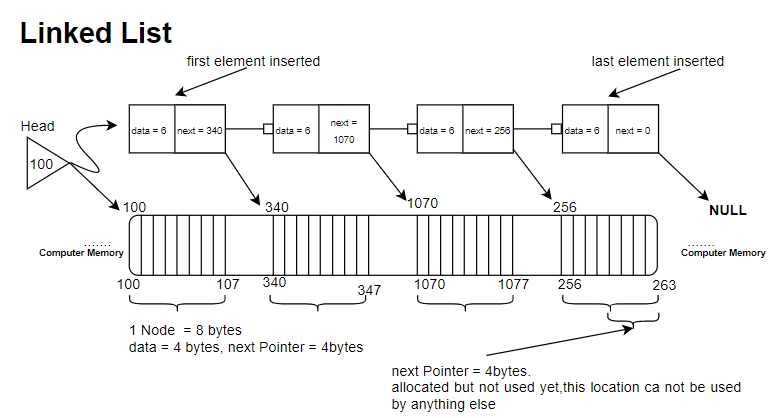
**LinkedList**



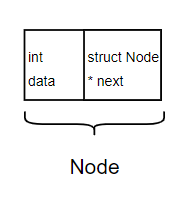
**Definition:**

It's a linear data structure (a linear collection of data elements), which are stored in known consecutive (non continuous) locations in memory, each element points to next. It is a data structure consisting of a collection of nodes which together represent a sequence.

LinkedList is non continuous data structure. Therefore, some other information need to be stored ( the address of the next Node). So the goal is to store the data in non continuous form, so we can add more values to the collection anytime at any free space in the memory. ( Arrays are continuous, Adding needs extend of the size.)

In the figure above you can see that LinkedList, each node has data, and next.

1. To initialize a LinkedList Node in C language we use:



#include <stdio.h>

#include <stdlib.h>

struct Node{

int data;

struct Node \*next;

};

Node contains two data types, data is an integer value, and next is a Pointer stores address of the next Node in the list.

1. To create the first Node in the linked list:

Code:

int main (){

struct Node \*head, \*newnode;

newnode = (struct Node \*) malloc(sizeof(struct Node));

printf ("Enter Data: ");

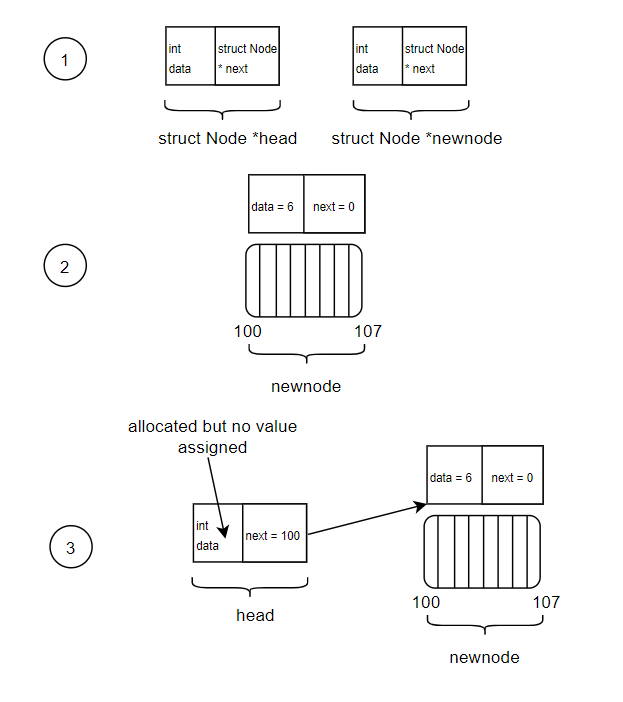
scanf ("%d", &newnode -> data); // Data entered is 6

newnode -> next = 0;

head = newnode;

}

Visulaization:



Explanation:

1. To create a linked list:

Code:

struct Node \* CreateLinkedList(struct Node \*\*Emptyhead){

struct Node \*head = \*Emptyhead, \*newnode, \*temp;

int choice = 1;

while (choice){

newnode = (struct Node \*) malloc(sizeof(struct Node));

printf ("Enter Data: ");

scanf ("%d", &newnode -> data);

newnode -> next = 0;

if (head == 0){

head = newnode;

temp = head;

}else{

temp -> next = newnode;

temp = newnode;

}

printf ("More Data ? Enter 0 for No and 1 for Yes: ");

scanf ("%d", &choice);

};

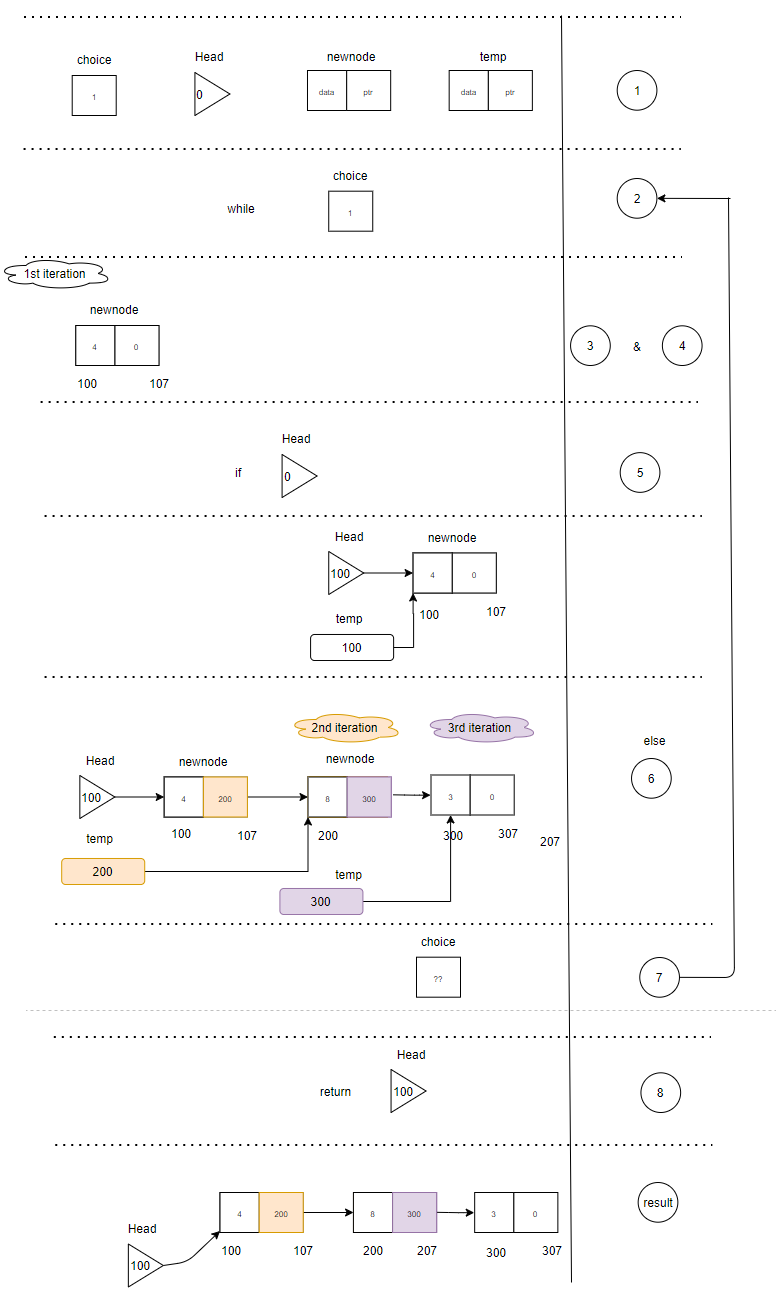
return head;

};

Explanation:

ss

Visulaization:



A drawback of linked list is, I can not access the “n”th element data directely as in arrays. Since linkedlist is stored in non-continuous locations in the memory, only sequential access is possible.

|  |  |
| --- | --- |
| **Accessing of any element in array** | **Accessing of any element in Linkedlist** |
| a[3] = {1,2,3};  printf (“%d”, a[1]); // 2 | While ( i <= pos){  temp = temp -> next;  }  Print (“%d”, temp -> data); |
| Time Complexity = > O(1) | Time Complexity = > O(n) |

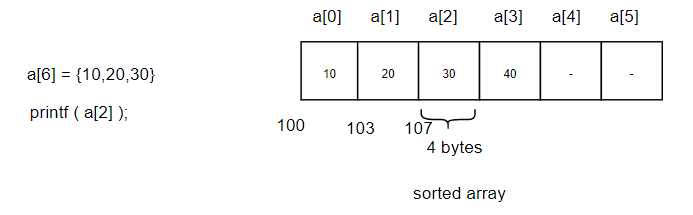
**Array vs LinkedList**

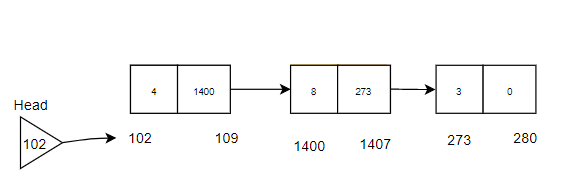
1. **Time Complexity of accessing an element**:

In arrays, if you know the base address ( fixed postion ), you can access any element in the array in constant time O(1).

In linkedlist, only sequential access is possible. If you know the base address (head), you should access (n) elements to reach to the target Node, with the target data in time O(n).

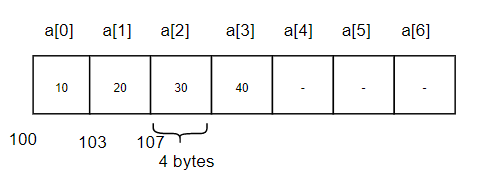
Array O (1) is better than a linkedlist O(n) in accessing elements, if you are accessing the elements frequently.



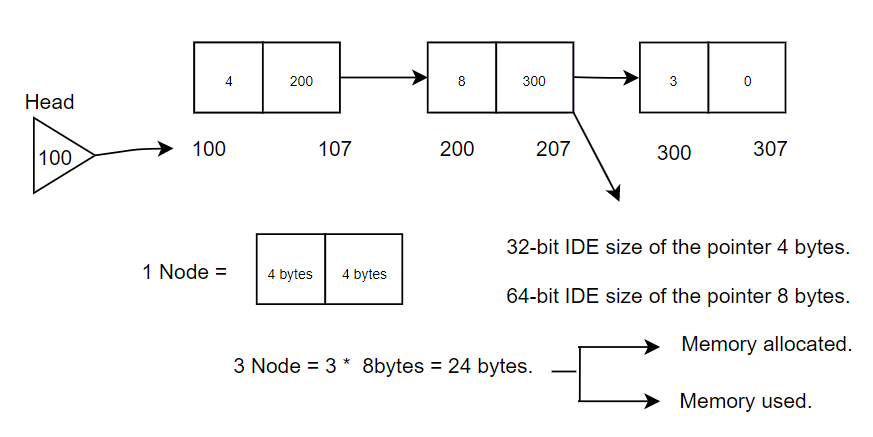


1. **Memory requirement + Memory Utilization:**
   1. Array:

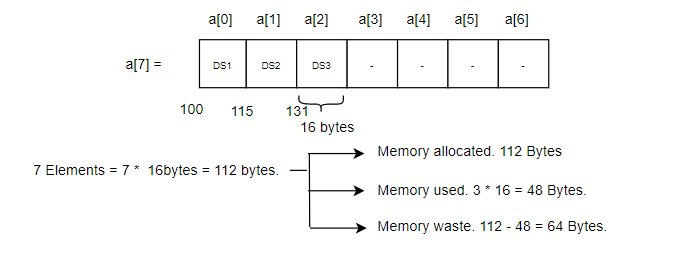
* Fixed size data structure. Generally, we have to give the size of the array at the decleration time only. We can’t change it at front ime. So we give maximum size to array in deleration e.g. a[100], maybe we are only using 10 elements only 🡪 **wastage of memory** 🡪 **ineffective memory utilization**.



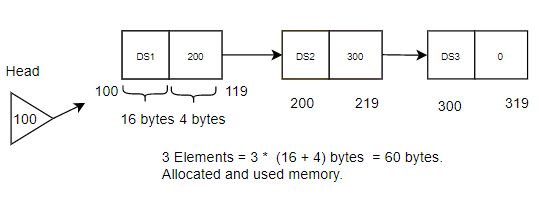
* Memory requirement:-
  + Memory allocated: 7 \* 4 = 28 bytes.
  + Used Memory: 3\* 4 = 12 bytes.
  + If you want to insert a new value in the array, you can add directly if you have space in the array, size of a[7] stay the same 28 bytes, used memory (3+1) \* 4 = 16 bytes.
  1. Linkedlist:
* Dynamic size, we don’t have to create a new Node during the runtime. **No wastage of space 🡪 Memory Utilization 🡪 Very efficient**.
* Memory requirement:
* If you want to insert a new value in the linkedlist old : 3 elements \* 8 bytes = 24 bytes.
* New: (3+1) elements \* 8 bytes = 32 bytes.
* 4 elements in array allocates 16 bytes of the memory, and in linkedlist allocates 32 bytes.



* 1. Now, if the case is to use complex data types as elements of array & linkedlist, it’s better to use linkedlist. Say we need to store a complex data types of 16 bytes each.
     + Array:



* + - Linkedlist:

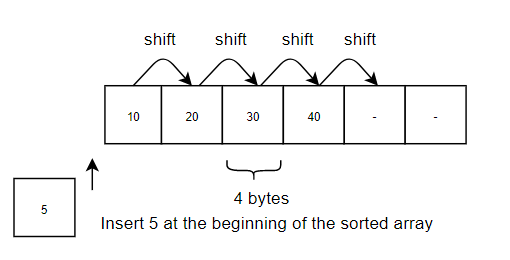


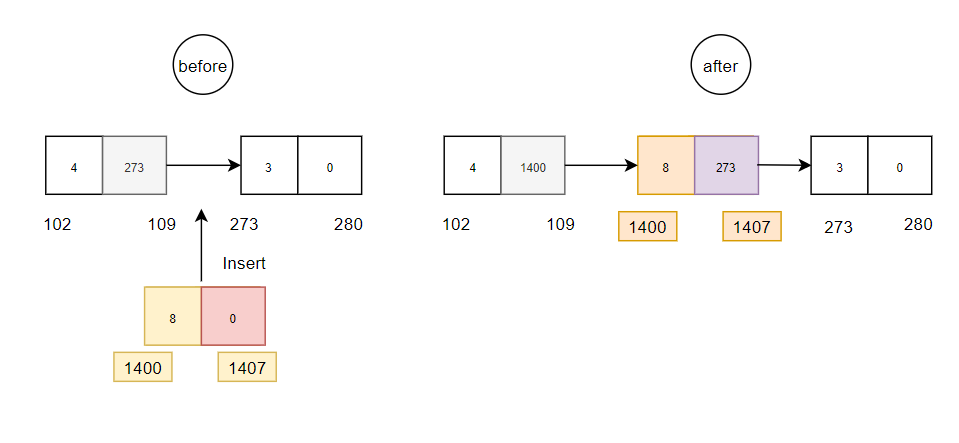
Therefore, it depends how many elements will be used: in array allocate say 4 bytes \* 7 elements = 28 bytes allocated memory and its all used no memory wastage, on the other hand linkedlist (4+4) bytes \* 7 elements = 56 bytes.

1. **Time complexity of insertion and deletion:**

Inseration, deletion, as well as other operations are easier to do in linked list than in an array. Linkedlist operations has a faster run time (**Time Complexity**).

* In arrays, we have to shift all elements O(n) first, then we can insert 5 in the beginning O(1).
* Deletion in arrays works same way, delete one element O(1), then shift n elements of the array to fill the void O(n).
* In linkedlist, only sequential access is possible.
* If you know the base address (head), you should access (n) elements to reach to the target Node, with the target data.
* Array O (1) is better than a linkedlist O(n) in accessing elements, if you are accessing the elements frequently.

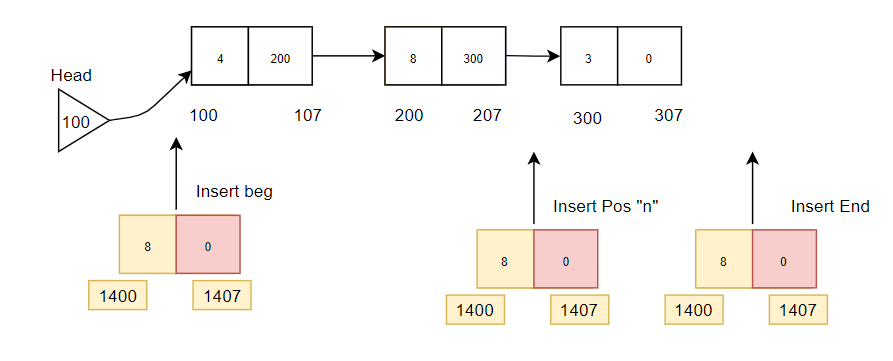




|  |  |  |
| --- | --- | --- |
|  | Array | Linkedlist |
| At the beginning | O(n)  You have to shift n elements & then insert. | O(1) 🡪 (Best case) |
| At “n”th position | O(n)  You have to shift n elements & then insert | O(n)  Only sequential access is possible, so you have to access n elements to reach the target location for insert. |
| At the End | O(1) 🡪 (Best case) | O(n)  Only sequential access is possible, so you have to access n elements to reach the target location for insert. |

* In array linear & binary search are possible, on the other hand in linkedlist only linear search is possible !, only sequential access is possible.

1. Insertion/ Push:



* 1. At the beginning

Code:

static void InsertFromBeg(struct Node \*\*head\_ref){

struct Node \*temp, \*newnode;

temp = \*head\_ref;

newnode = (struct Node \*) malloc(sizeof(struct Node));

printf("Enter the dataue: ");

scanf ("%d",&newnode -> data);

\*head\_ref = newnode;

newnode -> next = temp;

};

Visulaization:

Explanation:

* 1. At “n”th position

Code:

static void InsertFromPos(struct Node \*\*head\_ref){

struct Node \*prevnode ,\*posnode, \*nextnode, \*newnode;

newnode = (struct Node \*) malloc(sizeof(struct Node));

int i = 1;

int pos;

int count = getLength(\*head\_ref);

posnode = \*head\_ref;

printf("Enter the Postion: ");

scanf ("%d",&pos);

if (pos > count){

printf ("Indataid Position.");

}else{

while (i < pos && posnode != 0){

prevnode = posnode;

posnode = posnode -> next;

//nextnode = posnode ->next;

i++;

};

printf (" Current dataue : %d -> ", posnode -> data);

printf("Enter the dataue: ");

scanf ("%d",&newnode -> data);

newnode -> next = posnode;

prevnode -> next = newnode;

};

};

Visulaization:

Explanation:

* 1. At the End

Code:

static void InsertFromEnd(struct Node \*\*head\_ref){

struct Node \*temp, \*newnode;

temp = \*head\_ref;

newnode = (struct Node \*) malloc(sizeof(struct Node));

while (temp -> next != 0){

temp = temp -> next;

};

printf("Enter the value: ");

scanf ("%d",&newnode -> data);

temp -> next = newnode;

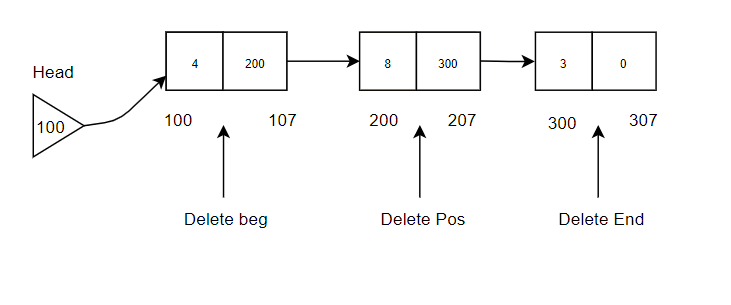
newnode -> next = 0;

};

Visulaization:

Explanation:

1. Deletion/ Pop:



* 1. At the beginning

Code:

static void DelFromBeg(struct Node \*\*head\_ref){

struct Node \*temp;

temp = \*head\_ref;

\*head\_ref = temp -> next;

free(temp);

};

Visulaization:

Explanation:

* 1. At “n”th position

Code:

static void DelFromPos(struct Node \*\*head\_ref ,int pos){

struct Node \*prevnode ,\*posnode, \*nextnode;

int i = 1;

posnode = \*head\_ref;

if (\*head\_ref == 0){

printf ("List is empty");

}else{

while (i < pos && posnode != 0){

prevnode = posnode;

posnode = posnode -> next;

nextnode = posnode ->next;

i++;

};

prevnode -> next = nextnode;

\*head\_ref = prevnode;

free(posnode);

};

};

Visulaization:

Explanation:

* 1. At the End

Code:

static void DelFromEnd(struct Node \*\*head\_ref){

struct Node \*prevnode, \*temp;

temp = \*head\_ref;

while (temp -> next!= 0){

prevnode = temp;

temp = temp -> next;

};

prevnode -> next = 0;

free(temp);

};

Visulaization:

Explanation:

1. Linkedlist getLength:

Code:

int getLength(struct Node \*head){

struct Node \*temp;

int count = 0;

temp = head;

while(temp != 0){

count++;

temp = temp -> next;

};

printf ( "\nLength is: %d \n" , count );

return count;

};

Visulaization:

Explanation:

1. Print linkedlist:

Code:

static void printList(struct Node \*head){

struct Node \*temp;

temp = head;

while (temp != 0){

printf ("%d -> ", temp -> data);

temp = temp -> next;

};

printf ("\n");

};

Visulaization:

Explanation:

1. Reverse a linkedlist

Code:

// Function recevices the head of a linkedlist as an input

static void Reverse(struct Node \*\*head\_ref){

struct Node\* prev = NULL;

struct Node\* current = \*head\_ref;

struct Node\* next = NULL;

while (current != NULL) {

// Store next

next = current->next;

// Reverse current node's pointer

current->next = prev;

// Move pointers one position ahead.

prev = current;

current = next;

}

\*head\_ref = prev;

};

Visulaization:

Explanation:

Types of Linkedlist:

|  |  |
| --- | --- |
| Singly | Double |
| Singly Circular | Double Circular |

LeetCode LinkedList questions:

1. Add Two Numbers ( Linkedlists )

Code:

Visulaization:

Explanation: