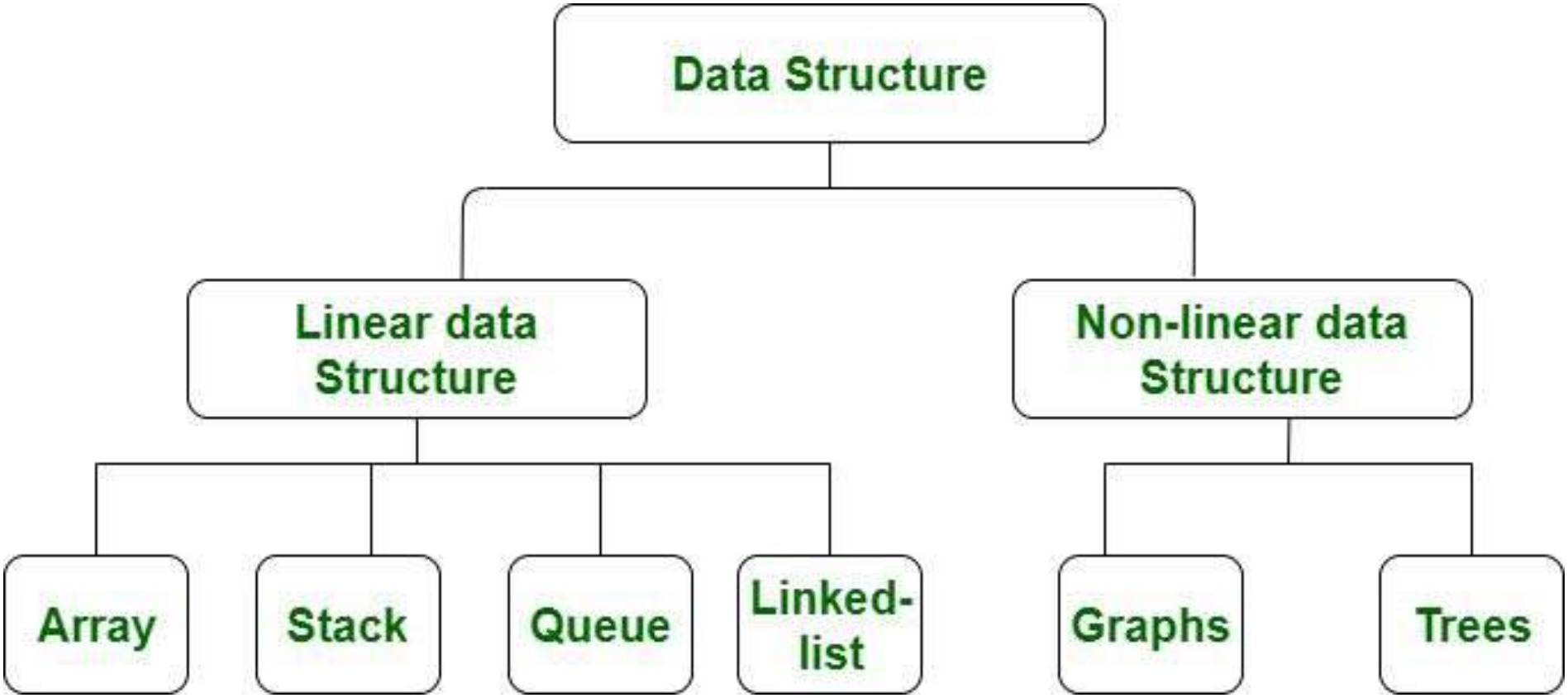
Data Structures

Dr. Ayesha Hakim
Assistant Professor, KJSCE

MODULE 2

LINEAR DATA STRUCTURES


Data Structure
Linear data Structure
Non-linear data Structure
Array
Stack
Queue
Linked-list
Graphs
Trees

Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

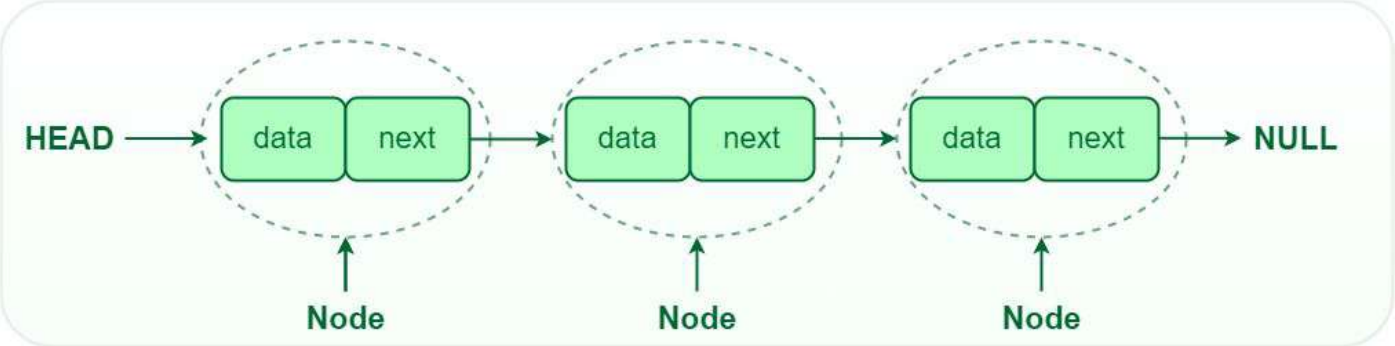
Linear Data Structures

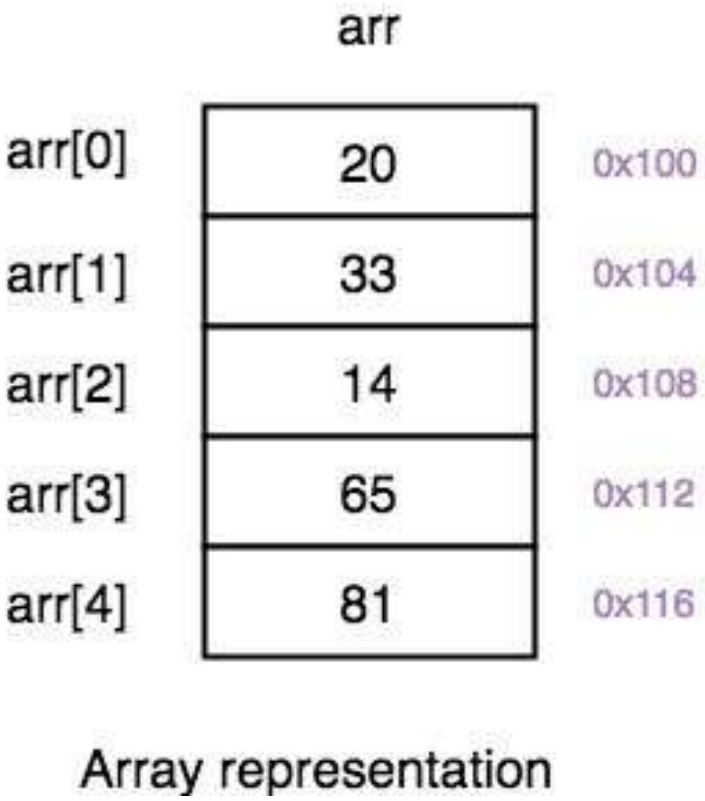
Linear list

- A sequence of elements
- There is first and last element
- Each element has previous and next
  - Nothing before first
  - Nothing after last

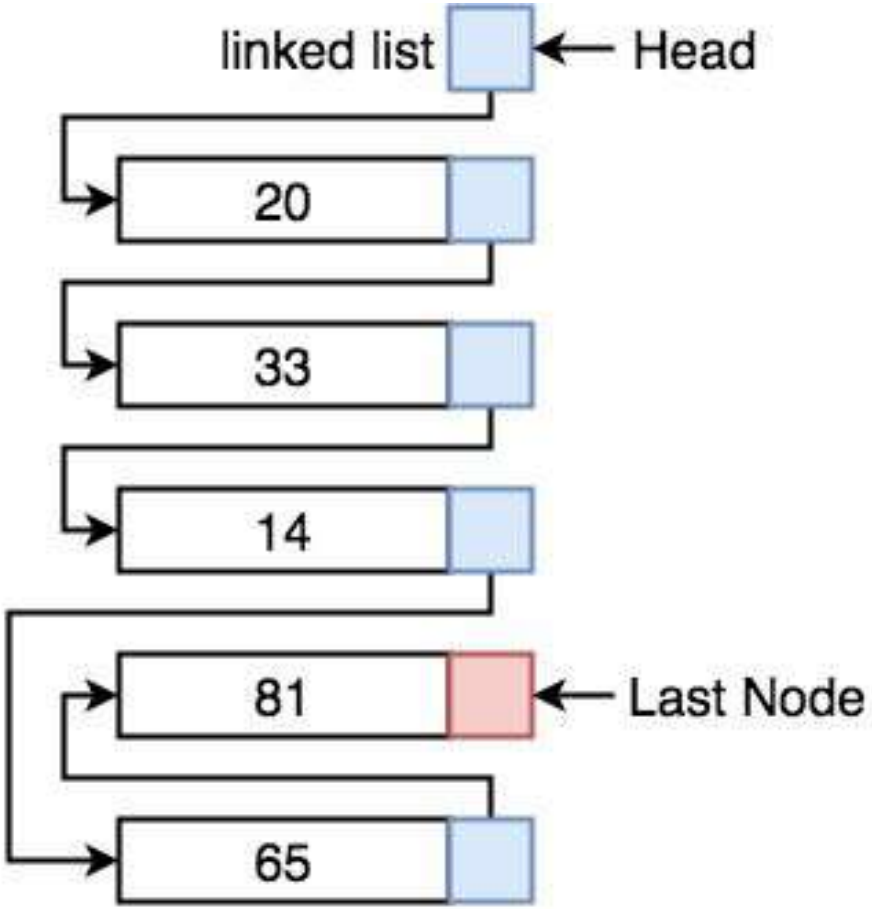
Why linked lists?

- A linked list is a dynamic data structure.
  - It can grow or shrink in size during the execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.





Array representation



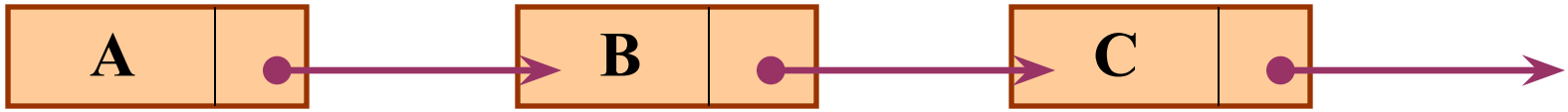
Array Vs. link list

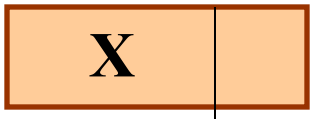
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Operations on LL

- What we can do with a linear list?
  - Delete element
  - Insert element
  - Find element
  - Traverse list

Illustration: Insertion





Item to be inserted

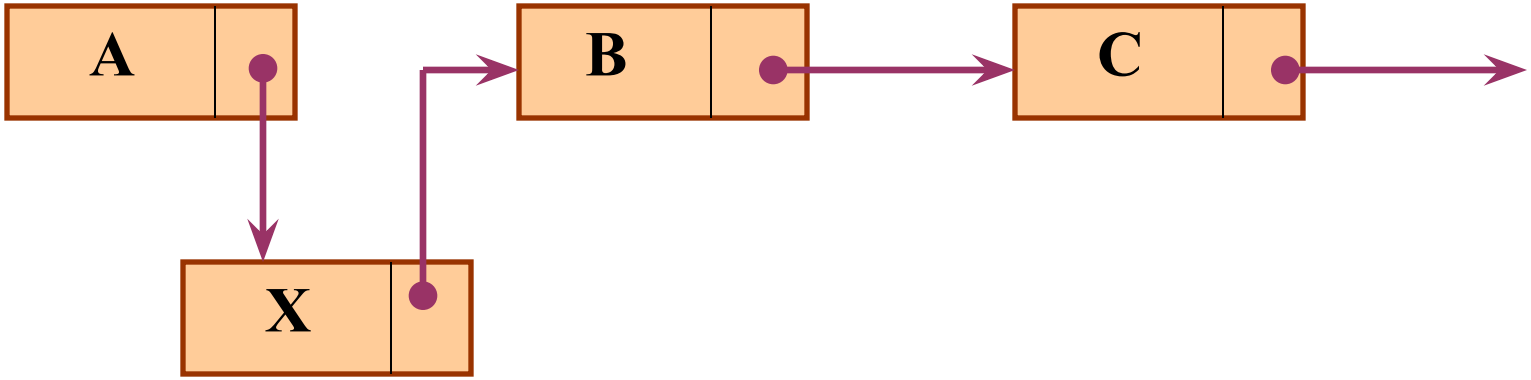
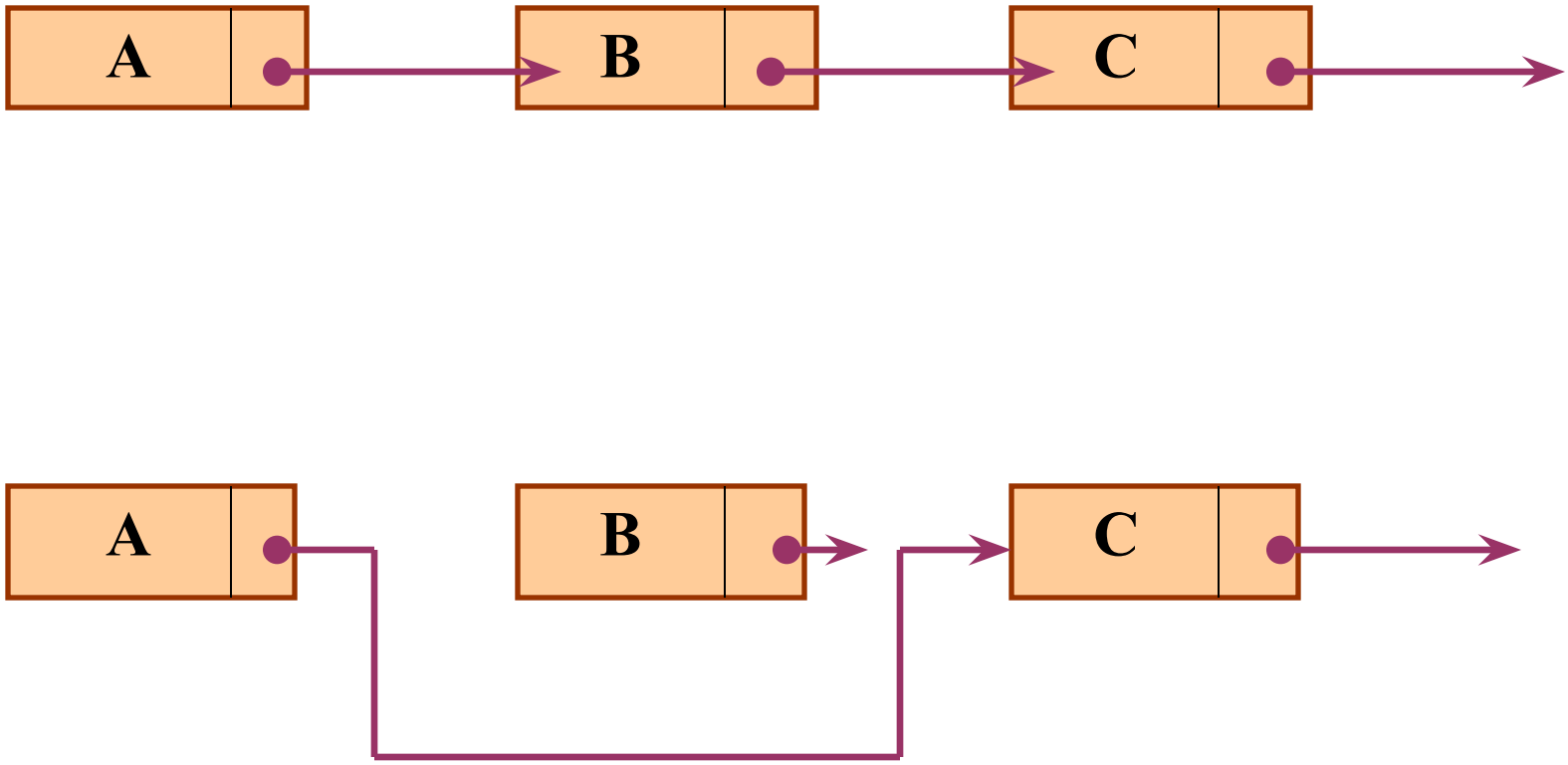
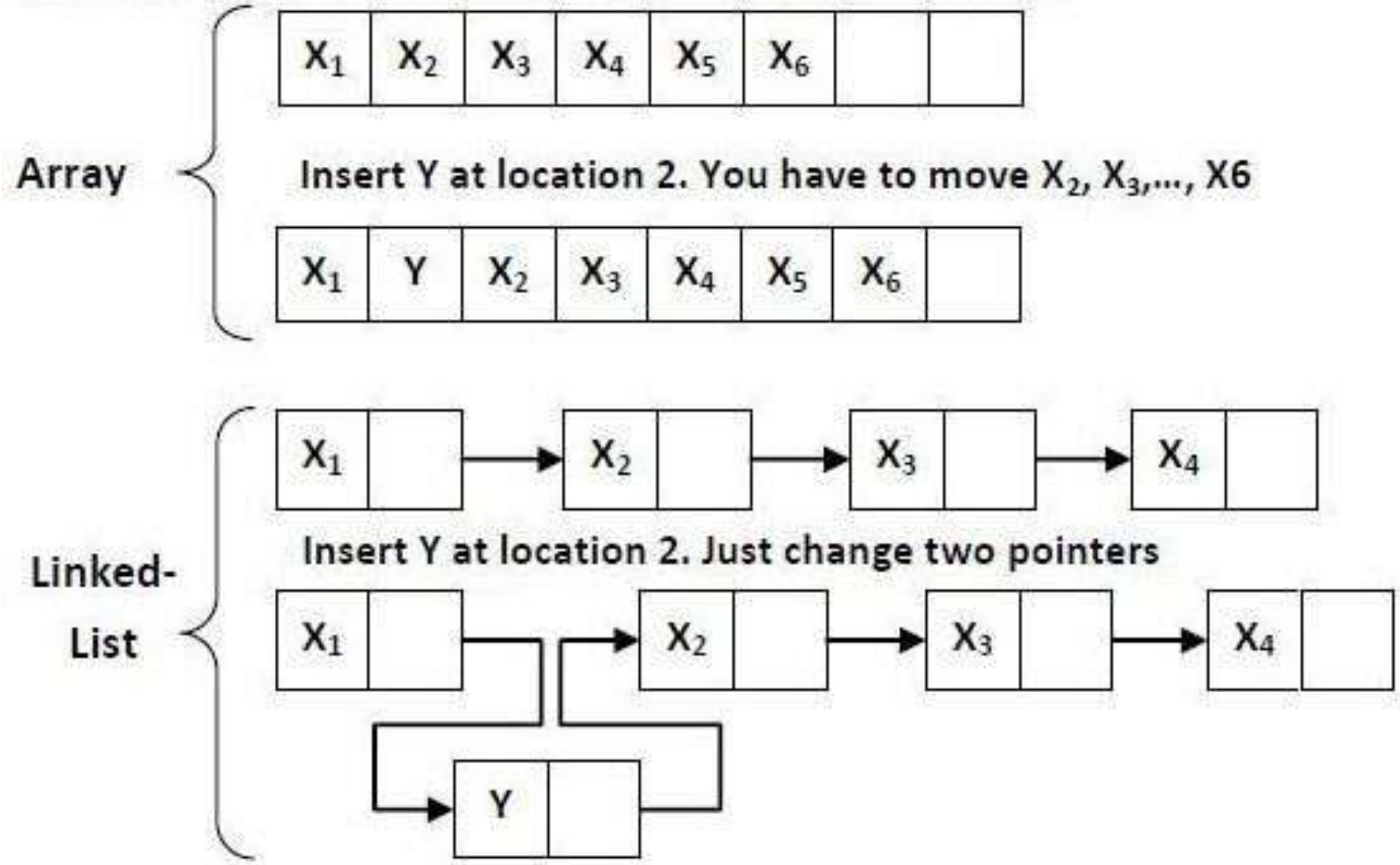


Illustration: Deletion



In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.


Array
X1 X2 X3 X4 X5 X6
Insert Y at location 2. You have to move X2, X3,..., X6
X1 Y X2 X3 X4 X5 X6
Linked-List
X1 X2 X3 X4
Insert Y at location 2. Just change two pointers
X1 X2 X3 X4
Y

Traverse: list ⇨ elements in order

- get_first(list) -
  - returns first element if it exists
- get_next(list) -
  - returns next element if it exists
- Both functions return NULL otherwise
- Calling get_next in a loop we will get one by one all elements of the list

How we can implement a list?

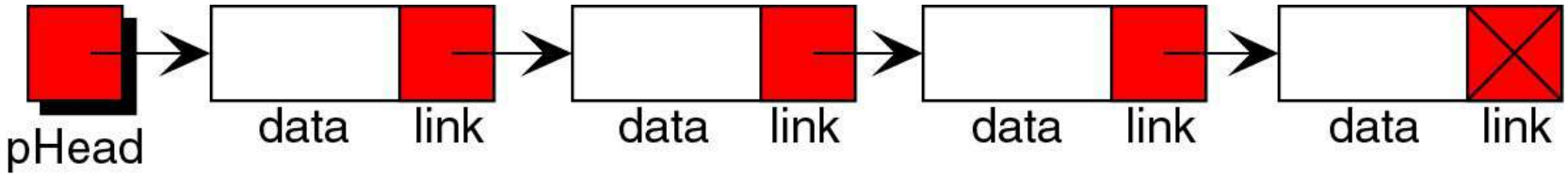
- Array?
- Search is easy (sequential or binary)
- Traversal is easy:

```
for(i = first; i <= last; ++i)
    process(a[i]);
```

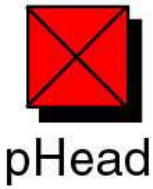
- Insert and delete is *not* easy
  – a good part of the array has to be moved!
- Hard to guess the size of an array

A *linked list* implementation

- Linked list is a chain of elements
- Each element has data part and link part pointing to the next element



(a) A linked list with a head pointer: pHead

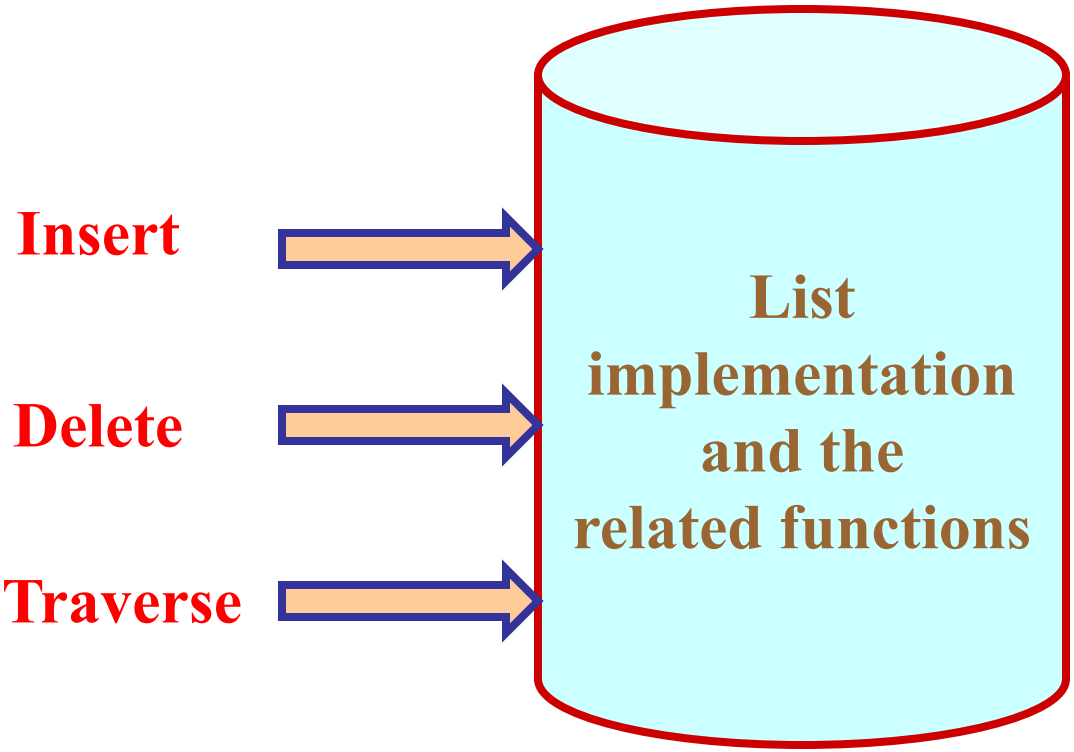


(b) An empty linked list

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:

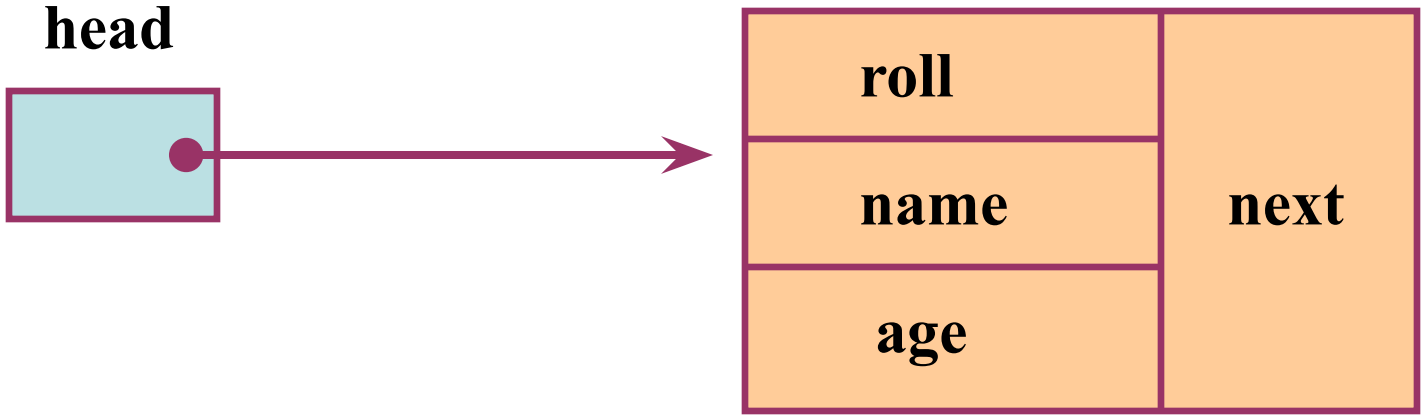
```
struct  stud  {
                int     roll;
                char  name[25];
                int     age;
                struct  stud  *next;
              };

        /* 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:

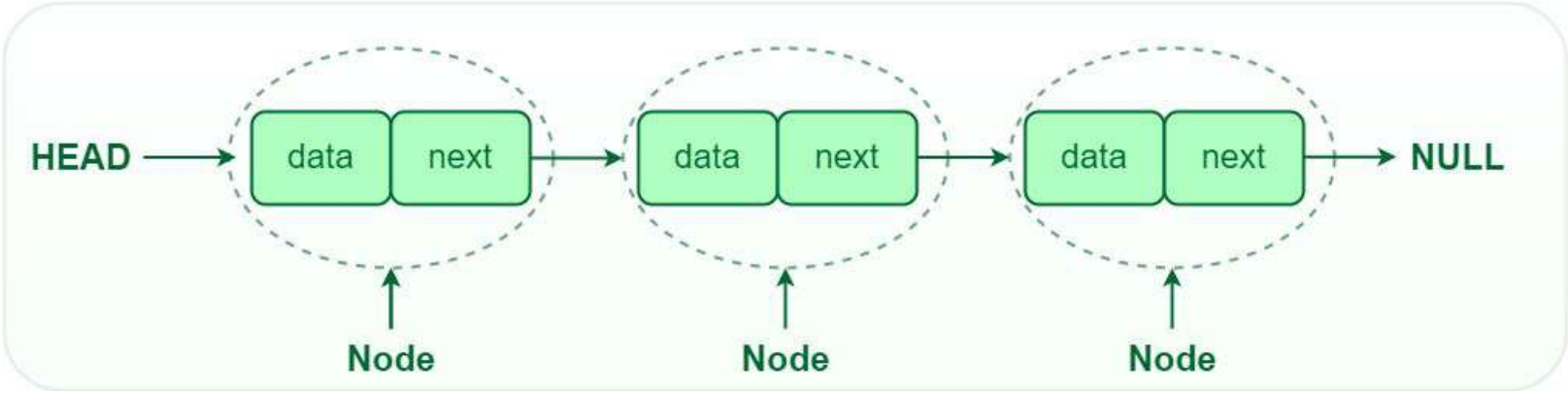
```
node  *head;
.......
create_list (head);
```

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

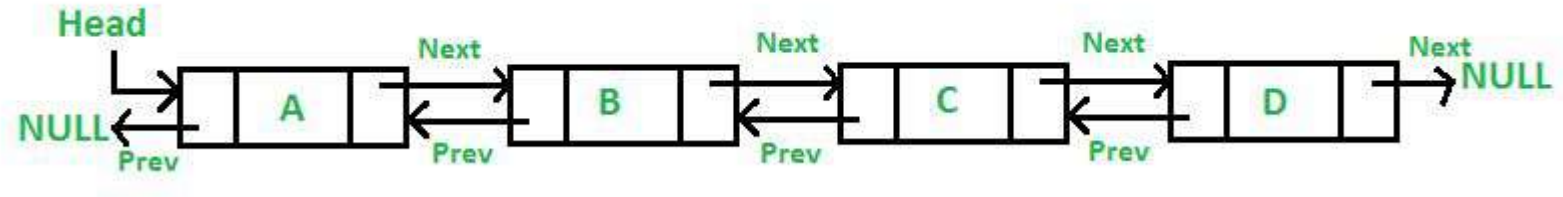
Singly Linked List: A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

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.

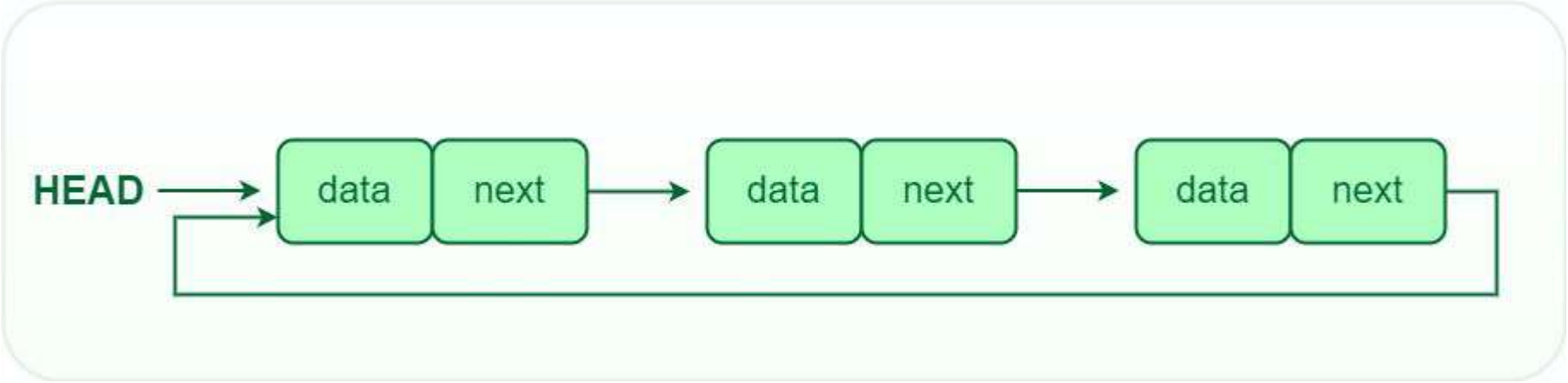
In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it **requires additional memory for the backward reference.**

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

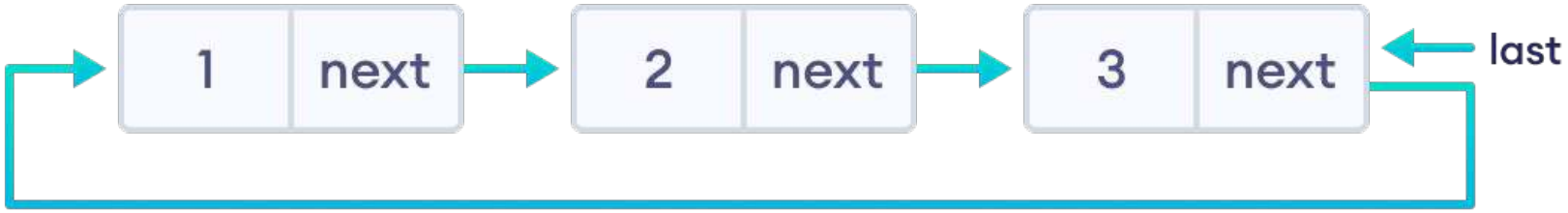
A doubly linked list is a type of linked list in which each node consists of 3 components:

- *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



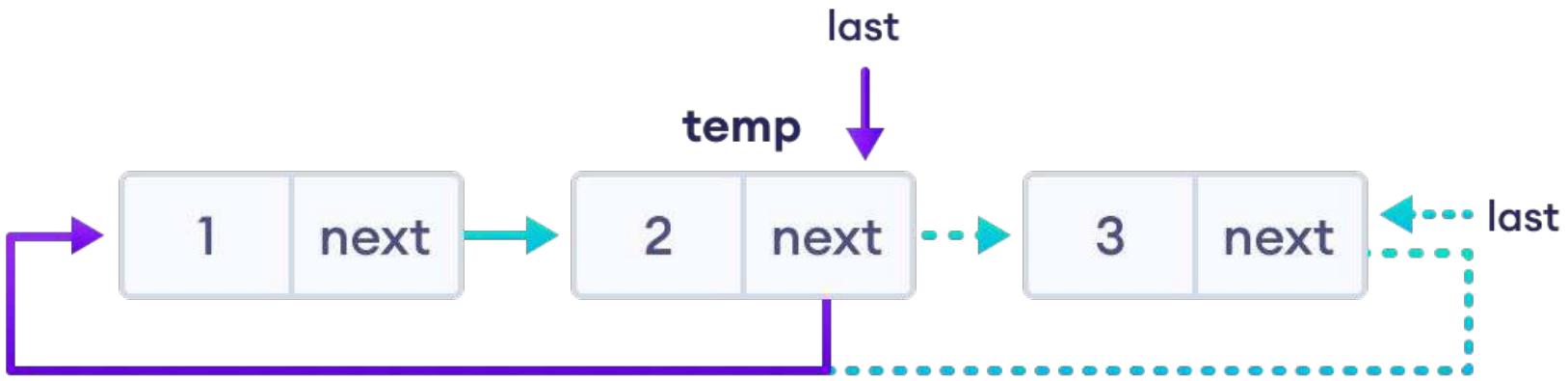
1. If the node to be deleted is the only node

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

2. If last node is to be deleted

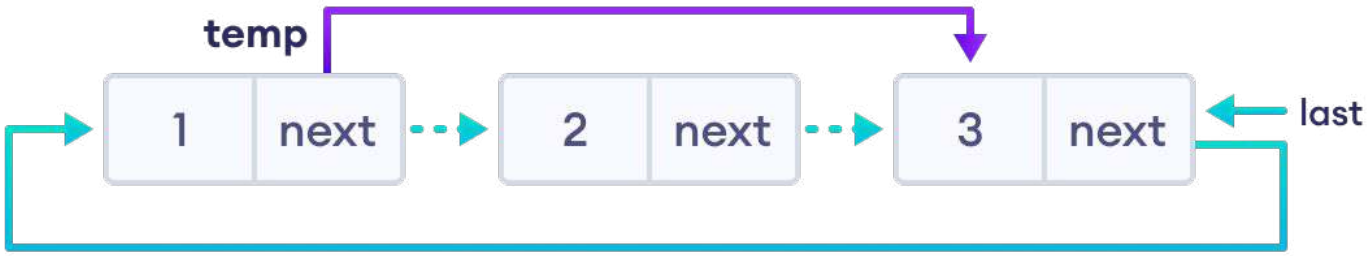
- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node

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:

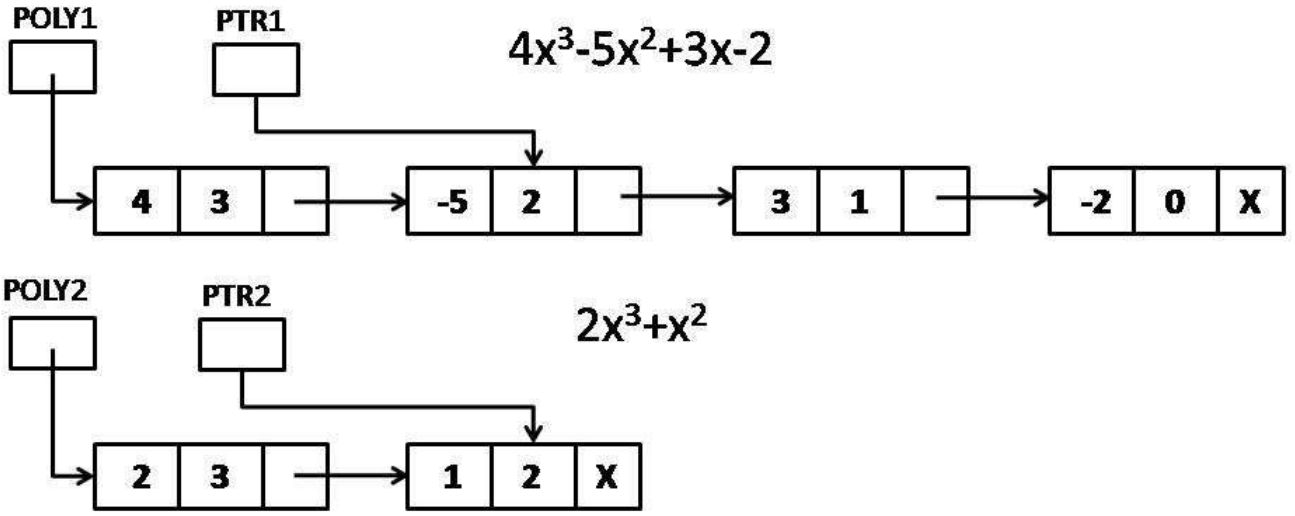
- one is the coefficient
- other is the exponent

Example:

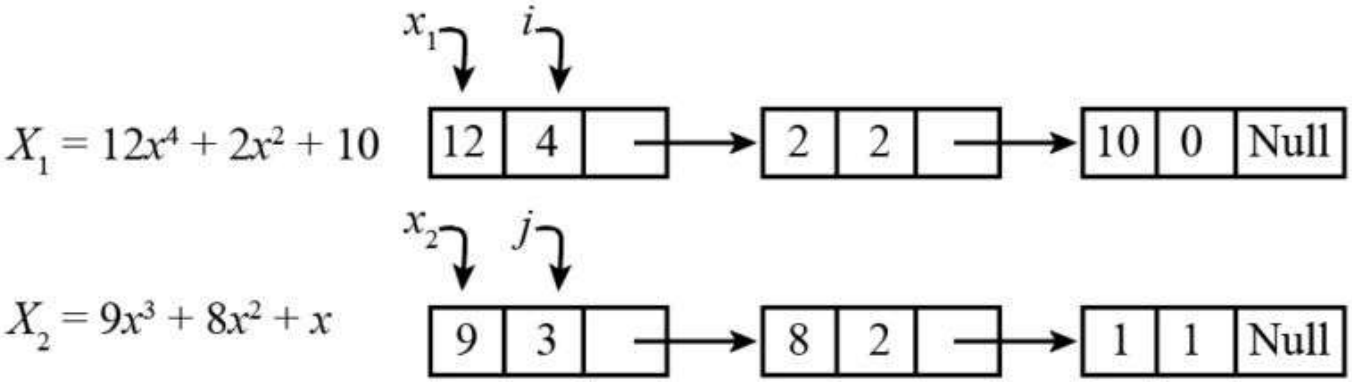
$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

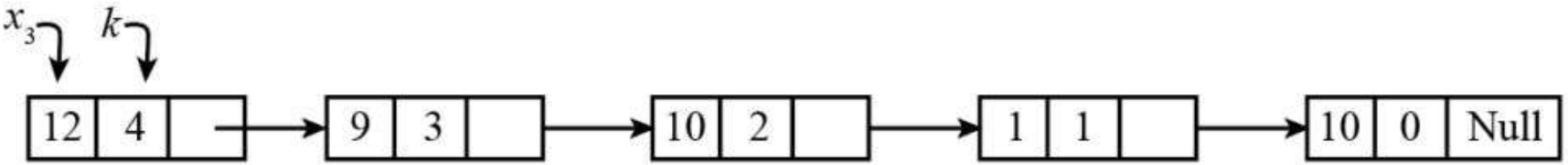
- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent
- For adding two polynomials that are stored as a linked list, we need to add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

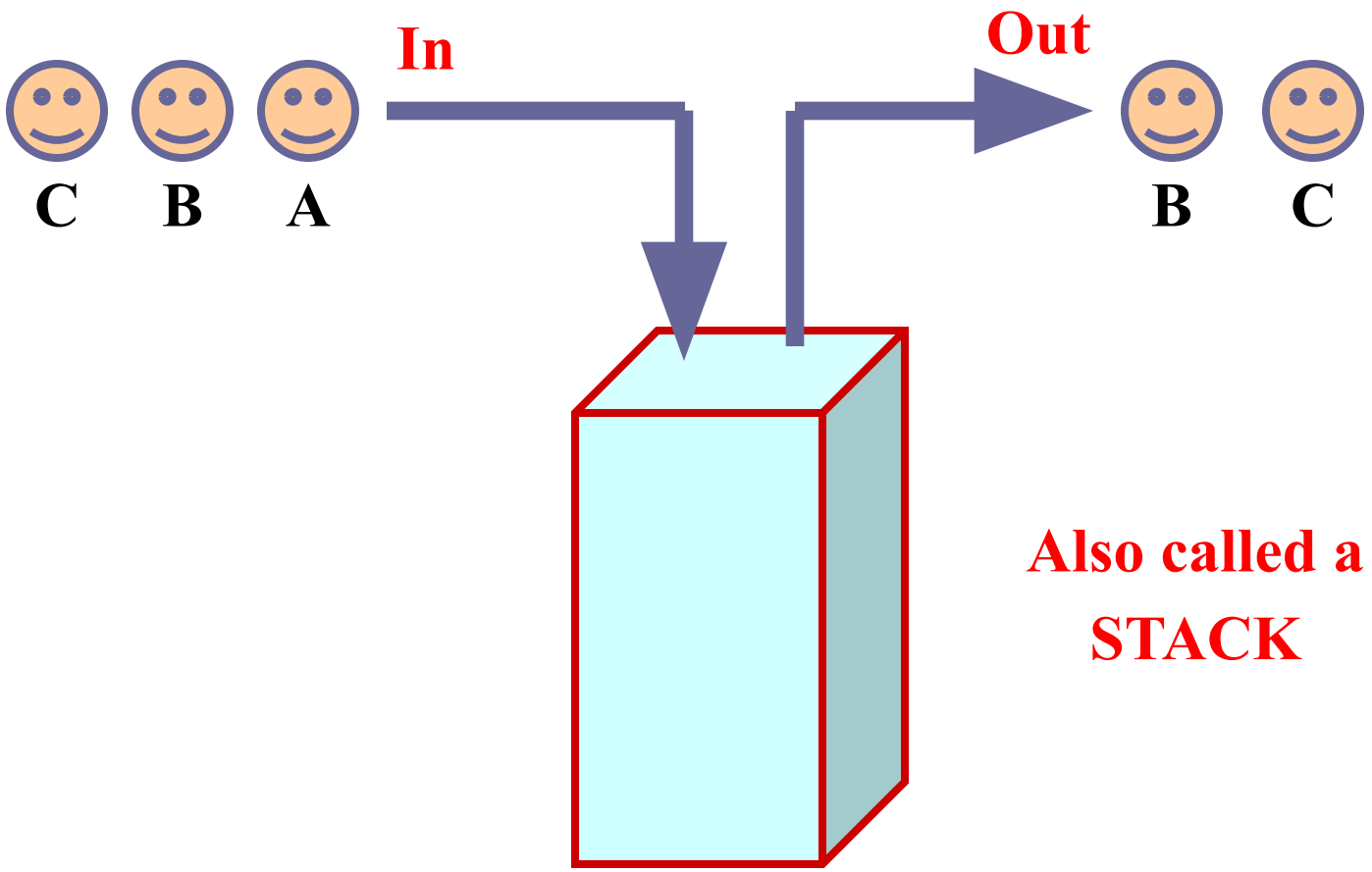
1. It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.
2. Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can 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:

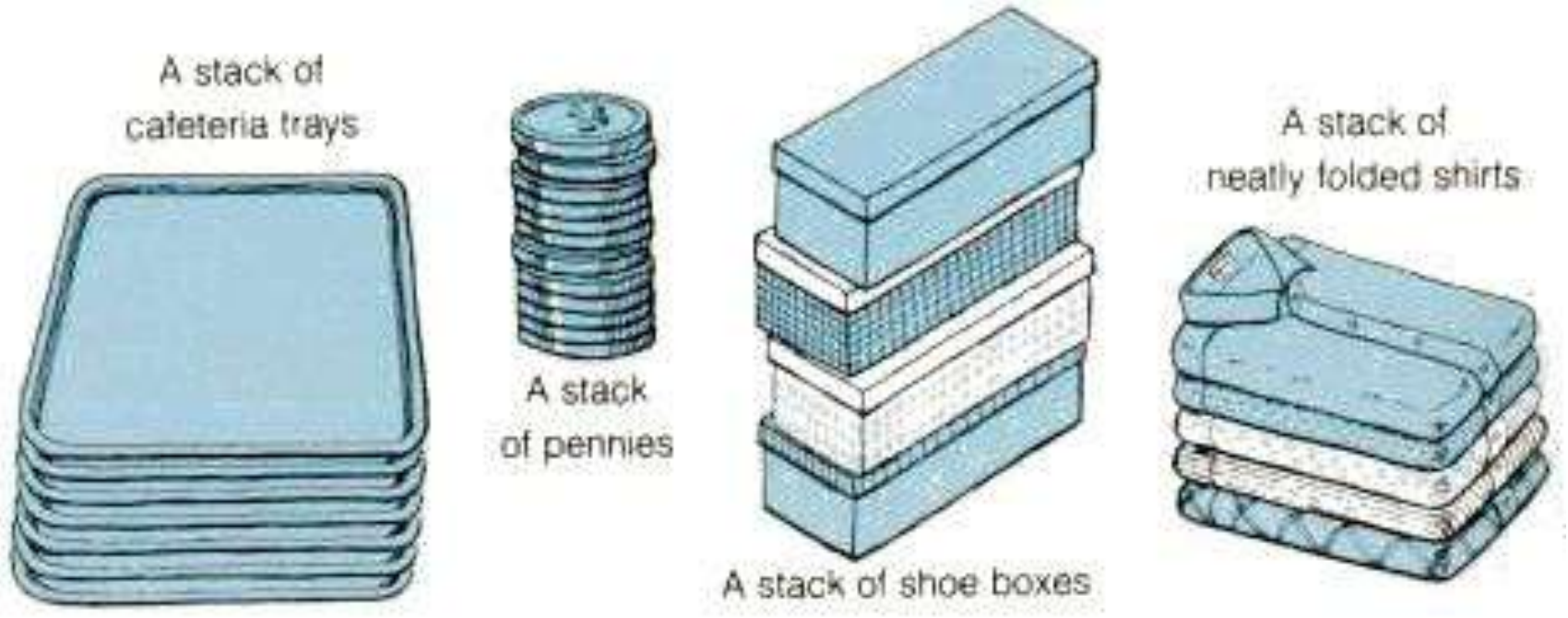
1. Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
2. In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
3. Again, it is easily possible to implement other data structures 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

A Last-in First-out (LIFO) List

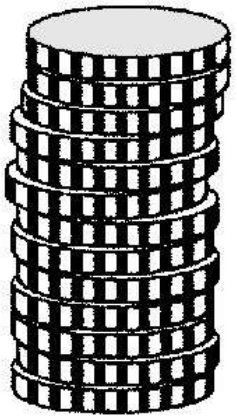


Stacks in Our Life

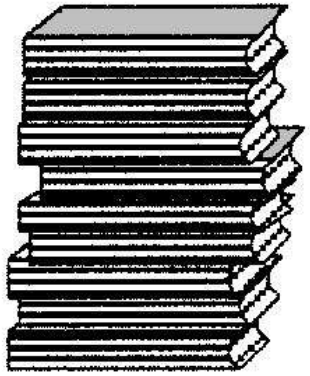


- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.

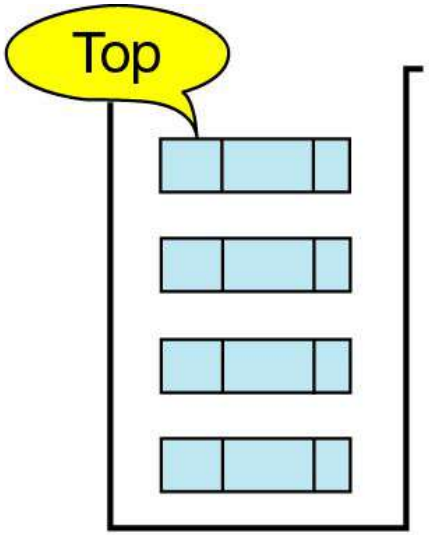
More Stacks



Stack of coins



Stack of books

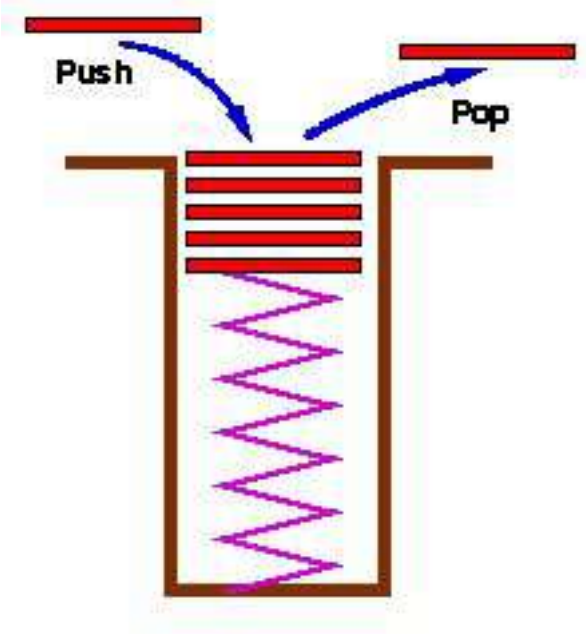


Computer stack

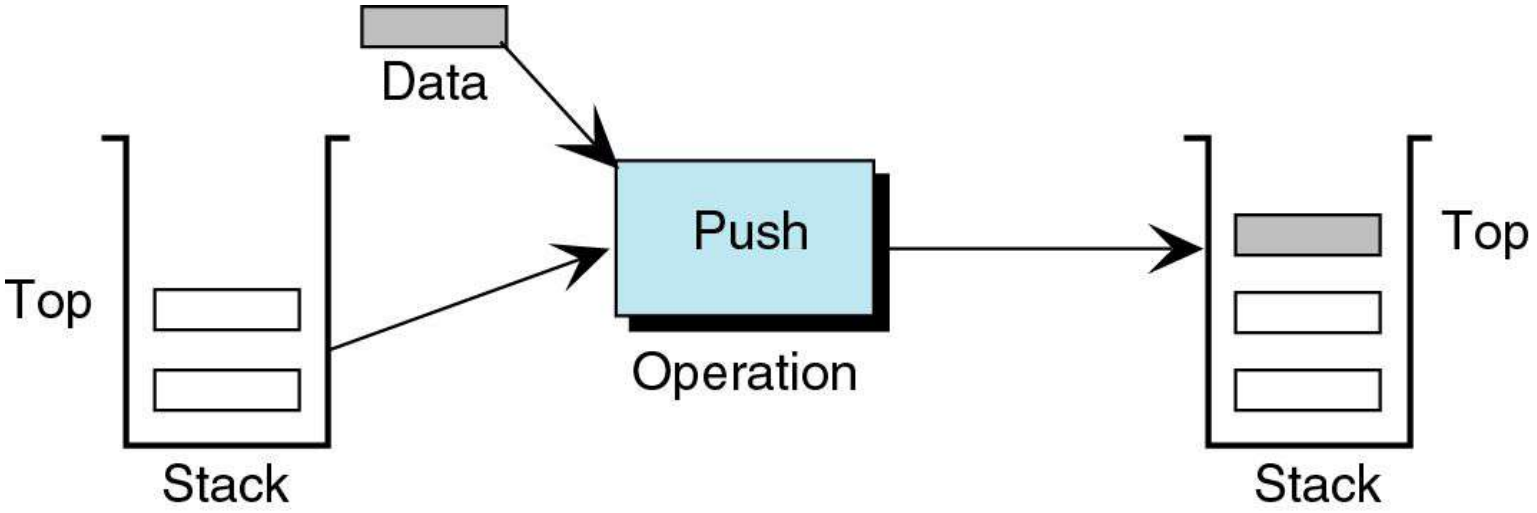
- A stack is a LIFO structure: Last In First Out

Basic Operations with Stacks

- Push
  - Add and item
    - Overflow
- Pop
  - Remove an item
    - Underflow
- Stack Top
  - What's on the Top
    - Could be empty

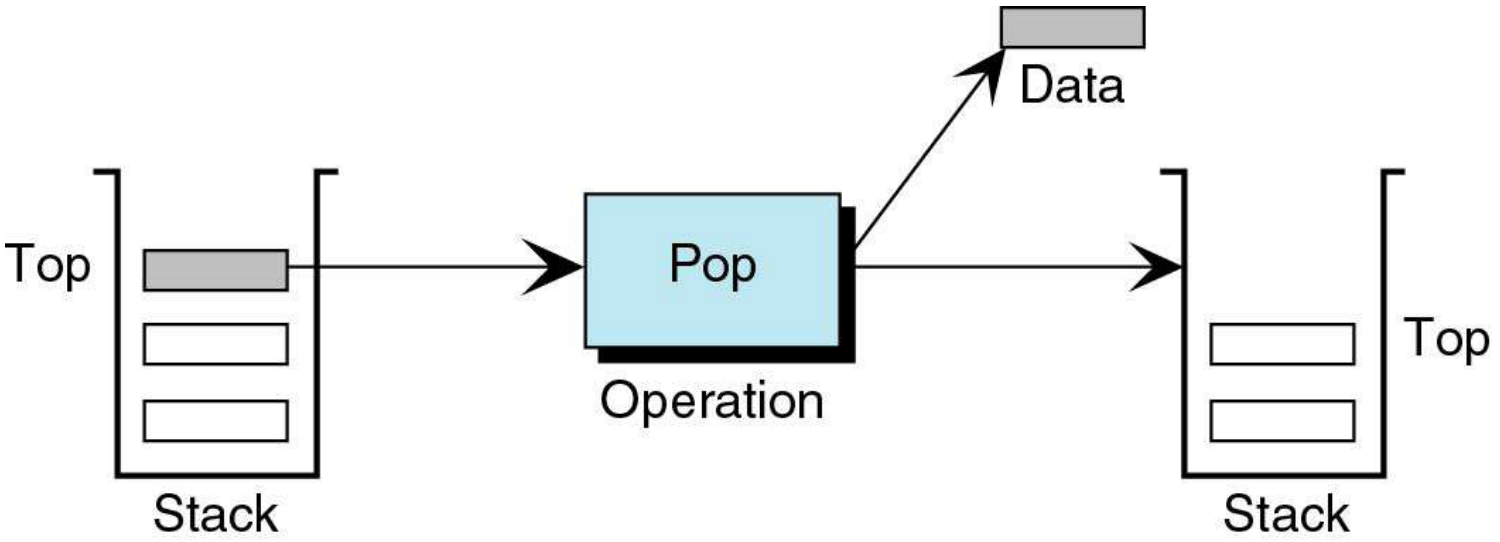


Push



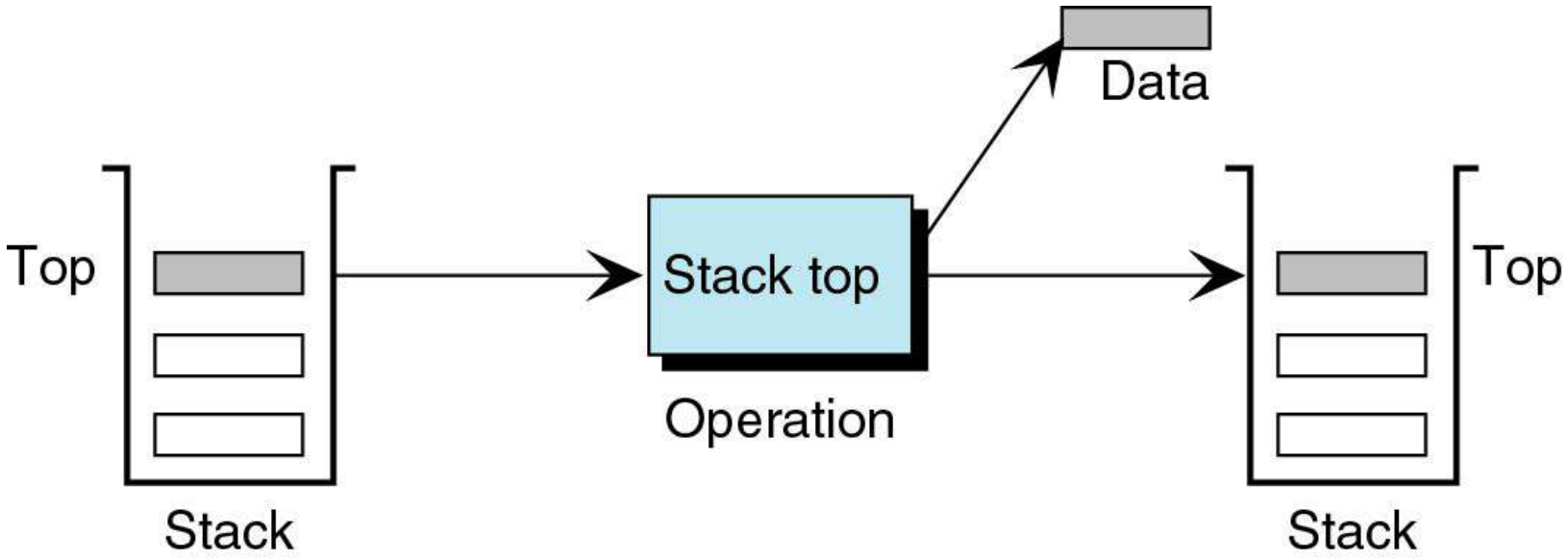
- Adds new data element to the top of the stack

Pop



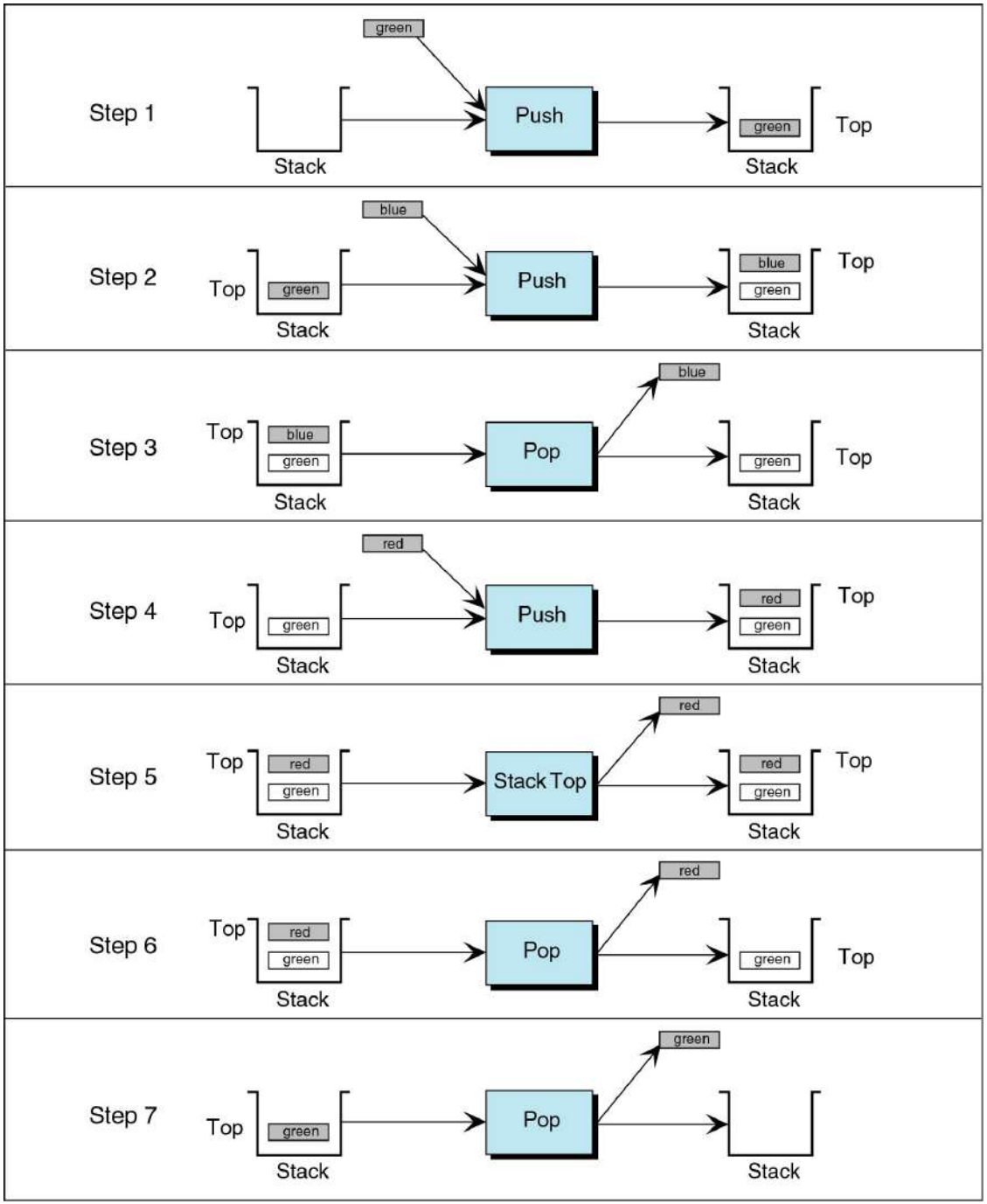
- Removes a data element from the top of the stack

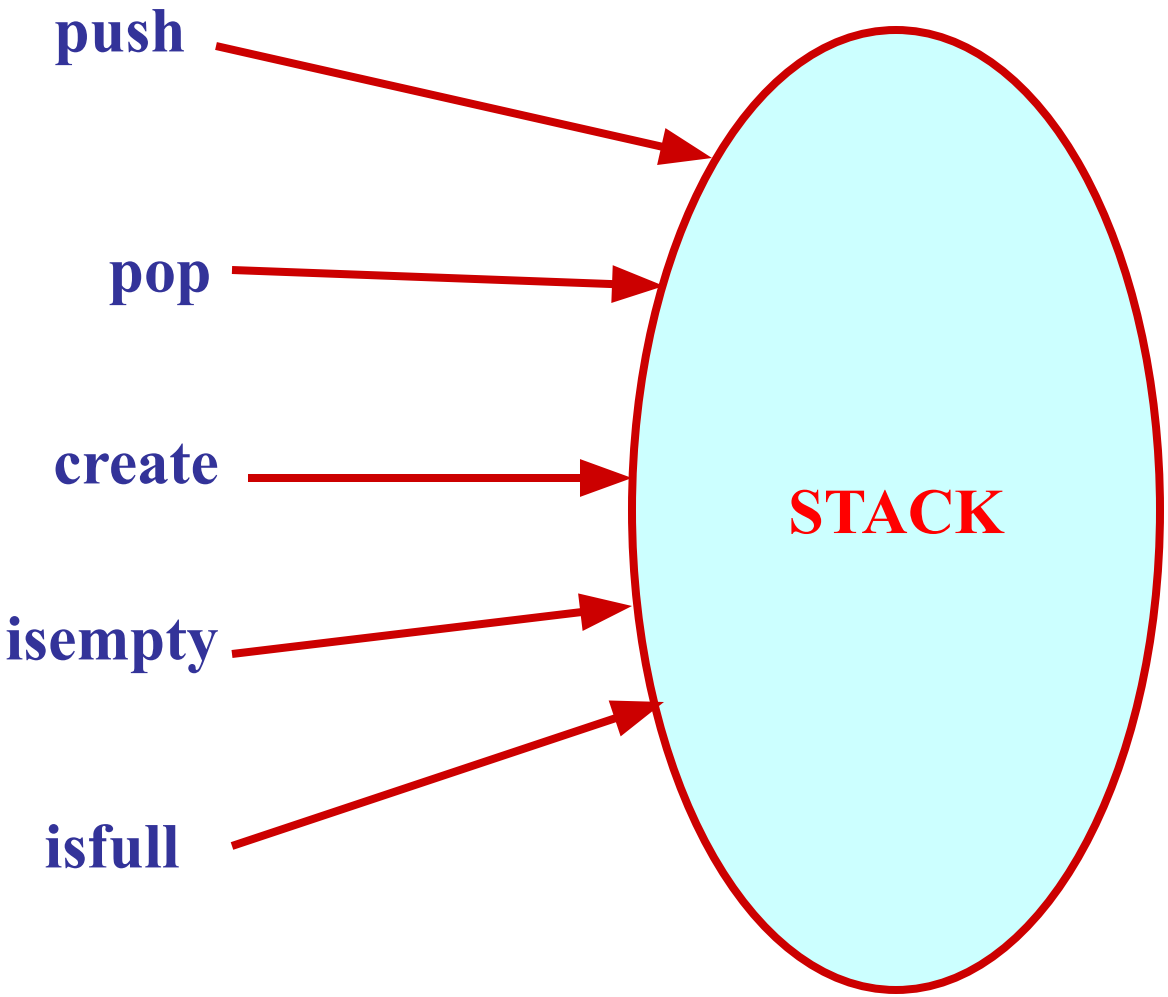
Stack Top



- Checks the top element. Stack is not changed

When an element is “pushed” onto the stack, it becomes the first item that will be “popped” out of the stack. In order to reach the oldest entered item, you must pop all the previous items.




push
pop
create
isempty
isfull
STACK

Assume:: stack (LIFO) contains integer elements

```
void  push  (stack s,  int element);
                                    /* Insert an element in the stack */
int  pop (stack s);
                                /* Remove and return the top element */
void  create  (stack  s);
                                     /* Create a new stack */
int  isempty  (stack  s);
                                     /* Check if stack is empty */
int  isfull  (stack  s);
                                     /* Check if stack is full */
```

- We shall look into two different implementations of stack:
  - Using **arrays** **(static implementation)**
  - Using **linked list** **(dynamic implementation)**

Stack Implementation

Stack Implementation Using Arrays

- Basic idea.
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the "**top**" of the stack.

Underlying Mechanics of Stacks

Initially, a pointer (`top`) is set to keep the track of the topmost item in the stack. The stack is initialized to `-1`.

Then, a check is performed to determine if the stack is empty by comparing `top` to `-1`.

As elements are added to the stack, the position of `top` is updated.

As soon as elements are popped or deleted, the topmost element is removed and the position of `top` is updated.

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);
}
```

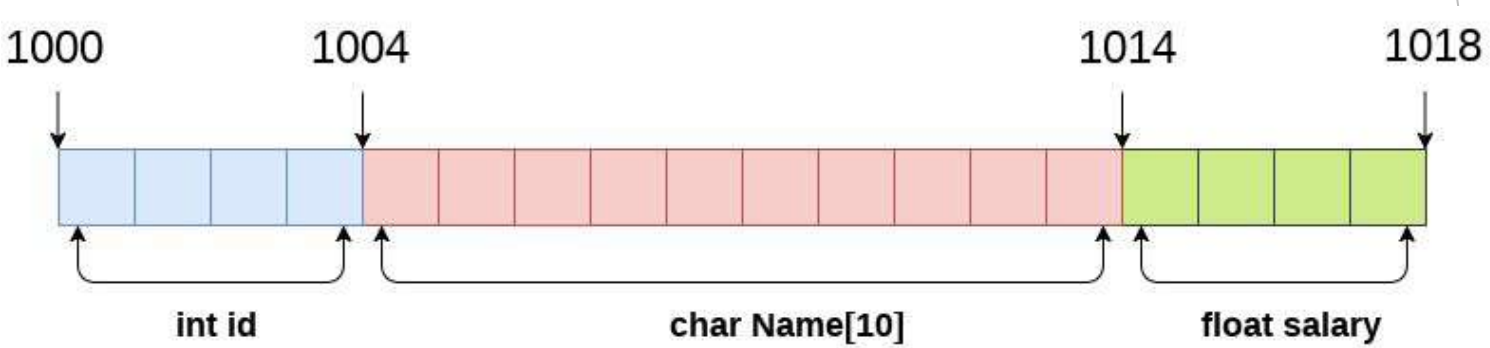
```
printf("\nStudent Information List:");
for(i=0;i<4;i++)
{
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

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.

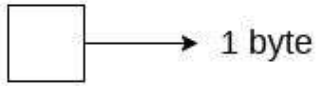
Memory Allocation in Structure




```
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.

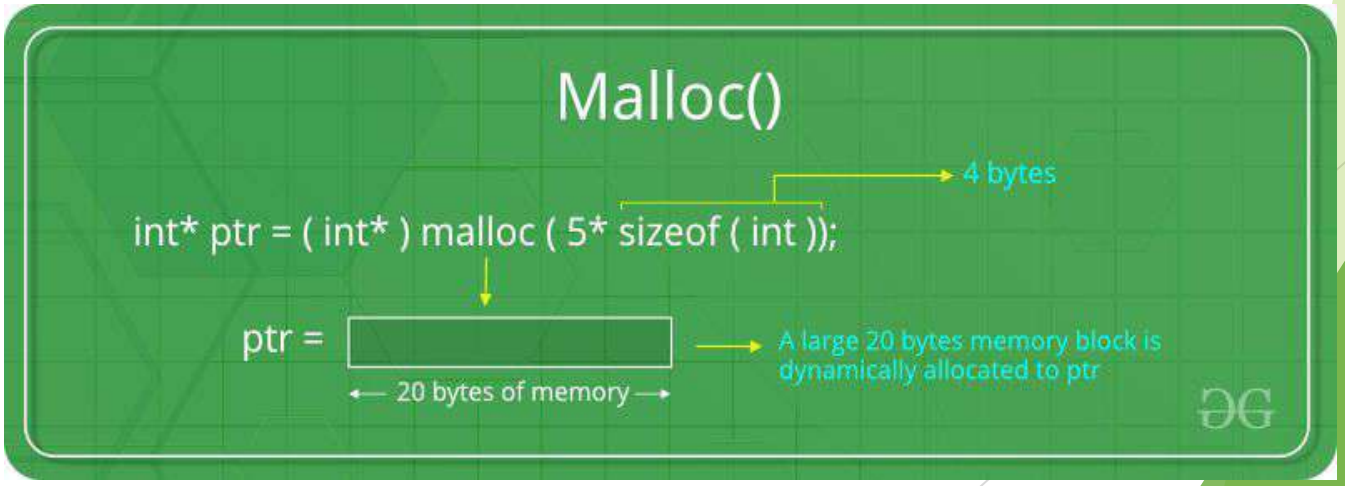
Syntax:

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

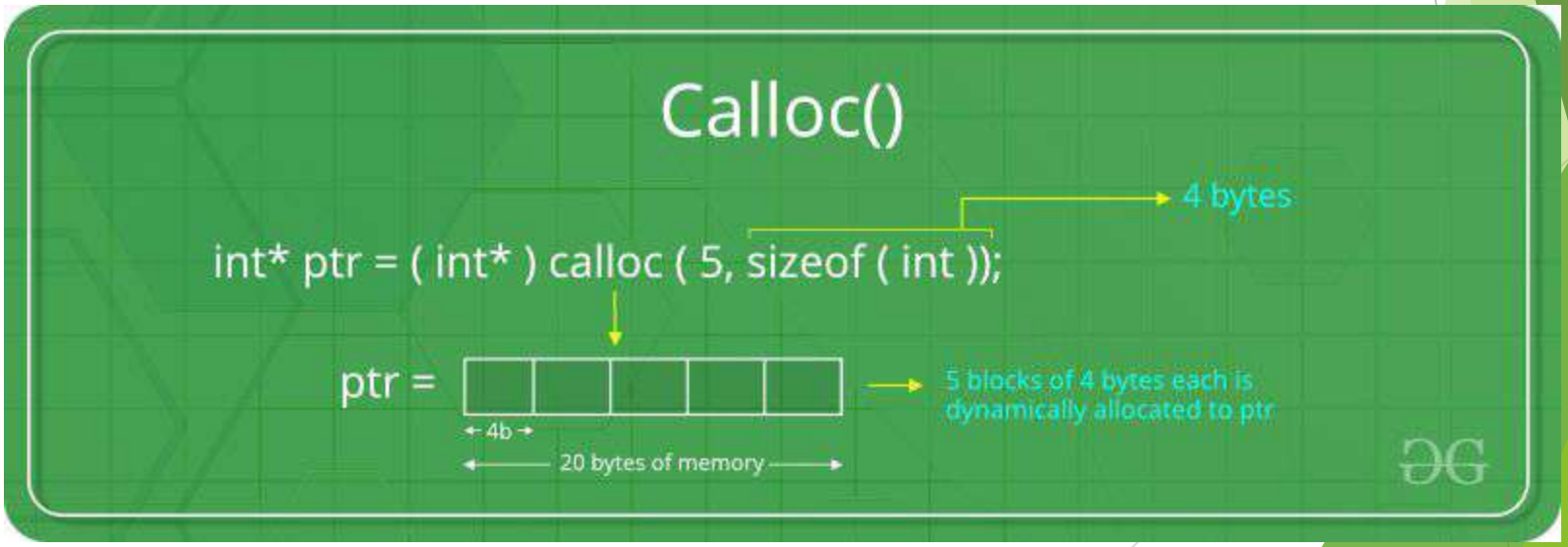
“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

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

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

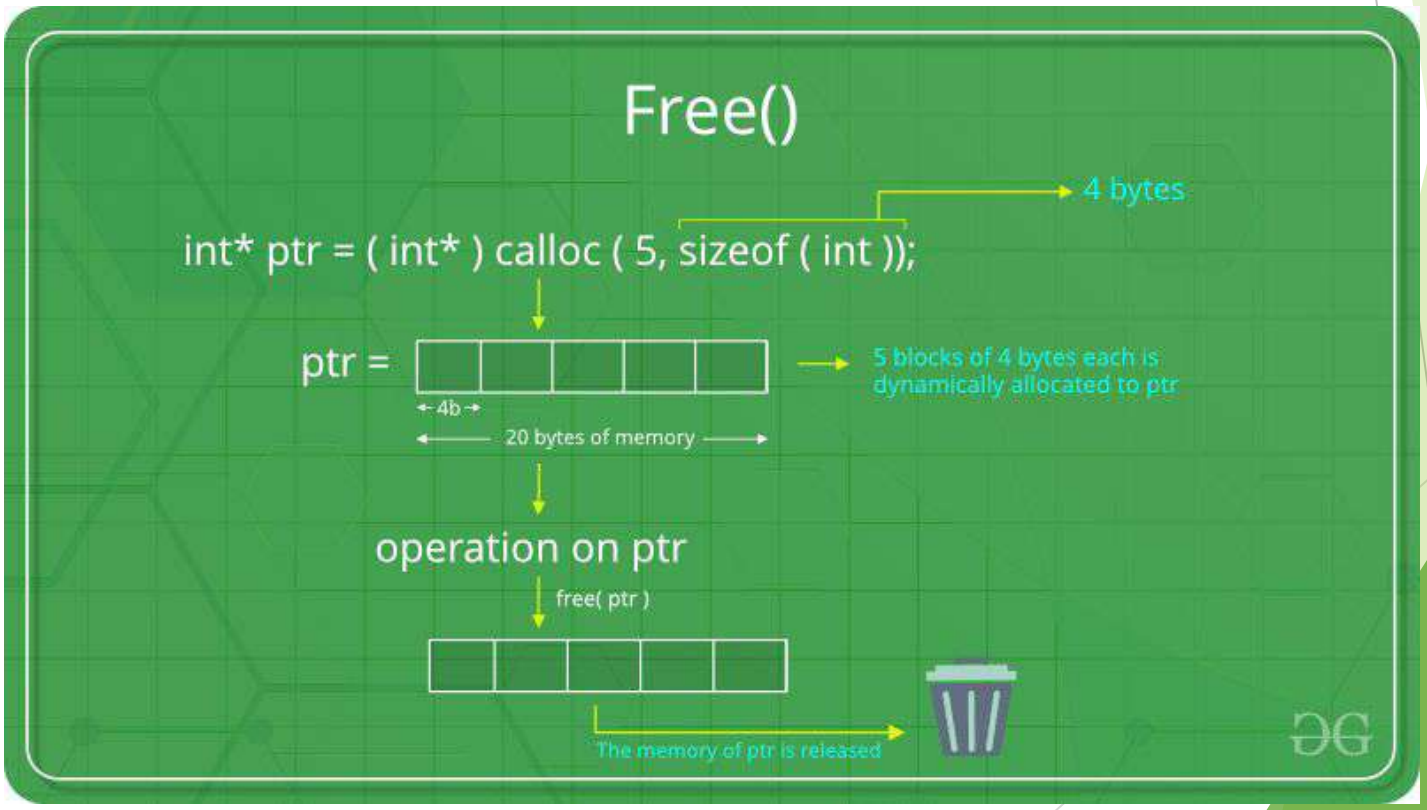
ptr = (cast-type*)calloc(n, element-size);

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



C free() method

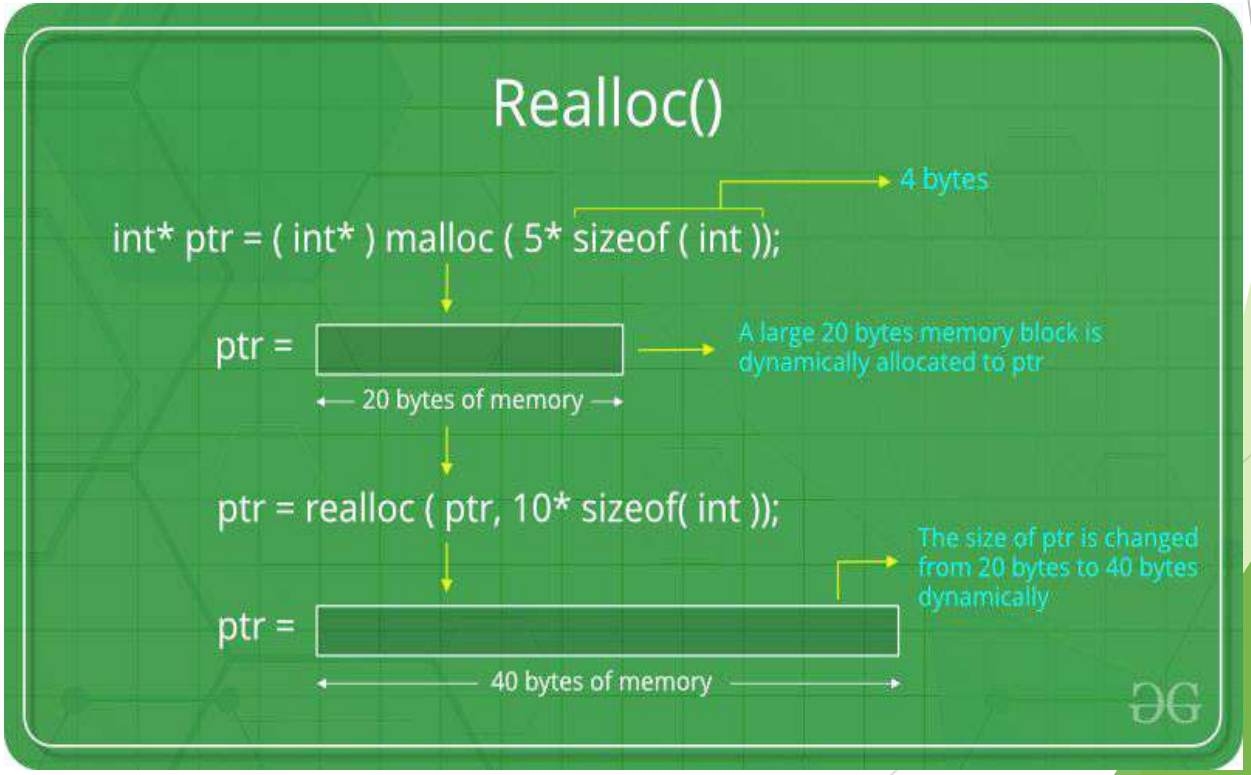
“free” method in C is used to dynamically **deallocate** the memory. The memory allocated using functions malloc() and calloc() is not deallocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.



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

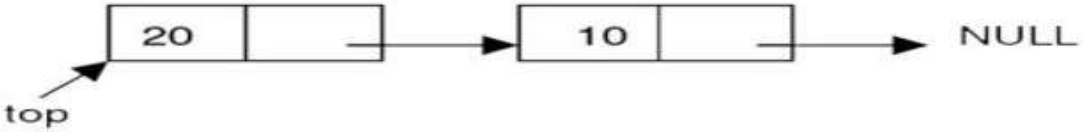
Understanding PUSH

head to SLL is top to STACK

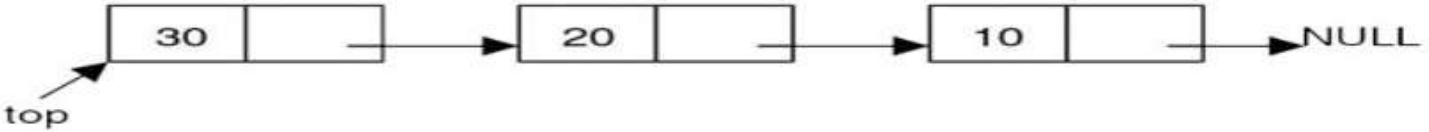
Initially

insertBegin to SLL is push to STACK

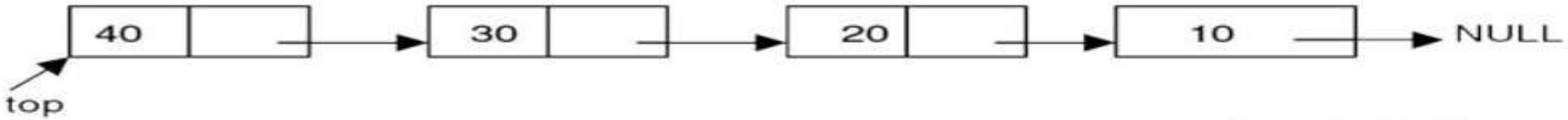
After first iteration



After second iteration



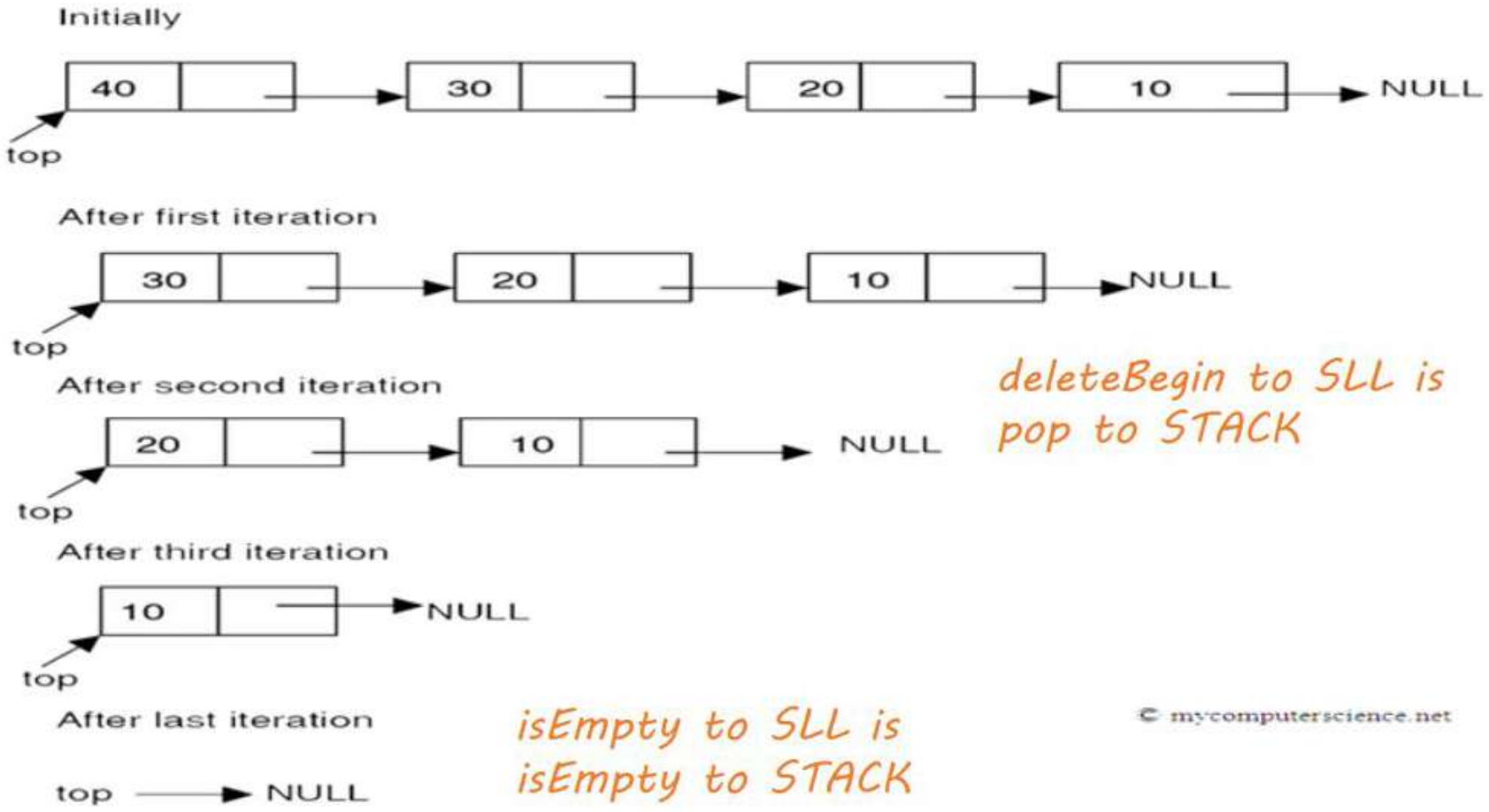
After third iteration



After last iteration

Stack using SLL

Understanding POP



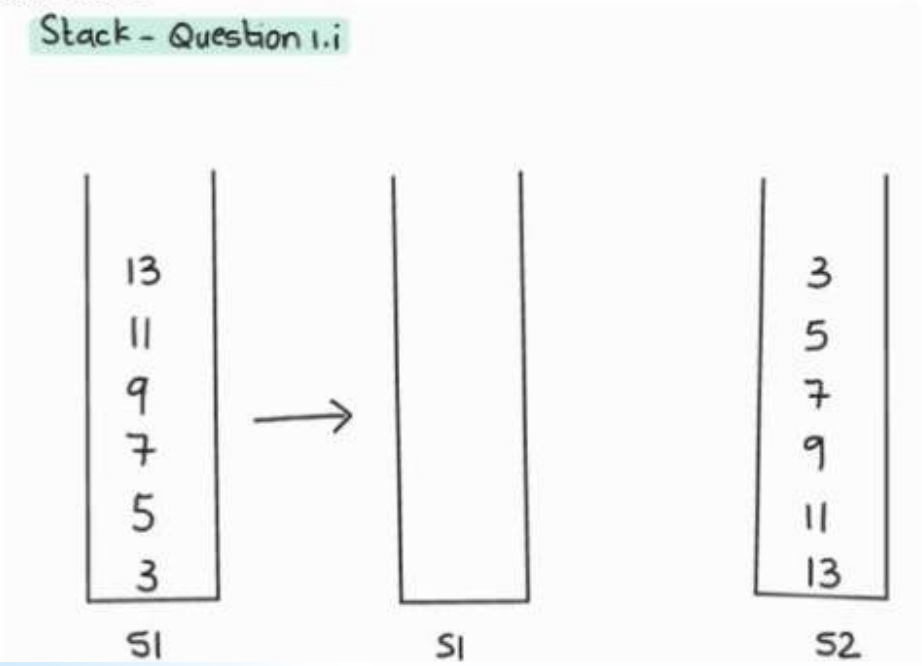
deleteBegin to SLL is pop to STACK

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.

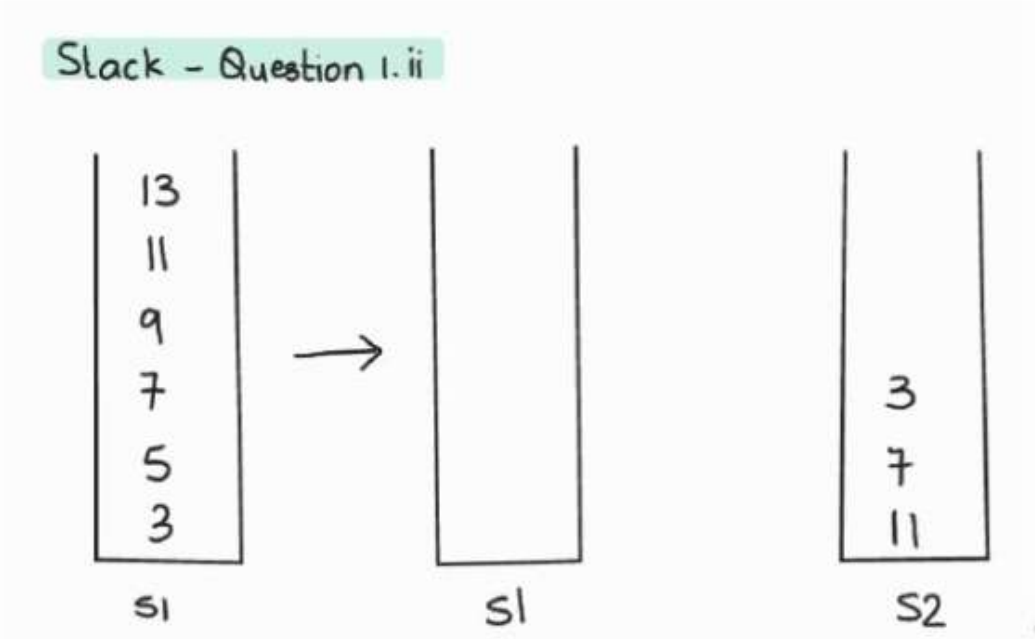
```
pushStack(s1, 3);
pushStack(s1, 5);
pushStack(s1, 7);
pushStack(s1, 9);
pushStack(s1, 11);
pushStack(s1, 13);
while (ftemptyStack(s1))
{
popStack(s1, x);
pushStack(s2,x);
}//while
```



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
```

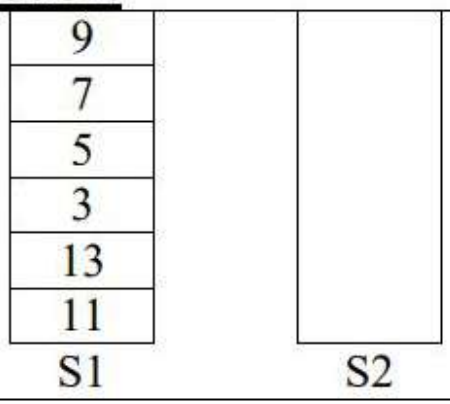


Correct

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:



Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Used to solve problem like Infix to Prefix Conversion
- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

STACK VERSUS LINKED LIST

STACK	LINKED LIST
An abstract data type that serves as a collection of elements with two principal operations which are push and pop	A linear collection of data elements whose order is not given by their location in memory
Push, pop and peek are the main operations performed on a stack	Insert, delete and traversing are the main operations performed on a linked list
Can read the topmost element	It is required to traverse through each element to access a specific element
Works according to the FIFO mechanism	Elements connect to each other by references
Simpler than linked list	More complex than stack

Application of Stacks

- Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule.
- In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions.

Infix and Postfix Notations

- Infix: operators placed between operands:

  A+B*C

- Postfix: operands appear before their operators:-

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C □ (A + (B * C)) □ (A + (B C *)) □ A B C * +

A + B * C + D □ ((A + (B * C)) + D) □ ((A + (B C*))+ D) □
((A B C *+) + D) □ A B C * + D +

Evaluation of Postfix Expression

Postfix expression: 6 5 2 3 + 8 * + 3 + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	**288**	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

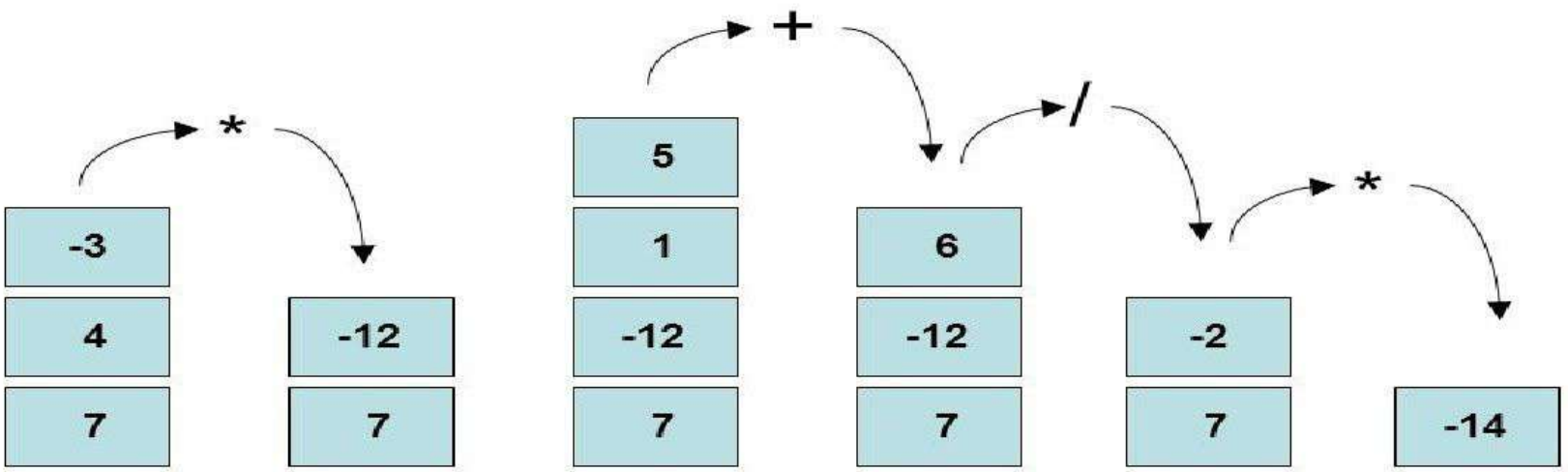
In **post-fix notation**, the operands come before their operators.

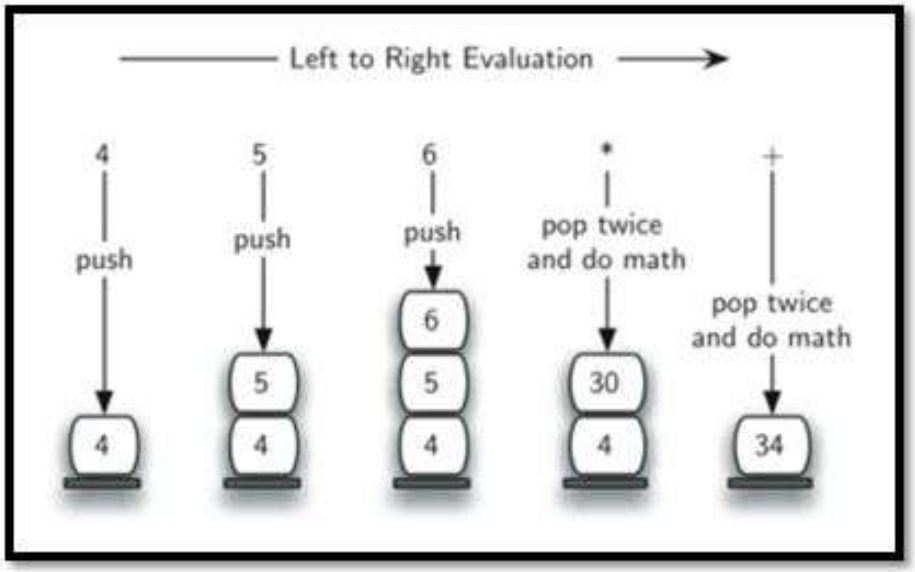
For the evaluation of post-fix notation, we use the **stack** data structure. The following are the **rules** of evaluating post-fix notation using stack:

- Start scanning from left to right.
- If the current value is an operand, push it onto the stack.
- If the current is an operator, pop two elements from the stack, apply the at-hand operator on those two popped operands, and push the result onto the stack.
- At the end, pop an element from the stack, and that is the answer.

Evaluating Postfix Expressions

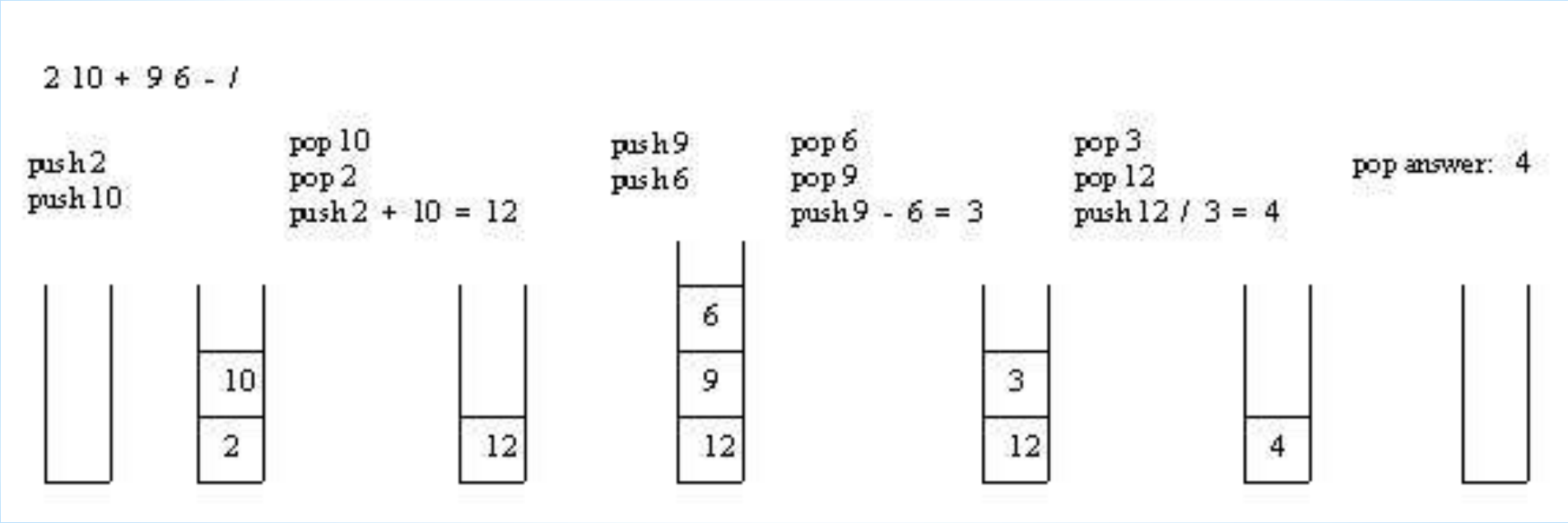
- Expression = 7 4 -3 * 1 5 + / *





Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Evaluation of Postfix Expression (using Stack)



Code:
https://www.geeksforgeeks.org/evaluation-of-postfix-expression/

Evaluation of Prefix Expression (using Stack)

Example: -*+4325

Symbol	opnd1	opnd2	value	opndstack
5				5
2				5, 2
3				5, 2, 3
4				5, 2, 3, 4
+	4	3	7	5, 2
				5, 2, 7
*	7	2	14	5
				5, 14
-	14	5	9	
				9

result

Algorithm:

```
EVALUATE_PREFIX(STRING)
Step 1: Put a pointer P at the end of the end
Step 2: If character at P is an operand push it to Stack
Step 3: If the character at P is an operator pop two
        elements from the Stack. Operate on these elements
        according to the operator, and push the result
        back to the Stack
Step 4: Decrement P by 1 and go to Step 2 as long as there
        are characters left to be scanned in the expression.
Step 5: The Result is stored at the top of the Stack,
        return it
Step 6: End
```

```
Expression: +9*26

Character | Stack       |  Explanation
Scanned   | (Front to   |
          |  Back)      |
-------------------------------------------
6           6             6 is an operand,
                            push to Stack
2           6 2           2 is an operand,
                            push to Stack
*           12 (6*2)      * is an operator,
                          pop 6 and 2, multiply
                          them and push result
                          to Stack
9           12 9          9 is an operand, push
                          to Stack
+           21 (12+9)     + is an operator, pop
                          12 and 9 add them and
                          push result to Stack

Result: 21
```

Examples:

```
Input : -+8/632
Output : 8

Input : -+7*45+20
Output : 25
```

Evaluation of Prefix Expression (using Stack)

In **prefix notation**, the operators come before their operands.

Code (C++):
https://www.geeksforgeeks.org/evaluation-prefix-expressions/

Code in C:
https://docs.google.com/document/d/11eq6pX267wqccTfWQDuptwkksjBzAvUkJax-DJBXuwc/edit?usp=sharing

Stack Application

1. Code to **reverse a stack using recursion:**

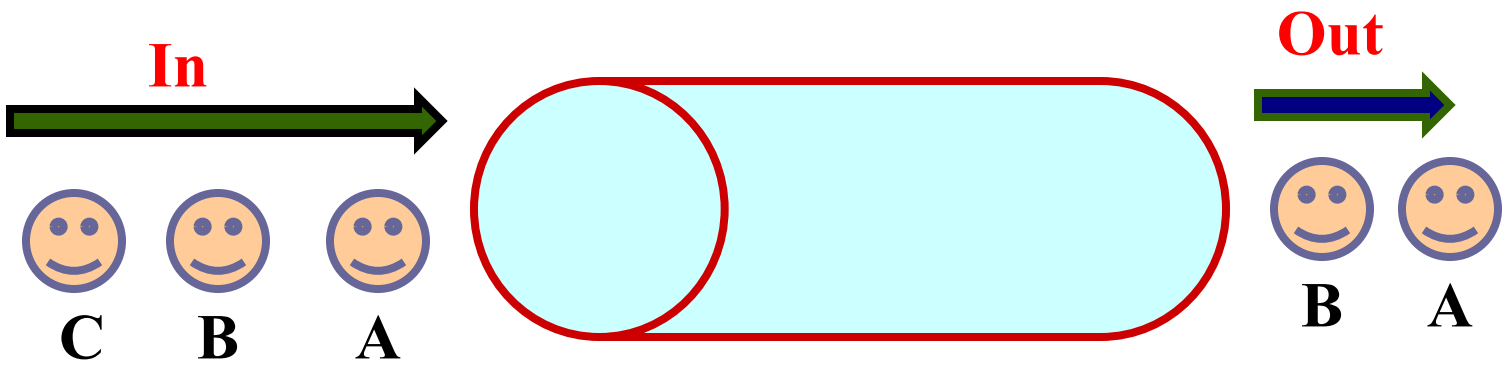
https://www.techcrashcourse.com/2016/06/c-program-to-reverse-stack-using-recursion.html

2. Code to **reverse a string using stack:**

https://www.geeksforgeeks.org/stack-set-3-reverse-string-using-stack/

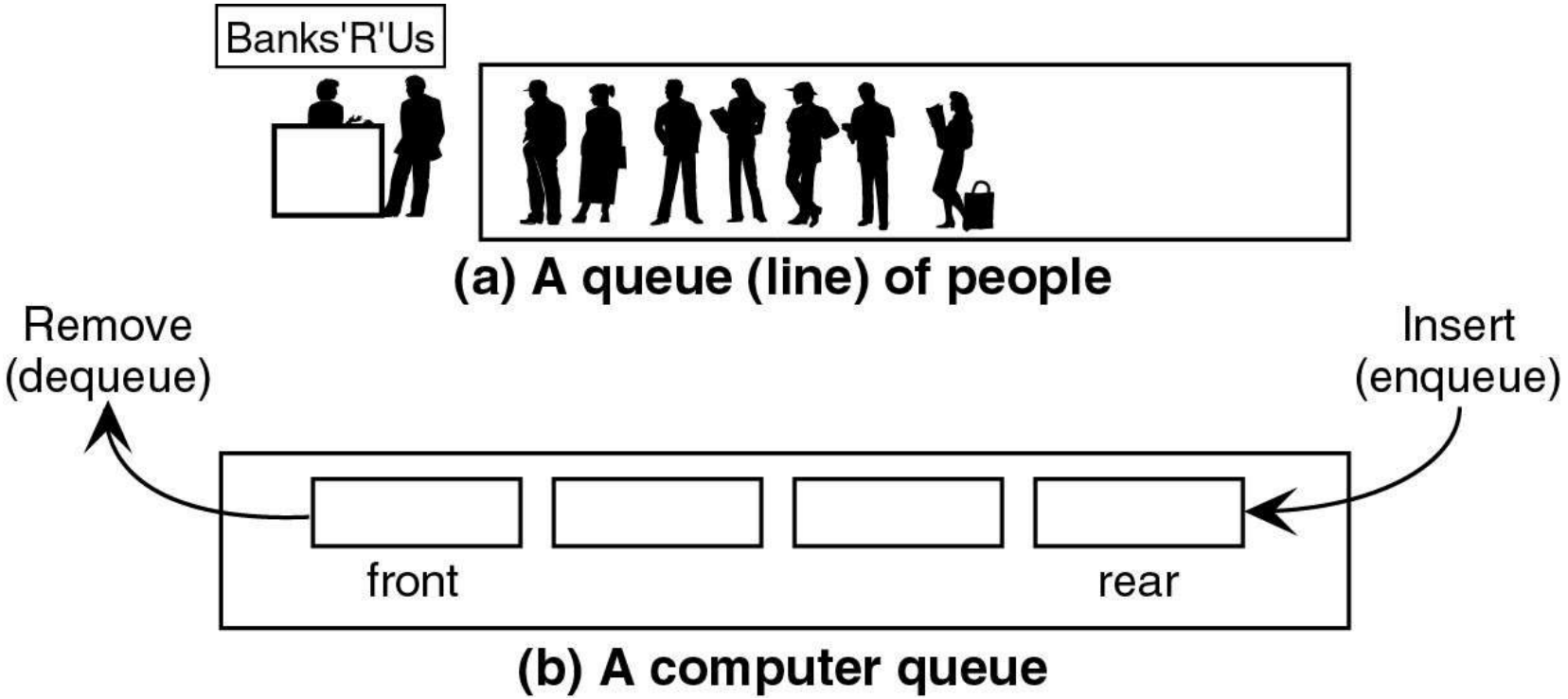
Queues

A First-in First-out (FIFO) List



Also called a QUEUE

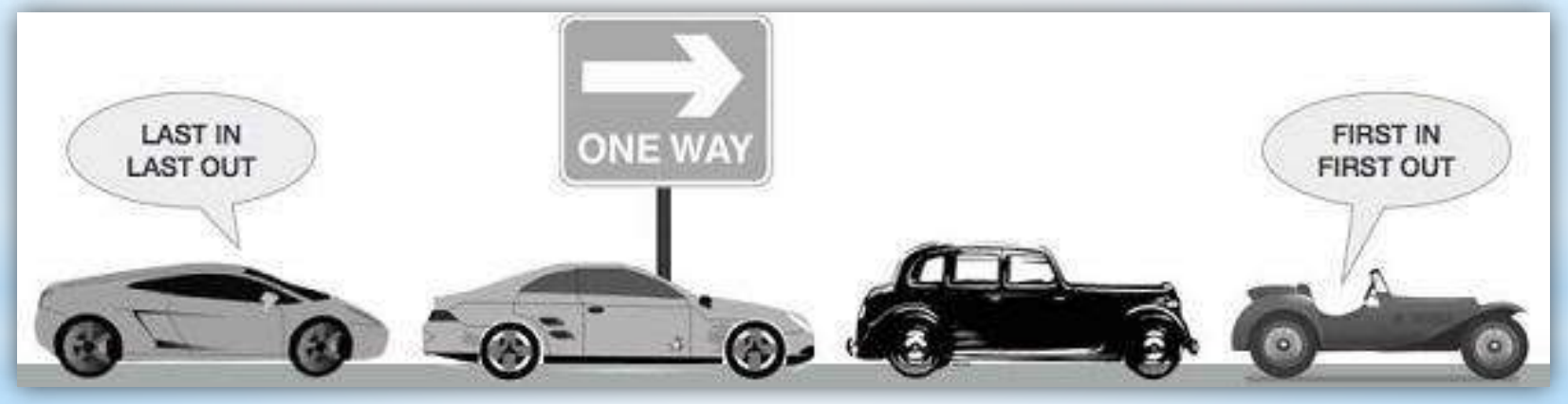
Queues in Our Life



- A queue is a **FIFO** structure: Fast In First Out

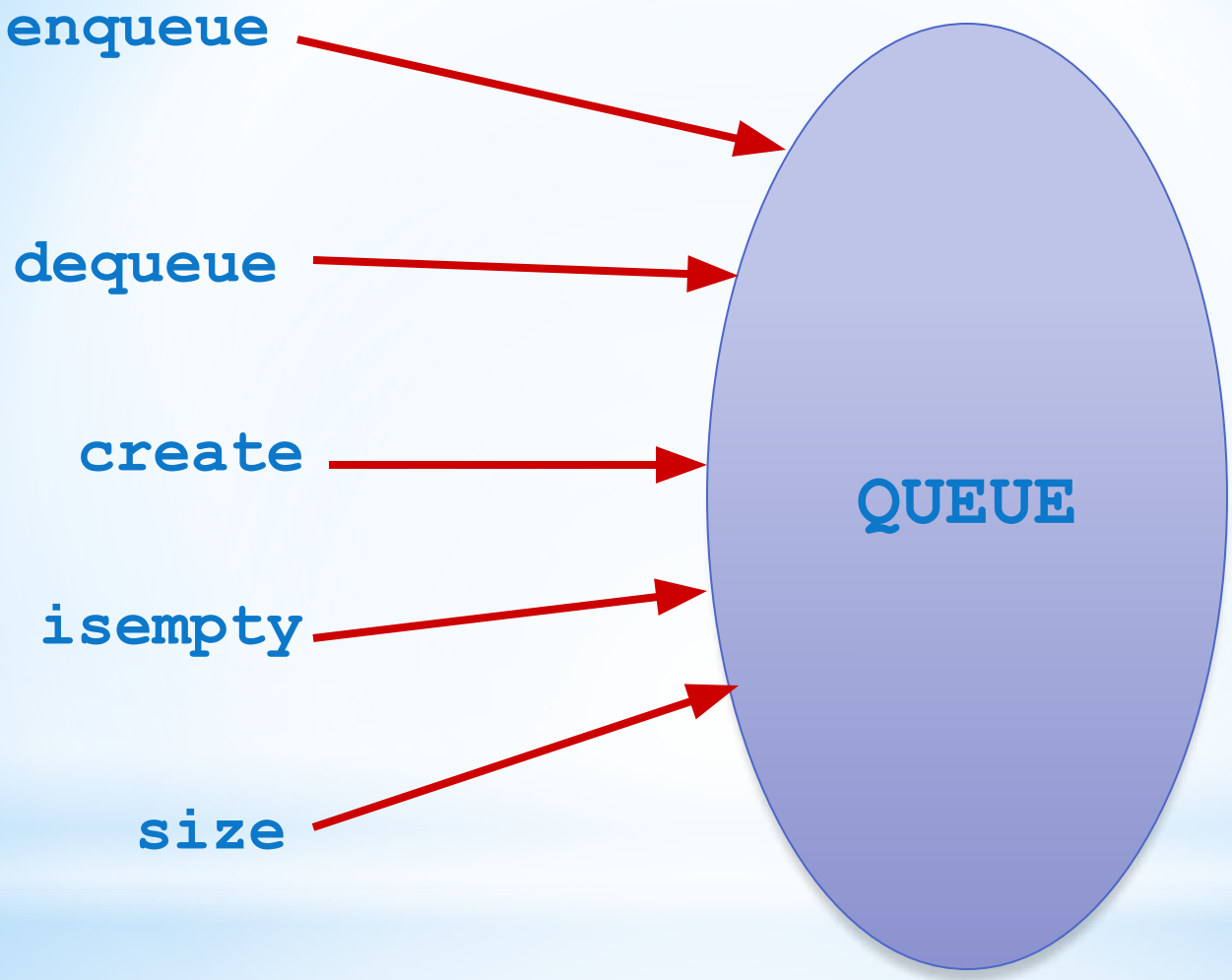
Basic Idea

- Queue is an abstract data structure, somewhat similar to Stacks.
- **Unlike stacks, a queue is open at both its ends.** One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).



Queue Representation

- As in stacks, a queue can also be implemented using Arrays (static implementation) and Linked-lists (dynamic implementation).


enqueue
dequeue
create
isempty
size
QUEUE

QUEUE: First-In-First-Out (LIFO)

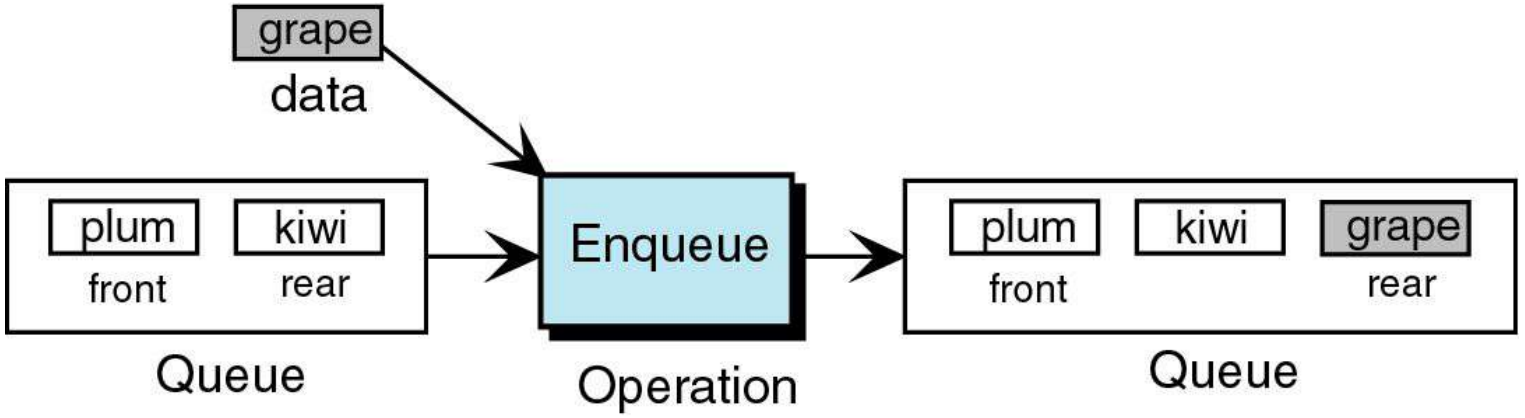
```
void enqueue (queue *q, int element);
                    /* Insert an element in the queue */
int dequeue (queue *q);
                    /* Remove an element from the queue */
queue *create();
                    /* Create a new queue */
int isempty (queue *q);
                    /* Check if queue is empty */
int size (queue *q);
                    /* Return the no. of elements in queue */
```

Assumption: queue contains integer elements!

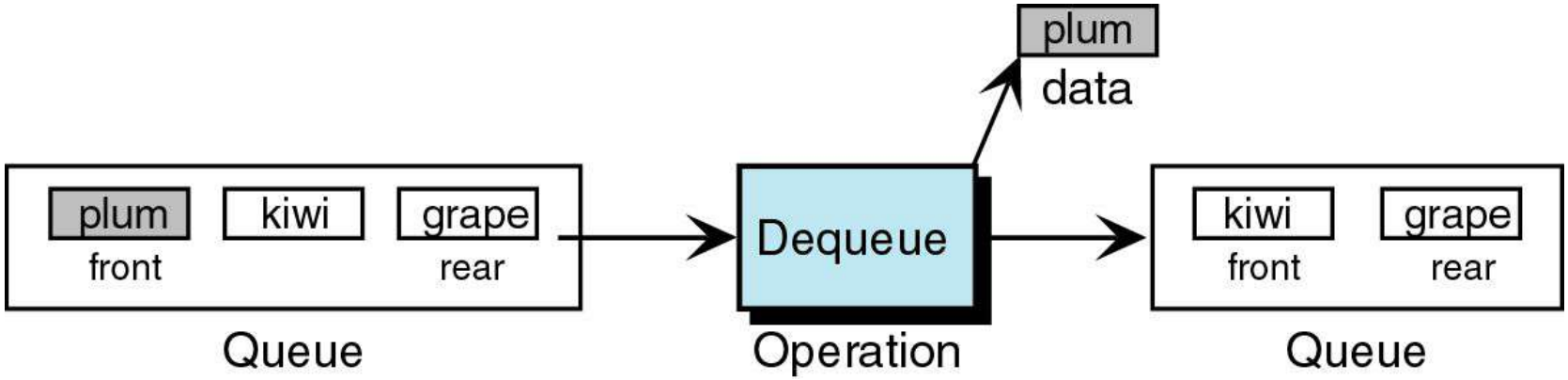
Basic Operations with Queues

- Enqueue - Add an item to the end of queue
  - Overflow
- Dequeue - Remove an item from the front
  - Could be empty
- Queue Front - Who is first?
  - Could be empty
- Queue End - Who is last?
  - Could be empty

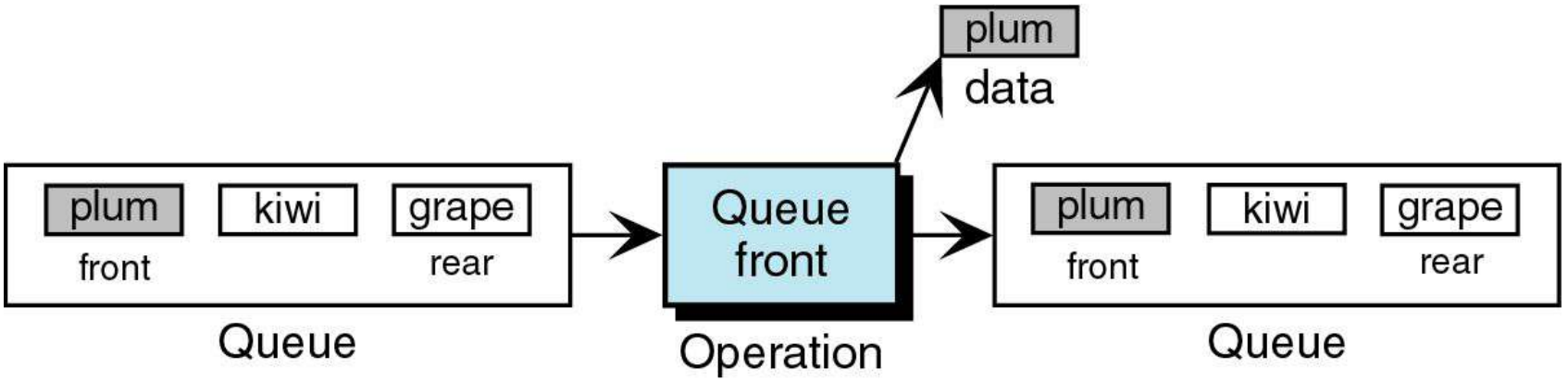
Enqueue


grape
data
plum
kiwi
front
rear
Queue
Enqueue
Operation
plum
kiwi
grape
front
rear
Queue

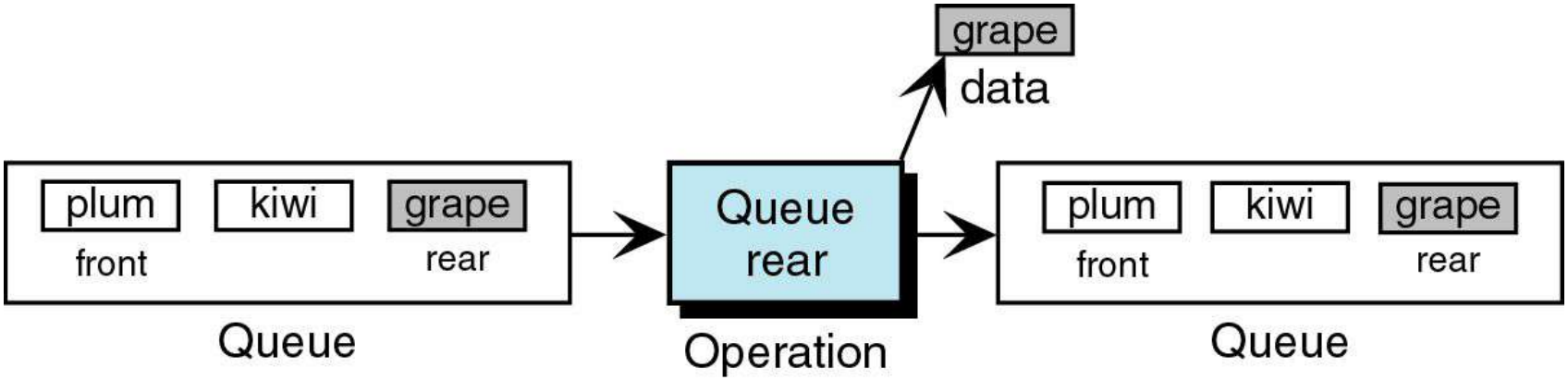
Dequeue

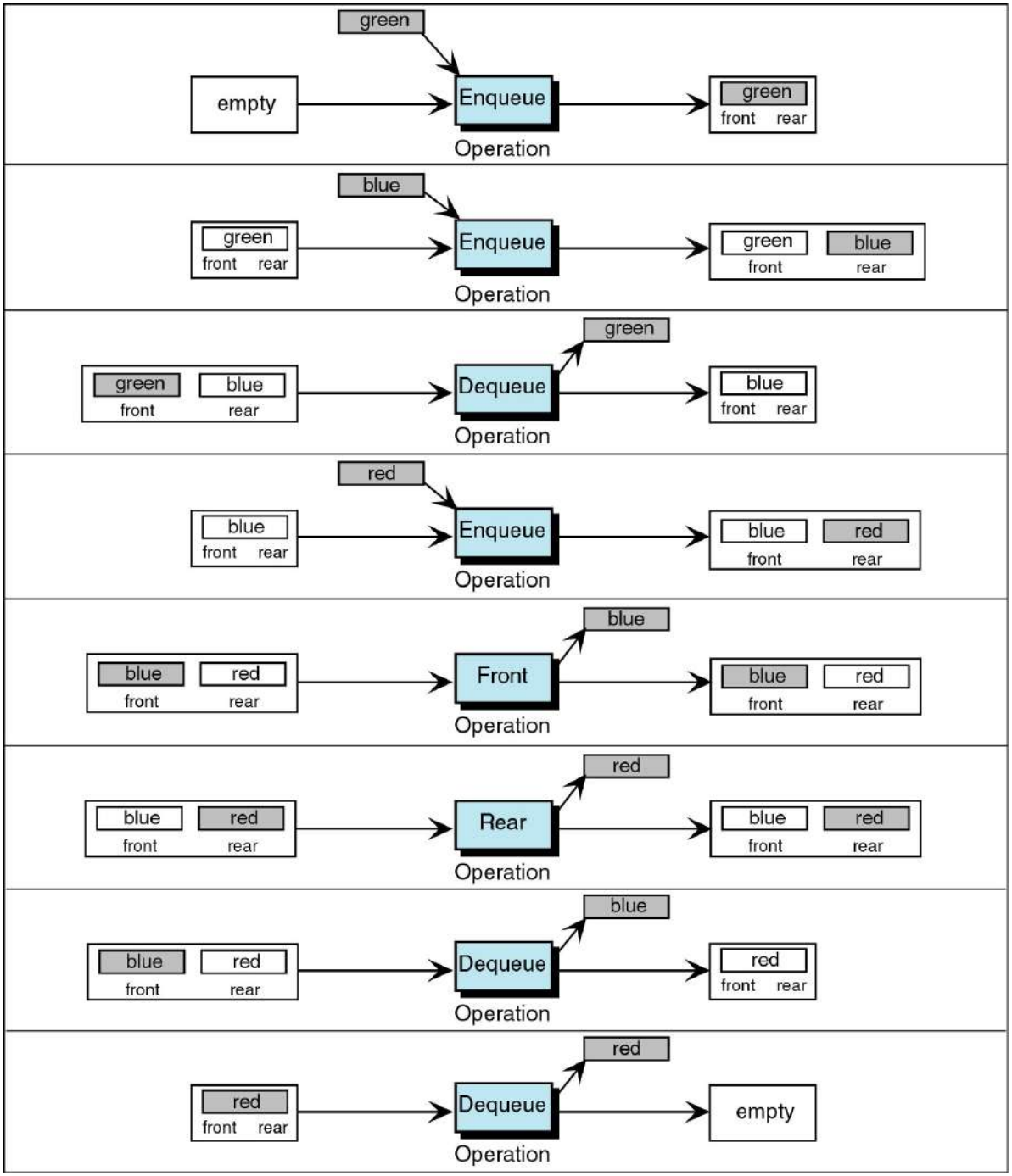

plum
data
plum
kiwi
grape
front
rear
Queue
Dequeue
Operation
kiwi
grape
front
rear
Queue

Queue Front


plum
data
plum
kiwi
grape
front
rear
Queue
Queue front
Operation
plum
kiwi
grape
front
rear
Queue

Queue Rear


grape
data
plum
kiwi
grape
front
rear
Queue
Queue
rear
Operation
plum
kiwi
grape
front
rear
Queue

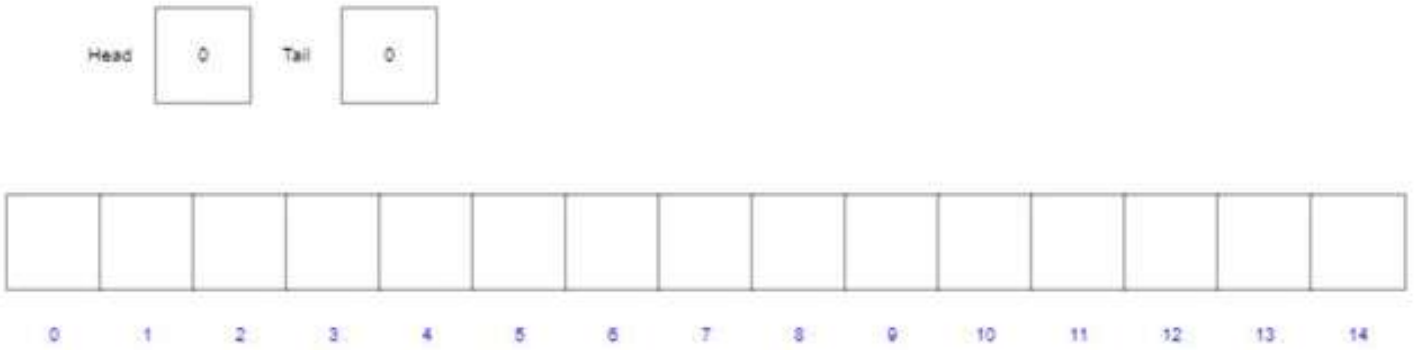

green
empty
Enqueue
Operation
green
front rear
blue
green
front rear
Enqueue
Operation
green
blue
front rear
green
blue
front rear
Dequeue
Operation
blue
front rear
red
blue
front rear
Enqueue
Operation
blue
red
front rear
blue
red
front rear
Front
Operation
blue
red
front rear
blue
red
front rear
Rear
Operation
red
blue
red
front rear
blue
red
front rear
Dequeue
Operation
blue
red
front rear
red
front rear
red
red
front rear
Dequeue
Operation
red
empty

Q. Draw the queue structure in each case when the following operations are performed on an empty queue.

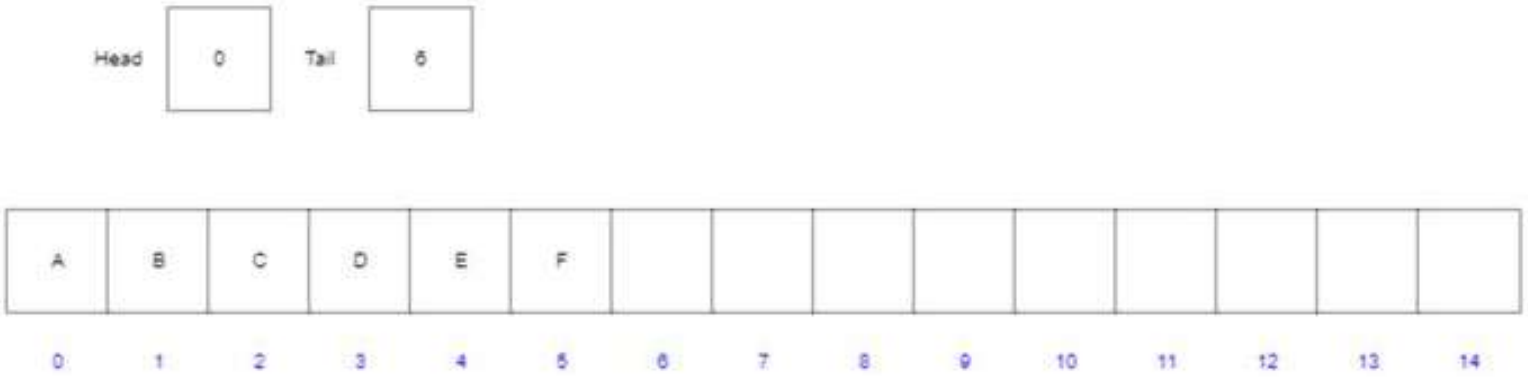
(a) Add A, B, C, D, E, F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I

Sol:

Intial Queue.

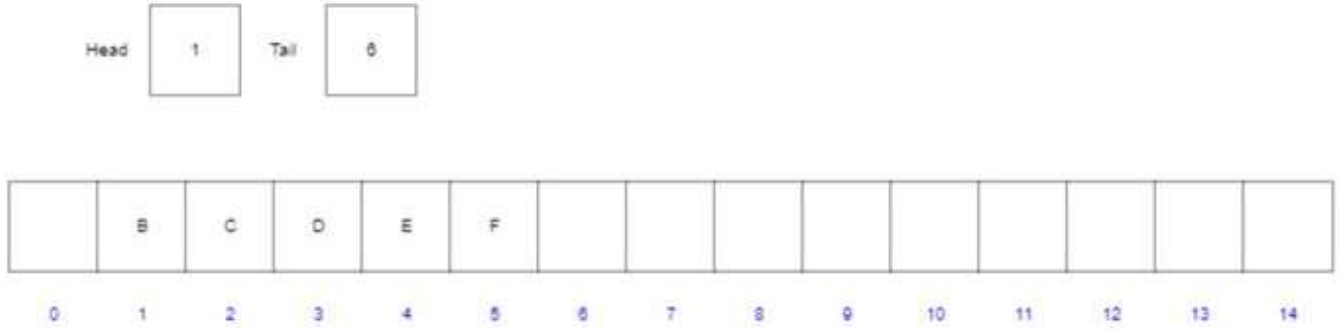


a.) Add A,B,C,D,E,F

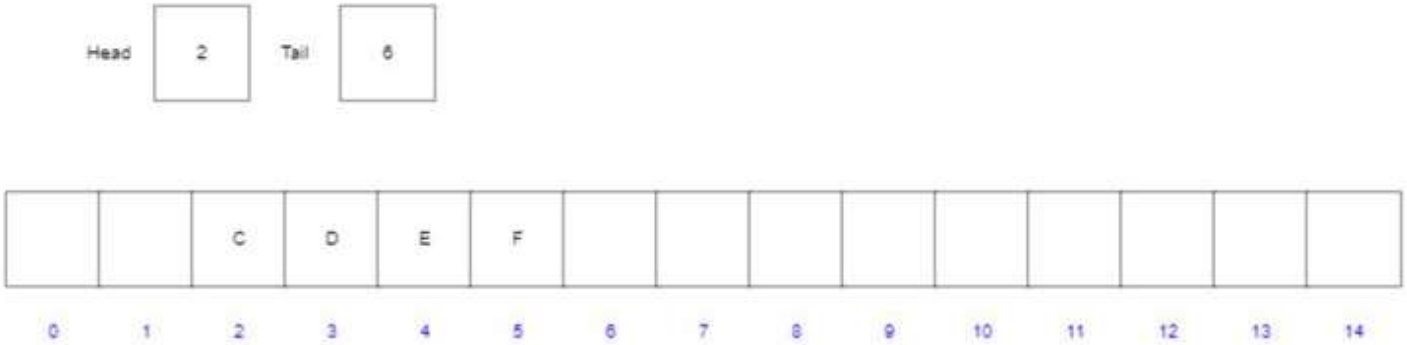


(b) Delete two letters

dequeuing A

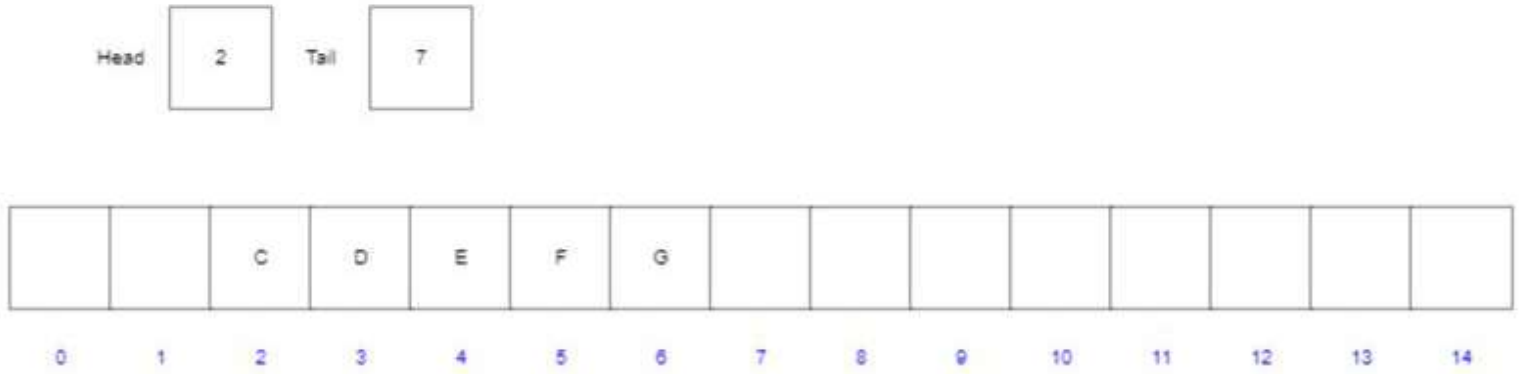


Dequeing B

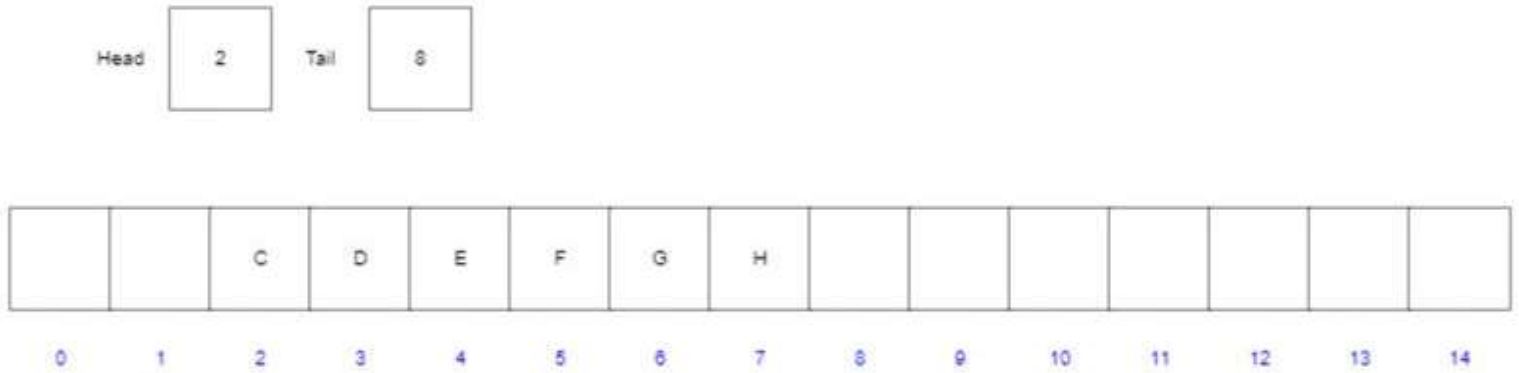


\

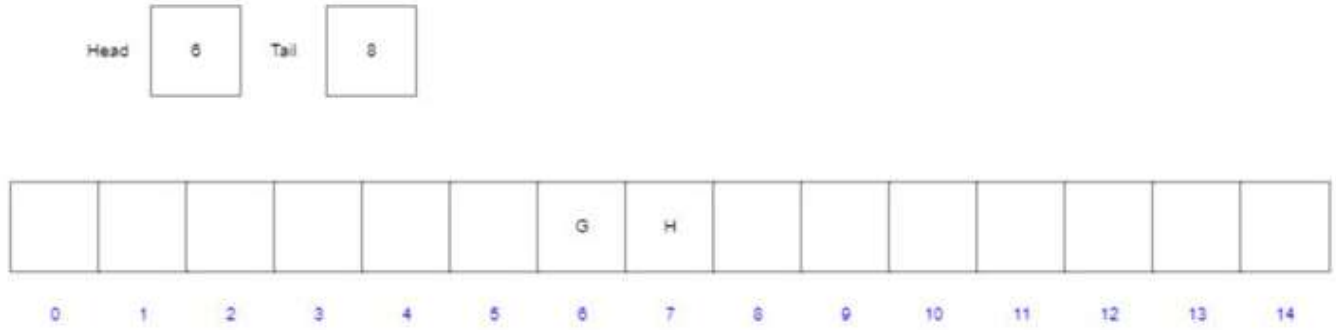
(c) Add G


Head
2
Tail
7
C
D
E
F
G
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

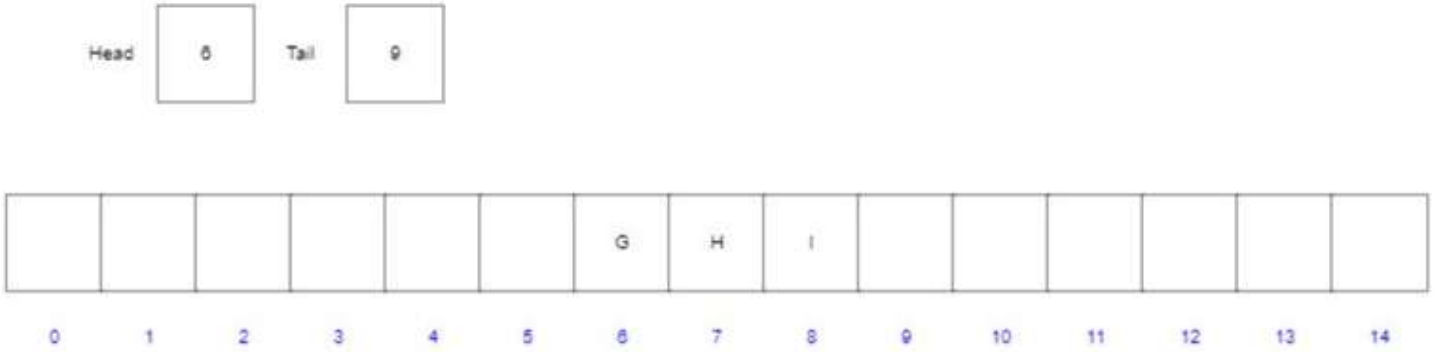
d) Add H


Head
2
Tail
8
C
D
E
F
G
H
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

(e) Delete four letters



(f) Add I



Stacks, Queues, and Linked Lists

- Stacks (Last in/First out List)
  - Operations: Push, Pop, Test Empty, Test Full, Peek, Size
- Queue(First in/First out List)
  - Operations: Insert, Remove, Test Empty, Test Full, Peek, Size
- Linked List(A list of elements connected by pointers)
  - Insert, Delete, Find, Traverse, Size
  - Advantages: can grow, delete/insert with assignments

Stack vs Queue

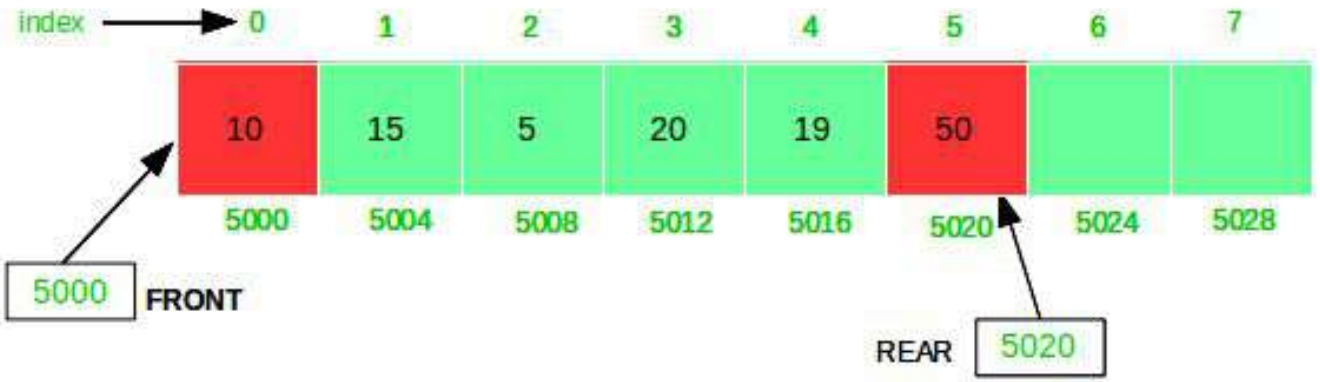
A stack is logically a LIFO type of list.	A queue is logically a FIIO type of list
In a stack insertions and del etions are possible only at one end	In queue insertion is done at one end and deletion is performed at other end
Only one item can be added at a time	only one item can be add at a time
only one item can be deleted at a time	only one item can be deleted at a time
no element other than the top element of stack is visible	no element other than the front and rear element are visible

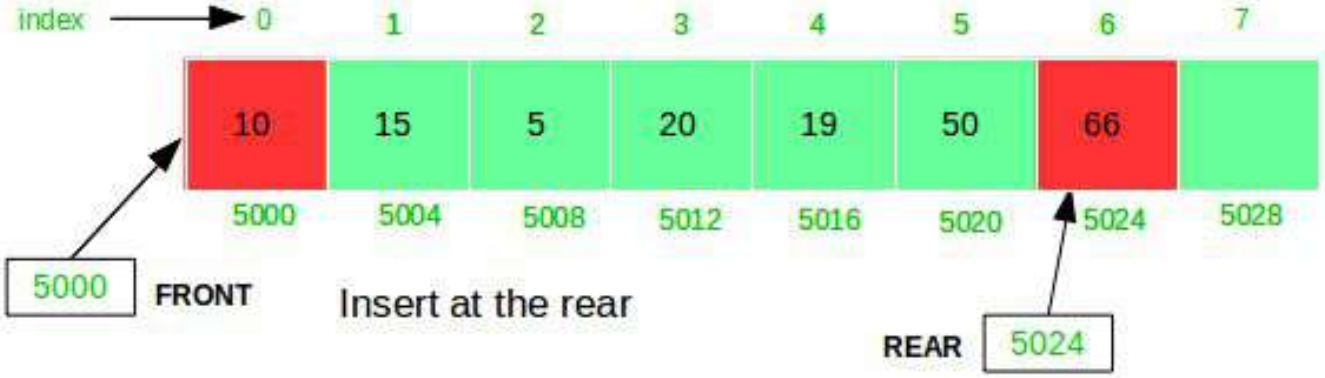
Implementing Queue with Arrays

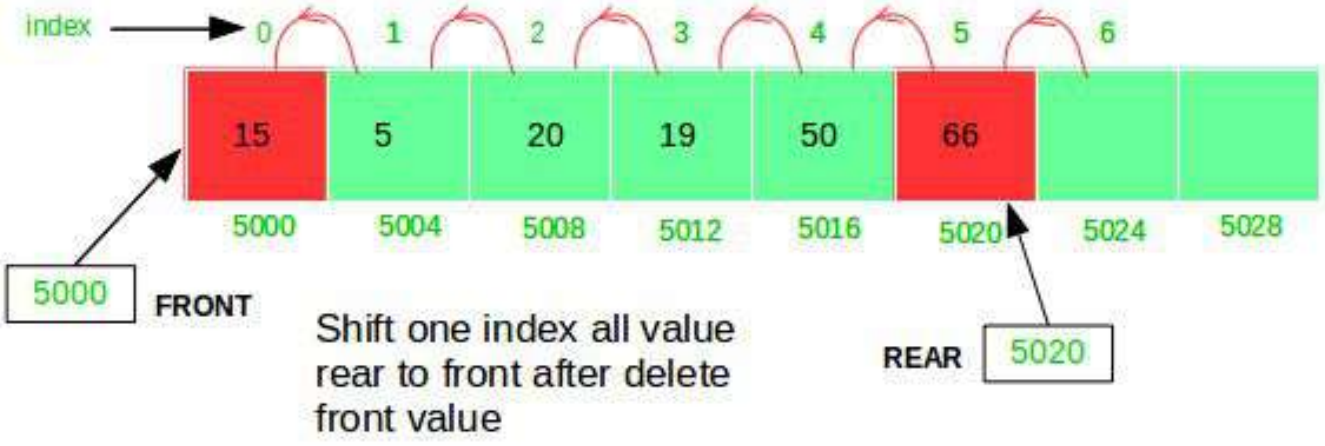
To implement a queue using an array-

- create an array arr of size **n** and
- take two variables **front** and **rear** both of which will be initialized to 0 which means the queue is currently empty.
- Element
  - rear is the index up to which the elements are stored in the array and
  - front is the index of the first element of the array.

Code:
https://towardsdev.com/implementing-a-queue-in-c-using-arrays-a-step-by-step-guide-66146af3f5e1


index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
REAR
5020


index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Insert at the rear
REAR
5024


index
0
1
2
3
4
5
6
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Shift one index all value rear to front after delete front value
REAR
5020

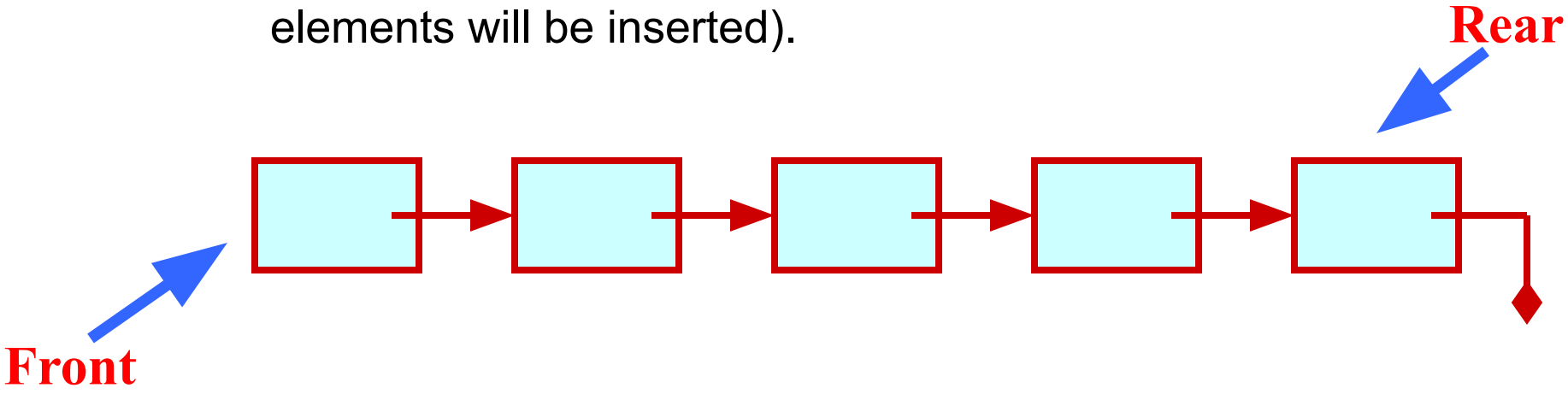
Types of Queues

Types of queue (theory):
https://www.javatpoint.com/ds-types-of-queues

LL implementation of Simple Queue:
https://www.javatpoint.com/linked-list-implementation-of-queue

Queue Implementation Using Linked List

- Basic idea:
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where new elements will be inserted).



Example :Queue using Linked List

```
struct qnode
{
  int val;
   struct qnode *next;
};

struct queue
{
   struct qnode *qfront, *qrear;
};
typedef struct queue QUEUE;
```

```
void enqueue (QUEUE *q,int element)
{
   struct qnode *q1;
   q1=(struct qnode *)malloc(sizeof(struct qnode));
   q1->val= element;
   q1->next=q->qfront;
   q->qfront=q1;
}
```

Example :Queue using Linked List

```
int size (queue *q)
{
    queue *q1;
    int count=0;
    q1=q;
    while(q1!=NULL)
    {
      q1=q1->next;
      count++;
    }
return count;
}
```

```
int peek (queue *q)
{
   queue *q1;
   q1=q;
   while(q1->next!=NULL)
      q1=q1->next;
return (q1->val);
}
```

```
int dequeue (queue *q)
{
    int val;
    queue *q1,*prev;
    q1=q;
    while(q1->next!=NULL)
    {
       prev=q1;
       q1=q1->next;
    }
    val=q1->val;
    prev->next=NULL;
    free(q1);
return (val);
}
```

Assume:: queue contains integer elements

```
void  enqueue  (queue q,  int element);
                              /* Insert an element in the queue */
int  dequeue (queue q);
                          /* Remove an element from the queue */
queue  *create  ();
                                  /* Create a new queue */
int  isempty  (queue q);
                                  /* Check if queue is empty */
int  size  (queue q);
                    /* Return the number of elements in queue */
```

Creating a queue

front = NULL;

rear = NULL;

Inserting an element in queue

```
void  enqueue  (queue q,  int x)
{
    queue  *ptr;
    ptr  =  (queue *)  malloc (sizeof (queue));

    if  (rear == NULL)                    /* Queue is empty */
        {
            front  =  ptr;    rear  =  ptr;
            ptr->element  =  x;
            ptr->next  =  NULL;
        }
    else                                  /* Queue is not empty
  */
        {
            rear->next  =  ptr;
            ptr ->element  =  x;
            ptr->next  =  NULL;
        }
}
```

Deleting an element from queue

```
int  dequeue  (queue q)
 {
    queue  *old;

    if  (front == NULL)                    /* Queue is empty */
        printf  ("\n Queue is empty");

    else   if  (front == rear)              /* Single element */
                {
                   k = front->element;
                   free (front);    front = rear = NULL;
                   return (k);
                }
           else
                {
                   k  =  front->element;    old  =  front;
                   front  =  front->next;
                   free (old);
                   return (k);
                }
 }
```

Checking if empty

```
int  isempty  (queue q)
{
    if  (front == NULL)
        return (1);
    else
        return (0);
}
```

Types of Queues

1. Deque

2. Circular Queue

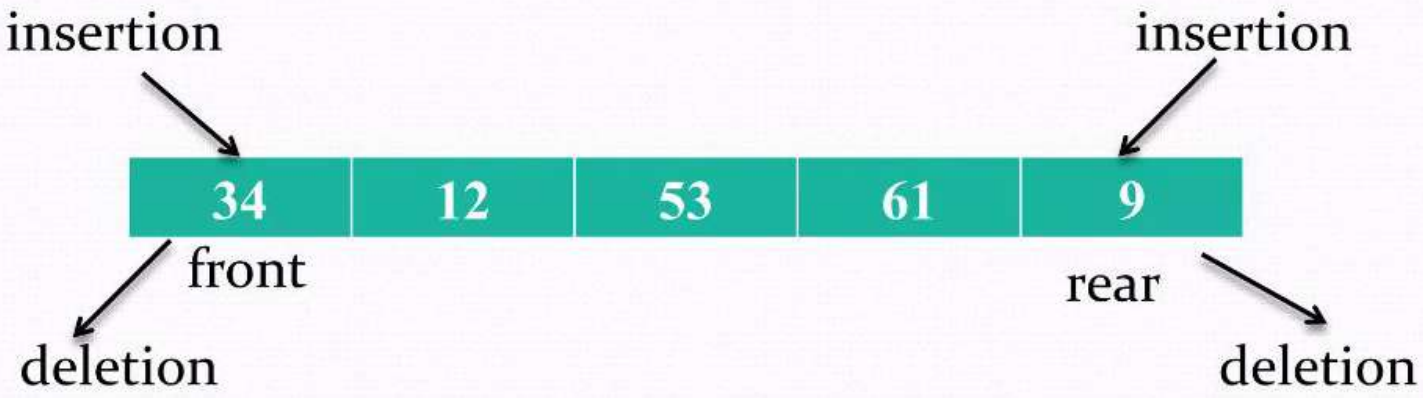
3. Priority Queue

DEQUES

1.Deque stands for *double ended queue*.

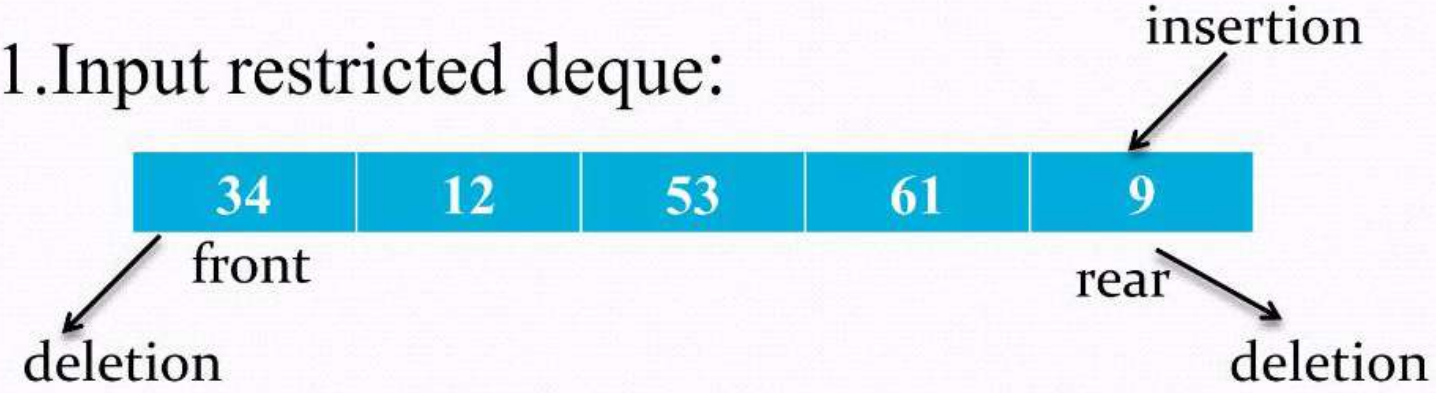
2.Elements can be inserted or deleted at either end.

3. Also known as *head-tail linked list*.

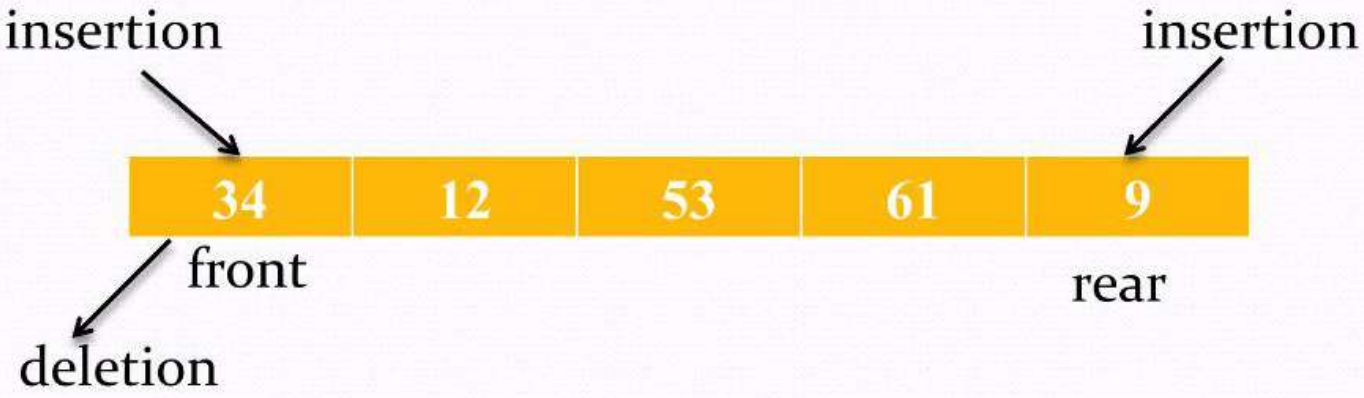


Types Of Deque

1.Input restricted deque:



2. Output restricted deque:



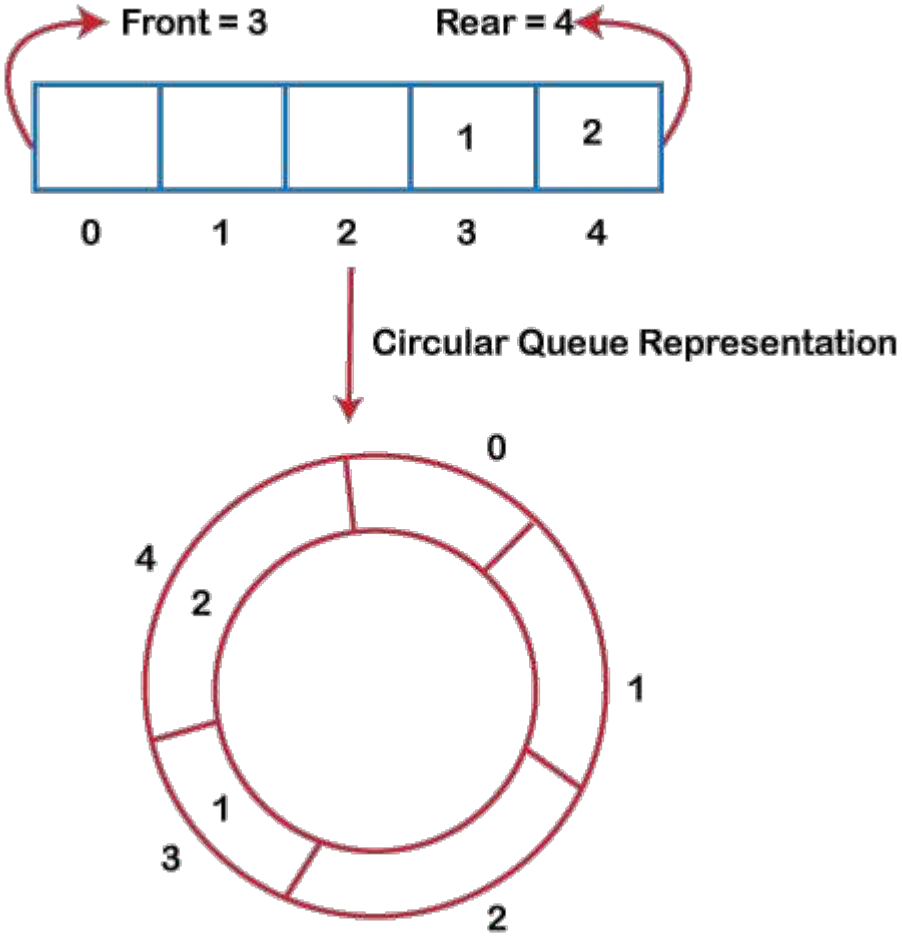
CIRCULAR QUEUES

- Circular queue are used to remove the drawback of simple queue.

- Both the front and the rear pointers wrap around to the beginning of the array.

- It is also called as "***Ring buffer***".

There was one **limitation** in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that **some vacant spaces are left in the beginning which cannot be utilized.** So, to overcome such limitations, the concept of the circular queue was introduced.



Why is circular queue better than a linear queue?

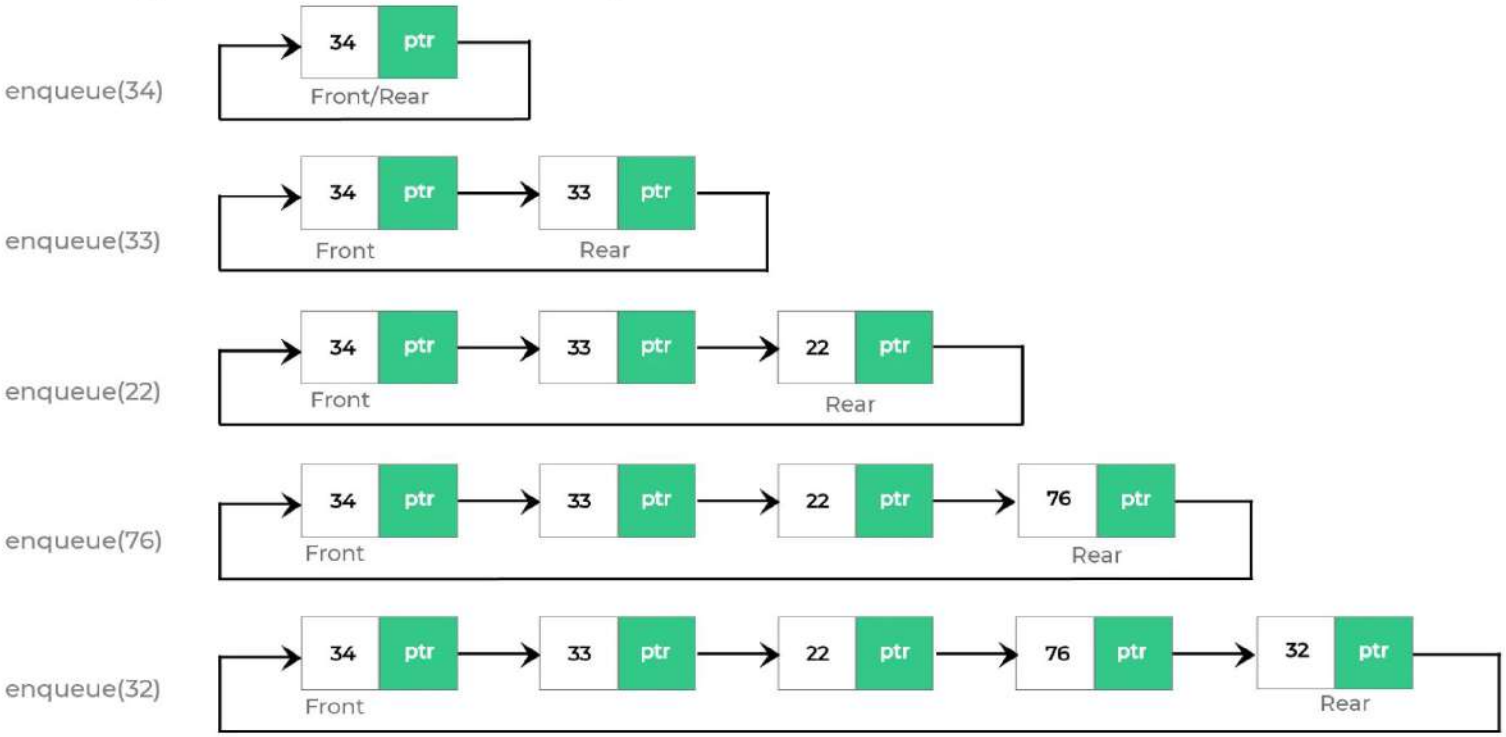
In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such **wastage of memory space** by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because **shifting all the elements will consume lots of time.** The efficient approach to avoid the wastage of the memory is to **use the circular queue data structure.**

CIRCULAR QUEUE

A circular queue is **similar to a linear queue** as it is also **based on the FIFO (First In First Out)** principle except that **the last position is connected to the first position in a circular queue that forms a circle.** It is also known as a ***Ring Buffer***.

Implementation of Circular Queues using Linked List in C

Adding the elements into Queue



Removing the elements from Queue

Implementing Circular Queue using Linked List in C

enqueue(data)

- Create a struct node type node.
- Insert the given data in the new node data section and NULL in address section.
- If Queue is empty then initialize front and rear from new node.
- Queue is not empty then initialize rear next and rear from new node
- New node next initialize from front

dequeue()

- Check if queue is empty or not.
- If queue is empty then dequeue is not possible.
- Else Initialize temp from front.
- If front is equal to the rear then initialize front and rear from null.
- Print data of temp and free temp memory.
- If there is more than one node in Queue then make front next to front then initialize rear next from front.
- Print temp and free temp.

print()

- Check if there is some data in the queue or not.
- If the queue is empty print “No data in the queue.”
- Else define a node pointer and initialize it with front.
- Print data of node pointer until the next of node pointer becomes NULL.

Linked representation of

- Circular queue
- Priority queue
- Deque

Linked Representation of Circular Queue

C CODE:
https://docs.google.com/document/d/10ePZus_Rgye9CY0HemJOLOOf8zWEpkaOkAQIE1hgITw/edit?usp=sharing

https://www.javatpoint.com/circular-queue

https://prepinsta.com/c-program/circular-queue-using-linked-list/

https://www.geeksforgeeks.org/introduction-to-circular-queue/ (theory)

PRIORITY QUEUE

1. It is collection of elements where elements are stored according to the their priority levels.
2. Inserting and removing of elements from queue is decided by the priority of the elements.
3. An element of the higher priority is processed first.
4. Two element of same priority are processed on first-come-first-served basis.

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.

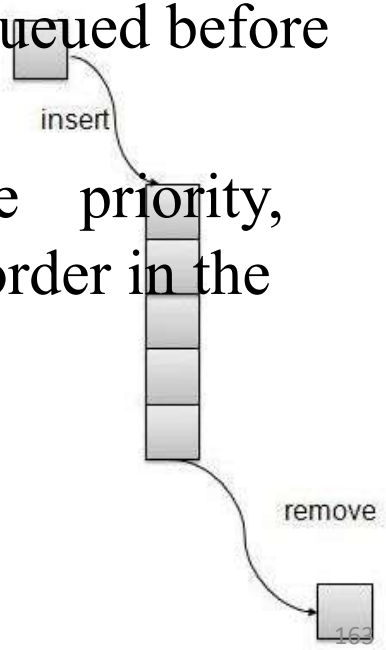
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.

Priority Queue

- Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.

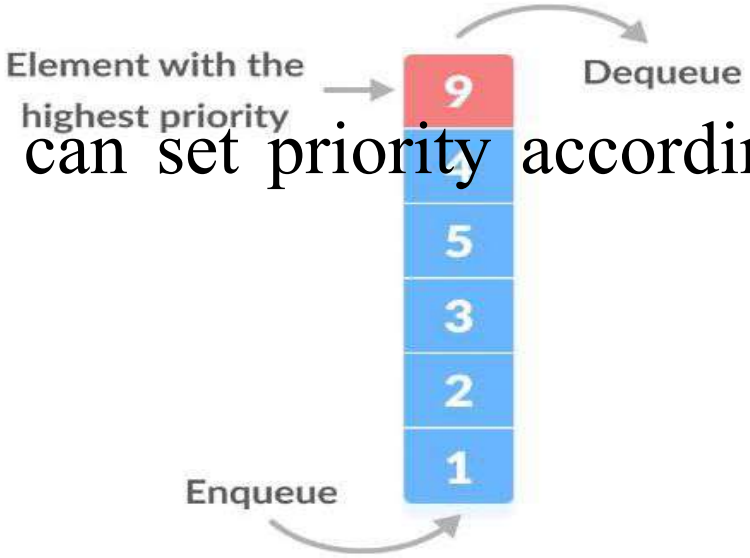
Priority Queue

- Priority Queue is an extensionof queue with following properties.
  - Every item has a priority associated with it.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.



Priority Queue

- The element with the highest value is considered as the highest priority element.
- However, in other case, we can assume the element with the lowest value as the highest priority element.
- In other cases, we can set priority according to our need.



Priority Queue

- Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements.
- The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Example: Suppose you have a few assignment from different subjects. Which assignment will you want to do first?

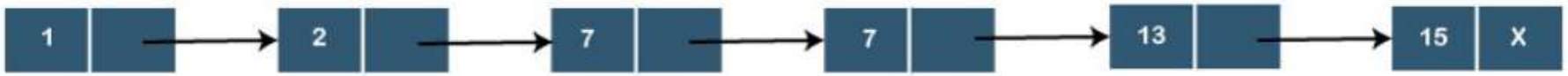
subjects	Due date	priority
DSGT	15 OCT	4
DLD	6 OCT	2
CYB	4 OCT	1
DS	8 OCT	3

Let's understand through an example.

Consider the below-linked list that consists of elements 2, 7, 13, 15.

Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes so we will insert the node at the beginning of the list shown as below:

Now we have to add 7 element to the linked list. We will traverse the list to insert element 7. First, we will compare element 7 with 1; since 7 has lower priority than 1, so it will not be inserted before 7. Element 7 will be compared with the next node, i.e., 2; since element 7 has a lower priority than 2, it will not be inserted before 2.. Now, the element 7 is compared with a next element, i.e., since both the elements have the same priority so they will be served based on the first come first serve. The new element 7 will be added after the element 7 shown as below:



Linked Representation of Priority Queue

Implement Priority Queue using Linked Lists.

- **push():** This function is used to insert a new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

The list is so created so that the **highest priority element is always at the head of the list.** The list is arranged in **descending order of elements based on their priority.** This allow us to remove the highest priority element quickly. **To insert an element we must traverse the list and find the proper position to insert the node** so that the overall order of the priority queue is maintained.

Algorithm :

PUSH(HEAD, DATA, PRIORITY):

- ***Step 1:*** *Create new node with DATA and PRIORITY*
- ***Step 2:*** *Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.*
- ***Step 3:*** *NEW -> NEXT = HEAD*
- ***Step 4:*** *HEAD = NEW*
- ***Step 5:*** *Set TEMP to head of the list*
- ***Step 6:*** *While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY*
- ***Step 7:*** *TEMP = TEMP -> NEXT*

  [END OF LOOP]
- ***Step 8:*** *NEW -> NEXT = TEMP -> NEXT*
- ***Step 9:*** *TEMP -> NEXT = NEW*
- ***Step 10****: End*

POP(HEAD):

- ***Step 1:*** *Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.*
- ***Step 2:*** *Free the node at the head of the list*
- ***Step 3:*** *End*

PEEK(HEAD):

- ***Step 1:*** *Return HEAD -> DATA*
- ***Step 2:*** *End*

Linked Representation of Priority Queue

C CODE:
https://docs.google.com/document/d/10ePZus_Rgye9CY0HemJOLOOf8zWEpkaOkAQIE1hgITw/edit?usp=sharing

Linked Representation of Deque

Code in C++:
https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/

Application of Queues

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3. Linear Queue: A linear queue is a type of queue where data elements are added to the end of the queue and removed from the front of the queue. Linear queues are used in applications where data elements need to be processed in the order in which they are received. Examples include printer queues and message queues.
4. Circular Queue: A circular queue is similar to a linear queue, but the end of the queue is connected to the front of the queue. This allows for efficient use of space in memory and can improve performance. Circular queues are used in applications where the data elements need to be processed in a circular fashion. Examples include CPU scheduling and memory management.
5. Priority Queue: A priority queue is a type of queue where each element is assigned a priority level. Elements with higher priority levels are processed before elements with lower priority levels. Priority queues are used in applications where certain tasks or data elements need to be processed with higher priority. Examples include operating system task scheduling and network packet scheduling.
6. Double-ended Queue: A double-ended queue, also known as a deque, is a type of queue where elements can be added or removed from either end of the queue. This allows for more flexibility in data processing and can be used in applications where elements need to be processed in multiple directions. Examples include job scheduling and searching algorithms.

Some common applications of Queue data structure :

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
9. **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

Useful Applications of Queue

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, etc.

Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management

Applications of Queue in Networks:

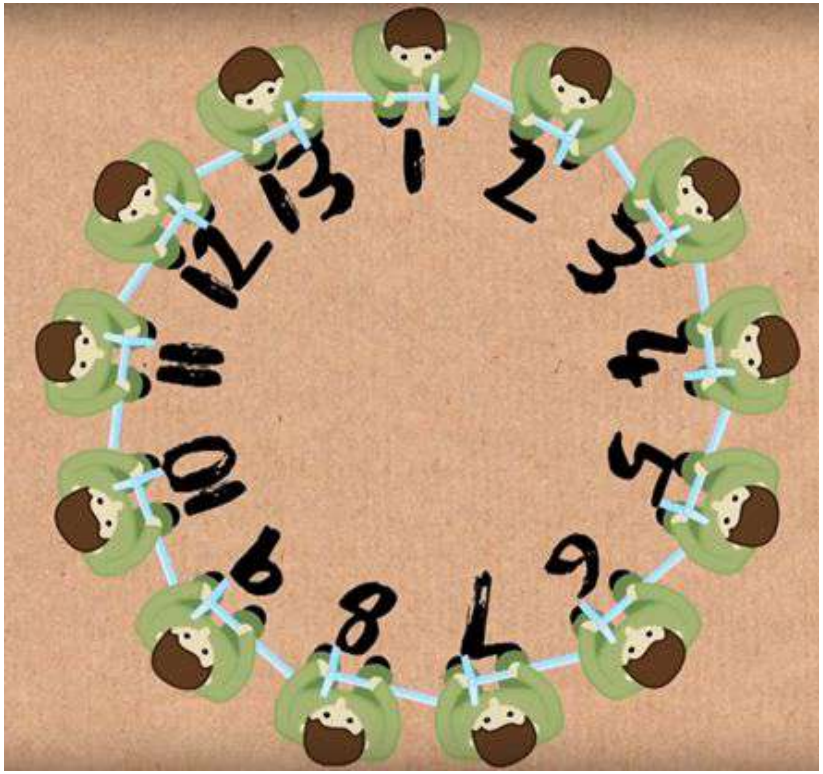
- Queues in routers/ switches
- Mail Queues
- **Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)

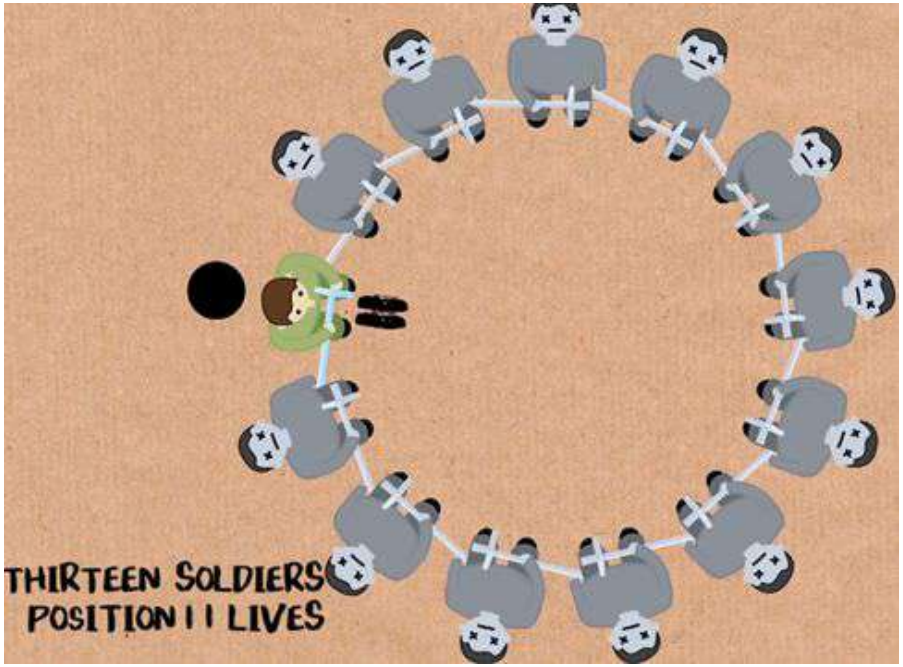
Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)

Application of Queue: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to





- History: N men surrounded by enemies. Preferred dying rather than captured as slaves. Every men killed the next living men until 1 man is left. That guy (Josephus) then surrendered (did not tell this initially 'cos others'd turn on him).

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

Pattern # 1

Winner always odd!

Makes sense as the first loop kills all the evens

Application: Josephus Problem

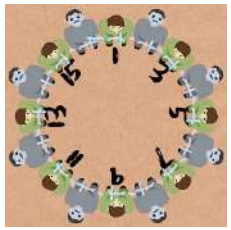
- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

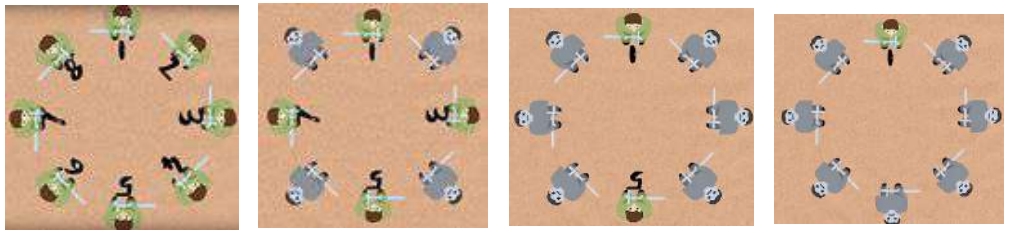
N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2

Jump by 2; reset at 2^a for some a!

Makes sense: Assume 2^a men in circle. 1 pass removes half of them; at 1; repeat on 2^{a-1} men; so winner is the starting point (1)





*

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

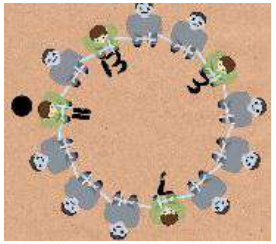
N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2 (cont'd)

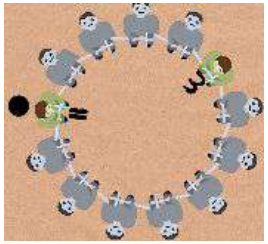
Jump by 2; reset at 2^a for some a!

Makes sense: Assume $2^a + b$ men in circle, where a is the biggest possible power; hence $b < 2^a$ (binary notation idea); after b men we are left with 2^a men, whose winner is the starting point (11 for below)

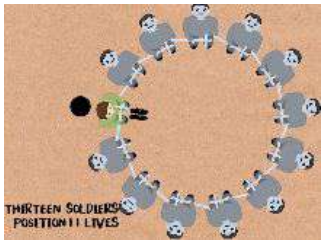
8 left



4 left



2 left



1 left

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2 (cont'd)

Jump by 2; reset at 2^a for some a!

So what is 11 for N=13?

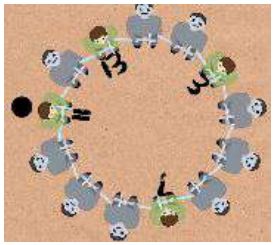
$N = 2^3 + 5$ (a=3, b=5); and $11 = 2*5 + 1$

In general, after b steps, we arrive at the position $2*b + 1$ (every 2^{nd} is killed). Hence,

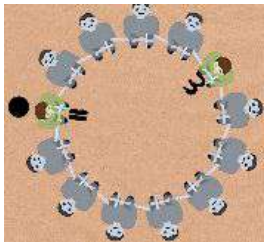
$$W(N) = 2*b + 1$$

where $N = 2^a + b$ and $b < 2^a$

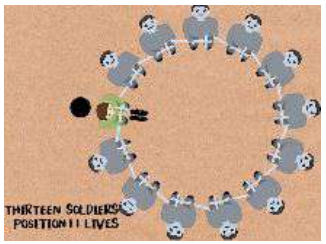
8 left



4 left



2 left



1 left

Application: Josephus Problem

- Based on W(N) = 2*b + 1 where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index

```
int W(int N) {
  int a = 0;
  while (N > 1) {
     N /= 2;
     a++;
  }
  return 2*(N – pow(2, a)) + 1;
  //or you could compute pow(2, a) in variable V like this:
  //int V = 1; for (int i = 0; i < a; i++) V *= 2;
}
```

Application: Josephus Problem

- Based on W(N) = 2*b + 1 where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index
- If you do not like math and cannot extract W(N) above, you can write the code using a Queue
- Math gets way complicated for the generic problem where you kill every k^{th} man where k > 1

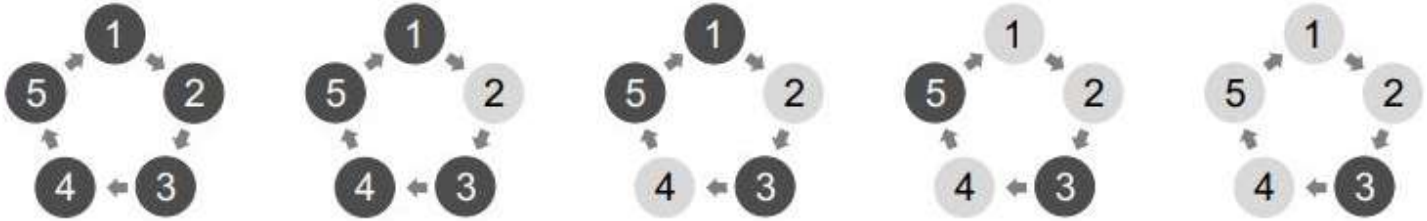
```
int Josephus(Q, k) { //Queue Q is built in advance with e.g., 1, 2, 3, 4, 5, 6.
  while (Q.size() > 1) {
    for (i = 1; i <= k-1; i++) //skip the k-1 men without killing
      Q.enqueue( Q.dequeue() );
    killed = Q.dequeue();
  }
* return Q.dequeue(); } //only one left in the Q, the winner ☺
```

Josephus Problem

Let us see how queues can be used for finding a solution to the Josephus problem.
In Josephus problem, `n` people stand in a circle waiting to be executed. The counting starts at some point in the circle and proceeds in a specific direction around the circle. In each step, a certain number of people are skipped and the next person is executed (or eliminated). The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the 'winner'.

Therefore, if there are `n` number of people and a number `k` which indicates that `k-1` people are skipped and `k-th` person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner.

For example, if there are 5 (`n`) people and every second (`k`) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.



Try the same process with n = 7 and k =3. You will find that person at position 4 is the winner. The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.

7. Write a program which finds the solution of Josephus problem using a circular linked list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int player_id;
        struct node *next;
};
struct node *start, *ptr, *new_node;

int main()
{
        int n, k, i, count;
        clrscr();
        printf("\n Enter the number of players : ");
        scanf("%d", &n);
        printf("\n Enter the value of k (every kth player gets eliminated): ");
        scanf("%d", &k);
        // Create circular linked list containing all the players
        start = malloc(sizeof(struct node));
        start->player_id = 1;
        ptr = start;
        for (i = 2; i <= n; i++)
        {
                new_node = malloc(sizeof(struct node));
                ptr->next = new_node;
                new_node->player_id = i;
                new_node->next=start;
                ptr=new_node;
        }
        for (count = n; count > 1; count--)
        {
                for (i = 0; i < k - 1; ++i)
                        ptr = ptr->next;
                ptr->next = ptr->next->next; // Remove the eliminated player from the
circular linked list
        }
        printf("\n The Winner is Player %d", ptr->player_id);
        getche();
        return 0;
}
```

\\getche() function reads a single character from the keyboard and displays immediately on the output screen without waiting for enter key.

Output

```
Enter the number of players : 5
Enter the value of k (every kth player gets eliminated): 2
The Winner is Player 3
```

Any questions?