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

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.

How Are Linked Lists Declared?

Linked lists, are classes. So in order to declare a linked list, we have to create an object that is allocated on the stack.

Linked List Declartion

Declaring a linked list in C++:

```
#include <iostream>

\\ Class definition goes here....

int main() {
    LinkedList testDeclaration; \\ Linked List Declaration
}
```

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.

Value Initialization

The function to initialize a node in the linked list with values can be seen below:

```
/*  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;
}
```