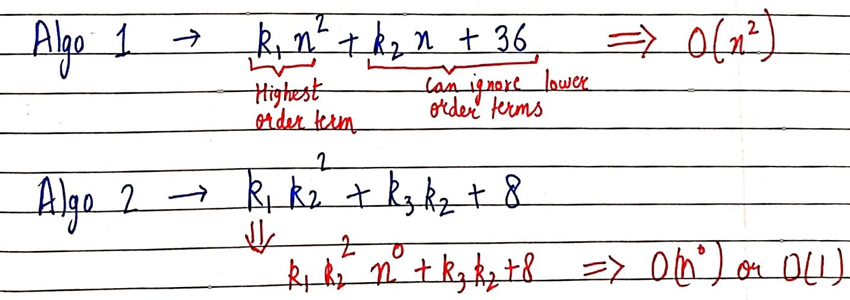
DSA

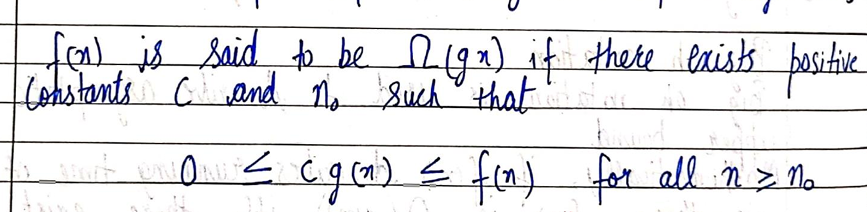


* In order to find big O of an function
* We need to first consider highest order term (in this case (n^2) and can ignore others (in this case (k2 n+36)
* Then removing the constant beside the highest order term and adding BIG O with it.

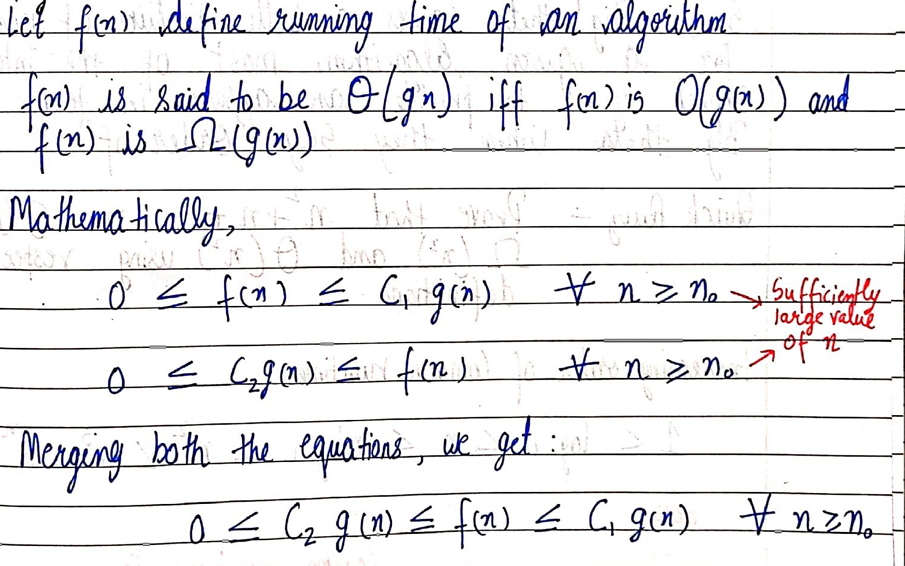
BIG O



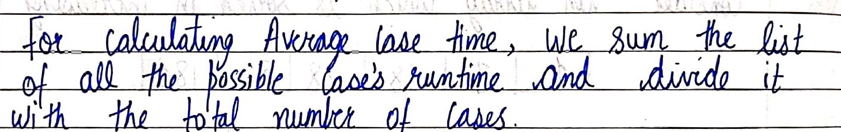
BIG OMEGA

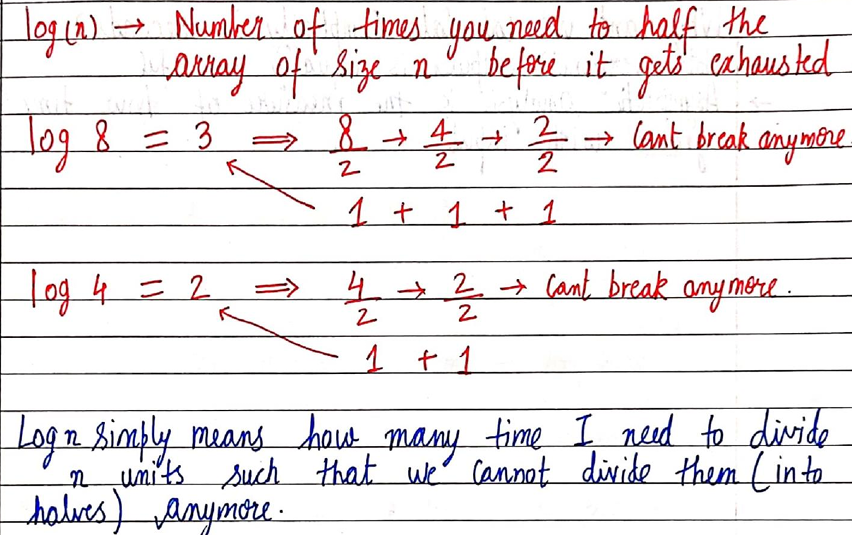


BIG THETA

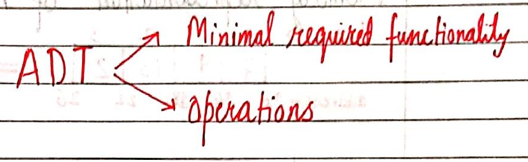


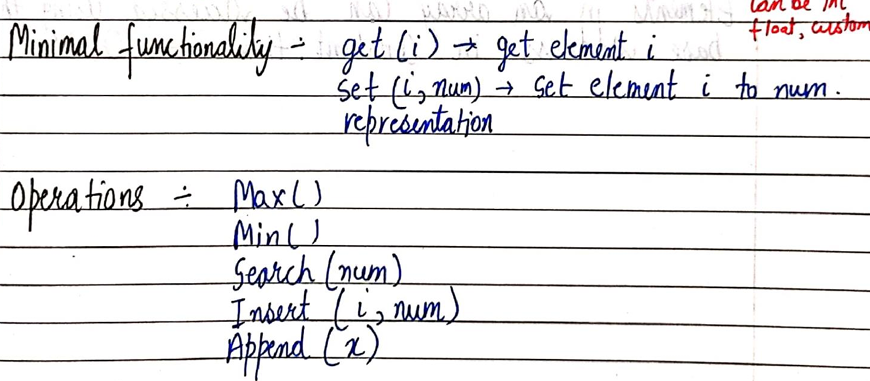
* BEST , WORST and AVERAGE CASE TIME

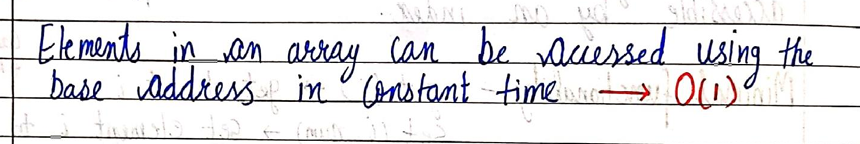




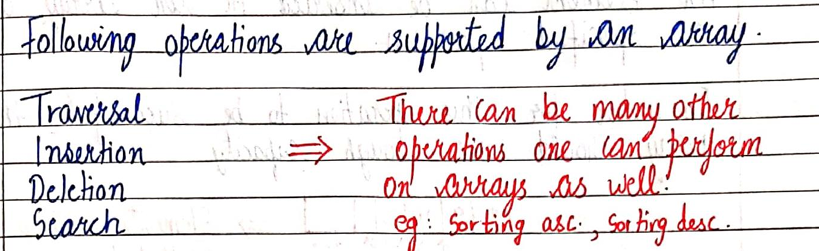
* ADT

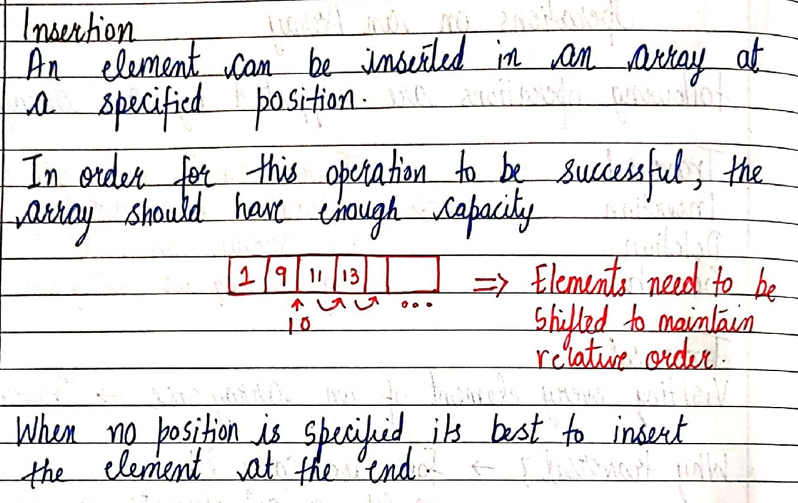






* Operations in Array





#include <stdio.h>

void display(int arr[], int n)

{

    for (int i = 0; i < n; i++)

    {

        printf("%d\t", arr[i]);

    }

    printf("\n");

}

int Insertion(int arr[], int size, int element, int capacity, int index)

// here the capacity refers to the totla size of the array

{

    if (size >= capacity)

    {

        printf("INVALID!!");

    }

    for (int i = size - 1; i >= index; i--)

    {

        arr[i + 1] = arr[i];

        /\* this statement states that array ka (i+1) member's value would be equal to the value of array ka (i) member

        basically right shifting is happening

        eg...4th element would now become 5 th element and 5 th element becomes 6 th element

        just because we took array already bigger therefore we can accomodate the 6 th element in the array\*/

    }

    arr[index] = element;

    /\*now the right shifting has happened and according to the for loop used we will have a position blank

    that is where we will insert our new element

    therefore we will have to adjust our for loop such that we can add our new number at our required position \*/

}

int main(int argc, char const \*argv[])

{

    int arr[100] = {7, 8, 12, 27, 18};

    // we start by initialising an array

    int size = 5, element = 45, index = 1;

    // size here represents the used size of an array

    // element here represents the number to be added in the array

    // the index here represents the postion at which the new number is to be added

    display(arr, size);

    // used just for displaying the initial array

    Insertion(arr, size, element, 10, index);

    size += 1;

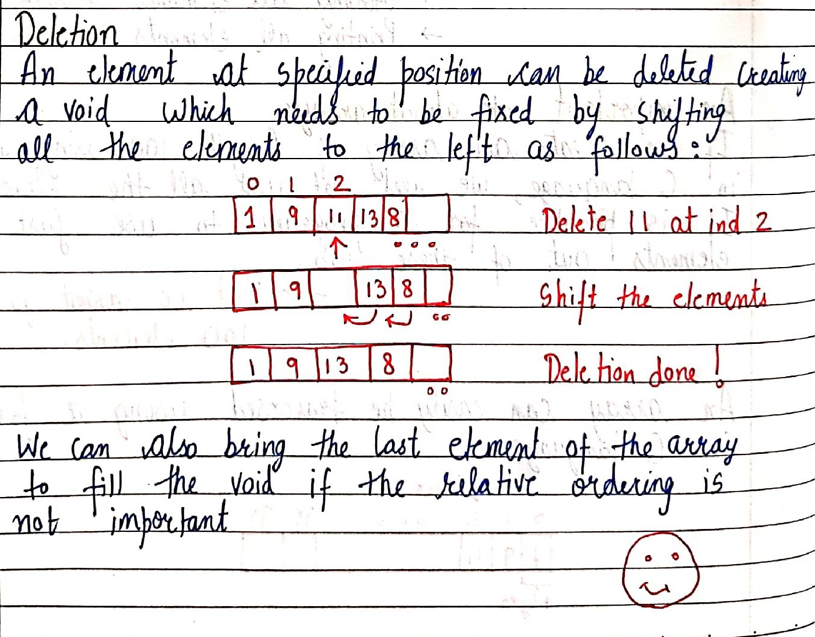
    /\* we are here increasing size because the the array would be displayed according to the size decided by us

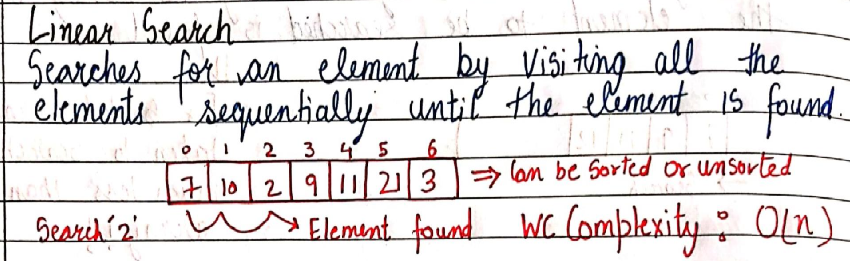
    now that new number is added in the array we would need to incrase the size of array to be dispayed \*/

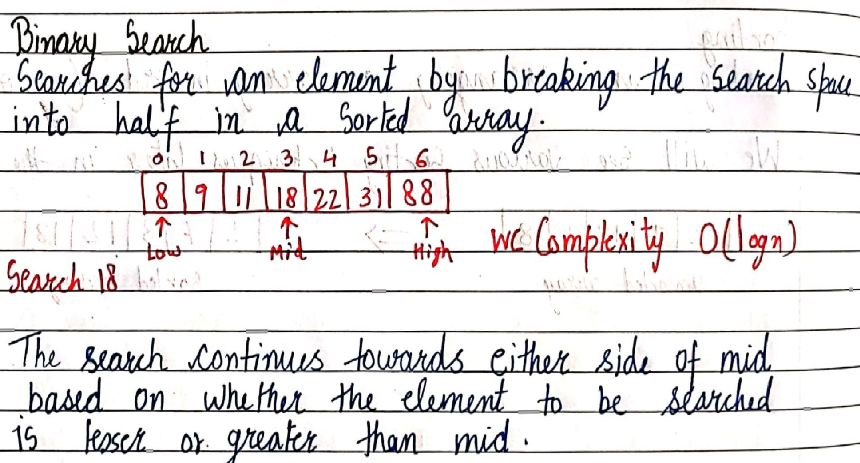
    display(arr, size);

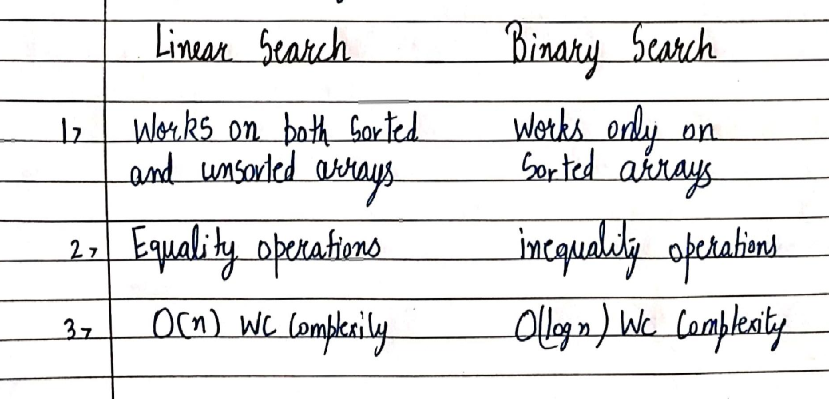
    return 0;

}

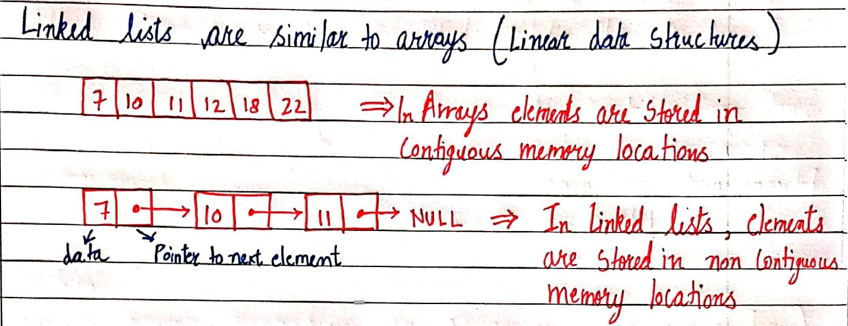


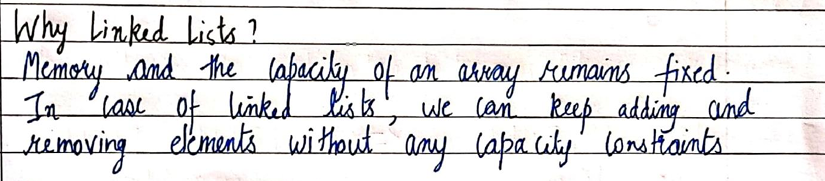


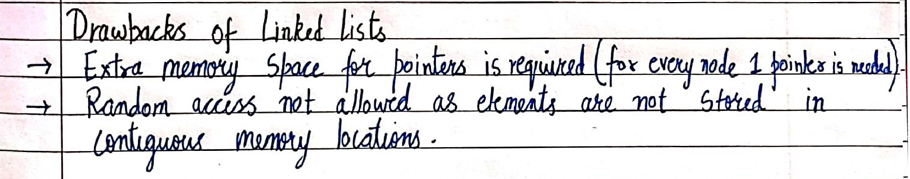




* LINKED LISTS









This statement means that in order to access the 5 th element of the linked list we need to go through all the 4 elements before it ,unlike array, wherein we can directly access any element as long as we know the index/Position of the element

* INSERTION AND DELETEION IN LINKED LIST
* #include <stdio.h>
* #include <stdlib.h>
* struct Node
* {
* int data;
* struct Node \*next;
* };
* void linkedlist(struct Node \*ptr)
* {
* while (ptr != NULL)
* // kabtk "ptr" null nhi hojata tabtk
* {
* printf("Element : %d\n", ptr->data);
* ptr = ptr->next;
* }
* }
* struct Node \*insertatfirst(struct Node \*head, int data)
* /\* function name se pehle uska return type hi aata hai (struct Node\*)
* phir kya intake lega vo () ke andar vo likha hai \*/
* {
* struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));
* // "struct node\* tells us the data type of "ptr"
* // "ptr" is also used to allocate memory dynamically
* // basically a new node is being created
* ptr->data = data;
* // "ptr" ka jo "data" hai vo equal hoga "data" ke
* ptr->next = head;
* //"ptr" ka jo "next" hai vo equal hoga "head" ke
* // "next" apne aap mein ek pointer hai which means "next" is pointing towards head
* return ptr;
* // this statement states that head=ptr
* };
* void insertinbetween(struct Node \*head, int data, int index)
* /\* phir kya intake lega vo () ke andar vo likha hai \*/
* {
* struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));
* // "struct node\* tells us the data type of ptr"
* // ptr is basically a variable jiske naam par memory allocate hori hai
* // basically creating a new node
* struct Node \*p = head;
* /\*"p" naam ka pointer is equal to "head"
* also "p" pointer ka data type "struct node\*" choose kra gya hai taaki vo
* "head" ke equal ho sake
* "struct node" structure ke andar do hi tareeke ke data type hai
* "int and pointer of the type struct node again " which means jo pointer hai vo
* struct node ko hi store kar sakta hai apne andar(which basically means uske equal ho jana)
* isliye "struct node" type ka varianle hum bana hi nhi sakte hai
* hum yahan chahe toh khud ka ek code bana sakte hai jahan easy hoga point vagera karna
* but abhi idhar hum harry ka code samjh rhe hai
* this is basically done to start "p" from the beginning and then after sometime move it forward\*/
* int i = 0;
* while (i != index - 1)
* // jabtk "p" humare index se ek pehle nhi pahunch jaata tabtk
* {
* p = p->next;
* // basically used to make "p" move forward and reach "index-1" position
* i++;
* }
* // jaise hi "p" pahunch jaaye "index-1" position par tab
* ptr->data = data;
* ptr->next = p->next;
* // "ptr" ek node hai
* /\*so this statement basicaly means that jis "next" naam ke pointer ka use hora tha
* to point towards the next member in linked list , using that same "next"
* "ptr" ka "next" (that is the pointer which point towards the next member of linked list) ab se
* vo equal hai "p"(index-1) ke "next" ke \*/
* p->next = ptr;
* // yeh kehra hai ki "p" ka "next" is equal to ptr
* // which basically means "p" ka "next" will point towards the new node "ptr"
* }
* void insertatend(struct Node \*head, int data)
* {
* struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));
* // basically creating a new node
* ptr->data = data;
* //"ptr" ka "data" jo "data" function ke through laaya gya hai usek equal hai
* struct Node \*p = head;
* /\*"p" naam ka pointer is equal to "head"
* also "p" pointer ka data type "struct node\*" choose kra gya hai taaki vo
* "head" ke equal ho sake
* "struct node" structure ke andar do hi tareeke ke data type hai
* "int and pointer of the type struct node again " which means jo pointer hai vo
* struct node ko hi store kar sakta hai apne andar(which basically means uske equal ho jana)
* isliye "struct node" type ka varianle hum bana hi nhi sakte hai
* hum yahan chahe toh khud ka ek code bana sakte hai jahan easy hoga point vagera karna
* but abhi idhar hum harry ka code samjh rhe hai
* this is basically done to start "p" from the beginning and then after sometime move it forward\*/
* while (p->next != NULL)
* // jabtk "p" ka "next" "NULL" nahi ho jata tabtk
* {
* p = p->next;
* // basically simple way to make "p" move forward from the start (head)
* }
* p->next = ptr;
* // "p" ka "next" ab new node "ptr" ko point karega (basically uske equal hojayega)
* ptr->next = NULL;
* // this states that new node "ptr" is the last node of linked list
* }
* struct Node \*deletefirst(struct Node \*head)
* /\* function name se pehle uska return type hi aata hai (struct Node\*)
* and kyuki humein iss function mein "head" ki zarurat hai isliye hum "head" bhi lenge \*/
* {
* struct Node \*ptr = head;
* // ek naya "pointer" banaya and isse point kara diya "head" par
* head = head->next;
* /\* ab " head " ki value hogyi jisse bhi "head" naam ke "node" mein jo "pointer"
* tha vo " pointer " jisse point karega uske jitni \*/
* free(ptr);
* /\* kyuki ab "head" ki value toh change hogi isliye poorana "head" lhaali hogya
* isliye hum usse "free" kara rhe hai\*/
* return head;
* // when the function will be called later on (in this program) this statement would mean "head"=new"head
* }
* struct Node \*deleteatindex(struct Node \*head, int index)
* /\* function name se pehle uska return type hi aata hai (struct Node\*)
* and kyuki humein iss function mein "head" and "index" ki zarurat hai isliye hum "head" and "index" bhi lenge \*/
* {
* struct Node \*p = head;
* // a pointer "p" is pointing towards "head"
* struct Node \*q = head->next;
* // a pointer "q" is pointing towards "pointer" part of the node "head" called "next"
* // "next" is basically pointing towards the next "node"
* // therefore "q" is pointing to the next "node" of "head" whatsoever
* for (int i = 0; i < index - 1; i++)
* // jabtk badaunga jabtk jisko delete karna wahan tak "q" na pahunch jaaye
* {
* p = p->next;
* q = q->next;
* }
* p->next = q->next;
* // jaise hi wahan tak pahunch gaya jahan delete karna hai tab
* // "p" ke "next" ko equal kardo "q" ke "next" ke
* free(q);
* // and "q" ko nasht kardo
* return head;
* // when the function will be called later on (in this program) this statement would mean "head"=new"head
* }
* struct Node \*deletewithvalue(struct Node \*head, int value)
* /\* function name se pehle uska return type hi aata hai (struct Node\*)
* and kyuki humein iss function mein "head" and "value" ki zarurat hai isliye hum "head" and "value" bhi lenge \*/
* {
* struct Node \*p = head;
* // a pointer "p" is pointing towards "head"
* struct Node \*q = head->next;
* // a pointer "q" is pointing towards "pointer" part of the node "head" called "next"
* // "next" is basically pointing towards the next "node"
* // therefore "q" is pointing to the next "node" of "head" whatsoever
* while (q->data != value && q->next != NULL)
* /\*jabtak "q" ka "data" humari given "value" ke equal nhi hojata tabtk
* and also
* incase vo value hai hi nahi humari linked list mein then
* jabtak "q" ka "next" "NULL" ke equal nhi hojata tabtk\*/
* {
* p = p->next;
* q = q->next;
* /\*basically pointer "p" and "q" ko aage badane ka tareeka hai\*/
* }
* if (q->data == value)
* // jaise hi "q" ka "data" humari guven "value" ke equal hojata hai vaise hi
* {
* p->next = q->next;
* // "p" ke "next" ko equal kardo "q" ke "next" ke
* free(q);
* // and "q" ko nasht kardo
* }
* return head;
* // when the function will be called later on (in this program) this statement would mean "head"=new"head
* }
* struct Node \*deleteatlast(struct Node \*head)
* /\* function name se pehle uska return type hi aata hai (struct Node\*)
* and kyuki humein iss function mein "head" ki zarurat hai isliye hum "head" bhi lenge \*/
* {
* struct Node \*p = head;
* // a pointer "p" is pointing towards "head"
* struct Node \*q = head->next;
* // a pointer "q" is pointing towards "pointer" part of the node "head" called "next"
* // "next" is basically pointing towards the next "node"
* // therefore "q" is pointing to the next "node" of "head" whatsoever
* while (q->next != NULL)
* // jabtk "q" ka "next" "NULL" ko point na kare tabtk
* {
* p = p->next;
* q = q->next;
* /\*basically pointer "p" and "q" ko aage badane ka tareeka hai\*/
* }
* p->next = NULL;
* //    jab "q" ka "next" "NULL" ke equal hojaye tab "p" ke "next" ko "NULL" ke equal kardo
* // basically "p" ke "next" ko equal kardo "q" ke "next" ke
* free(q);
* // and "q" ko destroy kardo
* return head;
* // when the function will be called later on (in this program) this statement would mean "head"=new"head
* }
* int main(int argc, char const \*argv[])
* {
* struct Node \*head;
* // pointer banaye jaare hai as to allocate memory dynamically aage jaakar
* /\* "head" ko as a "pointer" banaya jaara hai na ki as a simple "member of structure"
* so that we can allocate memory dynamically using it\*/
* struct Node \*second;
* // same
* struct Node \*third;
* // same
* head = (struct Node \*)malloc(sizeof(struct Node));
* /\* memory is allocated here
* idhar memory bhi struct node jitni banayi gayi hai kyuki head usss struct ka hi member hai
* and jo bhi data struct mein aana tha vo same head mein hai \*/
* second = (struct Node \*)malloc(sizeof(struct Node));
* // same
* third = (struct Node \*)malloc(sizeof(struct Node));
* // same
* head->data = 7;
* // head ke andar jo data naam ka variable hai uski value
* head->next = second;
* // head ke andar jo pointer vo equal hai second ke
* second->data = 8;
* // same
* second->next = third;
* // same
* third->data = 9;
* // same
* third->next = NULL;
* // last member of linked list points to NULL
* linkedlist(head);
* // ek baar "head" ki value print karo jab kuchh change nhi kiya
* // head = insertatfirst(head, 56);
* // head = "insertatfirst" function which returns ptr
* // therefore head = ptr
* // insertinbetween(head, 56,1);
* // insertatend(head,56);
* // head = deletefirst(head);
* // "head"= new "head"
* // head=deleteatindex(head,1);
* // "head"= new "head"
* // head=deleteatlast(head);
* // "head"= new "head"
* head = deletewithvalue(head, 9);
* // "head"= new "head"
* linkedlist(head);
* // ek baar "head" ki value print karo jab changes laaye gaye hai
* return 0;
* }

LINKED LIST

NULL

DATA

POINTER

DATA

DATA

POINTER

POINTER

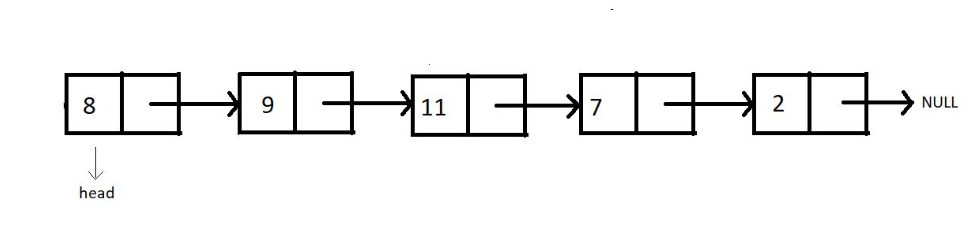
DATA

HEAD

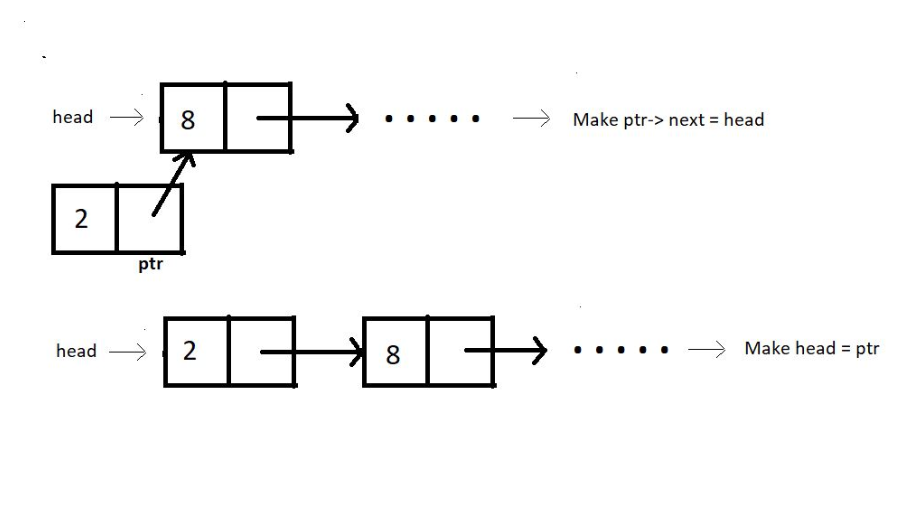
NULL because last member of linked list

points to NULL

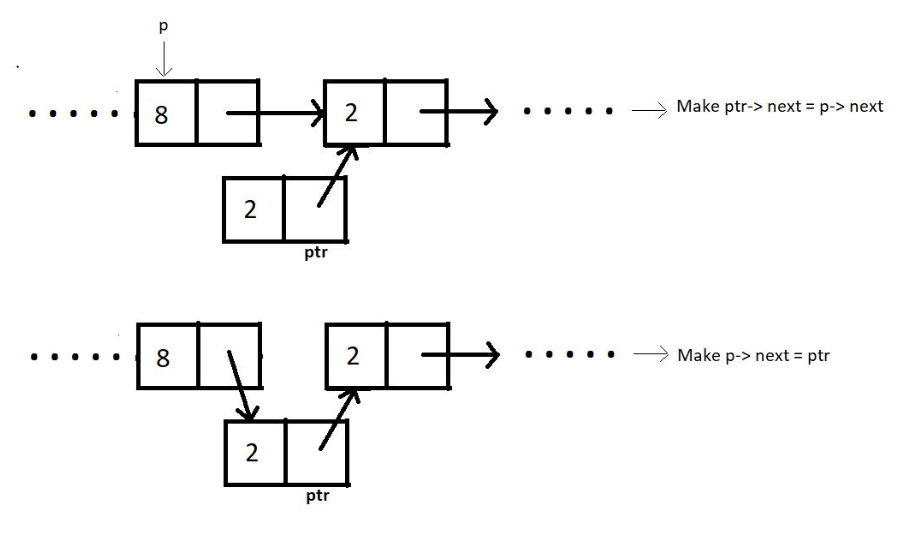
LINKED LIST



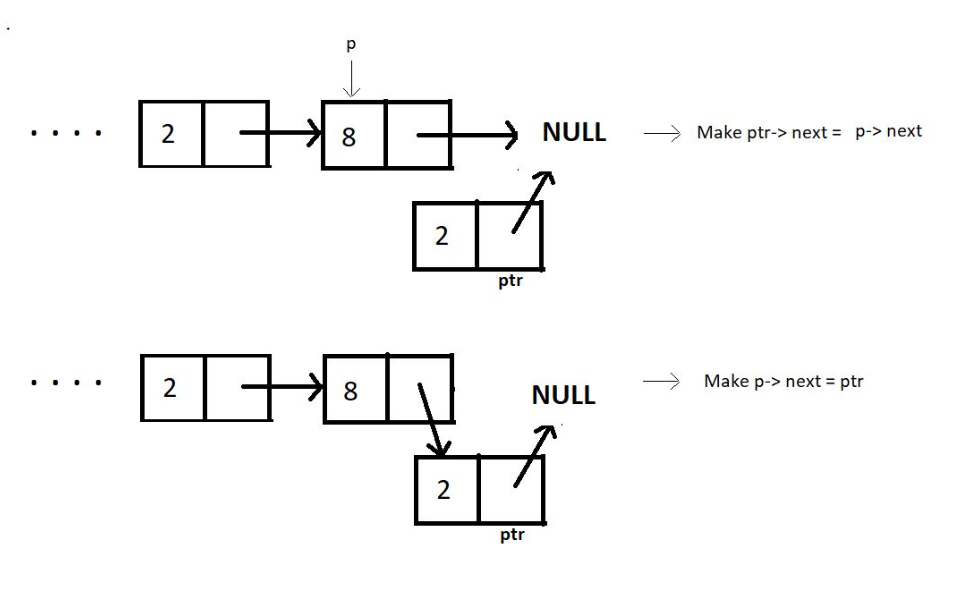
INSERTION AT BEGINNING



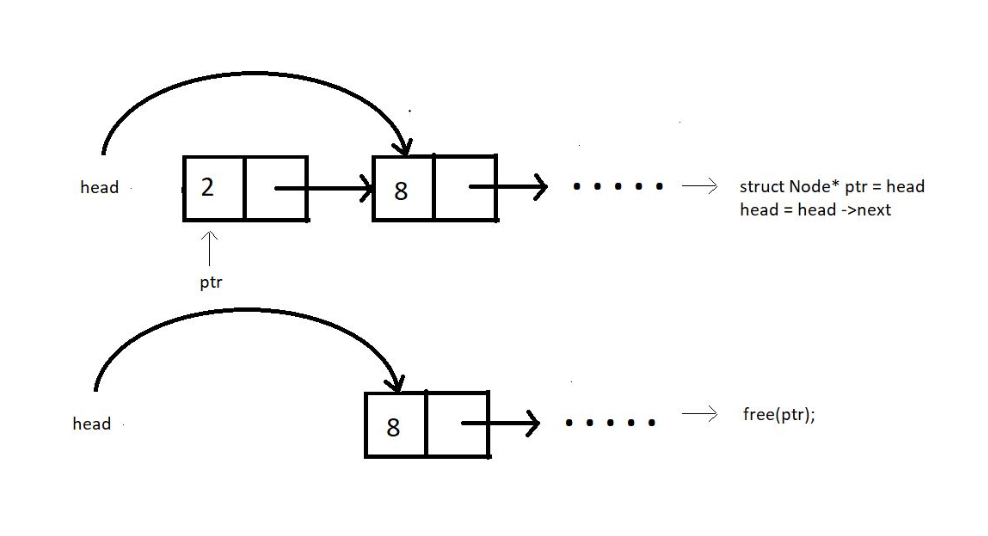
INSERTION IN BETWEEN



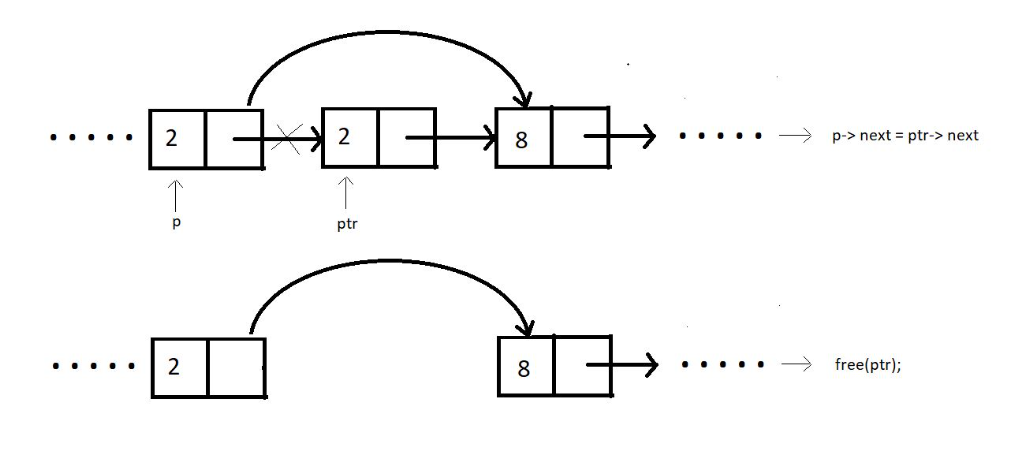
INSERT AT END



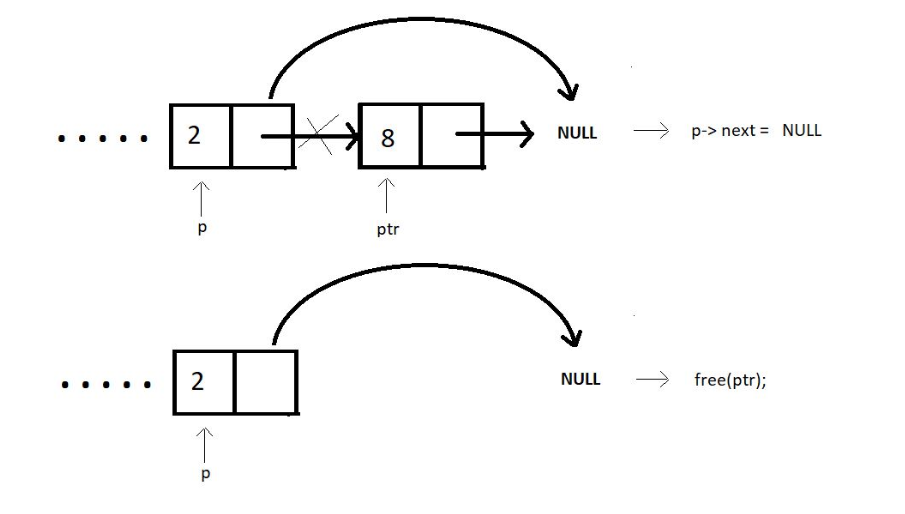
DELETE AT BEGINNING



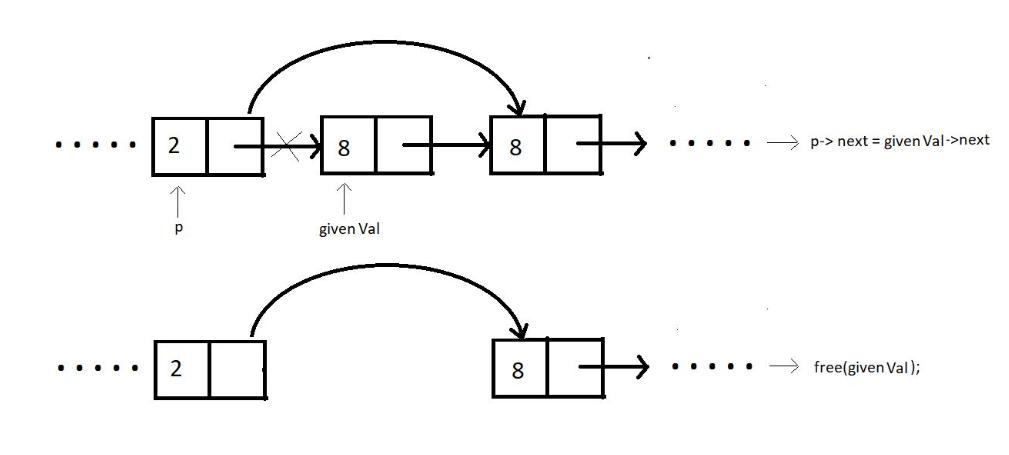
DELETE IN BETWEEN



DELETE AT END

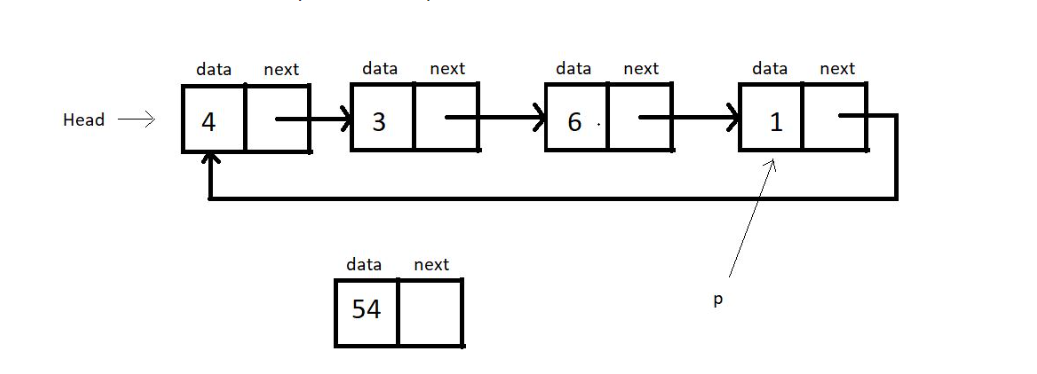


DELETE FOR ANY GIVEN VALUE

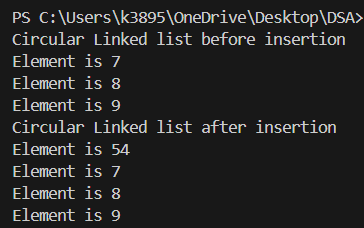


* CIRCULAR LINKED LIST

1. Unlike singly-linked lists, a circular linked list has no node pointing to NULL. Hence it has no end. The last element points at the head node.



* #include <stdio.h>
* #include <stdlib.h>
* struct Node
* {
* int data;
* struct Node \*next;
* };
* void linkedlisttraversal(struct Node \*head)
* /\* function name se pehle uska return type hi aata hai
* phir kya intake lega vo () ke andar vo likha hai \*/
* {
* struct Node \*ptr = head;
* /\*"p" naam ka pointer is equal to "head"
* also "p" pointer ka data type "struct node\*" choose kra gya hai taaki vo
* "head" ke equal ho sake
* "struct node" structure ke andar do hi tareeke ke data type hai
* "int and pointer of the type struct node again " which means jo pointer hai vo
* struct node ko hi store kar sakta hai apne andar(which basically means uske equal ho jana)
* isliye "struct node" type ka variable hum bana hi nhi sakte hai \*/
* do
* {
* printf("Element is %d\n", ptr->data);
* ptr = ptr->next;
* // "ptr" equal hai "next" ke jabtak vo head ke equal na hojaye
* } while (ptr != head);
* }
* struct Node \*insertAtFirst(struct Node \*head, int data)
* {
* struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));
* // "struct node\* tells us the data type of ptr"
* // ptr is basically a variable jiske naam par memory allocate hori hai
* // basically creating a new node
* ptr->data = data;
* // "ptr" ek node hai toh uss "node" ka jo "data" waala part hai vo equal hai "data" coming from function
* struct Node \*p = head->next;
* // "p" ko "node" ke "next" waale part ke equal kardiya hai
* while (p->next != head)
* // jabtk "p" ka "next" "head" ke equal nhi hojata tabtk
* {
* p = p->next;
* }
* // At this point p points to the last node of this circular linked list
* p->next = ptr;
* // badically "ptr" ek naya "node" hai jo inset hua hai first par
* ptr->next = head;
* // "ptr" ka "next" points towards " (old) head"
* head = ptr;
* // ab "head" "ptr" ban chuka hai
* return head;
* }
* int main(int argc, char const \*argv[])
* {
* struct Node \*head;
* // pointer banaye jaare hai as to allocate memory dynamically aage jaakar
* /\* "head" ko as a "pointer" banaya jaara hai na ki as a simple "member of structure"
* so that we can allocate memory dynamically using it\*/
* struct Node \*second;
* // same
* struct Node \*third;
* // same
* head = (struct Node \*)malloc(sizeof(struct Node));
* /\* memory is allocated here
* idhar memory bhi struct node jitni banayi gayi hai kyuki head usss struct ka hi member hai
* and jo bhi data struct mein aana tha vo same head mein hai \*/
* second = (struct Node \*)malloc(sizeof(struct Node));
* // same
* third = (struct Node \*)malloc(sizeof(struct Node));
* // same
* head->data = 7;
* // head ke andar jo data naam ka variable hai uski value
* head->next = second;
* // head ke andar jo pointer vo equal hai second ke
* second->data = 8;
* // same
* second->next = third;
* // same
* third->data = 9;
* // same
* third->next = head;
* // last member of linked list points to "head" to make it a "circular linked list"
* printf("Circular Linked list before insertion\n");
* linkedlisttraversal(head);
* head = insertAtFirst(head, 54);
* printf("Circular Linked list after insertion\n");
* linkedlisttraversal(head);
* return 0;
* }



* DOUBLY LINKED LIST

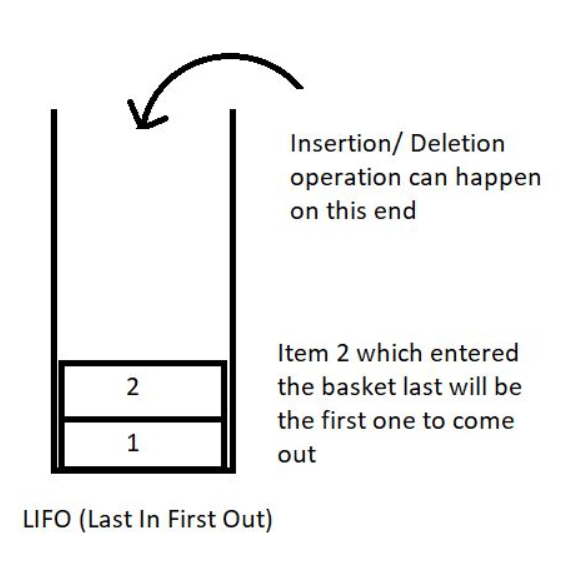
Each node contains a data part and two pointers in a doubly-linked list, one for the previous node and the other for the next node.



 Both the end pointers point to the NULL.

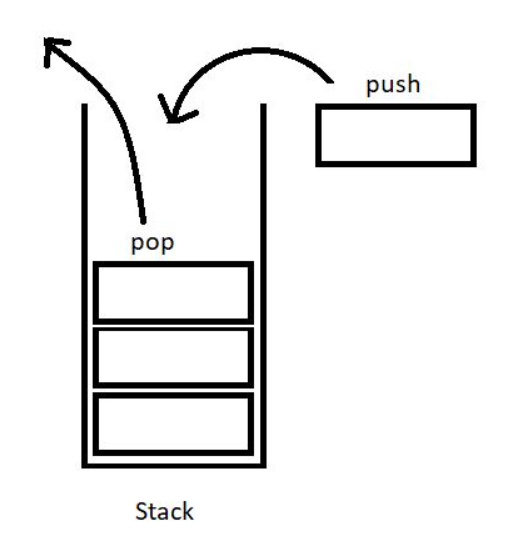
* A doubly linked list allows traversal in both directions. We have the addresses of both the next node and the previous node. So, at any node, we’ll have the freedom to choose between going right or left.
  + STACK

Any operation on the stack is performed in LIFO (Last In First Out) order. This means the element to enter the container last would be the first one to leave the container.



Here are some of the basic operations we would want to perform on stacks:

1. push(): to push an element into the stack
2. pop(): to remove the topmost element from the stack
3. peek(index): to return the value at a given index
4. isempty() / isfull() : to determine whether the stack is empty or full to carry efficient push and pull operations.



#include <stdio.h>

#include <stdlib.h>

struct Stack

{

    int size;

    int top;

    int\* arr;

};

int isEmpty(struct Stack\* ptr)

// return type is "int"

// basically "ptr" here is "s" as passed when the function is called

{

    if (ptr->top == -1)

// condition will be true when "ptr" ka "top" =-1

    {

        return 1;

        // TRUE

    }

    else

    {

        return 0;

        // FALSE

    }

}

int isFull(struct Stack \*ptr)

// return type is "int"

// basically "ptr" here is "s" as passed when the function is called

{

    if (ptr->top == ptr->size - 1)

    // condition will be true when "ptr" ka "top" = "ptr" ka "size-1"

    {

        return 1;

        // TRUE

    }

    else

    {

        return 0;

        // FALSE

    }

}

int main(int argc, char const \*argv[])

{

    struct Stack\* s;

    // "struct Stack" data type ka pointer "s"

    s->size = 80;

    // "s" ka "size"

    s->top = -1;

    // "s" ka "top"

    // "top" basically tells us about the topmost position of the stack

    // if it is "-1" then it is empty

    // if it is equal to a number then that number is the topmost number

    s->arr = (int \*)malloc(s->size \* sizeof(int));

    // "s" ka jo "array" hai of the type "int" uska size hai equal to the "size"\*"int"

    s->arr[0]=7;

    s->top++;

    // pushing an element manually

    if (isEmpty(s))

    // "if" the condition is true then

    {

        printf("the stack is empty");

    }

    else if (isFull(s))

    {

        printf("the stack is full");

    }

    else

    {

        printf("the stack is not empty");

    }

    return 0;

}

/////////

#include <stdio.h>

#include <stdlib.h>

struct stack

{

    int size;

    int top;

    int \*arr;

};

int isEmpty(struct stack \*ptr)

// return type is "int"

// basically "ptr" here is "sp" as passed when the function is called

{

    if (ptr->top == -1)

    // if "ptr" ka "top" is equl to -1 then

    {

        return 1;

        // TRUE

    }

    else

    {

        return 0;

        // FALSE

    }

}

int isFull(struct stack \*ptr)

// return type is "int"

// basically "ptr" here is "sp" as passed when the function is called

{

    if (ptr->top == ptr->size - 1)

    {

        return 1;

        // TRUE

    }

    else

    {

        return 0;

        // FALSE

    }

}

void push(struct stack \*ptr, int val)

// basically "ptr" here is "sp" as passed when the function is called

// jab function call hoga tab ek "value(val)" aayegi vo yahan use karenge

{

    if (isFull(ptr))

    // first of all we need to check if the "stack" is not full already

    {

        printf("Stack Overflow");

    }

    else

    {

        ptr->top++;

        //"ptr" ka "top" bada diya(increment)

        // jo upar likha tha ki jaise jaise members add hinge vaise "top" ki value badegi vo hi hora hai

        ptr->arr[ptr->top] = val;

        // "ptr" ke "arr" ki jo "top" position hai vo equal hai "val" ke

        /\* yahan "[ptr->top]" isliye likha gaya sirf yeh batane ke liye ki jis "top" position ki baat hori hai

         vo abhi bhi "ptr" waale "top" ki hi position ki baat hori hai\*/

        /\*basically "a" ke "b" ke "c" ki value batane ke liye  \*/

    }

}

int pop(struct stack \*ptr)

// basically "ptr" here is "sp" as passed when the function is called

// return type "int"

{

    if (isEmpty(ptr))

    // first of all we need to check if the "stack" is not empty already

    {

        printf("Stack Underflow");

    }

    else

    {

        int val = ptr->arr[ptr->top];

        // "val" ko iss function mein yahan initialize kara gaya hai

        // "val" is equal to "ptr" ka jo "arr" tha uska jo "top" hai uske equal

        /\* yahan "[ptr->top]" isliye likha gaya sirf yeh batane ke liye ki jis "top" position ki baat hori hai

         vo abhi bhi "ptr" waale "top" ki hi position ki baat hori hai\*/

        /\*basically "a" ke "b" ke "c" ki value batane ke liye  \*/

        ptr->top--;

        //"ptr" ka "top" ghata diya(decrement)

        // jo upar likha tha ki jaise jaise members niklenge vaise "top" ki value kam hogi

        return val;

        // basically "pop" ki retrun value batayi gayi hai

    }

}

int peek(struct stack \*sp, int i)

// return type "int"

// "sp" mein toh sab kuchh hora hai isliye vo chahiye and "i" is used to bring "j" from the for loop

{

    int arrayind = sp->top - i + 1;

    // this is basically done so that the lowest position we can get to is "0"

    // "0" ke neeche toh koi element hi nahi hote so wahaan tk jaana hi nahi hai

    // "arrayind" naam ka  integer variable jo equal hai "sp" ke "top"-"i"+1 ke

    // "top" ki value fix rahegi whatever it is but "i" ki value "j" ke hisaab se change hogi

// "i" ki value hum denge using function recall

    if (arrayind < 0)

    // if "arrayind" is less than 0 then....

    {

        printf("Not a valid position for the stack\n");

    }

    else

    {

        return sp->arr[arrayind];

        // otherwise return "Sp" ke "arr" mein jo "arrayInd" uski value

    }

}

int main(int argc, char const \*argv[])

{

    struct stack \*sp = (struct stack \*)malloc(sizeof(struct stack));

    // "struct stack" type ka pointer named "sp" is used to allocate memory for.......

    sp->size = 10;

    sp->top = -1;

    /\* "top" is a variable which tells us the index of the topmost element of stack

        humne beginning "-1" se kari hai which potrays that abhi "stack" mein kuchh bhi nahi hai

        further jaise jaise "stack" mein cheezei bharti jaayengi vaise vaise "top" ki value change hoti rahegi\*/

    sp->arr = (int \*)malloc(sp->size \* (sizeof(int)));

    // "arr" ek pointer hi hai

    // "sp" ke "arr" ko bhi isliye banaya gaya hai taaki memory allocate kari jaaye

    // equal to the size of "sp ka jitna size lenge \* size of int"

    printf("The stack has been created succesfully\n");

    printf("%d\n", isFull(sp));

    // to verify wheteher the stack is empty or not

    // if full then result will be 1 otherwise 0

    printf("%d\n", isEmpty(sp));

    // to verify whether the stack is empty or not

    // if empty then result will be 1 otherwise 0

    push(sp, 566);

    // "sp" pointer ke andar hi saare kaam hore hai toh usko pass kara hai and required value

    push(sp, 50);

// ''

    push(sp, 59);

// ''

    push(sp, 58);

// ''

    push(sp, 57);

// ''

    push(sp, 56);

// ''

    push(sp, 55);

// ''

    push(sp, 53);

// ''

    push(sp, 52);

// ''

    push(sp, 51);

// ''

    printf("%d\n", isFull(sp));

    // to verify wheteher the stack is empty or not

    // if full then result will be 1 otherwise 0

    printf("%d\n", isEmpty(sp));

    // to verify wheteher the stack is empty or not

    // if empty then result will be 1 otherwise 0

    printf("Popped %d from the stack\n", pop(sp));

    // "pop" ki return value ke hisaab se answer aayega

    for (int j = 0; j < sp->top + 1; j++)

    // basically a "for" loop is used to write elements of stack

    // "top" starts from -1 therefore first elemnt will be filled in 0th index

    // yeh tbtk chlta jaayega jab tk "sp" ka "top" + 1 se chhota hai "j" (basically topmost position tak)

    {

        printf("the value at index is %d is %d\n", j, peek(sp, j));

    }

    return 0;

}

* STACK IMPLEMENTATION USING LINKED LIST

#include<stdio.h>

#include<stdlib.h>

struct Node

{

    int data;

    struct Node \* next;

};

struct Node\* top = NULL;

/\*

we have "top" in "main function" as well as in "pop" function

therefore we have created "top" as global variable

\*/

/\*

we have equalled "top" to "NULL" because as we can see is

in "linkedlisttraversal function" it will print element until it reaches "NULL"

so if we have mentioned "top" as "NULL" it will keep running even if

"top" doesn't even equal to some "node"

\*/

/\*

"top" basically is that "Node" which is at the "TOP"

\*/

void linkedListTraversal(struct Node \*ptr)

{

    while (ptr != NULL)

    {

        printf("Element: %d\n", ptr->data);

        ptr = ptr->next;

    }

}

int isEmpty(struct Node\* top)

/\*

Since it is returning "1" and "0"

we have mentioned the return type as "int"

\*/

{

    if (top==NULL)

    /\*

    "top" is an pointer pointing towards the top of linked list

    therefore if the top of linked list = NULL

    then the "stack is empty"

    thus it will "return 1"(true)

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int isFull(struct Node\* top)

/\*

yahan bracket ke andar "struct Node\* top" liya gaya hai

taaki hum check kar sake ki "Linked List" "full" hai ya nahi

"top" points towards the "top of linked list"

\*/

{

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

    /\*

    we have created a pointer "p" dynamically

    so if the "heap" will already be full

    then it wont be able to create a new node (in this case p)

    and when "malloc" is unable to dynamically alot memory

    it returns "NULL"

    \*/

    if(p==NULL)

    {

        return 1;

    }

    else{

        return 0;

    }

}

struct Node\* push(struct Node\* TOP, int x)

/\*

since this fucntion is returning "top"

and "top" is of "data type" "struct Node\*"

thus we have mentioned "return data type" as "Struct Node \*"

\*/

{

    if(isFull(TOP))

    // before "pushing" we will first check if the "linkedlist" is full or not

    {

        printf("Stack Overflow\n");

    }

    else

    {

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

        /\*

        agar "full" nahi hai toh

        ek naya "node" banao "n"

        \*/

        n->data = x;



        /\*

        "n ka data"="x" jo ki function call karte time pass kara jaayega

        \*/

        n->next = TOP;

        /\*



        "n ke next" ko "top" ko equal kardo

        "TOP"="top"

        \*/



        TOP = n;

        /\*

        and now our "new top"="n"

        basically jitne elements "push" hote jaayenge

        vo "top" bante jaynege

        jo "last" mein "push" hoga vo "top" hoga

        \*/

        return TOP;

    }

}

int pop(struct Node\* tp)

/\*

"tp" is basically "top"

\*/

/\*

as "pop" will return "the integer which is to be popped"

therefore we have mentioned return type as "int"

\*/

{

    if(isEmpty(tp))

    // before "popping" we will frist check if the "linkedlist" is empty or not



    {

        printf("Stack Underflow\n");

    }

    else

    {

        struct Node\* z = tp;

        /\*

        we have created a "pointer z of data type struct node "

        which is equal to "top"

        \*/

        top = (tp)->next;

        /\*

        our "new top" will be equal to now

        "next of previous top"

        \*/

        int x = z->data;



        /\*

        since "z"="previous top "

        therefore "x"="previous top ka data"

        \*/

        free(z);

        // now that function of "z" is done we can clear it from heap

        return x;

    }

}

int main(){

    top = push(top, 78);

    /\*

    "top"="return value of push"

    \*/

    top = push(top, 7);

    top = push(top, 8);

    // linkedListTraversal(top);

    int element = pop(top);

    /\*

    "element"="return calue of pop"

    \*/

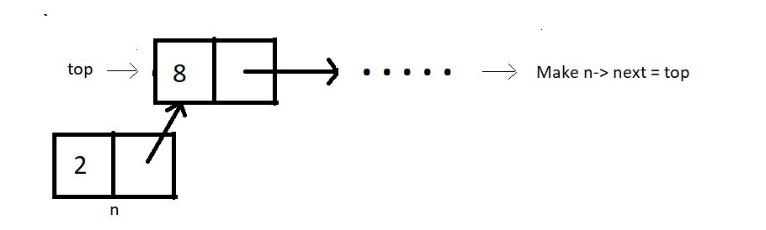
    printf("Popped element is %d\n", element);

    linkedListTraversal(top);

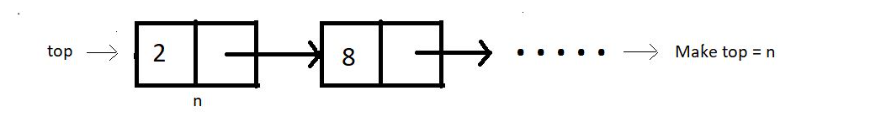


    return 0;

}

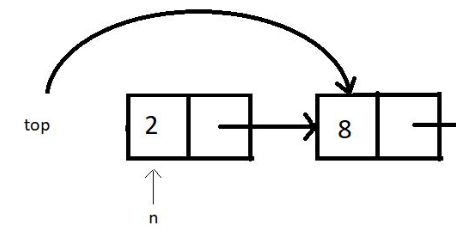




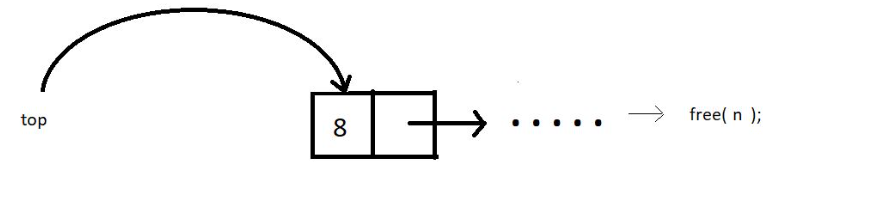




//POP









* PARENTHESIS MATCHING

we tried making parentheses matching intuitive and more understandable using stacks. We followed one simple algorithm to accomplish that.

The algorithm states:

* Everytime you come across an opening parenthesis, push it in the stack.
* Everytime you come across a closing parenthesis, pop one opening parenthesis out from the stack.
* We call this match of parentheses unbalanced when we encounter either of the two of these troubles:

1. There is no more opening bracket inside the stack to pop, and you come across a closing bracket.
2. The stack size is not zero, or there are still more than zero opening brackets present in the stack after you come across EOE(end-of-expression).

#include <stdio.h>

#include <stdlib.h>

struct stack

{

    int size;

    int top;

    char \*arr;

};

int isEmpty(struct stack \*ptr)

{

    if (ptr->top == -1)

    /\*

    agar "sp" ka "top"="-1"

    then it is "empty"

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int isFull(struct stack \*ptr)

{

    if (ptr->top == ptr->size - 1)

    /\*

    agar "sp" ka "top" = "sp" ka "size" -1

    then it is "Full"

    since "sp" ka "top" was starting from "top=-1"(our 1st index is 0th position)

    tabhi aisa likha hai yahan

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void push(struct stack \*ptr, char val)

{

    if (isFull(ptr))

    {

        printf("Stack Overflow! Cannot push %c to the stack\n", val);

    }

    else

    {

        ptr->top++;

        /\*

        since "ptr"="sp"

        therefore "sp" ka "top" increase kardo

        \*/

        ptr->arr[ptr->top] = val;

        /\*

        Arrays in C/C++ are indexed using integers, regardless of the type of elements stored in the array.

        "ptr->top" is used as an "index" to access elements in the array "arr".

        The statement "ptr->arr[ptr->top]" accesses the element at "top index"  in the array "arr".

        The data type of "arr" is "char[]"", so "ptr->arr[ptr->top]" refers to a "char" element.

        Assigning "val" (which is a char) to "ptr->arr[ptr->top]"" is perfectly valid because both are of type "char"

        \*/

    }

}

char pop(struct stack \*ptr)

{

    if (isEmpty(ptr))

    {

        printf("Stack Underflow! Cannot pop from the stack\n");

        return -1;

    }

    else

    {

        char val = ptr->arr[ptr->top];

        /\*

        "val"="sp" ka "arr" ka jo bhi "top" element hai

        \*/

        ptr->top--;

        /\*

        "sp" ke "top" ko decrease kardo by 1

        \*/

        return val;

    }

}

int parenthesisMatch(char \*exp)

{

    // Create and initialize the stack

    struct stack \*sp;

    /\*

    we have created a "Pointer" "sp" without actually assigning "Address of any variable"

    to it,therefore it will just store "memory address" only

    \*/

    sp->size = 100;

    // size of the "pointer sp"

    sp->top = -1;

    // "top" is made as a "pointer" to "index" of "stack" and sometimes of "arr"(character array)

    sp->arr = (char \*)malloc(sp->size \* sizeof(char));

    /\*

    "sp" ka jo "character pointer" hai named "arr"

    we are using that "arr" to allocate memory dynamically

    \*/

    for (int i = 0; exp[i] != '\0'; i++)

    {

    /\*

    jabtk hum "character array/string" ke last member tak nahi pahunch jaate

    that is "\0"

    \*/

        if (exp[i] == '(')

        /\*

        agar "string" ka koi bhi element "(" hai toh

        we need to use the "push" function

        \*/

        {

            push(sp, '(');

        }

        else if (exp[i] == ')')

        /\*

        agar "string" ka koi bhi element ")" hai toh

        we need to use the "pop" function

        \*/

        {

            if (isEmpty(sp))

            {

                return 0;

            }

            pop(sp);

        }

    }

    if (isEmpty(sp))

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int main()

{

    /\*

    When working with character arrays in C programming, double inverted commas ("") are used to enclose string literals.

    This means that when you declare a character array using a double-quoted string,

    the compiler automatically adds a null-terminator character (\0) at the end of the array, which is essential for proper string manipulation.

    n C programming, a string literal is a sequence of characters enclosed in double quotes (""), such as "hello" or "goodbye"

    \*/

    char \*exp = "((8)(\*--$$9))";

    // parenthesis to be checked

    /\*

    we are using "char \*exp" instead of "char exp" because

    we are initially reserving "blocks of memory" using the "pointer exp"

    then in that "blocks of memory" we are inserting "characters"

    that is why the "pointer" is of the "data type char"

    \*/

    if (parenthesisMatch(exp))

    /\*

    agar "stack" "empty" ya "full" nahi hua and we didn't get

    any false statement throughout the code then this will run

    \*/

    {

        printf("The parenthesis is matching");

    }

    else

    /\*

    agar "stack" "empty" ya "full" hua and we get

    any false statement throughout the code then this will run

    \*/

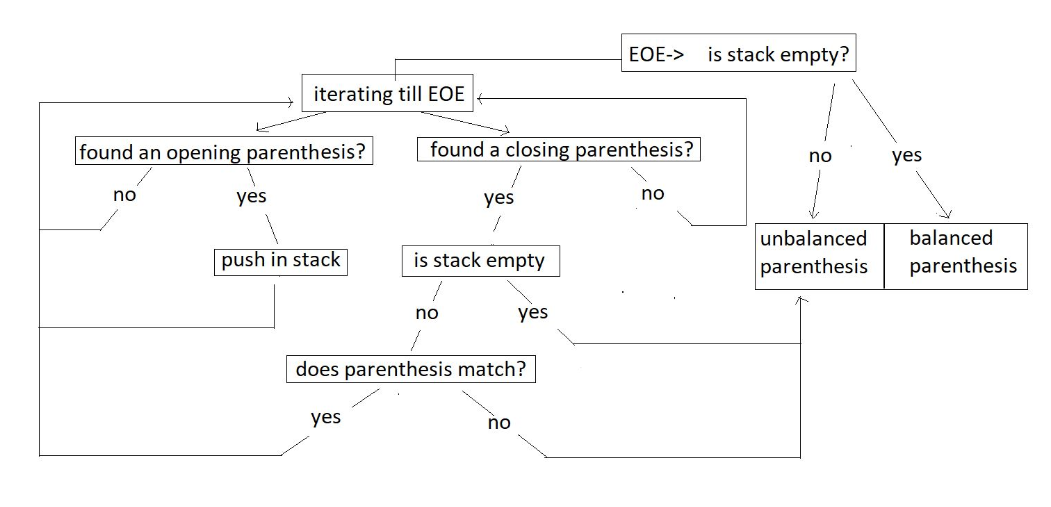
    {

        printf("The parenthesis is not matching");

    }

    return 0;

}



#include <stdio.h>

#include <stdlib.h>

struct stack

{

    int size;

    int top;

    char \*arr;

};

int isEmpty(struct stack \*ptr)

{

    if (ptr->top == -1)

    /\*

    agar "sp" ka "top"="-1"

    then it is "empty"

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int isFull(struct stack \*ptr)

{

    if (ptr->top == ptr->size - 1)

    /\*

    agar "sp" ka "top" = "sp" ka "size" -1

    then it is "Full"

    since "sp" ka "top" was starting from "top=-1"(our 1st index is 0th position)

    tabhi aisa likha hai yahan

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void push(struct stack\* ptr, char val)

{

    if(isFull(ptr))

    {

        printf("Stack Overflow! Cannot push %d to the stack\n", val);

    }

    else

    {

        ptr->top++;

        /\*

        since "ptr"="sp"

        therefore "sp" ka "top" increase kardo

        \*/

        ptr->arr[ptr->top] = val;

        /\*

        Arrays in C/C++ are indexed using integers, regardless of the type of elements stored in the array.

        "ptr->top" is used as an "index" to access elements in the array "arr".

        The statement "ptr->arr[ptr->top]" accesses the element at "top index"  in the array "arr".

        The data type of "arr" is "char[]"", so "ptr->arr[ptr->top]" refers to a "char" element.

        Assigning "val" (which is a char) to "ptr->arr[ptr->top]"" is perfectly valid because both are of type "char"

        \*/

    }

}

char pop(struct stack\* ptr)

{

    if(isEmpty(ptr))

    {

        printf("Stack Underflow! Cannot pop from the stack\n");

        return -1;

    }

    else

    {

        char val = ptr->arr[ptr->top];

         /\*

        "val"="sp" ka "arr" ka jo bhi "top" element hai

        \*/

        ptr->top--;

        /\*

        "sp" ke "top" ko decrease kardo by 1

        \*/

        return val;

    }

}

int match(char a, char b)

{

    if(a=='{' && b=='}'){

        return 1;

    }

    if(a=='(' && b==')'){

        return 1;

    }

    if(a=='[' && b==']'){

        return 1;

    }

  return 0;

}

int parenthesisMatch(char \* exp)

{

    // Create and initialize the stack

    struct stack\* sp;

    sp->size = 100;

    sp->top = -1;

    sp->arr = (char \*)malloc(sp->size \* sizeof(char));

    char popped\_ch;

    for (int i = 0; exp[i]!='\0'; i++)

    {

        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')

        {

            push(sp, exp[i]);

        }

        else if(exp[i]==')'|| exp[i]=='}' || exp[i]==']')

        /\*

        agar inme so koi bhi "bracket" humare expression mein aaya while traversing

        and iske aane ke baad

        \*/

        {

            if(isEmpty(sp))

            /\*

            agar inme so koi bhi "bracket" humare expression mein aaya while traversing

            and the "stack" is "empty" i.e it doesn't have "(,{,["

            then it will leave this loop and "parenthesis arent balanced"

            \*/

            {

                return 0;

            }

            popped\_ch = pop(sp);

            /\*

            if the "stack" isnt "empty" and there are "parenthesis" present

            then "pop"

            "popped\_ch"="return value" of "pop"

            \*/

            if(!match(popped\_ch, exp[i]))

            /\*

            if it doesn't "match" then leave the loop below

            //

            if(parenthesisMatch(exp))

            {

            printf("The parenthesis is balanced");

            }

            //

            "match function" is made to see if both the "parenthesis"

            the one "popped" and the other one we got while "traversing"

            is of the same kind or not

            it is done so because "pop" function will work even if

            it is any kind of "parenthesis"(which means even on two different kinds of

            "parenthesis" which shouldn't happen)

            \*/

            {

              return 0;

            }

        }

    }

    if(isEmpty(sp))

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int main()

{

    char \* exp = "[4-6]((8){(9-8)})";

    if(parenthesisMatch(exp))

    {

        printf("The parenthesis is balanced");

    }

    else

    {

        printf("The parenthesis is not balanced");

    }

    return 0;

}

* INFIX,PREFIX,POSTFIX

Infix : A\*(B+C)\*D

Postfix: ABC+\*D

[first the computer try ro reach an “operand”

When it reaches and “operand”

In this case “+” then

It will do the “operation” as mentioned by the

“OPERAND” to the previous two variables

In this case B and C(B+C)

Then it will go to the next “operand”

And do the “operation” as mentioned by the

“operand” to previous two “variables”

In this case A\*(B+C) and so on…

**Infix:**

This is the method we have all been studying and applying for all our academic life. Here the operator comes in between two operands. And we say, two is added to three. For eg: 2 + 3, a \* b, 6 / 3 etc.

< operand 1 >< **operator** >< operand2 >

**Prefix:**

This method might seem new to you, but we have vocally used them a lot as well. Here the operator comes before the two operands. And we say, Add two and three. For e.g.:  + 6 8, \* x y, -  3 2 etc.

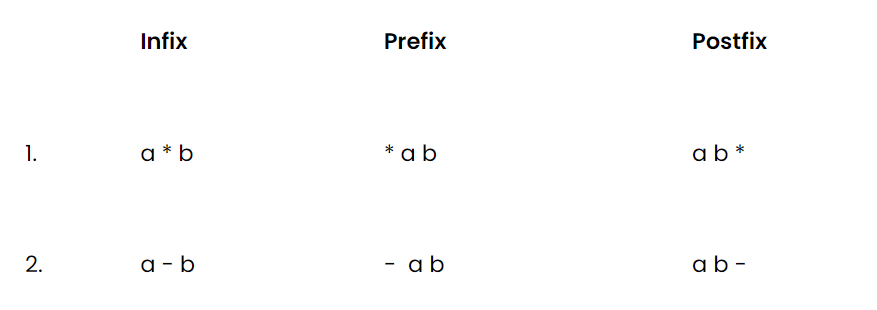
<**operator**>< operand 1 >< operand2 >

**Postfix:**

This is the method that might as well seem new to you, but we have used even this in our communication. Here the operator comes after the two operands. And we say, Two and three are added. For e.g.:  5 7 +, a  b \*,  12 6 / etc.

< operand 1 >< operand2 >< **operator** >

NOTE: Machine finds it easy to evaluate Postfix



**Converting infix to prefix:**

Consider the expression, **x - y \* z**.

1. Parentheses the expression. The infix expression must be parenthesized by following the operator precedence and associativity before converting it into a prefix expression. Our expression now becomes **( x - ( y \* z ) )**.

2. Reach out to the innermost parentheses. And convert them into prefix first, i.e.  **( x - ( y \* z ) )**changes to **( x - [ \* y z ] )**.

3. Similarly, keep converting one by one, from the innermost to the outer parentheses.  **( x - [ \* y z ] )  → [ - x \* y z ].**

4. And we are done.

**Converting infix to postfix:**

Consider the same expression, **x - y \* z**.

5. Parentheses the expression as we did previously. Our expression now becomes **( x - ( y \* z ) )**.

6. Reach out to the innermost parentheses. And convert them into postfix first, i.e.  **( x - ( y \* z ) )**changes to **( x - [ y z \* ] )**.

7. Similarly, keep converting one by one, from the innermost to the outer parentheses.  **( x - [ y z \* ] )  → [ x y z \* - ].**

8. And we are done.

Similarly the expression p - q -  r / a, follows the following conversions to become a prefix expression:

* **p - q -  r / a**  →  ( ( p - q ) -  ( r / a ) ) →  ( [ - p q ] - [ / r a ]  )  →**- - p q / r a**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct stack

{

    int size;

    int top;

    char \*arr;

};

int stackTop(struct stack \*sp)

{

    return sp->arr[sp->top];

}

int isEmpty(struct stack \*ptr)

{

    if (ptr->top == -1)

    /\*

    agar "sp" ka "top"="-1"

    then it is "empty"

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

int isFull(struct stack \*ptr)

{

    if (ptr->top == ptr->size - 1)

    /\*

    agar "sp" ka "top" = "sp" ka "size" -1

    then it is "Full"

    since "sp" ka "top" was starting from "top=-1"(our 1st index is 0th position)

    tabhi aisa likha hai yahan

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void push(struct stack \*ptr, char val)

{

    if (isFull(ptr))

    {

        printf("Stack Overflow! Cannot push %d to the stack\n", val);

    }

    else

    {

        ptr->top++;

        /\*

        since "ptr"="sp"

        therefore "sp" ka "top" increase kardo

        \*/

        ptr->arr[ptr->top] = val;

        /\*

        Arrays in C/C++ are indexed using integers, regardless of the type of elements stored in the array.

        "ptr->top" is used as an "index" to access elements in the array "arr".

        The statement "ptr->arr[ptr->top]" accesses the element at "top index"  in the array "arr".

        The data type of "arr" is "char[]"", so "ptr->arr[ptr->top]" refers to a "char" element.

        Assigning "val" (which is a char) to "ptr->arr[ptr->top]"" is perfectly valid because both are of type "char"

        \*/

    }

}

char pop(struct stack \*ptr)

{

    if (isEmpty(ptr))

    {

        printf("Stack Underflow! Cannot pop from the stack\n");

        return -1;

    }

    else

    {

        char val = ptr->arr[ptr->top];

        /\*

        "val"="sp" ka "arr" ka jo bhi "top" element hai

        \*/

        ptr->top--;

        /\*

        "sp" ke "top" ko decrease kardo by 1

        \*/

        return val;

    }

}

int precedence(char ch)

/\*

Yeh “function” banaya hi “precedence” check karne ke liye hai

“operands” ki

Jo zyaada “precedence” ka “operand” hoga vo greater

Number return karega

And neeche humne “if” statement aise use kari hai ki

Whoever returns greates number will get higher “precedence”

{

    if (ch == '\*' || ch == '/')

        return 3;

    else if (ch == '+' || ch == '-')

        return 2;

    else

        return 0;

}

int isOperator(char ch)

{

    if (ch == '+' || ch == '-' || ch == '\*' || ch == '/')

        return 1;

    else

        return 0;

}

char \*infixToPostfix(char \*infix)

/\*

since this "function" is gonna "return"

"string" that is "postfix"

therefore it is of the type "char \*"

\*/

{

    struct stack \*sp = (struct stack \*)malloc(sizeof(struct stack));

    sp->size = 10;

    sp->top = -1;

    sp->arr = (char \*)malloc(sp->size \* sizeof(char));

    char \*postfix = (char \*)malloc((strlen(infix) + 1) \* sizeof(char));

    int i = 0; // Track infix traversal

    int j = 0; // Track postfix addition

    while (infix[i] != '\0')

    /\*

    jabtak "infix" i.e the "string passed"

    is not equal to /0 tabtk

    \*/

    {

        if (!isOperator(infix[i]))

        /\*

        agar "infix" mein "traverse" karte huye

        koi bhi "operand" nahi aaya toh

        \*/

        {

            postfix[j] = infix[i];

            /\*

            "postfix" "string" mein elements daaldo "infix" string ke

            \*/

            j++;

            i++;

            /\*

            aisa isliye kiya taaaki jo next elements hai

            vo next index mein bhare

            \*/

        }

        else

        /\*

        agar humein traverse karte huye koi "operand" mil jaata hai toh

        \*/

        {

            if (precedence(infix[i]) > precedence(stackTop(sp)))

            /\*

            agar "precedence" of "infix ka traversed member" i.e the passed "string"

            is more than the "precedence of top member of stack" then...

            \*/

            /\*

            for the very first time there wont be anything in "arr" because

            nothing has been pushed before

            but for the next time (when something would have been pushed)

            for that time we have mentioned "precedence(infix[i]) > precedence(stackTop(sp)"

            \*/

            {

                push(sp, infix[i]);

                i++;

            }

            else

            /\*

           agar "precedence" of "infix ka traversed member" i.e the passed "string"

           is less than or equal the "precedence of top member of stack" then...

           \*/

            {

                postfix[j] = pop(sp);

                /\*

                "postfix" "string" ke andar daaldo "popped element"

                \*/

                j++;

                /\*

                aisa isliye kiya taaaki jo next elements hai

                vo next index mein bhare

                \*/

            }

        }

    }

    while (!isEmpty(sp))

    /\*

    jabtk "stack" khaali na hojaye tabtak

    \*/

    {

        postfix[j] = pop(sp);

        /\*

        "postfix" ke "element" ko equal kardo "popped element" ke

        \*/

        j++;

    }

    /\*

    jab aisa hojaye to "postfix" ke "last element" par hum "traverse" kar rahe honge

    \*/

    postfix[j] = '\0';

    /\*

    toh uss "last element" ko equal lardo "\0"

    \*/

    return postfix;

}

int main()

{

    char \*infix = "x-y/z-k\*d";

    /\*

    char \*xyz="STRING"

    is one way of writing a string

    \*/

    printf("postfix is %s", infixToPostfix(infix));

    return 0;

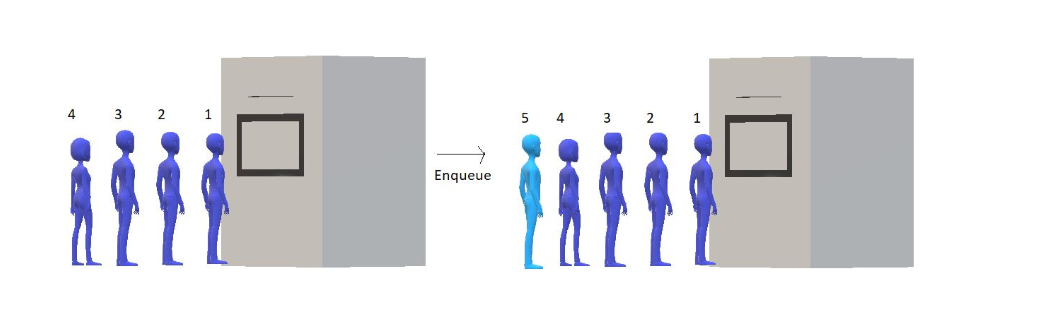
}

* Queue

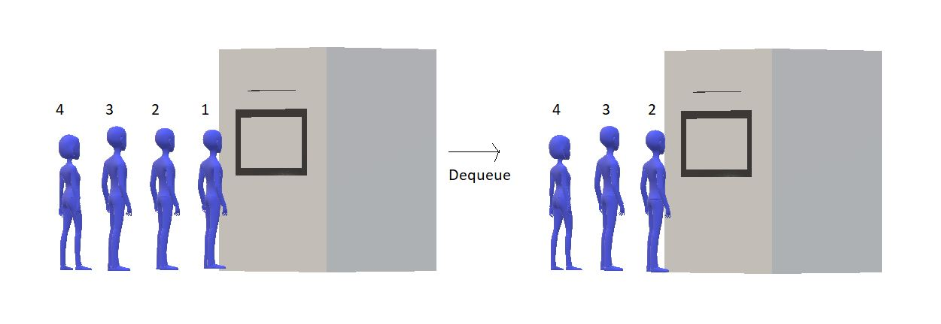
Unlike stacks, where we followed LIFO( Last In First Out ) discipline, here in the queue, we have FIFO( First In First Out).

In stacks, we had to maintain just one end, *head,* where both insertion and deletion used to take place, and the other end was closed. But here, in queues, we have to maintain both the ends because we have insertion at one end and deletion from the other end.

* 1. enqueue() : to insert an element in a queue.



* 1. dequeue(): to remove an element from the queue



#include <stdio.h>

#include <stdlib.h>

struct queue

{

    int size;

    int f;

    int r;

    int \*arr;

};

int ISEMPTY(struct queue \*q)

{

if (q->r==q->f)

/\*

agar "q" ka "r"="q" ka "f" then the "array" is "empty"

this is so because

"r" is the "index manager" which manages the latest member of the queue

that is the "member" recently added

whereas

"f" is the "index manager" which manages the oldest member of the queue

that is the "member" to be removed

and the "member" to be "added" and the "member" to be "removed"

can only be same when there is nothing in queue

\*/

{

    return 1;

}

else

{

    return 0;

}

}

int ISFULL(struct queue \*q)

{

    if (q->r == q->size - 1)

    /\*

    agar "q" ka "r" = "q" ka "size"-1 then it is "full"

    it is so because

    lets suppose "size=10" so when "r" reaches "9" th index

    array is full as the "index" starts from "0" and reaches "9"

    \*/

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void enqueue(struct queue \*q, int val)

{

    if (ISFULL(q))

    {

        printf("This queue is full\n");

    }

    else

    {

        q->r++;

        /\*

        "q" ke "r" ko increase kardo

        \*/

        q->arr[q->r] = val;

        /\*

        "q" ke "arr" ki "r" index par "val" daaldo

        \*/

    }

}

int dequeue(struct queue \*q)

{

    int a;

    if (ISEMPTY(q))

    {

        printf("This queue is empty\n");

    }

    else

    {

        q->f++;

        /\*

        "q" ke "f" ko "increase" kardo

        \*/

        a = q->arr[q->f];

        /\*

        "a" = "q" ke "arr" par "f"th index par jo value hai

        \*/

    }

       return a;

}

int main(int argc, char const \*argv[])

{

    struct queue q;

    q.size = 100;

    q.f = q.r = -1;

    /\*

    both the "index managers" starts from "-1" as

    "array" is "empty" initially

    \*/

    q.arr = (int \*)malloc(q.size \* sizeof(int));

    enqueue(&q, 12);

    enqueue(&q, 15);

    enqueue(&q, 1);

    printf("Dequeuing element %d\n",dequeue(&q));

    printf("Dequeuing element %d\n",dequeue(&q));

    return 0;

}

NOTE: Whenever a “structure” is made

It is meant that a “block of space” is being allocated to that “structure”

The “size” of “block of space” depends on the “data types” present inside the “structure”

struct queue

{

    int size;

    int f;

    int r;

    int \*arr;

};

    struct queue q;

This basically means that

“queue” name ka “structure” banaya gaya hai

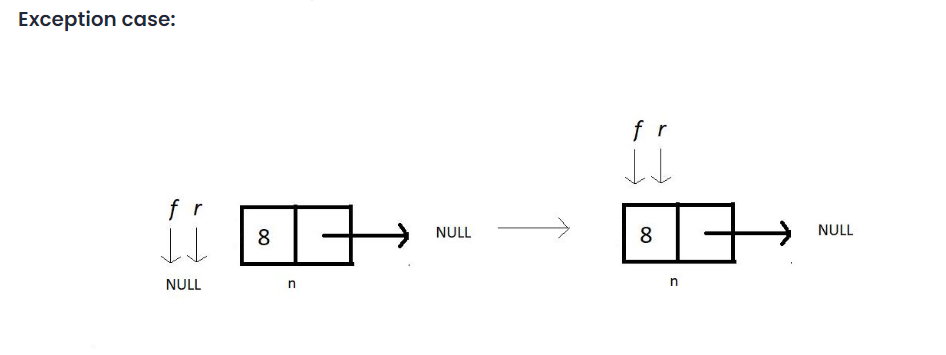
And “q” is one “variable” of “data type” “struct queue”

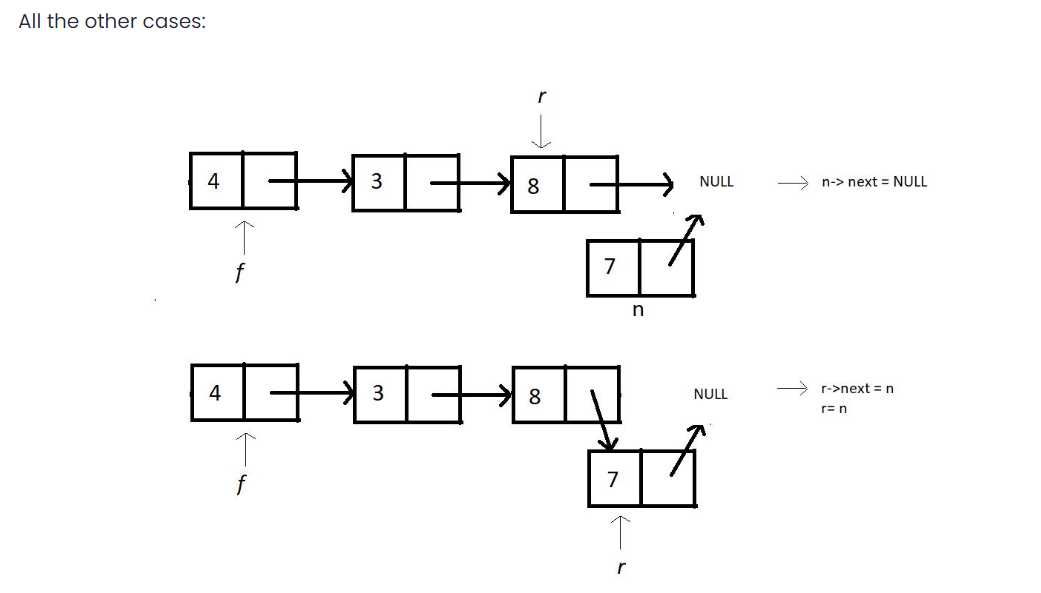
Which further has “many data types”

* CIRCULAR QUEUE
* #include<stdio.h>
* #include<stdlib.h>
* struct circularQueue
* {
* int size;
* int f;
* int r;
* int\* arr;
* };

* int isEmpty(struct circularQueue \*q)
* {
* if(q->r==q->f)
* /\*
* agar "q" ka "r"="q" ka "f" then the "array" is "empty"
* this is so because
* "r" is the "index manager" which manages the latest member of the queue
* that is the "member" recently added
* whereas
* "f" is the "index manager" which manages the oldest member of the queue
* that is the "member" to be removed
* and the "last member" and "first member" can only be same when there is nothing in queue
* \*/
* {
* return 1;
* }
* return 0;
* }
* int isFull(struct circularQueue \*q)
* {
* if((q->r+1)%q->size == q->f)
* /\*
* Earlier, "isFull" checked if our rear has reached the limit of the array
* if it did it was "full"
* but the queue wasn’t full technically
* as the "members" which left made "empty spaces" but
* "front index" kept moving forward
* and once "front index"="rear index" we thought it was "full"
* but wasn't
* \*/
* /\*
* the "front" coming next to the "rear" indicates that the queue is full
* in circular queue
* \*/
* /\*
* "Circular increment" lets us access the "queue indices" circularly, which means,
* after we finish visiting the "7th index" in the above illustration,
* we again come at the "zeroth index".
* \*/
* /\*
* "q->r+1%q->size" is an expression used to do "circular increment"
* \*/
* {
* return 1;
* }
* return 0;
* }
* void enqueue(struct circularQueue \*q, int val)
* {
* if(isFull(q))
* {
* printf("This Queue is full");
* }
* else
* {
* q->r = (q->r +1)%q->size;
* /\*
* here we are using the method of "circular increment"
* \*/
* q->arr[q->r] = val;
* printf("Enqued element: %d\n", val);
* }
* }
* int dequeue(struct circularQueue \*q)
* {
* int a;
* if(isEmpty(q))
* {
* printf("This Queue is empty");
* }
* else
* {
* q->f = (q->f +1)%q->size;
* /\*
* since the "f" holds the "index" of the "first element", we can just remove that
* we will access "f" using "circular increment"
* \*/
* /\*
* here again we are using the method of "circular increment"
* \*/
* a = q->arr[q->f];
* /\*
* "a"="q" ka "arr" ka "f"th index par jo "element" hai
* \*/
* }
* return a;
* }

* int main()
* {
* struct circularQueue q;
* q.size = 4;
* q.f = q.r = 0;
* q.arr = (int\*) malloc(q.size\*sizeof(int));
* // Enqueue few elements
* enqueue(&q, 12);
* enqueue(&q, 15);
* enqueue(&q, 1);
* /\*
* we cannot enter more than "3" elements in queue
* for "size"="4"
* because it will fail our "full" condition
* \*/
* printf("Dequeuing element %d\n", dequeue(&q));
* printf("Dequeuing element %d\n", dequeue(&q));
* printf("Dequeuing element %d\n", dequeue(&q));
* // we can "enqueue" again on "starting indexes" due to "circular increment"
* enqueue(&q, 45);
* enqueue(&q, 45);
* if(isEmpty(&q))
* {
* printf("Queue is empty\n");
* }
* if(isFull(&q))
* {
* printf("Queue is full\n");
* }
* return 0;
* }
* QUEUE USING LINKED LIST
* #include <stdio.h>
* #include <stdlib.h>
* struct Node \*f = NULL;
* struct Node \*r = NULL;
* struct Node
* {
* int data;
* struct Node \*next;
* };
* void linkedListTraversal(struct Node \*ptr)
* {
* printf("Printing the elements of this linked list\n");
* while (ptr != NULL)
* {
* printf("Element: %d\n", ptr->data);
* ptr = ptr->next;
* }
* }
* // we only use the rear pointer and add a new node at the end of the list.
* void enqueue(int val)
* {
* struct Node \*n = (struct Node \*)malloc(sizeof(struct Node));
* if (n == NULL)
* /\*
* if there is no space in the heap
* then only "n" will return "NULL"
* otherwise it will allocate memor for itself dynamically
* \*/
* {
* printf("Queue is Full");
* }
* else
* {
* n->data = val;
* // insert the val in the data member of n
* n->next = NULL;
* // make this node point to "NULL".
* if (f == NULL)
* /\*
* done for a "special case" which is only for the
* time when "first element" is inserted
* since "f" initially is totally "NULL" that is
* just not the "next part of f" is "NULL" but the whole "F"
* is "NULL" in that case..
* \*/
* {
* f = r = n;
* }
* /\*
* it is done so because "f" is the "index manager" for the
* "first element" to be added
* agar hum aisa nahi karenge toh
* "F" humesha ke liye "NULL" rehjayega
* \*/
* else
* {
* r->next = n;
* // done to connect the "2 nodes"
* r = n;
* }
* }
* }
* // Dequeue needs you to just delete the head node, which is the f node here.
* int dequeue()
* {
* int val = -1;
* struct Node \*ptr = f;
* /\*
* Create a "struct Node pointer ptr" to hold the node we will delete.
* thus Make "ptr" equal to "f".
* \*/
* if (f == NULL)
* {
* printf("Queue is Empty\n");
* }
* else
* {
* f = f->next;
* /\*
* as we know "f" is of "data type" "struct Node"
* i.e uska "next" part bhi hai and "data" part bhi hai
* toh hum yahan "f" ko "f" ke "next" ke equal kara rahe hai
* basically making "f" equal to the "next node" to f.
* \*/
* val = ptr->data;
* // Store the data of "ptr"(previous f) in an integer variable "val".
* free(ptr);
* }
* return val;
* }
* int main()
* {
* linkedListTraversal(f);
* printf("Dequeuing element %d\n", dequeue());
* enqueue(34);
* enqueue(4);
* enqueue(7);
* enqueue(17);
* printf("Dequeuing element %d\n", dequeue());
* printf("Dequeuing element %d\n", dequeue());
* printf("Dequeuing element %d\n", dequeue());
* printf("Dequeuing element %d\n", dequeue());
* linkedListTraversal(f);
* return 0;
* }





* DE-QUEUE

//code to be written

* SORTING

criteria for analyzing different sorting algorithms and why one differs from the other

1. Time Complexity

We observe the time complexity of an algorithm to see which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets. What if one sorting algorithm sorts only 4 elements efficiently and fails to sort 1000 elements. What if it takes too much time to sort a large data set? These are the cases where we say the time complexity of an algorithm is very poor.

In general, O(N log N) is considered a better algorithm time complexity than O(N2), and most of our algorithms’ time complexity revolves around these two.

1. Space Complexity

The space complexity criterion helps us compare the space the algorithm uses to sort any data set. If an algorithm consumes a lot of space for larger inputs, it is considered a poor algorithm for sorting large data sets. In some cases, we might prefer a higher space complexity algorithm if it proposes exceptionally low time complexity, but not in general.

And when we talk about space complexity, the term **in-place sorting algorithm**arises. The algorithm which results in constant space complexity is called an in-place sorting algorithm. Inplace sorting algorithms mostly use swapping and rearranging techniques to sort a data set. One example is Bubble Sort

1. Stability

The stability of an algorithm is judged by the fact whether the order of the elements having equal status when sorted on some basis is preserved or not. It probably sounded technical, but let me explain.

Suppose you have a set of numbers, 6, 1, 2, 7, 6, and we want to sort them in increasing order by using an algorithm. Then the result would be 1, 2, 6, 6, 7. But the key thing to look at is whether the 6s follow the same order as that given in the input or they have changed. That is, whether the first 6 still comes before the second 6 or not. If they do, then the algorithm we followed is called stable, otherwise unstable.

1. Internal & External Sorting Algorithms

When the algorithm loads the data set into the memory (RAM), we say the algorithm follows internal sorting methods. In contrast, we say it follows the external sorting methods when the data doesn’t get loaded into the memory.

1. Adaptivity

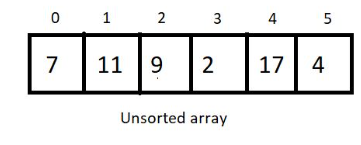
Algorithms that adapt to the fact that if the data are already sorted and it must take less time are called **adaptive algorithms**.  And algorithms which do not adapt to this situation are not adaptive.

1. Recursiveness

If the algorithm uses recursion to sort a data set, then it is called a recursive algorithm. Otherwise, non-recursive.

* BUBBLE SORT ALGORITHM

With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration.





Segment

Bubble sort intends to sort an array using (n-1) passes where n is the array's length.



In every iteration, segment will be sorted in such a way that the larger element will take place larger index

And when, the largest element of the current unsorted part reaches its final position

It completes  one pass

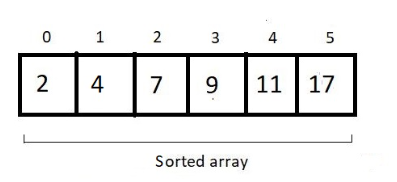
After every pass,

 unsorted part of the array reduces by 1, and the sorted part increases by 1.

At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array reaches its final position at the end.

Since our array is of length 6, we will make 5 passes.

After 5th pass,



For an array of length n, we would have (n-1) + (n-2) + (n-3) + (n-4) + . . . . . + 1 comparison and possible swaps.

the sum from 1 to n-1, which is n(n-1)/2, and hence our complexity of runtime becomes **O(n^2).**

we never made a swap when two elements of a pair become equal. Hence the algorithm is a **stable algorithm**

This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness.