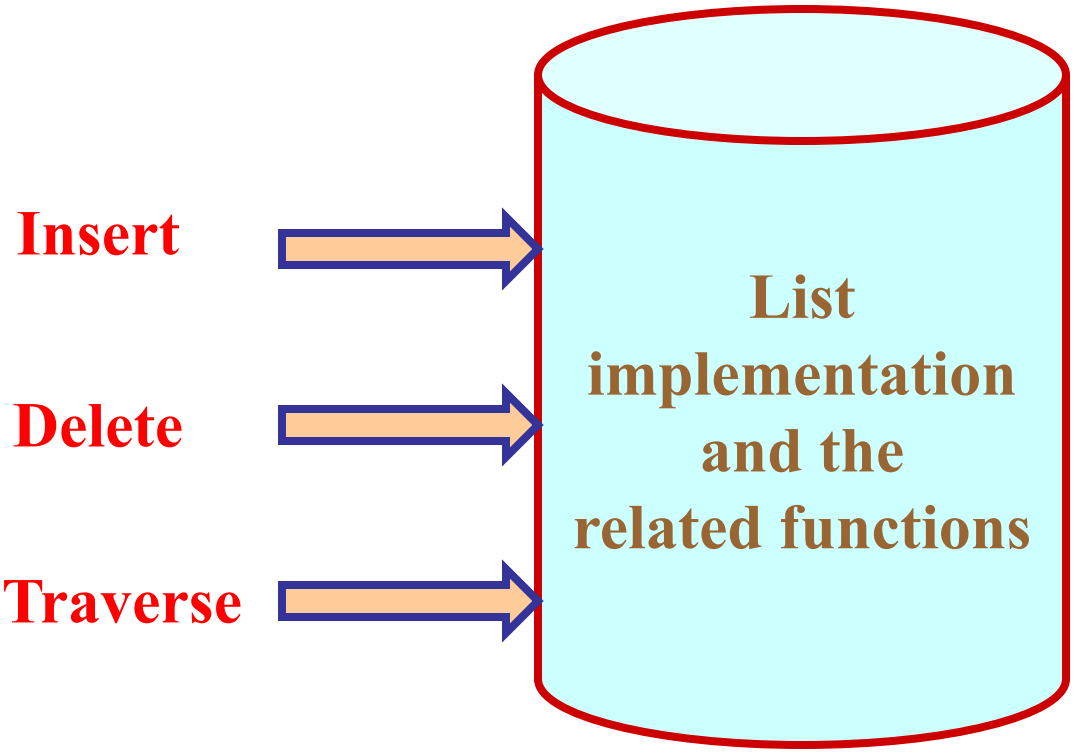
Main operations

- Create list
- Add node
  - beginning,middle or end
- Delete node
  - beginning,middle or end
- Find node
- Traverse list

Conceptual Idea



https://www.programiz.com/dsa/linked-list-operations

Example: Working with linked list

- Consider the structure of a node as follows:

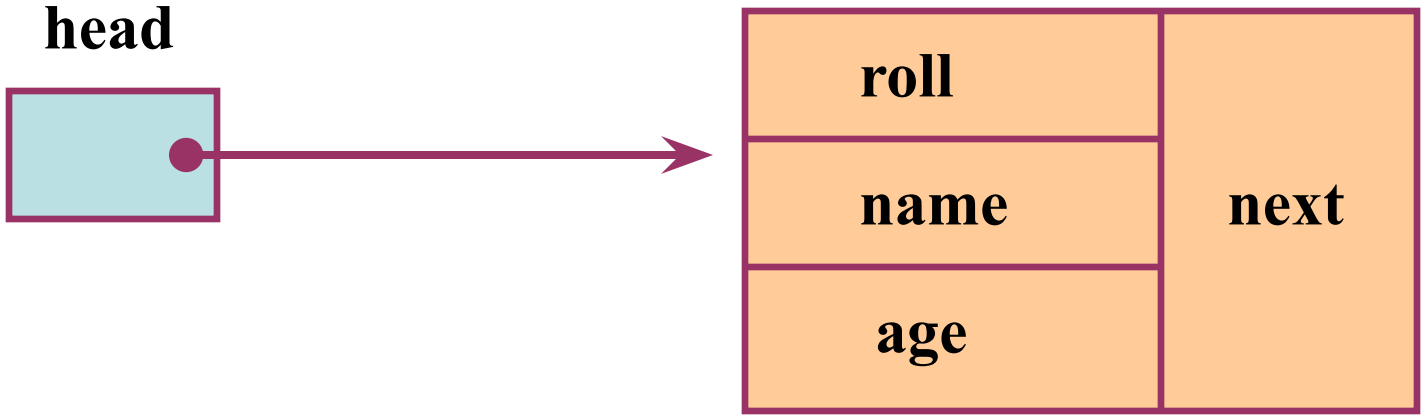
```
struct  stud  {
                int     roll;
                char  name[25];
                int     age;
                struct  stud  *next;
              };

        /* A user-defined data type called “node” */
typedef  struct  stud  node;
node  *head;
```

##typedef is used to define a data type in C.

Creating a List

- To start with, we have to create a node (the first node), and make head point to it.

```
head = (node *) malloc (sizeof (node));
```



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.

```
void  create_list  (node  *list)
{
   int  k, n;
   node  *p;
   printf  ("\n How many elements?");
   scanf ("%d", &n);

   list  =  (node *)  malloc (sizeof (node));
   p = list;
   for  (k=0; k<n; k++)
   {
      scanf ("%d %s %d", &p->roll,
                          p->name, &p->age);
      p->next  = (node *) malloc
                              (sizeof (node));
      p  =  p->next;
   }
   free (p->next);
   p->next  =  NULL;
}
```

To be called from the main() function as:

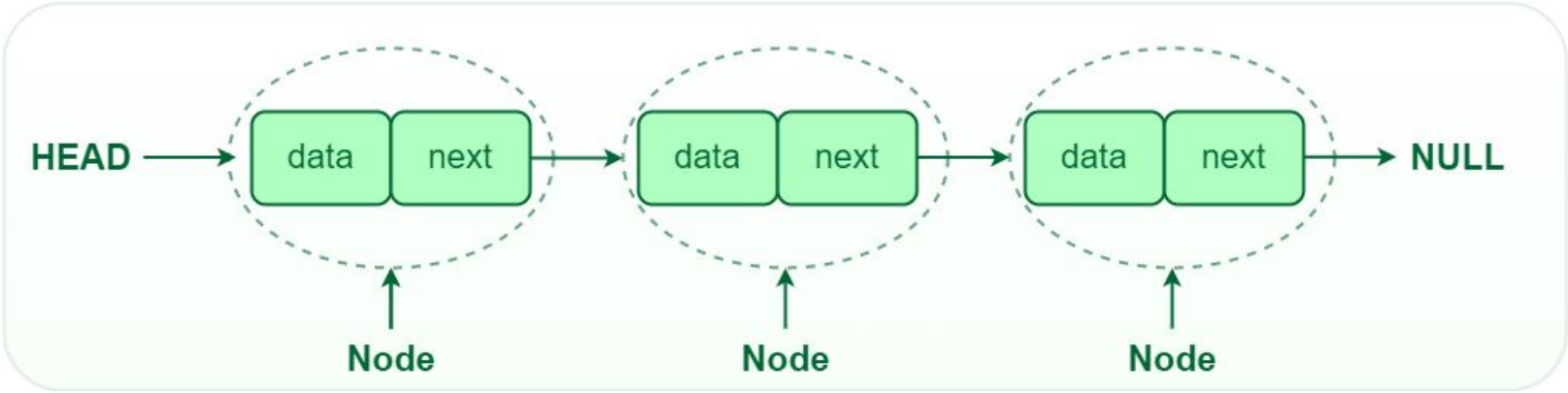
```
node  *head;
.......
create_list (head);
```

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to `malloc`.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file `stdlib.h`.

Traversing the List

- Once the linked list has been constructed and head points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the next pointer points to NULL.



Single-linked list

```
void  display_list  (node  *list)
{
   int  k = 0;
   node  *p;

   p = list;
   while  (p != NULL)
   {
     printf (“Node %d:  %d %s %d”, k, p->roll,
                                        p->name, p->age);
    k++;
     p  =  p->next;
   }
}
```

Inserting a Node in the List

- The problem is to insert a node before a specified node.
  - Specified means some value is given for the node (called key).
  - Here it may be roll.
- Convention followed:
  - If the value of roll is given as negative, the node will be inserted at the end of the list.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - head is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void  insert_node  (node  *list)
{
   int  k = 0, rno;
   node  *p, *q, *new;

   new  =  (node *)  malloc (sizeof (node));
   scanf ("%d %s %d", &new->roll, new->name,
                                        &new->age);

   printf ("\nInsert before roll (-ve for end):");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll  ==  rno)      /* At the beginning */
   {
      new->next  =  p;
      list  =  new;
   }
```

```
   while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }

   if  (p == NULL)          /* At the end */
   {
      q->next = new;
      new->next = NULL;
    }

    if  (p->roll  == rno)     /* In the middle */
   {
      q->next = new;
      new->next = p;
   }
}
```

The pointers q and p always point to consecutive nodes.

Deleting an Item

- Here also we are required to delete a specified node.
  - Say, the node whose roll field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```
void  delete_node  (node  *list)
{
   int  rno;
   node  *p, *q;

   printf ("\nDelete for roll :");
   scanf ("%d", &rno);

   p = list;
   if  (p->roll == rno)          /* Delete the first element */
   {
      list = p->next;
      free (p);
   }
```

```
  while  ((p != NULL) && (p->roll != rno))
   {
      q = p;
      p  =  p->next;
   }


   if  (p == NULL)              /* Element not found */
       printf ("\nNo match :: deletion failed");

   if  (p->roll == rno)         /* Delete any other element */
   {
      q->next  =  p->next;
      free (p);
   }
}
```

https://www.log2base2.com/data-structures/linked-list/linked-list-in-c.html

Types of Linked List

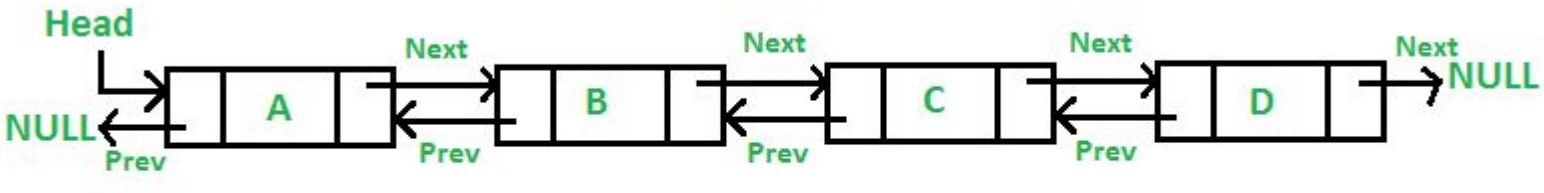
Singly Linked List: A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

To create and display Singly Linked List:

https://www.w3resource.com/c-programming-exercises/linked_list/c-linked_list-exercise-1.php (code is well commented)**

https://www.javatpoint.com/program-to-create-and-display-a-singly-linked-list

Doubly linked list



A **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

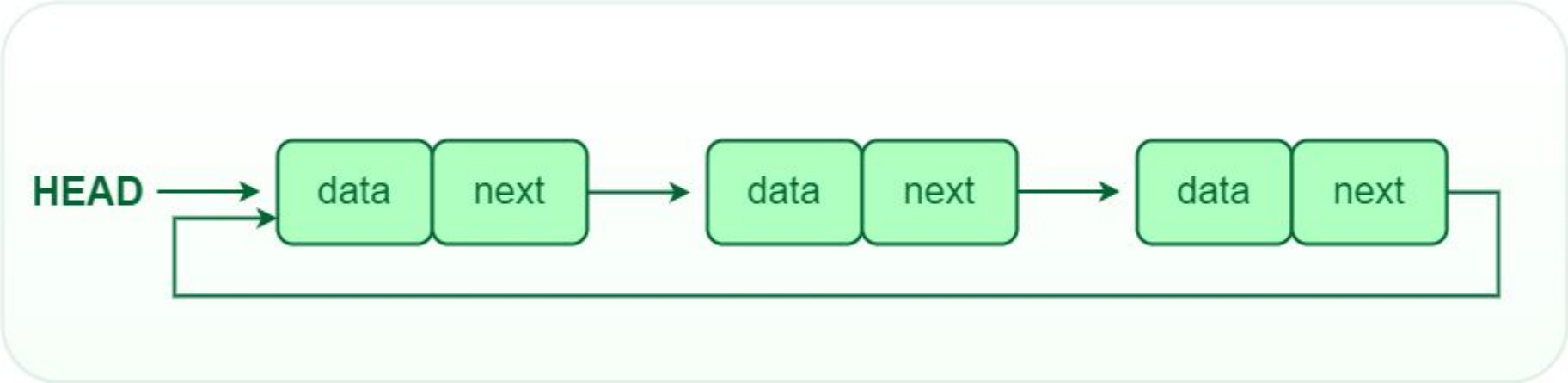
In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it **requires additional memory for the backward reference.**

Code: https://www.programiz.com/dsa/doubly-linked-list

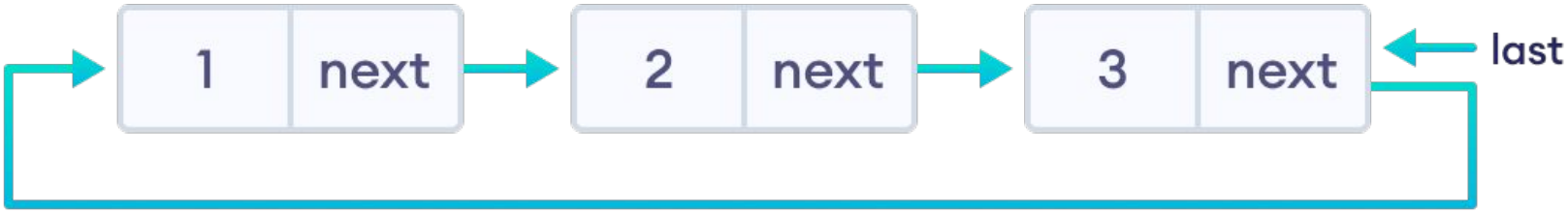
A doubly linked list is a type of linked list in which each node consists of 3 components:

- *prev - address of the previous node
- data - data item
- *next - address of next node

- **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the **last node contains the address of the first node,** forming a **circular loop** in the Circular Linked List. It can be either singly or doubly linked.



Circular Linked List



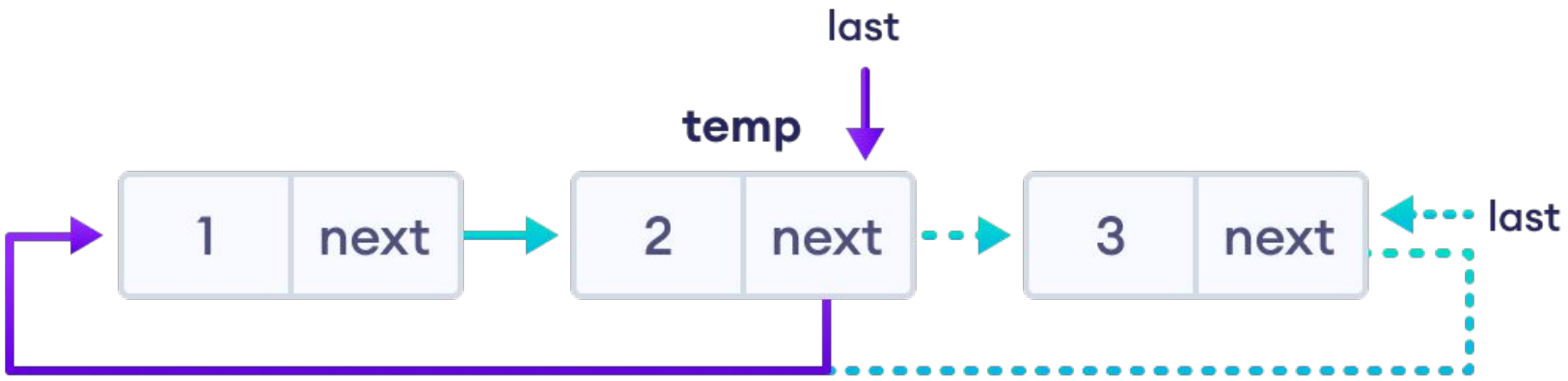
1. If the node to be deleted is the only node

- free the memory occupied by the node
- store NULL in `last`

2. If last node is to be deleted

- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node

Deletion of Last Node



3. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be `temp`
- store the address of the node next to 2 in `temp`
- free the memory of 2

Code: https://www.programiz.com/dsa/circular-linked-list

Counting Nodes in Circular LL

Code:
https://www.educative.io/answers/how-to-count-nodes-in-a-circular-linked-list

What is a Polynomial?

What is Polynomial?

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

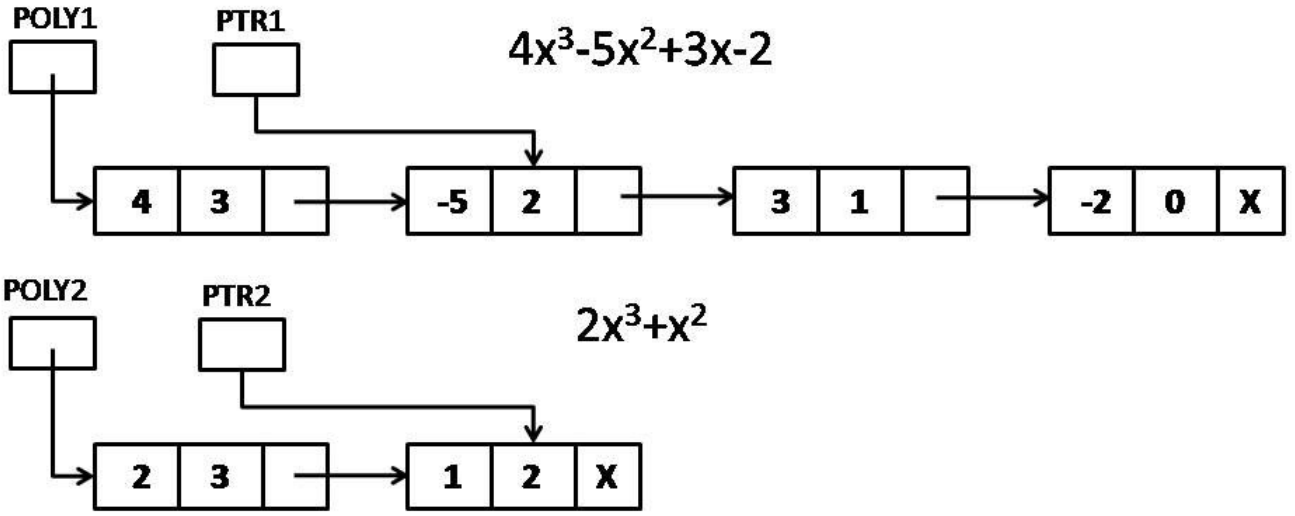
- one is the coefficient
- other is the exponent

Example:

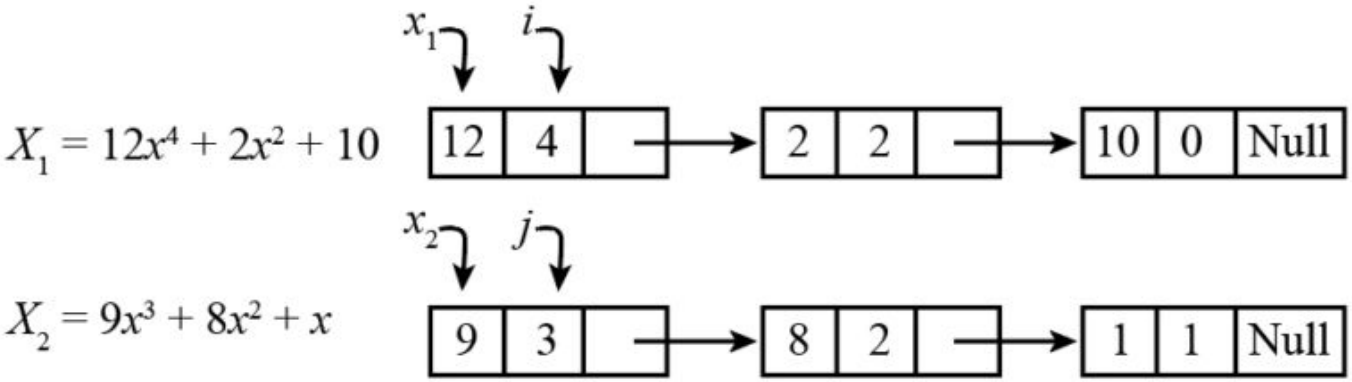
$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

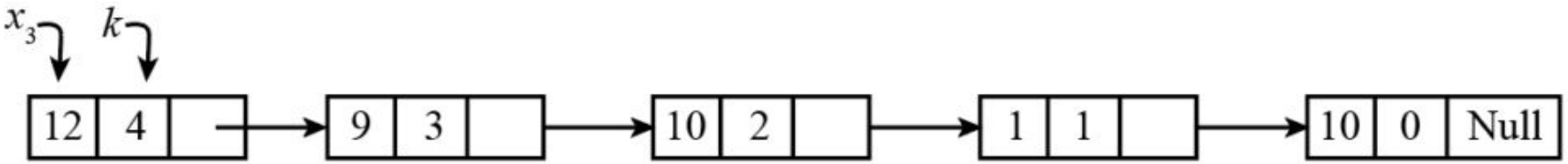
- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent
- For adding two polynomials that are stored as a linked list, we need to add the coefficients of variables with the same power.



Adding two polynomials using Linked List



The resultant liked list :-



Code & Steps: https://www.javatpoint.com/application-of-linked-list

https://mycareerwise.com/programming/category/linked-list/polynomial-addition-using-linked-list-313

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In undo functionality of softwares
- Hash tables, Graphs

Advantages of Linked Lists

. **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
. **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
. **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists

. **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
. **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Applications of Singly Linked List are as following:

1. It is used to implement **stacks** and **queues** which are like fundamental needs throughout computer science.
2. To prevent the collision between the data in the **hash map**, we use a singly linked list.
3. If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.
4. We can think of its use in a photo viewer for having look at photos continuously in a slide show.
5. In the system of train, the idea is like a singly linked list, as if you want to add a Boggie, either you have to take a new boggie to add at last or you must spot a place in between boggies and add it.

Applications of Circular Linked List are as following:

1. It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.
2. Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.
3. It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism (this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking).
4. Multiplayer games use a circular list to swap between players in a loop.
5. In photoshop, word, or any paint we use this concept in undo function.

Applications of Doubly Linked List are as following:

1. Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
2. In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
3. Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.
4. It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to. Basically it provides full flexbility to perform functions and make the system user-friendly.
5. In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).
6. It is used in a famous game concept which is a deck of cards.

Stack

This sample program presents the user with four options:

1. Push the element
2. Pop the element
3. Show
4. End

It waits for the user to input a number.

- If the user selects 1, the program handles a push(). First, it checks to see if top is equivalent to SIZE - 1. If true, "Overflow!!" is displayed. Otherwise, the user is asked to provide the new element to add to the stack.
- If the user selects 2, the program handles a pop(). First, it checks to see if top is equivalent to -1. If true, "Underflow!!" is displayed. Otherwise, the topmost element is removed and the program outputs the resulting stack.
- If the user selects 3, the program handles a show(). First, it checks to see if top is equivalent to -1. If true, "Underflow!!" is displayed. Otherwise, the program outputs the resulting stack.
- If the user selects 4, the program exits.

Code: https://www.digitalocean.com/community/tutorials/stack-in-c

Declaration

```
#define  MAXSIZE  100

struct  lifo  {
                int  st[MAXSIZE];
                int  top;
              };
typedef  struct  lifo  stack;
```

Stack Creation

```
void  create  (stack  s)
{
    s.top = 0;        /* Points to last element
  pushed in */
}
```

Pushing an element into the stack

```
void  push  (stack s,  int  element)
{
    if  (s.top == (MAXSIZE-1))
        {
            printf (“\n Stack overflow”);
            break;
         }
    else
         {
              s.top ++;
              s.st [s.top]  =  element;
          }
```

Removing/Popping an element from the stack

```
int   pop  (stack s)
{
    if  (s.top == 0)
        {
            printf ("\n Stack underflow");
            break;
        }
    else
        {
            return  (s.st [s.top --]);
        }
}
```

Checking for stack full / empty

```
int   isempty  (stack s)
{
    if  (s.top == 0)    return 1;
    else   return (0);
}
```

```
int   isfull  (stack s)
{
    if  (s.top == (MAXSIZE – 1))    return 1;
    else   return (0);
}
```

Structure and Memory Allocation in C

Concept of Structure

Structure in C is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. The **struct** keyword is used to define the structure.

Syntax of Structure:

```
struct structure_name
{
    data_type
    member1;
    data_type
    member2;
    .
    .
    data_type memberN;
};
struct employee
{       int id;
    char name[20];
    float salary;
};
```

Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

To declare the structure variable by struct keyword, it should be declared within the main function.

```
struct
employee
{    int id;
     char name[50];
     float salary;
};

void main()
{
   struct employee
   e1,e2;
}
```

To declare variable at the time of defining the structure.

```
struct employee
{  int id;
   char name[50];
   float salary;
}e1,e2;
```

Accessing members of the structure

By . (member or dot operator)

Eg. e1.id=101

C program to print Student information using structure

```
#include<stdio.h>
#include <string.h>  struct
student
{    int id;
    char name[50];  char
    branch[50];
};
void main()
{    struct student    st1;
    scanf("%d %s %s",&st1.id ,st1.name,st1.branch);
    printf("%d %s %s",st1.id ,st1.name,st1.branch);

}
```

Array of Structures in C

An array of structures in C can be defined as the collection of multiple structure variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

C program to print Student information using array of structure

```
#include<stdio.h>
#include <string.h>
struct student
{
int rollno;
char name[10];
};
int main()
{
int i;
struct student st[4];
printf("Enter Records of 4
students");  for(i=0;i<4;i++)
{
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter  Name:");
scanf("%s",st[i].name);
}
```

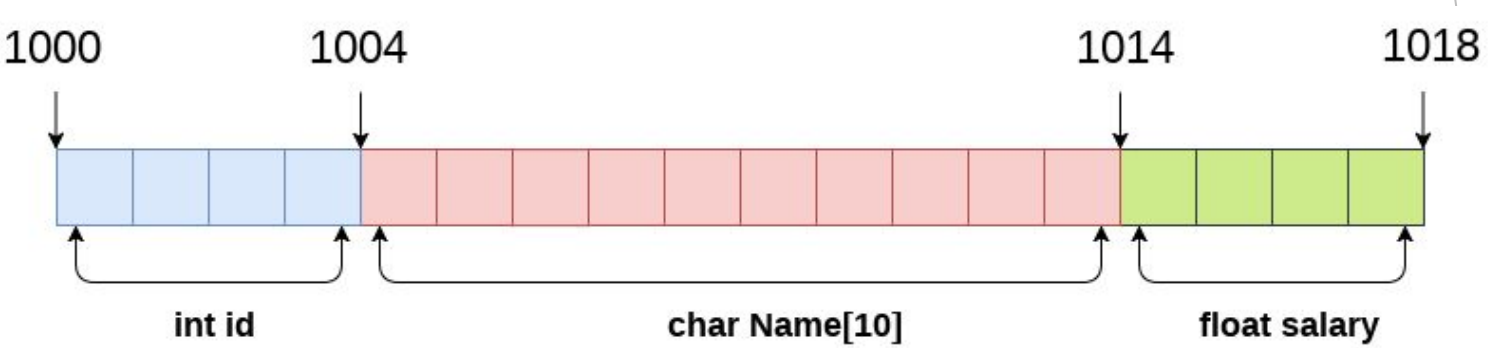
```
printf("\nStudent Information List:");
for(i=0;i<4;i++)
{
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

```
Id    Name     Branch
-------------------------
1     shailesh       EX
2      Amar          ECS
3      Pratik        Comp
4      Mugda         ECS
```

Examples for Practice

- A Hospital needs to maintain details of patients. Details to be maintained are First name, Middle name, Surname, Date of Birth, Disease. Write a program which will print the list of all patients with given disease.
- Define a structure called Player with data members Player name, team name, batting average Create array of objects, store information about players, Sort and display information of players in descending order of batting average.

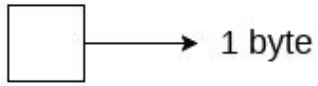
Memory Allocation in Structure




```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



Dynamic Memory Allocation in C.

C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

- malloc()
- calloc()
- free()
- realloc()

C malloc() method

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

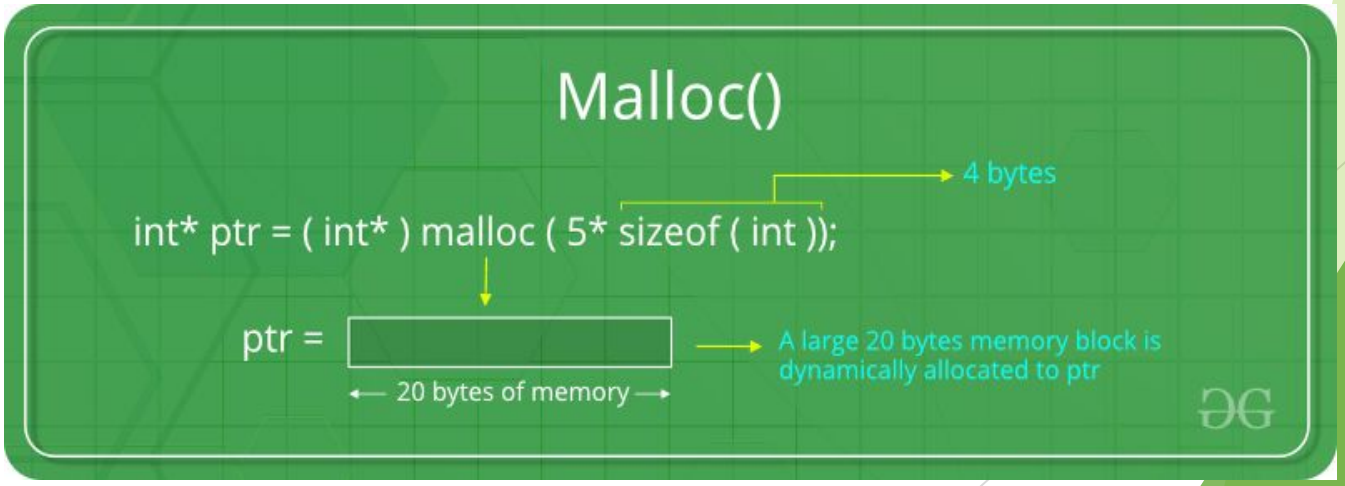
Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



calloc() method

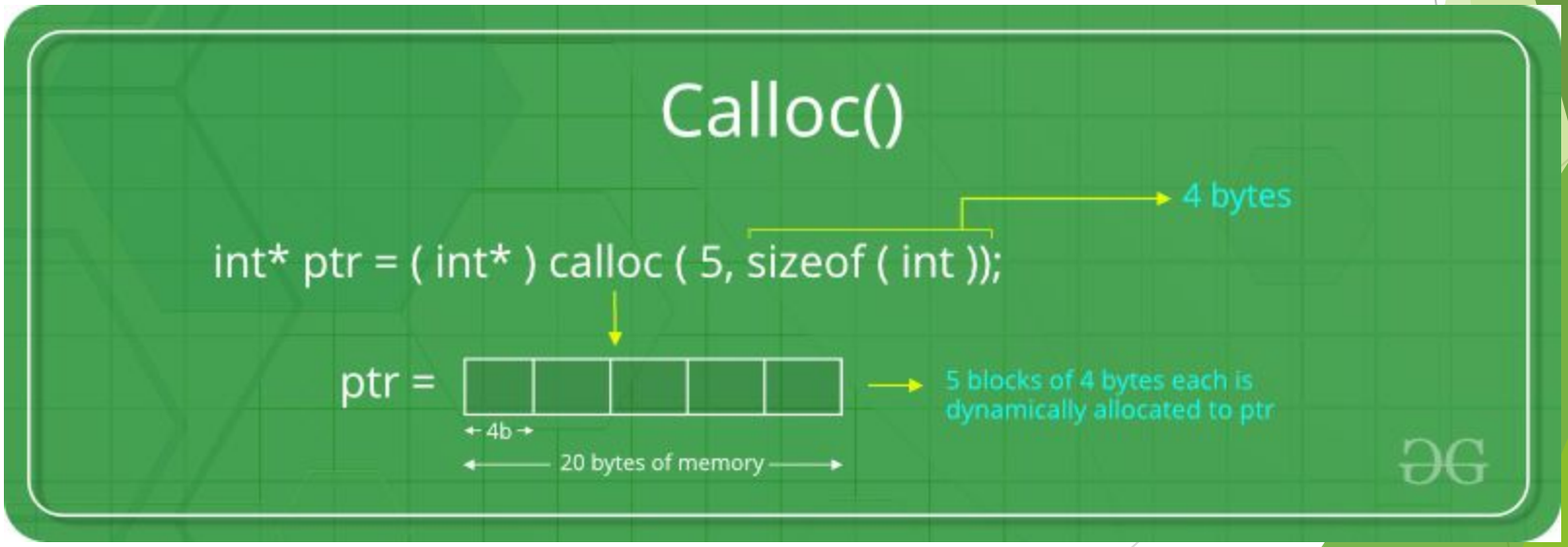
“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().

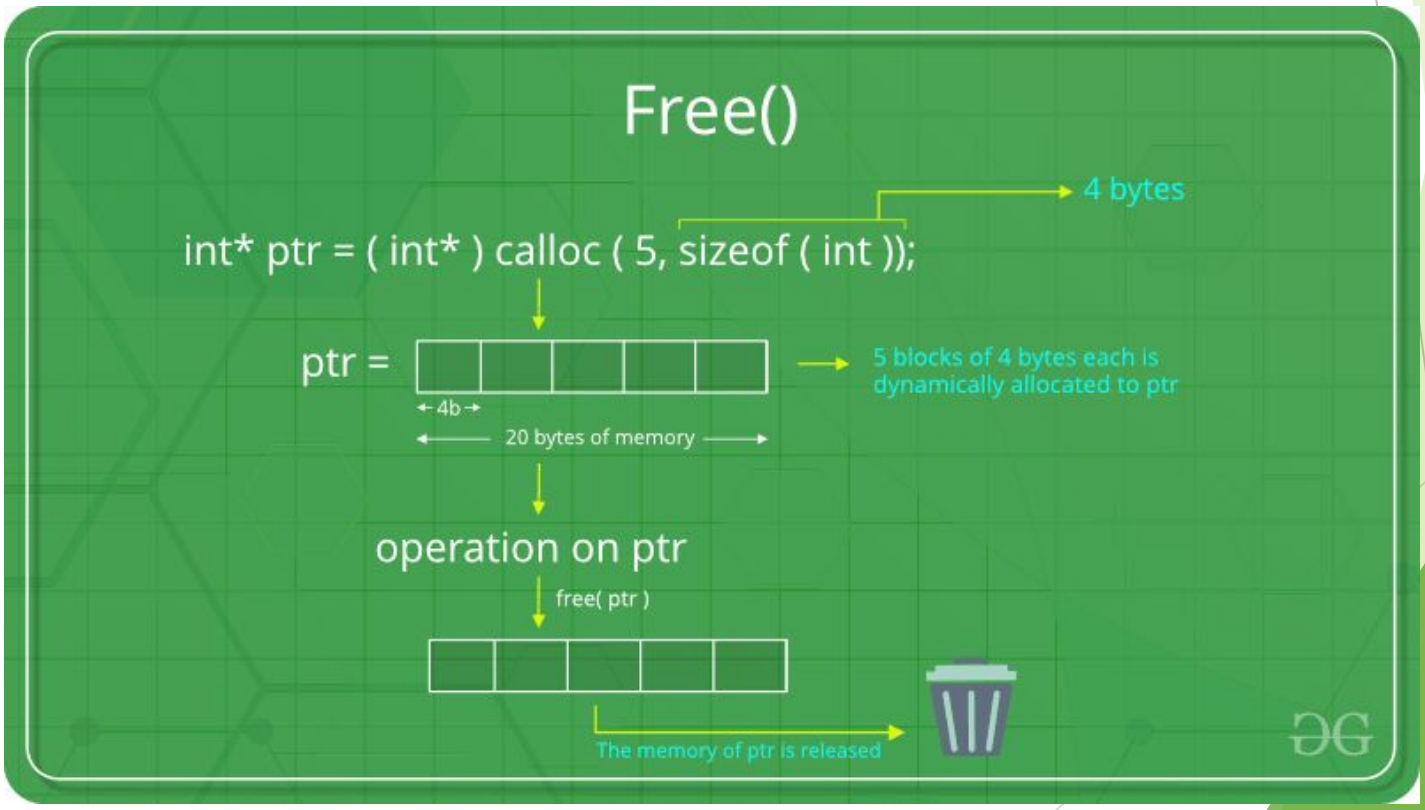
ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.



C free() method

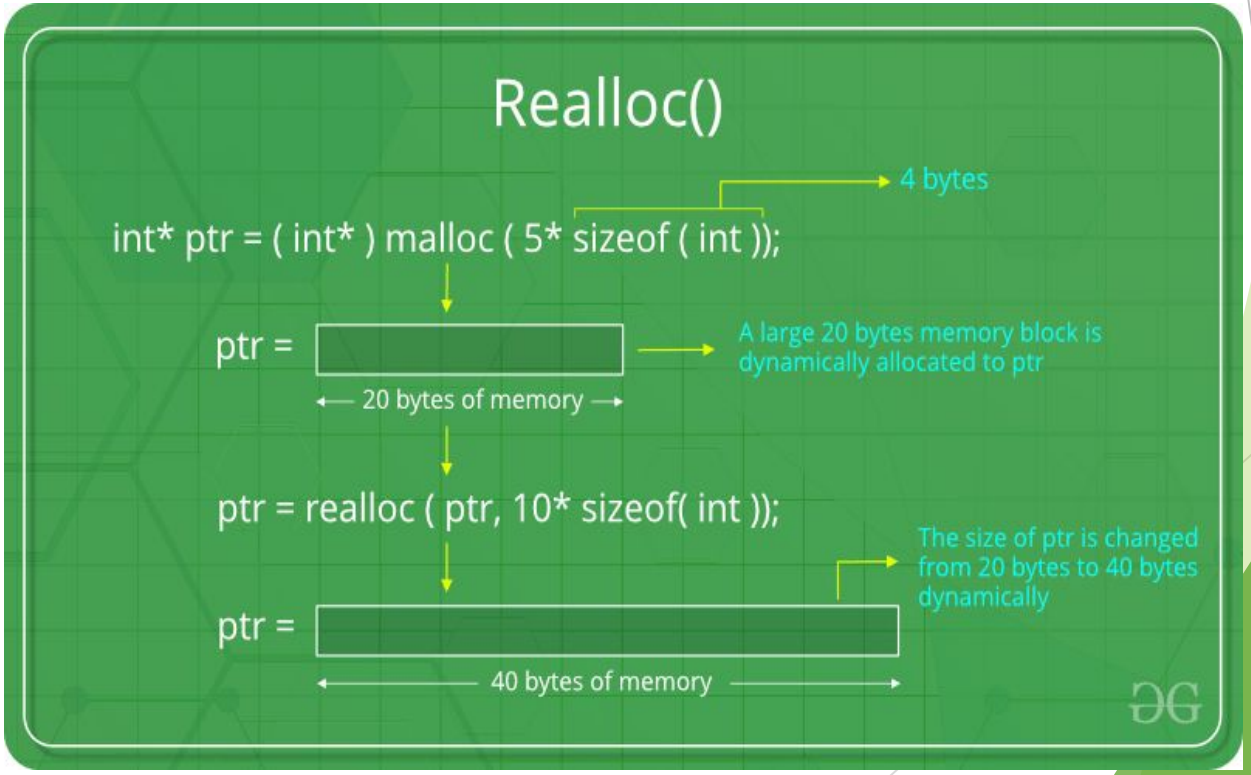
“free” method in C is used to dynamically **deallocate** the memory. The memory allocated using functions malloc() and calloc() is not deallocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.



C realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.



Stack Implementation Using Linked List

- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Code: https://www.scaler.com/topics/c/stack-using-linked-list-in-c/

https://www.codesdope.com/blog/article/making-a-stack-using-linked-list-in-c/ **(commented)**

Contd.

- Basic concept:
  - Insertion (push) and deletion (pop) operations take place at one end of the list only.
  - For stack creation / push operation
    - Required to call malloc function
  - How to check stack underflow?
    - Simply check if top points to NULL.
  - How to check overflow?
    - Check if **malloc*** returns –1.

The malloc() function stands for memory allocation, that allocate a block of memory dynamically. It reserves the memory space for a specified size and returns the null pointer, which points to the memory location.

Sample Usage

```
stack  A, B;

create (A);    create (B);
push (A, 10);  push (A, 20);  push (A, 30);
push (B, 5);   push (B, 25);  push (B, 10);

printf ("\n%d %d %d", pop(A), pop(A), pop(A));
printf ("\n%d %d %d", pop(B), pop(B), pop(B));

if  (not  isfull (A))
   push (A, 50);
if  (not  isempty (A))
   k  =  pop (A);
```

30 20 10

10 25 5

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo
                  stack;
stack s;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo
                   stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
   s->top = -1;

   /* s->top points to
      last element
      pushed in;
      initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
   *top = NULL;

   /* top points to NULL,
      indicating empty
      stack              */
}
```

LINKED LIST

Pushing an element into stack

```
void push (stack *s, int element)
  {
   if (s->top == (MAXSIZE-1))
   {
     printf ("\n Stack
overflow");
         exit(-1);
     }
     else
     {
         s->top++;
         s->st[s->top] = element;
     }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *)malloc
(sizeof(stack));
    if (new == NULL)
    {
       printf ("\n Stack is full");
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf ("\n Stack
underflow");
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
   int t;
   stack *p;
   if (*top == NULL)
   {
      printf ("\n Stack is empty");
      exit(-1);
   }
   else
   {
      t = (*top)->value;
      p = *top;
      *top = (*top)->next;
      free (p);
      return t;
   }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for Stack Full

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Example: A Stack using an Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
  return;
}
```

Example: A Stack using Linked List

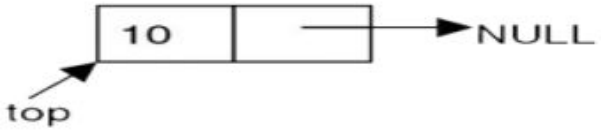
```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;

main() {
  stack *A, *B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf ("\n B is empty");
  return;
}
```

Stack using SLL

Understanding PUSH

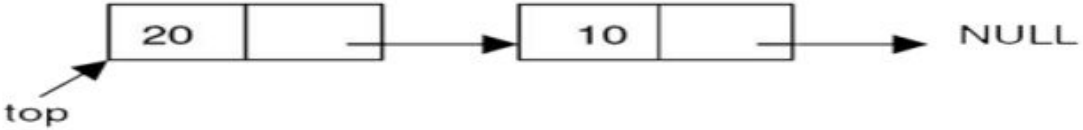
head to SLL is top to STACK

Initially



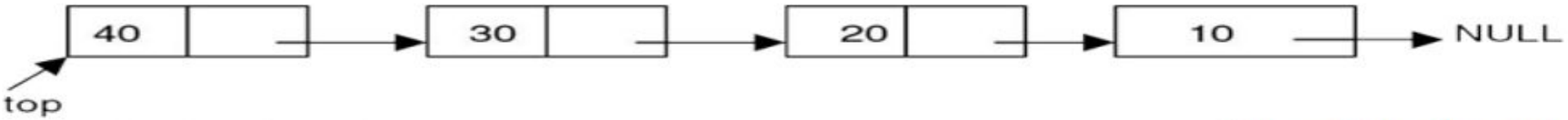
insertBegin to SLL is push to STACK

After first iteration



After second iteration

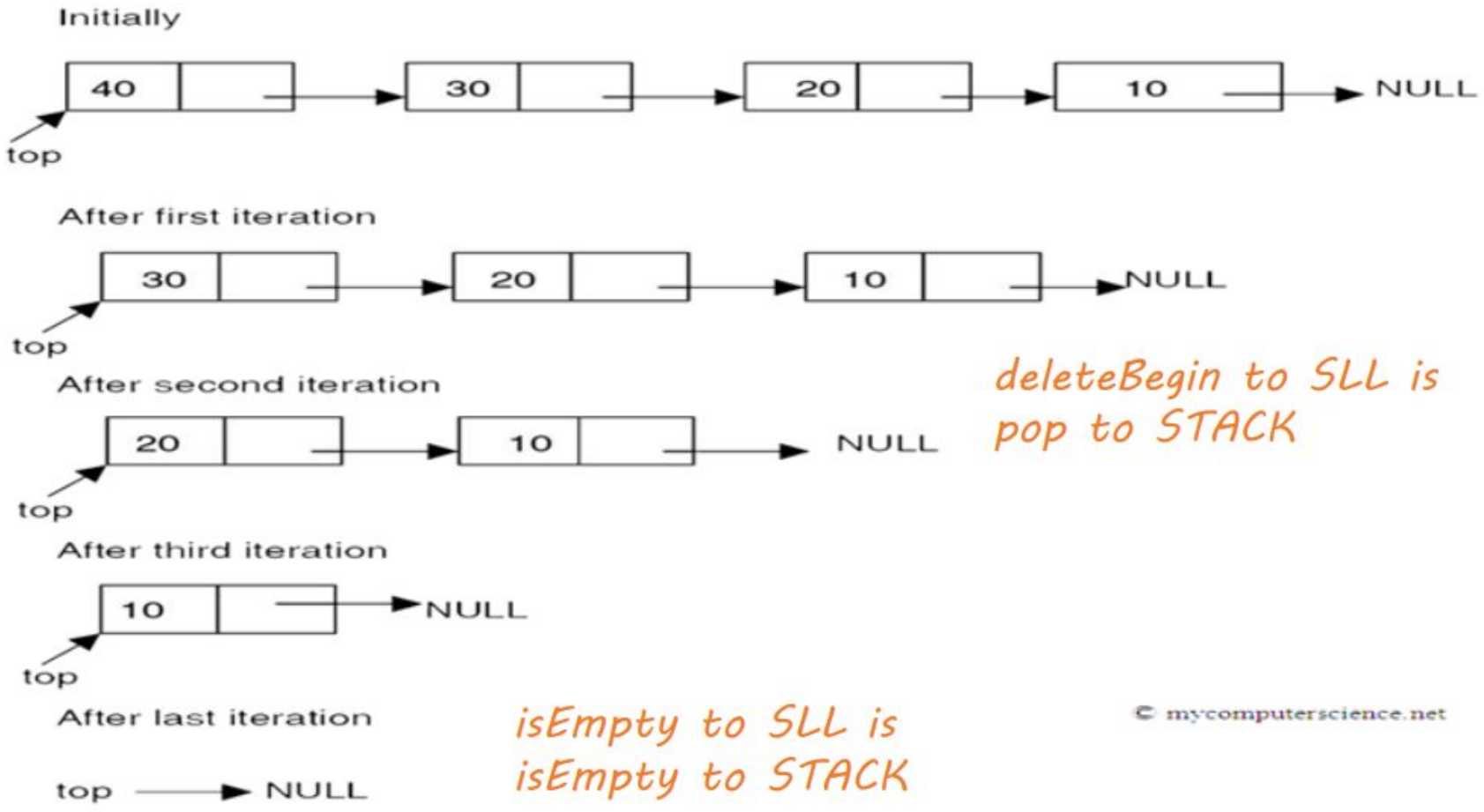
After third iteration



After last iteration

Stack using SLL

Understanding POP



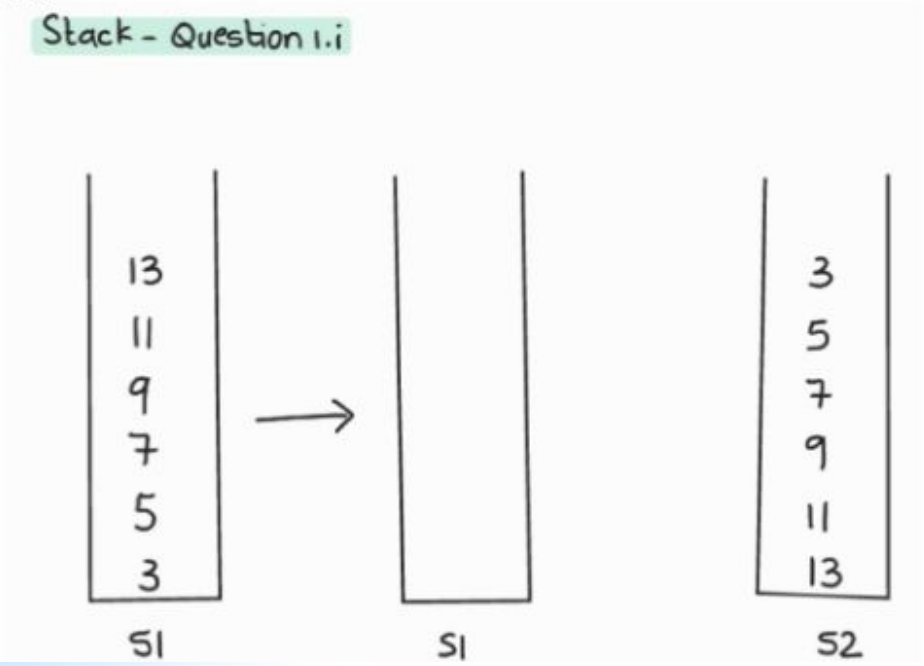
deleteBegin to SLL is pop to STACK

isEmpty to SLL is isEmpty to STACK

1. Imagine we have two empty stacks of integers, s1 and s2. Draw a picture of each stack after the following operations:

i.

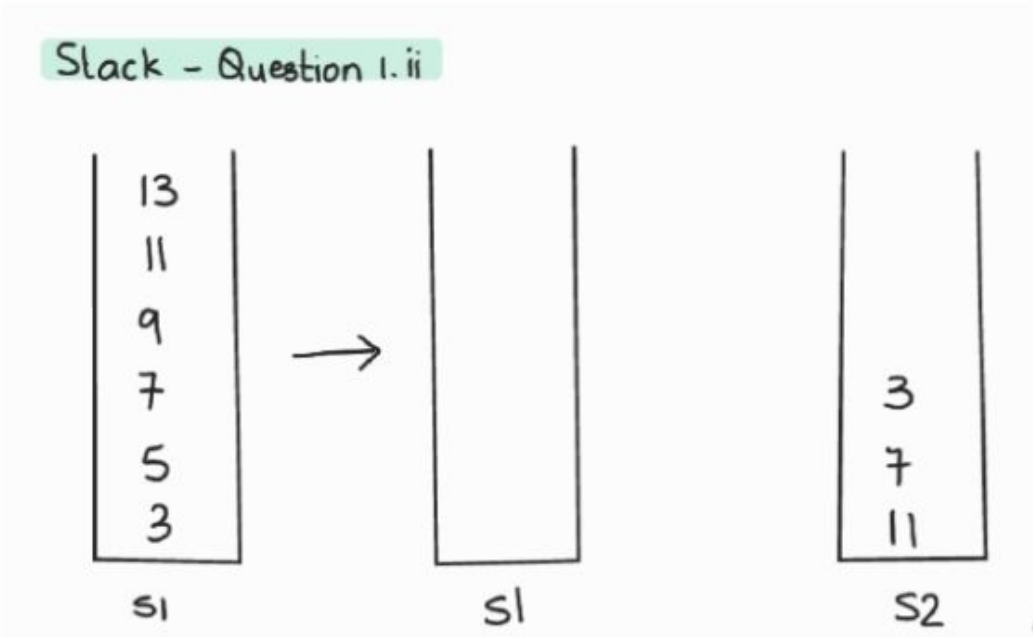
```
pushStack(s1, 3);
pushStack(s1, 5);
pushStack(s1, 7);
pushStack(s1, 9);
pushStack(s1, 11);
pushStack(s1, 13);
while (ftemptyStack(s1))
{
popStack(s1, x);
pushStack(s2,x);
}//while
```



Correct

ii.

```
pushStack(s1, 3);
pushStack(s1, 5);
pushStack(s1, 7);
pushStack(s1, 9);
pushStack(s1, 11);
pushStack(s1, 13);
while (ftemptyStack(s1))
{
popStack(s1, x);
popStack(s1, x);
pushStack(s2,x);
}//while
```



Correct

Imagine we have two empty stacks of integers, `s1` and `s2`. Draw a picture of each stack after the following operations:

```
pushStack (s1, 3);
pushStack (s1, 5);
pushStack (s1, 7);
pushStack (s1, 9);
while (!emptyStack (s1)) {
popStack (s1, x);
pushStack (s2, x);
} //while
pushStack (s1, 11);
pushStack (s1, 13);
while (!emptyStack (s2)) {
popStack (s2, x);
pushStack (s1, x);
} //while
```

Solution: