {
        return;
    }
    bool swapped;
    // Execute a do-while loop process to make sure the sorting
    // happens at least once
    do {
        // Set swapped to false and create new nodes
        swapped = false;
        shared_ptr<node> current = top_ptr_;
        shared_ptr<node> previous = nullptr;
        // Traverse through the list until we reach the end
        while (current->next != nullptr) {
            if (current->data > current->next->data) {
                // Swap the data member of the current node with that of the next node
                int temp = current->data;
                current->data = current->next->data;
                current->next->data = temp;
                // Set swapped to true
                swapped = true;
            }
            // Update the previous and current nodes
            previous = current;
            current = current->next;
        }
    } while (swapped);
}
```