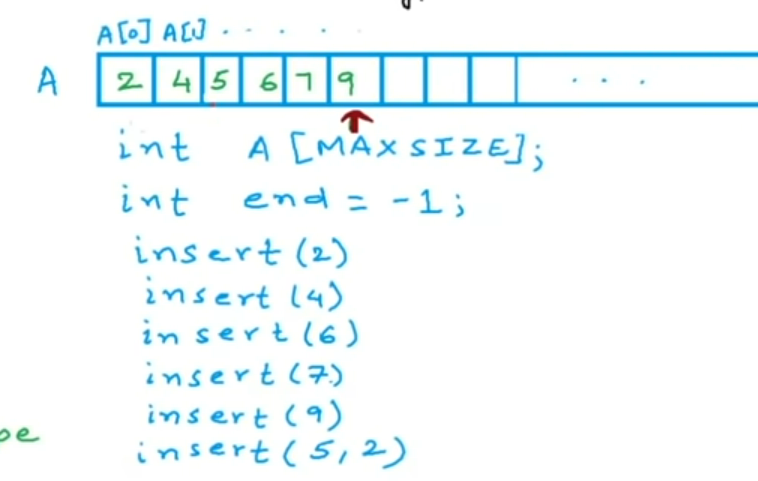
**Lists**

* Store a given number of elements of a given data type
* Write/modify element of a position
* Read element of a position
* empty list has size 0
* insert, remove, count, read/modify at a position
* specify data type



if we remove the number 2 at index 0, we need to shift all elements to the left

-1 = variable to determine end of the list, so if N is -1, the list is empty

When array is full, create a new larger array,copy previous array into the new array. Free the memory for the previous array.

Time Complexity

1. Access – Read/write element at an index, O(1)
2. Insert – Shift elements to the right, length of the list n, O(n)
3. Remove – O(n)
4. Add – O(n)

**Linked List**

struct node{

int data;

Node\* next;

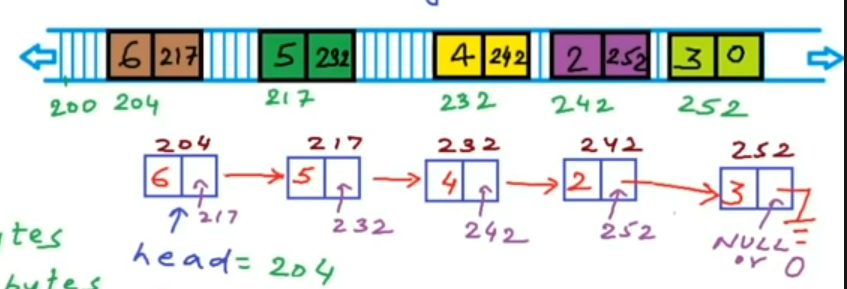
}

Node\* - pointer to node, stores address of next node in the list

Address of the head – address of the first node, gives us access to the complete list

Address of the last node = null or zero, which means that the last note does not point to any other node

To traverse through the linked list, we start at the head, then we go through the addresses



Time Complexity

1. Access – Read/write element starts at head, O(n)
2. Insert – No shifting, O(n)

**Linked List Implementation**

*struct Node{*

*int data;*

*Node\* next;*

*};*

*//Initialize linked list*

*Node\* A;*

*A = NULL; //empty list*

*Node\* temp = new Node(); //(Node\*)malloc(sizeof(node));*

*temp->data = 2; //(\*temp).data = 2;*

*(\*temp).next = NULL;*

*A = temp;*

*//Create a new node*

*temp = new Node();*

*temp->data = 4;*

*temp->next = NULL;*

*//Traversal to insert at end*

*Node\* temp1 = A;*

*while(temp1->next != NULL){*

*temp1 = temp1->next;*

*cout << temp->data;*

*}*

*temp1->next = temp;*