What is true about removing an element from a vector data structure?
(a) It is computationally cheaper to remove an element at a lower index than it is to remove an element at a higher index.
(b) Removing the element at highest index is always an O(1) operation.
(c) The vector data structure is designed to be a container of elements; it does not support element removal.
(d) Removing the highest index element involves shifting mySize-1 many elements one position to the right.

Q4: A vector data structure is implemented so that the two data members `int theSize` and `int myCapacity` will always remain properly synchronized. The implementations will make sure that ...
(a) The value of `theSize` will never be equal to the value of `myCapacity`.
(b) The value of `theSize` will always be less or equal `myCapacity`.
(c) The value of `theSize` will always be equal to `myCapacity`.
(d) The value of `myCapacity` will be increased when `theSize` has reached a value that is a constant amount larger than myCapacity.

Q6: If the list data structure did not maintain a data member `mySize` for the purpose of keeping track of the number of elements contained, which code fragment below could compute the size of a list?

(a) `iterator i = begin(); while (i < 0) i++; return i;`
(b) `return tail - 1;`
(c) `Node<T>* i = head->next; int k=0; while (i != 0) {k++; i = i->next;} return *i;`
(d) `Node<T>* i = head->next; int k=0; while (i != 0) {k++; i= i->next;} return k;`

Q18. What does the following code fragment compute for a given List mylst of integers? Assume that the list is not empty and elements are stored in ascending order of their values.

```
List<int>::iterator i1 = mylst.begin();
List<int>::iterator i2 = mylst.end();

while (true) {
   if (i1 == i2) break;
   if (i1->next == i2) break;
   ++i1; ++i2;
}
if (i1 == i2)
  return *i1;
else
  return (*i1+*i2)/2;
```

(a) The middle element of the vector.
(b) The median value of the vector.
(c) The average between the smallest and largest value in the vector.
(d) This won't work.

Study:

Vectors, Linked Lists, stack, Queue, Big O( ) } C++ Implements + Conceptual Understanding

Vectors:

- Consume more memory than arrays in exchange for the ability to manage storage & grow dynamically in an efficient way.
- Can allocate for extra storage than what is strictly needed.
- do not reallocate each time an element is added to the container.
- efficient at adding values to the begin & end, but inefficient at any other index

Linked List:

- A linear data structure in which elements are not stored at continguous memory locations. The elements in a linked list are linked using pointers.
- IOW, A linked list consists of nodes where each node contains a data field & a reference (link) to the next node in the list.
- Dynamic sizing
- Ease of insertion & deletion

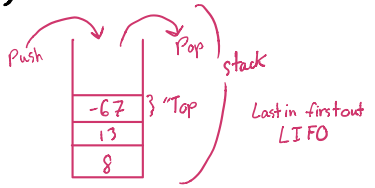
Source: Geeks for geeks.org/linked-lists.

Limits:
- Random access is not allowed. We have to access elements sequentially from the first node. So we cannot do binary search with linked lists efficiently with its default implementation
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are continguous locations, there is locality of reference which is not there in case of linked lists

Representation:
- a pointer to the first node is called the head.
- If the list is empty, the value of the head is NULL
- Each node consists of 2 parts:
  1) Data
  2) Pointer to the next node

Stack:

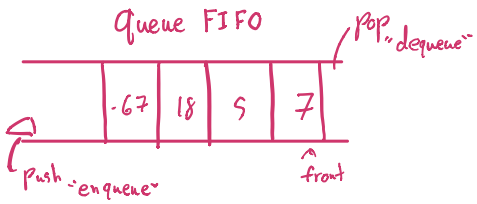
- A type of container adaptors with Last In First Out type of working where a new element is added at one end & an element is removed from that end only.



Queue:

- A type of container adaptors with First In First Out type of working where a new element is added at one end of the container & extracted from the other.

First in firstout



Big O( ) Notation:

expresses the runtime interms of how quickly it grows relative to the input, as the input grows grows arbitrarily large.

1) How quickly the runtime grows
2) the size relative to the input (n)
3) The input get Arbitrarily large

O(1) = constant time ——— 1 item.

With a loop, we get O(n), called linear time

——— 1000 items = 1000 time printed

With a nested loop, we get $O(n^2)$, called quadratic time

1000 items = 1000000 times printed

N could be the actual input, or the size of the input

3) As n gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.