Data is an important part of a program. So, when a program is dealing with the data, how it will organize the data in the main memory, that’s what is Data Structure.

Types of Data structures:

* **Physical Data Structures** – They define how the data is arranged in memory.
  + Array
  + Matrices
  + Linked List
* **Logical data structures –** These define how the data can be utilized, how the arrangement is and how it should be utilized.
  + Stack
  + Queues
  + Trees
  + Graph
  + Hashing

**ARRAYS**

Arrays are defined as a collection of similar data elements.

When a program is running. It runs inside the main memory. The main memory is divided into three sections – code section, stack, and heap. When an Array is created, that array will be created in the stack, and it will be directly accessible to the main function.

Syntax:

Int A[5];

Int B[5] = {2,4,6,8,10};

**STRUCTURES**

Structures can be defined as a collection of data members that are related data members under one name, and those data members may be of similar type, maybe of different types. So, usually it is defined as collection of similar data items under one name, that is grouping the data items.

Structure is used for defining user defined data types apart from the primitive types.

**Padding:** Adjustment in the memory of Structures is called as padding.

**POINTERS**

Pointer is an address variable that is meant for storing the address of data, not data itself.

Normal variables are data variables, but the pointers are address variables and are used for indirectly accessing the data.

**Why do we need to access the data indirectly or why we need pointers?**

Since, we know that the memory is divided into code section, stack, and heap. The stack can be directly accessed but the data in heap cannot be directly accessed. Therefore, to access that, we need pointers. It is not the only reason, but it is one of the reasons.

Next is, let’s say we have some files on our system, and we need to access those files in our program, then we need a pointer which is of FILE type through which we will access that file.

Pointers are also used for parameter passing.

When we write something like : printf(“%d”, \*p);, this is called as dereferencing.

#include<stdlib.h> is added when using malloc. Syntax is shown below:

*int* main(){

*int* \*p;

    p = (*int*\*) malloc(sizeof(*int*));

    // the malloc function returns a void pointer,

    //Therefore, we are typecasting it to int pointer.

}

Now this code above will allocate memory in heap. So, here this p will be pointing to the data stored in heap.

p = new *int*[10];

now, here we have declared different types of pointers:

*struct* Rectangle{

*int* length, width, height;

};

*int* main(){

*int* \*p1;

*char* \*p2;

*float* \*p3;

*double* \*p4;

*struct* Rectangle \*p5;

    printf("%d\n",sizeof(p1));

    printf("%d\n",sizeof(p2));

    printf("%d\n",sizeof(p3));

    printf("%d\n",sizeof(p4));

    printf("%d\n",sizeof(p5));

    return 0;

}

In each of them, the output is 4 bytes only. (in case of a 32 bit machine, the size of pointers will be 4 bytes and in case of 64 bit machine, it will be 8 bytes). **Therefore, it can be concluded that the size of a pointer is independent of its data.**

**Pointer to a structure**

#include<stdio.h>

*struct* Rectangle {

*int* length;

*int* breadth;

};

*int* main(){

*struct* Rectangle r ={10,5};

*struct* Rectangle\*p = &r;

    /\*

    if we want to change the value of length and breadth,

    we cannot do it like p.lenght = 15,

    instead we have to use \* before it

    and that too, \*p will in brackets like (\*P)

    \*/

   (\*p).length = 15;

   printf("%d\n", (\*p).length);

   /\*

   also, we can write it in another way using

   '->' instead

   \*/

  printf("%d\n", p->breadth);

  return 0;

}

**How to create an object dynamically in a heap using pointer.**

Here, we will create an object or variable of type rectangle dynamically.

#include<stdio.h>

#include<stdlib.h>

*struct* Rectangle {

*int* length;

*int* breadth;

};

*int* main(){

*struct* Rectangle \*p;

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

    p->length = 10;

    p->breadth = 5;

    printf("length: %d\n", p->length);

    printf("breadth: %d\n", p->breadth);

}

**FUNCTIONS**

Function is a piece of code which performs a specific task. The difference between function and structure is that a structure is a group of related data members whereas a function is a group of related instructions which perform a specific task.

#include<stdio.h>

*int* add(*int* *a*, *int* *b*){

*int* c;

    c=*a*+*b*;

    return c;

}

*int* main(){

*int* x,y,z;

    x=10;

    y=5;

    z=add(x,y);

    printf("sum is %d",z);

    return 0;

}

Parameter passing methods:

* Pass by value
* Pass by Address
* Pass by reference ( only in C++)

In pass by value, any changes done in formal parameters will not reflect in actual parameters.

In pass by address, any changes done in formal parameters will also reflect in actual parameters. **Call by Address uses Pointers.**

**Example:**

*void* swap(*int* \**x*,*int* \**y*){

*int* temp;

    temp=\**x*;

    \**x*=\**y*;

    \**y*=temp;

}

*int* main(){

*int* a,b;

    a=10;

    b=20;

    printf("before swapping a:%d, b:%d\n",a,b);

    swap(&a,&b);

    printf("a:%d,b:%d",a,b);

    return 0;

}

**Call by reference: (only supported in C++)**

*void* swap(*int* &*x*,*int* &*y*){

*int* temp;

    temp=x;

    x=y;

    y=temp;

}

*int* main(){

*int* a,b;

    a=10;

    b=20;

    swap(a,b);

    cout<<"a:"<<a<<"b:"<<b<<endl;

    return 0;

}

**Array as parameter:**

When passing Arrays as parameter, those can only be passed using pass by Address method.

*int* fun(*int* \**a*, *int* *n*){

*int* i;

    for (i=0; i<*n*; i++)

        printf("%d ",  *a*[i]);

}

*int* main(){

*int* a[5] = {2,4,6,8,10};

    fun(a,5);

    return 0;

}

There are two ways to pass an array as parameter. One is simply we can write a[] in function parameter. Other is by using pointer(\* as shown in the above program).

**How a function can return an array**

In order to return an array, we need to define the return type of function as array as shown in the below example:

#include<stdio.h>

#include<stdlib.h>

*int* \* func(*int* *n*){

*int* \*p;

    p = (*int* \*)malloc(*n*\*sizeof(*int*));

    for (*int* i = 0; i < *n*; i++){

        p[i] =i+1;

    }

    return p;

}

*int* main(){

*int* \*A;

    A = func(5);

    for( *int* i =0; i < 5; i++){

        printf("%d\n",A[i]);

    }

    return 0;

}

**Passing Structure as a parameter in a function:**

If we want to send a structure as parameter to some function, it may be call by value or call by address.

1. **Call by Value method:**

In call by value, a new object is created, and values are copied in that. Any changes in the values inside function will not change the values in main function.

#include<Stdio.h>

*struct* Rectangle{

*int* length;

*int* breadth;

};

*int* area(*struct* Rectangle *r1*){

    return *r1*.length\**r1*.breadth ;

}

*int* main(){

*struct* Rectangle r = {10,5};

    printf("Area: %d\n",area(r));

}

1. **Call by Address method**

*void* changelength(*struct* Rectangle \**p*, *int* *l*){

*p*->length = *l*;

}

*int* main(){

*struct* Rectangle r = {10,5};

    changelength(&r, 20);

    printf("\n%d", r.length);

    return 0;

}

**Note: we can pass a structure by value even if it is having an array inside it.**

**STRUCTURE AND FUNCTION (VERY IMPORTANT)**

here, in this program, everything is done through struct and functions. There is no logic implemented in code except calling of functions.

#include<stdio.h>

*struct* Rectangle{

*int* length;

*int* breadth;

};

*void* initialize(*struct* Rectangle \**r*, *int* *l*, *int* *b*){

*r*->length = *l*;

*r*->breadth = *b*;

}

*int* area(*struct* Rectangle *r1*){

    return *r1*.length\**r1*.breadth;

}

*void* changelength(*struct* Rectangle \**r*, *int* *l*){

*r*->length = *l*;

}

*int* main(){

*struct* Rectangle r;

    initialize(&r, 10,5);

    printf("%d\n",area(r));

    changelength(&r,20);

    printf("%d\n",area(r));

}

**Template Classes:**

C++ supports generic functions and generic classes. Generic functions are template functions, and classes are template classes.

A class which can be used for any type of data is called as a Generic Class.

Example:

#include<iostream>

using *namespace* std;

*template*<*class* T>

*class* Arithmetic{

*private:*

    T a;

    T b;

*public:*

    Arithmetic(T *a*, T *b*);

    T add();

    T sub();

};

*template*<*class* T>

    Arithmetic<T>::Arithmetic(T *a*, T *b*){

        this->a = *a*; // this is a pointer to a current object

        this->b = *b*;

    }

*template*<*class* T>

    T Arithmetic<T>::add(){

        T c;

        c=a+b;

        return c;

    }

*template*<*class* T>

    T Arithmetic<T>::sub(){

        T c;

        c=a-b;

        return c;

    }

*int* main(){

    Arithmetic<*int*> ar(10,5);

    cout<<"Add "<<ar.add()<<endl;

    cout<<"Subtract "<<ar.sub()<<endl;

    return 0;

}

Data Structures

We can say that a program is nothing but a set of instructions which perform operations on data to get some results. So, without data, there is no need of instruction, no use of instructions, then the term data, which we use in many places like **Data Structures, Databases, Data warehouse, big data.**

**Data Structure:** Data Structure can be defined as a arrangement of collection of data items so that they can be utilized efficiently, operations on that data can be done efficiently.

**Database:** When the data is larger in size, or commercial data that is used in businesses, like banks or retail stores or manufacturing farms, they will have a lot of data, and they will have some organized data in the form of database tables or relational data, and where they keep that relational data, all that data is stored on the disk. So, a database means arranging the data in some model like say a relational model in the permanent storage, so that it can be retrieved or to accessed by applications easily. That arrangement in the hard disk, or in the permanent storage is called as database.

**STATIC AND DYNAMIC MEMORY ALLOCATION**

Topics to be covered:

1. About main memory
2. How a program uses memory
3. Static allocation
4. Dynamic allocation

**Main memory**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 6 | 7 | 8 | 9 | 10 | 11 |
|  |  |  |  |  |  |

0 1 2 3 4 5

Suppose this block shows a memory, this is a memory. So, the memory is divided into smaller addressable unit that are called as bytes. So, memory is divided into bytes.

So, these boxes, these check boxes, let’s assume that these are bytes, and this is the entire memory. Every byte is having its address. If we see on the above table, 0,1,2,3,……….11……….., these are the addresses.

So, we are starting from bottom, one thing to observer is that diagram which is drawn as a 2-D but the addresses re single dimensional addresses, linear addresses. So, address will have just 1 value, not like coordinate system, (x,y), it will have a single value. So, addresses are linear.

**0-65535 = 65536 = 64 \* 1024 = 64 KB,** In this the first address will be 0 and the last address will be 65535.

**HOW HEAP MEMORY IS UTILIZED BY A PROGRAM:**

Heap means Piling up. If the things are kept one above another, or just randomly, we use the term Heap. Heap is used in two cases.

1. If the things are properly organised like a tower-like thing, then also it is a heap.
2. If it is not organised and, it’s looking like a tower, then also we call it a heap.

Programs cannot directly access the Heap memory. It can access anything inside code section, stack.

Program to show how we can get some memory inside the heap, with the help of a pointer.

Heap memory should be treated like a resource.

After malloc, we need t o de-allocate the memory and make the pointer NULL.

So to that, we have commands:

Delete []p;

P = NULL;

**PHYSICAL DATA STRUCTURES:**

There are mainly two types of physical data structures:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 8 | 3 | 5 | 9 |  |  |  |

1. Array

0 1 2 3 4 5 6

1. Linked List

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Head** | -> | 8 |  | -> | 3 |  | -> | 5 |  | -> | 9 | / |

These two data types decide how the memory is organized, how the memory is allocated.

**LOGICAL DATA STRUCTURES:**

The types of logical data structures are mentioned below:

1. **Linear Data Structures**
   1. Stack - **LIFO**
   2. Queues - **FIFO**
2. **Non-Linear Data Structures**
   1. Trees
   2. Graph
3. Hash Table – **Linear or Tabular Data Structure**

**ADT – ABSTRACT DATA TYPE**

A data type is defined in two terms:

1. Representation of Data
2. Operation on Data

Let’s say that if we have defined an integer in C/C++, then we will have 2 bytes ( let’s assume), then its representation will be that 1st bit will be the sign bit, and remaining 15 bits will store the data.

Abstract means hiding internal details. Now if we take the example of ‘int’, for performing operations on int, do we really need to know how they are performed in the binary form inside the main memory.

**TIME COMPLEXITY**

In daily life, when we do any work, any task, we want to know how much time it takes for that, performing that particular task.

O(n)- Order of n

If we are spending time on a tree along the height, then time complexity will be O(log n).

**SPACE COMPLEXITY**

We want to know how much space is consumed in main memory during the execution of program.

In case of array, n elements are there, then it will be O(n).

Linked list – n elements – O(n).

**RECURSION**

Topics to be covered:

1. What is recursion.
2. Example of recursion.
3. Tracing recursion.
4. Stack used in recursion.
5. Time complexity.
6. Recurrence relation.

**WHAT IS RECURSION:**

When a function is calling itself, it is called recursion.

Type fun(param){

If (<base condition>){

1. ……………………….
2. Fun(param);
3. ……………………….

}

}

There must be a base condition to terminate the recursion otherwise, it will go into infinite loop.

**How recursion uses Stack:**

We already know that the memory is divided into three sections: code section, Stack and Heap.

**How to find the time complexity for a recursive function:**

The very basic concept for finding hr time complexity is that we assume that every statement in program takes one unit of time for execution.

Let’s take the example of below written program:

*void* func1(*int* *n*){

    if(*n*>0){

        printf("%d\n",*n*);

        func1(*n*-1);

    }

}

*void* main(){

*int* x = 3;

    func1(x);

}

Here, our function is printing a value n, which will take one unit of time.

But, since it is a recursive function, it is calling itself again and again. so, to find out how many ties it is printing, we have to look into the tracing tree or recursion tree.

In our function, it is printing 3 units, therefore, we can say that it takes 3 units of time for the value of n as 3. Therefore, we can say that for n units, it will take n units of time. Therefore, time complexity of this will be O(n).

**There is one more way to find the time complexity by using the recurrence relation.**

**Explanation:**

*void* func1(*int* *n*){

    if(*n*>0){

        printf("%d\n",*n*);

        func1(*n*-1);

    }

}

*void* main(){

*int* x = 3;

    func1(x);

}

*using recurrence relation, we assume that the time taken by this function func1 is T(n), T is for time, then that total time should be sum of all the times taken by the statements inside. So, let’s look at the statements:*

*first is the if conditional statement, which will take one unit of time.*

*Then, there is the printf statement, which also takes one unit of time.*

*Then, there is function call, which should also take one unit of time. But this is wrong, since it is not just a normal statement, it will call again itself. So, there is something more behind this one. So, we need to know how much time that function call is taking.*

*Let us see closely when I said that this func(n) function call total time T(n), f(n-1) is also similar to that one only, so it will take T(n-1) time.*

*Therefore, total time is T(n) = T(n-1) +2.*

*Therefore, recurrence relation is:*

*T(n) = T(n-1) + 2, when n is greater than 0.*

*T(n) = 1, when n=0*

*So, the time complexity of recursive function can be represented in the form of recurrence relation, now if we solve this using induction method, or also called as successive substitution method, we can get the answer.*

*Before solving this, if we have any constant value in the function, we will write it as 1.*

*Therefore, our T(n) = T(n-1) + 1 ------------- eq(1)*

*T(n-1) = T(n-2) + 1*

*Which gives: T(n) = T(n-2) + 1 + 1 ---------------- eq(2)*

*.*

*.*

*.*

*.*

*.*

*. Let us continue this for ‘k’ times.*

*T(n) = T(n-k) + K*

*Assume that n-k = 0, that means n = k.*

*T(n) = T(n-n) + n*

*T(n) = T(0) + n*

*And we know that T(o) is 1, therefore, T(n) = n + 1, which is O(n).*

**STATIC AND GLOBAL VARIABLES IN RECURSION:**

Here, we will see how to treat the static variables inside Recursive function.

If there are any static variables in a recursive function, don’t show them in each step of tracing tree, write them just like global or outside variable and maintain a single copy of it.

Example:

#include <iostream>

#include <stdio.h>

using namespace std;

*int* fun(*int* *n*) {

    static *int* x = 0;

    if(*n* > 0) {

        x++;

        return fun(*n* - 1) + x;

    }

    return 0;

}

*int* main()

{

*int* x = 5;

    printf("%d",fun(x));

    return 0;

}

**Output – 25**

**Even if we use global variable instead of static variable, then also it will be treated same as static variable is treated.**

**TYPES OF RECURSIONS:**

1. Tail Recursion
2. Head recursion
3. Tree recursion
4. Indirect Recursion
5. Nested Recursion

**Tail Recursion:**

If a recursive function is calling itself, and that recursive call is the last statement in a function, then it is called as Tail recursion. After this call, there is nothing, it is not performing anything.

#include <iostream>

#include<stdio.h>

*void* fun(*int* *n*) {

    if (*n* > 0) {

        printf("%d", *n*);

        fun(*n* - 1);

    }

}

*int* main()

{

    fun(3);

    return 0;

}

Tail recursion means, that at return time, it doesn’t have to perform anything at all.

**Comparison of Tail recursion with Loop**

Every recursive function can be written as a loop or vice versa. Every Loop can also be converted in the form of a recursion.

The above program can also be written using while loop as shown below:

*void* fun(*int* *n*) {

    while (*n* > 0) {

        printf("%d", *n*);

*n*--;

    }

}

*int* main()

{

    fun(3);

    return 0;

}

Time complexity – Time taken by both is O(n)

When it comes to space complexity, in case of Recursive function, it internally used a stack. So, for the value of 3, it will create 4 activation records in the stack. Therefore, space taken by it is O(n).

In case of While loop, only one activation record will be created. Therefore, space complexity of this will be O(1).

**So, the conclusion is if you must write a Tail recursion, then better you convert it into a loop tat us more efficient in terms of space.**

**Head Recursion:**

If the first statement inside the function is the recursive call. Then it is a Head Recursion. There is no statement, no operation before the function call.

Basically, Head recursion means that the function doesn’t have to process or perform any operation at the time of calling. It has to do everything only at the time of returning.

*void* func1(*int* *n*){

    if(*n*>0){

        func1(*n*-1);

        printf("%d\n",*n*);

    }

}

*void* main(){

*int* x = 3;

    func1(x);

}

**Comparison of Head recursion and Loop:**

In case of Head recursion, or if a recursive function must do something at returning time, it cannot be easily converted in the form of a loop, but it can be converted. It doesn’t look as it is.

**Tree recursion:**

Before we go to Tree recursion, we should understand what linear recursion is. If a function is calling itself only one time, then it is a linear recursion.

When a function is calling itself more than one time, then it is called as Tree recursion.

Code:

*void* fun(*int* *n*)

{

 if(*n*>0)

 {

 printf("%d", *n*);

 fun(*n*-1);

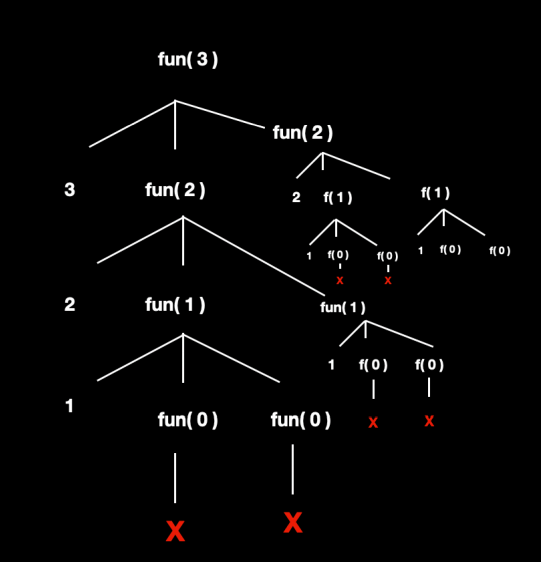
 fun(*n*-1);

 }

}

fun(3);

Explanation:

A white board with red writing

Description automatically generated

Ouptut: 3,2,1,1,2,1,1

Order in which the calls are made:

A diagram of a tree

Description automatically generated

Time complexity:

A close up of a white board

Description automatically generated

Space complexity depends on how much space in stack was utilized:

Therefore, we require the space equal to the height of the tree, i.e. O(n+1) 🡪 O(n)

**Indirect recursion**

In indirect recursion, there may be more than one function and they are calling one another in a circular fashion, So let’s say if the first function calls the second function, the second function calls the third function and the third function calls back the first function.

**Example:**

*void* funA(*int* *n*){

    if (*n*>0){

        printf("%d",*n*);

        funB(*n*-1);

    }

}

*void* funB(*int* *n*){

    if (*n*>1){

        print("%d",*n*);

        funA(*n*/2);

    }

}

Output: 20 19 9 8 4 3 1

**Nested recursion**

In a nested recursion, a recursive function will pass parameter as a recursive call.

Example:

*int* fun(*int* *n*) {

    if (*n* > 100)

        return *n* - 10;

    else

        return fun(fun(*n* + 11));

}

*int* main()

{

    printf("%d", fun(95));

    return 0;

}

Tracing:

Fun(95)

|

Fun(fun(95+11))

|

Fun(96)

|

Fun(fun(107))

|

Fun(97)

|

Fun(fun(108)

|

Fun(98)

|

Fun(fun(109))

|

Fun(99)

|

Fun(fun(110))

|

Fun(100)

|

Fun(fun(111))

|

Fun(101)

|

**It will return 91**

**Recursion for mn**

*int* exp(*int* *m*, *int* *n*) {

    if (*n* == 0) {

        return 1;

    }

    if (*n* % 2 == 0) {

        return pow(*m* \* *m*, *n* / 2);

    }

    else {

        return *m* \* pow(*m* \* *m*, (*n* - 1) / 2);

    }

}

*int* main()

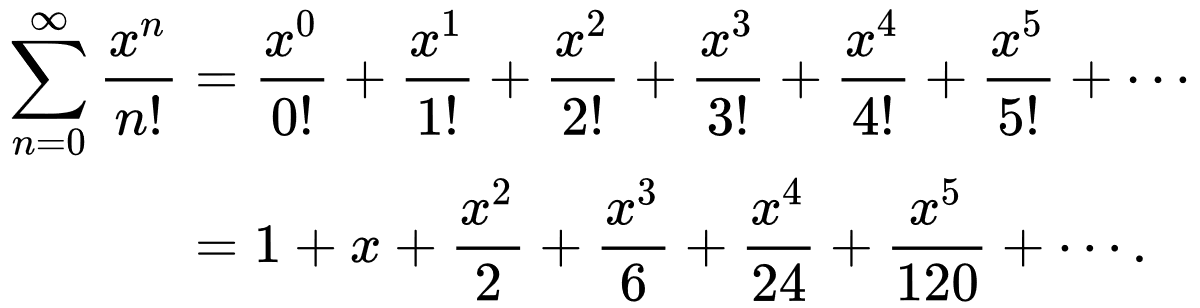
{

    printf("%d", exp(2,5));

    return 0;

}

**Taylor series ex printing**

****

For Taylor series, what we want is three things, sum, power as well as factorial. We take two static variables, P and F.

#include <iostream>

using namespace std;

*double* e(*int* *x*, *int* *n*) {

    static *double* P = 1, F = 1;

*double* r;

    if (*n* == 0) {

        return 1;

    }

    else {

        r = e(*x*, *n* - 1);

        P = P \* *x*;

        F = F \* *n*;

        return r + P / F;

    }

}

*int* main()

{

    printf("%lf \n", e(1, 10));

    return 0;

}

**Another solution to this problem:**

we can take commons, and by doing this we can reduce the number of multiplications.

**Method-1**

double e(double x, double n) {

double s = 1;

for (; n > 0; n--) {

s = 1 + x / n \* s;

}

return s;

}

int main()

{

printf("%lf \n", e(1, 10));

return 0;

}

**Method-2 : this is called Horner’s rule**

double e(double x, double n) {

static double s = 1;

if (n == 0) {

return s;

}

else {

s = 1 + x / n \* s;

return e(x, n - 1);

}

}

int main()

{

printf("%lf \n", e(1, 10));

return 0;

}

**Fibonacci series printing using recursion.**

When printing Fibonacci series using recursion, we need to take into consideration three conditions as mentioned below:

* f(n) = 0, if n = 0
* f(n) = 1, if n = 1
* f(n) = f(n-2) + f(n-1) if n>1

In case of iterative function, the time complexity is O(n).But in case of recursive function, it’s not the same.

When a function calls itself twice with the reduced value of n, then the time is O(2n). The cod for this time complexity is given below:

int fib(int n) {

if (n <= 1)

return n;

else {

return fib(n - 2) + fib(n - 1);

}

}

If we see in the code above, we can that the value of same function is called multiple times, for example, fib(3), fib(2), fib(1), fib(0), etc. This is excessive recursion.

We can make use of memorization technique to reduce the time complexity of this function. Memoization is a technique of storing the results of function call so that they can be called again.

Suppose, there is a global array F[10], which we will use to store the results of function call.

int F[10] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, };

int fib(int n) {

if (n <= 1) {

F[n] = n;

return n;

}

else {

if (F[n - 2] == -1)

F[n - 2] = fib(n - 2);

if (F[n - 1] == -1)

F[n - 1] = fib(n - 1);

return F[n - 2] + F[n - 1];

}

}

**Combination formula**

nCr = n!/(r!)(n-r)!

simple implementation:

int fact(int n) {

if (n == 0)

return 1;

else {

return n \* fact(n - 1);

}

}

int C(int n, int r) {

int t1, t2, t3;

t1 = fact(n);

t2 = fact(r);

t3 = fact(n - r);

return t1 / (t2 \* t3);

}

Recursive implementation for the same will be implemented as explained below:

In order to make a recursive function, we have to understand the pascal’s triangle. **Pascal’s triangle says that the values of nCr can be obtained by performing the addition recursively.**

A white paper with red writing on it

Description automatically generated

nCr

n-1Cr

n-1Cr-1

int C(int n, int r) {

if (r == 0 || n == r) {

return 1;

}

else {

return C(n - 1, r - 1) + C(n - 1, r);

}

}

**Tower of Hanoi Problem**

void TOH(int n, int A, int B, int C) {

if (n > 0) {

TOH(n - 1, A, C, B);

printf("(%d, %d)\n", A, C);

TOH(n - 1, B, A, C);

}

}

**Array**

Static array – Inside Stack

Dynamic array – Inside Heap

Deallocation – in case of C++, it is done using delete. In case of C, it is done using “free”

Once an array is created, it’s size cannot be changed.

If at all, we want to increase the size of array, it is possible in another way, but it is only possible in heap. Same array cannot be resized, but we have some alternative for that.

**Demonstration of creation of Array in stack and Heap:**

int main()

{

int A[5] = {2,4,6,8,10};

int\* p;

int i;

p = (int\*)malloc(5 \* sizeof(int));

p[0] = 3;

p[1] = 5;

p[2] = 7;

p[3] = 9;

p[4] = 11;

for (i = 0; i < 5; i++)

printf("%d ", A[i]);

printf("\n");

for (i = 0; i < 5; i++)

printf("%d ", p[i]);

return 0;

}

**How to increase the Array size:**

**Method -1 :**

* An array is created in stack, so in order to increase the array size we can use another pointer of larger size then point it to the array this will transfer all the elements to the new array
* After allotting the array to the new pointer it a must to delete the previous pointer so that there is no memory leakage.
* The command used to delete is delete[ A black and white image of a number

  Description automatically generated with medium confidence

Code for increasing the size of array:

int main() {

int \*p, \*q;

int i;

p = (int\*)malloc(5 \* sizeof(int));

p[0] = 3;

p[1] = 5;

p[2] = 7;

p[3] = 9;

p[4] = 11;

q = (int\*)malloc(10 \* sizeof(int));

for (i = 0; i < 5; i++) {

q[i] = p[i];

}

free(p);

p = q;

q = NULL;

for (i = 0; i < 5; i++) {

printf("%d ", q[i]);

}

return 0;

}

**2D Array**

There are three methods of creating a 2D-array.

1. We can access 2S array using 2 indices (one for row and other for column)
2. Array of pointers
3. Double pointer

**Normal method:**

int main()

{

int a[3][4] = { {1,2,3,4},{2,4,6,8},{3,5,7,9} };

return 0;

}

**Array of pointers:**

A diagram of a crossword puzzle

Description automatically generated

Here the array of pointer will be in stack and the actual array will be in the heap.

int main() {

int\* a[3]; // this array will be present in the stack

a[0] = new int[4];// this will be stored in heap

a[1] = new int[4];// this will be stored in heap

a[2] = new int[4];// this will be stored in heap

return 0;

}

**Third method – Double pointer**

Here, almost everything will be in heap.

int main() {

int\*\* a;

a = new int\* [3];

a[0] = new int[4];

a[1] = new int[4];

a[2] = new int[4];

return 0;

}

**How a compiler handles 1D Array**

In our programs, we use variables as names for representing some data. But when compiler converts into machine code, machine code will not have variable names.

Any address in an array can be accessed with the help of base address.

**The formula use by the compiler to convert it is:**

**Add(A[i]) = Lo + i\*w**

**Where:**

**Lo – Base address**

**I – Index**

**W – Size of data type**

**How a compiler handles a 2D array**

1. Row major formula

Add(A[i][j]) = Lo + [i\*n + j]\*w

Where:

Lo – base address

I – row

J – column

N – total number of column

W – size of data type

1. Column major formula

**Add(A[i][j]) = Lo + [j\*m + i]\*w - Column major formula**

**Add(A[i][j]) = Lo + [i\*n + j]\*w - Row major formula**

**How a compiler handles a 4D array**

1. Row major formula

Add (A[ i1 ] [ i2 ][ i3 ] [ i4 ] ) = L0 + [ i1 \* d2\* d3 \* d4 + i2 \* d3 \* d4 + i3 \*d4 +i4 ] \* w

1. Column major formula

Add (A[ i1 ] [ i2 ][ i3 ] [ i4 ] ) = L0 + [ i4 \* d1 \* d2 \* d3 + i3 \* d1 \* d2 + i2 \*d1+i1 ] \* w

**General formula for n dimensional array**

1. Row major

L0 + SUM[(ip, p from 1 to n) + Product(dq, q from p+1 to n)] \*w

We can see that in the above formula, the time complexity is 0(n2). Let’s see if we can reduce it:

Let’s take the formula:

Add (A [ i1] [ i2] [ i3] [ i4]) = L0 + [ i1 \* d2\* d3 \* d4 + i2 \* d3 \* d4 + i3 \*d4 +i4 ] \* w

It can also be written as: i4 + d4\*[i3 + d3\*[i2 + d2\*i1]], and this formula will reduce the time complexity to 0(n). **This rule which we have applied to reduce the number of multiplications is Horner’s rule.**

**Row major - left to right**

**Column major – Right to left**

**ARRAY ADT**

Array ADT means, Array as Abstract Data type. Abstract Data Type means representation of data and the set of operations on the data.

The Data Structure Array and the set of operations on it may be called as Array ADT.

**Data:**

1. Array Space
2. Size
3. Length (No. of elements)

**Operations:**

1. Display()
2. Add(x)/Append(x)
3. Insert ( Index x)
4. Delete(Index)
5. Search(x)
6. Get(Index)
7. Set(index , x)
8. Max()/ Min()
9. Reverse()
10. Shift()/ Rotate(), and many more

**Display()**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

struct Array {

int A[20];

//int\* A;

int size;

int length;

};

void Display(struct Array arr) {

int i;

printf("\nElemnets are \n");

for (i = 0; i < arr.length; i++)

printf("%d ", arr.A[i]);

}

int main()

{

struct Array arr = { {2,3,4,5,6}, 20, 5 };

//struct Array arr;

/\* int n, i; \*/// for asking how many numbers we want to enter and the taking them as input

/\*printf("Enyter size of an array: ");

scanf("%d", &arr.size);

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

arr.length = 0;

printf("Enter the count of numbers you want to enter: ");

scanf("%d", &n);

printf("\nEnter all elements:\n");

for (i = 0; i < n; i++)

scanf("%d", &arr.A[i]);

arr.length = n;\*/

Display(arr);

return 0;

}

**Functions in an Array:**

Let’s consider an array as an example:

Size = 10

Length = 6

A number grid with numbers

Description automatically generated with medium confidence

**Add ()/Append ()** is adding a new element at the end of array.

**--------------------------------------------------------------------------------------------------------------------------------------**

**Insert(Index, x) :**

For(i=length; i>index; i--)

{

A[i] = A[i-1]

}

A[index] = x

Length++

Since we are adding/appending the array, it should be a call by address type.

Code:

void Append(struct Array \*arr, int x) {

if (arr->length < arr->size)

arr->A[arr->length++] = x;

}

void Insert(struct Array\* arr, int index, int x) {

int i;

if (index >= 0 && index <= arr->length) {

for (i = arr->length; i > index; i--) {

arr -> A[i] = arr->A[i - 1];

}

arr->A[index] = x;

arr->length++;

}

}

**Delete(index) operation:**

int Delete(struct Array \*arr, int index) {

int x = 0;

int i;

if (index >= 0 && index < arr->length) {

x = arr->A[index];

for (i = index; i < arr->length - 1; i++)

arr->A[i] = arr->A[i + 1];

arr->length--;

return x;

}

return 0;

}

**Linear Search in Array:**

Linear search is a method of finding an element of an array and returning the index at which the element is found. In case the element is not found, -1 is returned.

Start from 0 go all the way to length of array and then return.

Time complexity:

Best – O(n)

Worst – O(n)

Average – O(n)

Although, there are a few ways to improve the Linear search which are mentioned below:

1. Transposition – move the element to one index further.
2. Move to Front/ Head – move the found element to index 0.

**Method-1:**

int LinearSearch(struct Array arr, int key) {

int i;

for (i = 0; i < arr.length; i++) {

if (key == arr.A[i])

return i;

}

return -1;

}

**Method-2: Transposition**

int LinearSearch(struct Array \*arr, int key) {

int i;

for (i = 0; i < arr->length; i++) {

if (key == arr->A[i]) {

swap(&arr->A[i], &arr->A[i - 1]);

return i;

}

}

return -1;

}

**Method-3: Move to Header**

int LinearSearch(struct Array \*arr, int key) {

int i;

for (i = 0; i < arr->length; i++) {

if (key == arr->A[i]) {

swap(&arr->A[i], &arr->A[0]);

return i ;

}

}

return -1;

}

**Binary Search in Array**

For performing the binary search, the pre-requisite is that the array must be sorted.

A number grid with numbers

Description automatically generated with medium confidence

It is called binary search because it will always search the key element in the middle of the sorted array.

For performing the binary search, we need three index variables: lower(l), higher(h), and mid(m).

Mid is floor value of (l+h)/2.

If the found value is greater than the key value, we will assign mid-value+1 to lower. If found value is smaller than the key value, we will assign mid-1 to higher.

**Code:**

**Method – 1:**

int BinarySearch(struct Array arr, int key) {

int l, mid, h;

l = 0;

h = arr.length - 1;

while (l <= h) {

mid = (l + h) / 2;

if (key == arr.A[mid])

return mid;

else if (key < arr.A[mid])

h = mid - 1;

else

l = mid + 1;

}

return -1;

}

**Method – 2: Recursion**

int RBinarySearch(struct Array arr, int l, int h, int key) {

int mid;

if (l <= h) {

mid = (l + h) / 2;

if (key == arr.A[mid])

return mid;

else if (key < arr.A[mid]) {

return RBinarySearch(arr, l, mid - 1, key);

}

else

return RBinarySearch(arr, mid + 1, h, key);

}

return -1;

}

**Tracing Tree for Binary Search:**

**A diagram of a structure

Description automatically generated**

**Time complexity in case of Binary Search:**

Best case – O(1)

Worst Case – O(log(n))

**Average case analysis of Binary Search:**

Average case means totals time taken by all cases divided by total number of cases.

It is also log(n) only.

E=I+2N

No of external nodes: e=i+1

**Merging of Two arrays:**

Here the scenario is that two arrays arr1 and arr2 have elements and we have to merge those two arrays and store in a third array arr3.

**Function:**

struct Array\* merge(struct Array\* arr1, struct Array\* arr2) {

int i, j, k;

i = j = k = 0;

struct Array\* arr3 = (struct Array\*)malloc(sizeof(struct Array));

while (i < arr1->length && j < arr2->length) {

if (arr1->A[i] < arr2->A[j])

arr3->A[k++] = arr1->A[i++];

else

arr3->A[k++] = arr2->A[j++];

}

for (; i < arr1->length; i++)

arr3->A[k++] = arr1->A[i];

for (; j < arr2->length; j++)

arr3->A[k++] = arr2->A[j];

arr3->length = arr1->length + arr2->length;

arr3->size = 10;

return arr3;

}

To display the arr3, we have to call the display function as shown below:

Display(\*arr3);

**Set operations on Arrays:**

When performing the set operations on Array, our methods change depending upon whether the arrays arguments are sorted or unsorted.

Let us say we have two arrays, A and B and A has m elements, B has n elements.

**UNION**

1. **When the arrays are unsorted**

When the arrays are not sorted, what we do is that we copy the first array to a new array C, and then before blindly copying the elements from array B, we first compare it with elements in array C, if it is there, leave the element, if it is not there, add the element to array C.

The time complexity in this case is O(n2).

1. **When the arrays are sorted**

When the arrays are sorted, we use the similar concept as merging, except when the elements of both A and B are same, we copy only once and increment both I and j.

Time complexity in this case will be O(m+n) 🡪 O(n)

**Code:**

struct Array\* Union(struct Array\* arr1, struct Array\* arr2) {

int i, j, k;

i = j = k = 0;

struct Array\* arr3 = (struct Array\*)malloc(sizeof(struct Array));

while (i < arr1->length && j < arr2->length) {

if (arr1->A[i] < arr2->A[j])

arr3->A[k++] = arr1->A[i++];

else if (arr2->A[j] < arr1->A[i])

arr3->A[k++] = arr2->A[j++];

else {

arr3->A[k++] = arr1->A[i++];

j++;

}

}

for (; i < arr1->length; i++)

arr3->A[k++] = arr1->A[i];

for (; j < arr2->length; j++)

arr3->A[k++] = arr2->A[j];

arr3->length = k;

arr3->size = 10;

return arr3;

}

**INTERSECTION**

1. **When the arrays are unsorted**

The procedure is start checking the elements of array A, copy them in C but before copying, we need to check if this element is already present in array B. If present, add the element in array C, if not increment the element in array A.

The time complexity in this case is O(n2).

1. **When the arrays are sorted**

We start with I,j,k = 0, compare the elements, if the element of one array is smaller, increment it, if element value is same in both the arrays, then copy that value to array C, increment array C.

**Code:**

struct Array\* Intersection(struct Array\* arr1, struct Array\* arr2) {

int i, j, k;

i = j = k = 0;

struct Array\* arr3 = (struct Array\*)malloc(sizeof(struct Array));

while (i < arr1->length && j < arr2->length) {

if (arr1->A[i] < arr2->A[j])

i++;

else if (arr2->A[j] < arr1->A[i])

j++;

else {

arr3->A[k++] = arr1->A[i++];

j++;

}

}

arr3->length = k;

arr3->size = 10;

return arr3;

}

**DIFFERENCE:**

**Code:**

struct Array\* Difference(struct Array\* arr1, struct Array\* arr2) {

int i, j, k;

i = j = k = 0;

struct Array\* arr3 = (struct Array\*)malloc(sizeof(struct Array));

while (i < arr1->length && j < arr2->length) {

if (arr1->A[i] < arr2->A[j])

arr3->A[k++] = arr1->A[i++];

else if (arr2->A[j] < arr1->A[i])

j++;

else {

i++;

j++;

}

}

for (; i < arr1->length; i++)

arr3->A[k++] = arr1->A[i];

arr3->length = k;

arr3->size = 10;

return arr3;

}

**C++ Program for MENU DRIVEN PROGRAM FOR ARRAY**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include <stdio.h>

#include<stdlib.h>

using namespace std;

class Array

{

private:

int\* A;

int size;

int length;

void swap(int\* x, int\* y);

public:

Array()

{

size = 10;

length = 0;

A = new int[size];

}

Array(int sz)

{

size = sz;

length = 0;

A = new int[size];

}

~Array()

{

delete []A;

}

void Display();

void Append(int x);

void Insert(int index, int x);

int Delete(int index);

int LinearSearch(int key);

int BinarySearch(int key);

//int RBinSearch(int a[], int l, int h, int key);

int Get(int index);

void Set(int index, int x);

int Max();

int Min();

int Sum();

float Avg();

void Reverse();

void Reverse2();

void ArrayReverse();

void InsertSort(int x);

int isSorted();

void Rearrange();

Array\* Merge(Array \*arr2);

Array\* Union(Array \*arr2);

Array\* Intersection(Array \*arr2);

Array\* Difference(Array \*arr2);

};

void Array::Display()

{

int i;

printf("\nElements are\n");

for (i = 0; i < length; i++)

printf("%d ", A[i]);

}

void Array::Append(int x)

{

if (length < size)

A[length++] = x;

}

void Array::Insert(int index, int x)

{

int i;

if (index >= 0 && index <= length)

{

for (i = length; i > index; i--)

A[i] =A[i - 1];

A[index] = x;

length++;

}

}

int Array::Delete(int index)

{

int x = 0;

int i;

if (index >= 0 && index < length)

{

x = A[index];

for (i = index; i < length - 1; i++)

A[i] = A[i + 1];

length--;

return x;

}

return 0;

}

void Array::swap(int\* x, int\* y)

{

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

int Array::LinearSearch(int key)

{

int i;

for (i = 0; i < length; i++)

{

if (key == A[i])

{

swap(&A[i], &A[0]);

return i;

}

}

return -1;

}

int Array::BinarySearch(int key)

{

int l, mid, h;

l = 0;

h = length - 1;

while (l <= h)

{

mid = (l + h) / 2;

if (key == A[mid])

return mid;

else if (key < A[mid])

h = mid - 1;

else

l = mid + 1;

}

return -1;

}

/\*

int RBinSearch(int a[], int l, int h, int key)

{

int mid;

if (l <= h)

{

mid = (l + h) / 2;

if (key == a[mid])

return mid;

else if (key < a[mid])

return RBinSearch(a, l, mid - 1, key);

else

return RBinSearch(a, mid + 1, h, key);

}

return -1;

}

\*/

int Array::Get(int index)

{

if (index >= 0 && index < length)

return A[index];

return -1;

}

void Array::Set(int index, int x)

{

if (index >= 0 && index < length)

A[index] = x;

}

int Array::Max()

{

int max = A[0];

int i;

for (i = 1; i < length; i++)

{

if (A[i] > max)

max = A[i];

}

return max;

}

int Array::Min()

{

int min = A[0];

int i;

for (i = 1; i < length; i++)

{

if (A[i] < min)

min = A[i];

}

return min;

}

int Array::Sum()

{

int s = 0;

int i;

for (i = 0; i < length; i++)

s += A[i];

return s;

}

float Array::Avg()

{

return (float)Sum() / length;

}

void Array::ArrayReverse()

{

int\* B;

int i, j;

B = (int\*)malloc(length \* sizeof(int));

for (i = length - 1, j = 0; i >= 0; i--, j++)

B[j] = A[i];

for (i = 0; i < length; i++)

A[i] = B[i];

}

void Array::Reverse2()

{

int i, j;

for (i = 0, j = length - 1; i < j; i++, j--)

{

swap(&A[i], &A[j]);

}

}

void Array::InsertSort(int x)

{

int i = length - 1;

if (length == size)

return;

while (i >= 0 && A[i] > x)

{

A[i + 1] = A[i];

i--;

}

A[i + 1] = x;

length++;

}

int Array::isSorted()

{

int i;

for (i = 0; i < length - 1; i++)

{

if (A[i] > A[i + 1])

return 0;

}

return 1;

}

void Array::Rearrange()

{

int i, j;

i = 0;

j = length - 1;

while (i < j)

{

while (A[i] < 0)i++;

while (A[j] >= 0)j--;

if (i < j)swap(&A[i], &A[j]);

}

}

//Array\* Merge(Array arr2)

//{

// int i, j, k;

// i = j = k = 0;

//

// Array\* arr3 = new Array(length + arr2.length);

//

// while (i < length && j < arr2.length)

// {

// if (A[i] < arr2.A[j])

// arr3->A[k++] = A[i++];

// else

// arr3->A[k++] = arr2.A[j++];

// }

// for (; i < length; i++)

// arr3->A[k++] = A[i];

// for (; j < arr2.length; j++)

// arr3->A[k++] = arr2.A[j];

// arr3->length = length + arr2.length;

//

// return arr3;

//}

// Array\* Union( Array arr2)

//{

// int i, j, k;

// i = j = k = 0;

//

// Array\* arr3 = new Array(length+arr2.length)

//

// while (i < length && j < arr2.length)

// {

// if (A[i] < arr2.A[j])

// arr3->A[k++] = A[i++];

// else if (arr2.A[j] < A[i])

// arr3->A[k++] = arr2.A[j++];

// else

// {

// arr3->A[k++] = A[i++];

// j++;

// }

// }

// for (; i < length; i++)

// arr3->A[k++] = A[i];

// for (; j < arr2.length; j++)

// arr3->A[k++] = arr2.A[j];

//

// arr3->length = k;

//

//

// return arr3;

//}

//Array\* Intersection(Array arr2)

//{

// int i, j, k;

// i = j = k = 0;

//

// Array\* arr3 = new Array(length + arr2.length);

//

// while (i < length && j < arr2.length)

// {

// if (A[i] < arr2.A[j])

// i++;

// else if (arr2.A[j] < A[i])

// j++;

// else if (A[i] == arr2.A[j])

// {

// arr3->A[k++] = A[i++];

// j++;

// }

// }

//

// arr3->length = k;

// return arr3;

//}

//Array:: Array\* Difference(Array arr2)

//{

// int i, j, k;

// i = j = k = 0;

//

// Array\* arr3 = new Array(length + arr2.length);

//

// while (i < length && j < arr2.length)

// {

// if (A[i] < arr2.A[j])

// arr3->A[k++] = A[i++];

// else if (arr2.A[j] < A[i])

// j++;

// else

// {

// i++;

// j++;

// }

// }

// for (; i < length; i++)

// arr3->A[k++] = A[i];

//

// arr3->length = k;

//

// return arr3;

//}

int main()

{

Array \*arr1;

int ch,sz;

int x, index;

cout<<"Enter Size of Array";

cin>>sz;

arr1 = new Array(sz);

do

{

cout<<"\n\nMenu\n";

cout<<"1. Insert\n";

cout<<"2. Delete\n";

cout<<"3. Search\n";

cout<<"4. Sum\n";

cout<<"5. Display\n";

cout<<"6.Exit\n";

cout<<"enter you choice ";

cin>>ch;

switch (ch)

{

case 1: cout<<"Enter an element and index";

cin >> x >> index;

arr1->Insert(index, x);

break;

case 2: cout<<"Enter index ";

cin >> index;

x = arr1->Delete(index);

cout<<"Deleted Element is %d\n"<<x;

break;

case 3:cout<<"Enter element to search ";

cin >> x;

index = arr1->LinearSearch(x);

cout<<"Element index %d"<<index;

break;

case 4:cout<<"Sum is %d\n"<<arr1->Sum();

break;

case 5:arr1->Display();

}

} while (ch < 6);

return 0;

}

**TEMPLATE CLASS**

// We are making this program as generic to take values of any kind.

#include <iostream>

using namespace std;

template<class T>

class Array {

private:

T\* A;

int size;

int length;

public:

Array() { //non-parameterized constructor

size = 10;

A = new T[10];

length = 0;

}

Array(int sz) { // paratemetrizecd constructor

size = sz;

length = 0;

A = new T[size];

}

~Array() { // destructor

delete[]A;

}

void Display();

void insert(int index, T x);

T Delete(int index);

};

template <class T>

void Array<T>::Display() {

for (int i = 0; i < length; i++) {

cout << A[i] << " ";

}

cout << endl;

}

template<class T>

void Array<T>::insert(int index,T x) {

if (index >= 0 && index <= length) {

for (int i = length - 1; i >= index; i--) {

A[i + 1] = A[i];

}

A[index] = x;

length++;

}

}

template<class T>

T Array<T>::Delete(int index) {

int x = 0;

if (index >= 0 && index < length) {

x = A[index];

for (int i = index; i < length - 1; i++) {

A[i] = A[i + 1];

}

length--;

}

return x;

}

int main()

{

Array<int> arr(10);

arr.insert(0, 5);

arr.insert(1, 6);

arr.insert(2, 9);

arr.Display();

cout << arr.Delete(1)<<endl;

arr.Display();

}

**Challenge -1: Find single missing number in an Array**

**Method – 1:** We know that the sum of first n natural numbers is n(n+1)/2. So we will do the sum of all elements in the array and subtract it from the sum of those first ‘n’ natural numbers and hence, we will get the missing single element.

**When the elements are not first n natural numbers:**

Let’s say the starting number is 6 and ending number is 17 in an array.

We need to have below information:

‘l’ = lower number

‘h’ = last number

‘n’ = number of elements

Here, we can do it by checking the difference between the number and its index.

diff = l - 0;

for (i = 0; i < n; i++) {

if (A[i] - i != diff) {

printf("missing element: %d\n", i + diff);

break;

}

}

**Challenge – 2 : Find multiple missing elements in a sorted Array**

diff = l - 0;

for (i = 0; i < n; i++) {

if (A[i] - i != diff) {

while (diff < A[i] - i) {

printf("%d\n", i + diff);

diff++:

}

}

}

**Challenge – 3 : Find multiple missing elements in an unsorted Array**

for (i = 0; i < n; i++) {

H[A[i]]++;

}

for (i = l; i <= h; i++) {

if (H[i] == 0)

printf("%d\n", i);

}

**Challenge – 3: Find duplicate elements in a sorted Array**

Iterate over the array and compare the ith ‑element to i+1th element for equality

**Count the number of duplicate elements in an array:**

for (i = 0; i < n; i++) {

if (A[i] == A[i + 1]) {

j = i + 1;

while (A[j] == A[i]) j++;

printf("%d is apprearing %d times\n", A[i], j - i);

i = j - 1;

}

}

**Finding duplicates in a sorted array using Hashing**

for (i = 0; i < n; i++) {

H[A[i]]++;

}

for (i = 0; i < +max; i++) {

if (H[i] > 0)

printf(:"%d is getting repeated %d times\n", i, H[i]);

}

**Finding duplicates in an unsorted array**

int main()

{

for (i = 0; i < n - 1; i++) {

int count = 1;

if (A[i] != -1) {

for (j = i + 1; j < n) {

if (A[i] == A[j]) {

count++;

A[j] = -1;

}

}

if (count > 1) {

printf(" % d is present % d times\n", A[i], count);

}

}

}

}

To perform it using hashing, process is same as in case of sorted array.

**STRINGS**

ASCII – American standard code for information interchange

A-Z : 65-90

a-z : 97-122

0-9 : 48-57

Enter – 10

Space – 13

Esc – 27

**Program to find if two strings are anagrams of each other or not.**

int main()

{

char ch[] = "decimal";

char ch1[] = "medical";

int i, H[26] = { 0 };

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

H[ch[i] - 97] += 1;

for (i = 0; ch1[i] != '\0'; i++) {

H[ch1[i] - 97] -= 1;

if (H[ch1[i] - 97] < 0) {

printf("the two strings are not anagrams of eachother!!.." );

break;

}

}

if (ch1[i] == '\0')

printf("Anagram");

}

**Permutations of a string:**

If a string “ABC” is given, then permutation of that string means finding all the arrangements of string.

If a string contains “n” alphabets, then the total number of possible permutations are “n!”.

The tree representing all the possible permutations is called as “State Space Tree”.

Back Tracking

From this String, what all permutations are possible, it is shown in the tree, everything is explored here. So, this is called as brute force.

**Code:**

void perm(char s[], int k) {

static int A[10] = { 0 };

static char Res[10];

int i;

if (s[k] == '\0') {

Res[k] = '\0';

printf("%s\n", Res);

}

else {

for (i = 0; s[i] != '\0'; i++) {

if (A[i] == 0) {

Res[k] = s[i];

A[i] = 1;

perm(s, k + 1);

A[i] = 0;

}

}

}

}

int main()

{

char ch[] = "abc";

perm(ch, 0);

}

**Another method:**

void perm1(char s[], int l, int h) {

int i;

if (l == h)

printf("%s ", s);

else {

for (i = l; i <= h; i++) {

swap(s[l], s[i]);

perm1(s, l + 1, h);

swap(s[l], s[i]);

}

}

}

**MATRICES**

**Diagonal Matrix:**

#include<stdio.h>

#include<iostream>

using namespace std;

struct Matrix

{

int A[10];

int n;

};

void Set(struct Matrix\* m, int i, int j, int x)

{

if (i == j)

m->A[i - 1] = x;

}

int Get(struct Matrix m, int i, int j)

{

if (i == j)

return m.A[i - 1];

else

return 0;

}

void Display(struct Matrix m)

{

int i, j;

for (i = 0; i < m.n; i++)

{

for (j = 0; j < m.n; j++)

{

if (i == j)

printf("%d ", m.A[i]);

else

printf("0 ");

}

printf("\n");

}

}

int main()

{

struct Matrix m;

m.n = 4;

Set(&m, 1, 1, 5); Set(&m, 2, 2, 8); Set(&m, 3, 3, 9); Set(&m,4, 4, 12);

printf("%d \n", Get(m, 2, 2));

Display(m);

return 0;

}

**C++ Class for Diagonal matrix:**

class diagonal {

private:

int n;

int\* A;

public:

diagonal() {

n = 2;

A = new int[2];

}

diagonal(int n) {

this->n = n;

A = new int[n];

}

~diagonal() {

delete []A;

}

void set(int i, int j, int x);

int get(int i, int j);

void Display();

};

void diagonal::set(int i, int j, int x) {

if (i == j)

A[i - 1] = x;

}

int diagonal::get(int i, int j) {

if (i == j)

return A[i - 1];

else

return 0;

}

void diagonal::Display() {

int i, j;

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {

if (i == j)

cout << A[i]<<" ";

else

cout << "0 ";

}

cout << endl;

}

}

int main() {

diagonal d(4);

d.set(1, 1, 5);

d.set(2, 2, 10);

d.set(3, 3, 15);

d.set(4, 4, 20);

d.Display();

return 0;

}

**Lower Trianglar Matrix**

**Program for both row-major and column-major representation of lower triangular matrix.**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

struct matrix {

int\* A;

int n;

};

// ################ ROW-MAJOR #################################

//void set(struct matrix\* m, int i, int j, int x) {

// if (i >= j) {

// m->a[i \* (i - 1) / 2 + j - 1] = x;

//

// }

//}

//

//int get(struct matrix m, int i, int j) {

// if (i >= j)

// return m.a[i \* (i - 1) / 2 + j - 1];

// else

// return 0;

//}

//

//void display(struct matrix m) {

// int i, j;

// for (i = 1; i <= m.n; i++) {

// for (j = 1; j <= m.n; j++) {

// if (i >= j)

// printf("%d ", m.a[i \* (i - 1) / 2 + j - 1]);

// else

// printf("0 ");

// }

// printf("\n");

// }

//}

// ########################## COLUMN-MAJOR #######################

void set(struct matrix\* m, int i, int j, int x) {

if (i >= j) {

m->A[m->n\*(j-1)+(j-2)\*(j-1)/2+i-j] = x;

}

}

int get(struct matrix m, int i, int j) {

if (i >= j)

return m.A[m.n \* (j - 1) + (j - 2) \* (j - 1) / 2 + i - j];

else

return 0;

}

void display(struct matrix m) {

int i, j;

for (i = 1; i <= m.n; i++) {

for (j = 1; j <= m.n; j++) {

if (i >= j)

printf("%d ", m.A[m.n \* (j - 1) + (j - 2) \* (j - 1) / 2 + i - j]);

else

printf("0 ");

}

printf("\n");

}

}

int main()

{

struct matrix m;

int i, j, x;

printf("Enter the dimension of matrix:");

scanf("%d", &m.n);

m.A = (int \*)malloc(m.n \* (m.n + 1) / 2 \* sizeof(int));

printf("Enter all the elements:\n");

for (i = 1; i <= m.n; i++) {

for (j = 1; j <= m.n; j++) {

scanf("%d", &x);

set(&m, i, j, x);

}

}

printf("\n\n");

display(m);

return 0;

}

**C++ class for lower triangular matrix**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

class matrix {

private:

int\* A;

int n;

public:

matrix() {

n = 5;

A = new int[2\*(2+1)/2];

}

matrix(int n) {

this->n = n;

A = new int[n\*(n+1)/2];

}

~matrix(){

delete []A;

}

void set(int i, int j, int x);

int get(int i, int j);

void display();

};

// ################# ROW-MAJOR ####################

/\*

void matrix::set(int i, int j, int x) {

if (i >= j) {

A[i \* (i - 1) / 2 + j - 1] = x;

}

}

int matrix::get(int i, int j) {

if (i <= j)

return A[i \* (i - 1) / 2 + j - 1];

else

return 0;

}

void matrix::display() {

int i, j;

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (i >= j)

cout << A[i \* (i - 1) / 2 + j - 1]<<" ";

else

cout << "0 ";

}

cout << endl;

}

}\*/

//####################### COLUMN-MAJOR ##########################

void matrix::set(int i, int j, int x) {

if (i >= j) {

A[n \* (j - 1) + (j - 2) \* (j - 1) / 2 + i - j] = x;

}

}

int matrix::get(int i, int j) {

if (i <= j)

return A[n \* (j - 1) + (j - 2) \* (j - 1) / 2 + i - j];

else

return 0;

}

void matrix::display() {

int i, j;

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (i >= j)

cout << A[n \* (j - 1) + (j - 2) \* (j - 1) / 2 + i - j] << " ";

else

cout << "0 ";

}

cout << endl;

}

}

int main()

{

int d;

cout << "Enter dimensions of matrix: ";

cin >> d;

matrix m(d);

int x;

cout << "Enter all elements";

for (int i = 1; i <= d; i++) {

for (int j = 1; j <= d; j++) {

cin >> x;

m.set(i, j, x);

}

}

m.display();

}

**Upper Triangular matrix**

// UPPER TRIANGULAR MATRIX

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

class matrix {

private:

int\* A;

int n;

public:

matrix() {

n = 5;

A = new int[2 \* (2 + 1) / 2];

}

matrix(int n) {

this->n = n;

A = new int[n \* (n + 1) / 2];

}

~matrix() {

delete[]A;

}

void set(int i, int j, int x);

int get(int i, int j);

void display();

};

// ################# COLUMN-MAJOR ####################

void matrix::set(int i, int j, int x) {

if (i <= j) {

A[j \* (j - 1) / 2 + i - 1] = x;

}

}

int matrix::get(int i, int j) {

if (i <= j)

return A[j \* (j - 1) / 2 + i - 1];

else

return 0;

}

void matrix::display() {

int i, j;

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (i <= j)

cout << A[j \* (j - 1) / 2 + i - 1]<<" ";

else

cout << "0 ";

}

cout << endl;

}

}

//####################### ROW-MAJOR ##########################

/\*

void matrix::set(int i, int j, int x) {

if (i <= j) {

A[n \* (i - 1) + (i - 2) \* (i - 1) / 2 + j - i] = x;

}

}

int matrix::get(int i, int j) {

if (i <= j)

return A[n \* (i - 1) + (i - 2) \* (i - 1) / 2 + j - i];

else

return 0;

}

void matrix::display() {

int i, j;

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (i >= j)

cout << A[n \* (i - 1) + (i - 2) \* (i - 1) / 2 + j - i] << " ";

else

cout << "0 ";

}

cout << endl;

}

}

\*/

int main()

{

int d;

cout << "Enter dimensions of matrix: ";

cin >> d;

matrix m(d);

int x;

cout << "Enter all elements";

for (int i = 1; i <= d; i++) {

for (int j = 1; j <= d; j++) {

cin >> x;

m.set(i, j, x);

}

}

m.display();

}

**Menu driven program for Diagonal matrix**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

using namespace std;

int main()

{

int\* A, n, ch, x, i, j;

printf("Enter dimensions of array: ");

scanf("%d", &n);

A = (int\*)malloc(n \* sizeof(int));

do {

printf("Display menu:\n 1. create\n 2. get\n 3. Set\n 4. Display\n Enter your choice: ");

scanf("%d", &ch);

switch (ch) {

case 1:

for (i = 1; i <= n; i++)

scanf("%d",& A[i - 1]);

break;

case 2:

printf("Enter indices: ");

scanf("%d %d", &i, &j);

if (i == j)

printf("%d", A[i - 1]);

else

printf("0");

break;

case 3:

printf("Enter row col details:");

scanf("%d %d %d", &i, &j, &x);

if (i == j)

A[i - 1] = x;

break;

case 4:

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (i == j)

printf("%d", A[i - 1]);

else

printf("0");

}

printf("\n");

}

break;

}

} while (ch > 0 && ch <= 4);

}

**SPARSE MATRIX**

**Creation of a sparse matrix:**

// creation of a sparse matrix

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

struct element {

int i;

int j;

int x;

};

struct sparse {

int m;

int n;

int num;

struct element\* e;

};

void create(struct sparse\* s) {

int i;

printf("Enter dimensions:");

scanf("%d %d", &s->m, &s->n);

printf("Enter number of non-zero elements:");

scanf("%d", &s->num);

s->e = (element\*)malloc(s->num \* sizeof(element));

printf("Enter all non-zero elements:");

for (i = 0; i < s->num; i++) {

scanf("%d %d %d", &s->e[i].i, &s->e[i].j, &s->e[i].x);

}

}

void display(struct sparse s) {

int i, j, k = 0;

for (i = 0; i < s.m; i++) {

for (j = 0; j < s.n; j++) {

if (i == s.e[k].i && j == s.e[k].j)

printf("%d ", s.e[k++].x);

else

printf("0 ");

}

printf("\n");

}

}

int main()

{

struct sparse s;

create(&s);

display(s);

return 0;

}

**Addition of two sparse matrix**

// sparse\_matrix.cpp : This file contains the 'main' function. Program execution begins and ends there.

// creation of a sparse matrix

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

using namespace std;

struct Element

{

int i;

int j;

int x;

};

struct Sparse

{

int m;

int n;

int num;

struct Element\* ele;

};

void create(struct Sparse\* s)

{

int i;

printf("Eneter Dimensions");

scanf("%d%d", &s->m, &s->n);

printf("Number of non-zero");

scanf("%d", &s->num);

s->ele = (struct Element\*)malloc(s->num \* sizeof(struct

Element));

printf("Eneter non-zero Elements");

for (i = 0; i < s->num; i++)

scanf("%d%d%d", &s->ele[i].i, &s->ele[i].j, &s -> ele[i].x);

}

void display(struct Sparse s)

{

int i, j, k = 0;

for (i = 0; i < s.m; i++)

{

for (j = 0; j < s.n; j++)

{

if (i == s.ele[k].i && j == s.ele[k].j)

printf("%d ", s.ele[k++].x);

else

printf("0 ");

}

printf("\n");

}

}

struct Sparse\* add(struct Sparse\* s1, struct Sparse\* s2)

{

struct Sparse\* sum;

int i, j, k;

i = j = k = 0;

if (s1->n != s2->n && s1->m != s2->m)

return NULL;

sum = (struct Sparse\*)malloc(sizeof(struct Sparse));

sum->ele = (struct Element\*)malloc((s1->num + s2 -> num) \* sizeof(struct Element));

while (i < s1->num && j < s2->num)

{

if (s1->ele[i].i < s2->ele[j].i)

sum->ele[k++] = s1->ele[i++];

else if (s1->ele[i].i > s2->ele[j].i)

sum->ele[k++] = s2->ele[j++];

else

{

if (s1->ele[i].j < s2->ele[j].j)

sum->ele[k++] = s1->ele[i++];

else if (s1->ele[i].j > s2->ele[j].j)

sum->ele[k++] = s2->ele[j++];

else

{

sum->ele[k] = s1->ele[i];

sum->ele[k++].x = s1->ele[i++].x + s2->ele[j++].x;

}

}

}

for (; i < s1->num; i++)sum->ele[k++] = s1->ele[i];

for (; j < s2->num; j++)sum->ele[k++] = s2->ele[j];

sum->m = s1->m;

sum->n = s1->n;

sum->num = k;

return sum;

}

int main()

{

struct Sparse s1, s2, \* s3;

create(&s1);

create(&s2);

s3 = add(&s1, &s2);

printf("First Matrix\n");

display(s1);

printf("Second Matrix\n");

display(s2);

printf("Sum Matrix\n");

display(\*s3);

return 0;

}

**C++ Class for Sparse matrix**

#include <iostream>

using namespace std;

class element {

public:

int i;

int j;

int x;

};

class sparse {

private:

int m;

int n;

int num;

element\* e;

public:

sparse(int m, int n, int num) {

this->m = m;

this->n = n;

this->num = num;

e = new element[this->num];

}

~sparse(){

delete[] e;

}

void read() {

cout << "Enter non-zero elements:\n";

for (int i = 0; i < num; i++) {

cin >> e[i].i >> e[i].j >> e[i].x;

}

}

void display() {

int k = 0;

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (e[k].i == i && e[k].j == j) {

cout << e[k++].x << " ";

}

else

cout << "0 ";

}

cout << endl;

}

}

};

int main()

{

sparse s1(5, 5, 5);

s1.read();

s1.display();

return 0;

}

**Istream and ostream concept in sparse matrix**

#include <iostream>

using namespace std;

class element {

public:

int i;

int j;

int x;

};

class sparse {

private:

int m;

int n;

int num;

element\* e;

public:

sparse(int m, int n, int num) {

this->m = m;

this->n = n;

this->num = num;

e = new element[this->num];

}

~sparse() {

delete[] e;

}

friend istream& operator>>(istream& is, sparse& s);

friend ostream& operator<<(ostream& os, sparse& s);

};

istream& operator>>(istream& is, sparse& s) {

cout << "Enter non-zero elements:\n";

for (int i = 0; i < s.num; i++) {

cin >> s.e[i].i >> s.e[i].j >> s.e[i].x;

}

return is;

}

ostream& operator<<(ostream& os, sparse& s) {

int k = 0;

for (int i = 0; i < s.m; i++) {

for (int j = 0; j < s.n; j++) {

if (s.e[k].i == i && s.e[k].j == j)

cout << s.e[k++].x << " ";

else

cout << "0 ";

}

cout << endl;

}

return os;

}

int main()

{

sparse s1(5, 5, 5);

cin >> s1;

cout << s1;

return 0;

}

**LINKED LIST**

Problem with array is that once it is created, it’s size cannot be changed.

When we want to store a data which can either increase or decrease in size, then we use the variable size data structure which is Linked List.

**Definition:**

Linked list is a collection of nodes where each node contains data and pointer to next node. Head is a pointer which points to first node or Head node of the linked list.

A Self-referential structure is used when creating a structure Node.

**C program to create a linked list and display it**

// Linked List creation and display

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

using namespace std;

struct node {

int data;

struct node\* next;

}\*first = NULL;

void create(int A[], int n) {

int i;

struct node\* t,\*last;

first = (struct node\*)malloc(sizeof(struct node));

first->data = A[0];

first->next = NULL;

last = first;

for (i = 1; i < n; i++) {

t = (struct node\*)malloc(sizeof(struct node));

t->data = A[i];

t->next = NULL;

last->next = t;

last = t;

}

}

void display(struct node\* p) {

while (p != NULL) {

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

p = p->next;

}

}

int main() {

int A[] = { 3,5,7,10,15 };

create(A, 5);

display(first);

}

**Recursive function to display the linked list**

void display(struct node\* p) {

if (p != NULL) {

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

display(p->next);

}

}

***If we want to print the elements of linked list in reverse order then we have to call the recursive function before printing the p->data in the above code.***

**Function to count the number of nodes in a linked list:**

int count(struct node\* p) {

int count = 0;

while (p != NULL) {

count += 1;

p = p->next;

}

return count;

}

**Recursive function to count the number of nodes in a linked list:**

/\* Recursive function for counting the nodes in linked list \*/

int count(struct node\* p) {

if (p == NULL)

return 0;

else

return count(p->next) + 1;

}

**Function to find the max element in a linked list**

int max1(struct node\* p) {

int m = INT32\_MIN;

while (p) {

if (p->data > m) {

m = p->data;

}

p = p->next;

}

return m;

}

**Searching in Linked List**

In case of linked list, binary search is not suitable as we don’t know the size of linked list beforehand. Therefore, we must go ahead with linear search.

Ways to improve searching:

1. Transposition
2. Move to head

**Code:**

/\*function for linear search \*/

node\* search(struct node\* p, int key) {

while (p) {

if (key == p->data) {

return p;

}

p = p->next;

}

return NULL;

}

/\*imporved function with move to head implemented for liner search\*/

node\* search1(struct node\* p, int key) {

struct node\* q = NULL;

while (p) {

if (key == p->data) {

q->next = p->next;

p->next = first;

first = p;

}

q = p;

p = p->next;

}

return NULL;

}

int main() {

struct node\* temp;

int A[] = { 3,5,7,10,15,20,25,30,35,40,45 };

create(A, 11);

display(first);

//printf("\nThe no. of elements in linked list is: %d", count(first));

//printf("\n The sum of elements of linked list is: %d", sum(first));

//printf("\n The maximum element in linked list is: %d", max1(first));

temp = search1(first, 25);

if (temp)

printf("\nkey is found: %d\n", temp->data);

else

printf("\nKey is not found\n");

display(first);

return 0;

}

**Function to insert elements in a Linked list**

void insert(struct node\* p, int index, int x) {

struct node\* t;

int i;

// checking if the index enetered is correct or not

if (index<0 || index>count(p)) {

return;

}

t = (struct node\*)malloc(sizeof(node));

t->data = x;

// if the index entered is 0

if (index == 0) {

t->next = first;

first = t;

}

// if the index entered is other than 0

else {

for (i = 0; i < index - 1; i++) {

p = p->next;

}

t->next = p->next;

p->next = t;

}

}

***Insert function can also be used to create a linked list.***

**Deleting a node from a linked list**

While deleting a node from the linked list, two cases arise:

1. Deleting the first node
2. Deleting node at a given position

/\* Function to delete a node from linked list \*/

int Delete(struct node\* p, int index) {

struct node\* q = NULL;

int x = -1,i;

if (index<1 || index>count(p)) // check if the index entered is valid or not

return -1;

if (index == 1) { // if first noe neds to be deleted

q = first;

x = first->data;

first = first->next;

free(q);

return x;

}

// if other nodes at specific index needs to be deleted

else {

for (i = 0; i < index - 1; i++) {

q = p;

p = p->next;

}

q->next = p->next;

x = p->data;

free(p);

return x;

}

}

**Function to check if the linked list is sorted or not**

/\* function to check if the linked list is sorted or not \*/

int Is\_sorted(struct node\* p) {

int x = INT32\_MIN;

while (p != NULL) {

if (p->data < x)

return 0;

x = p->data;

p = p->next;

}

return 1;

}

/\* function to rempve duplicates from the linked list \*/

void rduplicate(struct node\* p) {

// we have to take one more pointer pointing to the next element

struct node\* q = p->next;

// iterate q till null and check if p->data is equal to q->data

while (q != NULL) {

// if p and q data are not same, increase both

if (q->data != p->data) {

p = q;

q = q->next;

}

// if the data is equal in both, then perform the below steps

else {

p->next = q->next;

free(q);

q = p->next;

}

}

}

/\* function to reverse the elements in linked list by reversing the elements method \*/

void reverse\_elements(struct node\* p) {

// first we are taking an array dynamically to store the elements of linked list

int\* A;

A = (int\*)malloc(count(first) \* sizeof(int));

int i = 0;

// copying all the elements of linked list to A

while (p != NULL) {

A[i++] = p->data;

p = p->next;

}

p = first; // pointing p back to first node

i--;

// copying the elements back to linked list from array

while (p != NULL) {

p->data = A[i--];

p = p->next;

}

}

/\* function to reverse the elements in linked list by reversing the links \*/

void reverse\_links(struct node\* p) {

/\* Here we are using the concept of sliding pointer that is why we have taken the two pointers

p and q of node type as shown below \*/

struct node\* q = NULL;

struct node\* r = NULL;

while (p != NULL) {

r = q;

q = p;

p = p->next;

q->next = r;

}

first = q;

}

/\*recursive function to reverse te elements in the linked list \*/

void recursive\_reverse(struct node\* q, struct node\* p) {

if (p != NULL) {

recursive\_reverse(p, p->next);

p->next = q;

}

else {

first = q;

}

}

/\*function to concanate two linked lists \*/

void concat(struct node\* p, struct node\* q) {

third = p;

while (p->next != NULL)

p = p->next;

p->next = q;

}

/\* function to merge two linked lists \*/

void merge(struct node\* p, struct node\* q) {

struct node\* last, third;

if (p->data < q->data) {

third = last = p;

p = p->next;

third->next = NULL;

}

else {

third = last = q;

q = q->next;

third->next = NULL;

}

while (p && q) {

if (p->data < q->data) {

last->next = p;

last = p;

p = p->next;

last->next = NULL;

}

else {

last->next = q;

last = q;

q = q->next;

last->next = NULL;

}

}

if (p) last->next = p;

if (q) last->next = q;

}

int isloop(struct node\* f) {

struct node\* p, \* q;

p = q = f;

do {

p = p->next;

q = q->next;

q = q ? q->next : q;

} while (p && q && p != q);

if (p == q)

return 1;

else

return 0;

}

**Circular Linked List**

In a circular linked list, the last node points to the first node so that it is like a circle.

**Code:**

#define CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include<stdio.h>

using namespace std;

/\*node creation\*/

struct node {

int data;

struct node\* next;

}\*head;

/\*creation of a linked list\*/

void create(int A[], int n) {

int i;

struct node\* t, \* last;

head = (struct node\*)malloc(sizeof(node));

head->data = A[0];

head->next = head;

last = head;

for (i = 1; i < n; i++) {

t = (struct node\*)malloc(sizeof(node));

t->data = A[i];

t->next = last->next;

last->next = t;

last = t;

}

}

/\*Recursive display of linked list\*/

void Rdisplay(struct node\* h) {

static int flag = 0;

if (h != head || flag == 0) {

flag = 1;

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

Rdisplay(h->next);

}

flag = 0;

}

/\*Display of linked list\*/

void display(struct node\* h) {

do {

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

h = h->next;

} while (h != head);

printf("\n");

}

int main()

{

int A[] = { 2,3,4,5,6 };

create(A, 5);

Rdisplay(head);

}

**Inserting a new element in Circular linked list:**

There are two possibilities for entering a new element in a linked list as mentioned below:

1. Insert before head/Inserting after last node.
2. Insert at any other position.

/\*Inserting a new element in circular linked list \*/

void insert(struct node\* p, int index, int x) {

struct node\* t;

int i;

if (index == 0) {

t = (struct node\*)malloc(sizeof(node));

t->data = x;

if (head == NULL) {

head = t;

head->next = head;

}

else {

while (p->next != head)

p = p->next;

p->next = t;

t->next = head;

head = t;

}

}

for (i = 0; i < index - 1; i++) p = p->next;

t = (struct node\*)malloc(sizeof(struct node));

t->data = x;

t->next = p->next;

p->next = t;

}

**Deleting a node from a circular linked list**

There are also two possibilities for deleting a node from a circular linked list:

1. Delete head node
2. Deleting a node from a given position

/\*Function to delete a particular node from circular linked list \*/

int Delete(struct node\* p, int index) {

struct node\* q;

int i, x;

if (index == 1) {

while (p->next != head) p = p->next;

x = head->data;

if (p == head) {

free(head);

head = NULL;

}

else {

p->next = head->next;

free(head);

head = p->next;

}

}

else {

for (i = 0; i < index - 2; i++)

p = p->next;

q = p->next;

p->next = q->next;

x = q->data;

free(q);

}

return x;

}

**Doubly Linked List**

Code for implementing a doubly linked list:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <stdio.h>

#include<stdlib.h>

using namespace std;

struct node {

struct node\* prev;

int data;

struct node\* next;

}\*first = NULL;

void create(int A[], int n) {

struct node\* t, \*last;

int i;

first = (struct node\*)malloc(sizeof(struct node));

first->data = A[0];

first->prev = first->next = NULL;

last = first;

for (i = 1; i < n; i++) {

t = (struct node\*)malloc(sizeof(struct node));

t->data = A[i];

t->next = last->next;

t->prev = last;

last->next = t;

last = t;

}

}

void Display(struct node\* p) {

while (p != NULL) {

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

p = p->next;

}

printf("\n");

}

int Length(struct node\* p) {

int len = 0;

while (p) {

len++;

p = p->next;

}

return len;

}

int main()

{

int A[] = { 10,20,30,40 };

create(A, 4);

printf("\n");

Display(first);

return 0;

}

**Inserting in a doubly linked list:**

When we talk about inserting a new node in doubly linked list, two cases arise:

1. Before first node
2. At a given index

void Insert(struct node\* p, int index, int x) {

struct node\* t;

if (index < 0 || index > Length(p))

return;

if (index == 0) { /\* If we are inserting on the first place\*/

t = (struct node\*)malloc(sizeof(struct node));

t->data = x;

t->prev = NULL;

t->next = first;

first->prev = t;

first = t;

}

/\*If we are inserting after a certain node \*/

else {

int i = 0;

while (i < index-1) {

p = p->next;

i++;

}

t = (struct node\*)malloc(sizeof(struct node));

t->data = x;

t->next = p->next;

t->prev = p;

if(p->next)

p->next->prev = t;

p->next = t;

}

}

**Deleting a node from the doubly linked list**

When we talk about deleting the node, two scenarios arise:

1. Deleting the first node
2. Deleting a node at a given index

int Delete(struct node\* p, int index) {

struct node\* q;

int x = -1, i;

if (index<1 || index>Length(p)) {

return -1;

}

if (index == 1) { /\*deleting first node\*/

first = first->next;

if (first) first->prev = NULL;

x = p->data;

free(p);

}

else { /\*deleting other nodes\*/

for (i = 0; i < index - 1; i++) {

p = p->next;

}

p->prev->next = p->next;

if (p->next) p->next->prev = p->prev; /\*check if the node to be deleted is the last node\*/

x = p->data;

free(p);

}

return x;

}

**Reversing a doubly linked list**

void Reverse(struct node\* p) {

struct node\* temp;

while (p) {

temp = p->next;

p->next = p->prev;

p->prev = temp;

p = p->prev;

if (p!=NULL && p->next == NULL)

first = p;

}

}

**Circular doubly linked list**

void create(int A[], int n) {

struct node\* t, \* last;

int i;

first = (struct node\*)malloc(sizeof(struct node));

first->data = A[0];

first->prev = first->next = NULL;

last = first;

for (i = 1; i < n; i++) {

t= (struct node\*)malloc(sizeof(struct node));

t->data = A[i];

t->prev = last;

t->next = last->next;

last->next = t;

last = t;

}

first->prev = last;

last->next = first;

}

void Display(struct node\* p) {

do {

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

p = p->next;

} while (p != first);

printf("\n");

}

**Exercise – Finding the middle element in a linked list:**

int middle\_node(struct node \*p){

    struct node \*q;

    p=q=first;

    while(q){

        q=q->next;

        if(q) q=q->next;

        if(q) p=p->next;

    }

    return p->data;

}