C++ Foundation with Data Structures

Topic : Pointers

Address of Operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand
sign (&), known as *address-of operator*. For example:

```
cout << (&var) << endl;
```

This would print address of variable *var*; by preceding the name of the variable var with the *address-of operator* (&), we are no longer printing the content of the variable itself, but its address.

What are Pointers?

Pointers are one of the most important aspects of C++. Pointers are another type of variables in CPP and these variables store addresses of other variables.

While creating a pointer variable, we need to mention the type of data whose address is stored in the pointer. e.g. in order to create a pointer which stores address of an integer, we need to write:

```
int* p;
```

This means that p will contain address of an integer. So, if a pointer is going to store address of datatype X, it will be declared like this:

```
X* p;
```

Now let's say we have an integer i & an integer pointer p, we will use addressof(&) operator in order to put address of i in p. Address of operator:& as studied abpve is a unary operator which returns address of a variable. e.g. &i will give us address of variable i. Here is the code to put address of i in p.

```
int i = 10;
int* p;
p = &i;
```

Dereference Operator

As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, consider the following statement:

int a = *p;

So in this assignment we are assigning value pointed to by pointer p(i.e. value of to int i) to int variable a.

The reference and dereference operators are thus complementary:

- & is the *address-of operator*, and can be read simply as "address of"
- * is the *dereference operator*, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with *.

Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the *dereference operator* seen above, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.

Following

```
#include <iostream>
using namespace std;

int main ()
{
 int firstvalue = 5, secondvalue = 15;
char thirdvalue = ‘a’;
 int * p1, * p2;
char *p3;
```

```
firstvalue is 10
secondvalue is
20
thirdvalue is b
```

```
 p1 = &firstvalue;  // p1 = address of firstvalue
 p2 = &secondvalue; // p2 = address of secondvalue
 p3 = &thirdvalue;    // p3 = address of thirdvalue
 *p1 = 10;         // value pointed to by p1 = 10
 *p2 = *p1;        // value pointed to by p2 = value
pointed to by p1
 p1 = p2;          // p1 = p2 (value of pointer is copied)
 *p1 = 20;         // value pointed to by p1 = 20
*p3 = ‘b’   // value pointed to by p3 = ‘b ’

 cout << "firstvalue is " << firstvalue << '\n';
 cout << "secondvalue is " << secondvalue << '\n';
 cout << "thirdvalue is " << thirdvalue << '\n';
 return 0;
}
```

Note: While solving pointers question, you should use pen and paper and draw things to get better idea.

Null Pointer

Consider the following statement –

int *p;

Here we have created a pointer variable that contains garbage value. In order to dereference the pointer, we will try reading out the value at the garbage stored in the pointer. This will lead to unexpected results or segmentation faults. Hence we should never leave a pointer uninitialized and instead initialize it to NULL, so as to avoid unexpected behavior.

int *p = NULL; // NULL is a constant with a value 0
int *q = 0;   // Same as above

So now if we try to dereference the pointer we will get segmentation fault as 0 is a reserved memory address.

Pointer Arithmetic

Arithmetic operations on pointers behave differently than they do on simple data types we studied earlier. Only addition and subtraction operations are

allowed; the others aren't allowed on pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

For example: char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Suppose now that we define three pointers in this compiler:

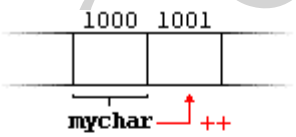
```
char *mychar;
short *myshort;
long *mylong;
```

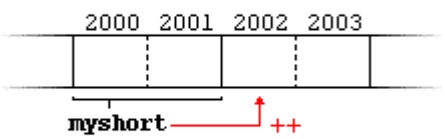
and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

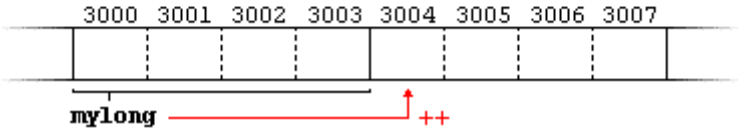
Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.







This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

Pointer and Arrays

Pointers and arrays are intricately linked. An Array is actually a pointer that points to the first element of the array. Because the array variable is a pointer, you can dereference it, which returns array element 0.

Consider the following code –

```
int a[] = {1,2,3,4,5};
int *b = &a[0];
cout << b << endl;
cout << a << endl;
cout << *b << endl; // This will print 1
```

Both b and a will print same same address as they are referring to first element of the array.
Also in arrays - *a[i]* is same as **(a + i).*

Consider an example for the same–

```
#include<iostream>
using namespace std;

int main(){

  int a[5] = {1,2,3,4,5};
  cout << *(a + 2) << endl;

}
```

Output:

3

Differences between arrays and pointers:

1. **the sizeof operator:**
   sizeof(array) returns the amount of memory used by all elements in array whereas sizeof(pointer) only returns the amount of memory used by the pointer variable itself.

```
#include<iostream>
using namespace std;

int main(){

  int a[5] = {1,2,3,4,5};
  int *b = &a[0];
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
}
```

   Output :

```
20
8  // Size of pointer is compiler dependent. Here it is 8.
```

2. **the & operator**
   In the example above &a is an alias for &a[0] and returns the address of the first element in array
   &b returns the address of pointer.

3. **Pointer variable can be assigned a value whereas array variable cannot be.**

```
int a[10];
int *p;
p=a;        //legal
a=p;        //illegal
```

4. **Arithmetic on pointer variable is allowed, but not allowed on array variable.**

```
p++;        //Legal
a++;        //illegal
```

Double Pointer

As we know by now that pointers are variables that store address of other variables, so we can create variables that store address of pointer itself i.e. a pointer to a pointer. Let's see how can we create one.

```
int a = 10;
int *p = &a;
int **q = &p;
```

Here q is a pointer to a pointer i.e. a double pointer,as indicated by **.

Consider the following code for better understanding –

```
#include<iostream>
using namespace std;
int main(){
 int a = 10;
 int *p = &a;
 int **q = &p;

// Next three statements will print same value i.e. address of a
 cout << &a << endl;
 cout << p << endl;
 cout << *q << endl;

// Next two statements will print same value i.e. address of p
 cout << &p << endl;
 cout << q << endl;
```

```
// Next two statements will print same value i.e. value of a which is 10
  cout << a << endl;
  cout << *p << endl;
  cout << **q << endl;
}
```

Void Pointer

A void pointer is a generic pointer, it has no associated type with it. A void pointer can hold address of any type and can be typcasted to any type. Void pointer is declared normally the way we do for pointers.

```
void *ptr;
```

This statement will create a void pointer.

Example:

```
void *v;
int *i;
int ivar;
char chvar;
float fvar;
v = &ivar; // valid
v = &chvar; //valid
v = &fvar; // valid
i = &ivar; //valid
i = &chvar; //invalid
i = &fvar; //invalid
```

Thus we can use void pointer to store address of any variable.

Dynamic Allocation

Address typecasting

While declaring a pointer, why can't we just write like this:

```
int a = 5;
pointer p = &a;
```

Why do we have a complicated syntax like:

```
int a = 5;
int *p = &a;
```

It's generally because we need to specify that, when we invoke a particular pointer at any point, then how will the compiler know what type of value a pointer has stored and while invoking/transferring data, how much space needs to be allotted to it.

That is why while declaring a pointer, we start with the datatype and then assign the name to the pointer. That datatype specifies what type of value we are storing in the pointer.

The term **typecasting** means assigning one type of data to another type like storing an integer value to a char data type. For example:

```
int x= 65;
char c = x;
```

When you will check the value stored in variable 'c', it will print the ASCII value stored at that integer variable i.e., 'x' and will print 'A' (whose ASCII value is 65).

This type of typecasting done above is known as **implicit typecasting** as the compiler itself interprets the conversion of integer value to ASCII character value.

Now consider the example below:

```
int i = 65;
int *p = &i;
char *pc = (char*) p;
```

You can see that in the third line, we can't directly do like this:

char *pc = p;

This will give an error as we are trying to store a integer-type pointer value into a character-type value. To remove the error we have to type-cast by ourselves by providing (char*) on the right hand side as done in the code above. This type of typecasting is known as **explicit typecasting**.

Reference and pass by Reference

As you are familiar, that (&) symbol is known as a reference operator which means 'Address of operator'.

This operator is used to copy the values of any variable along with the guarantee that the reflected changes will also be visible in the copied variable.

For example:

```
int a = 5;
int b = a;
a++;
cout<<b<<endl;
```

Output:

```
5
```

Means that only the value is copied and when the value of the variable is increased by one, then the changes are not reflected in variable 'b'.

Now, look at the following piece of code:

```
int a = 5;
int &b = a;
a++;
cout<<b<<endl;
```

Here, (&b=a) means that now variables b and a are pointing to the same address and making changes in any of them gets reflected to both of the variables.

The above code outputs: 6

This concept of referencing the variables is useful in those cases where we want to update the value passed to the function. When we normally pass a value to a variable then a copy of those is created in the system and the original values remain unchanged. But by passing the reference, the changes are also reflected in the original variables as there is no extra copy created. The first type of argument passing is known as **pass by value** and the later one is known as **pass by reference.**

Syntax for pass by reference:

```
#include <iostream>
using namespace std;

void fun(int &a) {
    a++;
}

int main() {
    int a = 5;
    fun(a);
    cout << a << endl;
}
```

Output:

```
6
```

Advantages of Pass by reference:

- Reduction in memory storage.
- Changes can be reflected easily.

Dynamic memory allocation

In the array, it is always advised that while declaring an array, we should always provide the size that is a constant value and not a variable in order to prevent Runtime error. Runtime error can happen if the variable contains a garbage value or some type of undefined value. Generally, we have two types of memory in our systems:

- **Stack memory:** It has a fixed value of size for which an array could be declared in the contiguous form.
- **Heap memory:** It is the memory where the array declared is not stored in a contiguous way. Suppose, if we want to declare an array of size 100000, but we do not have the contiguous space in the memory of the same size, then we will be declaring the array using heap memory as it necessarily don't require a contiguous allocation, it will allot the space wherever it gets and links all the memory blocks together.

  Note: Global variables are stored using heap memory.

The memory declaration using heap memory is known as **dynamic memory allocation** while the one using stack is known as **compile-time memory allocation**. So, how can we access heap memory?

The syntax is as follows:

```
int *arr = new int[size_of_array];
```

For simply allocating variables dynamically, use this:

```
int *var = new int;
```

You can see here, we are using a keyword **new** which is used to declare the dynamic memory in C++. There are other ways too but this one is the most efficient.

Note: Stack memory releases itself as the scope of the variable gets over, but in case of heap memory, it needs to be released manually at last otherwise the memory gets accumulated and in the end leads to memory leakage.

Syntax for releasing an array:

```
delete [ ] arr;
```

Syntax for releasing the memory of the variable:

```
delete var;
```

The keyword **delete** is used to clear the heap memory.

Now moving on how to create 2D arrays in the dynamic memory allocation...

Dynamic allocation of 2D arrays

To create a 2D array in the heap memory, we use the following syntax:

Suppose we want to create a 2D array of size n x m, with the name of the array as 'arr':

```
int **arr = new int*[n];
for (int i=0; i < n; i++) {
        arr[i] = new int[m];
}
```

You can see that first-of-all we have declared a pointer of pointers of size n and then at each pointer while traversing we allocated the array of size m using *new* keyword.

Similarly, we can release this memory, using the following syntax:

```
for (int i=0; i < n; i++) {
     delete [ ] arr[i];
}
delete [ ] arr;
```

First-of-all, we have cleared the memory allocated to each of the n pointers and then finally deleted those n pointers also.

This way, using *delete* keyword, we can release the memory of the 2D arrays.

Macros and global variable

Suppose in your code you are using the value pi(3.14) many times, instead of always writing 3.14 everywhere in the code, what we can do is define this value to some variable say pi in the beginning of the program and now everytime we need the value 3.14, we just need to write pi.

What we can do is that we can create a global variable with the value 3.14 and then use it anywhere as desired. There is one issue in using global variables and that is if someone changes the value of pi in our code at any point of time in any of the functions, then we could lose that original value.

To avoid such a situation, what we do is use macros. Syntax for using it:

```
#define pi 3.14
```

This is done after declaring the header files in the beginning itself so that this value can be used in the later encountered functions. Here, what we are actually doing is that we are declaring the value of the pi as 3.14 using the macros (#) define which basically locks the value and makes it unchangeable.

Another advantage is it prevents extra storage in the memory for declaring a new variable as by using macros we have specified to the compiler that the value we are using is a part of the compiler code, hence no need to create any extra memory.

Now, discussing about the advantages of the global variable:

- When we want to use the same variable with modified values in each of them, then we can use global variables as the changes done in one function are visible in all the other.
- It saves time for passing the values by reference in the functions.

Due to accidental changes, this method is not preferred much.

Inline and default arguments

Disadvantages of using a normal function:

- Uses time as it pauses some portion of code to get complete unless we are done with it we can't move forward.
- Creates a copy of variables each time while calling a function.

To prevent this what we can do is create inline functions. Inline functions replace the function call with its definition of when invoked.

Syntax:

```
inline fun() {
  ...
}
```

Just write the keyword inline before the function call.

When it is invoked like this:

```
fun();
```

Inside any other function, then what happens is it gets replaced with the function declaration hence preventing the pausing of the code at any point of time.

Disadvantages of using inline functions:

- Code becomes bulky.
- Time taken to copy code can be huge.

Now moving on to the use of default arguments...

Actually, sometimes we are unsure about any value(s) to be passed into the function as an argument but in the further calculations we need to use them as it is necessary to use them either by defined value or through some default value. This purpose is served by default arguments in C++. Using these we can specify a value to any variable in the function declaration to some default value that could be used if no value is passed to it by the function call.

Note: Be careful about using these arguments in the function call, default arguments can only be declared as the rightmost set of parameters.

For example: To calculate the sum of numbers, we are unsure that if we want to find the sum of two or three numbers, then:

```
int sum(int a, int b, int c = 0) {        // here, c is the default argument
    return a + b + c;
}

int main() {
    int a = 2, b = 3, c = 4;
    cout << sum(a, b) << endl;        // here, c will automatically be taken as 0
    cout << sum(a, b, c) << endl;     // as the value of c is provided, the value of c will
be 4
}
```

Output:

```
5
9
```

Constant variables

Now moving on to the constant variables which are identified by **const** keyword in C++. As the name suggests, if we declare any keyword as constant, then we can't change it's value throughout the program.

Note: The constant variable needs to be assigned during initialization only else it will store garbage value in it which can't be changed further.

Syntax:

```
const datatype variable_name = value;
```

For example:

```
const int a = 5;
```

C++ Foundation with Data Structures

Topic: Recursion

Recursion

a. What is Recursion?

In previous lectures, we used iteration to solve problems. Now, we'll learn about recursion for solving problems which contain smaller sub-problems of the same kind.

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. By same nature it actually means that the approach that we use to solve the original problem can be used to solve smaller problems as well.

So in other words in recursion a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts and the code is also shorter and easier to understand.

b. How Does Recursion Work?

We can define the steps of a recursive solution as follows:

1. **Base Case:**
   A recursive function must have a terminating condition at which the function will stop calling itself. Such a condition is known as a base case.
2. **Recursive Call:**
   The recursive function will recursively invoke itself on the smaller version of problem. We need to be careful while writing this step as it is important to correctly figure out what your smaller problem is on whose solution the original problem's solution depends.
3. **Small Calculation:**
   Generally we perform a some calculation step in each recursive call. We can perform this calculation step before or after the recursive call depending upon the nature of the problem.

It is important to note here that recursion uses stack to store the recursive calls. So, to avoid memory overflow problem, we should define a recursive solution with minimum possible number of recursive calls such that the base condition is achieved before the recursion stack starts overflowing on getting completely filled.

Now, let us look at an example to calculate factorial of a number using recursion.

Example Code 1:

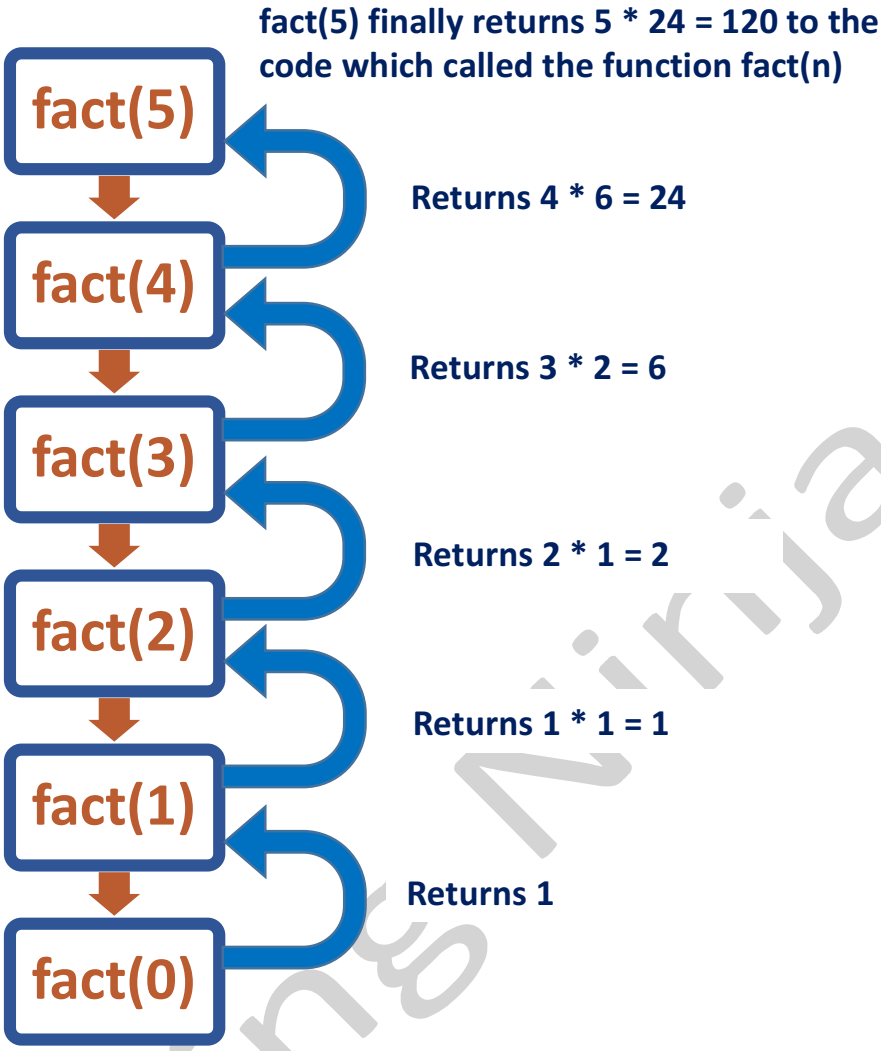
```
#include<iostream>
using namespace std;

int fact(int n)
{
   if(n==0)                          //Base Case
   {
      return 1;
   }
   return n * fact(n-1);             //Recursive call with small calculation
}

int main()
{
   int num;
   cin>>num;
   cout<<fact(num);
   return 0;
}
```

Output:

```
120          //For num=5
```

Explanation:

Here, we called factorial function recursively till number became 0. Then, the statements below the recursive call statement were executed. We can visualize the recursion tree for this function, where let n=5, as follows:



We are calculating the factorial of n=5 here. We can infer that the function recursively calls fact(n) till n becomes 0, which is the base case here. In the base case, we returned the value 1. Then, the statements after the recursive calls were executed which returned n*fact(n-1) for each call. Finally, fact(5) returned the answer 120 to main() from where we had invoked the fact() function.

Now, let us look at another example to find n^{th} Fibonacci number . In Fibonacci series to calculate nth Fibonacci number we can use the formula F(n) = F(n – 1) + F(n – 2) i.e. nth Fibonacci term is equal to sum of n-1 and n-2 Fibonacci terms. So let's use this to write recursive code for nth Fibonacci number.

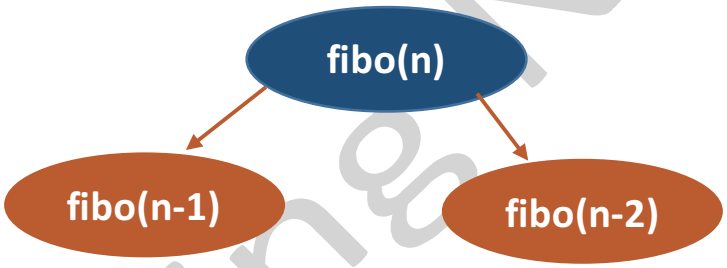
Example Code 2:

// Recursive function:

```
int fibo(int n) {
   if(n==0 || n==1) {          //Base Case
      return n;
   }
   int a = fibo(n-1);          //Recursive call
   int b = fibo(n-2);          //Recursive call
   return a+b;                 //Small Calculation and return statement
}
```

Explanation:
As we are aware of the Fibonacci Series (0, 1, 1, 2, 3, 5, 8,… and so on), let us assume that the index starts from 0, so, 5^{th} Fibonacci number will correspond to 5; 6^{th} Fibonacci number will correspond to 8; and so on.

Here, in recursive Fibonacci function, we have made two recursive calls which are depicted as follows:



Note: One thing that we should be clear about is that both recursive calls don't happen simultaneously. First fibo(n-1) is called, and only after we have its result and store it in "a" we move to next statement to calculate fibo(n – 2).

It is interesting to note here that the concept of recursion is based on the mathematical concept of **PMI** (Principle of Mathematical Induction). When we use PMI to prove a theorem, we have to show that the base case (usually for x=0 or x=1) is true and, the induction hypothesis for case x=k is true must imply that case x=k+1 is also true. We can now understand how the steps which we followed in recursion are based on the induction steps, as in recursion also, we have a base case while the assumption corresponds to the recursive call.

Recursion 1

INTRODUCTION

Since computer programming is a fundamental application of mathematics, so first, let's try to understand the mathematical logic behind recursion.

In general, we all are aware of the concept of functions. Briefly, functions are the mathematical equations that produce an output on providing input. Different types of inputs can have a single value of output, but different types of outputs can't have a single input. **For example:** Suppose F(x) is a function defined by:

$$\mathbf{F(x) = x^2 + 4}$$

where F(x) is the output received for a particular input x. We can observe that if we put x = 2, we receive output as 8 and if we put x = -2, then also we receive output as 8. Representing this in the form of a computer program:

```
int F(int x) {
        return (x * x + 4);
}
```

Now, we can pass different values of x to this function and receive our output accordingly.

This proves **one-to-one mapping** of programming functions with mathematical functions.

As we vaguely recall, there was another concept called "**Principle of Mathematical Induction**" in mathematics, let's try to figure out if we have something in programming for this concept.

Before we dig deeper into it, let's revise. Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about every natural number. It has following three steps:

1. Step of trivial case: In this step, we will prove the desired statement for n = 1.

2. Step of assumption: In this step, we will assume that the desired statement is valid for n = k.
3. To prove step: From the results of assumption step, we will prove that n = k + 1 also holds for the desired equation.

For Example: Let's prove that:

S(n): 1 + 2 + 3 + ... + n = (n * (n + 1))/2 (the sum of first n natural numbers)

Proof:

Step 1: For n = 1, S(1) = 1 is true.

Step 2: Assume, the given statement is true for n = k, i.e.,

1 + 2 + 3 + ... + k = (k * (k + 1))/2

Step 3: Let's prove the statement for n = k + 1 using step 2.

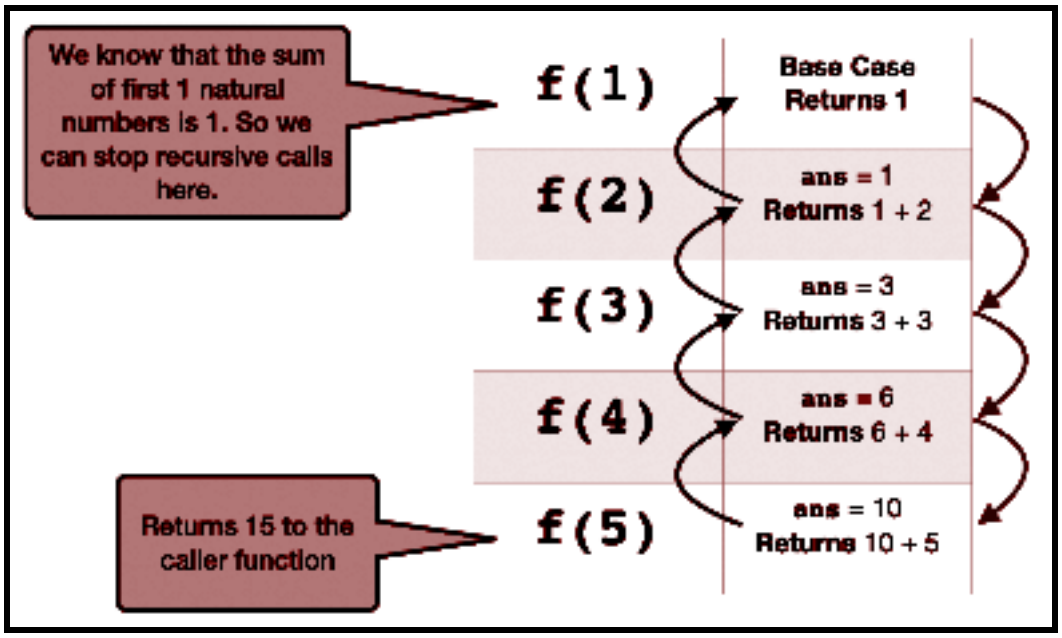
Adding (k+1) to both LHS and RHS in the result obtained on step 2: **1 + 2 + 3 + ... + (k+1) = (k*(k+1))/2 + (k+1)** *Now, taking (k+1) common from RHS side:* **1 + 2 + 3 + ... + (k+1) = (k+1)*((k + 2)/2)** *According the statement that we are trying to prove:* **1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2** *Which is the same as we obtained above. Hence proved*

One can think, why are we discussing these over here. To answer this question, we need to know that these three steps of PMI are related to the three steps of recursion, which are as follows:

1. Induction Step (IS) and Induction Hypothesis (IH): Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define Induction Step and Induction Hypothesis for our running example:
   Induction Step: Sum of first n natural numbers - F(n)
   Induction Hypothesis: This gives us the sum of first n-1 natural numbers - F(n-1)
2. Express F(n) in terms of F(n-1) and write code: F(n) = F(n-1) + n.

```
int f(int n) {
    int ans = f(n-1);    // Induction Hypothesis step
    return ans + n;    // Solving problem from result in previous step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop.



4. After the dry run, we can conclude that for n equals 1 answer is 1, which we already know. so we'll use this as a base case; hence the final code becomes:

```
int f(int n) {
   if(n == 1) {                    // Base case
      return 1;
   }
   int ans = f(n-1);
   return ans + n;
}
```

This is the main idea to solve recursive problems. To summarize, we will always focus on finding the solution to our starting problem and tell the function to compute rest for us using the particular hypothesis. This idea will be studied in detail in further sections with more examples.

Now, we'll learn more about recursion by solving problems which contain smaller subproblems of the same kind. Recursion in computer science is a method where the solution to the question depends on solutions to smaller instances of the same problem. By the exact nature, it means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So, in other words, in recursion, a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts, and the code is also shorter and easier to understand.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

- **Base case:** A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the recursion depth* will be exceeded and it will throw an error.
- **Recursive call:** The recursive function will recursively invoke itself on the smaller version of the problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is. On the solution of smaller problem, the original problem's solution depends.
- **Small calculation:** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Note: Recursion uses an in-built stack which stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow.

Now, let us see how to solve a few common problems using Recursion.

Example 1: Let's look at the model for a better explanation of the concept...

Problem statement: We want to find out the factorial of a natural number.

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

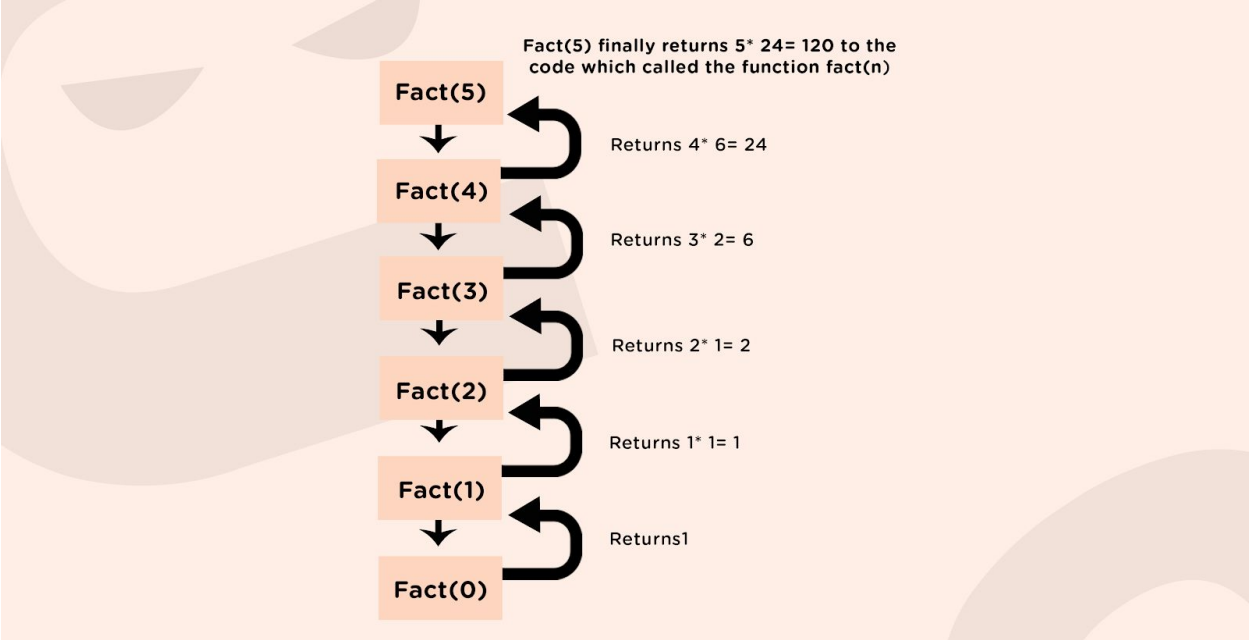
1. **Induction Step:** Calculating the factorial of a number n **- F(n)**

   Induction Hypothesis: We assume we have already obtained the factorial of n-1, through recursion **- F(n-1)**

2. Expressing F(n) in terms of F(n-1): **F(n)=n*F(n-1)**. Thus we get:

```
int f(int n) {
    int ans = f(n-1);                  //Assumption step
    return ans * n;                    //Solving problem from Assumption step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop. Consider n = 5:



As we can see above, we already know the answer of n = 0, that is 1. So we will keep this as our base case. Hence, the code now becomes:

```
#include<iostream>
using namespace std;
int fact(int n) {
        if(n==0) {                     //Base Case
                return 1;
```

```
        }
        int ans = fact(n-1);            // Recursive call
        return n * ans;                 //Small calculation
}

int main() {
          int num;
          cin >> num;
          cout << fact(num);
          return 0;
}
```

Example 2: Let's look at another example of finding the Fibonacci numbers using recursion.

Function for Fibonacci series:

F(n) = F(n-1) + F(n-2),
with F(0) = 0 and F(1) = 1

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

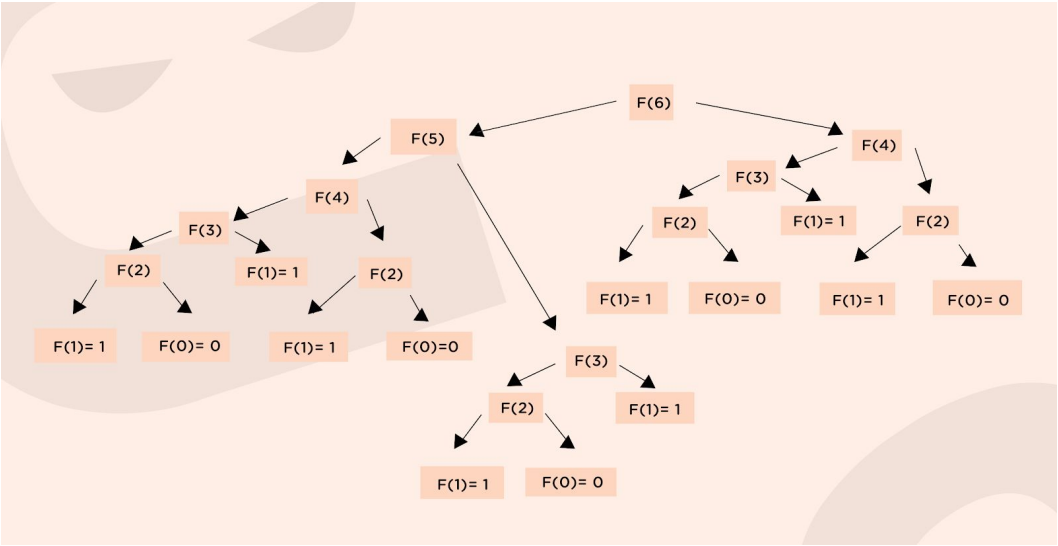
1. **Induction Step:** Calculating the nth Fibonacci number n.

   Induction Hypothesis: We have already obtained the (n-1)th and (n-2)th Fibonacci numbers.

2. Expressing F(n)in terms of F(n-1) and F(n-2): $\mathbf{F_n = F_{n-1} + F_{n-2}}$.

```
int f(int n){
    int ans = f(n-1) + f(n-2);          //Assumption step
    return ans;                         //Solving problem from assumption step
}
```

3. Let's dry run the code for achieving the base case: (Consider n= 6)



From here we can see, that every recursive call either ends at 0 or 1 for which we already know the answer: F(0) = 0 and F(1) = 1; hence using this as our base case in the code below:

```
#include <iostream>
using namespace std;

int fib(int n) {
        if (n == 0) {                   // Base Case for n = 0
                return 0;
        }

        if (n == 1) {                   // Base case for n  = 1
                return 1;
        }

        int smallOutput1 = fib(n - 1);          // Recursive call 1
        int smallOutput2 = fib(n - 2);          // Recursive call 2
        return smallOutput1 + smallOutput2;     // Small calculation
}

int main() {
        cout << fib(6) << endl;                 // Will give output as 8
}
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem statement: We have to tell whether the given array is sorted or not using recursion.

For example: If the array is {2, 4, 8, 9, 9, 15}, then the output should be **YES.**

If the array is {5, 8, 2, 9, 3}, then the output should be **NO.**

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Problem:** Given an array, find whether it is sorted.

   Assumption step: We assume that we have already obtained the answer to the array starting from the index 1. In other words, we assume that we know whether the array (starting from the first index) is sorted.

2. Solving the problem from the results of the "Assumption step": Before going to the assume step, we must check the relation between the first two elements. Find if the first two elements are sorted or not. If the elements are not in sorted order, then we can directly return false. If the first two elements are in sorted order, then we will check for the remaining array through recursion.

```
bool is_sorted(int a[], int size) {
        if (a[0] > a[1]) {                              // Small Calculation
                return false;
        }
        bool isSmallerSorted = is_sorted(a + 1, size - 1);      //Assumption step
        return isSmallerSorted;
}
```

3. We can see that in the case when there is only a single element left or no element left in our array, at that time, the array is always sorted. Let's check the final code now...

```
#include <iostream>
```

```
using namespace std;

bool is_sorted(int a[], int size) {
        if (size == 0 || size ==1) {                    // Base case
                return true;
        }

        if (a[0] > a[1]) {                              // Small calculation
                return false;
        }
        bool isSmallerSorted = is_sorted(a + 1, size - 1);      // Recursive call
        return isSmallerSorted;
}

int main() {
        int arr[ ] = {2, 3, 6, 10, 11};
        if(is_sorted(arr, 5)){
                cout << "Yes" << endl;
        } else {
                Cout << "No" << endl;
        }
}
```

First Index

Let us solve another problem named the **First Index of an array**. The question states that you are given an array of length N and an integer x, you need to find and return the first index (0-based indexing) of integer x present in the array. Return -1, if it is not present in the array. The first index denotes the index of the first occurrence of x in the input array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4

Output: 1 (as 4 is present at 1st index in the array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4

Output: -1 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4

Output: 2 (as 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the first occurrence of the target value, so the answer should be 2).

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[0] == x)
        return 0;
```

Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**.

The recursive call will look like this:

```
rec_call(arr+1, size-1, x)
```

In the recursive call, we are incrementing the pointer and decrementing the size of the array. We have to assume that the answer will come for the recursive call. The answer will come in the form of an integer. If the answer is -1, this denotes that element is not present in the remaining array. If the answer is any other integer (other than -1), then this denotes that element is present in the remaining array. So, if the element is present at the ith index in the remaining array, then it will be present at i+1 in the array. For instance, in the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

However, the base case is still left to be added. The base case for this question can be identified by dry running the case: when you are trying to find an element that is not present in the array. For example: [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero. When the size

of the array becomes zero, then we will return -1. This is because if the size of the array becomes zero, then this means that we have traversed the entire array and we were not able to find the target element. Therefore, this is the **base case step**.

Code:

```
if( size == 0 )
        return -1;
```

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Last index

In this problem, we are given an array of length N and an integer x. You need to find and return the last index of integer x present in the array. Return -1 if it is not present in the array. The last index means: if x is present multiple times in the array, return the index at which x comes last in the array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4
Output: 1 (as 4 is present at 1st position in the array, which is the last and the only place where 4 is present in the given array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4
Output: -1 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4
Output: 4 (as 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the last occurrence of the target value, so the answer should be 4).

Approach:
Now, to solve the question, we have to figure out the following three elements of the solution.

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[0] == x)
        return 0;
```

Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**.

The recursive call will look like this:

```
rec_call(arr+1, size-1, x)
```

In the recursive call, we are incrementing the pointer and decrementing the size of the array. We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer. If the answer is -1, this denotes that element is not present in the remaining array; otherwise, we need to add 1 to our answer as for recursion, it might be the 0^{th} index, but for the previous recursive call, it was the first position. For instance, in the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

However, the base case is still left to be added. The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array. For example: [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero. When the size of the array becomes zero, then we will return -1. This is because if the size of the array becomes zero, then this means that we have traversed the entire array and we were not able to find the target element. Therefore, this is the **base case step**.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

All indices of a number

Here, given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in an array (in increasing order) and return the size of the array.

Case 1: Array = {1, 4, 5, 7, 2}, target value = 4
Output: [1] and the size of the array will be 1 (as 4 is present at 1st position in the array, which is the only position where 4 is present in the given array).

Case 2: Array = {1, 3, 5, 7, 2}, target value = 4
Output: [] and the size of the array will be 0 (as 4 is not present in the array).

Case 3: Array = {1, 3, 4, 4, 4}, target value = 4
Output: [2, 3, 4] and the size of the array will be 3 (as 4 is present at three positions in the array; i.e., [2, 3, 4]).

Now, let's think about solving this problem...

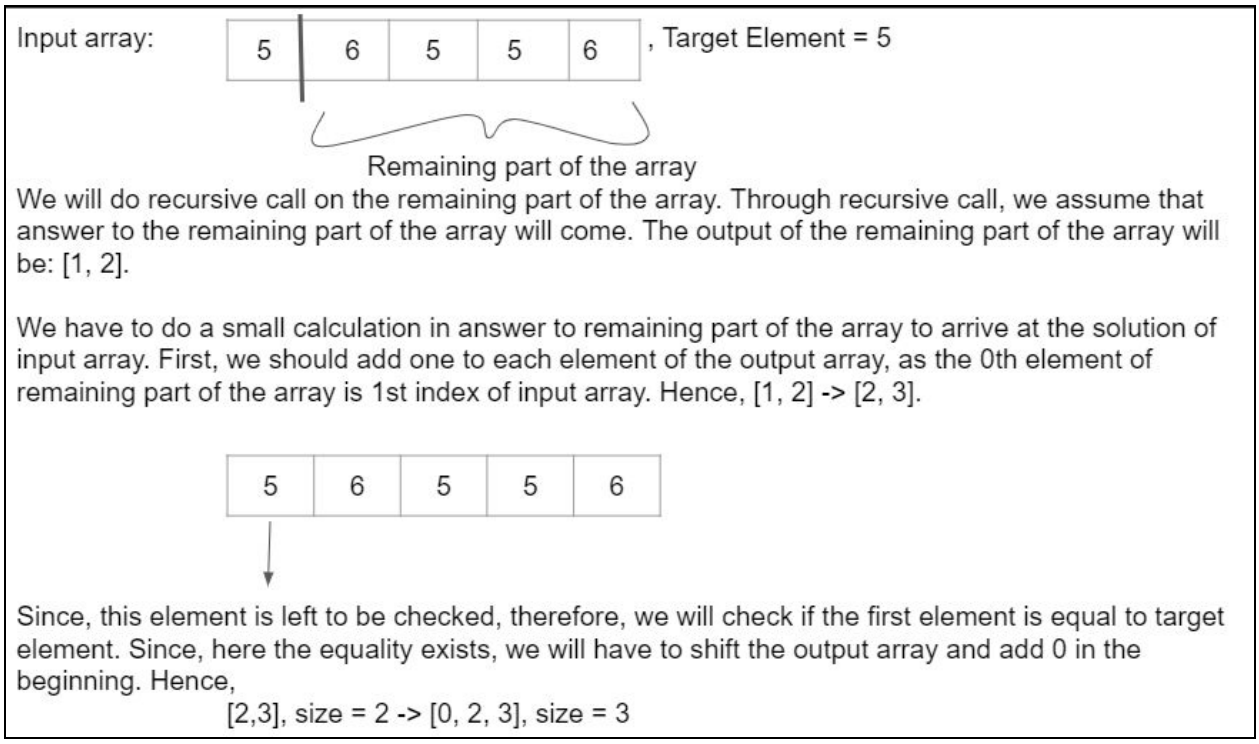
Approach:
Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let us assume the given array is: [5, 6, 5, 5, 6] and the target element is 5, then the output array should be [0, 2, 3] and for the same array, let's suppose the target element is 6, then the output array should be [1, 4].

To solve this question, the base case should be the case when the size of the input array becomes zero. In this case, we should simply return 0, since there are no elements.

The next two components of the solution are Recursive call and Small calculation. Let us try to figure them out using following images:



So, following are the recursive call and small calculation components of the solution:

Recursive Call

Code:

```
int Size = fun(arr + 1, size - 1, x, output);
```

Small Calculation:

1. Update the elements of the output array by adding one to them.
2. If the equality exists, then shift the elements and add 0 at the first index of the output array. Moreover, increment the size, as we have added one element to the output array.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Using the same concept, other problems can be solved using recursion, just remember to apply PMI and three steps of recursion intelligently.

Recursion 2

In this lecture, we are going to understand how to solve different kinds of problems using recursion. It is strictly advised to complete the assignments for the earlier topics, if due.

Recursion and Strings

Here, we are going to discuss the different problems that can be solved using recursion on strings:

- Finding the length of the string

```
#include <iostream>
using namespace std;

int length(char s[]) {
        if (s[0] == '\0') {                          // Base case
                return 0;
        }
        int smallStringLength = length(s + 1);       // Recursive call
        return 1 + smallStringLength;                // Small calculation
}


int main() {
        char str[100];
        cin >> str;

        int l = length(str);
        cout << l << endl;
}
```

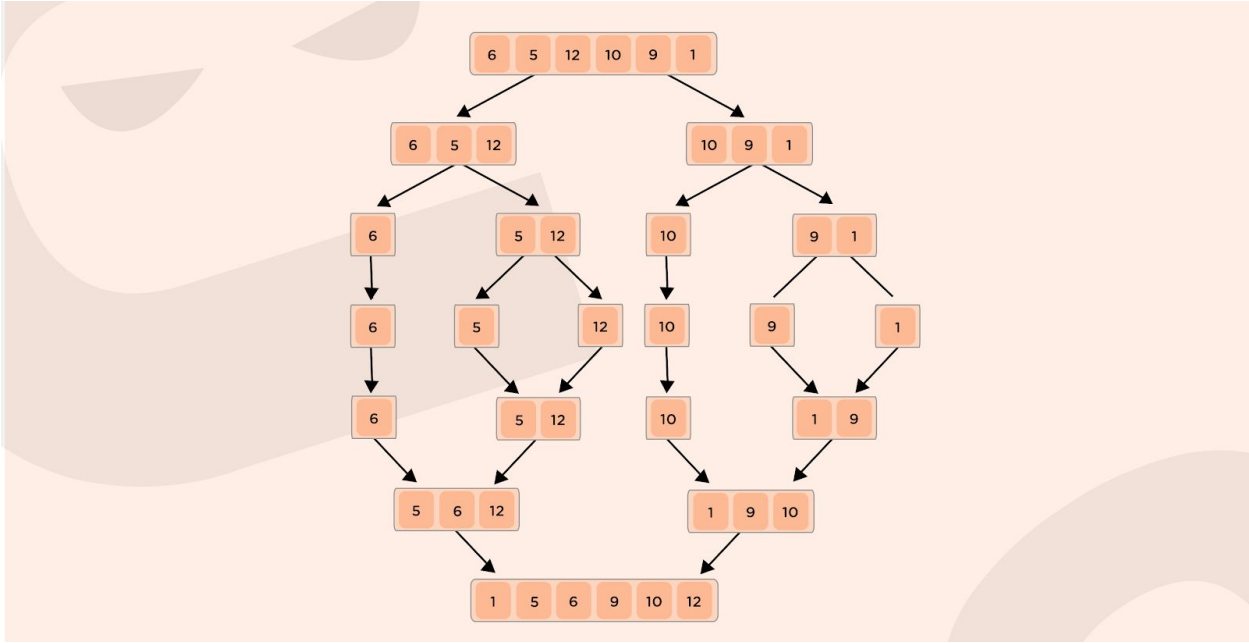
- Remove X from a given string

```
#include <iostream>
using namespace std;

void removeX(char s[]) {
        if (s[0] == '\0') {                          // Base case
                return;
        }

        if (s[0] != 'x') {                           // Small calculation
                removeX(s + 1);                      // Recursive call
        } else {
                int i = 1;
                for (; s[i] != '\0'; i++) {
                        s[i - 1] = s[i];
                }
                s[i - 1] = s[i];
                removeX(s);
        }
}

int main() {
        char str[100];
        cin >> str;
        removeX(str);
        cout << str << endl;
}
```

Sorting Technique

Merge Sort

Consider the example below that shows the working of merge sort over the given array and also how it **divides and conquers** the array.



Let's now look at the algorithm associated with it...

Algorithm for Merge Sort:

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists, keeping the new list sorted too.

Step 1 − If it is only one element in the list it is already sorted, return.

Step 2 − Divide the list recursively into two halves until it can no more be divided.

Step 3 − Merge the smaller lists into new list in sorted order.

Now, you can code it easily by following the above three steps.

It has just one disadvantage and it is that it's **not an in-place sorting** technique, which means it creates a copy of the array and then works on that copy.

Time Complexity : O(nlogn)

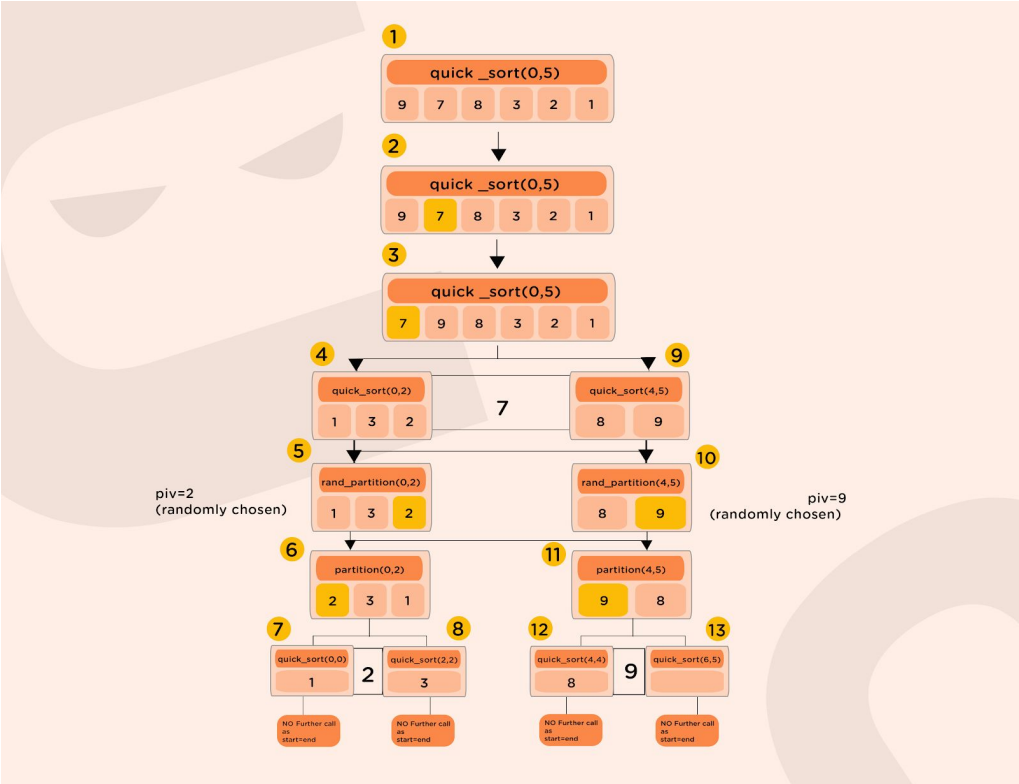
Extra Space Complexity : O(n)

Quick sort

Quick sort is based on the **divide-and-conquer approach** based on the idea of choosing one element as a pivot element and partitioning the array around it, such that: Left side of pivot contains all the elements that are less than the pivot element and Right side contains all elements greater than the pivot.

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

To get a better understanding of the approach consider the following example where a pictorial representation of the quick sort on an array is shown.



Algorithm for Quick Sort:

On the basis of Divide and Conquer approach, quicksort algorithm can be explained as:

- **Divide**

  The array is divided into subparts, taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.

- **Conquer**

  The left and right sub-parts are again partitioned by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.

- **Combine**

This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

Advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, therefore, it is an in-place sorting technique.

Time Complexity : O(nlogn)

Extra Space Complexity : O(1)

Strings

Let us think of string as a class, instead of a data type. We all already know that strings are the character arrays that ends with a NULL character itself.

Syntax for declaring a string:

```
string str;
```

Syntax for declaring a string dynamically:

```
string *str = new string;
```

To take input of the string, we can use the **getline()** function. Newline is the delimiter for the getline() function. Follow the syntax below:

```
getline(cin, str);        // where str is the name of the string
```

You can treat the string like any other character array as well. You can go to any specific index (indexing starts from 0), assign any string value by using equals to (=) operator, etc.

To concatenate two strings you can use '+' operator. For example, to concatenate two strings s1 and s2 and put it in another string str, syntax is as below:

```
string str = s1 + s2;
```

To calculate the length of the string you can use the in-built function (.length()) for that:

```
int len = str.length();
```

You can do the same using the following syntax also:

```
int len = str.size();
```

To extract a particular segment of the string, we can use .substr() function.

```
string s = str.substr(beginning_index, length_ahead_of_starting_index);
```

Here, if you omit the second argument, then automatically the function takes the complete string, after the specified starting index.

You can also find any particular substring in the given string using .find() function. It will return the starting index of the substring to be found in the given string.

```
int index = str.find(name_of_the_pattern_to_be_checked);
```

Note: .find() function finds the first occurring index of the given pattern.

These concepts would be much clearer after studying OOPs.

Let's practice some of the questions over the topics covered in this lecture...

Questions:

Visit the following links for practicing some questions...

- https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/practice-problems/
- https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/practice-problems/
- https://www.techiedelight.com/recursion-practice-problems-with-solutions (On this webpage, visit the subheading **String**)

C++ Foundation with Data Structures

Topic: Time Complexity

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we are mostly concerned about CPU time.

Be careful to differentiate between:

1. Performance: how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

2. Complexity: how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.
Complexity affects performance but not vice-versa.
The time required by a function/method is proportional to the number of "*basic operations*" that it performs.

Here are some examples of basic operations:

- one arithmetic operation (e.g. a+b / a*b)
- one assignment (e.g. int x = 5)
- one condition/test (e.g. x == 0)
- one input read (e.g. reading a variable from console)
- one output write (e.g. writing a variable on console)

Some functions/methods perform the same number of operations every time they are called.
For example, the size function/method of the string class always performs just one operation: return number of Items; the number of operations is independent of the size of the string. We say that functions/methods like this (that always perform a fixed number of basic operations) require constant time.

Other functions/methods may perform different numbers of operations, depending on the value of a parameter. For example, for the array implementation of the Vector/list(Java) class(vector/list classes are implemented similar to the dynamic class we have built), the remove function/method has to move over all of the items that were to the right of the item that was removed (to fill in the gap). The number of moves depends both on the position of the removed item and the number of items in the list. We call the important factors (the parameters and/or fields whose values affect the number of operations performed) the problem size or the input size.

When we consider the complexity of a function/method, we don't really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? double? increase in some other way? For constant-time functions/methods like the size function/method, doubling the problem size does not affect the number of operations (which stays the same).

Furthermore, we are usually interested in the worst case: what is the most operations that might be performed for a given problem size. For example, as discussed above, the remove function/method has to move all of the items that come after the removed item one place to the left in the array. In the worst case, all of the items in the array must be moved. Therefore, in the worst case, the time for remove is proportional to the number of items in the list, and we say that the worst-case time for remove is linear to the number of items in the array. For a linear-time function/method, if the problem size doubles, the number of operations also doubles.

Big-O Notation

We express complexity using big-O notation. For a problem of size N:
a constant-time function/method is "order 1": O(1)
a linear-time function/method is "order N": O(N)
a quadratic-time function/method is "order N squared": O(N^2)
Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time

function/method, which will be faster than a quadratic-time function/method). See below for an example.
Formal definition:

A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:
T(N) <= c * F(N)
The idea is that T(N) is the exact complexity of a function/method or algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e. the actual time/space or whatever for a problem of size N will be no worse than F(N)). In practice, we want the smallest F(N) - the least upper bound on the actual complexity.
For example, consider T(N) = 3 * N^2 + 5. We can show that T(N) is O(N^2) by choosing c = 4 and n_0 = 2. This is because for all values of N greater than 2:

3 * N^2 + 5 <= 4 * N^2
T(N) is not O(N), because whatever constant c and value n0 you choose, I can always find a value of N greater than n_0 so that 3 * N^2 + 5 is greater than c * N.

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
 ...
statement k;
```

The total time is found by adding the times for all statements:
total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

2. if-then-else statements

```
if (condition) {
   sequence of statements 1
}
else {
   sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

3. for loops

```
for (i = 0; i < N; i++) {
   sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

4. Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      sequence of statements
   }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = i+1; j < N; j++) {
      sequence of statements
   }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. the total is $O(N^2)$.

5. **Statements with function/method calls:**

When a statement involves a function/method call, the complexity of the statement includes the complexity of the function/method call. Assume that you know that function/method f takes constant time, and that function/method g takes time proportional to (linear in) the value of its parameter k. Then the statements below have the time complexities indicated.

```
f(k);  // O(1)
g(k);  // O(k)
```

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) {
        g(N);
}
```

has complexity (N^2). The loop executes N times and each function/method call g(N) is complexity O(N).

Best-case and Average-case Complexity

Some functions/methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, consider the add function/method that adds an item to the end of the Vector/list. In the worst case (the array is full), that function/method requires time proportional to the number of items in the Vector/list (because it has to copy all of them into the new, larger array). However, when the array is not full, add will only have to copy one value into the array, so in that case its time is independent of the length of the Vector/list; i.e. constant time.

In general, we may want to consider the best and average time requirements of a function/method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a function/method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant -- the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

Note that calculating the average-case time for a function/method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

When do Constants Matter?

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the same big-O time

complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires 10 * N^2 + N time. For both algorithms, the time is O(N^2), but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms do matter in terms of which algorithm is actually faster.

However, it is important to note that constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires 10*N^2 time, for both algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have different big-O time complexity, the constants and low-order terms only matter when the problem size is small. For example, even if there are large constants involved, a linear-time algorithm will always eventually be faster than a quadratic-time algorithm. This is illustrated in the following table, which shows the value of 100*N (a time that is linear in N) and the value of N^2/100 (a time that is quadratic in N) for some values of N. For values of N less than 104, the quadratic time is smaller than the linear time. However, for all values of N greater than 104, the linear time is smaller.

N	**100*N**	**N^2/100**
10^2	10^4	10^2
10^3	10^5	10^4
10^4	10^6	10^6
10^5	10^7	10^8
10^6	10^8	10^{10}
10^7	10^9	10^{12}

C++ Foundation with Data Structures

Topic : Object Oriented Programming - 1

What is Object Oriented Programming?

Object Oriented Programming could be best understood with help of an example. Consider a library management system. Using procedural programming, the problem will be viewed in terms of working happening in the library i.e., issuing of book, returning the book, adding new book etc. The OOP paradigm, however aims at the objects and their interface. Thus in OOP, library management problem will be viewed in terms of the objects involved. Objects are real world entities around which the system revolves. The objects involved are: librarian, book, member etc. Hence, the object-oriented approach views a problem in terms of objects involved rather than procedure for doing it.

Why use Object Oriented Programming?

In real life we deal with lot of objects such as people, car, account, etc. Hence, we need our software to be analogous to real-world objects. Real world objects have data-type properties such as name, age for people, model name for car and balance for account, etc. Moreover, real-world objects can also do certain things such as people talk, cars move, account accumulates, etc. We want our code to mimic the way these objects behave and interact. Hence, Object Oriented Programming allows the program to be closer to real world and thereby making it less complex. Also it makes the software reuse feasible and possible, for example, we don't want to define a student every time we use it, hence using OOP, we just create the blueprint for the student object and use it whenever required.

Classes and objects

The class is a single most important C++ feature that implements OOP concepts and ties them together. Classes are needed to represent real-world entities. A class is a way to bind the data describing an entity and its associated functions together. For instance, consider a ***user*** having characteristics ***username*** and ***password*** and some of its associated operations are ***sign up***, ***sign in*** and ***logout***.

Class is just a template, which declares and defines characteristics and behavior, hence we need to declare objects of the class for it to be usable. In other words class represents a group of similar objects.

Data members and member functions

Data members are the data-type properties that describe the characteristics of the class.

Member functions are the set of operations that may be applied to objects of that class, i.e., they represent the behavioral aspect of the object. They are usually referred as class interface.

Syntax for definition of a class :

```
class class_name {
        Access Modifier:
                Data members
                Member functions
};
```

The class body contains the declaration of members (data and functions).

Declaring objects of a class

We can declare objects of a class either statically or dynamically just as we declare variables of primitive data types.

Statically

Syntax for declaring objects of a class statically is:

```
class_name    object_name;
```

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student s1;       // Declaration of object s1 of type Student
    Student s2;       // Declaration of object s2 of type Student
}
```

Dynamically

Syntax for declaring objects of a class dynamically is:

class_name *object_name = new class_name

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student *s1 =new Student;   // Declaration of object of type Student
                                // dynamically
}
```

Access Modifiers

The class body contains the declaration of its members (data and functions). They are generally two types of members in a class: private and public, which are correspondingly grouped under two sections namely private: and public: .

The **private members** can be accessed only from within the class. These members are hidden from the outside world. Hence they can be used only by member functions of the class in which it is declared.

The **public members** can be accessed outside the class also. These can be directly accessed by any function, whether member function of the class or non-member function. These are basically the public interface of the class.
By default, members of a class are private if no access specifier is provided.

Example code :

```
class A {
    int x, y;
    int sqr(){
            return x * x;
    }
    public :
            int z;
            int twice(){
                    return 2 * y;
            }
```

```
        int test(int i){
                int q = sqr( );       // private function being
        invoked                                  // by member function
                return q + i;
        }
};
int main() {
    A obj;
    obj.z = 10;              // valid. z is a public member
    obj.x = 4;               // Invalid. x is a private member and hence can
                             // be   accessed only by member functions
                            // not  directly by using object
    int j = obj.twice( );   // valid. twice( ) is a public member function
    int k = obj.sqr( );     // Invalid. sqr() is a private member function
    int l = obj.test( );    // valid. test is a public member function
}
```

Getter and setters

The private members of the class are not accessible outside the class, although sometimes there is a necessity to provide access even to private members, in these cases we need to create functions called getters and setters. Getters are those functions that allow us to access data members of the object. However, these functions do not change the value of data members. These are also called accessor function.

Setters are the member functions that allow us to change the data members of an object. These are also called mutator function.

Example code :

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( ){
            return rollno;
        }
        int getMarks( ){
```

```
        return marks;
    }

    void setGrade( ){
        if (marks > 90) grade ='A';;
        else if (marks > 80) grade = 'B';
        else if (marks > 70) grade = 'C';
        else if (marks > 60) grade = 'D';
        else grade = 'E';
    }
};
```

getRollno( ) and getMarks( ) are getter functions and setGrade( ) is a setter function.

Defining Member functions outside the class

We can also define member functions outside the class using scope resolution operator :: .
For example lets move the definition of the two functions defined in student class above outside the class.

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( );

        int getMarks( );

        void setGrade( ){
            if (marks > 90) grade ='A';;
            else if (marks > 80) grade = 'B';
            else if (marks > 70) grade = 'C';
            else if (marks > 60) grade = 'D';
            else grade = 'E';
        }
};
```

```
int  Student::getMarks(){
  return marks;
}

int  Student::getRollNo(){
  return rollNo;
}
```

We can access member functions in similar manner via an object of class Student and using dot operator.

Constructors

A constructor in a class is means of initializing or creating objects of a class. A constructor allocates memory to the data members when an object is created. It may also initialize the object with legal initial value.
A constructor has following characteristics:

- Constructor is a member function of a class and has same name as that of the class.
- Constructor functions are invoked automatically when the objects are created.
- Constructor functions obey the usual access rules. That is, private constructors are available only for member functions, however, public constructors are available for all functions
- Constructor has no return type, not even void.

Default constructor

A constructor that accepts no parameter is called default constructor. The compiler automatically supplies a default constructor implicitly. This constructor is the public member of the class. It allocates memory to the data members of the class and is invoked automatically when an object of that class is created. Having a default constructor simply means that a program can declare instances of the class.

Example code :

```
class Sum {
    int a, b;
    public :
            int getSum(){
```

```
            return a + b;
        }
} ;
int main() {
    Sum obj;       //implicit default constructor invoked
}
```

Whenever an object of person class is created implicit default constructor is invoked automatically that assigns memory to its data members, i.e., ***name*** and ***age.***

- **Creating your own default constructor**

One can define their own default constructor. When a user-defined default constructor is created, the compiler's implicit default constructor is overshadowed.

Example code :

```
class Sum {
    int a, b;
    public :
        Sum( ) {              // user-defined default constructor
            cout << "constructor invoked";
            a = 10;
            b = 20;
        }
        int getSum(){
            return a + b;
        }
} ;
int main() {
    Sum obj;                  //explicitly defined default constructor invoked
}
```

Output :

constructor invoked

In this case, user-defined default constructor will be invoked when an object **obj** of person class is created and the data members of that object, ***a*** and ***b***, are initialized to default values 10 and 20.

Parameterized constructor

The constructors that can take arguments are called parameterized constructor.

Example code :

```
class Sum {
    int a, b;
    public :
            Sum(int num1, int num2 ) {          // parameterized constructor
                    a = num1;
                    b = num2;
            }
            int getSum(){
                    return a + b;
            }
 } ;
 int main() {
     Sum obj(4, 2);                  //parametrized constructor invoked
  }
```

Declaring a constructor with arguments hides the default constructor. Hence, the object declaration statement such as

Person obj;

may not work. It is necessary to pass the initial value arguments to the constructor function when an object is declared. This can be done in two ways :

1. By calling constructor explicitly.
   Sum obj = Sum(4 ,2) ;
2. By calling constructor implicitly.(as illustrated in example code)
   Sum obj(4, 2) ;

Destructors

Just as the objects are created, so are they destroyed. If a class can have constructor to set things up, it should have a destructor to destruct the objects. A destructor as the name itself suggests, is used to destroy the objects that have been created by a constructor. A destructor is also a member function whose name is the same as the class name but preceded by tilde ('~'). For instance, destructor for class Sum will be ~Sum( ).

A destructor takes no arguments, and no return types can be specified for it, not even void. It is automatically called by the compiler when an object is destroyed. A destructor frees up the memory area of the object that is no longer accessible.

Example code:

```
class Sum {
    int a, b;
    public :
        Sum(int num1, int num2 ) {          // parameterized constructor
            cout << "Constructor at work" << endl;
            a = num1;
            b = num2;
        }
        ~Sum( ){                            //destructor
            cout<< "Destructor at work" << endl;
        }
        int getSum(){
            return a + b;
        }
}
int main() {
    Sum obj(4, 6);
 }
```

Output :

Constructor at work
Destructor at work

As soon as obj goes out of scope, destructor is called and obj is destroyed releasing its occupied memory.

NOTE:

- If we fail to define a destructor for a class, the compiler automatically generates one for static allocations.
- Destructors are invoked in the reverse order in which the constructors were called.
- Only the function having access to the constructor and destructor of a class, can define objects of this class types otherwise compiler reports an error.

this keyword

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which this function was called. For example, the function call for obj.getSum() will set the

pointer **this** to the address of the object obj. This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

Example code:

```
class Sum {
    int a, b;
    public :
            Sum(int a, int b ) {
                    this->a = a;
                    this->b = b;
            }
            int getSum(){
                    return a + b;
            }
 }
```

In the constructor of the Sum class, since the data members and data members have the same name, this keyword is used to differentiate between the two. Here, ***this->a*** refers to the data member ***a*** of the object obj.

OOPs 2

Shallow and Deep Copy

Shallow Copy:

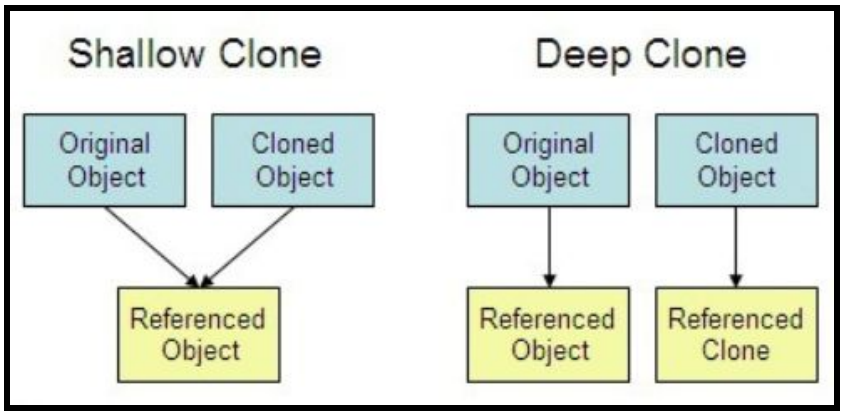
It copies all of the member field values. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred.

Note: Assignment operator and the default copy constructor makes a shallow copy.

Deep Copy:

It copies all the fields but creates a copy of dynamically allocated memory pointed to by the fields. Here, we need to make our copy constructor and overload the assignment operator. Advantages of the deep copy are:

- When we need to initialise the class variables to some value or NULL, then a parameterized constructor can be used.
- A destructor that can be used to delete the dynamically allocated memory.



Kindly refer to the code below:

```
class Student {
```

```
        int age;
        char *name;
        public :
        Student(int age, char *name) {   // parameterized constructor
                this -> age = age;
                // Shallow copy
                // this -> name = name;

                // Deep copy
                this -> name = new char[strlen(name) + 1];  // Created a new memory
                strcpy(this -> name, name);
        }
};
```

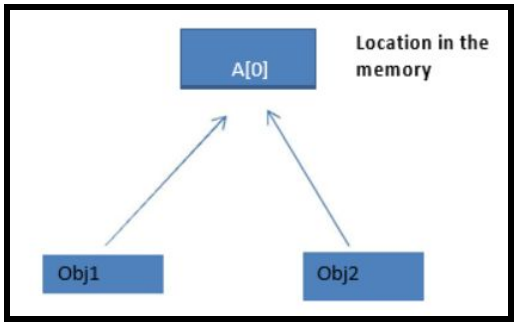
Copy Constructor

The copy constructor is used to create a copy of an already existing object of class type. We can make this copy in two ways:

Shallow copy constructor:

Let's take an example to understand it better. Suppose, two students are entering their details simultaneously from two different machines that are connected over the same network. It will reflect the changes made by both of them in the database because they both are entering their details in the same database.

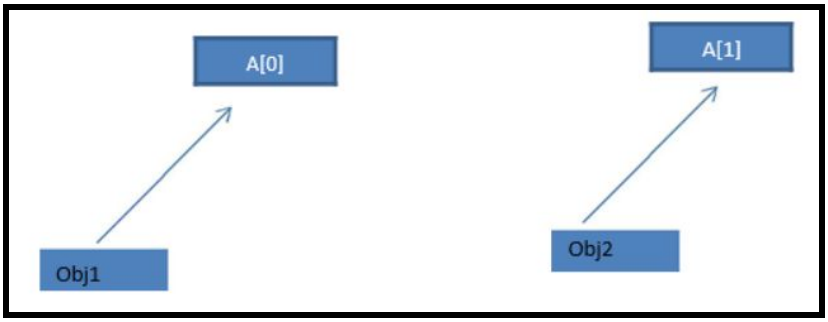
The same is the case with the shallow copy as both share the same locations. It copies references to the original object. **Default copy constructor** uses the same way of copying. It is generally preferred in the cases when the class does not contain any dynamically allocated memory.



Deep copy constructor:

Suppose you have to submit the homework but are short of time. So, you decide to copy the same from your friend. Now, you and your friend have the same work but possess different copies. Now, you choose to modify yours's a bit to prevent the risk of the teacher figuring it out. But the changes you did will not affect your friend's work. Deep copy follows the same concept.

Deep memory allocates different memory for the copied task. It is generally used when we assign the dynamic memory using pointers.



Kindly refer to the code below:

```
// Copy constructor
Student(Student const &s) {      // keyword const assured us that original copy is
                                 // unchanged
        this -> age = s.age;
        // this -> name = s.name;          // Shallow Copy

        // Deep copy
        this -> name = new char[strlen(s.name) + 1];
        strcpy(this -> name, s.name);
}
```

Initialisation list

Using the initialisation list, we can directly assign the values to the data members of the class. Though we can use the initialisation list as needed, in the following cases, we mandatorily need to use the same:

- When no base class default constructor is present
- When the data members are of type **const**
- When the data member and parameter have the same name
- When the data member of the reference type is used.

The syntax begins with a colon(:) and then each variable along with its value separated by a comma. It does not end in a semicolon.

Kindly refer to the code below for better understanding:

```
class Student {
        public :

                int age;
                const int rollNumber;     // Const type data member
                int &x;                   // age reference variable
                // Parameterised list is used
                Student(int r, int age) : rollNumber(r), age(age), x(this -> age) {
                        //rollNumber = r;
                }
};
```

Constant functions

Constant functions are those which don't change any property of current objects. Only constant objects of the class could invoke these.

Syntax:

```
datatype function_name const();
```

Static Members

Suppose, we have a student database and we are creating objects for each student specifying their name, roll number and school name. As discussed earlier, deep copy constructor will be used for the same. For each student, their names and roll number can be different so we will be allotting distinct memories to each. But the school name should be the same. By using a deep copy, we will allocate each student a different memory for the school name. To overcome this, we will be using the **static** keyword.

Syntax for declaring static members:

```
static datatype variable_name;
```

Since it is a property of the class and not of an object, we can't simply access these values from the object name followed by the variable name.

Syntax for invoking static members:

```
Class_name :: variable_name;
```

Note: (::) is called the Scope Resolution Operator.

Apart from variables, functions can also be made static. By declaring a function as static, we make it independent of any particular class object. A static function can be invoked even if no objects of the class exist and the static functions are accessed using only the class name and the (::) operator.

Kindly refer to the code below for better understanding:

```
#include <iostream>
using namespace std;

class Student {
        static int totalStudents;          // total number of students present

        public :
        int rollNumber;
```

```
        int age;

        Student() {
                totalStudents++;
        }

        int getRollNumber() {
                return rollNumber;
        }

        static int getTotalStudent() {          // Static member function
                return totalStudents;
        }
};
int Student :: totalStudents = 0;   // initialize static data members

int main() {
        Student s1;
        Student s2;
        Student s3, s4, s5;

        cout << Student :: getTotalStudent() << endl;  // static function invoked
}
```

Operator Overloading

In C++, we can specify more than one definition of an operator in the same scope, which is called operator overloading. It makes the program more intuitive. Example, if we want to add two fraction numbers, then we can do the same by calling over the numerator and denominator of both fraction object F1 and F2. Using operator Overloading, it can be simply done by F1 + F2. You can see that we are adding two class objects, which is primitively not possible but is possible using operator overloading of '+' operator.

Syntax:

```
return_type operator symbol_to_be_overloaded(arguments){}
```

Here, **operator** is a keyword.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).
- Operators = and & are already overloaded in C++, so we can avoid overloading them.
- Precedence and associativity of operators remain intact.

List of operators that can be overloaded in C++:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

Kindly refer to the code below:

```
class Fraction {
        private :
                int numerator;
                int denominator;

        public :
                Fraction(int numerator, int denominator) {
                        this -> numerator = numerator;
                        this -> denominator = denominator;
```

```
    }

    void print() {
        cout << this -> numerator << " / " << denominator << endl;
    }

    void simplify() {
        int gcd = 1;
        int j = min(this -> numerator, this -> denominator);
        for(int i = 1; i <= j; i++) {
            if(this -> numerator % i == 0 && this -> denominator % i
== 0) {
                gcd = i;
            }
        }
        this -> numerator = this -> numerator / gcd;
        this -> denominator = this -> denominator / gcd;
    }

    Fraction add(Fraction const &f2) {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator+(Fraction const &f2) const {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator*(Fraction const &f2) const {
        int n = numerator * f2.numerator;
        int d = denominator * f2.denominator;
        Fraction fNew(n, d);
```

```
                fNew.simplify();
                return fNew;
        }

        bool operator==(Fraction const &f2) const {
                return (numerator == f2.numerator && denominator ==
f2.denominator);
        }

        void multiply(Fraction const &f2) {
                numerator = numerator * f2.numerator;
                denominator = denominator * f2.denominator;
                simplify();
        }

        // Pre-increment
        Fraction& operator++() {
                numerator = numerator + denominator;
                simplify();

                return *this;
        }
        // Post-increment
        Fraction operator++(int) {
                Fraction fNew(numerator, denominator);
                numerator = numerator + denominator;
                simplify();
                fNew.simplify();
                return fNew;
        }
        // Short-hand addition operator overloaded
        Fraction& operator+=(Fraction const &f2) {
                int lcm = denominator * f2.denominator;
                int x = lcm / denominator;
                int y = lcm / f2.denominator;

                int num = x * numerator + (y * f2.numerator);

                numerator = num;
                denominator = lcm;
                simplify();

                return *this;
        }
};
```

Dynamic array class

Let us now implement a dynamic array class where we will be creating functions to add an element to the array, extract the array element using the provided index and print the complete array. Kindly refer to the code below:

```
class DynamicArray {
        int *data;
        int nextIndex;
        int capacity;                   // total size

        public :
        DynamicArray() {
                data = new int[5];          // Starting with capacity  = 5
                nextIndex = 0;
                capacity = 5;
        }

        DynamicArray(DynamicArray const &d) {
                //this -> data = d.data;                // Shallow copy

                // Deep copy
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }
       // operator overloading: We are simply equating two arrays
        void operator=(DynamicArray const &d) {
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }

       // inserting a new element to the array
        void add(int element) {
                if(nextIndex == capacity) {         // If the capacity is not enough
                        int *newData = new int[2 * capacity];   // We doubled the
                                                                // capacity of the array
```

```
                for(int i = 0; i < capacity; i++) {
                        newData[i] = data[i];    // Elements copied to the array
                }
                delete [] data;
                data = newData;
                capacity *= 2;
        }
        data[nextIndex] = element;
        nextIndex++;
    }

    int get(int i) const {     // For returning the element at particular index
        if(i < nextIndex) {
                return data[i];
        }
        else {
                return -1;
        }
    }

    void add(int i, int element) {
                if(i < nextIndex) {
                        data[i] = element;
                }
                else if(i == nextIndex) {
                        add(element);
                }
                else {
                        return;
                }
    }

    void print() const {                    // For printing the complete array
        for(int i = 0; i < nextIndex; i++) {
                cout << data[i] << " ";
        }
        cout << endl;
    }
};
```

Linked List 1

Data Structures

Data structures are just a way to store and organise our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked list is a part of them.

Introduction to linked list

Let's first discuss what are the disadvantages of using an array:

- Arrays have a fixed size that is required to be initialised at the time of declaration i.e., the addition of extra elements would throw an error (Index out of range) in C++.
- Arrays could store a maximum of 10^6 - 10^7 elements . If we try to store more elements, then the program might lead to memory overflow.
- Even deletion of elements from the array is an expensive task as it would require the complete shift of further elements by some positions to the left as the data is stored in the form of contiguous memory blocks.

In order to overcome these problems, we can use linked lists...

- A linked list is a linear data structure where each element is a separate object.
- Each element or node of a list is comprising of two items:
    - Data
    - Pointer(reference) to the next node.
- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as head.
- The last node of a linked list is known as tail.
- The last node has a reference to null.

Linked list class

```
class Node {
        public :
        int data;                       // to store the data stored
        Node *next;                     // to store the address of next pointer

        Node(int data) {
                this -> data = data;
                next = NULL;
        }
};
```

Note: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and is, if lost, would lead to losing of the list.

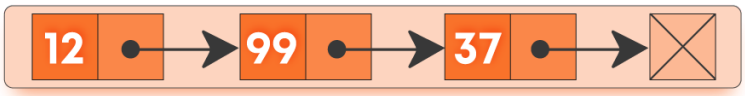
Printing of the linked list

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the NULL pointer which will always be the tail pointer. Follow the code below:

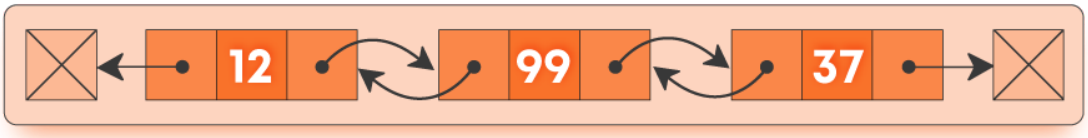
```
void print(Node *head) {
      Node *tmp = head;
      while(tmp != NULL) {
                  cout << tmp->data << " ";
                  tmp = tmp->next;
      }
      cout << endl;
}
```

There are generally three types of linked list:

- **Singly:** Each node contains only one link which points to the subsequent node in the list.



- **Doubly:** It's a two-way linked list as each node points not only to the next pointer but also to the previous pointer.



- **Circular:** There is no tail node i.e., the next field is never NULL and the next field for the last node points to the head node.

Let's now move onto all other operations like insertion, deletion, input in linked list...

Taking input in list

Refer the code below:

```
Node* takeInput() {
        int data;
        cin >> data;
        Node *head = NULL;
        Node *tail = NULL;
        while(data != -1) {                                    // -1 is used for terminating
                Node *newNode = new Node(data);
                if(head == NULL) {
                        head = newNode;
                        tail = newNode;
                }
                else {
                        tail -> next = newNode;
                        tail = tail -> next;
                        // OR
                        // tail = newNode;
                }

                cin >> data;
        }
        return head;
}
```

To take input in the user, we need to keep few things in the mind:

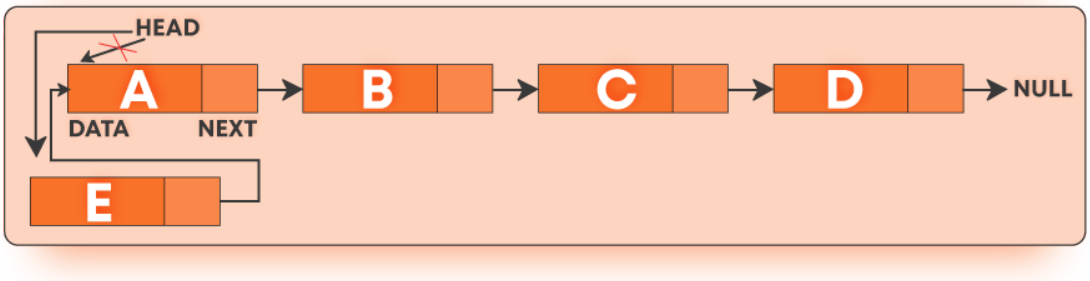
- Always use the first pointer as the head pointer.
- When initialising the new pointer the next pointer should always be referenced to NULL.
- The current node's next pointer should always point to the next node to connect the linked list.

Insertion of node

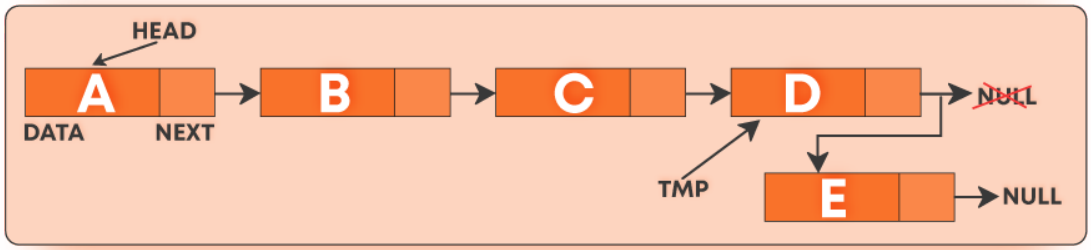
There are 3 cases:

- **Case 1: Insert node at the last**

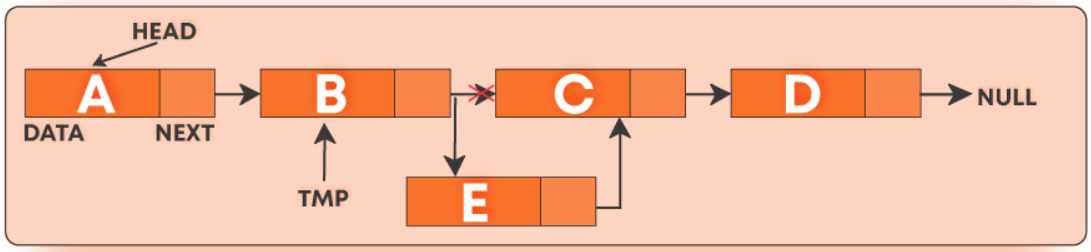
  This can be directly done by normal insertion as discussed above.



- **Case 2: Insert node at the beginning**
  - First-of-all store the head pointer in some other pointer.
  - Now, mark the new pointer as the head and store the previous head to the next pointer of the current head.
  - Update the new head.



- **Case 3: Insert node anywhere in the middle**
    - For this case, we always need to store the address of the previous pointer as well as the current pointer of the location at which new pointer is to be inserted.
    - Now let the new inserted pointer be curr. Point the previous pointer's next to curr and curr's next to the original pointer at the given location.
    - This way the new pointer will be inserted easily.



Let's now check out the code for all the above cases...

```
Node* insertNode(Node *head, int i, int data) {
        Node *newNode = new Node(data);
        int count = 0;
```

```
        Node *temp = head;

        if(i == 0) {                                        //Case 2
                newNode -> next = head;
                head = newNode;
                return head;
        }

        while(temp != NULL && count < i - 1) {              //Case 3
                temp = temp -> next;
                count++;
        }
        if(temp != NULL) {
                Node *a = temp -> next;
                temp -> next = newNode;
                newNode -> next = a;
        }
        return head;                    //Returns the new head pointer after insertion
}
```

Deletion of node

There are 2 cases:

- **Case 1: Deletion of the head pointer**

  In order to delete the head node, we can directly remove it from the linked list by pointing the head to the next.

- **Case 2: Deletion of any node in the list**

  In order to delete the node from the middle/last, we would need the previous pointer as well as the next pointer to the node to be deleted. Now directly point the previous pointer to the current node's next pointer.

Try to code it yourself and then check for the solution provided against the corresponding problem in the course curriculum.

Now, let's move on to the recursive approach for insertion and deletion of the node in a linked list.

Insert node recursively

Follow the steps below and try to implement it yourselves:

- If Head is null and position is not 0. Then exit it.
- If Head is null and position is 0. Then insert a new Node to the Head and exit it.
- If Head is not null and position is 0. Then the Head reference set to the new Node. Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or end.

For the code, refer to the Solution section of the problem...

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is root, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop position-1 times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem...

You will find more practice problems from other sources in the Linked List 2 module.

Linked List 2

Now moving further with the topic, let's try to solve some problems now...

Here we are only gonna discuss the approach, but you will have to implement it yourselves and in case you are stuck then refer to the solution section of the corresponding problem.

Midpoint of LL

Midpoint of a linked list can be found out very easily by taking two pointers. One named **slow** and the other named **fast**. As their names suggest, they will move in the same way respectively. Fast pointer will move ahead two pointers at a time, while the slower one will move at a speed of a pointer at a time. In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.

They will move like these:

slow = slow -> next;

fast = fast -> next -> next;

Also, be careful with the even length scenario of the linked lists. For odd length there will be only one middle element, but for the even length there will be two middle elements. The above approach will return the first middle element and the other one(in case of even length list) will be the direct next of the first middle element.

Merge Two sorted linked lists

We will be merging the linked list, similar to the way we performed merge over two sorted arrays.

We will be using the two head pointers, compare their data and the one found smaller will be directed to the new linked list and increase the head pointer of the corresponding linked list. Just remember to maintain the head pointer separately for the new sorted list. And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list. Now try to code it...

Mergesort over linked list

Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list. just the difference is that in case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach and merging part of the divided lists can also be done using the merge sorted linked lists code as discussed above. Basically the functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.

Reverse the linked list

Recursive approach:

Basically, we will store the last element of the list in the small answer, and then update that by adding the next last node and so on. Finally, when we will be reaching the first element, we will assign the **next** to NULL. Follow the code below, for better understanding...

```
Node* reverseLL(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                return head;
        }

        Node *smallAns = reverseLL(head -> next);           // Recursive call

        Node *temp = smallAns;                              // small answer that
        while(temp -> next != NULL) {                       // stores the reversed
                temp = temp -> next;                        // list by traversing the
        }                                                   // reversed list and
                                                            // then appending the
        temp -> next = head;                                // next element to it.
        head -> next = NULL;
        return smallAns;
}
```

After calculation you can see that this code has a time complexity of O(n^2). Now let's think on how to improve it...

There is another recursive approach in which we can simply use the O(n) approach. What we will be doing is creating a pair class that will be storing the reference of not only the head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing. Checkout the code for your reference...

```
class Pair {
        public :                        //Pair class about which we were talking above
                Node *head;
                Node *tail;
};

Pair reverseLL_2(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                Pair ans;
                ans.head = head;
                ans.tail = head;
                return ans;
        }

        Pair smallAns = reverseLL_2(head -> next);          //Recursive call

        smallAns.tail -> next = head;                       // you can see that the time
        head -> next = NULL;                                // is reduced as we do
        Pair ans;                                           //not need to find the tail
        ans.head = smallAns.head;                           // pointer each time
        ans.tail = head;
        return ans;
}
```

Now improving this code further...

A simple observation is that the tail is always the head->next. By making the recursive call we can directly use this as our tail pointer and reverse the linked list by tail->next = head.

Refer to the code below...

```
Node* reverseLL_3(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                return head;
        }
        Node *smallAns = reverseLL_3(head -> next);         //Recursive call

        Node *tail = head -> next;                          //Small calculation
        tail -> next = head;                                //discussed above
        head -> next = NULL;
        return smallAns;
}
```

Iterative approach:

We will be using three-pointers in this approach: **previous, current and next.** Initially, the previous pointer would be NULL as in the reversed linked list, we want the original head to be the last element pointing to NULL. Current pointer will be the current node whose next will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element. Similarly, iteratively, we will keep updating the pointers as current to the next, previous to the current and next to current's next.

You will be solving this problem yourself now...

Variations of the linked list

In the lecture notes of Linked list - 1, we have already seen the three different types of linked list and discussed them diagrammatically also. Prefer to that section for the reference...

Practice problems

Try over the following link to practice some good questions related to linked lists:

https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=linked-lists

Abstract Data Type (ADT)

Data Abstraction & ADT

When you switch on the fan and it starts rotating at its desired speed, you don't care about the process of its internal working. Fan works unless there is a power cut. This real-world example highlights the programming concept of **data abstraction**, which allows a programmer to protect/hide the implementation of a process and only give the "keys" to other functions or user data. Similarly, in the case of programming when a user tries to find out some information, it is provided to him/her without showing the internal working of the functions in our program.

Data types created using the same concept are known as abstract data types. An **abstract data type** (or ADT) is a class that has a defined set of operations and values. If we consider the above example, then making the fan's switch as the entire abstract data type will prevent the user from knowing all of its internal working.

Features of ADT

- Prevents users from knowing the working of the program.
- Provides only specific data asked for if it's asking for an accurate set of data.

Methods of Abstraction

There are mainly three ways of data abstraction:

Procedural Abstraction

- Performs already stored repeated tasks, if asked for.
- Follows a fixed procedure of instructions provided.
- Generally, used in programs by invoking them. For example: While implementing a square root function in C++, we prefer to directly call the in-built **sqrt()** function and provide arguments to it, but we are not allowed to view the complete program written for this function.
- **Disadvantage:** Data is public, hence could be modified by any programmer using it.

Modular (File) Abstraction

- Contains both public and private sections and hence, can be used as per requirements.
- **Disadvantage:** Data can either be public or private at a single time.

Object-Oriented Programming

- The most efficient method of data abstraction.
- Data and procedures are instantiated together within an object.
- Multiple instantiations possible overcoming the disadvantage of modular abstraction.
- Accessing data can be controlled by the programmer.
- The program becomes more robust and readable using it.

Templates

Templates are a good way of making classes more abstract by letting you define the behavior of the class without actually knowing what data type will be handled by the operations of the class. The advantage of having a single class - that can handle several different data types is that it makes the code easier to maintain, and it makes classes more reusable.

Syntax for one type of variable only:

```
template <typename T>
```

Syntax for two types of variables:

```
template <typename T, typename V>
```

For more than two types of variables, more type names can be added within the angular brackets.

For Example: You want to create a pair class and use it for different types of variables, like int, char, double etc... Follow the code below:

```
template <typename T>

class Pair {
        T x;            //variables x and y are declared of type T which
        T y;            //could take place of any known data type

        public :

        void setX(T x) {
                this -> x = x;
        }

        T getX() {
                return x;
        }
```

```
        void setY(T y) {
                this -> y = y;
        }

        T getY() {
                return y;
        }
};
```

Now suppose you want to create *x* of one type (like int) and *y* of another type (for eg. double), then you can do it in the following way:

```
template <typename T, typename V>

class Pair {
        T x;          //variables x and y are declared of type T and V respectively,
        V y;          //which could take place of any known data types(different/same)

        public :

        void setX(T x) {
                this -> x = x;
        }

        T getX() {
                return x;
        }

        void setY(V y) {
                this -> y = y;
        }

        V getY() {
                return y;
        }
};
```

To use these in the main() function, simply call the class like this:

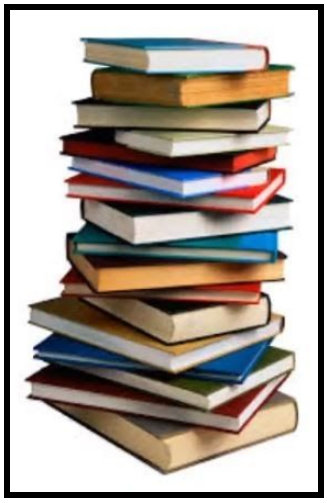
```
Pair<int, double> p1;
```

and the pair of the required types will be created.

Stacks

Introduction

- It is also a linear data structure like arrays and linked lists.
- It is an abstract data type(ADT).
- This is also known as recursion type data structure.
- It follows LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:



Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- **Insertion**: This is known as **push** operation.
- **Deletion**: This is known as **pop** operation.
- **Access to the topmost element:** This operation is performed by the **top** function.
- **Size of the stack:** This operation is performed by the **size** function which returns the number of elements in the stack.
- **To check for filled/empty stack:** This operation is performed by **isEmpty** function which returns boolean value. It returns true for empty stack and false otherwise.

Implementing stack using array

As in the arrays we can access any elements but here we want to use it as a stack which means we should be only allowed to access the topmost element of it. Hence, to use an array as a stack we will be keeping the access to the former as **private** in the class. This way we can put restrictions on the various operations discussed above.

Check the code below for better understanding of the approach (follow-up in the comments)...

```
#include <climits>
class StackUsingArray {
        //Privately declared
        int *data;                          // Dynamic array created serving as stack
        int nextIndex;                      // To keep the track of current top index
        int capacity;                       // To keep the track of total size of stack

        public :
        StackUsingArray(int totalSize) {    //Constructor to initialise the values
                data = new int[totalSize];
                nextIndex = 0;
```

```
        capacity = totalSize;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }

    bool isEmpty() {
        /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        }
        */

        return nextIndex == 0;        //Above program written in short-hand
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            cout << "Stack full " << endl;
            return;
        }
        data[nextIndex] = element;
        nextIndex++;                  //Size incremented
    }
    // delete element
    int pop() {
//Before deletion we checked if it was initially not empty to prevent underflow
        if(isEmpty()) {
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        nextIndex--;                  //Conditioned satisfied so deleted
        return data[nextIndex];
    }
   //to return the top element of the stack
    int top() {
        if(isEmpty()) {               // checked for empty stack to prevent overflow
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        return data[nextIndex - 1];
```

```
    }
};
```

Dynamic stack

There is one limitation to the above approach, which is the size of the stack is fixed. In order to overcome this limitation, whenever the size of the stack reaches its limit we will simply double its size. To get the better understanding of this approach, look at the code below...

```
#include <climits>

class StackUsingArray {
    int *data;
    int nextIndex;
    int capacity;

    public :
    StackUsingArray() {
        data = new int[4];              //initially declared with a small size of 4
        nextIndex = 0;
        capacity = 4;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }
    bool isEmpty() {
        return nextIndex == 0;
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            int *newData = new int[2 * capacity];       //Capacity doubled
            for(int i = 0; i < capacity; i++) {
                newData[i] = data[i];                   //Elements copied
            }
            capacity *= 2;
            delete [] data;
            data = newData;
            /*cout << "Stack full " << endl;
            return;*/
```

```
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        int pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            nextIndex--;
            return data[nextIndex];
        }
        int top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            return data[nextIndex - 1];
        }
};
```

Stack using templates

While implementing the dynamic stack, we kept ourselves limited to the data of type integer only, but what if we want a generic stack(something that works for every other data type as well). For this we will be using templates. Refer the code below(based on the similar approach as used while creating dynamic stack):

```
#include <climits>

template <typename T>                    // Templates initialised
class StackUsingArray {
        T *data;                         // Template type of data used
        int nextIndex;
        int capacity;

        public :

        StackUsingArray() {
```

```
            data = new T[4];
            nextIndex = 0;
            capacity = 4;
        }
        // return the number of elements present in my stack
        int size() {
            return nextIndex;
        }
        bool isEmpty() {
            return nextIndex == 0;
        }

        // insert element
        void push(T element) {
            if(nextIndex == capacity) {
                T *newData = new T[2 * capacity];
                for(int i = 0; i < capacity; i++) {
                    newData[i] = data[i];
                }
                capacity *= 2;
                delete [] data;
                data = newData;
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        T pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            nextIndex--;
            return data[nextIndex];
        }
        // For extracting top element
        T top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            return data[nextIndex - 1];
        }
};
```

You can see that every function whose return type was int initially now returns T type (i.e., template-type).

Generally, the template approach of stack is preferred as it can be used for any data type irrespective of it being int, char, float, etc...

Stack using LL

Till now we have learnt how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists.

All the five functions that stacks can perform could be made using linked lists. Below is the template for you to work upon the same In case you are unable to come up with the program, kindly refer to the code in the solution section of the problem.

```
#include <iostream>
using namespace std;
template <typename T>
class Node {                    //Node class for Linked list
        public :
                T data;
                Node<T> *next;
                Node(T data) {
                        this -> data = data;
                        next = NULL;
                }
                ~Node() {
                        delete next;
                }
};

template <typename T>
class Stack {
        Node<T> *head;
         Node<T> *tail;
        int size;               // number of elements present in stack
```

```
        public :
        Stack() {                 // Constructor to initialize the head and tail to NULL and
                                  // size to zero
        }

        int getSize() {           // traverse the whole linked list and return its length

        }

        bool isEmpty() {          // Just check if the head pointer is NULL or not

        }

        void push(T element) {    // insert the newNode at the end of the list and
                                  // update the tail node
        }

        T pop() {    // remove the tail node from the list and then update the tail
                     // pointer to the previous position by traversing the list again.
        }

        T top() {    //return the value at the tail pointer. No change needed.

        }
};
```

In-built stack using STL

C++ provides the in-built stack in it's **standard temporary library (STL)** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the header file:

```
#include <stack>
```

To declare a stack use the following syntax:

```
stack <datatype_of_variables_that_will_be_stored> Name_of_stack;
```

For example: to create a stack of integer type with name 'st':

stack <int> st;

Now, you can simply perform all the operations using the in-built stack functions:

- **st.push(value_to_be_inserted)** : To insert a value in the stack
- **st.top()** : Returns the value at the top of the stack
- **st.pop()** : Deletes the value at the top from the stack.
- **st.size()** : Returns the total number of elements in the stack.
- **st.isEmpty()** : Returns a boolean value (True for empty stack and vice versa).

Practice Problems:

- https://www.hackerrank.com/challenges/equal-stacks/problem
- https://www.hackerrank.com/challenges/simple-text-editor/problem
- https://www.techiedelight.com/design-a-stack-which-returns-minimum-element-without-using-auxiliary-stack/
- https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/practice-problems/algorithm/monk-and-order-of-phoenix/

Queues

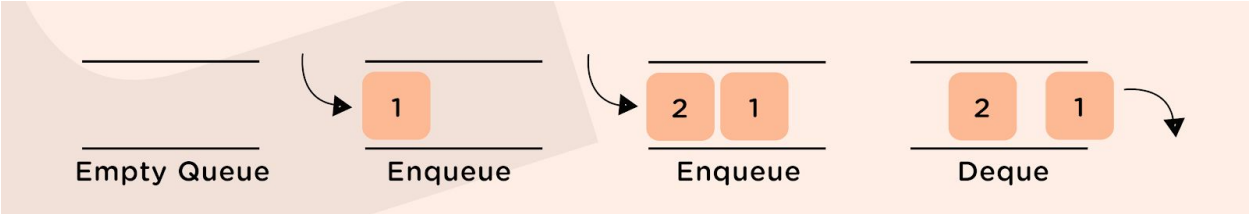
Introduction

Like stack, the queue is also an abstract data type. As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end. Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:

Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line; this order is known as **First In First Out (FIFO)**, a principle that queue data structure follows

In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".

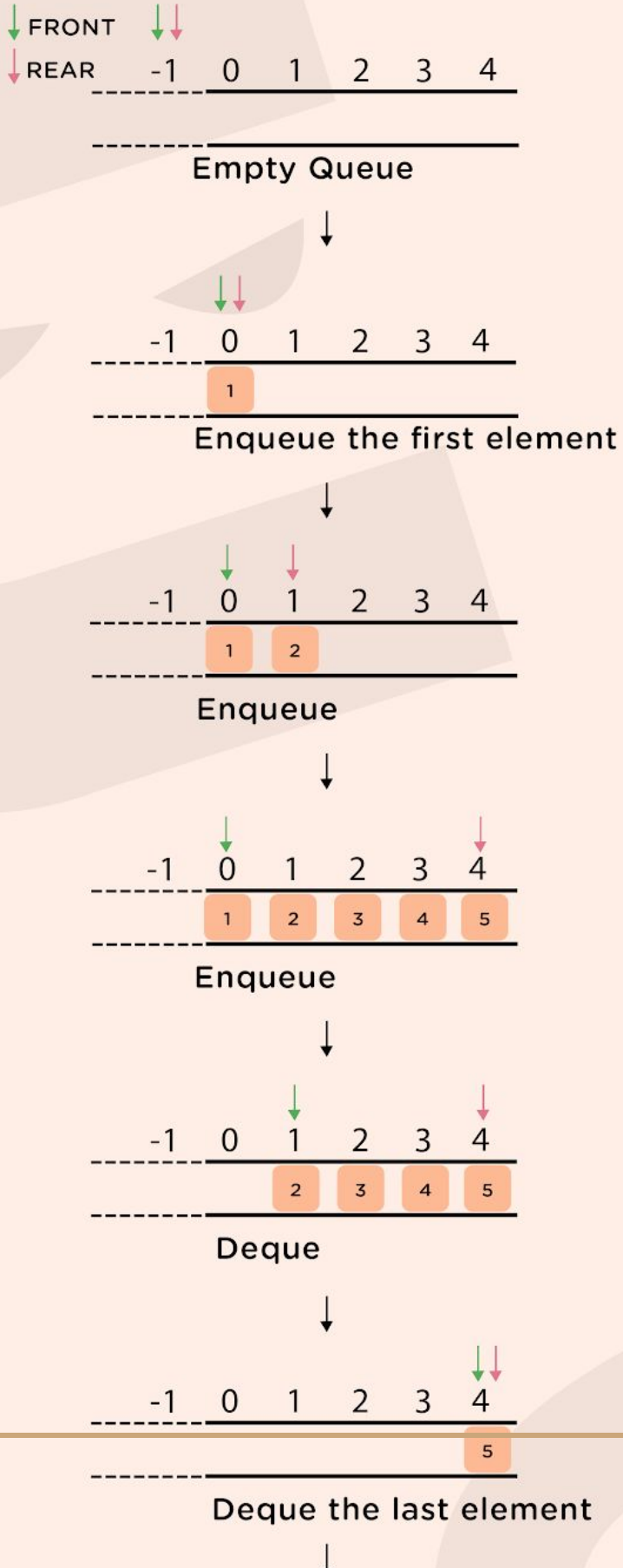


How does the queue work?

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueuing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueuing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueuing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.

Refer to the pictorial representation below:


FRONT
REAR
-1 0 1 2 3 4
Empty Queue
-1 0 1 2 3 4
1
Enqueue the first element
-1 0 1 2 3 4
1 2
Enqueue
-1 0 1 2 3 4
1 2 3 4 5
Enqueue
-1 0 1 2 3 4
2 3 4 5
Deque
-1 0 1 2 3 4
5
Deque the last element

Applications of queue:

- CPU Scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processesQueue is used for synchronization. eg: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people in order of their calling.

Queue using array

Queue contains majorly these five functions that we will be implementing:

- **enqueue()**: Insertion of element
- **dequeue()**: Deletion of element
- **front()**: returns the element present in the front position
- **getSize()**: returns the total number of elements present at current stage
- **isEmpty()**: returns boolean value, TRUE for empty and FALSE for non-empty.

Now, let's implement these functions in our program.

NOTE: We will be using templates in the implementation, so that it can be generalised.

Implementation of queue using array

Follow up the code along with the comments below:

```
template <typename T>                          // Generalised using templates
class QueueUsingArray {
        T *data;                               // to store data
        int nextIndex;                         // to store next index
        int firstIndex;                        // to store the first index
        int size;                              // to store the size
        int capacity;                          // to store the capacity it can hold

        public :
        QueueUsingArray(int s) {               // Constructor to initialize values
                data = new T[s];
                nextIndex = 0;
                firstIndex = -1;
                size = 0;
                capacity = s;
        }

        int getSize() {                        // Returns number of elements present
                return size;
        }

        bool isEmpty() {                       // To check if queue is empty or not
                return size == 0;
        }

        void enqueue(T element) {              // Function for insertion
                if(size == capacity) {         // To check if the queue is already full
                        cout << "Queue Full ! " << endl;
                        return;
                }
                data[nextIndex] = element;     // Otherwise added a new element
                nextIndex = (nextIndex + 1) % capacity ;    // in cyclic way
                if(firstIndex == -1) {         // Suppose if queue was empty
                        firstIndex = 0;
                }
                size++;                        // Finally, incremented the size
        }
```

```
        T front() {                                 //  To return the element at front position
                if(isEmpty()) {                     // To check if the queue was initially empty
                        cout << "Queue is empty ! " << endl;
                        return 0;
                }
                return data[firstIndex];    // otherwise returned the element
        }

        T dequeue() {                               // Function for deletion
                if(isEmpty()) {                     // To check if the queue was empty
                        cout << "Queue is empty ! " << endl;
                        return 0;
                }
                T ans = data[firstIndex];
                firstIndex = (firstIndex + 1) % capacity;
                size--;                             // Decrementing the size by 1
                if(size == 0) {                     // If queue becomes empty after deletion, then
                        firstIndex = -1;    // resetting the original parameters
                        nextIndex = 0;
                }
                return ans;
        }
};
```

Like stack, we can also make dynamic queues (discussed in the further sections)...

Dynamic queue

In the dynamic queue. we will be preventing the condition where the queue becomes full and we were not able to insert any further elements in that. Let's now check the implementation of the same.

Implementation is pretty similar to the static approach discussed above. A few minor changes are there which could be followed with the help of comments in the code below.

```
template <typename T>

class QueueUsingArray {
        T *data;
        int nextIndex;
        int firstIndex;
        int size;
        int capacity;

        public :
        QueueUsingArray(int s) {
                data = new T[s];
                nextIndex = 0;
                firstIndex = -1;
                size = 0;
                capacity = s;
        }

        int getSize() {
                return size;
        }

        bool isEmpty() {
                return size == 0;
        }

        void enqueue(T element) {
                if(size == capacity) {                          // When size becomes full
                        T *newData = new T[2 * capacity];       // we simply doubled the
                                                                // capacity
                        int j = 0;
                        for(int i = firstIndex; i < capacity; i++) {  // Now copied the
                                                                      Elements to new one
                                newData[j] = data[i];
                                j++;
                        }
                        for(int i = 0; i < firstIndex; i++) {       // Overcoming the initial
                                                                    //  cyclic insertion by copying
                                                                    //  the elements linearly
                                newData[j] = data[i];
                                j++;
                        }
                        delete [] data;
                        data = newData;
```

```
                firstIndex = 0;
                nextIndex = capacity;
                capacity *= 2;                  // Updated here as well
                //cout << "Queue Full ! " << endl;
                // return;
        }
        data[nextIndex] = element;
        nextIndex = (nextIndex + 1) % capacity ;
        if(firstIndex == -1) {
                firstIndex = 0;
        }
        size++;
    }

    T front() {
        if(isEmpty()) {
                cout << "Queue is empty ! " << endl;
                return 0;
        }
        return data[firstIndex];
    }

    T dequeue() {
        if(isEmpty()) {
                cout << "Queue is empty ! " << endl;
                return 0;
        }
        T ans = data[firstIndex];
        firstIndex = (firstIndex + 1) % capacity;
        size--;
        if(size == 0) {
                firstIndex = -1;
                nextIndex = 0;
        }
        return ans;
    }
};
```

The STL queues in C++ are implemented in a similar fashion.

We can also implement the queues with the help of linked lists.

Queues using LL

Check the function description below and try to implement it yourselves.

```
template <typename T>
class Node {                                    // Node class for linked list, no change needed
        public :
                T data;
                Node<T> *next;
                Node(T data) {
                        this -> data = data;
                        next = NULL;
                }
};
template <typename T>
class Queue {
        Nod<T> *head;                   // for storing front of queue
        Node<T> *tail;                  // for storing tail of queue
        int size;                       // number of elements in queue

        public :
        Queue() {                       // Constructor to initialise head, tail to NULL
                                        // and size to 0
        }

        int getSize() {                 // just return the size of linked list

        }

        bool isEmpty() {                // just check if head is NULL or not

        }

        void enqueue(T element) {   // Simply insert the new node at the tail of LL

        }

        T front() {                     // Returns the head pointer of LL. Be careful for
                                        // the case when size is 0
        }

        T dequeue() {                   // moves the head pointer one position ahead
                                        // and deletes the head pointer. Also decrease the
```

```
        }                                   // size by 1
};
```

In-built queue

C++ provides the in-built queue in it's **standard temporary library (STL)** which can be used instead of creating/writing a queue class each time.

Header file used:

```
#include <queue>
```

Syntax for declaration of queue:

```
queue <datatype> name_of_queue
```

Key functions of this in-built queue:

- **.push(element_value)** : Used to insert the element in the queue
- **.pop()** : Used to delete the element from the queue
- **.front()** : Returns the element at front of the queue
- **.size()** : Returns the total number of elements present in the queue
- **.empty()** : Returns TRUE if the queue is empty and vice versa
-

Let us now consider an example to implement queue using STL:

Problem Statement: Implement the following parts using queue:

1. Declare a queue of integers and insert the following elements in the same order as mentioned: 10, 20, 30, 40, 50, 60.
2. Now tell the element that is present at the front position of the queue
3. Now delete an element from the front side of the queue and again tell the element present at the front position of the queue.

4. Print the size of the queue and also tell if the queue is empty or not.

5. Now, print all the elements that are present in the queue.

Solution: Check the code for the above stated...

```
#include <iostream>
#include <queue>                        // Header file for using in-built queue
using namespace std;

int main() {
        queue<int> q;                   // queue declared of type int with name q
        q.push(10);                     // Now inserted elements using .push()
        q.push(20);                     // Part 1
        q.push(30);
        q.push(40);
        q.push(50);
        q.push(60);

        cout << q.front() << endl;      // Part 2
        q.pop();                        // Part 3
        cout << q.front() << endl;      // Part 3
        cout << q.size() << endl;       // Part 4
        cout << q.empty() << endl;          // prints 1 for TRUE and 0 for FALSE(Part 4)

        while(!q.empty()) {                 // prints all the elements until the queue
                                            // is empty (Part 5)
                cout << q.front() << endl;
                q.pop();
        }
}
```

Output of the following code is:

```
10
20
5
0
20
30
40
50
60
```

Practice problems:

- https://www.spoj.com/problems/ADAQUEUE/
- https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/practice-problems/algorithm/number-recovery-0b988eb2/
- https://www.codechef.com/problems/SAVJEW
- https://www.hackerrank.com/challenges/down-to-zero-ii/problem

Trees

Introduction

There are various systems around us which are hierarchical in nature; military, government organisations, corporations, they all have a hierarchical form of command chain. There are various systems around us which are hierarchical in nature; military, government organisations. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, those subordinates further have people reporting to them and so on. If we wanted to model this company with a data structure, it would be natural to think of the president as the head of all, the vice presidents at level 1, and their subordinates at lower levels as we go down the organizational hierarchy.

The examples of hierarchical models that we discussed above cannot be represented/stored using a linear data structure. For this very scenario, a data structure called a tree comes to our rescue. Now there are various types of tree, depending on the rule/property they follow. The simplest variant of tree is called a **general tree** (or N-ary tree). As in above example, the number of vice presidents is likely to be more than zero, we need to use a data structure that represents the data in the form of a hierarchy.

In this module we will examine general tree terminology and define a basic ADT for general trees.

Trees are non-linear hierarchical data structures. It is a collection of nodes connected to each other by means of “edges” which are either directed or undirected. One of the nodes is designated as “Root node” and the remaining nodes are called child nodes or the leaf nodes(nodes with no chid nodes).

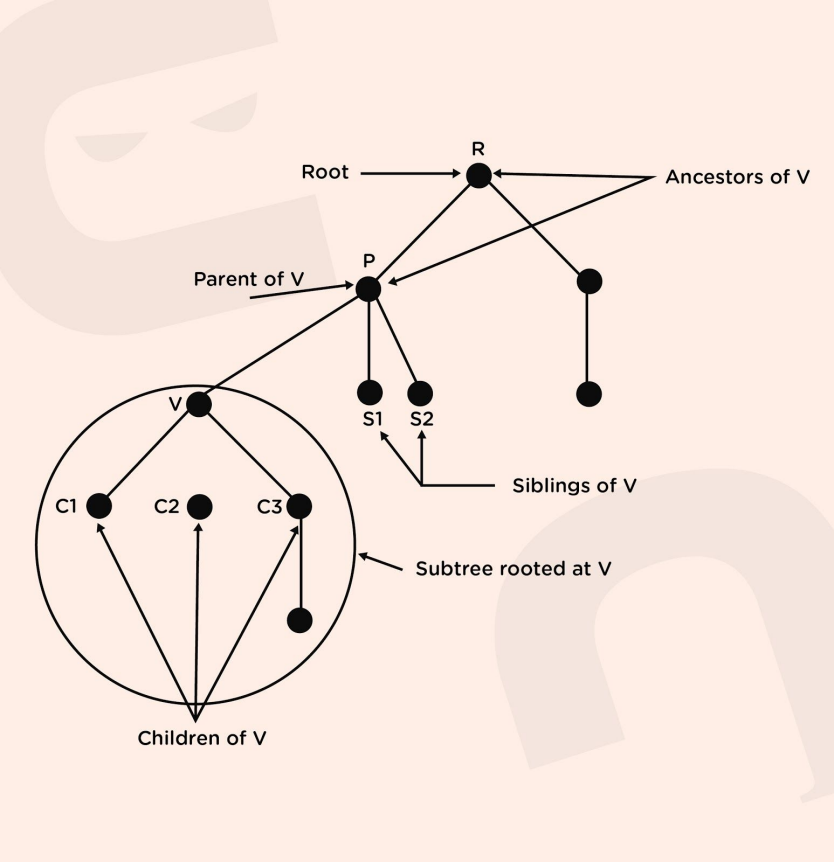
In general, each node can have as many children but only one parent node.

More formally, a tree T is a finite set of one or more nodes such that there is one designated node R, called the root of the tree. If the set (T - {R}) is not empty, these nodes are partitioned into n > 0 disjoint sets T_0, T_1, T_2,..., T_{n-1}, each of which is a tree, and whose roots R_1, R_2,..., R_n, respectively, are children of R. The subsets T_i (($0 \leq i < n$) are said to be ***subtrees*** of T. These subtrees, as ordered in that T_i set, are said to come before T_j if $i < j$. By convention, the subtrees are arranged from left to right with subtree T_0 called the leftmost child of R.

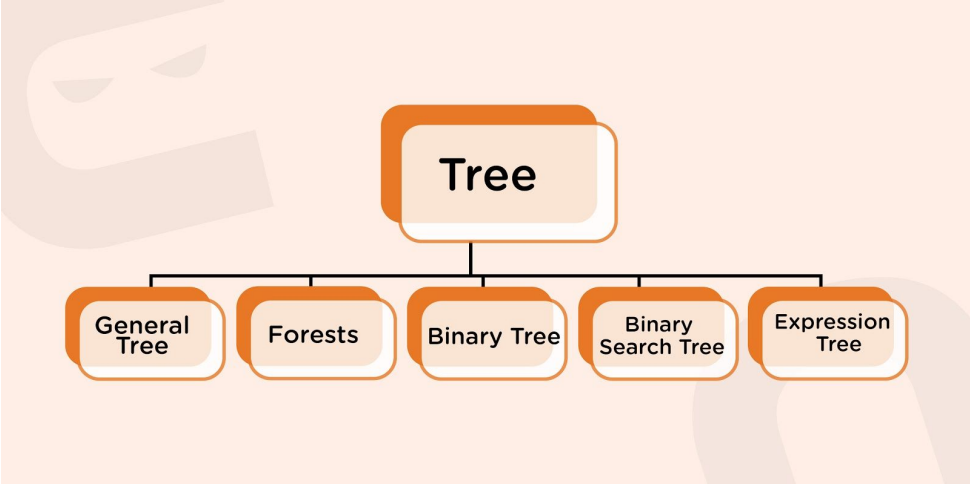
Let us go through some basic terminology associated with trees:

- **Root node:** This is the topmost node in the tree hierarchy.
- **Leaf node:** These are the bottommost nodes in a tree hierarchy. The leaf nodes do not have any child nodes. They are also known as external nodes.
- **Subtree:** Subtree represents various descendants of a node when the root is not null. A tree usually consists of a root node and one or more subtrees.
  Parent node: Any node except the root node that has a child node and an edge upward towards the parent.
- **Ancestor Node:** It is any predecessor node on a path from the root to that node. Note that the root does not have any ancestors.
- **Key:** It represents the data stored in a node.
- **Level:** Represents the generation of a node. A root node is always at level 1. Child nodes of the root are at level 2, grandchildren of the root are at level 3 and so on. In general, each node is at a level higher than its parent.
- **Path:** The path is a sequence of consecutive edges.
- **Degree:** Degree of a node indicates the number of children that a node has.

Below is provided the pictorial representation of above:



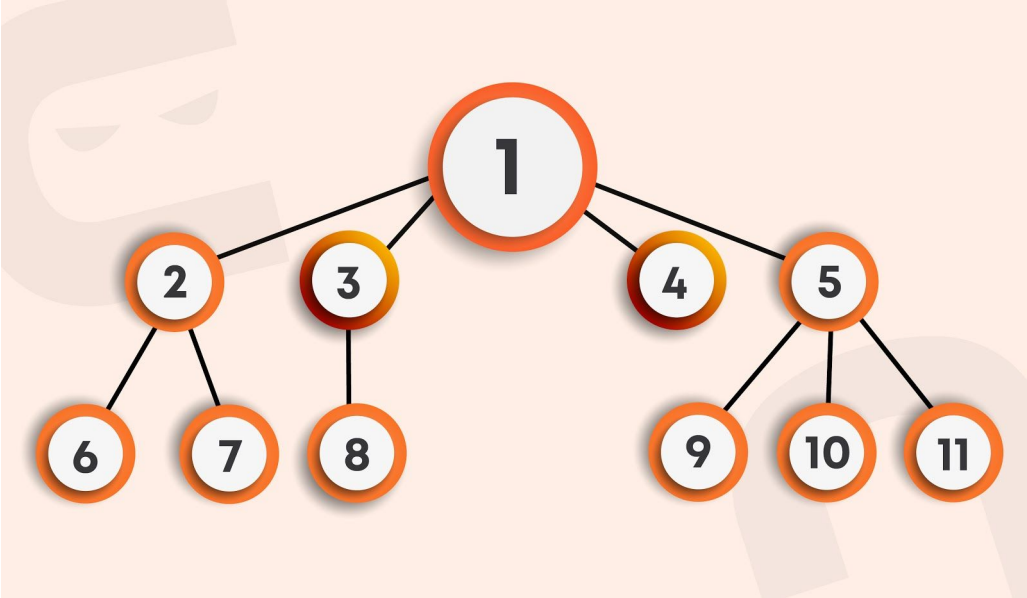
Types of Trees



In this course, we will discuss only General trees, Binary trees and Binary Search trees.

Tree node class

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Firstly, any implementation must be able to initialize a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of the ADT for binary tree nodes, this was done by providing member functions that give explicit access to the left and right child pointers. Unfortunately, because we do not know in advance how many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.



One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for

node implementations in favor of an array-based approach, because these functions favor random access to a list of children. In practice, an implementation based on a linked list is often preferred.

An alternative is to provide access to the first (or leftmost) child of a node, and to provide access to the next (or right) sibling of a node.

Here are the class declarations for general trees and their nodes. Based on these two access functions, the children of a node can be traversed like a list. Trying to find the next sibling of the rightmost sibling would return null.

Kindly refer below for code- (we will name this file as **TreeNode.h** to use it further in our program)

```
#include <vector>
using namespace std;

template <typename T>
class TreeNode {
        public:
        T data;                                 // To store data
        vector<TreeNode<T>*> children;          // To store children for each node

        TreeNode(T data) {                      // Constructor to initialize data
                this->data = data;
        }
};
```

Taking input and print Recursive

Kindly follow the comments in the code to understand it better...

```
#include <iostream>
#include "TreeNode.h"                   // TreeNode.h file included as told above
using namespace std;
```

```
TreeNode<int>* takeInput() {   // Function that returns root node after taking input
        int rootData;                // To store root data
        cout << "Enter data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);
        // Dynamically created a root node and initialized with constructor

        int n;                    // To store number of children of the node
        cout << "Enter num of children of " << rootData << endl;
        cin >> n;
        for (int i = 0; i < n; i++) {
                TreeNode<int>* child = takeInput();        // Input taken recursively for
                                                           // each child node of the current node
                root->children.push_back(child);      // Each child node is inserted into
                                                           // the list of children nodes'
        }
        return root;
}

void printTree(TreeNode<int>* root) {    // Function to print the tree that takes the
                                         //  root node as its argument

        if (root == NULL) {                    // Base case
                return;
        }

        cout << root->data << ":";
        for (int i = 0; i < root->children.size(); i++) {    // Traversing over the vector of
                                                             // its child nodes and printing each of it
                cout << root->children[i]->data << ",";
        }
        cout << endl;
        for (int i = 0; i < root->children.size() ; i++) {      // Now recursively calling print
                printTree(root->children[i]);           // function over each child
        }
}

int main() {
        TreeNode<int>* root = takeInput();
        printTree(root);
      // TODO : Delete tree
}
```

We learnt that the memory that is created dynamically also needs to be deleted. The same will be followed here also. We suggest you implement it yourself as an exercise.

HINT: Idea will be recursively traversing over each node of the tree and first delete each child and then delete the root node.

Take input level-wise

For taking input level-wise, we will use **queue data structure.** Follow the comments in the code below:

```
TreeNode<int>* takeInputLevelWise() {           // Function to take level-wise input
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);

        queue<TreeNode<int>*> pendingNodes; // Queue declared of type TreeNode
        pendingNodes.push(root);                // Root data pushed into queue at first
        while (pendingNodes.size() != 0) {      // Runs until the queue is not empty
                TreeNode<int>* front = pendingNodes.front();  // stores front of queue
                pendingNodes.pop();         // deleted that front node stored previously
                cout << "Enter num of children of " << front->data << endl;
                int numChild;
                cin >> numChild;    // get the number of child nodes
                for (int i = 0; i < numChild; i++) {  // iterated over each child node to
                                                      //  input it
                        int childData;
                        cout << "Enter " << i << "th child of " << front->data << endl;
                        cin >> childData;
                        TreeNode<int>* child = new TreeNode<int>(childData);
                        front->children.push_back(child);   // Each child node is pushed
//into the queue as well as the list of child nodes as it is taken input so that next
// time we can take its children as input while we kept moving in the level-wise
// fashion
                        pendingNodes.push(child);
                }
```

```
        }
        return root;                // Finally returns the root node
}
```

Similarly, we can also print the child nodes using a queue itself. Now, try doing the same yourselves and for solution refer to the solution tab of the respective question.

Count total nodes in a tree

To count the total number of nodes in the tree, we will just traverse the tree recursively starting from the root node until we reach the leaf node by iterating over the vector of child nodes. As the size of the child nodes vector becomes 0, we will simply return. Kindly check the code below:

```
int numNodes(TreeNode<int>* root) {
        if(root == NULL) {                          // Edge case
                return 0;
        }
        int ans = 1;                                // To store total count
        for (int i = 0; i < root->children.size(); i++) {      // iterating over children vector
                ans += numNodes(root->children[i]);   // recursively storing the count
                                                      // of children's children nodes.
        }
        return ans;                 // ultimately returning the final answer
}
```

Height of the tree

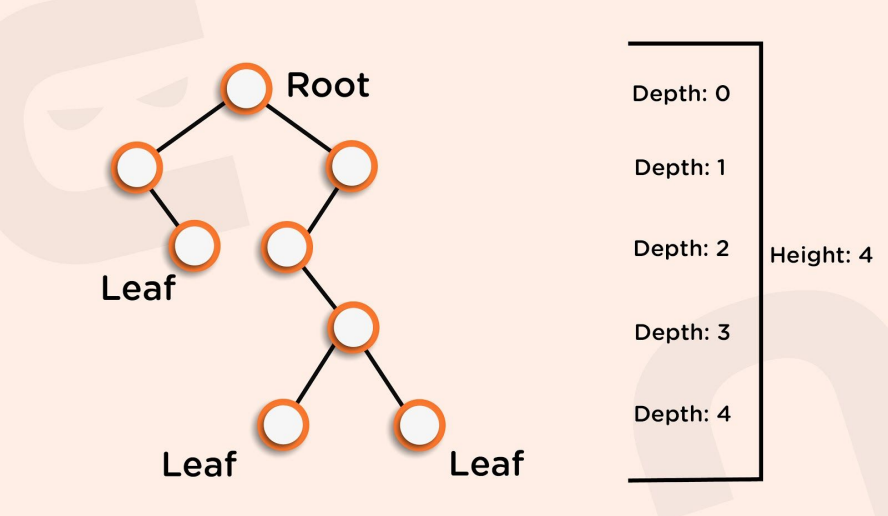
Height of a tree is defined as the length of the path from the tree's root node to any of its leaf nodes. Just think what should be the height of a tree with just one node? Well, there are a couple of conventions; we can define the height of a tree with just one node to be either 1 or zero. We will be following the convention where the height of a NULL tree is zero and that with only one node is one. This has been left

as an exercise for you, if need be you may follow the code provided in the solution tab of the topic corresponding to the same question.

Approach: Consider the height of the root node as 1 instead of 0. Now, traverse each child of the root node and recursively traverse over each one of them also and the one with the maximum height is added to the final answer along by adding 1 (this 1 is for the current node itself).

Depth of a node

Depth of a node is defined as it's distance from the root node. For example, the depth of the root node is 0, depth of a node directly connected to root node is 1 and so on. Now we will write the code to find the same... (Below is the pictorial representation of the depth of a node)



If you observe carefully, then the depth of the node is just equal to the level in which it resides. We have already figured out how to calculate the level of any node,using a similar approach we will find the depth of the node as well. Suppose, we want to find all the nodes at level 3, then from the root node we will tell its children to find the node that is at level 3 - 1 = 2, and similarly keep this up

recursively until we reach the depth = 0. Look at the code below for better understanding...

```
void printAtLevelK(TreeNode<int>* root, int k) {
        if(root == NULL) {                                  // Edge case
                return;
        }

        if(k == 0) {                                        // Base case: when the depth is 0
                cout << root->data << endl;
                return;
        }

        for(int i = 0; i < root->children.size(); i++) {    // Iterating over each child and
                printAtLevelK(root->children[i], k - 1);    // recursively calling with with 1
                                                            // depth less
        }
}
```

Count Leaf nodes

To count the number of leaves, we can simply traverse the nodes recursively until we reach the leaf nodes (the size of the children vector becomes zero). Following recursion, this is very similar to finding the height of the tree. Try to code it yourself and for the solution refer to the solution tab of the same.

Traversals

Traversing the tree is the manner in which we move on the tree in order to access all its nodes. There are generally 4 types of traversals in a tree:

- Level order traversal
- Preorder traversal
- Inorder traversal

- Postorder traversal

We have already discussed level order traversal. Now let's discuss the other traversals.

In Preorder traversal, we visit the current node first(starting with root) and then traverse the left sub-tree. After covering all nodes there, we will move towards the right subtree and visit in a similar manner. Refer the code below:

```
void preorder(TreeNode<int>* root) {
        if(root == NULL) {
            return;
        }
        cout << root->data << endl;
        for(int i = 0; i < root->children.size(); i++) {
            preorder(root->children[i].size());
        }
}
```

In postorder traversal, we visit all the child nodes first (from left to right order) and then we visit the current node. You will be coding this yourself (very similar to preorder traversal) and for solution, refer the solution tab in the corresponding questions.

We will study inorder traversal in further sections...

Destructor

Now, let's check the code to delete the tree that we left earlier. We will first-of-all delete the child nodes and then delete their root nodes and ultimately the main root node of the tree. If we simply delete the root node, then we will lose the references to its child nodes and hence only the root node will be deleted.

Kindly, refer to the code below for your reference:

```
void deleteTree(TreeNode<int> *root) {
        for(int i = 0; i < root->children.size(); i++) {
                deleteTree(root->children[i]);
        }
        delete root;
}
```

We will call this function in the main() and the tree will be deleted. But to make the code robust and easy to understand, we will be using destructors. We just want to call **delete root;** in the main() and it should delete the complete tree. Let's check the destructor part below:

```
~TreeNode() {
        for(int i = 0; i < children.size(); i++) {      // We will call delete on all its
                delete children[i];                     // children which will invoke
        } // corresponding destructor and ultimately delete the root node itself.
}
```

This destructor works in the same way as the recursive functions but is a better choice as code looks clean and easy-to interpret.

Vectors in C++

Introduction

Vectors are very similar to dynamic arrays for which we don't have to determine the size as done in case of arrays.

Vectors are sequence containers representing arrays that can change in size.

They can be used in the same way as the arrays and use the contiguous storage locations for storing data as in case of arrays. Just the difference is that these containers can change their size as per the requirements.

Dynamic implementation of arrays is used by vectors to store the elements with some initial size. In case of arrays, we need to reallocate the memory and then copy back these elements which is a time-expensive task. This is easily avoidable in case of vectors as these directly allocates some extra storage to accommodate the elements. There is a possibility that a vector's capacity is more than the number of elements present in it as it increases it's space using square function.

This is achieved by *amortized constant time* complexity which is internally achieved in C++.

Syntax for vector declaration:

```
vector<data_type> vector_name;
```

For example: To declare a vector of integer with a name v:

```
vector<int> v;
```

To dynamically allocate the same:

```
vector<int> *v = new vector<int>();
```

Header file used is : **#include<vector>**

Some important functions of vector:

push_back	Add element at the end
pop_back	Delete last element
insert	Insert elements
size	Return size
capacity	Return size of allocated storage capacity

Practice to implement a vector in C++

Statement: Perform the following tasks:

1. Create a vector of integer.
2. Insert the elements from 1 to 10 and print size along with capacity at each step.

Solution Code:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
        vector<int> v;                          // Part 1

        for(int i = 1; i <= 10; i++) {          // Part 2
                cout << v.capacity() << " " << v.size() << endl;
                v.push_back(i);
        }
        return 0;
}
```

Output:

```
0 0
1 1
2 2
4 3
4 4
8 5
8 6
8 7
8 8
16 9
```

Observation:

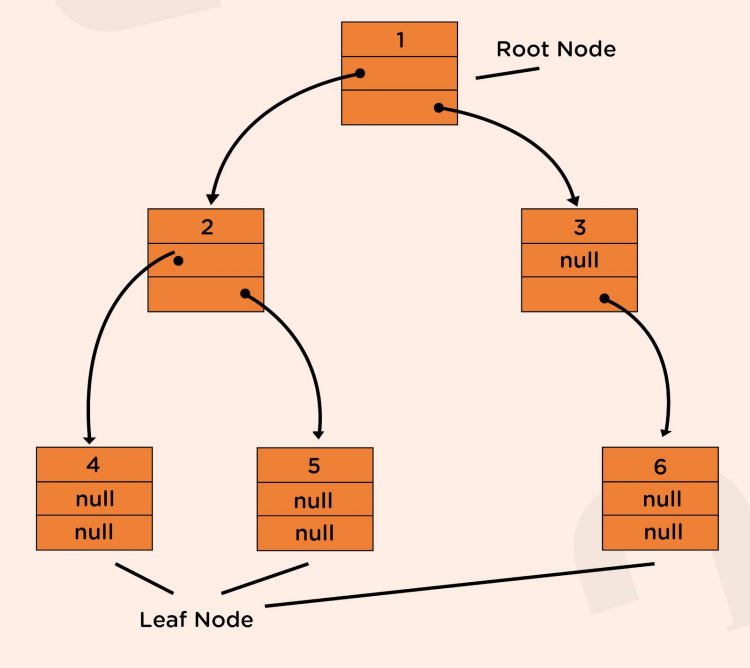
You can see that capacity doubles each time as the vector gets an extra element. But size represents the total number of elements present in the vector irrespective of its capacity.

Binary Trees

Introduction

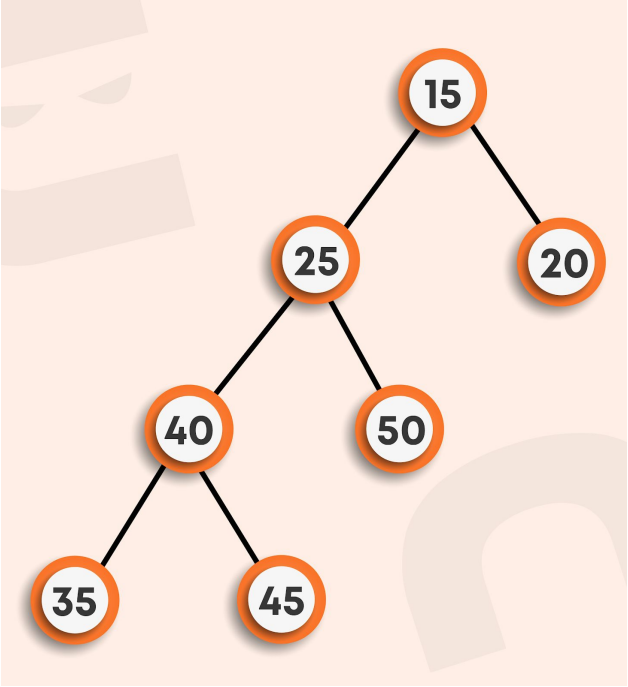
A generic tree with at most two child nodes for each parent node is known as a binary tree.

A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree. The left and right pointers recursively point to smaller **subtrees** on either side. A null pointer represents a binary tree with no elements, i.e., an empty tree. A formal definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.



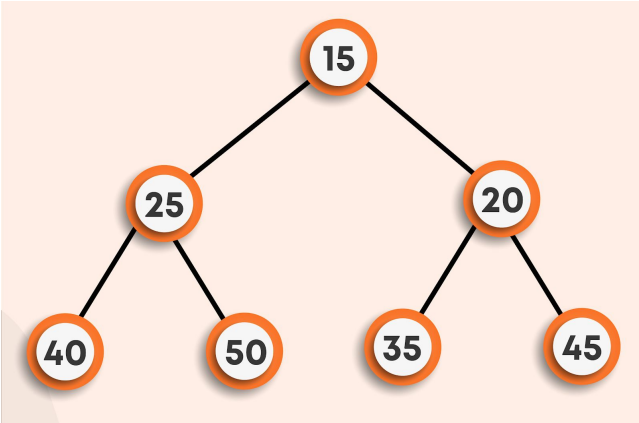
Types of binary trees:

- **Full binary trees:** A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

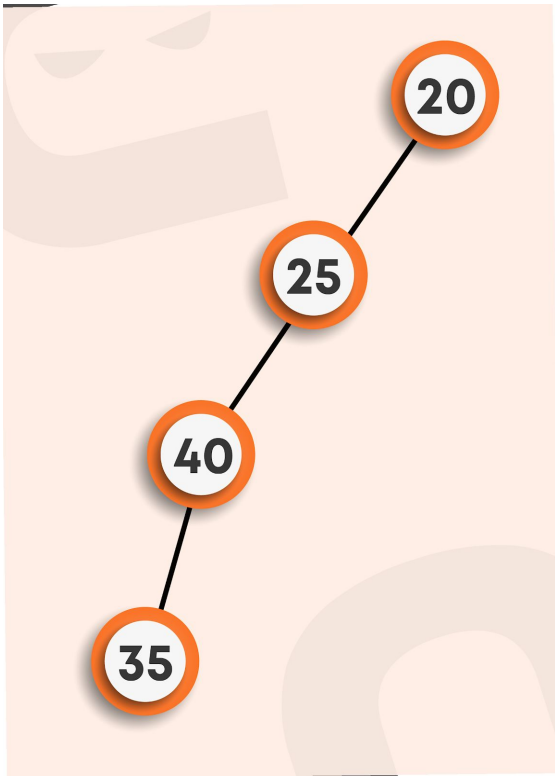


- **Complete binary tree:** A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

- **Perfect binary tree:** A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.

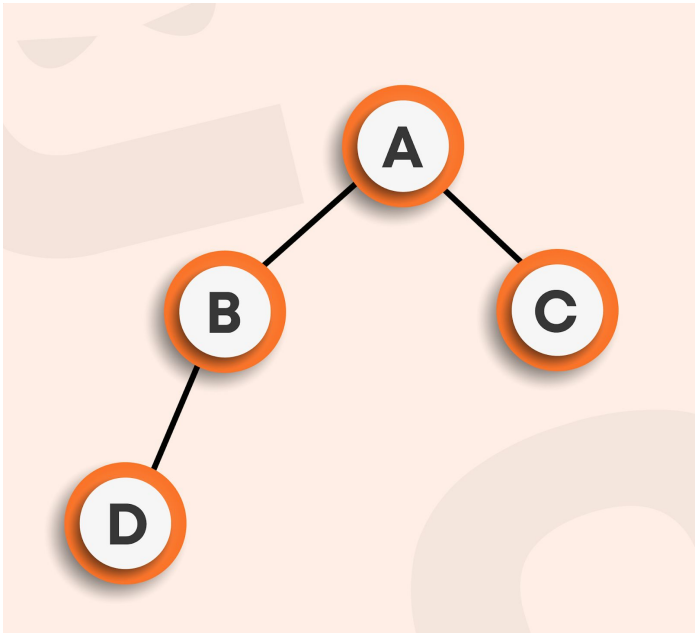


- **A degenerate tree:** In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

- **Balanced binary tree:** A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



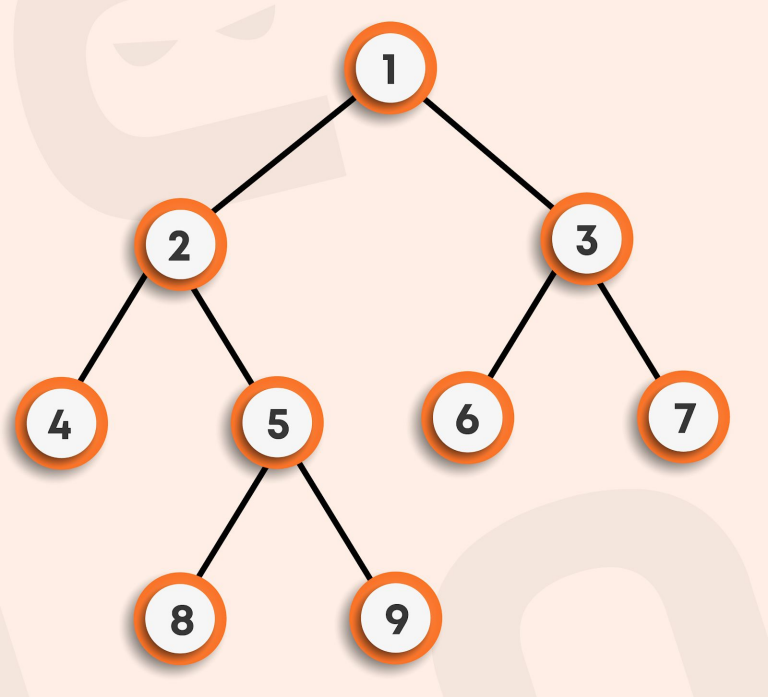
Binary tree representation:

Binary trees can be represented in two ways:

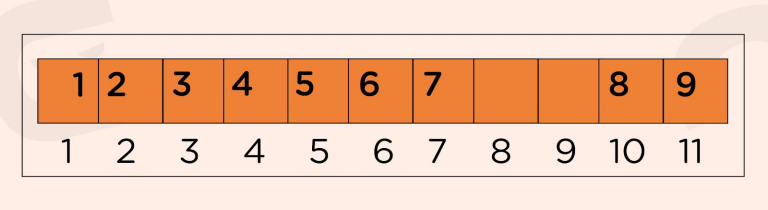
- **Sequential representation:** This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes. The number of nodes in a tree defines the size of the array. The root node of the tree is held at the first index in the array.

  In general, if a node is stored at the ith location, then its left and right child is kept at 2i and 2i+1 locations, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:



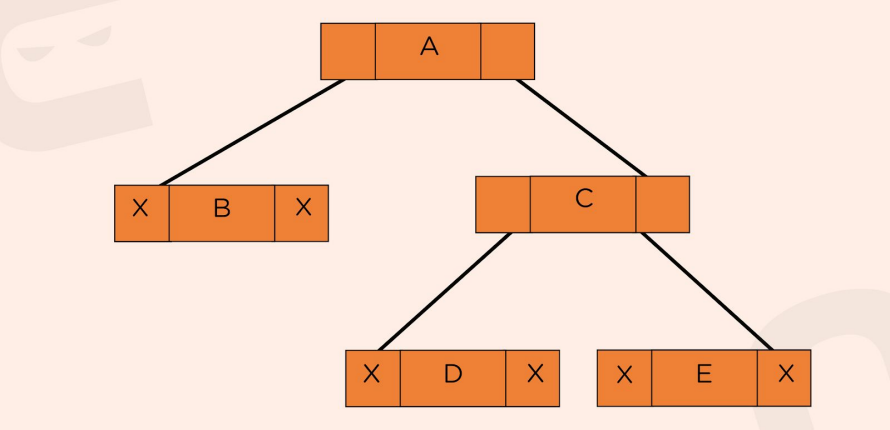
Here, we see that the left and right child of each node is stored at locations 2*(node_position) and 2*(node_position)+1, respectively.

For Example: The location of node 3 in the array is 3. So its left child will be placed at 2*3 = 6. Its right child will be at the location 2*3 +1 = 7. As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

Linked list representation: In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree.

The following diagram shows a linked list representation for a tree.



As shown in the above representation, each linked list node has three components:

- Left pointer
- Data part
- Right pointer

The left pointer has a pointer to the left child of the node; the right pointer has a pointer to the right child of the node whereas the data part contains the actual data of the node. If there are no children for a given node (leaf node), then the left and right pointers for that node are set to null . Let's now check the implementation of binary tree class. Like TreeNode class, here also we will be creating a separate file with **.h extension** and then use it wherever necessary. (Here the name of the file will be: **BinaryTreeNode.h**)

```
template <typename T>
```

```
class BinaryTreeNode {
        public:
        T data;                                 // To store data
        BinaryTreeNode* left;                   // for storing the reference to left pointer
        BinaryTreeNode* right;                  // for storing the reference to right pointer
        // Constructor
        BinaryTreeNode(T data) {
                this->data = data;              // Initializes data of the node
                left = NULL;                    // initializes left and right pointers to NULL
                right = NULL;
        }
        // Destructor
        ~BinaryTreeNode() {
                delete left;                    // Deletes the left pointer
                delete right;                   // Deletes the right pointer
        }                                       // As it ends, deletes the node itself
};
```

Take input and print recursively

Let's first check on printing the binary tree recursively. Follow the comments in the code below...

```
void printTree(BinaryTreeNode<int>* root) {
        if (root == NULL) {                     // Base case
                return;
        }
        cout << root->data << ":";  // printing the data at root node
        if (root->left != NULL) {     // checking if left not NULL, then print it's data also
                cout << "L" << root->left->data;
        }

        if (root->right != NULL) { // checking if right not NULL, then print it's data also
                cout << "R" << root->right->data;
        }
        cout << endl;
```

```
        printTree(root->left);    // Now recursively, call on the left and right subtrees
        printTree(root->right);
}
```

Now, let's check the input function: (We will be following the level-wise order for taking input and -1 denotes the NULL pointer or simply, it means that a pointer over the same place is a NULL pointer.

```
BinaryTreeNode<int>* takeInput() {
        int rootData;
        cout << "Enter data" << endl;
        cin >> rootData;                        // taking data as input
        if (rootData == -1) {                   // if the data is -1, means NULL pointer
                return NULL;
        }
       // Dynamically create the root Node which calls constructor of the same class
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
       // Recursively calling over left subtree
        BinaryTreeNode<int>* leftChild = takeInput();
       // Recursively calling over right subtree
       BinaryTreeNode<int>* rightChild = takeInput();
        root->left = leftChild;     // now allotting left and right childs to the root node
        root->right = rightChild;
        return root;
}
```

Taking input iteratively

We have already discussed a recursive approach for taking input in a binary tree in a level order fashion. Now, let's discuss the iterative procedure for the same using queue as we did in Generic trees. Follow the code along with the comments...

```
BinaryTreeNode<int>* takeInputLevelWise() {
```

```
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;                                    // Taking node's data as input
        if (rootData == -1) {                               // As -1 refers to NULL pointer
                return NULL;
        }
       // Dynamic allocation of root node
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
       // Using queue to level wise traversal for iterative approach
        queue<BinaryTreeNode<int>*> pendingNodes;
        pendingNodes.push(root);                // root node pushed into the queue
        while (pendingNodes.size() != 0) {      // process continued until the size of queue ≠ 0
                BinaryTreeNode<int>* front = pendingNodes.front();//front of queue is stored
                pendingNodes.pop();
                cout << "Enter left child of " << front->data << endl;
                int leftChildData;
                cin >> leftChildData;       // Left child of current node is taken input
                if (leftChildData != -1) {    // If the value of left child is not -1 that is, not NULL
                        BinaryTreeNode<int>* child = new BinaryTreeNode<int>(leftChildData);
                        front->left = child;            // Value assigned to left part and then pushed
                        pendingNodes.push(child);   // to the queue
                }
               // Similar work is done for right subtree
                cout << "Enter right child of " << front->data << endl;
                int rightChildData;
                cin >> rightChildData;
                if (rightChildData != -1) {
                       BinaryTreeNode<int>* child = new BinaryTreeNode<int>(rightChildData);
                        front->right = child;
                        pendingNodes.push(child);
                }
        }
        return root;
}
```

Count nodes

Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node. Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not NULL. Kindly follow the comments in the upcoming code...

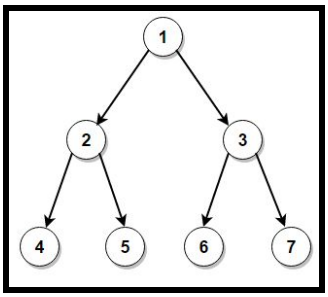
```
int numNodes(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Condition to check if the node is not NULL
                return 0;                       // counted as zero if so
        }
        return 1 + numNodes(root->left) + numNodes(root->right);   // recursive calls
            // on left and right subtrees with addition of 1(for counting current node)
}
```

Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Level order traversal**: Moving on each level from left to right direction
- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- **Level order traversal:** 1, 2, 3, 4, 5, 6, 7
- **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- **Post order traversal:** 4, 5, 2, 6, 7, 3, 1
- **Inorder traversal:** 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```
void inorder(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Base case when node's value is NULL
                return;
        }
        inorder(root->left);                   //Recursive call over left part as it needs
                                               // to be printed first
        cout << root->data << " ";             // Now printed root's data
        inorder(root->right);      // Finally a recursive call made over right subtree
}
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. For the answer, refer to the solution tab for the same.

Construct a binary tree from preorder and inorder traversal

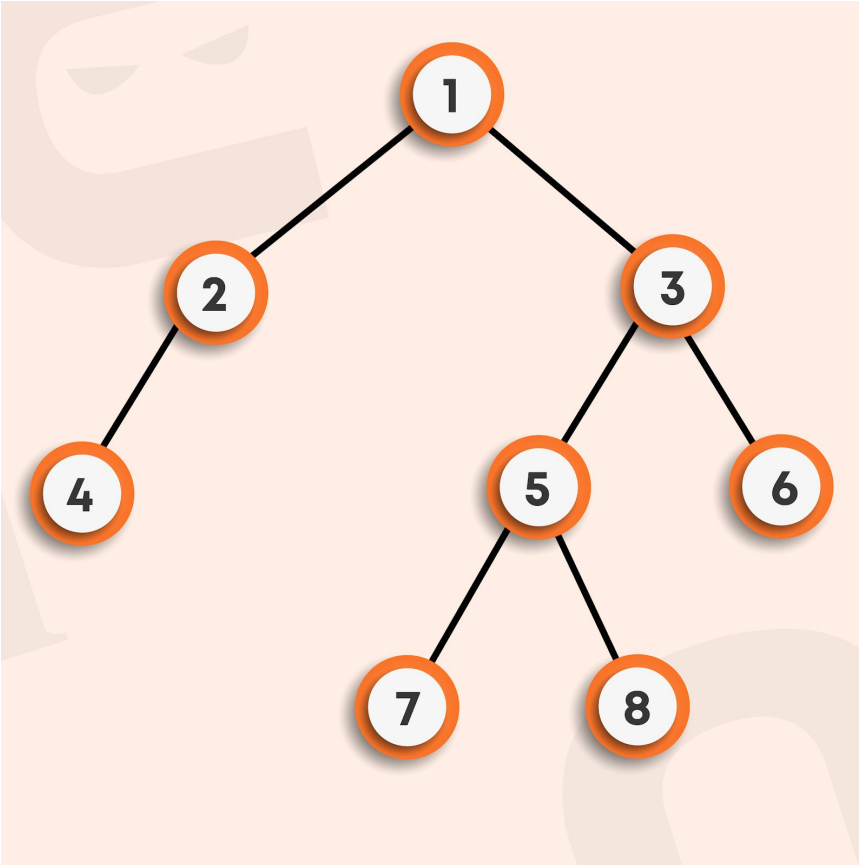
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



The idea is to start with the root node, which would be the first item in the preorder sequence and find the boundary of its left and right subtree in the inorder array. Now all keys before the root node in the inorder array become part of the left subtree, and all the indices after the root node become part of the right subtree. We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-

Inorder: {4, 2, 1, 7, 5, 8, 3, 6}

Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

The root will be the first element in the preorder sequence, i.e. 1. Next, we locate the index of the root node in the inorder sequence. Since 1 is the root node, all

nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:	**Right subtree:**
Inorder : {4, 2}	**Inorder :** {7, 5, 8, 3, 6}
Preorder : {2, 4}	**Preorder :** {3, 5, 7, 8, 6}

Follow the above approach until the complete tree is constructed. Now let us look at the code for this problem:

```
BinaryTreeNode<int>* buildTreeHelper(int* in, int* pre, int inS, int inE, int preS, int preE) {
        if (inS > inE) {                        // Base case
                return NULL;
        }

        int rootData = pre[preS];   // Root's data will be first element of the preorder array
        int rootIndex = -1;             // initialised root's index to -1 and searched for it's value
        for (int i = inS; i <= inE; i++) {      // in inorder list
                if (in[i] == rootData) {
                        rootIndex = i;
                        break;
                }
        }
       // Initializing the left subtree's indices for recursive call
        int lInS = inS;
        int lInE = rootIndex - 1;
        int lPreS = preS + 1;
        int lPreE = lInE - lInS + lPreS;
      // Initializing the right subtree's indices for recursive call
        int rPreS = lPreE + 1;
        int rPreE = preE;
        int rInS = rootIndex + 1;
        int rInE = inE;
      // Recursive calls follows
        BinaryTreeNode<int>*  root = new BinaryTreeNode<int>(rootData);
```

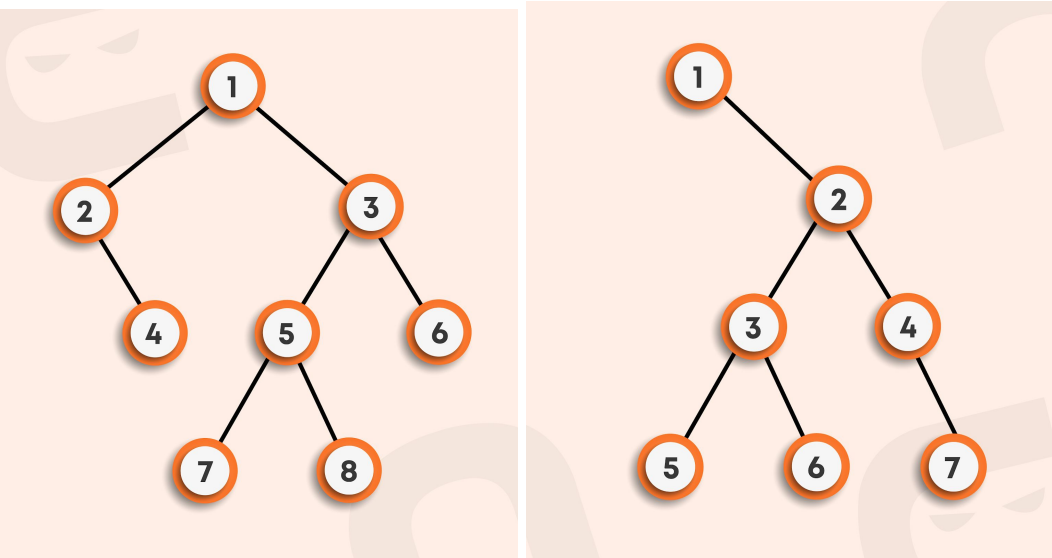
```
        root->left = buildTreeHelper(in, pre, lInS, lInE, lPreS, lPreE);    // over left subtree
        root->right = buildTreeHelper(in, pre, rInS, rInE, rPreS, rPreE); // over right subtree
        return root;    // finally returned the root
}
BinaryTreeNode<int>* buildTree(int* in, int* pre, int size) {    // this is the function called
 // from the main() with inorder and preorder traversals in the form of arrays and their
// size which is obviously same for both
        return buildTreeHelper(in, pre, 0, size - 1, 0, size - 1);  // These arguments are of the
   // form (inorder_array, preorder_array, inorder_start, inorder_end, preorder_start,
    // preorder_end) in the helper function for the same written above.
}
```

Now, try to construct the binary tree when inorder and postorder traversals are given...

The diameter of a binary tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes. The diameter of the binary tree may pass through the root (not necessary).

For example, the Below figure shows two binary trees having diameters 6 and 5, respectively (nodes highlighted in blue color). The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```
int height(BinaryTreeNode<int>* root) {   // Function to calculate height of tree
        if (root == NULL) {
                return 0;
        }
        return 1 + max(height(root->left), height(root->right));
}

int diameter(BinaryTreeNode<int>* root) {   // Function for calculating diameter
        if (root == NULL) {                 // Base case
                return 0;
        }

        int option1 = height(root->left) + height(root->right);   // Option 1
        int option2 = diameter(root->left);                       // Option 2
        int option3 = diameter(root->right);                      // Option 3
```

```
        return max(option1, max(option2, option3));   // returns the maximum value
}
```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be O(n).
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to O(n*h). (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, h is the height of the tree, which could be $O(n^2)$.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra n traversals for each node. To achieve this, move towards the other sections...

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a NULL tree, height and diameter both are equal to 0. Hence, pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = max(leftHeight, rightHeight)

Diameter = max(leftHeight + rightHeight, leftDiameter, rightDiameter)

Note: C++ provides an in-built pair class, which prevents us from creating one of our own. The Syntax for using pair class:

```
pair<datatype1, datatype2> name_of_pair_class;
```

For example: To create a pair class of int, int with a name p, follow the syntax below:

pair<int, int> p;

To access this pair class, we will use **.first** and **.second** pointers.

Follow the code below along with the comments to get a better grip on it...

```
pair<int, int> heightDiameter(BinaryTreeNode<int>* root) {
                                                    // pair class return-type function
        if (root == NULL) {          // Base case
                pair<int, int> p;
                p.first = 0;
                p.second = 0;
                return p;
        }
        // Recursive calls over left and right subtree
        pair<int, int> leftAns = heightDiameter(root->left);
        pair<int,int> rightAns = heightDiameter(root->right);
        // Hypothesis step
        // Left diameter, Left height
        int ld = leftAns.second;
        int lh = leftAns.first;
        // Right diameter, Right height
        int rd = rightAns.second;
        int rh = rightAns.first;

       // Induction step
        int height = 1 + max(lh, rh);                    // height of current root node
        int diameter = max(lh + rh, max(ld, rd));  // diameter of current root node
        pair<int, int> p;                                // Pair class for current root node
```

```
        p.first = height;
        p.second = diameter;
        return p;
}
```

Now, talking about the time complexity of this method, it can be observed that we are just traversing each node once while making recursive calls and rest all other operations are performed in constant time, hence the time complexity of this program is O(n), where n is the number of nodes.

Practice problems:

- https://www.hackerrank.com/challenges/tree-top-view/problem
- https://www.codechef.com/problems/BTREEKK
- https://www.spoj.com/problems/TREEVERSE/
- https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/approximate/largest-cycle-in-a-tree-9113b3ab/

Binary Search Tree (BST)

Introduction

- These are the specific types of binary trees.
- These are inspired by the binary search algorithm that elements on the left side of a particular element are smaller and greater element in case of the right side.
- Time complexity on insertion, deletion and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss it further).

How is data stored?

In BSTs, we always insert the node with smaller data towards the left side of the compared node and the larger data node as its right child. To be more concise, consider the root node of BST to be N, then:

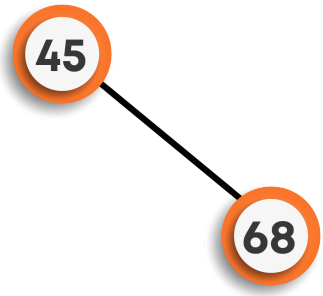
- Everything lesser than N will be placed in the left subtree.
- Everything greater than N will be placed in the right subtree.

For Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.

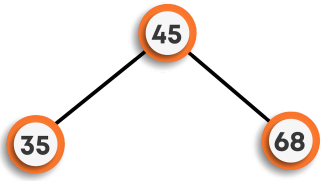
Solution: Follow the steps below:

1. Since the tree is empty, so the first node will automatically be the root node.

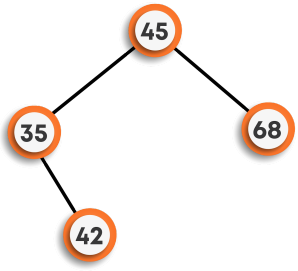
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



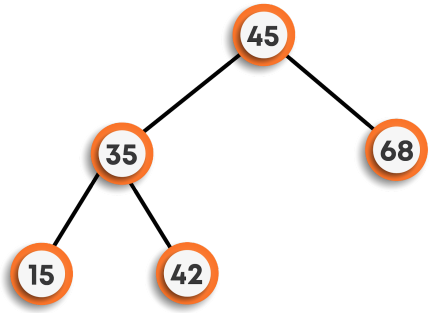
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



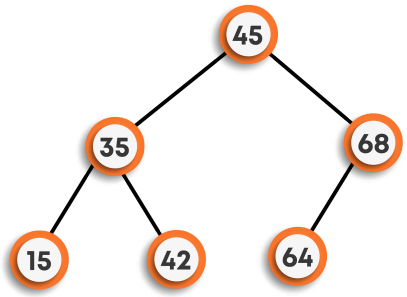
4. Moving on to inserting 42, we can see that 42 is less than 45 so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35 means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



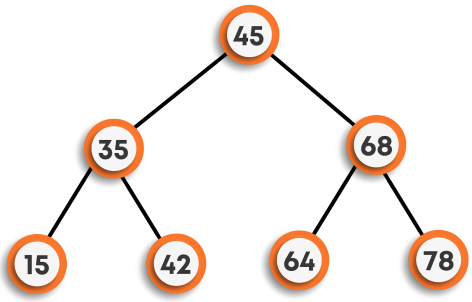
5. Now, on inserting 15, we will follow the same approach starting from the root node. Here, 15 < 45, means left subtree. Again, 15 < 35, means continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



6. Continuing further, to insert 64, now we found 64 > root node's data but less than 68, hence will be the left child of 68.



7. Finally, inserting 78, we can see that 78 > 45 and 78 > 68, so will be the right child of 68.



In this way, the data is stored in a BST.

If we follow the **inorder traversal** of the final BST, we will get the sorted array, hence to search we can now directly apply binary search algorithm technique over it to search for any particular data.

As seen above, to insert an element, we will be at most traversing either the left subtree's leaf node or right subtree's leaf node ignoring the other half straight away. Hence, the **time complexity of insertion** for each node is O(logh) (where h is the height of the tree).

For inserting n nodes, complexity will be O(nlogh).

Search in BST

Problem statement: Given a BST and a target value(x), we have to return the binary tree node with data x if present in BST; otherwise, return NULL.

Approach: As the given tree is BST, we can use the binary search algorithm here as we know that the left side of the node contains all elements smaller than it and the right side contains all elements greater than the value of node. Using recursion will make it easier as we just have to work on a small task, and rest recursion will handle itself.

- **Base Case:** If the tree is empty, it means the root node is NULL, then we will simply return NULL as the node is not present. Suppose if root's data is equal to x, we don't need to traverse forward in this tree as the target value has been found out, so we will simply return root from here.
- **Small Calculation:** In the case of binary trees, we were required to traverse both the left and right subtree of the root in order to find out the target value. But in case of BST, we'll only check for the condition of binary search, i.e., if x is greater than root's data, then we will make a recursive call over the

right subtree; otherwise, the recursive call will be made on the left subtree. This way, we are entirely discarding the half tree to be searched as done in case of a binary search. Therefore, the time complexity of searching is O(logh) (where h is the height of BST).

- **Recursive call:** After figuring out which way to move, we can make recursive calls on either left or right subtree.

This way, we will be able to figure out the search in a BST.

Print elements in a range:

Problem statement: Given a BST and a range (L, R), we need to figure out all the elements of BST that are present in the given range inclusive of L and R.

Approach: We will be using recursion and binary searching for the same...

- **Base case:** If the root is NULL, it means we don't have any tree to check upon, and we can simply return.
- **Small Calculation:** There are three conditions to be checked upon:
    - If root's data lies in the given range, then we can print it.
    - We will compare the root's data with the given range's maximum. If root's data is smaller than R, then we will have to traverse the right subtree.
    - Now, we will compare the root's data with the given range's minimum. If root's data is greater than L, then we will traverse on the left subtree.
- **Recursive call:** Recursive call will be made as per the small calculation part onto the left and right subtrees.

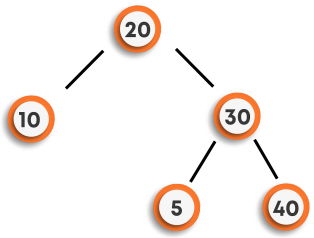
In this way, we will be able to figure out all the elements in the range. Try to code this yourself, and for the solution, refer to the solution tab of the same in the module.

Check BST

Problem statement: Given a binary tree, we have to check if it is a BST or not.

Approach: We will simply traverse the binary tree and check for the following cases:

- If the node's value is greater than the value of the node on it's left.
- If the node's value is smaller than the value of node on it's right.
- ***Important Case:*** Don't just compare the direct left and right children of the node; instead, we need to compare every node in the left and right subtree with the node' s value. Consider the following case:



Here, it can be seen that for root, left subtree is a BST, and right subtree is also a BST (individually), but the complete tree is not as a node with value 5 lies on the right side of the root node with value 20, whereas it should be on the left side of the root node. Hence, individual subtrees are BSTs, but it is possible that the complete binary tree is not a BST, hence, the third condition must also be checked.

To check over this condition, we will keep track of minimum and maximum values of right and left subtrees correspondingly, and at last, we will simply compare them with root.

- The left subtree's maximum value should be less than the root's data.
- The right subtree's minimum value should be greater than the root's data.

Now, let's look at the code for this approach starting from the brute-force approach.

```
int maximum(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // If root is NULL, then we simply return
                return INT_MIN;                // -∞ (negative infinity)
        }
        // Otherwise returning maximum of left/right subtree and root's data
        return max(root->data, max(maximum(root->left), maximum(root->right)));
}


int minimum(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // If root is NULL, then we simply return
                return INT_MAX;                // +∞ (positive infinity)
        }
        // Otherwise returning minimum of left/right subtree and root's data
        return min(root->data, min(minimum(root->left), minimum(root->right)));
}

bool isBST(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Base case
                return true;
        }

        int leftMax = maximum(root->left);     // Figuring out left's maximum
        int rightMin = minimum(root->right);   // Figuring out right's minimum
        bool output = (root->data > leftMax) && (root->data <= rightMin) &&
                                                isBST(root->left) && isBST(root->right);
        //Checked the conditions discussed above
        return output;
}
```

Time Complexity: in the isBST(), we are traversing each node, and for each node, we are then calculating the minimum and maximum value by again traversing that complete subtree's height. Hence if there are *n* nodes in total and height of the tree is *h*, the time complexity will be O(n*h).

To improve this further, we can see that for each node, minimum and maximum values are calculated separately. We want to calculate these along with checking the isBST condition.

We will follow a similar approach as that of the diameter calculation of binary trees. We will create a class that will be storing maximum value, minimum value, and BST status (True/False) for each node of the tree.

Let's look at its implementation now...

```
class IsBSTReturn {                                    // Class to store data for each node of tree
        public:
                bool isBST;
                int minimum;
                int maximum;
};

IsBSTReturn isBST2(BinaryTreeNode<int>* root) {
        if (root == NULL) {                            // Base Case
                IsBSTReturn output;                    // Object created for class and then values initialized
                output.isBST = true;                   // Empty tree is a BST
                output.minimum = INT_MAX;
                output.maximum = INT_MIN;
                return output;
        }
        IsBSTReturn leftOutput = isBST2(root->left);           // Left subtree Recursive call
        IsBSTReturn rightOutput = isBST2(root->right);         // Right subtree Recursive call

       // Small Calculation
       // Minimum and maximum values figured out side-by-side preventing extra traversals
        int minimum = min(root->data, min(leftOutput.minimum, rightOutput.minimum));
        int maximum = max(root->data, max(leftOutput.maximum, rightOutput.maximum));
      // Checking out for the subtree if it's a BST or not
        bool isBSTFinal = (root->data > leftOutput.maximum) && (root->data <=
                                rightOutput.minimum) && leftOutput.isBST && rightOutput.isBST;

       // Assigning values to the output class object
        IsBSTReturn output;
        output.minimum = minimum;
        output.maximum = maximum;
        output.isBST = isBSTFinal;
```

```
        return output;
}
```

Time Complexity: Here, we are going to each node and doing a constant amount of work. Hence, the time complexity for *n* nodes will be of O(n).

The time complexity for this problem can't be improved further, but there is a better approach to this problem, which makes our code look more robust. Let's discuss that approach now...

Approach: We will be checking on the left subtree, right subtree, and combined tree without using class. We will be using the concept of default argument over here. Check the code below:

```
// This time function is using default arguments for storing minimum and
// maximum value for each node
bool isBST3(BinaryTreeNode<int>* root, int min = INT_MIN, int max = INT_MAX) {
        if (root == NULL) {                    // Base case: Empty tree
                return true;
        }
// checking out the special condition first and returning false if not satisfied
        if (root->data < min || root->data > max) {
                return false;
        }
// Checking out left and right subtrees
        bool isLeftOk = isBST3(root->left, min, root->data - 1);
        bool isRightOk = isBST3(root->right, root->data, max);
// Returning true if both are BST and false otherwise.
        return isLeftOk && isRightOk;
}
```

Time Complexity: Here also, we are just traversing each node and doing a constant work on each of them; hence time complexity remains the same, i.e., O(n).

Construct BST from sorted array

Problem statement: Given a sorted array, we have to construct a BST out of it.

Approach: Suppose we take the first element as the root node, then the tree will be skewed as the array is sorted. In order to get a balanced tree (so that searching and other operations can be performed in O(logn) time), we will be using the binary search technique. Figure out the middle element and mark it as root. Now the elements on it's left form the left subtree, and elements on the right will form the right subtree. Just put root's left to be the recursive call made on the left portion of the array and root's right to be the recursive call made on the right portion of the array. Try it yourself and refer to the solution tab for code.

BST to sorted LL

Problem statement: Given a BST, we have to construct a sorted linked list out of it.

Approach: As discussed earlier, the inorder traversal of the BST, provides elements in a sorted fashion, so while creating the linked list, we will be traversing the tree in inorder style.

- **Base case:** If the root is NULL, it means the head of the linked list is NULL; hence we return NULL.
- **Small calculation:** Left subtree will provide us the head of LL, and the right subtree will provide the tail of LL; hence root node will be placed after the LL obtained from the left subtree and right LL will be connected after the root.

Left subtree's LL	Root	Right subtree's LL

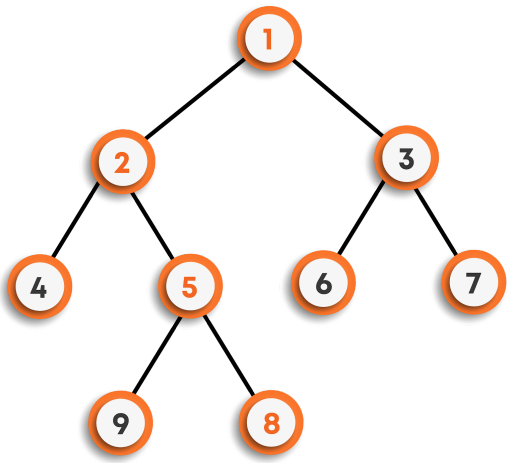
- **Recursive call:** Simply call the left subtree, connect the root node to the end of it, and then connect the right subtree's recursive call after root.

Try it yourselves, and for code, refer to the solution tab of the corresponding question.

Root to node path in a binary tree

Problem statement: Given a Binary tree, we have to return the path of the root node to the given node.

For Example: Refer to the image below...



Path from root to 8: [1 2 5 8]

Approach:

1. Start from the root node and compare it with the target value. If matched, then simply return, otherwise, recursively call over left and right subtrees.
2. Continue this process until the target node is found and return if found.
3. While returning, you need to store all the nodes that you have traversed on the path from the root to the target node in a vector.
4. Now, in the end, you will be having your solution vector.

Code:

```
vector<int>* getRootToNodePath(BinaryTreeNode<int>* root, int data) {
        if (root == NULL) {                    // Base Case:
                return NULL;
        }
```

```
        if (root->data == data) {                // Small calculation part: when node found
                vector<int>* output = new vector<int>();
                output->push_back(root->data); // inserted the node in solution vector
                return output;
        }
        // getting output vector out of left subtree
        vector<int>* leftOutput = getRootToNodePath(root->left, data);
        if (leftOutput != NULL) {
                leftOutput->push_back(root->data);
                return leftOutput;
        }
       // getting output vector out of right subtree
        vector<int>* rightOutput = getRootToNodePath(root->right, data);
        if (rightOutput != NULL) {
                        rightOutput->push_back(root->data);
                        return rightOutput;
        } else {
                return NULL;
        }
}
```

Now try to code the same problem with BST instead of a binary tree.

BST class

Here, we will be creating our own BST class to perform operations like insertion, deletion, etc...

Kindly, follow the code below:

```
#include <iostream>
using namespace std;
class BST {
        BinaryTreeNode<int>* root;              // root node

        public:

        BST() {                                 // Constructor to initialize root to NULL
                root = NULL;
        }
```

```
        ~BST() {                                    // Destructor to delete the BST
                delete root;
        }

        private:
        bool hasData(int data, BinaryTreeNode<int>* node) {     // function to detect
                if (node == NULL) {                             // the presence of a
                        return false;                           // node in BST
                }

                if (node->data == data) {
                        return true;
                } else if (data < node->data) {
                        return hasData(data, node->left);
                } else {
                        return hasData(data, node->right);
                }
        }

        public:
        bool hasData(int data) {
                return hasData(data, root);             // from here the value is returned
        }
};
```

You can observe that hasData() function has been declared under the private section to prevent it from being manipulated, presenting the concept of data abstraction.

Try insertion and deletion on your own, and in case of any difficulty, refer to the hints and solution code below...

Insertion in BST:

We are given the root of the tree and the data to be inserted. Follow the same approach to insert the data as discussed above using Binary search algorithm. Check the code below for insertion:

```
private:
        BinaryTreeNode<int>* insert(int data, BinaryTreeNode<int>* node) {
                // Using Binary Search algorithm
                if (node == NULL) {
```

```
                BinaryTreeNode<int>* newNode = new BinaryTreeNode<int>(data);
                return newNode;
        }

        if (data < node->data) {
                node->left = insert(data, node->left);
        } else {
                node->right = insert(data, node->right);
        }
        return node;
}

public:
void insert(int data) {                                    // Insertion function
        this->root = insert(data, this->root);
}
```

Deletion in BST:

Recursively, find the node to be deleted.

- **Case 1:** If the node to be deleted is the leaf node, then simply delete that node with no further changes and return NULL.
- **Case 2:** If the node to be deleted has only one child, then delete that node and return the child node.
- **Case 3:** If the node to be deleted has both the child nodes, then we have to delete the node such that the properties of BST remain unchanged. For this, we will replace the node's data with either the left child's largest node or right child's smallest node and then simply delete the replaced node.

Now, let's look at the code below:

```
private:
        BinaryTreeNode<int>* deleteData(int data, BinaryTreeNode<int>* node) {
                if (node == NULL) {                // Base case
                        return NULL;
                }
               // Finding that node by traversing the tree
                if (data > node->data) {
```

```
            node->right = deleteData(data, node->right);
            return node;
        } else if (data < node->data) {
            node->left = deleteData(data, node->left);
            return node;
        } else {    // found the node
            if (node->left == NULL && node->right == NULL) {  // Leaf node
                delete node;
                return NULL;
            } else if (node->left == NULL) {  // node having only left child
                BinaryTreeNode<int>* temp = node->right;
                node->right = NULL;
                delete node;
                return temp;
            } else if (node->right == NULL) {    // node having only right child
                BinaryTreeNode<int>* temp = node->left;
                node->left = NULL;
                delete node;
                return temp;
            } else {            // node having both the childs
                BinaryTreeNode<int>* minNode = node->right;
                while (minNode->left != NULL) {   // replacing node with
                    minNode = minNode->left;  // right subtree's min
                }
                int rightMin = minNode->data;
                node->data = rightMin;
               // now simply deleting that replaced node using recursion
                node->right = deleteData(rightMin, node->right);
                return node;
            }
        }
    }

    public:
    void deleteData(int data) {                    // Function to delete
        root = deleteData(data, root);
    }
```

Types of balanced BSTs

For a balanced BST:

|Height_of_left_subtree - Height_of_right_subtree| <= 1

This equation must be valid for each and every node present in the BST.

By mathematical calculations, it was found that the height of a Balanced BST is log(n), where n is the number of nodes in the tree.

This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in O(logn).

There are many Binary search types that maintain balance. We will not be discussing them over here. These are as follows:

- AVL Trees (also known as self-balancing BST, uses rotation to balance)
- Red-Black Trees
- 2 - 4 Tree

Practice Problems:

- **https://www.hackerearth.com/practice/data-structures/trees/binary-search-tree/practice-problems/algorithm/dummy3-4/**
- **https://www.codechef.com/problems/KJCP01**
- **https://www.codechef.com/problems/BEARSEG**

C++ Foundation with Data Structures

Lecture 15 : Priority Queues

Implementing Min Priority Queue :

A priority queue is just like a normal queue data structure except that each element inserted is associated with a “priority”.
It supports the usual push(), pop(), top() etc operations, but is specifically designed so that its first element is always the greatest of the elements it contains, i.e. max heap.

In STL, priority queues take three template parameters:

```
template <class T,
        class Container = vector<T>,
        class Compare = less<typename Container::value_type>
        >
```

class priority_queue;

- The first element of the template defines the class of each element. It can be user-defined classes or primitive data-types. Like in you case it can be int, float or double.
- The second element defines the container to be used to store the elements. The standard container classes std::vector and std::dequeue fulfil these requirements. It is usually the vector of the class defined in the first argument. Like in your case it can be vector<int>, vector<float>, vector<double>.
- The third element is the comparative class. By default it is less<T> but can be changed to suit your need. For min heap it can be changed to greater<T>.

Example

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
   priority_queue<int, vector<int>, greater<int> > pq;
   pq.push(40);
```

```
    pq.push(320);
    pq.push(42);
    pq.push(42);
    pq.push(65);
    pq.push(12);
    pq.push(245);
    cout << pq.top() << endl;
    return 0;
}
```

The above code used the greater<T> functional. Below is the code using a comparative class which performs operator overloading. The code below will make it clear.

```
#include<iostream>
#include <queue>
using namespace std;
class Comp{
    public:
    bool operator () (int a, int b) {
        return a > b;
    }
};

int main() {
    priority_queue<int, vector<int>, Comp> pq;
    pq.push(40);
    pq.push(320);
    pq.push(42);
    pq.push(42);
    pq.push(65);
    pq.push(12);
    pq.push(245);
    cout<<pq.top()<<endl;
    return 0;
}
```

The output for both the code will be 12.

The priority_queue uses the function inside Comp class to maintain the elements sorted in a way that preserves *heap properties*(i.e., that the element popped is the last according to this *strict weak ordering*).

In above example we have used custom function which will make the heap as min-heap.

Hashmaps

Introduction to hashmaps

Suppose we are given a string or a character array and asked to find the maximum occurring character. It could be quickly done using arrays. We can simply create an array of size 256, initialize this array to zero, and then, simply traverse the array to increase the count of each character against its ASCII value in the frequency array. In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called hashmaps.

In hashmaps, the data is stored in the form of keys against which some value is assigned. Keys and values don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example: The given string array is:

str[] = {"abc", "def", "ab", "abc", "def", "abc"}

Hashmap will look like as follows:

Key (datatype = string)	Value (datatype = int)
“abc”	3
“def”	2
“ab”	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: *One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.*

The values are stored corresponding to their respective keys and can be invoked using these keys. To insert, we can do the following:

- hashmap[key] = value
- hashmap.insert(key, value)

The functions that are required for the hashmaps are(using templates):

- **insert(k key, v value):** To insert the value of type v against the key of type k.
- **getValue(k key):** To get the value stored against the key of type k.
- **deleteKey(k key):** To delete the key of type k, and hence the value corresponding to it.

To implement the hashmaps, we can use the following data structures:

1. **Linked Lists:** To perform insertion, deletion, and search operations in the linked list, the time complexity will be O(n) for each as:
    - For insertion, we will first have to check if the key already exists or not, if it exists, then we just have to update the value stored corresponding to that key.

- For search and deletion, we will be traversing the length of the linked list.

2. **BST:** We will be using some kind of a balanced BST so that the height remains of the order O(logN). For using a BST, we will need some sort of comparison between the keys. In the case of strings, we can do the same. Hence, insertion, search, and deletion operations are directly proportional to the height of the BST. Thus the time complexity reduces to O(logN) for each.
3. **Hash table:** Using a hash table, the time complexity of insertion, deletion, and search operations, could be improved to O(1) (same as that of arrays). We will study this in further sections.

Inbuilt Hashmap

In the STL, we have two types of hashmaps:

- **map** (uses BST implementation)
- **unordered_map** (uses hash table implementation)

Note: Both have similar functions and similar ways to use, differing only in the time complexities. The time complexity of each of the operations(insertion, deletion, and searching) in the **map** is O(logN), while in the case of **unordered_map**, they are O(1).

unordered_map:

- Header file: **#include<unordered_map>**
- Syntax to declare:

```
unordered_map<datatype_for_keys, datatype_for_values> name;
```

- Operations performed:
    1. **Insertion:** Suppose, we want to insert the string “abc” with the value 1 in the hashmap named *ourmap*, there are two ways to do so:

- Simply create a pair of both and insert in the map using .**insert** function. Syntax:

```
pair<string, int> p = {"abc", 1};
ourmap.insert(p);
```

- An easier way to insert in a map is to insert like arrays. Syntax:

```
ourmap["abc"] = 1;
```

2. **Searching:** Suppose we want to find the value stored in the hashmap against key *"abc"*, there are two ways to do so:
    - As did in insertion like arrays, the same way we can figure out the value stored against the corresponding key. Syntax:

```
int value = ourmap["abc"];
```

    - Using .at() function. Syntax:

```
int value = ourmap.at("abc");
```

Note: If we try to access a key that is not present in the unordered_map, then there are two different outcomes:

- ➢ If we are accessing the value using .at() function, then we will get an error specifying that we are trying to access the value that is not present in the map.
- ➢ On the other hand, if we access the same using square brackets, then by default, a new key will be created with the default value to be 0 against it. This approach will not give any error.

But what if we want to check if the key is present or not on the map? For that, we will be using the .count() function, which tells if the key is present in the map or not. It returns 0, if not present, and 1, if present

Syntax:

```
int isPresent = ourmap.count("ghi");     // isPresent could only be 0 or 1
```

Note: *We can also check the size of the map by using .size() function, which returns the number of key-value pairs present on the map.*

Syntax:

```
int size_of_map = ourmap.size();
```

3. Deletion: Suppose we want to delete the key *"abc"* from the map, we will be using .erase() function.

Syntax:

```
ourmap.erase("abc");
```

Remove Duplicates

Problem statement: Given an array of integers, we need to remove the duplicate values from that array, and the values should be in the same order as present in the array.

For example: Suppose the given array is arr = {1, 3, 6, 2, 4, 1, 4, 2, 3, 2, 4, 6}, answer should be {1, 3, 6, 2, 4}.

Approach: We will add unique values to the vector and then return it. To check for unique values, start traversing the array, and for each array element, check if the value is already present in the map or not. If not, then we will insert that value in the vector and update the map; otherwise, we will proceed to the next index of the array without making any changes.

Let's look at the code for better understanding.

```
vector<int> removeDuplicates(int* a, int size) {
        vector<int> output;                     // to store the unique elements.
        unordered_map<int, bool> seen;          // unordered map created
        for (int i = 0; i < size; i++) {        // traversing the array
                if (seen.count(a[i]) > 0) {     // using .count() function to check if
                        continue;               // the value has already occurred.
                }
                seen[a[i]] = true;              // If not, then updating the map
                output.push_back(a[i]);         // and inserting that value in the vector
        }
        return output;
}
```

Iterators

To iterate over STL containers, we can use iterators. These are independent of the way the data is stored in the container. It just iterates over the desired and returns all the elements one-by-one. For example, consider the following piece of code:

```
unordered_map<string, int> ourmap;
ourmap["abc"] = 1;
ourmap["abc1"] = 2;
ourmap["abc2"] = 3;
ourmap["abc3"] = 4;
ourmap["abc4"] = 5;
ourmap["abc5"] = 6;
```

Now, we want to iterate the above unordered_map. It can be achieved using iterators.

Steps to use iterators:

1. Declare the iterator of type unordered_map with the same set of data types as that of key-value pairs and point it to the beginning of the map.

   ```
   unordered_map<string, int>::iterator it = ourmap.begin();
   ```

Note: *ourmap.begin() represents the starting position of the map, which could be any key in case of unordered_map as there is no specific order of data inserted in it.*

Although, we will be able to cover entire map elements using iterators irrespective of the position that ourmap.begin() points to.

2. Now, simply put the iterator value in the loop until it reaches the end.

```
while (it != ourmap.end()) {
        cout << "Key : " << it->first << " Value: " << it->second << endl;
        it++;       // increasing the iterator so that it points to next value
}
// Here, it->first points to the key and it->second points to the value
// corresponding to it.
```

Similarly, iterators can also be used on the vectors. Since vectors have a specific order of the elements stored in it, so .begin() will always point to the first position of the vector, i.e., the zeroth index. Refer to the code below for a better explanation.

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
v.push_back(5);

 // iterator declared and pointed to the first position
vector<int>::iterator it1 = v.begin();

// Now, using the while() loop to reach every element of the vector
while (it1 != v.end()) {
        cout << *it1 << endl;                   // *it1 prints the value pointed by itertor it1
        it1++;                                  // Incrementing the value of iterator so that
                                                //it points to the further indexes
}
```

Note: Using .find() on map, it returns an iterator to that position. To erase the element from the map, we can directly provide that iterator to the .erase() function. Refer to the code below:

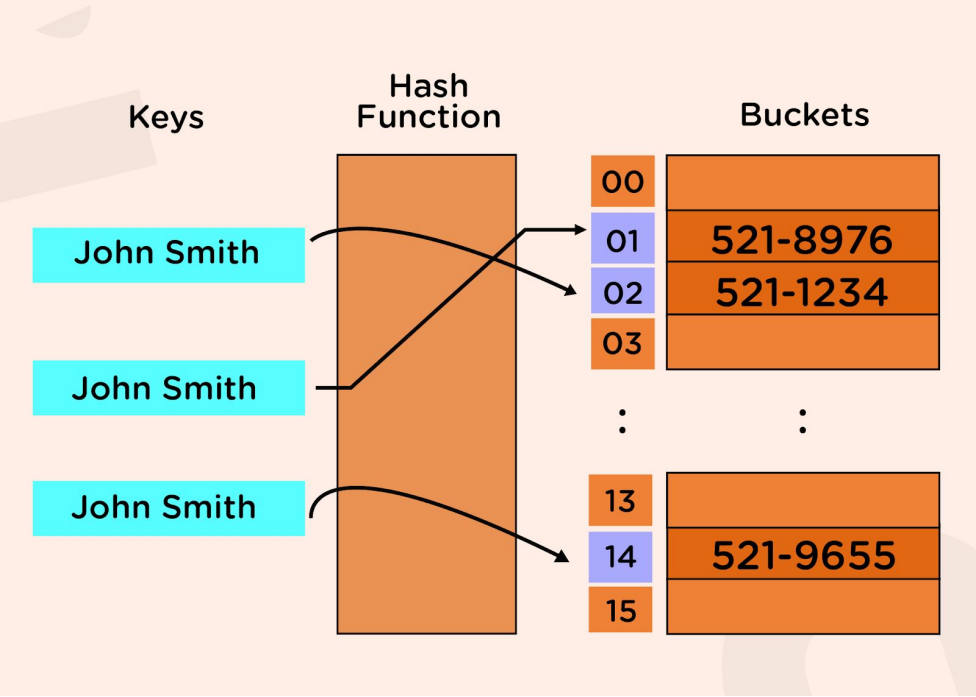
```
unordered_map<string, int>::iterator it2 = ourmap.find("abc");
ourmap.erase(it2);
```

Bucket array and hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Till now, we have seen that arrays are the fastest way to extract data as compared to other data structures as the time complexity of accessing the data in the array is O(1). So we will try to use them in implementing the hashmaps.

Now, we want to store the key-value pairs in an array, named as a **bucket array**. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a **hash function**. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following the image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using hashcode. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (% bucket_size).

One example of a hash code could be: (Example input: "abcd")

"abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0)
Where p is generally taken as a prime number so that they are well distributed.

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = "ab" and s2 = "cd". Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

Compression_function1 = 292 % 2 = 0

Compression_function2 = 298 % 2 = 0

This means they both lead to the same index 0.

This is known as a **collision**.

Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as **separate chaining**.

On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair over that index. If not, then will we find an alternate position for the same. To find the alternate position, we can use the following:

$\mathbf{h_i(a) = hf(a) + f(i)}$

Where hf(a) is the original hash function, and f(i) is the i-th try over the hash function to obtain the final position $h_i(a)$.

To figure out this f(i), following are some of the techniques:

1. **Linear probing:** In this method, we will linearly probe to the next slot until we find the empty index. Here, f(i) = i.

2. **Quadratic probing:** As the name suggests, we will look for alternate i^2 positions ahead of the filled ones, i.e., $f(i) = i^2$.
3. **Double hashing:** According to this method, f(i) = i * H(a), where H(a) is some other hash function.

In practice, we generally prefer to use separate chaining over open addressing, as it is easier to implement and is also more efficient.

Let's now implement the hashmap of our own.

Hashmap implementation - Insert

As discussed earlier, we will be implementing separate chaining. We will be using value as a template and key as a string as we are required to find the hash code for the key. Taking key as a template will make it difficult to convert it using hash code.

Let's look at the code for the same.

```
#include <string>
using namespace std;

template <typename V>
class MapNode {                          // class for linked list.
        public:
                string key;              // to store key of type string
                V value;                 // to store value of type template
                MapNode* next;           // to store the next pointer

                MapNode(string key, V value) {      //  constructor to assign values
                        this->key = key;
                        this->value = value;
                        next = NULL;
                }

                ~MapNode() {             // Destructor to delete the node.
                        delete next;
                }
};

template <typename V>
class ourmap {                   //  for storing the bucket array
```

```
        MapNode<V>** buckets;      // a 2D bucket array to store the head pointers
                                   // of the linked list corresponding to each index.
        int count;                 // to store the size
        int numBuckets;     //  to store number of buckets for compression function

        public:
        ourmap() {      // constructor: to initialize the values
                count = 0;     //
                numBuckets = 5;
                buckets = new MapNode<V>*[numBuckets];  // dynamically allocated
                for (int i = 0; i < numBuckets; i++) {
                        buckets[i] = NULL;   // assigning each head pointer to NULL
                }
        }

        ~ourmap() {   // destructor: to delete the storage used
                for (int i = 0; i < numBuckets; i++) {   // first-of-all delete each linked list
                        delete buckets[i];
                }
                delete [] buckets;    // the delete the total bucket
        }

        int size() {        // to return the size of the map
                return count;
        }

        V getValue(string key) {// for returning the value corresponding to a key
                // will see in the next section
        }

        private:
        int getBucketIndex(string key) {    // to provide the index using hash function
                int hashCode = 0;
                int currentCoeff = 1;
// using "abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0) as our hash code
                for (int i = key.length() - 1; i >= 0; i--) {
                        hashCode += key[i] * currentCoeff;
                        hashCode = hashCode % numBuckets;
                        currentCoeff *= 37;       // taking p = 37
                        currentCoeff = currentCoeff % numBuckets;
                }

                return hashCode % numBuckets;
        }

        public:
        void insert(string key, V value) {
```

```
    // To get the bucket index, i.e., passing it through the hash function
            int bucketIndex = getBucketIndex(key);
//  to insert the key-value pair in the linked list corresponding to index obtained
            MapNode<V>* head = buckets[bucketIndex];
            while (head != NULL) {
                if (head->key == key) {    // If key already present, then we are
                    head->value = value;  // just updating the value against it
                    return;
                }
                head = head->next;
            }
// otherwise creating a new node and inserting that node before head so making it
// as the head and marking the bucket index to this node as the new head
            head = buckets[bucketIndex];
            MapNode<V>* node = new MapNode<V>(key, value);
            node->next = head;
            buckets[bucketIndex] = node;
            count++;
        }
};
```

Hashmap implementation - Delete and search

Refer to the code below and follow the comments in it.

```
        // to search for the key
        V getValue(string key) {
            int bucketIndex = getBucketIndex(key);  // find the index
            MapNode<V>* head = buckets[bucketIndex];    // head of linked list
            while (head != NULL) {
                if (head->key == key) {     //  if found, returned the value
                    return head->value;
                }
                head = head->next;
            }
            return 0;   //  if not found, returning 0 as default value.
        }
       // to delete the key-value pair
        V remove(string key) {
            int bucketIndex = getBucketIndex(key); // find the index
            MapNode<V>* head = buckets[bucketIndex]; // head node
            MapNode<V>* prev = NULL; // previous pointer
            while (head != NULL) {
```

```
                    if (head->key == key) {
                        if (prev == NULL) {
                            buckets[bucketIndex] = head->next;
                        } else {
                            prev->next = head->next;   // connecting previous
                                                       // to the head's next pointer
                        }
                        V value = head->value;
                        head->next = NULL; // before calling delete over it, in
// order to avoid complete linked list deletion, we have to assign it's next to NULL
                        delete head;
                        count--;       // reducing the size
                        return value;// return the value stored at deleted node
                    }
                    prev = head;
                    head = head->next;
                }
                return 0;   // means that value not found
            }
```

Time Complexity and Load Factor

Let's define specific terms before moving forward:

1. n = Number of entries in our map.
2. l = length of the word (in case of strings)
3. b = number of buckets. On average, each box contains (n/b) entries. This is known as **load factor** means b boxes contain n entries. We also need to ensure that the load factor is always less than 0.7, i.e.,

   (n/b) < 0.7, this will ensure that each bucket does not contain too many entries in it.
4. To make sure that load factor < 0.7, we can't reduce the number of entries, but we can increase the bucket size comparatively to maintain the ratio. This process is known as **Rehashing**.

This ensures that time complexity is on an average O(1) for insertion, deletion, and search operations each.

Rehashing

Now, we will try to implement the rehashing in our map. After inserting each element into the map, we will check the load factor. If the load factor's value is greater than 0.7, then we will rehash.

Refer to the code below for better understanding.

```
void rehash() {
        MapNode<V>** temp = buckets;   // To store the old bucket
        buckets = new MapNode<V>*[2 * numBuckets];  // doubling the size
        for (int i = 0; i < 2*numBuckets; i++) {
                buckets[i] = NULL;      // initialising each head pointer to NULL
        }
        int oldBucketCount = numBuckets;
        numBuckets *= 2;              // updating new size
        count = 0;
        for (int i = 0; i < oldBucketCount; i++) {
                MapNode<V>* head = temp[i];
                while (head != NULL) {
                        string key = head->key;
                        V value = head->value;
                        insert(key, value);   // inserting each value of old bucket
                                              // into the new one
                        head = head->next;
                }
        }
        // deleting the old bucket
        for (int i = 0; i < oldBucketCount; i++) {
                MapNode<V>* head = temp[i];
                delete head;
        }
        delete [] temp;
}
void insert(string key, V value) {
        int bucketIndex = getBucketIndex(key);
        MapNode<V>* head = buckets[bucketIndex];
        while (head != NULL) {
                if (head->key == key) {
                        head->value = value;
```

```
                return;
            }
            head = head->next;
        }
        head = buckets[bucketIndex];
        MapNode<V>* node = new MapNode<V>(key, value);
        node->next = head;
        buckets[bucketIndex] = node;
        count++;
        // Now we will check the load factor after insertion.
        double loadFactor = (1.0 * count)/numBuckets;
        if (loadFactor > 0.7) {
            rehash();      // We will rehash.
        }
    }
```

Note: While solving the problems, we will be using the in-built hashmap only.

Practice problems:

- https://www.codechef.com/problems/STEM
- https://codeforces.com/problemset/problem/525/A
- https://www.spoj.com/problems/ADACLEAN/

Tries and Huffman Coding

Introduction to tries

Suppose we want to implement a word-dictionary using a C++ program and perform the following functions:

- Addition of the word(s)
- Searching a word
- Removal of the word(s)

To do the same, hashmaps can be thought of as an efficient data structure as the **average case time complexity** of insertion, deletion, and retrieval is O(1) for integer, character, float, and decimal values.

Let us discuss the time complexity of the same in case of strings.

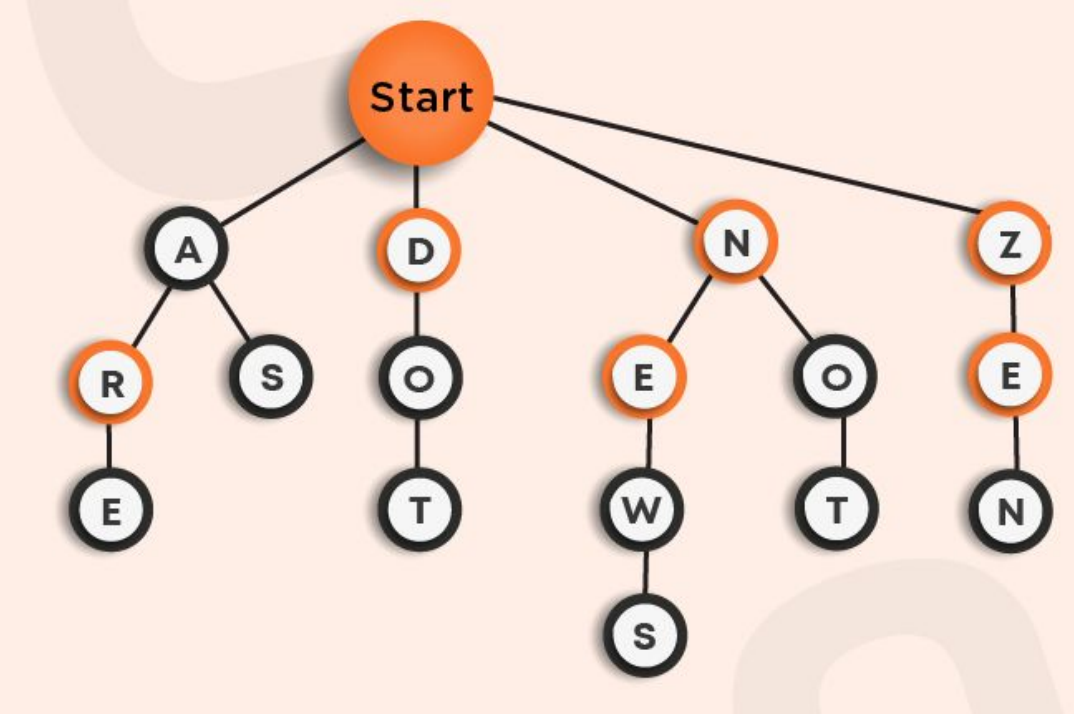
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above operations. Thus, we can conclude that the insertion of a string will be O(string_length).

To search a word in the hashmap, we again have to calculate the hashcode of the string to be searched, and for that also, it would require O(string_length) time.

Similarly, for removing a word from the hashmap, we would have to calculate the hashcode for that string. It would again require O(string_length) time.

For the same purpose, we can use another data structure known as **tries**.

Below is the visual representation of the trie:



Here, the node at the top named as the **start** is the root node of our **n-ary tree.**

Suppose we want to insert the word ***ARE*** in the trie. We will begin from the root, search for the first word ***A*** and then insert ***R*** as its child and similarly insert the letter ***E***. You can see it as the left-most path in the above trie.

Now, we want to insert the word ***AS*** in the trie. We will follow the same procedure as above. First-of-all, we find the letter ***A*** as the child of the root node, and then we will search for ***S***. If ***S*** was already present as the child of ***A***, then we will do nothing as the given word is already present otherwise, we will insert ***S***. You can see this in the above trie. Similarly, we added other words in the trie.

This approach also takes O(word_length) time for insertion.

Moving on to searching a word in the trie, for that also, we would have to traverse the complete length of the string, which would take O(word_length) time.

Let us take an example. Suppose we want to search for the word ***DOT*** in the above trie, we will first-of-all search for the letter ***D*** as the direct child of the start node and then search for ***O*** and ***T*** and, then we will return true as the word is present in the trie.

Consider another example ***AR***, which we want to search for in the given trie. Following the above approach, we would return true as the given word is present in it. However ideally, we should return false as the actual word was ***ARE*** and not ***AR***. To overcome this, it can be observed that in the above trie, some of the letters are marked with a bolder circle, and others are not. This boldness represents the termination of a word starting from the root node. Hence, while searching for a word, we will always be careful about the last letter that it should be bolded to represent the termination.

Note: While insertion in a trie, the last letter of the word should be bolded as a mark of termination of a word.

In the above trie, the following are all possibilities that can occur:

- ARE, AS
- DO, DOT
- NEW, NEWS, NOT
- ZEN

Here, to remove the word, we will simply unbold the terminating letter as it will make that word invalid. For example: If you want to remove the string ***DO*** from the above trie by preserving the occurrence of string DOT, then we will reach ***O*** and then unbolden it. This way the word ***DO*** is removed but at the same time, another word ***DOT*** which was on the same path as that of word ***DO*** was still preserved in the trie structure.

For removal of a word from trie, the time complexity is still O(word_length) as we are traversing the complete length of the word to reach the last letter to unbold it.

When to use tries over hashmaps?

It can be observed that using tries, we can improve the space complexity.

For example: We have 1000 words starting from character ***A*** that we want to store. Now, if you try to hold these words using hashmap, then for each word, we have to store character ***A*** differently. But in case of tries, we only need to store the character ***A*** once. In this way, we can save a lot of space, and hence space optimization leads us to prefer tries over hashmaps in such scenarios.

In the beginning, we thought of implementing a dictionary. Let's recall a feature of a dictionary, namely **Auto-Search.** While browsing the dictionary, we start by typing a character. All the words beginning from that character appear in the search list. But this functionality can't be achieved using hashmaps as in the hashmap, the data stored is independent of each other, whereas, in case of tries, the data is stored in the form of a tree-like structure. Hence, here also, tries prove to be efficient over hashmaps.

TrieNode class implementation

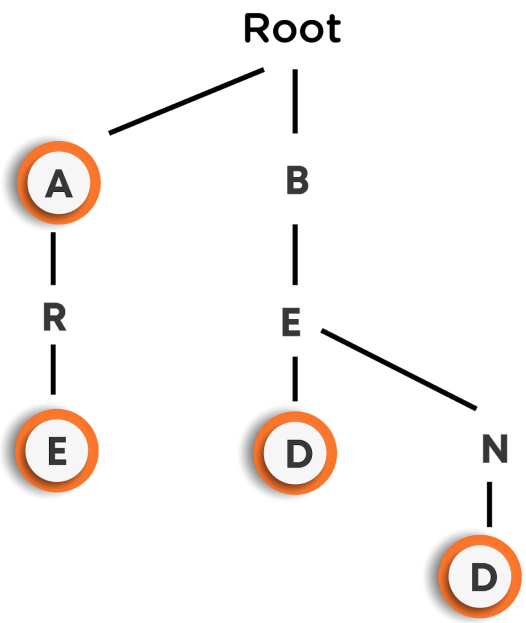
Follow the below-mentioned code (with comments)...

```
class TrieNode {
        public :
        char data;                              // To store data (character value: 'A' - 'Z')
        TreeNode **children;                    // To store the address of each child
        bool isTerminal;     // it will be TRUE if the word terminates at this character

        TrieNode(char data) {                   // Constructor for values initialization
                this -> data = data;
                children = new TrieNode*[26];
                for(int i = 0; i < 26; i++) {
                        children[i] = NULL;
                }
```

```
            isTerminal = false;
        }
};
```

Insert function

To insert a word in a trie, we will use recursion. Suppose we have the following trie:



Now we want to insert the word ***BET*** in our trie.

Recursion says that we need to work on a smaller problem, and the rest it will handle itself. So, we will do it on the root node.

Note: Practically, the functionality of bolding the terminal character is achieved using the boolean **isTerminal** variable for that particular node. If this value is true means that the node is the terminal value of the string, otherwise not.

Approach:

We will first search for letter ***B*** and check if it is present as the children of the root node or not and then call recursion on it. If ***B*** is present as a child of the root node as in our case it is, then we will simply recurse over it by shifting the length of the string by 1. In case, character ***B*** was not the direct child of the root node, then we have to create one and then call recursion on it. After the recursive call, we will see that be is now a root node, and the character ***E*** is now the word we are searching for. We will follow the same procedure as done in searching for character ***B*** against the root node and then move forward to the next character of the string, i.e., ***T,*** which happens to be the last character of our string. Now we will check character ***T*** as the child of character ***E***. In our case, it is not a child of character ***E***, so we'll create it. As ***T*** is the last character of the string, so we will mark its **isTerminal** value to **True**.

Following will be the three steps of recursion:

- **Small Calculation:** We will check if the root node has the first character of the string as one of its children or not. If not, then we will create one.
- **Recursive call:** We will tell the recursion to insert the remaining string in the subtree of our trie.
- **Base Case:** As the length of the string becomes zero, or in other words, we reach the NULL character, then we need to mark the **isTerminal** for the last character as True.

Follow the code below, along with the comments...

```
class Trie {
        TrieNode *root;

        public :
        Trie() {
                root = new TrieNode('\0');
        }

        void insertWord(TrieNode *root, string word) {
                // Base case
```

```
            if(word.size() == 0) {
                root -> isTerminal = true;
                return;
            }
            // Small Calculation
            int index = word[0] - 'a';      // As for 'a' refers to index 0, 'b' refers to
                // index 2 and so on, so to reach the correct index we will do so
            TrieNode *child;
            if(root -> children[index] != NULL) {   // If the first character of string is
                                // already present as the child node of the root node
                child = root -> children[index];
            }
            else {      // If not present as the child then creating one.
                child = new TrieNode(word[0]);
                root -> children[index] = child;
            }

            // Recursive call
            insertWord(child, word.substr(1));
        }
        // For user
        void insertWord(string word) {
            insertWord(root, word);
        }
};
```

Search in tries

Objective: Create a search function which will get a string as its argument to be searched in the trie and returns a boolean value. **True** if the string is present in the trie and **False** if not.

Approach: We will be using the same process as that used during insertion. We will call recursion over the root node after searching for the first character as its child. If the first character is not present as one of the children of the root node, then we will simply return false; otherwise, we will send the remaining string to the recursion. When we reach the empty string, then check for the last character's

isTerminal value; if it is **true,** means that word exists in our trie, and we will return true from there otherwise, return false.

Try to code it yourselves, and for the code, refer to the solution tab of the same.

Tries implementation: Remove

Objective: To delete the given word from our trie.

Approach: First-of-all we need to search for the word in the trie, and if found, then we simply need to mark the **isTerminal** value of the last character of the word to **false** as that will simply denote that the word does not exist in our trie.

Check out the code below: (Nearly same as that of insertion, just a minor changes which are explained along side)

```
void removeWord(TrieNode *root, string word) {
        // Base case
        if(word.size() == 0) {
                root -> isTerminal = false;
                return;
        }

        // Small calculation
        TrieNode *child;
        int index =  word[0] - 'a';
        if(root -> children[index] != NULL) {
                child = root -> children[index];
        }
        else {
                // Word not found
                return;
        }

        removeWord(child, word.substr(1));
// Suppose if the character of the string doesn't have any child and is a part of the word to be deleted, then we can simply delete that node also as it is not referencing to any other word in the trie

        // Removing child Node if it is useless
```

```
            if(child -> isTerminal == false) {
                for(int i = 0; i < 26; i++) {
                    if(child -> children[i] != NULL) {
                        return;
                    }
                }
                delete child;
                root -> children[index] = NULL;
            }
        }

        // For user
        void removeWord(string word) {
            removeWord(root, word);
        }
```

Types of tries

There are two types of tries:

- **Compressed tries:**
    - Majorly, used for space optimization.
    - We generally club the characters if they have at most one child.
    - **General rule:** Every node has at least two child nodes.

    Refer to the figure below:

    Suppose our regular trie looks like this-

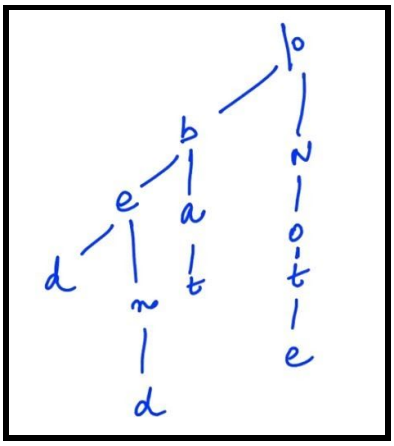
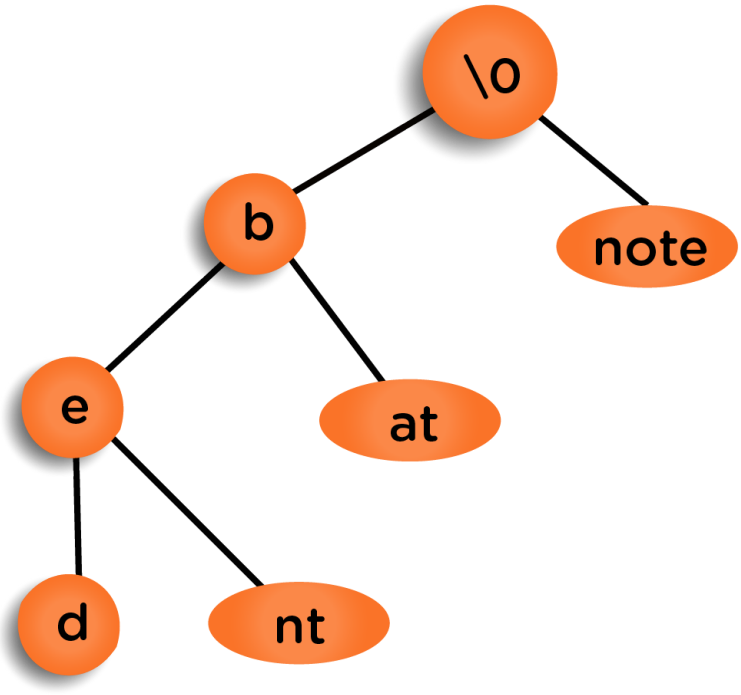


Figure - 1

Its compressed trie version will look as follows:



- **Pattern matching:**
    - Used to match patterns in the trie.

- Example: In the figure-1 (shown above), if we want to search for pattern **ben** in the trie, but the word **bend** was present instead, using the normal search function, we would return false, as the last character **n**'s isTerminal property was false, but in this trie, we would return true.

- To overcome this problem of the last character's identification, just remove the **isTerminal** property of the node.
- In the figure-1, instead of searching for the pattern **ben**, we now want to search for the pattern **en**. Our trie would return false if **en** is not directly connected to the root. But as the pattern is present in the word **ben**, it should return true. To overcome this problem, we need to attach each of the prefix strings to the root node so that every pattern is encountered.
    - **For example:** for the string **ben,** we would store **ben**, **en**, **n** in the trie as the direct children of the root node.

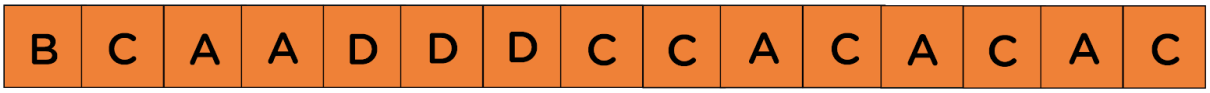
Huffman Coding

Huffman Coding is one approach followed for **Text Compression.** Text compression means reducing the space requirement for saving a particular text.

Huffman Coding is a lossless data compression algorithm, ie. it is a way of compressing data without the data losing any information in the process. It is useful in cases where there is a series of frequently occurring characters.

Working of Huffman Algorithm:

Suppose, the given string is:



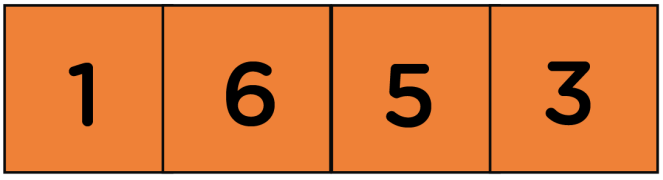
Intial String

Here, each of the characters of the string takes 8 bits of memory. Since there are a total of 15 characters in the string so the total memory consumption will be 15*8 = 120 bits. Let's try to compress its size using the Huffman Algorithm.

First-of-all, the Huffman Coding creates a tree by calculating the frequencies of each character of the string and then assigns them some unique code so that we can retrieve the data back using these codes.

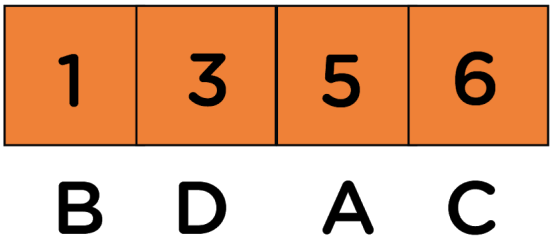
Follow the steps below:

1. Begin with calculating the frequency of each character value in the given string.



Frequency of String

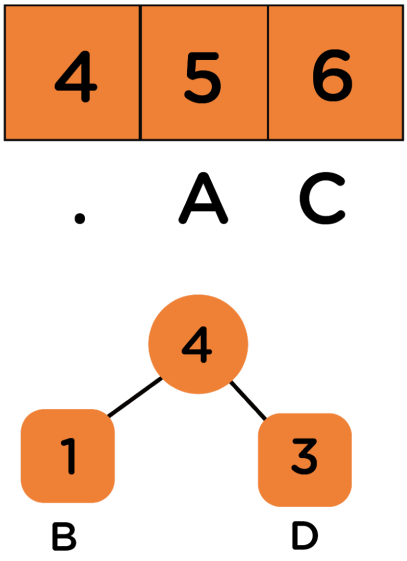
2. Sort the characters in ascending order concerning their frequency and store them in a priority queue, say **Q.**
3. Each character should be considered as



Characters sorted according to the frequency

a different leaf node.

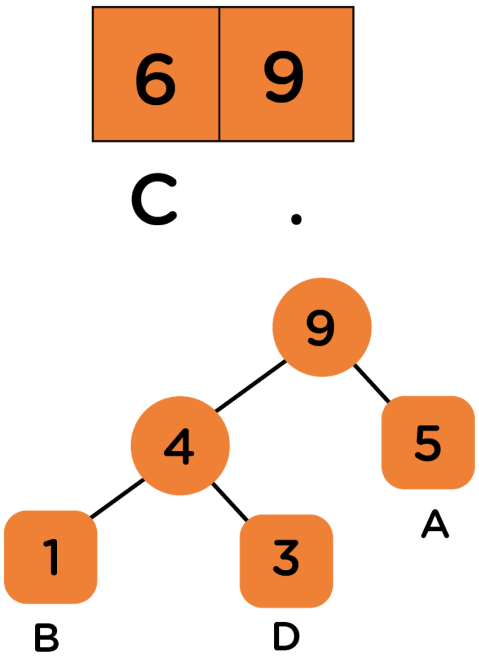
4. Make an empty node, say **z**. The left child of z marked as the minimum frequency and right child, the second minimum frequency. The value of z is calculated by summing up the first two frequencies.



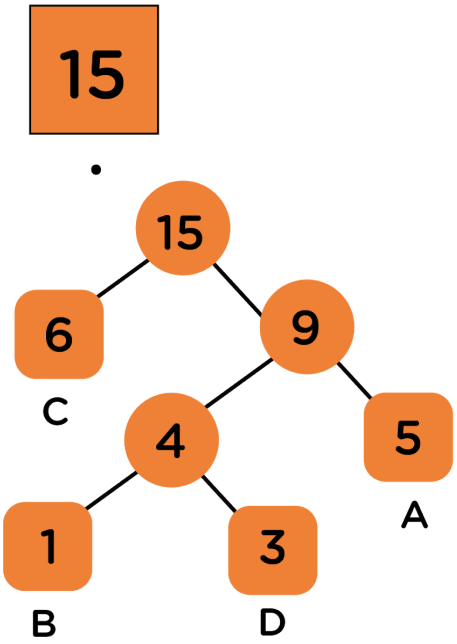
Getting the sum of the least numbers

Here, * denote the internal nodes.

5. Now, remove the two characters with the lowest frequencies from the priority queue Q and append their sum to the same.
6. Simply insert the above node z to the tree.
7. For every character in the string, repeat steps 3 to 5.

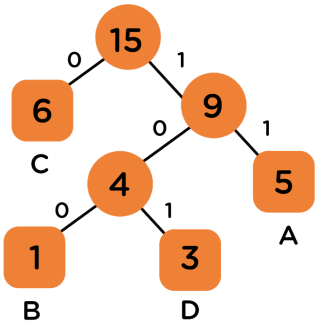


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. Assign 0 to the left side and 1 to the right side except for the leaf nodes.



Assign 0 to the left edge and 1 to the right edge

The size table is given below:

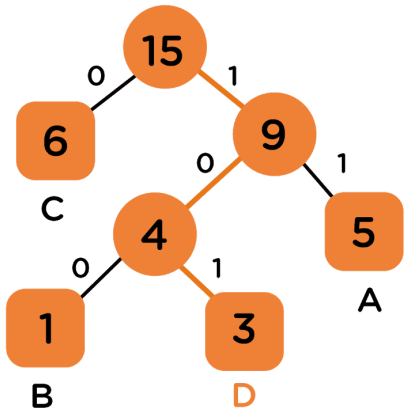
Character	Frequency	Code	Size
A	5	11	5*2 = 10

B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4*8 = 32 bits	15 bits		28 bits

Size before encoding: 120 bits

Size after encoding: 32 + 15 + 28 = 75 bits

To decode the code, simply traverse through the tree (starting from the root) to find the character. Suppose we want to decode 101, then:



Decoding

Time complexity:

In case of encoding, inserting each character into the priority queue takes O(log n) time. Therefore, for the complete array, the time complexity becomes O(nlog n).

Similarly, extraction of the element from the priority queue takes O(log n) time. Hence, for the complete array, the achieved time complexity is O(nlog n).

Applications of Huffman Coding:

- They are used for transmitting fax and text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.

Note: Rat in a Maze - Code Correction

Initialisation of solution 2D array

The code developed in the video may not work on your local IDE. This happens because the solution 2D array is not initialized with default value, before making recursive calls. Hence, to resolve it, please initialize the 2D-array, before initiating the recursive calls. The following code helps you understand where the initialization of solution 2D array has to be done:

```
void ratInAMaze(int maze[][20], int n){

  int** solution = new int*[n];
  for(int i=0;i<n;i++){
      solution[i] = new int[n];
  }
  // initialization of solution 2D arrays goes here.

  mazeHelp(maze,n,solution,0,0);

}
```

Complete code with initialisation of solution 2D array:

```
#include<bits/stdc++.h>
using namespace std;


void printSolution(int** solution,int n){
      for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                  cout << solution[i][j] << " ";
            }
      }
      cout<<endl;
}
void mazeHelp(int maze[][20],int n,int** solution,int x,int y){
```

```
      if(x == n-1 && y == n-1){
            solution[x][y] =1;
            printSolution(solution,n);
            solution[x][y] =0;
            return;
      }
      if(x>=n || x<0 || y>=n || y<0 || maze[x][y] ==0 || solution[x][y]
==1){
            return;
      }
      solution[x][y] = 1;
      mazeHelp(maze,n,solution,x-1,y);
      mazeHelp(maze,n,solution,x+1,y);
      mazeHelp(maze,n,solution,x,y-1);
      mazeHelp(maze,n,solution,x,y+1);
      solution[x][y] = 0;
}
void ratInAMaze(int maze[][20], int n){

  int** solution = new int*[n];
  for(int i=0;i<n;i++){
      solution[i] = new int[n];
  }
  // Initialization of solution 2D array with 0
  for(int i=0; i<n; i++){
    memset(solution[i], 0, n*sizeof(int));
  }

  mazeHelp(maze,n,solution,0,0);

}
```

Dynamic Programming - 1

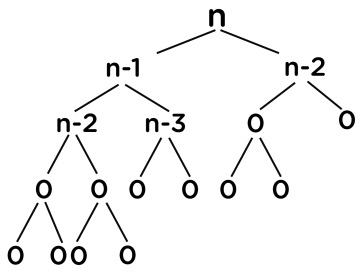
Introduction

Suppose we need to find the n^{th} Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
int fibo(int n) {
        if(n <= 1) {
                return n;
        }
        int a = fibo(n - 1);
        int b = fibo(n - 2);
        return a + b;
}
```

Here, for every n, we need to make a recursive call to f(n-1) and f(n-2).

For f(n-1), we will again make the recursive call to f(n-2) and f(n-3). Similarly, for f(n-2), recursive calls are made on f(n-3) and f(n-4) until we reach the base case. The recursive call diagram will look something like shown below:

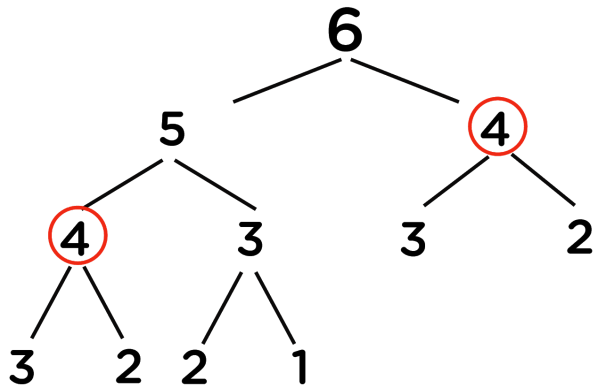


At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one). At every level, we are doing $2^n K$ work (where n = 0, 1, 2, ...). Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work. Total work can be calculated as:

$(2^0 + 2^1 + 2^2 + ... + 2^{n-1}) * k \simeq 2^n k$

Hence, it means time complexity will be $O(2^n)$.

We need to improve this complexity as it is terrible. Let's look at the example below for finding the 6^{th} Fibonacci number.



IMPORTANT OBSERVATION: We can observe that there are repeating recursive calls made over the entire program. As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4)(second similar recursive call over f(4)). Both of these recursive calls are shown above in the outlining circle. Similarly, there are many others for which we are repeating the recursive calls. Generally, in recursion, there are repeated recursive calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered value(preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again. This way, we can improve the running time of our code. This process of storing each recursive call's output and then using

them for further calculations preventing the code from calculating these again is called **memoization**.

To achieve this in our example we will simply take an answer array initialized to -1. Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not. If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value. After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

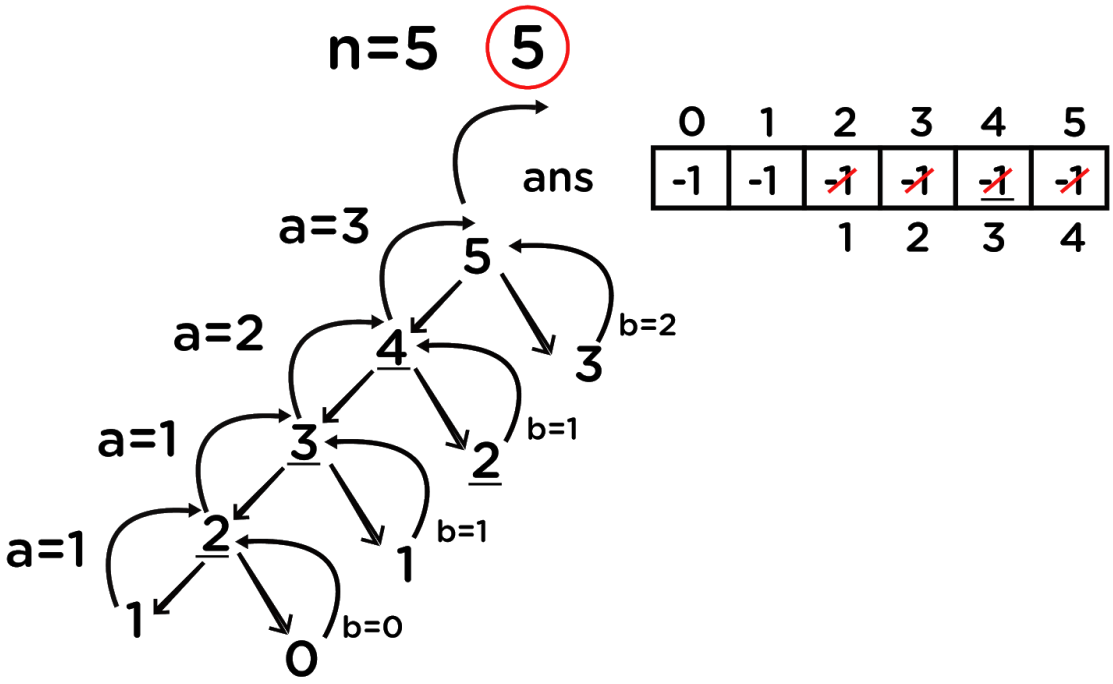
Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only.

Let's look at the memoization code for Fibonacci numbers below:

```
int fibo_helper(int n, int *ans) {
        if(n <= 1) {                            // Base case
                return n;
        }

        // Check if output already exists
        if(ans[n] != -1) {
                return ans[n];
        }

        // Calculate output
        int a = fibo_helper(n-1, ans);
        int b = fibo_helper(n-2, ans);

        // Save the output for future use
        ans[n] = a + b;

        // Return the final output
        return ans[n];
}


int fibo_2(int n) {
        int *ans = new int[n+1];
```

```
        for(int i = 0; i <= n; i++) {          // -1 marks that answer for corresponding
                ans[i] = -1;                   // number does not exist
        }
        return fibo_helper(n, ans);            // Now simply following the memoization
}
```

Let's dry run for n = 5, to get a better understanding:



Again, if we observe, we can see that for any number we are not able to make a recursive call on the right side of it which means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

Summary: Memoization is a **top-down approach** where we save the previous answers so that they can be used to calculate the future answers and improve the time complexity to a greater extent.

Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored. Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes. In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index. This can be simply done iteratively by running a loop from i = (2 to n). Finally, we will get our answer at the 5^{th} index of the answer array as we already know that the i-th index contains the answer to the i-th value.

Simply, we are first trying to figure out the dependency of the current value on the previous values and then using them calculating our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, following a **bottom-up approach** to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP).**

Let us now look at the code for calculating the n^{th} Fibonacci number:

```
int fibo_3(int n) {
        int *ans = new int[n+1];

        ans[0] = 0;             // Storing the independent values in the solution array.
        ans[1] = 1;

        for(int i = 2; i <= n; i++) {          // Following bottom-up approach to reach n
                ans[i] = ans[i-1] + ans[i-2];
        }

        return ans[n];              // final answer
}
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous outputs using memoization.
3. Finally, replace recursion by iteration using dynamic programming. (The best possible solution that seems to appear for every problem when in case of recursion the space complexity is more in comparison to iteration)

Min steps to 1

Let's now solve another problem named **min steps to 1**.

Problem statement: Given a positive integer n, find the minimum number of steps s, that takes n to 1. You can perform any one of the following three steps:

1. Subtract 1 from it. (n = n-1)
2. If its divisible by 2, divide by 2. (if n%2 == 0, then n= n/2) ,
3. If its divisible by 3, divide by 3. (if n%3 == 0, then n = n / 3).

Example 1: For n = 4:

STEP-1: n = 4/2 = 2

STEP-2: n = 2/2 = 1

Hence, the answer is 2.

Example 2: For n = 7:

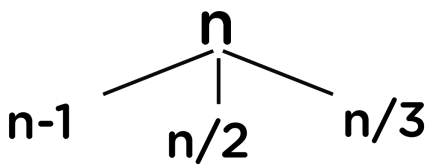
STEP-1: n = 7 - 1 = 6

STEP-2: n = 6/3 = 2

STEP-3: n = 2/2 = 1

Hence, the answer is 3.

Approach: We are only allowed to perform the above three mentioned ways to reduce any number to 1.



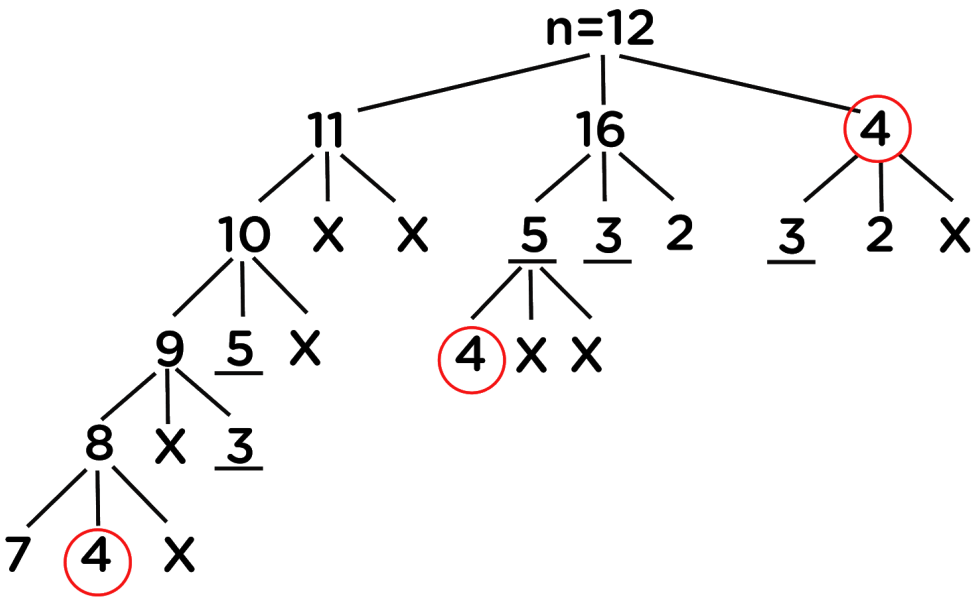
Let's start thinking about the brute-force approach first, i.e., recursion.

We will make a recursive call to each of the three steps keeping in mind that for dividing by 2, the number should be divisible by 2 and similarly for 3 as given in the question statement. After that take minimum value out of the three obtained and simply add 1 to the answer for the current step itself. Thinking about the base case, we can see that that on reaching 1, simply we have to return 0 as it is our destination value. Let's now look at the code:

```
int minSteps(int n) {
        if(n <= 1) {                                    // Base case
                return 0;
        }

        int x = minSteps(n - 1);                        // Recursive call - 1

        int y = INT_MAX, z = INT_MAX;                   // These values are intialized to +infinity
                                                        // so as to check if n is not divisible by 2 or 3
        if(n % 2 == 0) {
                y = minSteps(n/2);                      // Recursive call - 2
        }

        if(n % 3 == 0) {
                z = minSteps(n/3);                      // Recursive call - 3
        }

        // Calculate final output
        int ans = min(x, min(y, z)) + 1;
        return ans;
}
```

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones. To check this let's dry run the problem for n = 12:

(Here X represents that the calls are not feasible as the number is not divisible by either of 2 or 3)



Here, if we blindly make three recursive calls at each step, then the time complexity will approximately be $O(3^n)$.

From the above, it is clearly visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

Now, we need to figure out the number of unique calls, i.e., how many answers we are required to save. It is clear that we need at most n+1 responses to be saved, starting from n = 0, and the final answer will be present at index n.

The code will be nearly the same as the recursive approach; just we will not be making recursive calls for already stored outputs. Follow the code and comments below:

```
int minStepsHelper(int n, int *ans) {
        if(n <= 1) {                                    // Base case
                return 0;
        }

        if(ans[n] != -1) {                              // Check if output already exists
                return ans[n];
        }

        int x = minStepsHelper(n - 1, ans);   // Calculate output - 1

        int y = INT_MAX, z = INT_MAX;
        if(n % 2 == 0) {
                y = minStepsHelper(n/2, ans);       // Calculate output - 2
        }

        if(n % 3 == 0) {
                z = minStepsHelper(n/3, ans);     // Calculate output - 3
        }

        int output = min(x, min(y, z)) + 1;
        ans[n] = output;                                // Save output for future use

        return output;

}

int misSteps_2(int n) {
        int *ans = new int[n+1];

        for(int i = 0; i <= n; i++) {   // initialized to -1 denoting answer is
                ans[i] = -1;            // unknown at current stage for the particular index
        }

        return minStepsHelper(n, ans);
}
```

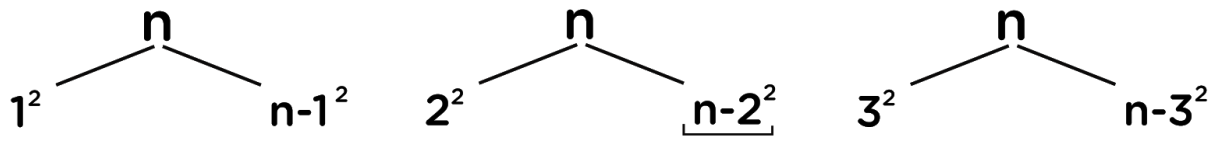
Time complexity has been reduced significantly to O(n) as there are only (n+1) unique iterations. Now, try to code the DP approach by yourself, and for the code, refer to the solution tab of the same.

Minimum Count

Problem statement: Given an integer N, find and return the count of minimum numbers, the sum of whose squares is equal to N.
That is, if N is 4, then we can represent it as : {1^2 + 1^2 + 1^2 + 1^2} and {2^2}. The output will be 1, as 1 is the minimum count of numbers required. (x^y represents x raised to the power y.)
Example: For n = 12, we have the following ways:

- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1
- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 2^2
- 1^1 + 1^1 + 1^1 + 1^1 + 2^2 + 2^2
- 2^2 + 2^2 + 2^2

Hence, the minimum count is obtained from the 4-th option. Therefore, the answer is equal to 3.
Approach: First-of-all, we need to think about breaking the problems into two parts, one of which will be handled by recursion and the other one will be handled by us(smaller sub-problem). We can break the problem as follows:



And so on...

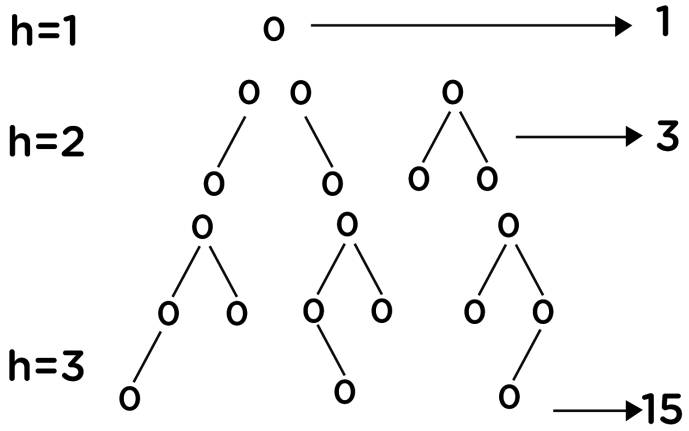
- In the above figure, it is clear that in the left subtree we are making ourselves try over a variety of values that can be included as a part of our solution.
- This left subtree's calculation will be done by us and the right subtree will be done by recursion.

- Hence, we will just handle the i^2 part and ($n-i^2$) will be handled by recursion.
- As by now, we have got ourselves an idea of solving this problem, the only thinking left is the loop's range on which we will be iterating, i.e., the values of **i** for which we will be deciding to consider while solving or not.
- As, the maximum value upto which i can be pushed to in order to reach n is √n as (√n*√n = n). Hence, we will be iterating over the range (1 to √n) and do consider each possible way by sending ($n-i^2$) over the recursion.
- This way we will get different subsequences and as per the question, we will simply return the minimum out of it.

This problem is left for you to try out using all the three approaches and for code, refer to the solution tab of the same.

No. of balanced BTs

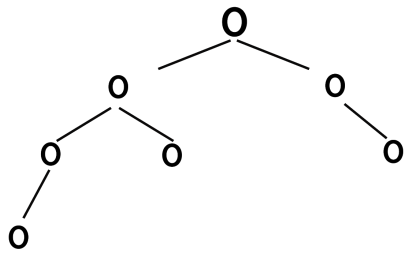
Problem Statement: Given an integer h, find the possible number of balanced binary trees of height h. You just need to return the count of possible binary trees which are balanced. This number can be huge, so return output modulo $10^9 + 7$.
For Example: In the figure below, the left side represents the height h, and the right side represents the possible binary trees along with the count.



Here for h = 1, the answer is 1. For h = 2, the answer is 3. For h = 3, the answer is 15.

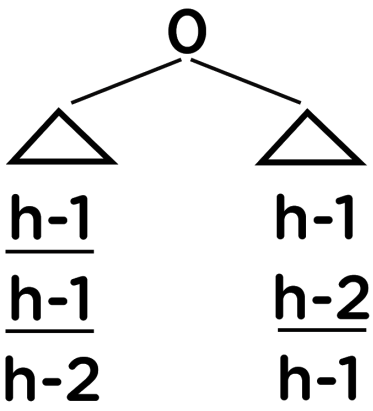
Approach: Suppose we have h = 3, so at level 3 there are four nodes and each node has two options, either it can be included or excluded from the binary tree, hence in total, we have 2^4 = 16 possibilities. Here, we need to exclude the possibility of the case when none out of 4 is present. Hence, the remaining options are 16-1 = 15. We can think that this approach could be efficient as we just need to know the number of nodes at the last level, and then we can simply apply the above formula.

Now, consider for h = 4, where the last level has got eight nodes, so according to the above approach, the answer could be 2^8 - 1 = 255 ways, but the solution for h = 4 is 315. Let's look at the cases which we missed. One of the examples could be:



Till now, we were only working over the last level, but in the above example, if we remove the nodes from upper levels, then also the binary tree could remain balanced.

Let's now think about implementing it using recursion on trees. If the height of the complete binary tree is h, that means the height of its left and right subtrees individually is at most h-1. So if the height of the left subtree is h-1, then the height of the right subtree could be any among {h-1, h-2} and vise versa.



Initially, we were given the problem of finding the output for height h. Now we have reduced the same to tell the output of height h-1 and h-2. Finally, we just need to figure out these counts, add them, and return.

Lets now look at the code below:

```
int balancedBTs(int h) {
        if(h <= 1) {                                    // Base case
                return 1;
        }

        int mod = (int) (pow(10, 9)) + 7;
        int x = balancedBTs(h - 1);                     // Answer for h-1
        int y = balancedBTs(h - 2);                     // Answer for h-2

/* Since, we need to find the total number of combinations, so will multiply the left
height's output and the right height's output as they are independent of each other
(Using law of multiplication in combinations)

    Possible Cases:
    • Both h-1          =       x*x
    • h-1 and h-2       =       x*y
    • h-2 and h-1       =       y*x

    Now, we will add all these together.
        */
```

```
        int temp1 = (int)(((long)(x)*x) % mod);
        int temp2 = (int)((2* (long)(x) * y) % mod);
        int ans = (temp1 + temp2) % mod;

        return ans;
}
```

Time Complexity: If we observe this function, then we can find it very similar to the pattern of the Fibonacci number. Hence, the time complexity is $O(2^h)$.

Now, try to reduce the time complexity of the code using memoization and DP by yourselves and for solution refer to the solution tab of the problem.

Dynamic Programming - 2

Let us now move to some advanced-level DP questions, which deal with 2D arrays.

Min cost path

Problem Statement: Given an integer matrix of size m*n, you need to find out the value of minimum cost to reach from the cell (0, 0) to (m-1, n-1). From a cell (i, j), you can move in three directions : (i+1, j), (i, j+1) and (i+1, j+1). The cost of a path is defined as the sum of values of each cell through which path passes.

For example: The given input is as follows-

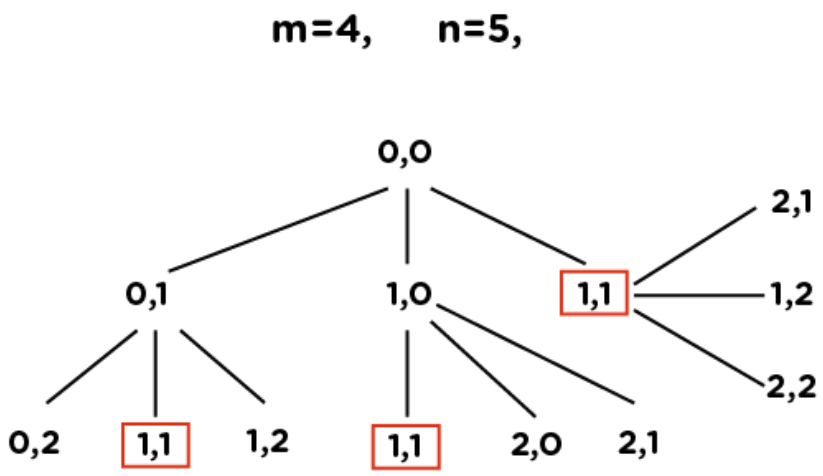
3 4
3 4 1 2
2 1 8 9
4 7 8 1
The path that should be followed is 3 -> 1 -> 8 -> 1. Hence the output is 13.

Approach: Thinking about the **recursive approach** to reach from the cell (0, 0) to (m-1, n-1), we need to decide for every cell about the direction to proceed out of three. We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.
Let's now look at the recursive code for this problem:

```
include <iostream>
using namespace std;
```

```
int minCostPath(int **input, int m, int n, int i, int j) {
        // Base case: reaching out to the destination cell
        if(i == m- 1 && j == n- 1) {
                return input[i][j];
        }

        if(i >= m || j >= n) {                  // Checking if the current row and column are
                return INT_MAX;                 // within the constraints or not, if not, then
        }           // returning +infinity, so that it will not be considered as the answer

        // Recursive calls
        int x = minCostPath(input, m, n, i, j+1);         // Towards right direction
        int y = minCostPath(input, m, n, i+1, j+1);       // Towards diagonally right-down
        int z = minCostPath(input, m, n, i+1, j);         // Towards the down direction

        // Small Calculation: figuring out the minimum value and then adding
        // current cells value to it
        int ans = min(x, min(y, z)) + input[i][j];
        return ans;
}

int minCostPath(int **input, int m, int n) {      // we will be using a helper function
        return minCostPath(input, m, n, 0, 0);  // as wee need to keep the track of
}                                               // current row and column

int main() {
        int m, n;
        cin >> m >> n;
        int **input = new int*[m];
        for(int i = 0; i < m; i++) {
                input[i] = new int[n];
                for(int j = 0; j < n; j++) {
                        cin >> input[i][j];
                }
        }
        cout << minCostPath(input, m, n) << endl;
}
```

Let's dry run the approach to see the code flow. Suppose, m = 4 and n = 5; then the recursive call flow looks something like below:



Here, we can clearly see that there are many repeated/overlapping recursive calls(for example: (1,1) is one of them), leading to exponential time complexity, i.e., $O(3^n)$. If we store the output for each recursive call after their first occurrence, we can easily avoid the repetition. It means that we can improve this using memoization.

Now, let's move on to the **Memoization approach**.

In memoization, we avoid repeated overlapping calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as we already discussed in our previous lectures that the storage used for the memoization is generally the same as the one that recursive calls use to their maximum.

Refer to the memoization code (along with the comments) below for better understanding:

```
int minCostPath_Mem(int **input, int m, int n, int i, int j, int **output) {
    if(i == m- 1 && j == n- 1) {            // Base case
        return input[i][j];
    }

    if(i >= m || j >= n) {
        return INT_MAX;
```

```
    }

    // Check if ans already exists
    if(output[i][j] != -1) {
        return output[i][j];    // as each cell stores its own ans
    }

    // Recursive calls
    int x = minCostPath_Mem(input, m, n, i, j+1, output);
    int y = minCostPath_Mem(input, m, n, i+1, j+1, output);
    int z = minCostPath_Mem(input, m, n, i+1, j, output);

    // Small Calculation
    int a = min(x, min(y, z)) + input[i][j];

    // Save the answer for future use
    output[i][j] = a;

    return a;
}

int minCostPath_Mem(int **input, int m, int n, int i, int j) {   // This function will be
    int **output = new int*[m];                                  // called from main()
    for(int i = 0; i < m; i++) {
        output[i] = new int[n];
        for(int j = 0; j < n; j++) {
            output[i][j] = -1;                  // Initialising the output array by -1. Here, -1
        }                                       // denotes that the value of the current cell is
    }                                           // unknown and could be replaced only after we
                                                // find the same
    return minCostPath_Mem(input, m, n, i, j, output);
}
```

Here, we can observe that as we move from the cell (0,0) to (m-1, n-1), in general, the i-th row varies from 0 to m-1, and the j-th column runs from 0 to n-1. Hence, the unique recursive calls will be a maximum of (m-1) * (n-1), which leads to the time complexity of O(m*n).

To get rid of the recursion, we will now proceed towards the **DP approach**.

The DP approach is simple. We just need to create a solution array (lets name that as **ans**), where ans[i][j] = minimum cost to reach from (i, j) to (m-1, n-1).

Now, initialize the last row and last column of the matrix with the sum of their values and the value, just after it. This is because, in the last row or column, we can reach there from their forward cell only (You can manually check it), except the cell (m-1, n-1), which is the value itself.

- ans[m-1][n-1] = input[m-1][n-1]
- ans[m-1][j] = ans[m-1][j+1] + input[m-1][j] (for 0 < j < n)
- ans[i][n-1] = ans[i+1][n-1] + input[i][m-1] (for 0 < i < m)

Next, we will simply fill the rest of our answer matrix by checking out the minimum among values from where we could reach them. For this, we will use the same formula as used in the recursive approach:

ans[i][j] = min(ans[i+1][j], min(ans[i+1][j+1], ans[i][j+1])) + input[i][j]

Finally, we will get our answer at the cell (0, 0), which we will return.

The code looks as follows:

```
int minCOst_DP(int **input, int m, int n) {
        int **ans = new int*[m];
        for(int i = 0; i < m; i++) {
                ans[i] = new int[n];
        }

        ans[m-1][n-1] = input[m-1][n-1];

        // Last row
        for(int j = n - 2; j >= 0; j--) {
                ans[m-1][j] = input[m-1][j] + ans[m-1][j+1];
        }

        // Last col
        for(int i = m-2; i >= 0; i--) {
                ans[i][n-1] = input[i][n-1] + ans[i+1][n-1];
        }

       // Calculation using formula
        for(int i = m-2; i >= 0; i--) {
                for(int j = n-2; j >= 0; j--) {
                     ans[i][j] = input[i][j] + min(ans[i][j+1], min(ans[i+1][j+1], ans[i+1][j]));
                }
        }
```

```
        return ans[0][0];     // Our Final answer as discussed above
}
```

Note: This is the bottom-up approach to solve the question using DP.

LCS (Longest Common Subsequence)

Problem Statement: The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

Note: Subsequence is a part of the string which can be made by omitting none or some of the characters from that string while maintaining the order of the characters.

If s1 and s2 are two given strings then z is the common subsequence of s1 and s2, if z is a subsequence of both of them.

Example 1:

s1 = “abcdef”

s2 = “xyczef”

Here, the longest common subsequence is “cef”; hence the answer is 3 (the length of LCS).

Example 2:

s1 = “ahkolp”

s2 = “ehyozp”

Here, the longest common subsequence is “hop”; hence the answer is 3.

Approach: Let’s first think of a brute-force approach using **recursion**. For LCS, we are required to first-of-all match the starting characters of both the strings. If they match, then simply we can break the problem as shown below:

s1 = “x | yzar”
s2 = “x | qwea”

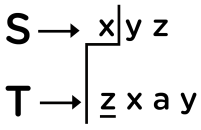
and the rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can’t be directly predicted by just looking at them, so we will be traversing over both of them one-by-one and check for the maximum value of LCS obtained among them to be considered for our answer.

For example:

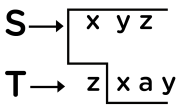
Suppose, string s1 = *“xyz”* and string s2 = *“zxay”*.

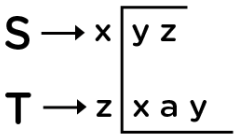
We can see that their first characters do not match so that we can call recursion over it in either of the following ways:

A =



B =





C=

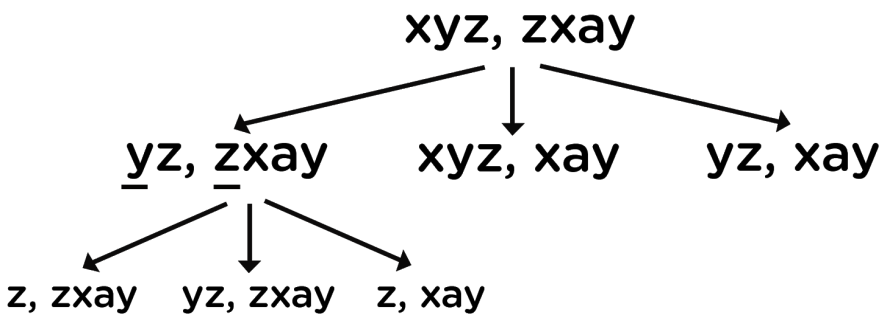
Finally, our answer will be:

LCS = max(A, max(B, C))

Check the code below and follow the comments for a better understanding.

```
int lcs(string s, string t) {
        // Base case
        if(s.size() == 0 || t.size() == 0) {
            return 0;
        }

        // Recursive calls
        if(s[0] == t[0]) {
            return 1 + lcs(s.substr(1), t.substr(1));
        }
        else {
            int a = lcs(s.substr(1), t);   // discarding the first character of string s
            int b = lcs(s, t.substr(1));   // discarding the first character of string t
            int c = lcs(s.substr(1), t.substr(1)); //discarding the first character of both
            return max(a, max(b, c));          // Small calculation
        }
}
```

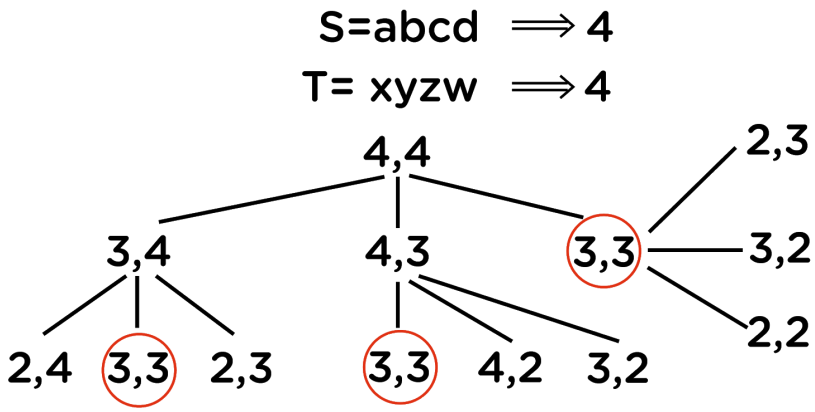
If we dry run this over the example: s = *"xyz"* and t = *"zxay"*, it will look something like below:



Here, as for each node, we will be making three recursive calls, so the time complexity will be exponential and is represented as $O(2^{m+n})$, where m and n are the lengths of both strings. This is because, if we carefully observe the above code, then we can skip the third recursive call as it will be covered by the two others.

Now, thinking over improving this time complexity...

Consider the diagram below, where we are representing the dry run in terms of its length taken at each recursive call:



As we can see there are multiple overlapping recursive calls, the solution can be optimized using **memoization** followed by DP. So, beginning with the memoization approach, as we want to match all the subsequences of the given two strings, we

have to figure out the number of unique recursive calls. For string s, we can make at most length(s) recursive calls, and similarly, for string t, we can make at most length(t) recursive calls, which are also dependent on each other's solution. Hence, our result can be directly stored in the form of a 2-dimensional array of size (length(s)+1) * (length(t) + 1) as for string s, we have 0 to length(s) possible combinations, and the same goes for string t.

So for every index 'i' in string *s* and 'j' in string *t*, we will choose one of the following two options:

1. If the character s[i] matches t[j], the length of the common subsequence would be one plus the length of the common subsequence till the i-1 and j-1 indexes in the two respective strings.
2. If the character s[i] does not match t[j], we will take the longest subsequence by either skipping i-th or j-th character from the respective strings.

Hence, the answer stored in the matrix will be the LCS of both strings when the length of string s will be 'i' and the length of string t will be 'j'.

Hence, we will get the final answer at the position matrix[length(s)][length(t)]. Moving to the code:

```
int lcs_mem(string s, string t, int **output) {
        int m = s.size();
        int n = t.size();

        // Base case
        if(s.size() == 0 || t.size() == 0) {
                return 0;
        }

        // Check if ans already exists
        if(output[m][n] != -1) {
                return output[m][n];
        }
```

```
        int ans;
        // Recursive calls
        if(s[0] == t[0]) {
                ans = 1 + lcs_mem(s.substr(1), t.substr(1), output);
        }
        else {
                int a = lcs_mem(s.substr(1), t, output);
                int b = lcs_mem(s, t.substr(1), output);
                int c = lcs_mem(s.substr(1), t.substr(1), output);
                ans = max(a, max(b, c));
        }

        // Save your calculation
        output[m][n] = ans;

        // Return ans
        return ans;
}

int lcs_mem(string s, string t) {
        int m = s.size();
        int n = t.size();
        int **output = new int*[m+1];
        for(int i = 0; i <= m; i++) {
                output[i] = new int[n+1];
                for(int j = 0; j <= n; j++) {
                        output[i][j] = -1;          // Intializing the 2D array to -1

                }
        }
        return lcs_mem(s, t, output);
}
```

Now, converting this approach into the **DP** code:

```
int lcs_DP(string s, string t) {
        int m = s.size();
        int n = t.size();
        // declaring a 2D array of size m*n
        int **output = new int*[m+1];
        for(int i = 0; i <= m; i++) {
                output[i] = new int[n+1];
        }
```

```
        // Fill 1st row
        for(int j = 0; j <= n; j++) {    // as if string t is empty, then the lcs(s, t) = 0
                output[0][j] = 0;
        }

        // Fill 1st col
        for(int i = 1; i <= m; i++) {     // as if string s is empty, then the lcs(s, t) = 0
                output[i][0] = 0;
        }

        for(int i = 1; i <= m; i++) {
                for(int j = 1; j <= n; j++) {
                        // Check if 1st char matches
                        if(s[m-i] == t[n-j]) {
                                output[i][j] = 1 + output[i-1][j-1];
                        }
                        else {
                                int a = output[i-1][j];
                                int b = output[i][j-1];
                                int c = output[i-1][j-1];
                                output[i][j] = max(a, max(b, c));
                        }
                }
        }
        return output[m][n];     // final answer
}
```

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to O(m*n) where m and n are the lengths of the given strings.

Edit Distance

Problem statement: Given two strings s and t of lengths m and n respectively, find the Edit Distance between the strings. Edit Distance of two strings is the minimum number of steps required to make one string equal to another. To do so, you can perform the following three operations only :

- Delete a character
- Replace a character with another one
- Insert a character

Example 1:

s1 = “but”
s2 = “bat”
Answer: 1
Explanation: We just need to replace ‘a’ with ‘u’ to transform s2 to s1.

Example 2:

s1 = “cbda”
s2 = “abdca”
Answer: 2
Explanation: We just need to replace the first ‘a’ with ‘c’ and delete the second ‘c’.

Example 3:

s1 = “ppsspqrt”
s2 = “passpot”
Answer: 3
Explanation: We just need to replace first ‘a’ with ‘p’, ‘o’ with ‘q’, and insert ‘r’.

Approach: Let’s think about this problem using **recursion** first. We need to apply each of the three operations on each character of s2 to make it similar to s1 and then find the minimum among them.

Let’s assume index1 and index2 point to the current indexes of s1 and s2 respectively, so we have two options at every step:

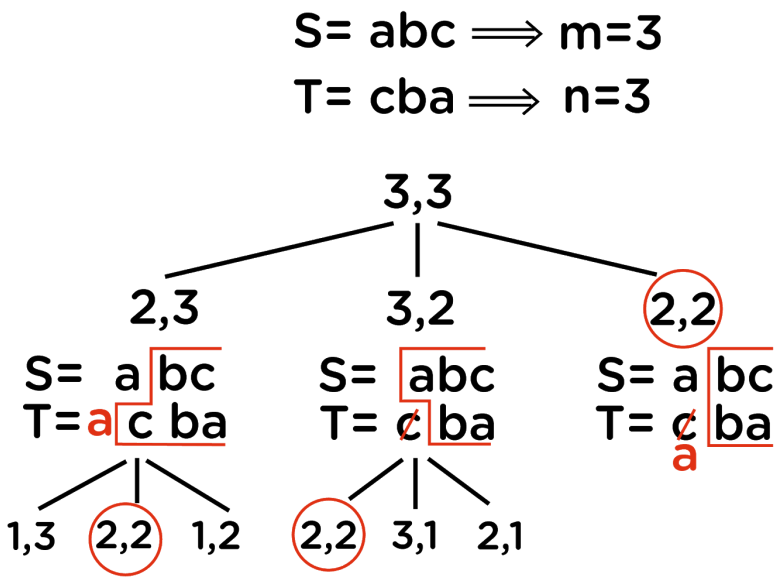
1. If the strings have the same character, we can recursively match for the remaining lengths of the strings.
2. If the strings do not match, we start three new recursive calls representing the three edit operations, as mentioned in the problem statement. Consider the minimum count of operations among the three recursive calls.

Kindly look at the code to get a better understanding of the same.

```
int editDistance(string s, string t) {
        // Base case
```

```
        if(s.size() == 0 || t.size() == 0) {
                return max(s.size(), t.size());
        }

        if(s[0] == t[0]) {                  // If the first character matches
                return editDistance(s.substr(1), t.substr(1));
        }
        else {
                int x = editDistance(s.substr(1), t) + 1;                  // Insert
                int y = editDistance(s, t.substr(1)) + 1;                  // Delete
                int z = editDistance(s.substr(1), t.substr(1)) + 1;        // Replace
                return min(x, min(y, z));
        }
}
```

Let's dry run the code:



From here, it is clear that the time complexity is again exponential, which is $O(3^{m+n})$, where m and n are the lengths of the given strings.

Also, we can see the overlapping/repeated recursive calls(for example: (2,2) is one of them), which means this problem can be further solved using **memoization**.

This problem is somehow similar to the LCS problem. We will be using a similar approach to solve this problem too. The answer to each recursive call will be stored in a 2-Dimensional array of size (m+1)*(n+1), and the final solution will be obtained at index (m,n) as each cell will be storing the answer for the given m length of s1 and n length of s2.

Refer to the code below:

```
int editDistance_mem(string s, string t, int **output) {
        int m = s.size();
        int n = t.size();
      // If one of them has reached the end, insert all remaining characters to other
        if(s.size() == 0 || t.size() == 0) {
                return max(s.size(), t.size());
        }

        // Check if ans already exists
        if(output[m][n] != -1) {
                return output[m][n];
        }

        int ans;
        if(s[0] == t[0]) {                      // First character matches
                ans = editDistance_mem(s.substr(1), t.substr(1), output);
        }
        else {
                int x = editDistance(s.substr(1), t, output) + 1;               // Insert
                int y = editDistance(s, t.substr(1), output) + 1;               // Delete
                int z = editDistance(s.substr(1), t.substr(1), output) + 1;     // Replace
                ans = min(x, min(y, z));
        }

        // Save the ans
        output[m][n] = ans;

        // Return the ans
        return ans;
}

int editDistance_mem(string s, string t) {
        int m = s.size();
        int n = t.size();
        int **ans = new int*[m+1];
        for(int i = 0; i <= m; i++) {
```

```
                ans[i] = new int[n+1];
                for(int j = 0; j <= n; j++) {
                        ans[i][j] = -1;
                }
        }
        return editDistance_mem(s, t, ans);
}
```

Time Complexity: As there are (m+1)*(n+1) number of unique calls, hence the time complexity becomes O(m*n), which is better than the recursive approach.

Let's move on to the DP approach...

We have already discussed the basic requirements like output array size, final output's position, and the value stored at each position of the output array in the memoization approach. We have already figured out that this problem is similar to the LCS question. So, forwarding directly towards the code:

```
int editDistance_DP(string s, string t) {
        int m = s.size();
        int n = t.size();

        int **output = new int*[m+1];
        for(int i = 0; i <= m; i++) {
                output[i] = new int[n+1];
        }

        // Fill 1st row
        for(int j = 0; j <= n; j++) {
                output[0][j] = j;
        }

        // Fill 1st col
        for(int i = 1; i <= m; i++) {
                output[i][0] = i;
        }

        for(int i = 1; i <= m; i++) {
                for(int j = 1; j <= n; j++) {
                        if(s[m-i] == t[n-j]) {          // checking the first characters
                                output[i][j] = output[i-1][j-1];
```

```
                    }
                    else {
                            int a = output[i-1][j];
                            int b = output[i][j-1];
                            int c = output[i-1][j-1];
                            output[i][j] = min(a, min(b, c)) + 1;
                    }
            }
        }
        return output[m][n];    // final answer
}
```

Time complexity: It is the same as the memoization approach, i.e., O(m*n).

Knapsack

Problem Statement: Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

For example:

Items: {Apple, Orange, Banana, Melon}
Weights: {2, 3, 1, 4}
Values: {4, 5, 3, 7}
Knapsack capacity: 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value
Apple + Banana (total weight 3) => 7 value
Orange + Banana (total weight 4) => 8 value
Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

Approach: First-of-all, let's discuss the brute-force-approach, i.e., the **recursive approach**. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.

Let's look at the recursive code for the same:

```
int knapsack(int *weight, int *values, int n, int maxWeight) {
        // Base case : if the size of array is 0 or we are not able to add any more weight
        // to the knapsack
        if(n == 0 || maxWeight == 0) {
                return 0;
        }

        // If the particular weight's value extends the limit of knapsack's remaining
        // capacity, then we have to simpy skip it
        if(weight[0] > maxWeight) {
                return knapsack(weight + 1, values + 1, n - 1, maxWeight);
        }

        // Recursive calls
        //1. Considering the weight
        int x = knapsack(weight + 1, values + 1, n - 1, maxWeight - weight[0]) + values[0];
        // 2. Skipping the weight and moving forward
        int y = knapsack(weight + 1, values + 1, n - 1, maxWeight);

        // finally returning the maximum answer among the two
        return max(x, y);
}
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some example and by dry running it.

Practice problems:

The link provided below contains 26 problems based on Dynamic programming and numbered as A to Z, A being the easiest, and Z being the toughest.

https://atcoder.jp/contests/dp/tasks

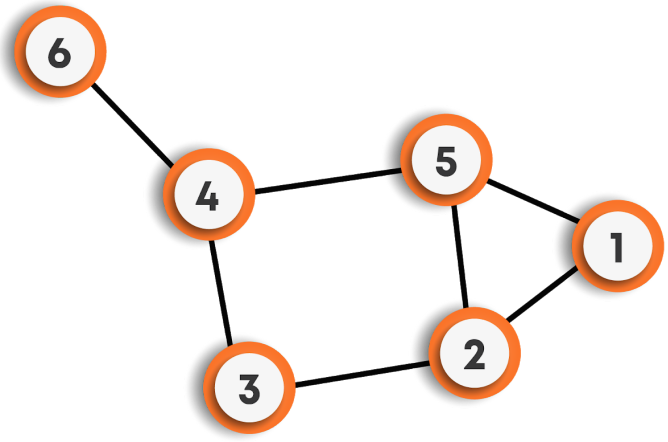
Graphs 1

Graphs Introduction

A **graph** is a pair $G = (V, E)$, where V is a set whose elements are called *vertices*, and E is a set of two-sets of vertices, whose elements are called *edges*.

The vertices x and y of an edge $\{x, y\}$ are called the *endpoints* of the edge. The edge is said to *join* x and y and to be *incident* on x and y. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure for better understanding.



Relationship between trees and graphs:

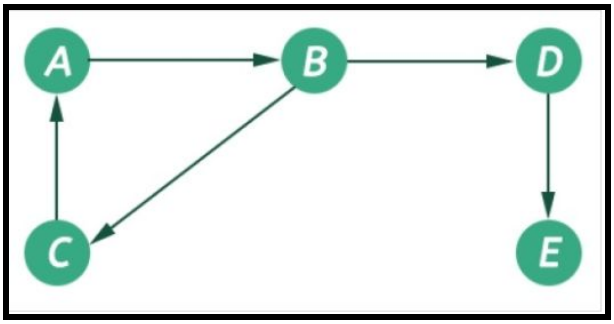
- A tree is a special type of graph in which we can reach any node to any other node using some path, unlike the graphs where this condition may or may not hold.
- A tree does not have any cycles in it.

Graphs Terminology

- Nodes are named as **vertices,** and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be (n-1), where n is the number of nodes.
- In a complete graph (where each node is connected to every other node by a direct edge), there are ${}^{n}\mathbf{C}_{2}$ number of edges means (n * (n-1)) / 2 edges, where n is the number of nodes. It is the maximum number of edges that a graph can have. Hence, if an algorithm works on the terms of edges, let's say O(E), where E is the number of edges, then in the worst case, the algorithm will take $O(n^2)$ time, where n is the number of nodes.

Graphs Implementation

Suppose the graph is as follows:

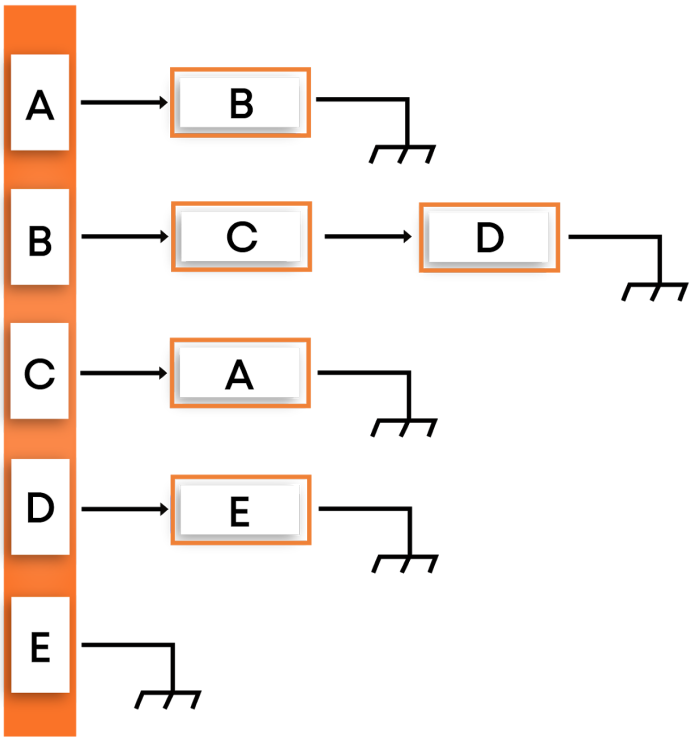


There are the following ways to implement a graph:

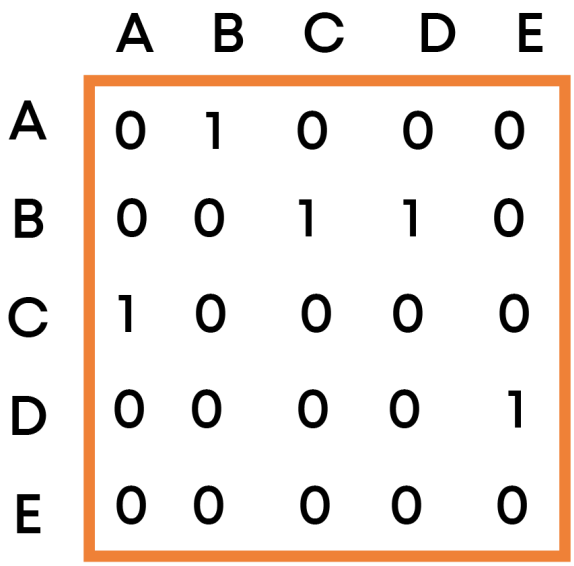
1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred as to check for a particular edge connecting two nodes; we have to traverse the complete array leading to $O(n^2)$ time complexity in the worst case. Pictorial representation for the above graph using edge list is given below:

2. **Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check

in its list if the target node is present or not. This will take O(n) work to figure out a particular edge. Visually, it looks as follows:



3. **Adjacency matrix:** Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, adjacency matrix looks as follows:



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

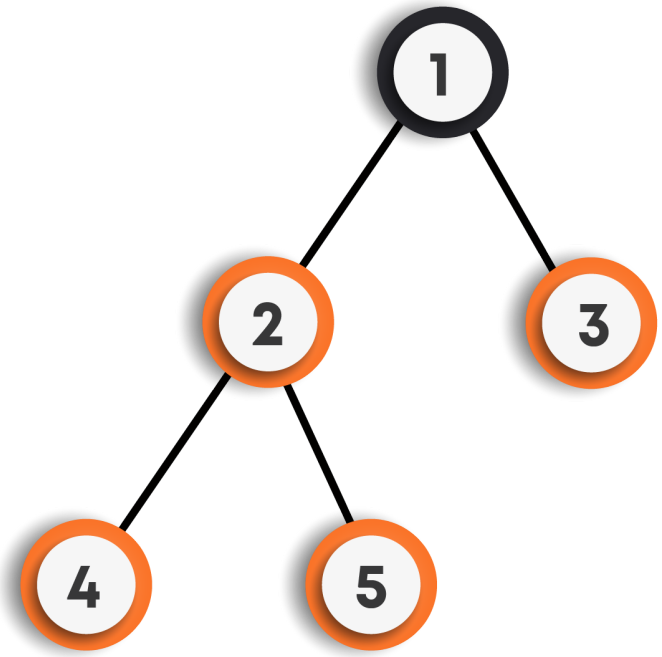
DFS - Adjacency matrix

Here, if we have n vertices(labelled: 0 to n-1). Then we will be asking the user for the number of edges. We will run the loop from 0 to the number of edges, and at each iteration, we will take input for the two connected nodes and correspondingly update the adjacency matrix. Let's look at the code for better understanding.

```
#include<iostream>
using namespace std;

void print(int** edges, int n, int sv, bool* visited){
  cout << sv << endl;
  visited[sv] = true;                  // marked the starting vertex true
  for(int i=0; i<n; i++){              // Running the loop over all n nodes and checking if
                                       // there is an edge between sv and i

    if(i==sv){
      continue;
    }
    if(edges[sv][i]==1){                        // As the edge is found, we then checked if the
```

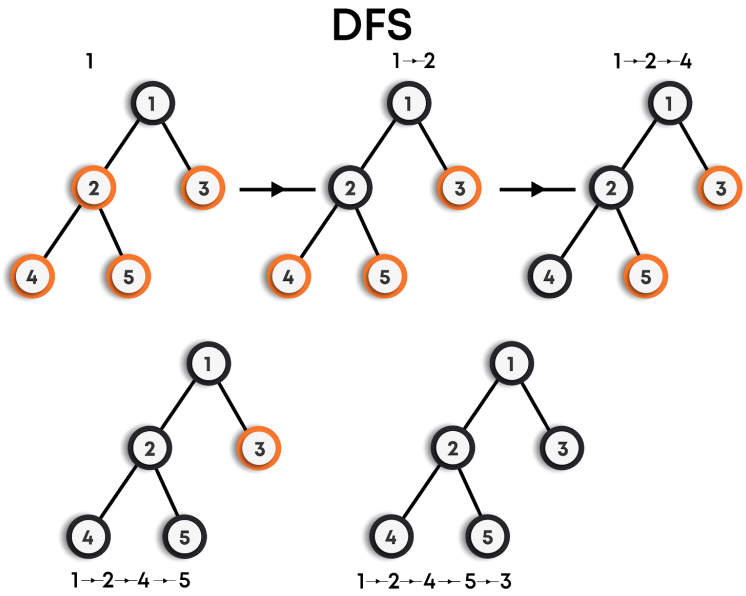
```
                                        // node i was visited or not
    if(visited[i]){
      continue;
    }
    print(edges, n, i, visited);        // Otherwise, recursively called over node i
                                        // taking it as starting vertex
  }
 }
}

int main(){
 int n;                         // Number of nodes
 int e;                         // Number of edges
 cin >> n >> e;

 int** edges = new int*[n];         // adjacency matrix of size n*n
 for(int i=0; i<n; i++){
  edges[i]=new int[n];
  for(int j=0; j<n; j++){
   edges[i][j]=0;                   // 0 indicates that there is no edge between i and j
  }
 }

 for(int i=0; i<e; i++){
  int f,s;
  cin >> f >> s;                    // Nodes having edges between them
  edges[f][s]=1;                    // marking f to s as 1
  edges[s][f]=1;                    // also, marking s to f as 1
 }

 bool* visited = new bool[n];      // this is used to keep the track of nodes if we have
                                   // visited them or not.
 for(int i=0; i<n; i++){
  visited[i]=false;          // Marking all the nodes as false which means not visited
 }

 print(edges, n, 0, visited);               // starting vertex is taken as 0
 // Delete all the memory: Do it yourselves
 return 0;
}
```

Let's take an example graph:



On dry running the above code, the output will be 1 2 4 5 3.

Here, we are starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes. This method of graph traversal is known as the **depth-first search (DFS)**. As the name suggests, this algorithm goes into the depth first and then recursively does the same in other directions. Follow the figure below, for step-by-step traversal using DFS.

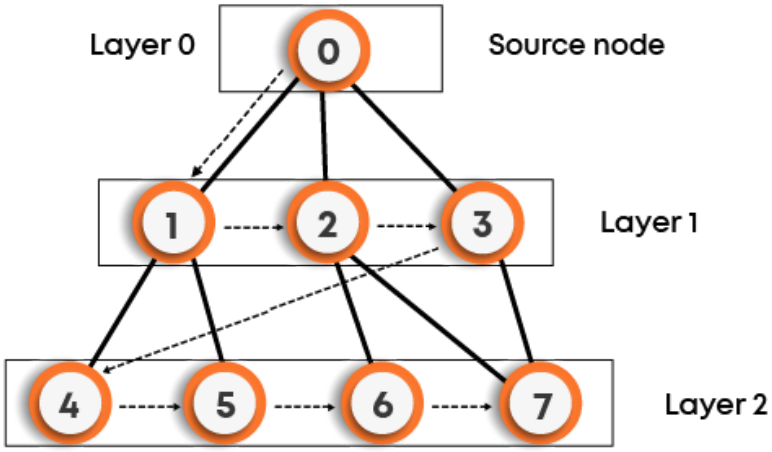


BFS Traversal

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

As the name suggests:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



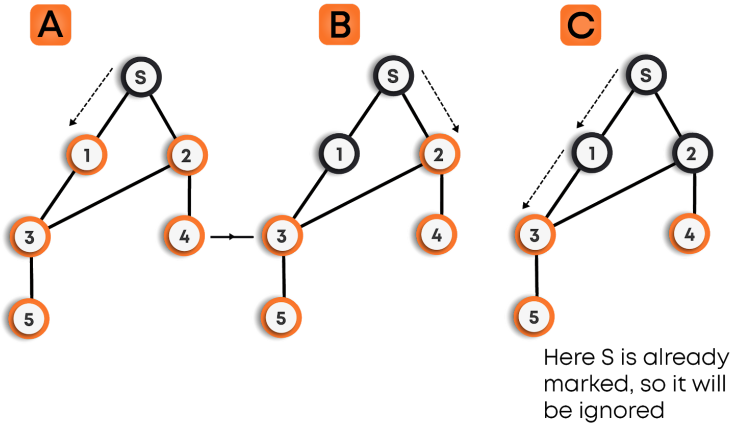
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

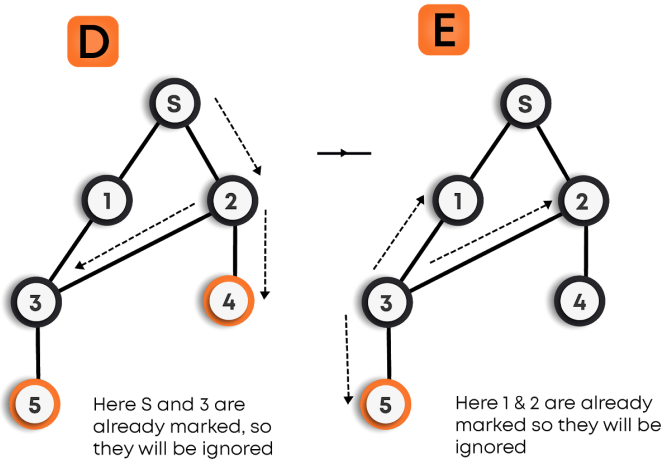
Let's look at the code below:

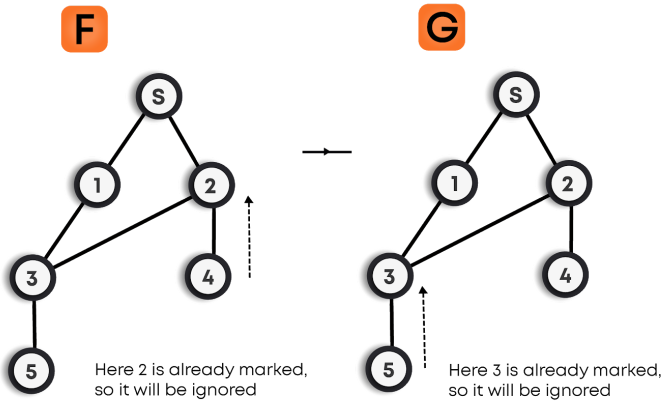
```
#include <queue>
#include <iostream>
#include <unordered_map>

void printBFS(int **edges, int n, int sv, bool *visited) {
        queue<int> pendingVertices;                    // queue
        pendingVertices.push(sv);                      // starting vertex directly pushed
        visited[sv] = true;
        while (!pendingVertices.empty()) {       // until the size of queue is not 0
                int currentVertex = pendingVertices.top();   // stored the top of queue
                pendingVertices.pop(); // deleted that top element
                cout << pendingVertices << " ";
                for (int i = 0; i < n; i++) {    // now checked for its vertices
                        if (i == currentVertex) {
                                continue;
                        }
                        if (edges[currentVertex][i] == 1 && !visited[i]) {
                                pendingVertices.push(i);// if found, then inserted into
                                                        // queue
```

```
                                visited[i] = true;
                        }
                }
        }
}

void BFS(int **edges, int n) {
        bool *visited = new bool[n];        // visited array to keep the track of nodes
        for (int i = 0; i < n; i++) {
                visited[i] = false;
        }

        printBFS(edges, n, 0, visited);     // starting vertex  = 0
        delete [ ] visited;                 // deleted the visited array
}

int main() {
        int n;                              // Number of nodes
        int e;                              // Number of edges
        cin >> n >> e;
        int **edges = new int*[n];          // Adjacency matrix
        for (int i = 0; i < n; i++) {
                edges[i] = new int[n];
                for (int j = 0; j < n; j++) {
                        edges[i][j] = 0;
                }
        }

        for (int i = 0; i < e; i++) {
                int f, s;
                cin >> f >> s;
                edges[f][s] = 1;
                edges[s][f] = 1;
        }

        BFS(edges, n);
        // delete the memory
        for (int i = 0; i < n; i++) {
                delete [ ] edges[i];
        }
        delete [ ] edges;
        return 0;
}
```

Consider the dry run over the example graph below for a better understanding of the same:


A
B
C
S
1
2
3
4
5
Here S is already marked, so it will be ignored


D
E
S
1
2
3
4
5
Here S and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored


F
G
S
1
2
3
4
5
Here 2 is already marked, so it will be ignored
Here 3 is already marked, so it will be ignored

BFS & DFS for disconnected graph

Till now, we have assumed that the graph is connected. For the disconnected graph, there will be a minor change in the above codes. Just before calling out the print functions, we will run a loop over each node and check if that node is visited or not. If not visited, then we will call a print function over that node, considering it as the starting vertex. In this way, we will be able to cover up all the nodes of the graph.

Consider the same for the BFS function. Just replace this function in the above code to make it work for the disconnected graph too.

```
void BFS(int **edges, int n) {
        bool *visited = new bool[n];        // visited array to keep the track of nodes
        for (int i = 0; i < n; i++) {
                visited[i] = false;
        }

        for (int i = 0; i < n; i++) {              // run a loop over each node
                If (!visited[i]) {                 // if a node is not visited, then called print()
                        printBFS(edges, n, i, visited);  // on it taking it as starting vertex
                }
        }
        delete [ ] visited;                        // deleted the visited array
}
```

Has Path

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Approach: This can be simply solved by considering the vertex v1 as starting vertex and then run either BFS or DFS as per your choice, and while traversing if we reach the vertex v2, then we will simply return true, otherwise return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Get Path - DFS

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.
Find the path using DFS and print the first path that you encountered irrespective of the length of the path.
V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1.
E is the number of edges present in graph G.
Print the path in reverse order. That is, print v2 first, then intermediate vertices and v1 at last.
Example: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be:

```
3 0 1
```

Explanation: Here, v1 = 1 and v2 = 3. The connected vertex pairs are (0, 1), (0, 3), (1, 2) and (2, 3). So, according to the question, we have to print the path from vertex v1 to v2 in reverse order using DFS only; hence the path comes out to be {3, 0, 1}.

Approach: We have to solve this problem by using DFS. Suppose, if the start and end vertex are the same, then we simply need to put the start in the solution array and return the solution array. If this is not the case, then from the start vertex, we will call DFS on the direct connections of the same. If none of the paths leads to the end vertex, then we do not need to push the start vertex as it is neither directly nor indirectly connected to the end vertex, hence we will simply return NULL. In case any of the neighbors return a non-null entry, it means that we have a path from that neighbor to the end vertex, hence we can now insert the start vertex into the solution array.
Try to code it yourself, and for the answer, refer to the solution tab of the same.

Get Path - BFS

Approach: It is the same problem as the above, just we have to code the same using BFS.

Approach: Using BFS will provide us the shortest path between the two vertices. We will use the queue over here and do the same until the end vertex gets inserted into the queue. Here, the problem is how to figure out the node, which led us to the end vertex. To overcome this, we will be using a map. In the map, we will store the resultant node as the index, and its key will be the node that led it into the queue. For example: If the graph was such that 0 was connected to 1 and 0 was connected to 2, and currently, we are on node 0 such that node 1 and node 2 are not visited. So our map will look like as follows:

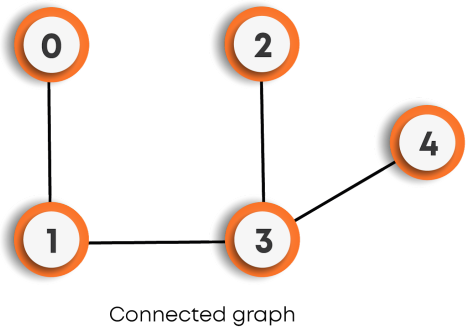
1	0
2	0

This way, as soon as we reach the end vertex, we can figure out the nodes by running the loop until we reach the start vertex as the key value of any node.

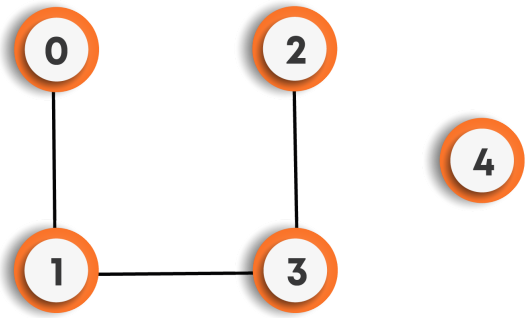
Try to code it yourselves, and for the solution, refer to the specific tab of the same.

Is connected?

Problem statement: Given an undirected graph G(V, E), check if the graph G is a connected graph or not. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.



Connected graph



Disconnected graph

Example 1: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be: **true**

Explanation: As the graph is connected, so according to the question, the answer will be true.

Example 2: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be: **false**

Explanation: The graph is not connected, even though vertices 0,1 and 3 are connected to each other, but there isn't any path from vertices 0,1,3 to vertex 2. Hence, according to the question, the answer will be false.

Approach: This is very start-forward. Take any vertex as the starting vertex as traverse the graph using either DFS or BFS. In the end, check if all the vertices are visited or not. If not, it means that the node was not connected to the starting vertex, which means it is a disconnected graph. Otherwise, it is a connected graph. Try to code it yourselves, and for the code, refer to the solution tab of the same.

Return all connected components

Problem statement: Given an undirected graph G(V, E), find and print all the connected components of the given graph G. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

You need to take input in the main and create a function that should return all the connected components. And then print them in the main, not inside a function.

Print different components in a new line. And each component should be printed in increasing order (separated by space). The order of different components doesn't matter.

Example: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be:

```
0 1 3
2
```

Explanation: As we can see that {0, 1, 3} is one connected component, and {2} is the other one. So, according to the question, we just have to print the same.

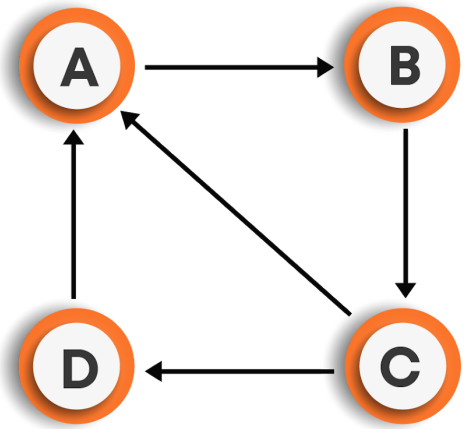
Approach: For this problem, start from vertex 0 and traverse until vertex n-1. If the vertex is not visited, then run DFS/BFS on it and keep track of all the connected vertices through that node. This way, we will get all the distinct connected components, and we can print them at last.

This problem is left for you to solve. For the code, refer to the solution tab of the same.

Weighted and directed graphs

There are two more variations of the graphs:

- **Directed graphs:** These are generally required when we have one-way routes. Suppose you can go from node A to node B, but you cannot go from node B to node A. Another example could be of social media(like Twitter) if you are following someone, it does not mean that they are following you too.



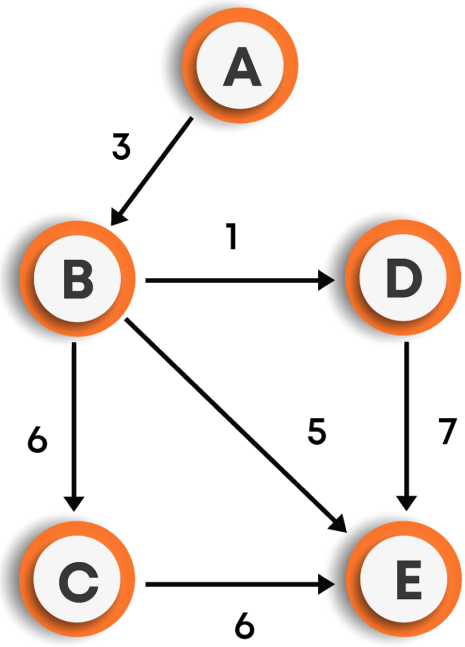
To implement these, there is a small change in the implementation of indirect graphs. In indirect graphs, if there was an edge between node i and j, then we did:

```
edges[i][j] = 1;
edges[j][i] = 1;
```

But, in the case of a directed graph, we will just do the following:

```
edges[i][j] = 1;
```

- **Weighted graphs:** These generally mean that all the edges are not equal, means somehow, each edge has some weight assigned to it. This weight can be the length of the road connecting the cities or many more.



To implement this, in the edges matrix, we will assign a weight to connected nodes instead of putting it 1 at that position. For example: If node i and j are connected, and the weight of the edge connecting them is 5, then edges[i][j] = 5.

Practice problems:

- https://www.codechef.com/problems/CHEFDAG
- https://www.spoj.com/problems/WORDS1/
- https://www.hackerrank.com/challenges/the-quickest-way-up/problem

Graphs 2

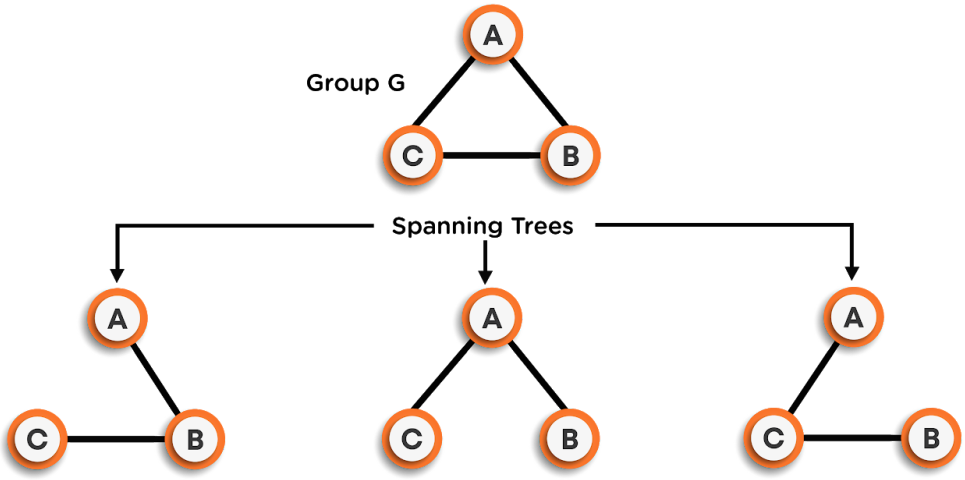
MST & Kruskal's introduction

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



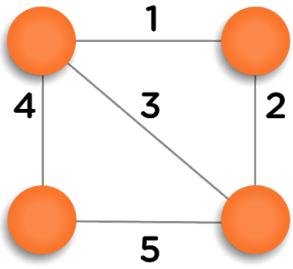
If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

Properties of spanning trees:

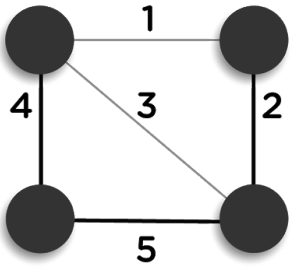
- A connected and undirected graph can have more than one spanning tree.
- Spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

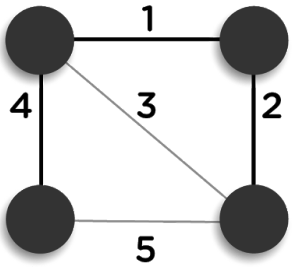


Undirected graph



Spanning tree

Cost=11(=4+5+2)



Minimum Spanning tree

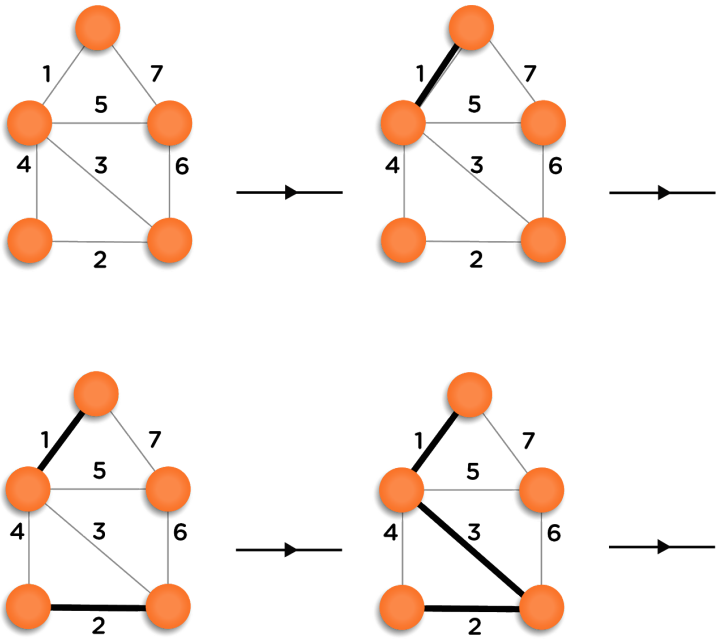
Cost=7(=4+1+2)

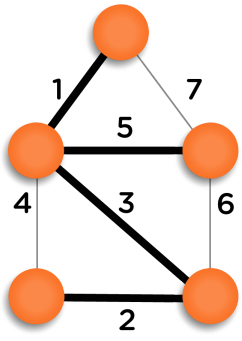
Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the

same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both the nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if anyone of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is O(E+V), where E is the number of edges in the graph and, V is the number of vertices. So, for (n-1) edges, this function will run (n-1) times, leading to bad time complexity, as in the worst case, $E = V^2$.

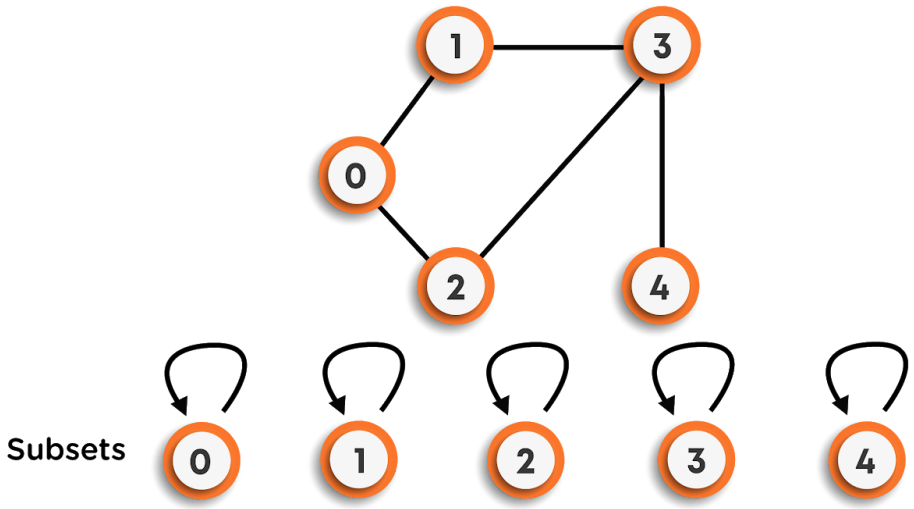
Now, moving on to a better approach for cycle detection in the graph.

Union-find algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

Following the steps of the algorithm:

- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.
- We will maintain a parent array specifying the parent vertex of each of the vertex of the graph. Initially, as each vertex belongs to the different disjoint set (connected component), hence each vertex will be its own parent.

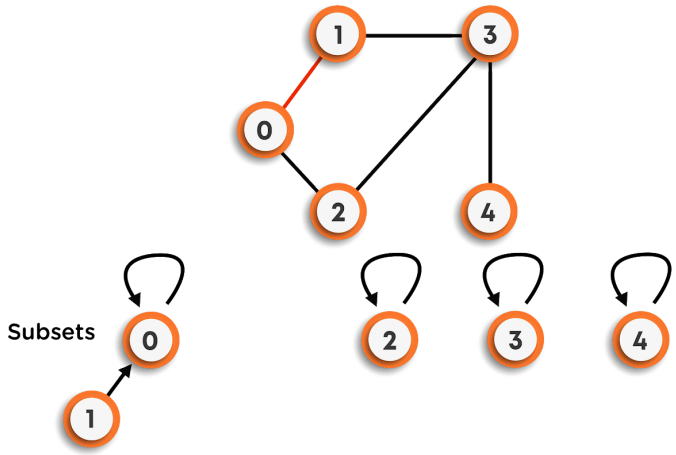


Initially all parent pointers are pointing to self
means only one element in each subset

- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge. Otherwise, we can add that

edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

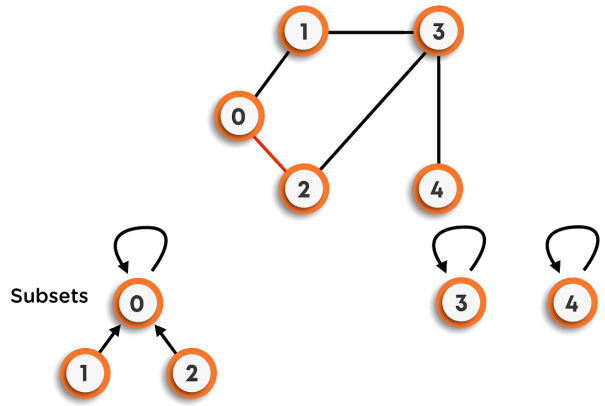
Look at the following example, for better understanding:



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

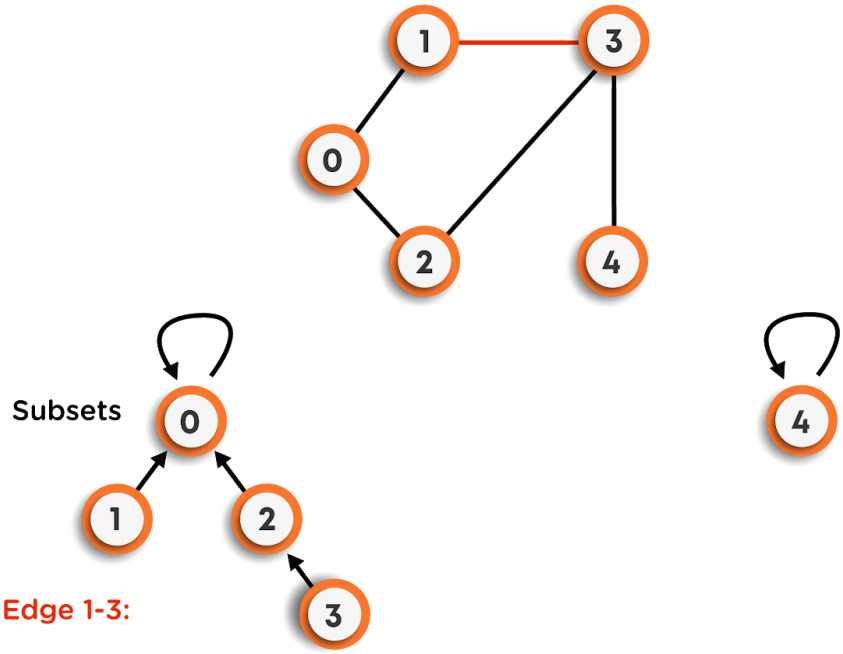
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.



Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

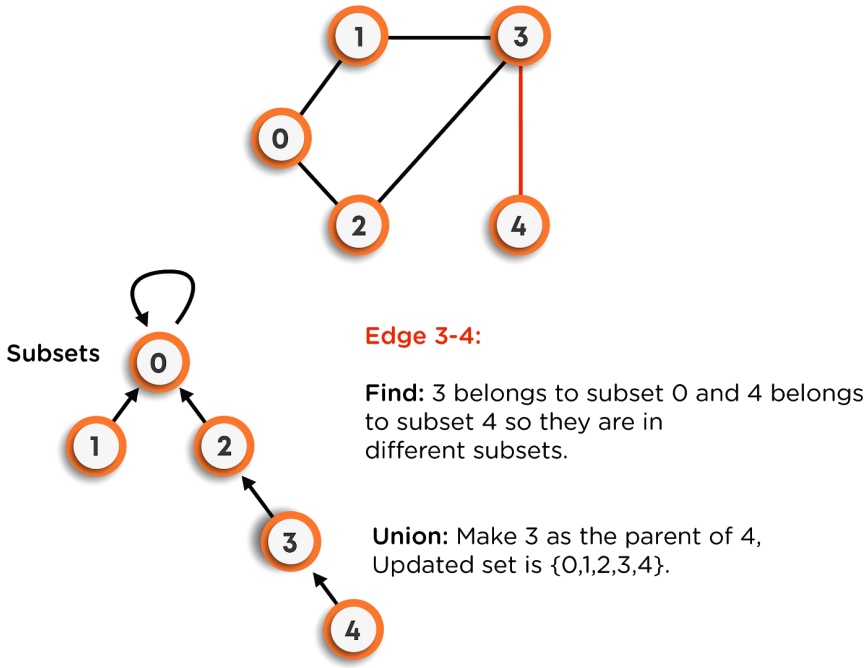
Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.



Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.



Note: While finding the parent of the vertex, we will be finding the topmost parent(Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

```
#include <iostream>
#include <algorithm>
using namespace std;

class Edge {                                    // Class that store values for each vertex
        public:
                int source;
                int dest;
                int weight;
};

bool compare(Edge e1, Edge e2) {                // Comparator function used to sort edges
        return e1.weight < e2.weight;           // Edges will sorted in order of their weights
}

int findParent(int v, int *parent) {            // Function to find the parent of a vertex
        if (parent[v] == v) {                   // Base case, when a vertex is parent of itself
                return v;
        }
        // Recursively called to find the topmost parent of the vertex.
        return findParent(parent[v], parent);
}

void kruskals(Edge *input, int n, int E) {
        sort(input, input + E, compare);    // In-built sort function: Sorts the edges in
                                            // increasing order of their weights

        Edge *output = new Edge[n-1];       // Array to store final edges of MST
        int *parent = new int[n];           // Parent array initialized with their indexes

        for (int i = 0; i < n; i++) {
                parent[i] = i;
        }

        int count = 0;      //  To maintain the count of number of edges in the MST
```

```
        int i = 0;              // Index to traverse over the input array
        while (count != n - 1) {    // As the MST contains n-1 edges.
                Edge currentEdge = input[i];
                // Figuring out the parent of each edge's vertices
                int sourceParent = findParent(currentEdge.source, parent);
                int destParent = findParent(currentEdge.dest, parent);
                // If their parents are not equal, then we added that edge to output
                if(sourceParent != destParent) {
                        output[count] = currentEdge;
                        count++;                                // Increased the count
                        parent[sourceParent] = destParent;// Updated the parent array
                }
                i++;
        }
        // Finally, printing the MST obtained.
        for (int i = 0; i < n-1; i++) {
                if(output[i].source < output[i].dest) {
                        cout << output[i].source << " " << output[i].dest << " " <<
output[i].weight << endl;
                }
                else {
                        cout<< output[i].dest << " " <<output[i].source << " " <<
output[i].weight << endl;
                }

        }
}

int main() {
        int n, E;
        cin >> n;
        cin >> E;

        Edge *input = new Edge[E];

        for (int i = 0; i < E; i++) {
                int s, d, w;
                cin >> s >> d >> w;
                input[i].source = s;
                input[i].dest = d;
                input[i].weight = w;
        }

        kruskals(input, n, E);
        return 0;
}
```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of size E.
- Sort the input array on the basis of edge-weight. This step has the time complexity of O(E log(E)).
- Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.

Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved. We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path Compression**. You need to explore this on yourselves. The basic idea in these algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

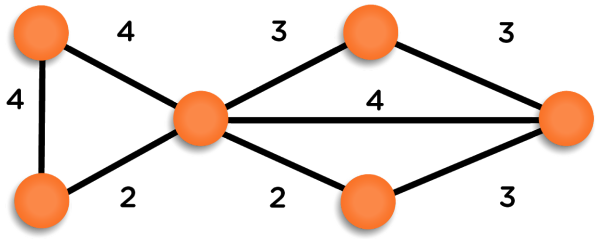
Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree

contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

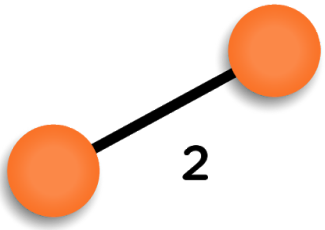


Step: 1

Start with a weighted graph

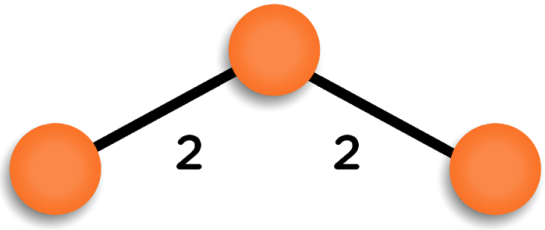
Step: 2

Choose a vertex



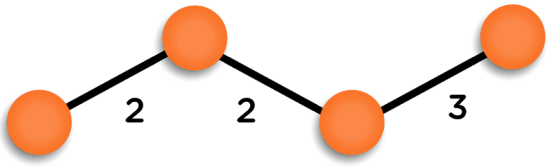
Step: 3

Choose the shortest edge from this vertex add it



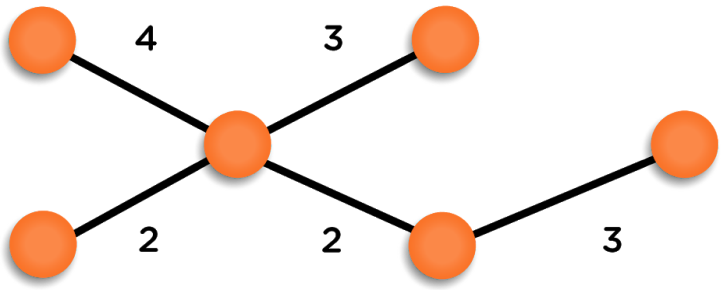
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.
- We will mark the vertex 0 as visited and rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

```
#include <iostream>
#include <climits>
using namespace std;

int findMinVertex(int *weights, bool *visited, int n) {

        int minVertex = -1;         // Initialized to -1 means there is no vertex till now
        for (int  i = 0; i < n; i++) {
        // Conditions : the vertex must be unvisited and either minVertex value is -1
        //  or if minVertex has some vertex to it, then weight of currentvertex
        // should be less than the weight of the minVertex.
                if (!visited[i] && (minVertex == -1 || weights[i] < weights[minVertex])) {
                        minVertex = i;
                }
        }
        return minVertex;
}

void prims(int **edges, int n) {

        int *parent = new int[n];
        int *weights = new int[n];
        bool *visited = new bool[n];
        // Initially, the visited array is assigned to false and weights array to infinity.
        for(int i = 0; i < n; i++) {
                visited[i] = false;
                weights[i] = INT_MAX;
        }
        // Values assigned to vertex 0.(the selected starting vertex to begin with)
        parent[0] = -1;
        weights[0] = 0;

        for (int i = 0; i < n-1; i++) {
                // Find min vertex
                int minVertex = findMinVertex(weights, visited, n);
                visited[minVertex] = true;
                // Explore unvisited neighbors
                for (int j = 0; j < n; j++) {
                        if(edges[minVertex][j] != 0 && !visited[j]) {
                                if(edges[minVertex][j] < weights[j]) {
                            // updating weight array and parent array
                                        weights[j] = edges[minVertex][j];
                                        parent[j] = minVertex;
                                }
                        }
                }
        }
```

```
        // Final MST printed
        for (int i = 0; i < n; i++) {
                if (parent[i] < i) {
                        cout << parent[i] << " " << i << " " <<weights[i] << endl;
                }
                else {
                        cout << i << " " << parent[i] << " " << weights[i] << endl;
                }
        }
}

int main() {

        int n;
        int e;
        cin >> n >> e;
        int **edges = new int*[n];   // Adjacency matrix used to store the graph
        for (int i = 0; i < n; i++) {
                edges[i] = new int[n];
                for (int j = 0; j < n; j++) {
                        edges[i][j] = 0;    // Initially all pairs are assigned 0 weight which
                                            // means that there is no edge between them
                }
        }

        for (int i = 0; i < e; i++) {
                int f, s, weight;
                cin >> f >> s >> weight;
                edges[f][s] = weight;
                edges[s][f] = weight;
        }

        prims(edges, n);

        for(int i = 0; i < n; i++) {
                delete [] edges[i];
        }
        delete [] edges;
        return 0;
}
```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit each and every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.
- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

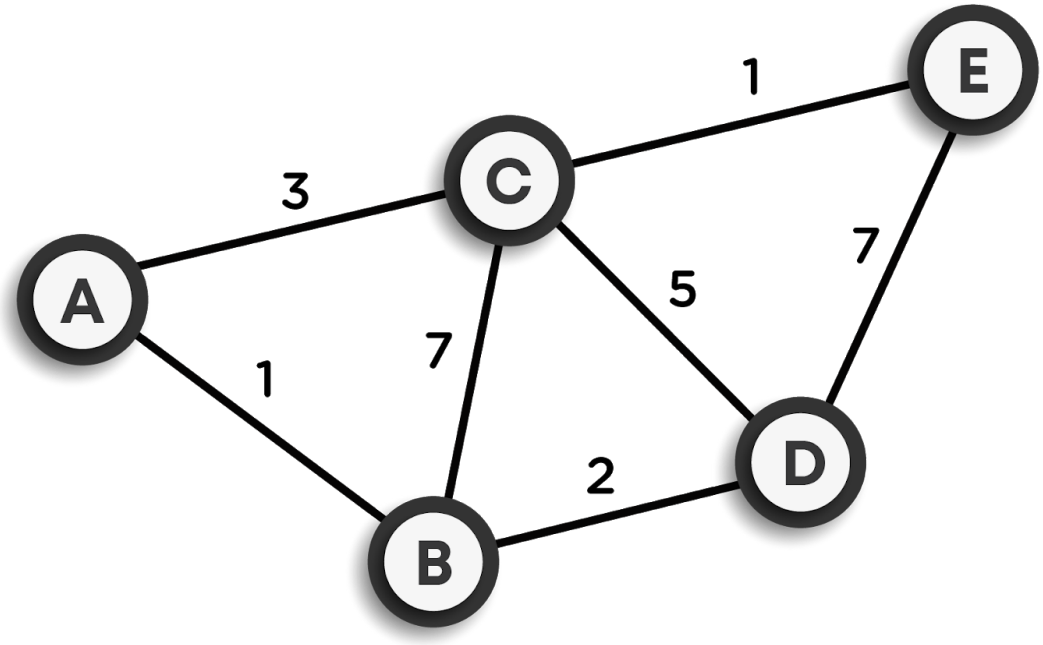
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

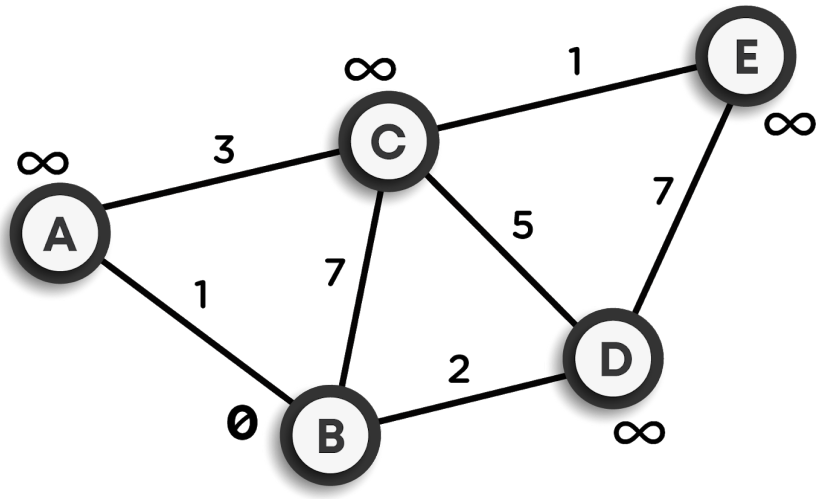
Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

Let's consider the algorithm with an example:

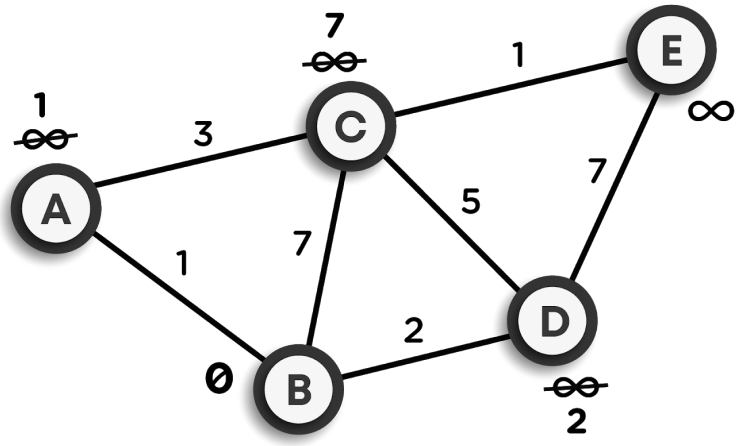
1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.



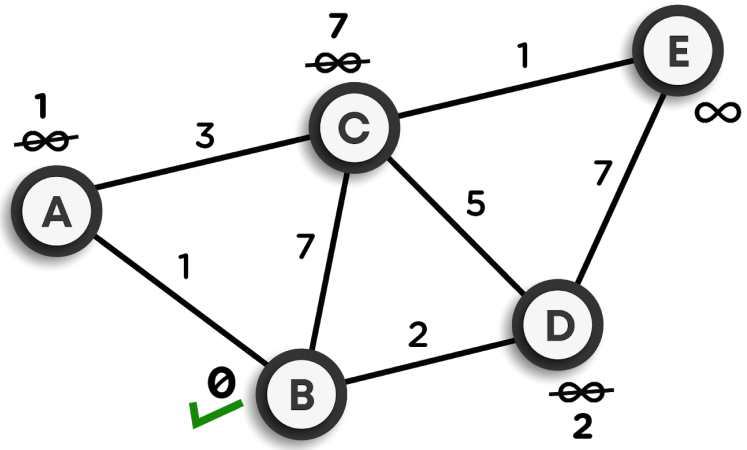
2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.



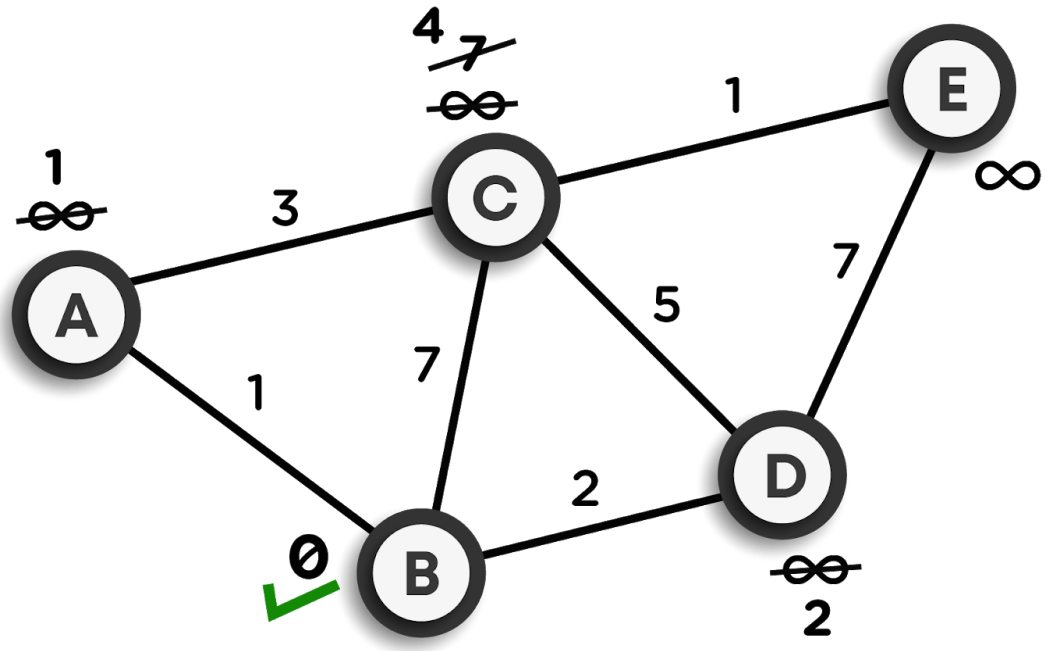
3. Now, we will check for the neighbors of the current node, which in our case is A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbor node. For example, for node B, it's weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.

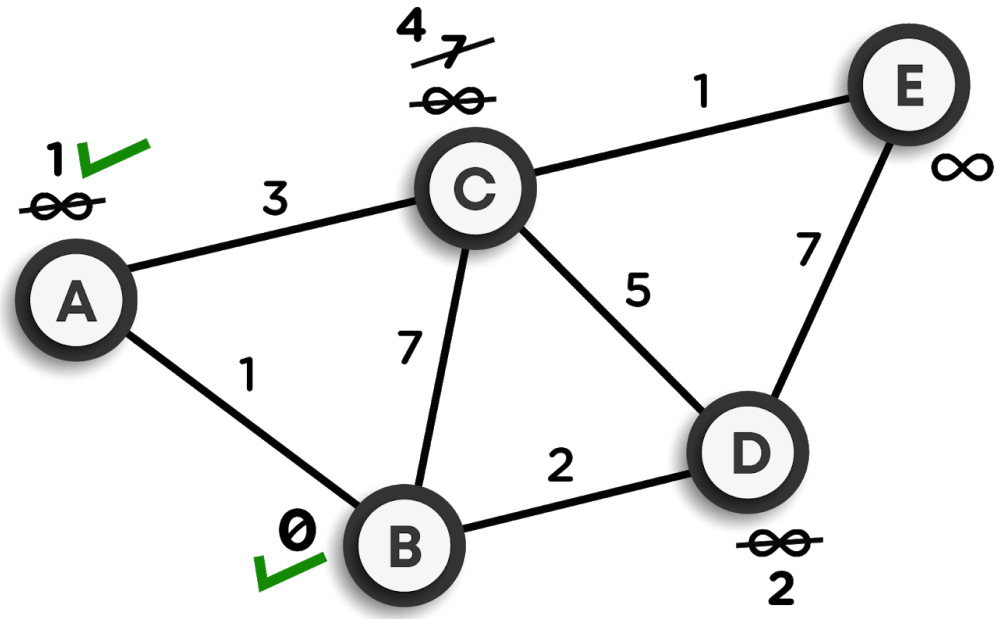


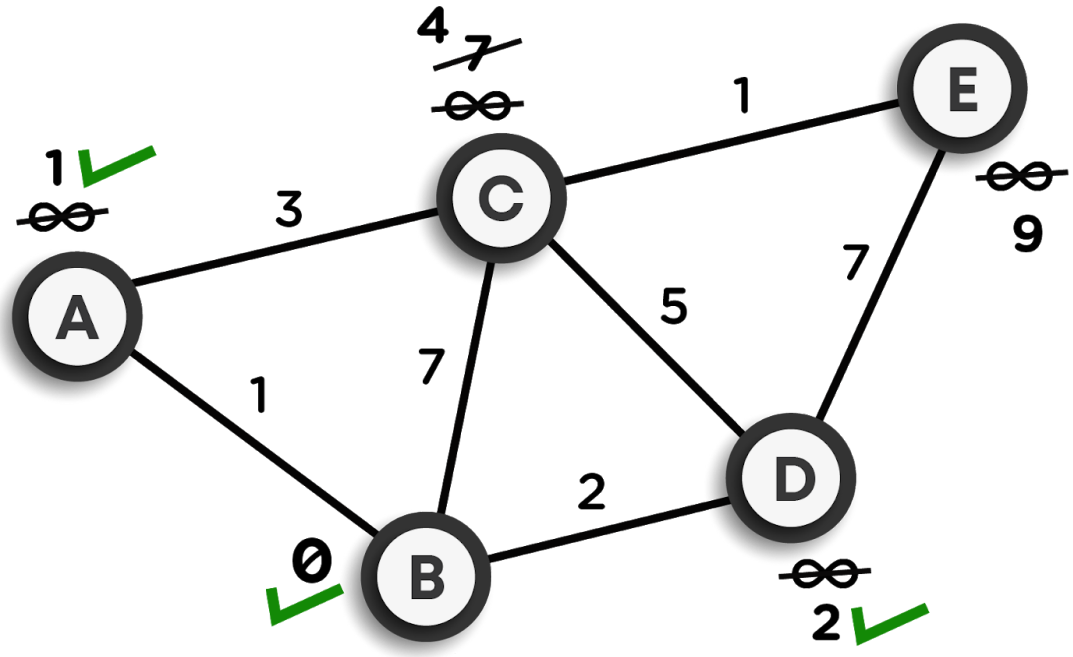
4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

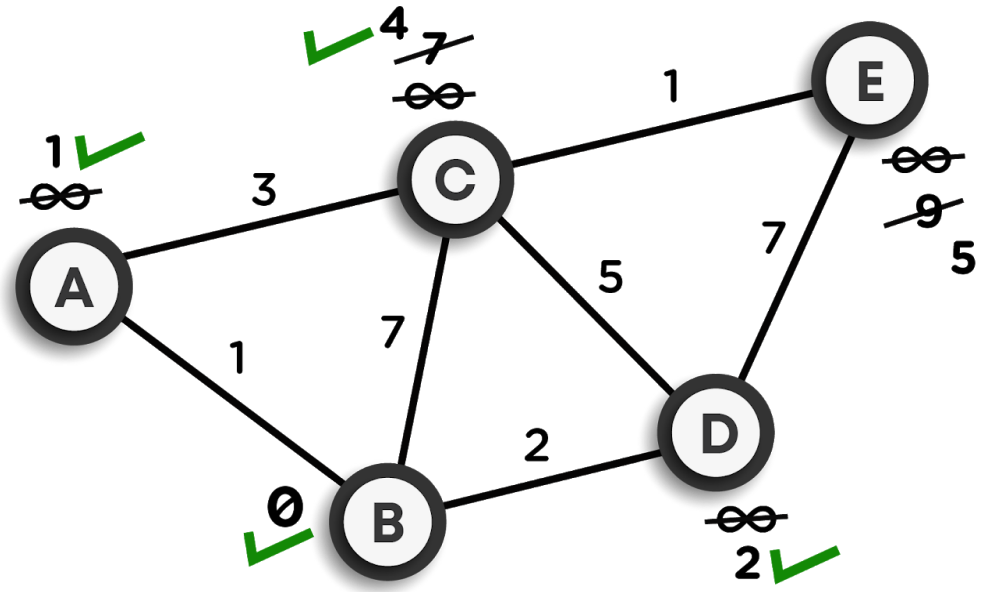


5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.
6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:

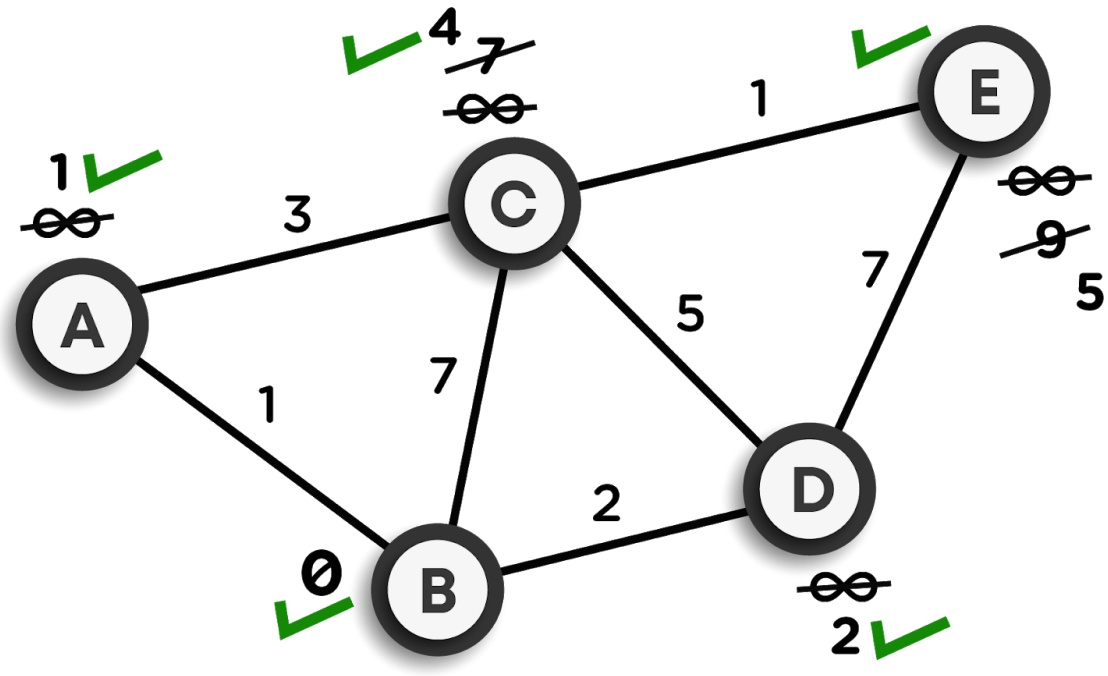



4
7
1
E
1
3
C
∞
A
7
5
7
1
2
D
0
B
2


4
7
1
E
1
3
C
9
A
7
5
7
1
2
D
0
B
2



7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

```
#include <iostream>
#include <climits>
using namespace std;

int findMinVertex(int *distance, bool *visited, int n) {

        int minVertex = -1;
        for (int  i = 0; i < n; i++) {
                if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex]))
{
                        minVertex = i;
                }
        }
        return minVertex;
}

void dijkstra(int **edges, int n) {

        int *distance = new int[n];
        bool *visited = new bool[n];

        for(int i = 0; i < n; i++) {
                visited[i] = false;
                distance[i] = INT_MAX;
        }

        distance[0] = 0;  // 0 is considered as the starting node.

        for (int i = 0; i < n-1; i++) {
                // Find min vertex
                int minVertex = findMinVertex(distance, visited, n);
                visited[minVertex] = true;
                // Explore unvisited neighbors
                for (int j = 0; j < n; j++) {
                        if(edges[minVertex][j] != 0 && !visited[j]) {
            // distance of any node will be the current node's distance + the weight
            // of the edge between them
                                int dist = distance[minVertex] + edges[minVertex][j];
```

```
                        if(dist < distance[j]) {     // If required, then updated.
                            distance[j] = dist;
                        }
                    }
                }
            }
            // Final output of distance of each node with respect to 0
            for (int i = 0; i < n; i++) {
                cout << i << " " << distance[i] << endl;
            }
}

int main() {

            int n;
            int e;
            cin >> n >> e;
            int **edges = new int*[n];
            for (int i = 0; i < n; i++) {
                edges[i] = new int[n];
                for (int j = 0; j < n; j++) {
                    edges[i][j] = 0;
                }
            }

            for (int i = 0; i < e; i++) {
                int f, s, weight;
                cin >> f >> s >> weight;
                edges[f][s] = weight;
                edges[s][f] = weight;
            }

            dijkstra(edges, n);

            for(int i = 0; i < n; i++) {
                delete [] edges[i];
            }
            delete [] edges;
            return 0;
}
```

Time Complexity of Dijkstra's algorithm:

The time complexity is also the same as that of Prim's algorithm, i.e., $O(n^2)$. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

C++ Foundation with Data Structures

Exception Handling

Exception Handling

An exception is a problem or error that arises during the execution of a program. There could be errors that cause the programs to fail or certain conditions that lead to errors. If these run time errors are not handled by the program, OS handles them and program terminates abruptly, which is not good. Few of such errors or error conditions are divide by zero, out of bound index, accessing memory not allowed to access, etc.

To avoid such conditions, C++ provides exception handling mechanism.
C++ exception handling is built upon three keywords: **try, catch, and throw**.

1. **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.
2. **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception
3. **try** − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks

Why we need Exception Handling:

1. Exception handling provide a way to transfer control from one part of a program to another and can avoid the brupt termination of program.

```
int main() {
    int n1, n2;
    try {
        cout << "Enter two nos:";
        cin >> n1 >> n2;
        if (n2 == 0)
            throw "Divide by zero";
        else
            throw n1/n2;
    }catch (char *s) {
        cout << s;
    }
    catch (int ans){
        cout << ans;
    }
    cout << "Done";
}
```

   In the above example we can see if user enters n2 as 0, then we will throw an exception and we can avoid abrupt termination of program.

2. Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

3. Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are

thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

4. Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types

Some examples to understand exception handling better

1. Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks

```
#include <iostream>
using namespace std;
int main() {
    int x = -1;
    cout << "Before try" << endl;
    try{
        cout << "Inside try \n";
        if (x < 0) {
            throw x;
            cout << "After throw" << endl;
        }
    } catch(int x) {
        cout << "Exception Caught" << endl;
    }
    cout << "After catch" << endl;
}
```

Output :
Before try
Inside try
Exception Caught
After catch

In the above example we can see the normal flow of throw, catch block.

We can observe that after the exception is thrown, the remaining try block after throw statement is not executed, instead program control goes to catch and never comes back. But all the code below the catch block is executed.
So to conclude code after throw is never executed but all the code after catch gets executed.

2. Above example shows that when exception is caught in try block it goes to a catch block whose parameter matched the throws argument, but if let's suppose I choose x to be double, then I need another catch block for that, which accepts double variable. To handle such conditions we have a special all catch block, which catches all types of exceptions.

```
#include <iostream>
```

```
using namespace std;
int main() {
    try {
        throw 10;
    } catch (char c) {
        cout <<  "character type exception" << endl;
    } catch(...) {
        cout << "Default Exception" << endl;
    }
}
```

Output:
Default Exception

We can see here that catch(...) catches all type of exceptions.
Few things about all catch blocks :

a) If there are multiple catch blocks, all catch block should be placed last.
b) Since all catch block (generic catch) is placed last, if any specific type exception occurs, it will try to find its specific catch block and if it is not found then it will go to the generic catch block.

3. Implicit type conversion doesn't happen for primitive types.

```
#include <iostream>
using namespace std;
int main() {
    try {
        throw 10;
    } catch (double c) {
        cout <<  "integer type exception" << endl;
    } catch(...) {
        cout << "Default Exception" << endl;
    }
}
```

Output :
Default Exception

Here in the above example, "throw 10" should be Implicitly get converted to double, but it doesn't get converted. Instead it get caught in generic catch block. So Implicit type casting doesn't happen for primitive type.

4. If an exception is thrown and not caught anywhere, the program terminates abnormally.

```
#include <iostream>
using namespace std;
int main() {
    try{
        char c = 'a';
        throw c;
    } catch(int x) {
        cout << "integer exception" << endl;
    }
```

```
}
```

Output :
terminating with uncaught exception of type char
Abort trap: 6

If the catch block for any throw doesn't exists it will terminate abruptly.

5. Nested try blocks

```
#include <iostream>
using namespace std;
int main() {
    try {
        try  {
            throw 20;
        } catch (int n) {
            cout << "Inner catch " << endl;
            throw;   //Re-throwing an exception
        }
    } catch (int n) {
        cout << "Outer catch" << endl;
    }
}
```

Output :
Inner catch
Outer catch

A function can also re-throw a function using same "throw". A function can handle a part and can ask the caller to handle remaining.

6. When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```
#include <iostream>
using namespace std;
class Test {
public:
          Test() { cout <<  "Constructor called" << endl; }
         ~Test() { cout << "Destructor called"  << endl; }
};
int main() {
           try {
            Test t1;
            throw 10;
        } catch(int i) {
            cout << "Caught " << i << endl;
        }
}
```

Output:
Constructor called

Destructor called
Caught 10

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Below is the example, which shows how you can use exception class to implement your own exception

```
#include <iostream>
#include <exception>
using namespace std;

class MyException : public exception {
    virtual const char * what () const throw () {
        return "C++ Exception";
    }
}myex;
int main() {
    try {
        throw myex;
    } catch(exception& e) {
        std::cout << e.what() << endl;
    } catch(MyException& e) {
        //Other errors
    }
}
```

Output for the above code :
MyException caught
C++ Exception

what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

OOPs 3

Object-Oriented Programming generally contains the following four concepts:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Let's study these in detail in other sections.

Abstraction & Encapsulation

You are travelling in a car and suddenly, you have to make it stop. For that, you will be applying the brake. This whole system is enclosed inside the car's body. This process is known as encapsulation. To stop the car, you just have to apply the brake without thinking of the internal functioning of the car's engine system.

Encapsulation refers to combining data along with the methods operating on that data into a single unit called class. It also refers to the mechanism of binding the direct access state values for all the variables of the particular object. The main reason for using encapsulation is to provide security to the data of the class.

Reasons for which encapsulation is considered are:

- Functionality is defined at a logical place and not at multiple locations.
- It prevents our data from being modified from any external influence.

Abstraction means providing only some part of the information to the user by hiding the internal implementation details of it. We just need to know about the methods of the objects that we need to call and the input parameters needed to trigger a specific operation excluding the details of implementation and type of action performed to get the result.

Reasons for using abstraction are:

- It allows us to group different related units(classes) as siblings.
- It helps in the reduction of design and implementation complexity.

Inheritance

Suppose we have three classes with names: car, bicycle, and truck. The properties for each are as follows:

Car	Bicycle	Truck
• Colour • MaxSpeed • Number of gears	• Colour • MaxSpeed • Is Foldable?	• Colour • MaxSpeed • Max weight

From above, we can see that two of the properties: Colour and MaxSpeed, are the same for every object. Hence, we can combine all these in one parent class and make the above three classes as its subclass. This property is called Inheritance.

Technically, inheritance is defined as the process of acquiring the features and behaviours of a class by another class. Here, the class that contains these members is called the **base class** and the class that inherits these members from the base class is called the **derived class** of that base class**.**

Access Modifiers

When creating a derived class from a base class, we need to use access modifiers to inherit data members of the base class. These are:

- Public
- Private
- Protected

The **public** data members can be accessed by any of its child class as well as the class objects.

Private data members are inaccessible outside the class. These can't be accessed even by the child classes.

Protected data members can only be accessed by the derived classes but are inaccessible using the class objects.

Inheritance: Syntax

The syntax for inheriting a base class by a derived class:

```
class derived_class_name : access_modifier base_class_name {
        ----
        ----
};
```

Kindly look at the following example for better understanding:

```
#include <iostream>
using namespace std;

class Vehicle {                                    // Parent class
        private :
                int maxSpeed;

        protected :
                int numTyres;

        public :
                string color;
};

class Car : public Vehicle {                       // Child class with Public access modifier
        public :
                int numGears;
                void print() {
                        // protected data member accessible
                        cout << "NumTyres : " << numTyres << endl;
                        // public data member accessible
                        cout << "Color : " << color << endl;
                        cout << "Num gears : " << numGears << endl;
                }
};
```

```
int main() {
        Vehicle v;
        v.color = "Blue";
        // v.maxSpeed = 100;          // Can't be accessed being private
        // v.numTyres = 4;            // Can't be accessed being protected

        Car c;
        c.color = "Black";
        // c.numTyres = 4;            // Can't be accessed being protected
        c.numGears = 5;
}
```

Order of Constructor/Destructor

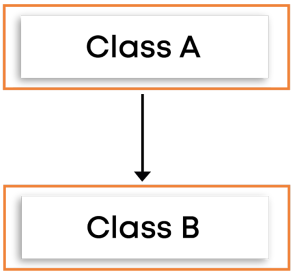
Constructors and Destructors follow a specific order of invocation.

- In the case of constructors, base class constructors are called first, and the derived class constructors are called next. Moreover, the order of constructor invocation depends on the order of how the base class is inherited.
- Destructors are called in reverse order of the constructor invocation, i.e., the destructor of the derived class is called first followed by the destructor of the base class sequentially.

Inheritance: Types

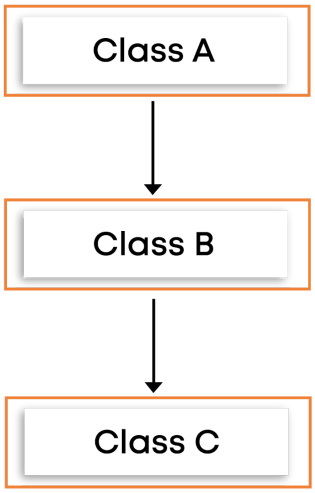
There are six types of inheritance:

- **Single Inheritance:** Here, a child class is created from a single parent class.



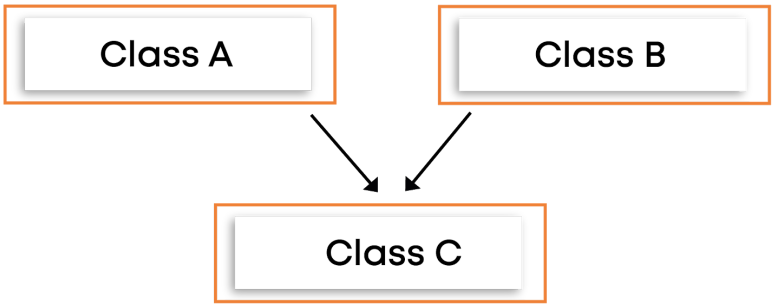
Single Inheritance

- **Multi-level Inheritance:** Here, there is a series of derived classes, which means a derived class is created from another derived class.



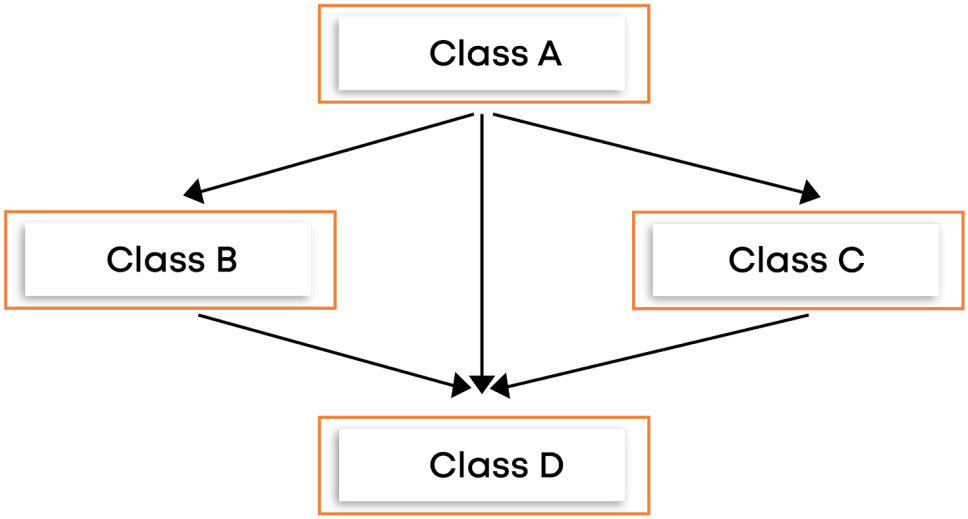
Multi-Level Inheritance

- **Multiple Inheritance:** Here, a child class is created from more than one parent/base class.



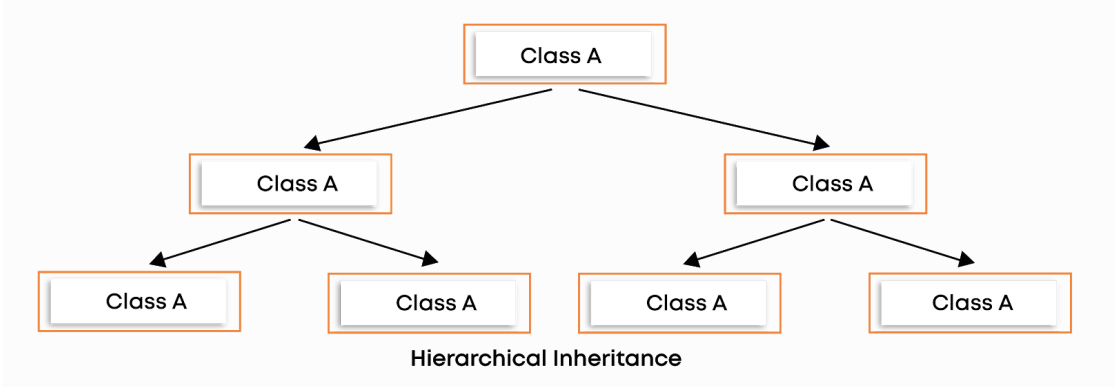
Multiple Inheritance

- **Multipath Inheritance:** In this type of inheritance, a derived class is created from other derived classes and the same base class of other derived classes.



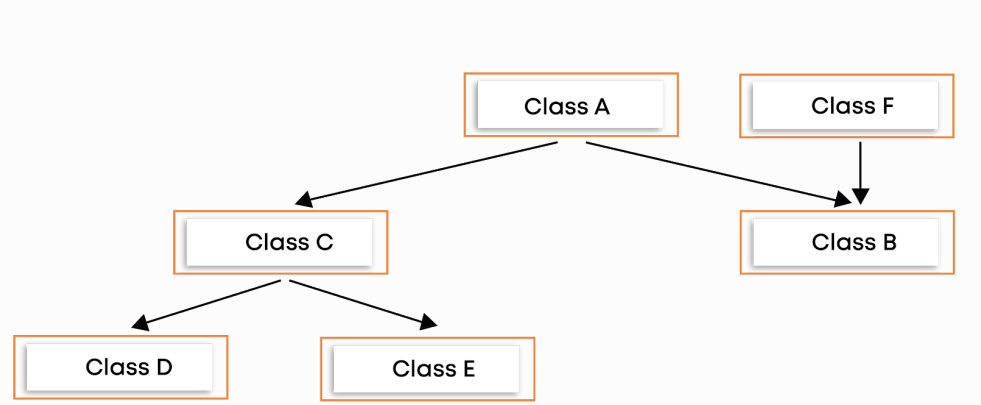
Multipath Inheritance

- **Hierarchical Inheritance:** Here, more than one derived classes are created from a single base class, and further, child classes act as parent classes for more than one child class.



Hierarchical Inheritance

- **Hybrid Inheritance:** It is the combination of more than one inheritances. It is also known as **Diamond inheritance**.



Hybrid Inheritance- (a combination of Hierarchical and multiple)

Polymorphism

The word polymorphism means **different forms**. It occurs when multiple classes are related to each other by inheritance. In C++, polymorphism means, a call to a member function will cause a different function to be executed depending on the type of object that invokes it. For example, (+) sign is used as an addition operator as well as the concatenation operator.

Polymorphism is generally of two types:

- Compile-time polymorphism
- Run-time polymorphism

Compile-time polymorphism:

It demonstrates the properties of static/early binding and occurs in the following cases:

- Function Overloading and Operator Overloading
- Function Overriding

We already have studied operator overloading. So, let's discuss function overloading.

When two or more functions have the same name but differ in any of the number of arguments or the return-type, then this process is known as **Function Overloading.**

Note: In static/early binding, the compiler (or linker) directly associates an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

For example:

```
#include <iostream>
using namespace std;

int test(int a, int b) {                // function with 2 arguments

}
int test(int a) {                       // function with 1 argument

}
int test() {                            // function with no argument

}
int main() {
    // All the above functions have same name but differ in number of arguments
}
```

Now, suppose we have two classes, A and B where A is the base/parent class, and B is the child/derived class. If the base class has a function named *print()* and a child

class of it also contains such a function, then this is a case of function overriding. Here, we are deciding at the compile time about the function to be called. Look at the code below:

```
#include <iostream>
using namespace std;

class Vehicle {
        public :
                string color;
                void print() {
                        cout << "Vehicle" << endl;
                }
};

class Car : public Vehicle {
        public :
                int numGears;
                void print() {
                        cout << "Car" << endl;
                }
};

int main() {
        Vehicle v;
        Car c;

        v.print();
        c.print();

        Vehicle *v1 = new Vehicle;
        Vehicle *v2;

        v2 = &c;
        v1 -> print();          // This will print Vehicle class' print()
        v2 -> print();          // This will also print Vehicle class' print() due to overriding
}
```

Run-time polymorphism:

Run-time polymorphism demonstrates the property of dynamic resolution or late binding and is achieved by using function overriding.

In the above code, we saw that by using the car class' object, when we were calling the print() function, we were redirected to Vehicle class's print() function. Suppose, in the aforementioned example, if we want that instead of the base class'(vehicle) print function, the derived class'(car) print function gets called, we can use **Virtual Functions** to achieve the same..

When the base class's function is overridden in the child class, then that function is known as a virtual function. Keyword **virtual** is used for the same. By doing so, we are redirecting the compiler to take the decision at runtime only.

```
#include <iostream>
using namespace std;

class Vehicle {
        public :
                string color;
                void print() {
                        cout << "Vehicle" << endl;
                }
};

class Car : public Vehicle {
        public :
                int numGears;
                virtual void print() {                  // function made virtual
                        cout << "Car" << endl;
                }
};

int main() {
        Vehicle v;
        Car c;

        v.print();
        c.print();

        Vehicle *v1 = new Vehicle;
        Vehicle *v2;
        v2 = &c;
        v1 -> print();      // This will print Vehicle class' print()
        v2 -> print();      // This will now print car class' print() due to virtual function
}
```

Pure Virtual functions and Abstract classes

A virtual function whose declaration ends with =0 is called a **pure virtual function**. These functions do not have any definition. An Abstract class has at least one pure virtual function. Abstract classes are those that can't be instantiated, i.e., we cannot create an object of this class. However, we can derive a class from it and instantiate the object of the derived class.

Properties of the abstract classes:

- It can have normal functions and variables along with the pure virtual functions.
- Prominently used for upcasting(converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as a base type), so its derived classes can use its interface.
- If an abstract class has derived class, they must implement all pure virtual functions, or else they will become abstract too.

Kindly check the following code for better understanding.

```
#include <iostream>
using namespace std;

class Vehicle {  // This class becomes abstract as it contains a pure virtual function
        public :
                string color;
                // Pure virtual function
                virtual void print() = 0;
};
```

Friend function and classes

Data hiding is an essential part of OOPs which means a non-member function is restricted from accessing an object's private or protected data. This restriction

sometimes forces us to write long and complicated codes; Instead we can use friend function/class to avoid the same.

Friend Function:

A friend function can access the private and protected data members of a class. To declare it, we use the keyword **friend.** For accessing the data, the friend function should be declared inside the body of the class.

Friend Class:

It is similar to the friend function. A class can be made a friend of another class by using the same keyword **friend.** Creating a class as a friend class, all the data functions of the class becomes friend functions.

Kindly follow the code below for better understanding.

```
#include <iostream>
using namespace std;

class Bus {
        public :
                void print();
};

void test();

class Truck {
        private :
                int x;

        protected :
                int y;

        public :
                int z;
                friend class Bus; // Bus class declared as friend class to the Truck class
        /*
                friend void Bus :: print();      // Invoking the print() using friend class

                friend void test();              // In short, making a friend function itself.
        */
```

```
};

void Bus :: print() {
        Truck t;
        t.x = 10;
        t.y = 20;
        cout << t.x << " " << t.y << endl;
}

void test() {
// Access truck private (Not Possible, hence gives the error of inaccessible private variables)
        Truck t;
        t.x = 10;
        t.y = 20;
        cout << t.x << " " << t.y << endl;
}

int main() {
        Bus b;
        b.print();
        test();
}
```