Introduction to Data Structures and Algorithms

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

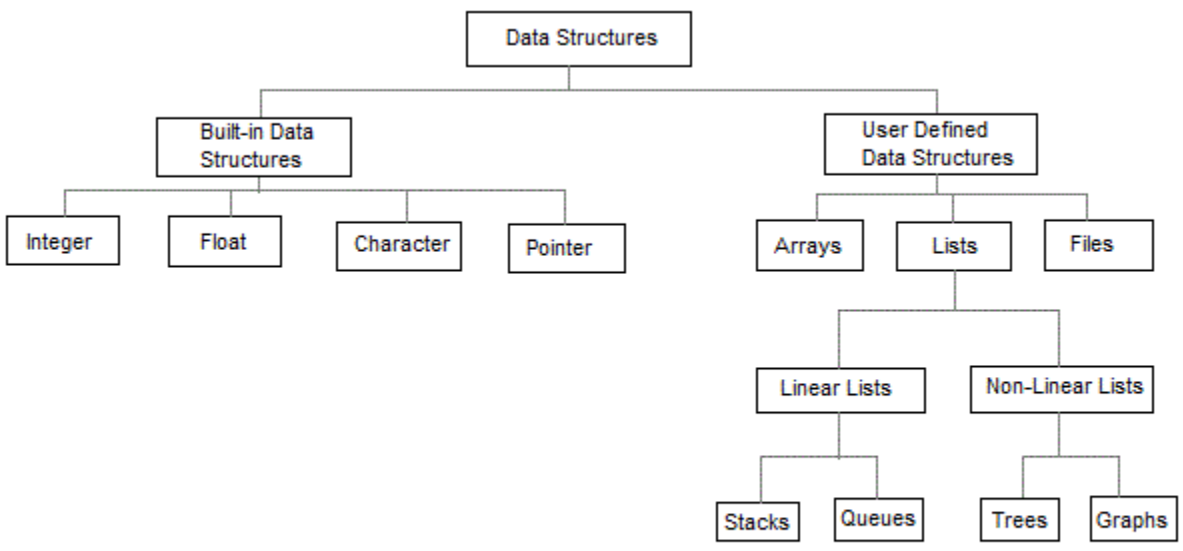
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	Description
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

What is an Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** Its the space required to store all the constants and variables(including temporary variables) value.
- **Environment Space:** Its the space required to store the environment information needed to resume the suspended function.

To learn about Space Complexity in detail, jump to the Space Complexity tutorial.

Time Complexity

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible. We will study about Time Complexity in details in later sections.

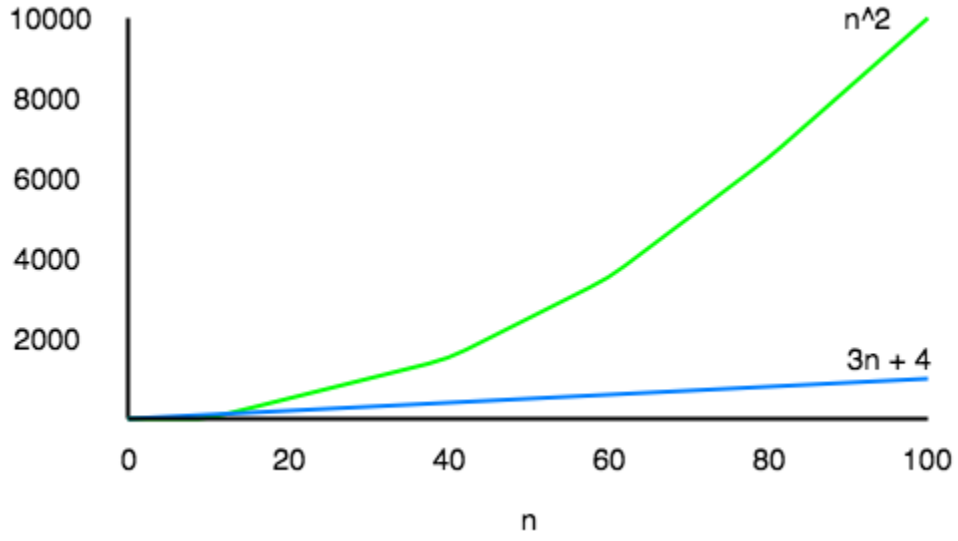
NOTE: Before going deep into data structure, you should have a good knowledge of programming either in C or in C++ or Java or Python etc.

Asymptotic Notations

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

Let us take an example, if some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation. For large values of n, the $3n + 4$ part will become insignificant compared to the n^2 part.



For n = 1000, n^2 will be 1000000 while $3n + 4$ will be 3004.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of $200n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 200. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

What is Asymptotic Behaviour

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Remember studying about **Limits** in High School, this is the same.

The only difference being, here we do not have to find the value of any expression where n is approaching any finite number or infinity, but in case of Asymptotic notations, we use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Let's take an example to understand this:

If we have two algorithms with the following expressions representing the time required by them for execution, then:

Expression 1: ($20n^2 + 3n - 4$)

Expression 2: ($n^3 + 100n - 2$)

Now, as per asymptotic notations, we should just worry about how the function will grow as the value of n(input) will grow, and that will entirely depend on n^2 for the Expression 1, and on n^3 for Expression 2. Hence, we can clearly say that the algorithm for which running time is represented by the Expression 2, will grow faster than the other one, simply by analysing the highest power coeeficient and ignoring the other constants(20 in $20n^2$) and insignificant parts of the expression($3n - 4$ and $100n - 2$).

The main idea behind casting aside the less important part is to make things **manageable**.

All we need to do is, first analyse the algorithm to find out an expression to define it's time requirements and then analyse how that expression will grow as the input(n) will grow.

Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

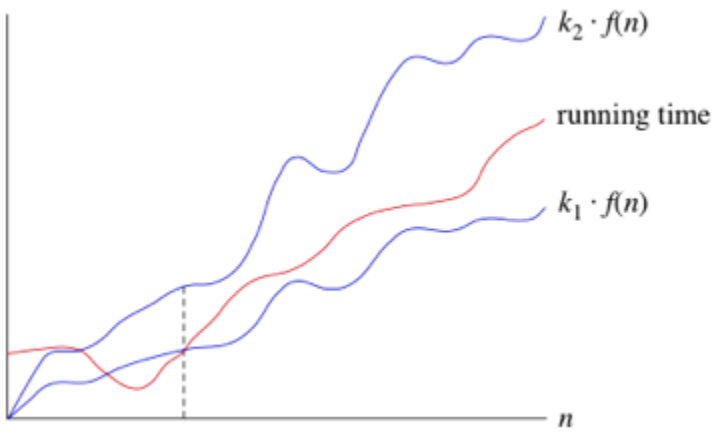
1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the avaerage time for any input n will remain in between, $k1 * n^2$ and $k2 * n^2$, where k1, k2 are two constants, therby tightly binding the expression rpresenting the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In **Worst case**, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of n, where n represents the number of total elements.

But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to n, as this is the right way to represent the average time complexity, but when we use the big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it makes more sense.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to cmplete it's execution. It can definitely take more time than this too.

Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = **Auxiliary Space + Input space**

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function `A()` calls function `B()` inside it, then all th variables of the function `A()` will get stored on the system stack temporarily, while the function `B()` is called and executed inside the funciton `A()`.

3. **Data Space**

   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte

__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{

    int z = a + b + c;

    return(z);

}
```

Copy

In the above expression, variables `a`, `b`, `c` and `z` are all integer types, hence they will take up 4 bytes each, so total memory requirement will be `(4(4) + 4) = 20 bytes`, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, this time a bit complex one,

```
// n is the length of array a[]

int sum(int a[], int n)

{

     int x = 0;      // 4 bytes for x

     for(int i = 0; i < n; i++)     // 4 bytes for i

     {

         x  = x + a[i];

     }

     return(x);

}
```

Copy

- In the above code, `4*n` bytes of space is required for the array `a[]` elements.
- 4 bytes each for `x`, `n`, `i` and the return value.

Hence the total memory requirement will be `(4n + 12)`, which is increasing linearly with the increase in the input value `n`, hence it is called as **Linear Space Complexity.**

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

Time Complexity of Algorithms

For any defined problem, there can be N number of solution. This is true in general. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number `n` is `n*n`):

One solution to this problem can be, running a loop for `n` times, starting with the number `n` and adding `n` to it, every time.

```
/*

    we have to calculate the square of n

*/

for i=1 to n

    do n = n + n

// when the loop ends n will hold its square

return n
```

Copy

Or, we can simply use a mathematical operator `*` to find the square.

```
/*

    we have to calculate the square of n

*/

return n*n
```

Copy

In the above two simple algorithms, you saw how a single problem can have many solutions. While the first solution required a loop which will execute for `n` number of times, the second solution used a mathematical operator `*` to return the result in one line. So which one is the better approach, of course the second one.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)

{

    statement;

}
```

Copy

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)

{

    for(j=0; j < N;j++)

    {

    statement;

    }

}
```

Copy

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)

{

    mid = (low + high) / 2;

    if (target < list[mid])

        high = mid - 1;

    else if (target > list[mid])

        low = mid + 1;

    else break;

}
```

Copy

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)

{

    int pivot = partition(list, left, right);

    quicksort(list, left, pivot - 1);

    quicksort(list, pivot + 1, right);

}
```

Copy

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.

4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose you've calculated that an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Copy

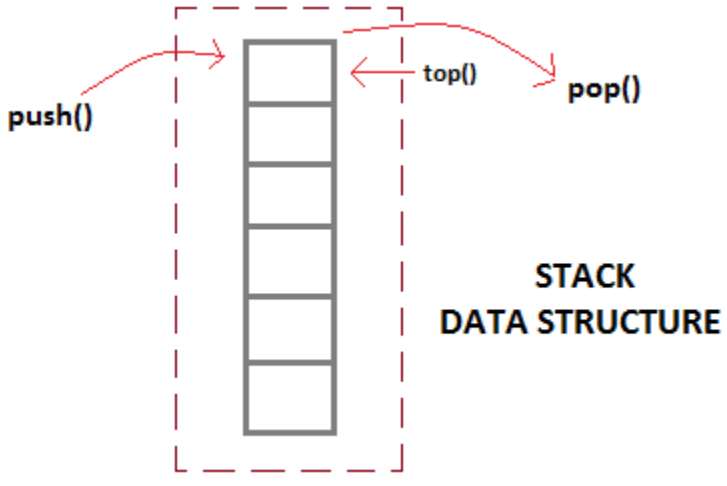
Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

Now that we have learned the Time Complexity of Algorithms, you should also learn about Space Complexity of Algorithms and its importance.

What is Stack Data Structure?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. `push()` function is used to insert new elements into the Stack and `pop()` function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Applications of Stack

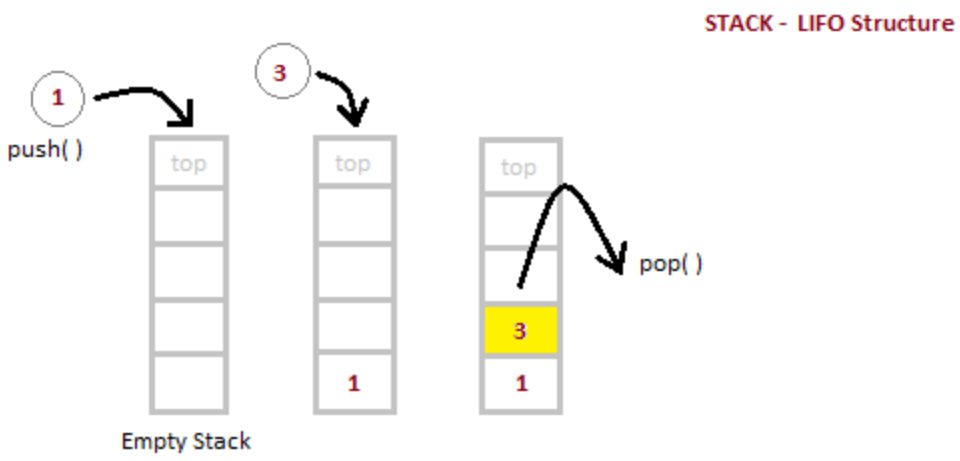
The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.
The "**pop**" operation removes the item on top of the stack.

Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

Below we have a simple C++ program implementing stack data structure while following the object oriented programming concepts.

If you are not familiar with C++ programming concepts, you can learn it form here.

```
/*  Below program is written in C++ language  */


# include<iostream>


using namespace std;


class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
```

```
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    void isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
    {
        a[++top] = x;
        cout << "Element Inserted \n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
```

```
        return 0;

    }

    else

    {

        int d = a[top--];

        return d;

    }

}


// function to check if stack is empty

void Stack::isEmpty()

{

    if(top < 0)

    {

        cout << "Stack is empty \n";

    }

    else

    {

        cout << "Stack is not empty \n";

    }

}


// main function

int main() {


    Stack s1;

    s1.push(10);

    s1.push(100);

    /*

        preform whatever operation you want on the stack
```

```
    */

}
```

Copy

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

The time complexities for `push()` and `pop()` functions are `O(1)` because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

Now that we have learned about the Stack in Data Structure, you can also check out these topics:

- Queue Data Structure
- Queue using stack

What is a Queue Data Structure?

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

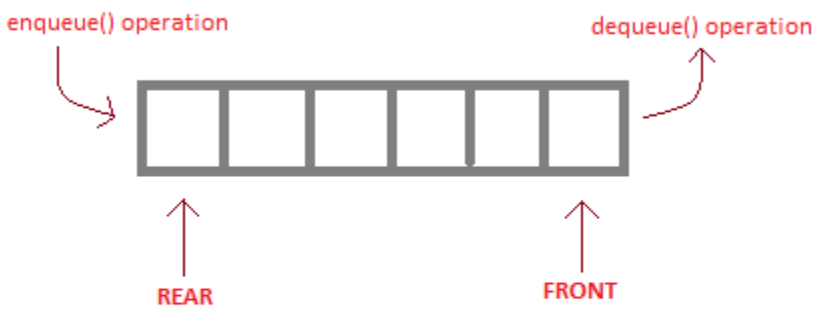
Before you continue reading about queue data structure, check these topics before to understand it clearly:

- Data Structures and Algorithms
- Stack Data Structure

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. `peek( )` function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

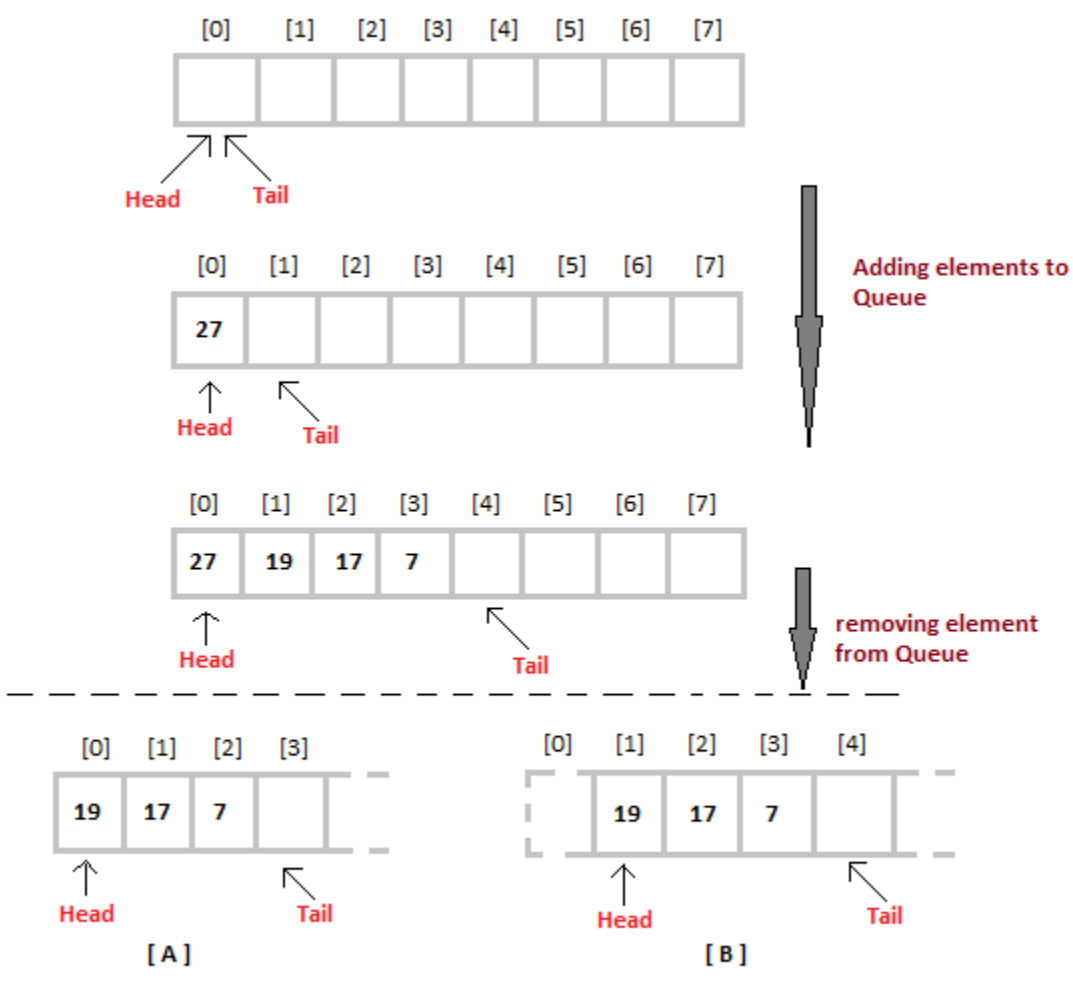
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

```
/* Below program is written in C++ language */

#include<iostream>

using namespace std;

#define SIZE 10

class Queue
{
    int a[SIZE];
    int rear;   //same as tail
    int front;  //same as head

public:
Queue()
{
    rear = front = -1;
}

//declaring enqueue, dequeue and display functions
void enqueue(int x);
int dequeue();
```

```
    void display();
};


// function enqueue - to add data to queue
void Queue :: enqueue(int x)
{
    if(front == -1) {
        front++;
    }
    if( rear == SIZE-1)
    {
        cout << "Queue is full";
    }
    else
    {
        a[++rear] = x;
    }
}


// function dequeue - to remove data from queue
int Queue :: dequeue()
{
    return a[++front];  // following approach [B], explained above
}


// function to display the queue elements
void Queue :: display()
{
    int i;
    for( i = front; i <= rear; i++)
```

```
    {
        cout << a[i] << endl;
    }
}


// the main function
int main()
{
    Queue q;
    q.enqueue(10);
    q.enqueue(100);
    q.enqueue(1000);
    q.enqueue(1001);
    q.enqueue(1002);
    q.dequeue();
    q.enqueue(1003);
    q.dequeue();
    q.dequeue();
    q.enqueue(1004);


    q.display();



    return 0;
}
```

Copy

To implement approach [A], you simply need to change the `dequeue` method, and include a `for` loop which will shift all the remaining elements by one position.

```
return a[0];    //returning first element
for (i = 0; i < tail-1; i++)    //shifting all other elements
{
```

```
    a[i] = a[i+1];

    tail--;

}
```

Copy

Complexity Analysis of Queue Operations

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

Implement Queue using Stacks

A Queue is defined by its property of **FIFO**, which means First in First Out, i.e the element which is added first is taken out first. This behaviour defines a queue, whereas data is actually stored in an **array** or a **list** in the background.

What we mean here is that no matter how and where the data is getting stored, if the first element added is the first element being removed and we have implementation of the functions `enqueue()` and `dequeue()` to enable this behaviour, we can say that we have implemented a Queue data structure.

In our previous tutorial, we used a simple **array** to store the data elements, but in this tutorial we will be using **Stack data structure** for storing the data.

While implementing a queue data structure using stacks, we will have to consider the natural behaviour of stack too, which is **First in Last Out**.

For performing **enqueue** we require only **one stack** as we can directly **push** data onto the stack, but to perform **dequeue** we will require **two Stacks**, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach(Last in First Out).

Implementation of Queue using Stacks

In all we will require two Stacks to implement a queue, we will call them `S1` and `S2`.

```
class Queue {

    public:

    Stack S1, S2;



    //declaring enqueue method

    void enqueue(int x);



    //declaring dequeue method

    int dequeue();

}
```

Copy

In the code above, we have simply defined a class `Queue`, with two variables `S1` and `S2` of type `Stack`.

We know that, Stack is a data structure, in which data can be added using `push()` method and data can be removed using `pop()` method.

You can find the code for `Stack` class in the Stack data structure tutorial.

To implement a queue, we can follow two approaches:

1. **By making the `enqueue` operation costly**
2. **By making the `dequeue` operation costly**

---

1. Making the Enqueue operation costly

In this approach, we make sure that the oldest element added to the queue stays at the **top** of the stack, the second oldest below it and so on.

To achieve this, we will need two stacks. Following steps will be involved while enqueuing a new element to the queue.

NOTE: First stack(`S1`) is the main stack being used to store the data, while the second stack(`S2`) is to assist and store data temporarily during various operations.

1. If the queue is empty(means `S1` is empty), directly **push** the first element onto the stack `S1`.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then add the new element to the first stack, then move back all the elements from the second stack back to the first stack.
3. Doing so will always maintain the right order of the elements in the stack, with the 1st data element staying always at the **top**, with 2nd data element right below it and the new data element will be added to the bottom.

This makes removing an element from the queue very simple, all we have to do is call the `pop()` method for stack `S1`.

---

2. Making the Dequeue operation costly

In this approach, we insert a new element onto the stack `S1` simply by calling the `push()` function, but doing so will will push our first element towards the bottom of the stack, as we insert more elements to the stack.

But we want the first element to be removed first. Hence in the `dequeue` operation, we will have to use the second stack `S2`.

We will have to follow the following steps for `dequeue` operation:

1. If the queue is empty(means `S1` is empty), then we return an error message saying, the queue is empty.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then remove the element at the **top** from the second stack, and then move back all the elements from the second stack to the first stack.
3. The purpose of moving all the elements present in the first stack to the second stack is to reverse the order of the elements, because of which the first element inserted into the queue, is positioned at the top of the second stack, and all we have to do is call the `pop()` function on the second stack to remove the element.

NOTE: *We will be implementing the **second approach**, where we will make the `dequeue()` method costly.*

Adding Data to Queue - `enqueue()`

As our Queue has a Stack for data storage instead of arrays, hence we will be adding data to Stack, which can be done using the `push()` method, therefore `enqueue()` method will look like:

```
void Queue :: enqueue(int x)

{

    S1.push(x);

}
```

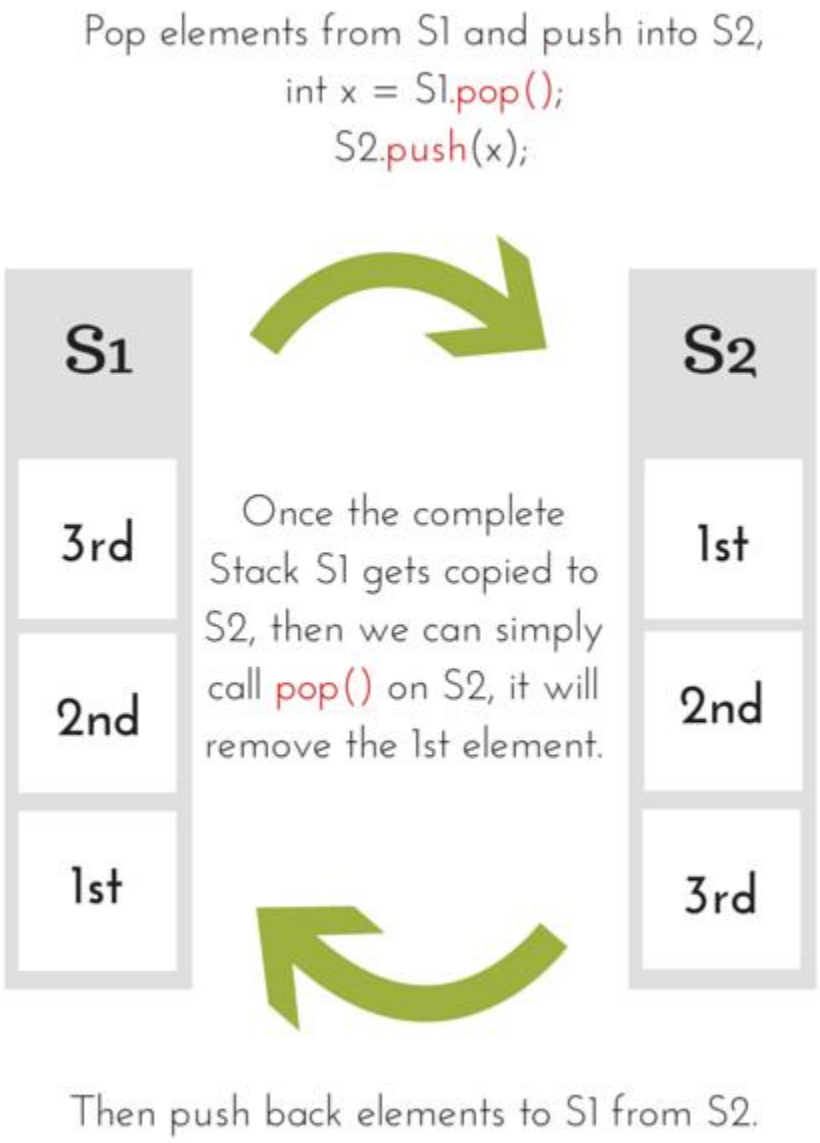
Copy

That's it, new data element is enqueued and stored in our queue.

Removing Data from Queue - `dequeue()`

When we say remove data from Queue, it always means taking out the element which was inserted first into the queue, then second and so on, as we have to follow the **FIFO approach**.

But if we simply perform `S1.pop()` in our **dequeue** method, then it will remove the Last element inserted into the queue first. So what to do now?



As you can see in the diagram above, we will move all the elements present in the first stack to the second stack, and then remove the **top** element, after that we will move back the elements to the first stack.

```
int Queue :: dequeue()

{

    int x, y;

    while(S1.isEmpty())

    {

        // take an element out of first stack

        x = S1.pop();

        // insert it into the second stack

        S2.push();

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}
```

Copy

Now that we know the implementation of `enqueue()` and `dequeue()` operations, let's write a complete program to implement a queue using stacks.

Implementation in C++(OOPS)

We will not follow the traditional approach of using pointers, instead we will define proper classes, just like we did in the Stack tutorial.

```
/*  Below program is written in C++ language  */
```

```
# include<iostream>

using namespace std;

// implementing the stack class
class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    bool isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
```

```
    {
        a[++top] = x;
        cout << "Element Inserted into Stack\n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
        int d = a[top--];
        return d;
    }
}

// function to check if stack is empty
bool Stack::isEmpty()
{
    if(top < 0)
    {
        return true;
    }
    else
    {
```

```
        return false;
    }
}


// implementing the queue class
class Queue {
    public:
    Stack S1, S2;

    //declaring enqueue method
    void enqueue(int x);

    //declaring dequeue method
    int dequeue();
};


// enqueue function
void Queue :: enqueue(int x)
{
    S1.push(x);
    cout << "Element Inserted into Queue\n";
}


// dequeue function
int Queue :: dequeue()
{
    int x, y;
    while(!S1.isEmpty())
    {
        // take an element out of first stack
```

```
        x = S1.pop();

        // insert it into the second stack

        S2.push(x);

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}


// main function

int main()

{

    Queue q;

    q.enqueue(10);

    q.enqueue(100);

    q.enqueue(1000);

    cout << "Removing element from queue" << q.dequeue();


    return 0;

}
```

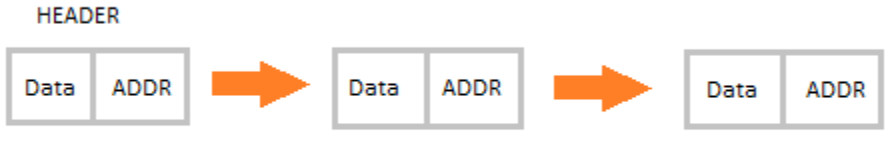
Copy

Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

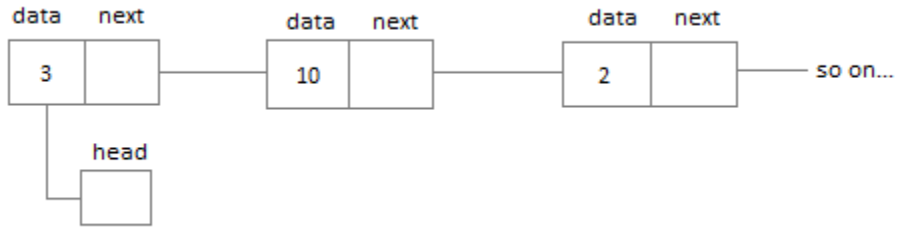
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

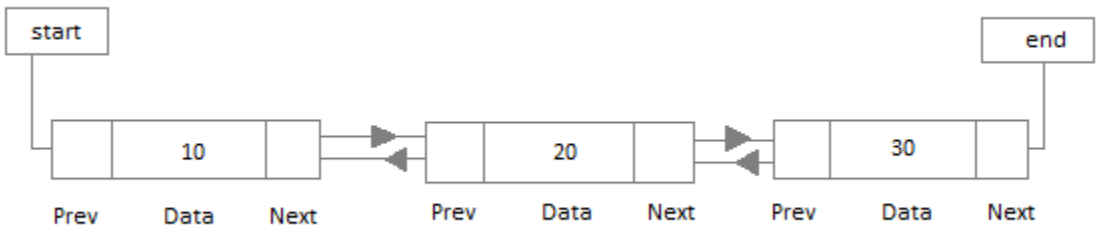
Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. `next`, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



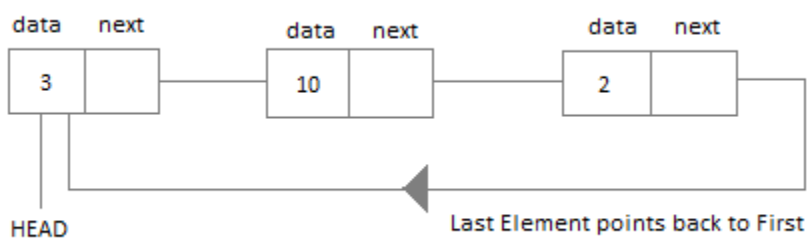
Doubly Linked List

In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



We will learn about all the 3 types of linked list, one by one, in the next tutorials. So click on **Next** button, let's learn more about linked lists.

Difference between Array and Linked List

Both Linked List and Array are used to store linear data of similar type, but an array consumes contiguous memory locations allocated at compile time, i.e. at the time of declaration of array, while for a linked list, memory is assigned as and when data is added to it, which means at runtime.

Before we proceed further with the differences between Array and Linked List, if you are not familiar with Array or Linked list or both, you can check these topics first:

- Array in C
- Linked List

This is the basic and the most important difference between a linked list and an array. In the section below, we will discuss this in details along with highlighting other differences.

Linked List vs. Array

Array is a datatype which is widely implemented as a default type, in almost all the modern programming languages, and is used to store data of similar type.

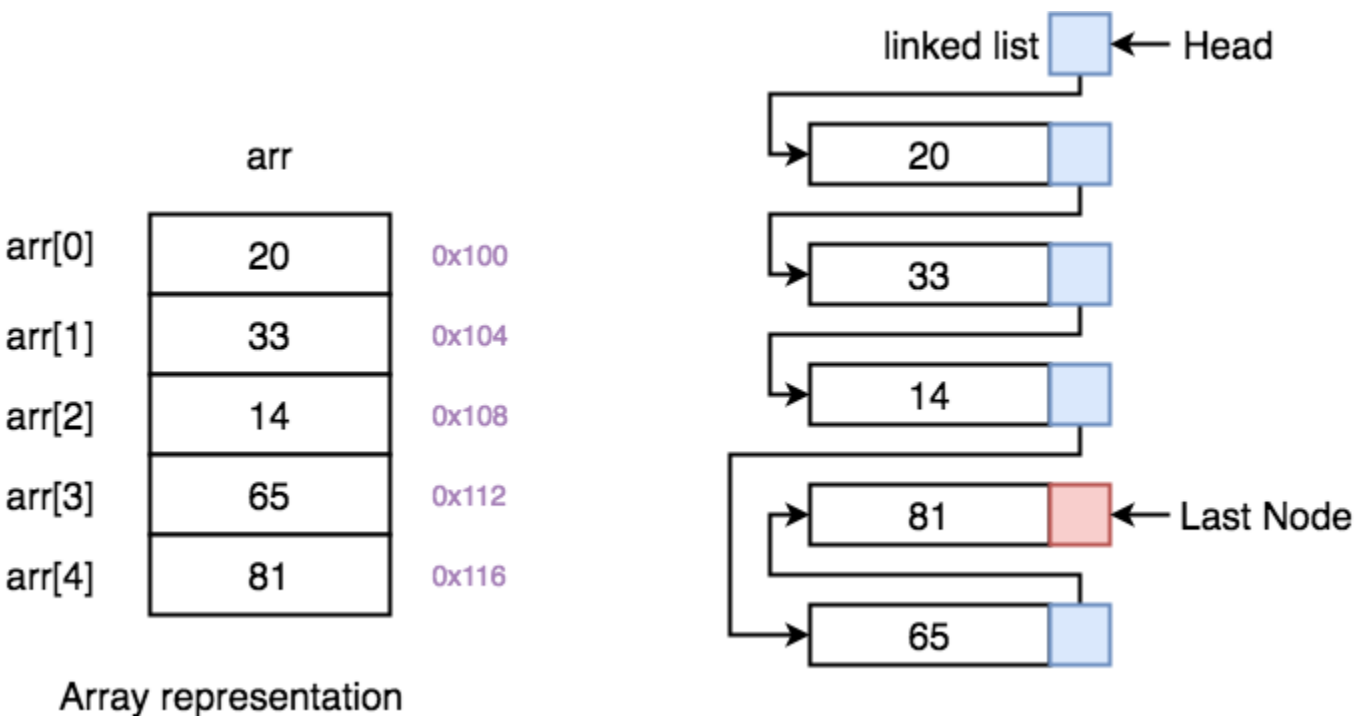
But there are many usecases, like the one where we don't know the quantity of data to be stored, for which advanced data structures are required, and one such data structure is **linked list**.

Let's understand how array is different from Linked list.

ARRAY	LINKED LIST
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index, like `arr[0]` for 1st element, `arr[6]` for 7th element etc. Hence, accessing elements in an array is **fast** with a constant time complexity of `O(1)`.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element. To access **nth** element of a linked list, time complexity is `O(n)`.
In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory.	In a linked list, new elements can be stored anywhere in the memory. Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements.

In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list. Insertion and Deletion operations are **fast** in linked list.
Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**.	Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.
In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can be **single dimensional**, **two dimensional** or **multidimensional**	Linked list can be **Linear(Singly) linked list**, **Doubly linked list** or **Circular linked list** linked list.
Size of the array must be specified at time of array decalaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the **Stack** section.	Whereas, linked list gets memory allocated in **Heap** section.

Below we have a pictorial representation showing how consecutive memory locations are allocated for array, while in case of linked list random memory locations are assigned to nodes, but each node is connected to its next node using **pointer**.



On the left, we have **Array** and on the right, we have **Linked List**.

Why we need pointers in Linked List? [Deep Dive]

In case of array, memory is allocated in contiguous manner, hence array elements get stored in consecutive memory locations. So when you have to access any array element, all we have to do is use the array index, for example `arr[4]` will directly access the 5th memory location, returning the data stored there.

But in case of linked list, data elements are allocated memory at runtime, hence the memory location can be anywhere. Therefore to be able to access every node of the linked list, address of every node is stored in the previous node, hence forming a link between every node.

We need this additional **pointer** because without it, the data stored at random memory locations will be lost. We need to store somewhere all the memory locations where elements are getting stored.

Yes, this requires an additional memory space with each node, which means an additional space of $O(n)$ for every n node linked list.

Linear Linked List

Linear Linked list is the default linked list and a linear data structure in which data is not stored in contiguous memory locations but each data node is connected to the next data node via a pointer, hence forming a chain.

The element in such a linked list can be inserted in 2 ways:

- Insertion at beginning of the list.
- Insertion at the end of the list.

Hence while writing the code for Linked List we will include methods to insert or add new data elements to the linked list, both, at the beginning of the list and at the end of the list.

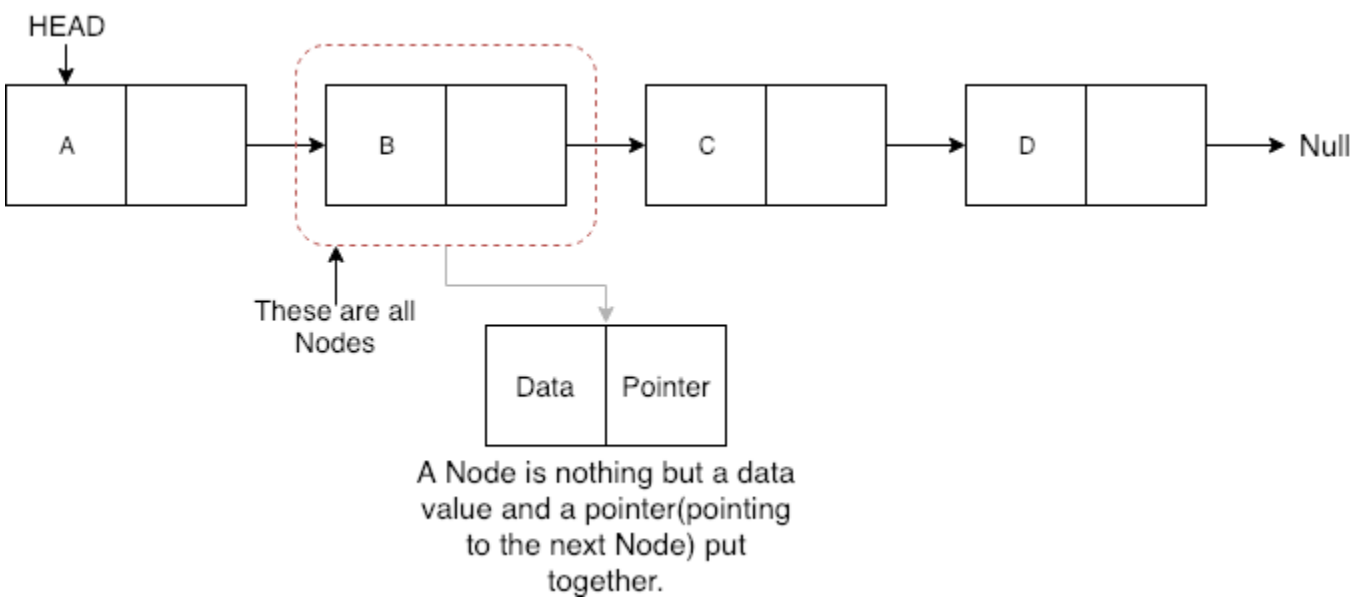
We will also be adding some other useful methods like:

- Checking whether Linked List is empty or not.
- Searching any data element in the Linked List
- Deleting a particular Node(data element) from the List

Before learning how we insert data and create a linked list, we must understand the components forming a linked list, and the main component is the **Node**.

What is a Node?

A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.



A Node is nothing but a data value and a pointer(pointing to the next Node) put together.

In the picture above we have a linked list, containing 4 nodes, each node has some data(A, B, C and D) and a pointer which stores the location of the next node.

You must be wondering **why we need to store the location of the next node**. Well, because the memory locations allocated to these nodes are not contiguous hence each node should know where the next node is stored.

As the node is a combination of multiple information, hence we will be defining a class for `Node` which will have a variable to store **data** and another variable to store the **pointer**. In C language, we create a structure using the `struct` keyword.

```
class Node
```

```
{
    public:
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;

    // default constructor
    Node()
    {
        data = 0;
        next = NULL;
    }

    // parameterised constructor
    Node(int x)
    {
        data = x;
        next = NULL;
    }
}
```

Copy

We can also make the `Node` class properties `data` and `next` as **private**, in that case we will need to add the getter and setter methods to access them(don't know what getter and setter methods are: Inline Functions in C++). You can add the getter and setter functions to the `Node` class like this:

```
class Node
{
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;
```

```
    // default constructor same as above


    // parameterised constructor same as above


    /* getters and setters */
    // get the value of data
    int getData()
    {
        return data;
    }


    // to set the value for data
    void setData(int x)
    {
        this.data = x;
    }
    // get the value of next pointer
    node* getNext()
    {
        return next;
    }
    // to set the value for pointer
    void setNext(node *n)
    {
        this.next = n;
    }
}
```

Copy

The `Node` class basically creates a node for the data to be included into the Linked List. Once the object for the class `Node` is created, we use various functions to fit in that node into the Linked List.

Linked List class

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all the methods like insertion, search, deletion etc. Also, the linked list class will have a pointer called `head` to store the location of the first node which will be added to the linked list.

```
class LinkedList
{
    public:
    node *head;
    //declaring the functions


    //function to add Node at front
    int addAtFront(node *n);
    //function to check whether Linked list is empty
    int isEmpty();
    //function to add Node at the End of list
    int addAtEnd(node *n);
    //function to search a value
    node* search(int k);
    //function to delete any Node
    node* deleteNode(int x);


    LinkedList()
    {
        head = NULL;
    }
}
```

Copy

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int LinkedList :: addAtFront(node *n) {
  int i = 0;
  //making the next of the new Node point to Head
  n->next = head;
  //making the new Node as Head
  head = n;
  i++;
  //returning the position where Node is added
  return i;
}
```

Copy

Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd(node *n) {
  //If list is empty
  if(head == NULL) {
    //making the new Node as Head
```

```
    head = n;

    //making the next pointe of the new Node as Null

    n->next = NULL;

  }

  else {

    //getting the last node

    node *n2 = getLastNode();

    n2->next = n;

  }

}



node* LinkedList :: getLastNode() {

  //creating a pointer pointing to Head

  node* ptr = head;

  //Iterating over the list till the node whose Next pointer points to
null

  //Return that node, because that will be the last node.

  while(ptr->next!=NULL) {

    //if Next is not Null, take the pointer one step forward

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```


node* LinkedList :: search(int x) {

  node *ptr = head;
```

```
  while(ptr != NULL && ptr->data != x) {

    //until we reach the end or we find a Node with data x, we keep
moving

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {

  //searching the Node with data x

  node *n = search(x);

  node *ptr = head;

  if(ptr == n) {

    ptr->next = n->next;

    return n;

  }

  else {

    while(ptr->next != n) {

      ptr = ptr->next;

    }

    ptr->next = n->next;

    return n;
```

```
  }

}
```

Copy

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is `NULL` or not.

```
int LinkedList :: isEmpty() {

  if(head == NULL) {

    return 1;

  }

  else { return 0; }

}
```

Copy

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.

If you are still figuring out, how to call all these methods, then below is how your `main()` method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

```
int main() {

  LinkedList L;

  //We will ask value from user, read the value and add the value to
our Node

  int x;

  cout << "Please enter an integer value : ";

  cin >> x;

  Node *n1;

  //Creating a new node with data as x

  n1 = new Node(x);

  //Adding the node to the list

  L.addAtFront(n1);
```

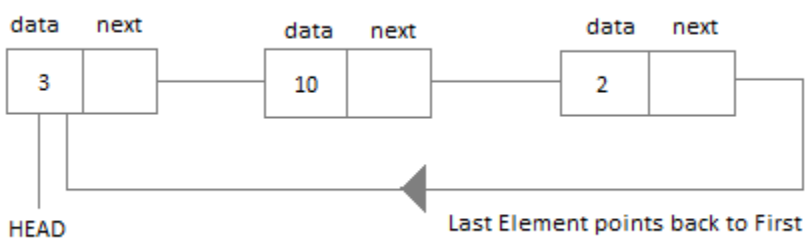
```
}
```

Copy

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;

  node() {
```

```
    data = 0;

    next = NULL;

  }



  node(int x) {

    data = x;

    next = NULL;

  }

}
```

Copy

Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

```
class CircularLinkedList {

  public:

  node *head;

  //declaring the functions



  //function to add Node at front

  int addAtFront(node *n);

//function to check whether Linked list is empty

int isEmpty();

//function to add Node at the End of list

int addAtEnd(node *n);

//function to search a value

node* search(int k);

//function to delete any Node

node* deleteNode(int x);
```

```
  CircularLinkedList() {

    head = NULL;

  }

}
```

Copy

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int CircularLinkedList :: addAtFront(node *n) {

  int i = 0;

  /* If the list is empty */

  if(head == NULL) {

    n->next = head;

    //making the new Node as Head

    head = n;

    i++;

  }

  else {

    n->next = head;

    //get the Last Node and make its next point to new Node

    Node* last = getLastNode();

    last->next = n;

    //also make the head point to the new first Node
```

```
    head = n;

    i++;

  }

  //returning the position where Node is added

  return i;

}
```

Copy

Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```
int CircularLinkedList :: addAtEnd(node *n) {

  //If list is empty

  if(head == NULL) {

    //making the new Node as Head

    head = n;

    //making the next pointer of the new Node as Null

    n->next = NULL;

  }

  else {

    //getting the last node

    node *last = getLastNode();

    last->next = n;

    //making the next pointer of new node point to head

    n->next = head;

  }

}
```

Copy

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* CircularLinkedList :: search(int x) {

  node *ptr = head;

  while(ptr != NULL && ptr->data != x) {

    //until we reach the end or we find a Node with data x, we keep
moving

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

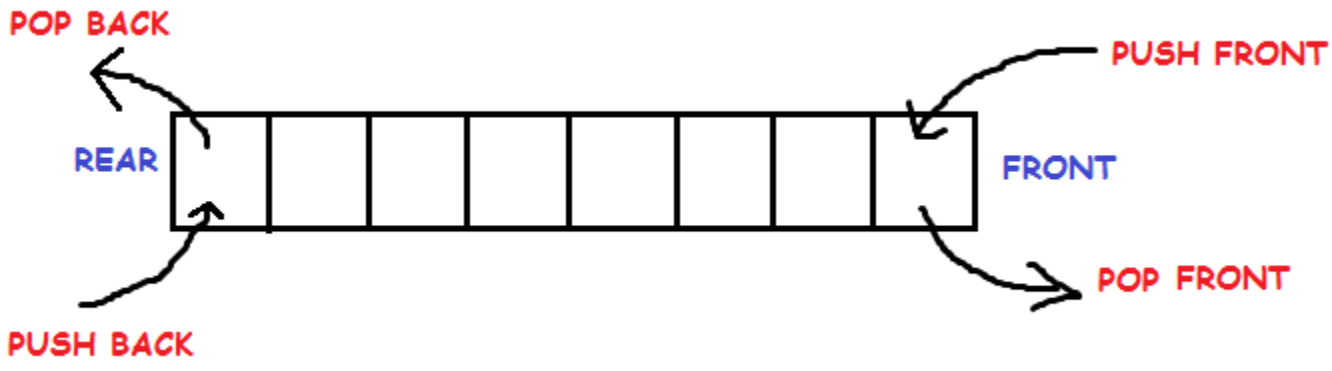
```
node* CircularLinkedList :: deleteNode(int x) {

  //searching the Node with data x

  node *n = search(x);

  node *ptr = head;

  if(ptr == NULL) {

    cout << "List is empty";
```

```
    return NULL;
  }
  else if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Copy

Double Ended Queue

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back.



Implementation of Double ended Queue

Here we will implement a double ended queue using a circular array. It will have the following methods:

- **push_back :** inserts element at back
- **push_front :** inserts element at front
- **pop_back :** removes last element
- **pop_front :** removes first element
- **get_back :** returns last element
- **get_front :** returns first element
- **empty :** returns true if queue is empty
- **full :** returns true if queue is full

```
// Maximum size of array or Dequeue



#define SIZE 5



class Dequeue

{

    //front and rear to store the head and tail pointers

    int  *arr;

    int front, rear;



    public :
```

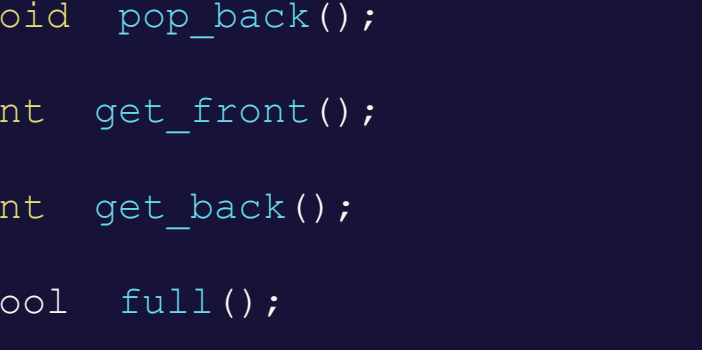
```
    Dequeue()

    {

     //Create the array

        arr = new int[SIZE];



        //Initialize front and rear with -1

        front = -1;

        rear = -1;

    }



    // Operations on Deque

    void  push_front(int );

    void  push_back(int );

    void  pop_front();

    void  pop_back();

    int  get_front();

    int  get_back();

    bool  full();

    bool  empty();

};
```

Copy

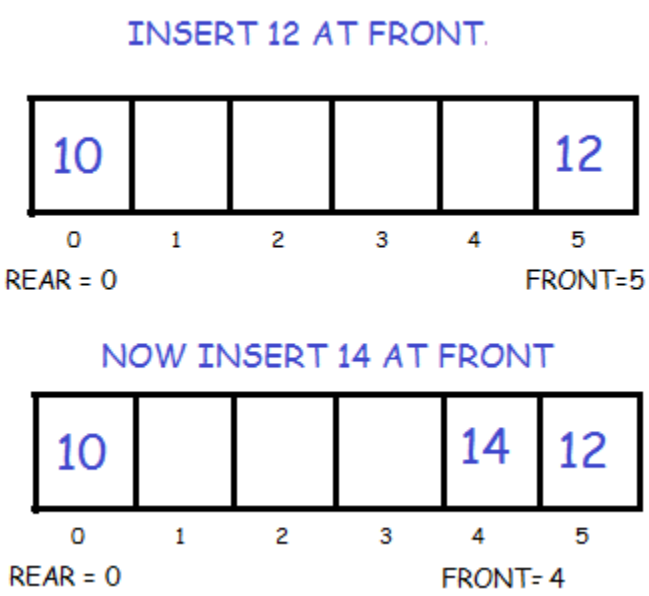
Insert Elements at Front

First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.



- Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.




```
void Dequeue :: push_front(int key)

{

    if(full())

    {

        cout << "OVERFLOW\n";

    }

    else

    {

     //If queue is empty

     if(front == -1)

          front = rear = 0;


   //If front points to the first position element

      else if(front == 0)

          front = SIZE-1;


      else

        --front;


      arr[front] = key;

  }
```

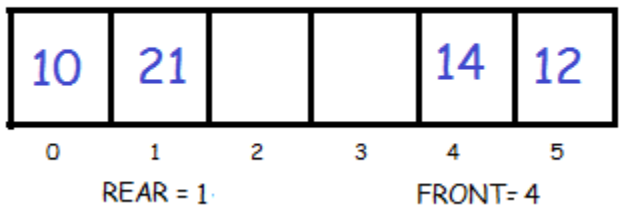
```
}
```

Copy

Insert Elements at back

Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.
- Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.




```
void Dequeue :: push_back(int key)

{

    if(full())

    {

        cout << "OVERFLOW\n";

    }

    else

    {

        //If queue is empty

    if(front == -1)

        front = rear = 0;


    //If rear points to the last element

    else if(rear == SIZE-1)

        rear = 0;


    else

      ++rear;
```

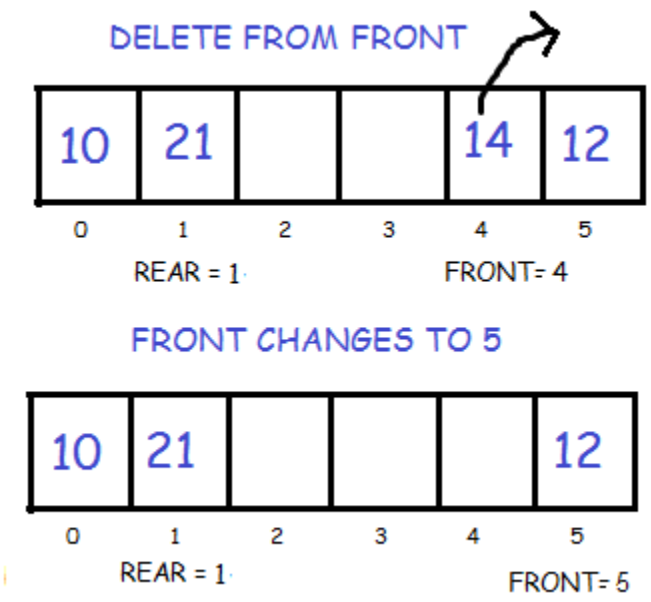
```
        arr[rear] = key;

    }

}
```

Copy

Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

- If only one element is present we once again make front and rear equal to -1.
- Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.




```
void Dequeue :: pop_front()

{

    if(empty())

    {

        cout << "UNDERFLOW\n";

    }

    else

    {

     //If only one element is present

        if(front == rear)

          front = rear = -1;
```

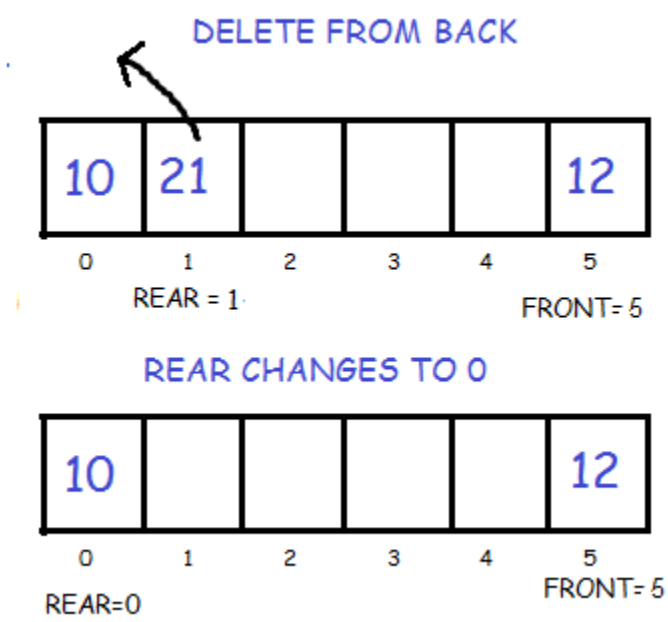
```
        //If front points to the last element
        else if(front == SIZE-1)
          front = 0;


        else
          ++front;
    }
}
```

Copy

Delete Last Element

Inorder to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :

- If only one element is present we make front and rear equal to -1.
- Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.




```
void Dequeue :: pop_back()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
```

```
    {
     //If only one element is present
        if(front == rear)
          front = rear = -1;

        //If rear points to the first position element
        else if(rear == 0)
          rear = SIZE-1;

        else
            --rear;
    }
}
```

Copy

Check if Queue is empty

It can be simply checked by looking where front points to. If front is still intialized with -1, the queue is empty.

```
bool Dequeue :: empty()
{
    if(front == -1)
     return true;
    else
     return false;
}
```

Copy

Check if Queue is full

Since we are using circular array, we check for following conditions as shown in code to check if queue is full.

```
bool Dequeue :: full()
```

```
{

    if((front == 0 && rear == SIZE-1)   ||

     (front == rear + 1))

        return true;

    else

        return false;

}
```

Copy

Return First Element

If the queue is not empty then we simply return the value stored in the position which front points.

```
int Dequeue :: get_front()

{

    if(empty())

    {cout << "f=" <<front << endl;

        cout << "UNDERFLOW\n";

        return -1;

    }

    else

    {

        return arr[front];

    }

}
```

Copy

Return Last Element

If the queue is not empty then we simply return the value stored in the position which rear points.

```
int Dequeue :: get_back()

{
```

```
    if(empty())
    {
        cout << "UNDERFLOW\n";
        return -1;
    }
    else
    {
        return arr[rear];
    }
}
```

Copy

Stack using Queue

A Stack is a **Last In First Out(LIFO)** structure, i.e, the element that is added last in the stack is taken out first. Our goal is to implement a Stack using Queue for which will be using two queues and design them in such a way that **pop** operation is same as dequeue but the **push** operation will be a little complex and more expensive too.

Implementation of Stack using Queue

Assuming we already have a class implemented for Queue, we first design the class for Stack. It will have the methods `push()` and `pop()` and two queues.

```
class Stack

{

    public:

    // two queue

    Queue Q1, Q2;

    // push method to add data element

    void push(int);

    // pop method to remove data element

    void pop();

};
```

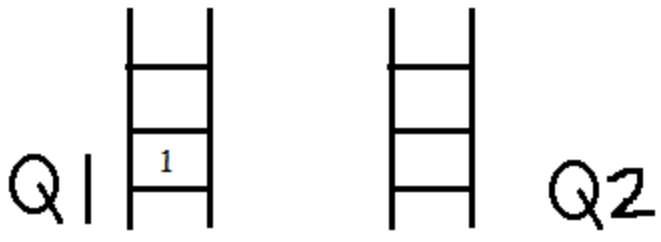
Copy

Inserting Data in Stack

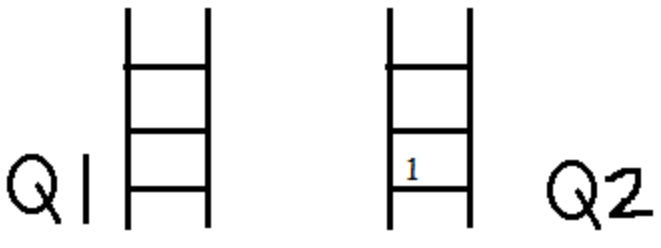
Since we are using Queue which is **First In First Out(FIFO)** structure , i.e, the element which is added first is taken out first, so we will implement the **push** operation in such a way that whenever there is a **pop** operation, the stack always pops out the last element added.

In order to do so, we will require two queues, `Q1` and `Q2`. Whenever the **push** operation is invoked, we will enqueue(move in this case) all the elements of `Q1` to `Q2` and then enqueue the new element to `Q1`. After this we will enqueue(move in this case) all the elements from `Q2` back to `Q1`.

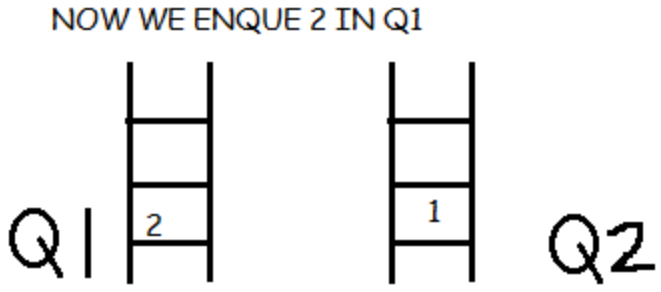
INITIALLY BOTH THE QUEUES ARE EMPTY.
WHEN WE GET A QUERY PUSH(1), SINCE Q1 IS EMPTY,
WE DIRECTLY INSERT 1 TO Q1.



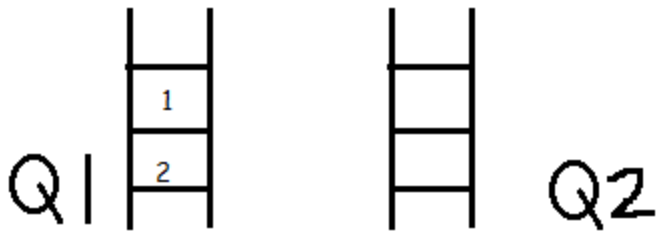
NOW IF WE GET A QUERY PUSH(2), WE WILL
FIRST ENQUE ALL ELEMENTS FROM Q1 TO Q2



NOW WE ENQUE 2 IN Q1



FINALLY WE DEQUE ALL THE ELEMENTS
FROM Q2 AND ENQUE THEM IN Q1



HENCE WE SEE THAT THE LAST ELEMENT ADDED , I.E., 2
IS IN THE FRONT OF Q1. SO IF WE GET A POP QUERY,
JUST USING THE DEQUE OPERATION IN Q1 WILL POP
OUT THE LAST ELEMENT ADDED IN THE STACK.

So let's implement this in our code,

```
void Stack :: push(int x)

{

     // move all elements in Q1 to Q2

     while(!Q1.isEmpty())

     {

          int temp = Q1.deque();

          Q2.enque(temp);

     }



     // add the element which is pushed into Stack

     Q1.enque(x);
```

```
        // move back all elements back to Q1 from Q2
        while(!Q2.isEmpty())
        {
              int temp = Q2.deque();
              Q1.enque(temp);
        }
}
```

Copy

It must be clear to you now, why we are using two queues. Actually the queue `Q2` is just for the purpose of keeping the data temporarily while operations are executed.

In this way we can ensure that whenever the **pop** operation is invoked, the stack always pops out the last element added in `Q1` queue.

Removing Data from Stack

Like we have discussed above, we just need to use the dequeue operation on our queue `Q1`. This will give us the last element added in Stack.

```
int Stack :: pop()
{
      return Q1.deque();
}
```

Copy

Time Complexity Analysis

When we implement Stack using a Queue the **push** operation becomes expensive.

- Push operation: O(n)
- Pop operation: O(1)

Conclusion

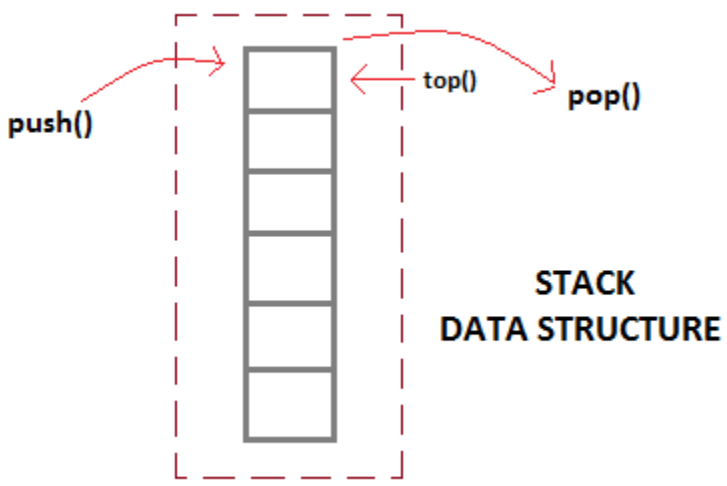
When we say "implementing Stack using Queue", we mean how we can make a Queue behave like a Stack, after all they are all logical entities. So for any data structure to act as a Stack, it should have `push()` method to add data on top and `pop()` method to remove data from top. Which

is exactly what we did and hence accomplished to make a Queue(in this case two Queues) behave as a Stack.

Stack using Linked List

Stack as we know is a **Last In First Out(LIFO)** data structure. It has the following operations :

- **push:** push an element into the stack
- **pop:** remove the last element added
- **top:** returns the element at top of stack



Implementation of Stack using Linked List

Stacks can be easily implemented using a linked list. Stack is a data structure to which a data can be added using the `push()` method and data can be removed from it using the `pop()` method. With Linked list, the **push** operation can be replaced by the `addAtFront()` method of linked list and **pop** operation can be replaced by a function which deletes the front node of the linked list.

In this way our Linked list will virtually become a Stack with `push()` and `pop()` methods.

First we create a class **node**. This is our Linked list node class which will have **data** in it and a **node pointer** to store the address of the next node element.

```
class node

{

     int data;

     node *next;

};
```

Copy

Then we define our stack class,

```
class Stack

{

     node *front;  // points to the head of list

     public:

     Stack()
```

```
	{
		front = NULL;
	}
	// push method to add data element
	void push(int);
	// pop method to remove data element
	void pop();
	// top method to return top data element
	int top();
};
```

Copy

Inserting Data in Stack (Linked List)

In order to insert an element into the stack, we will create a node and place it in front of the list.

```
void Stack :: push(int d)
{
	// creating a new node
	node *temp;
	temp = new node();
	// setting data to it
	temp->data = d;

	// add the node in front of list
	if(front == NULL)
	{
		temp->next = NULL;
	}
	else
	{
		temp->next = front;
```

```
	}

	front = temp;

}
```

Copy

Now whenever we will call the `push()` function a new node will get added to our list in the front, which is exactly how a stack behaves.

Removing Element from Stack (Linked List)

In order to do this, we will simply delete the first node, and make the second node, the head of the list.

```
void Stack :: pop()

{

	// if empty

	if(front == NULL)

		cout << "UNDERFLOW\n";


	// delete the first element

	else

	{

		node *temp = front;

		front = front->next;

		delete(temp);

	}

}
```

Copy

Return Top of Stack (Linked List)

In this, we simply return the data stored in the head of the list.

```
int Stack :: top()

{

	return front->data;
```

```
}
```

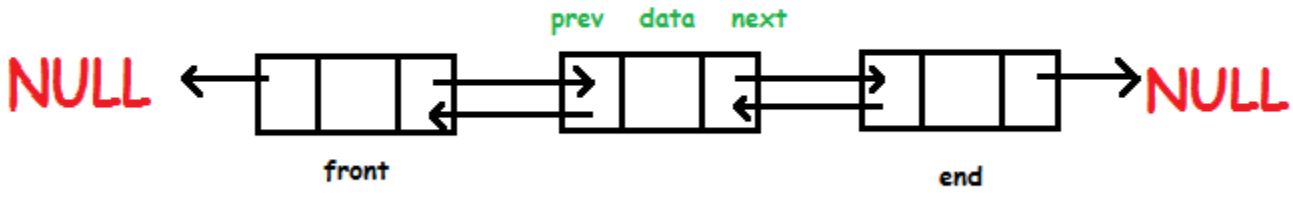
Copy

Conclusion

When we say "implementing Stack using Linked List", we mean how we can make a Linked List behave like a Stack, after all they are all logical entities. So for any data structure to act as a Stack, it should have `push()` method to add data on top and `pop()` method to remove data from top. Which is exactly what we did and hence accomplished to make a Linked List behave as a Stack.

Doubly Linked List

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.



The two links help us to traverse the list in both backward and forward direction. But storing an extra link requires some extra space.

Implementation of Doubly Linked List

First we define the node.

```
struct node

{

     int data;       // Data

     node *prev;     // A reference to the previous node

     node *next;     // A reference to the next node

};
```

Copy

Now we define our class Doubly Linked List. It has the following methods:

- **add_front:** Adds a new node in the beginning of list
- **add_after:** Adds a new node after another node
- **add_before:** Adds a new node before another node
- **add_end:** Adds a new node in the end of list
- **delete:** Removes the node
- **forward_traverse:** Traverse the list in forward direction
- **backward_traverse:** Traverse the list in backward direction

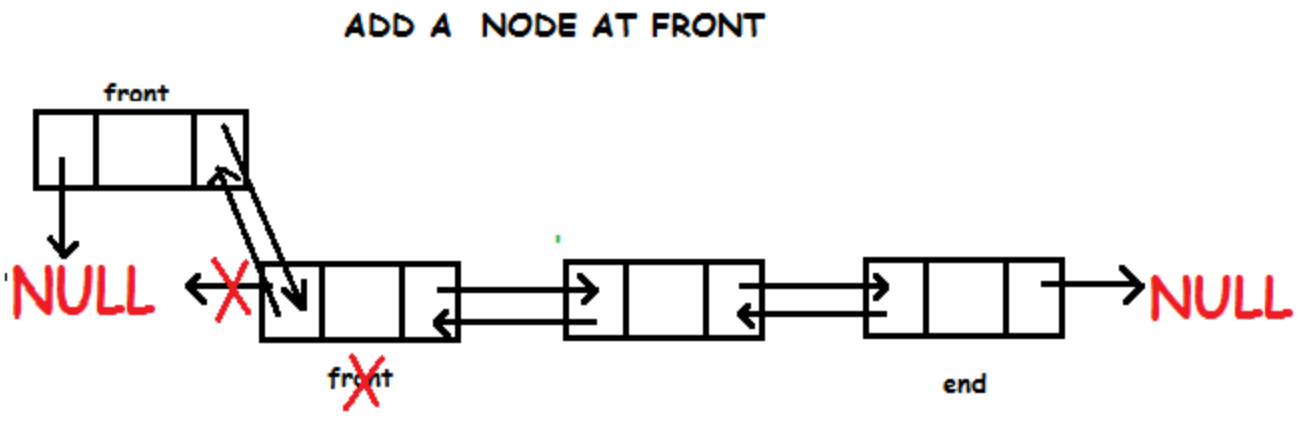
```
class Doubly_Linked_List

{

     node *front;    // points to first node of list
```

```
    node *end;      // points to first las of list
    public:
    Doubly_Linked_List()
    {
        front = NULL;
        end = NULL;
    }
    void add_front(int );
    void add_after(node* , int );
    void add_before(node* , int );
    void add_end(int );
    void delete_node(node*);
    void forward_traverse();
    void backward_traverse();
};
```

Copy

Insert Data in the beginning

1. The **prev** pointer of first node will always be NULL and **next** will point to **front**.
2. If the node is inserted is the first node of the list then we make **front** and **end** point to this node.
3. Else we only make **front** point to this node.



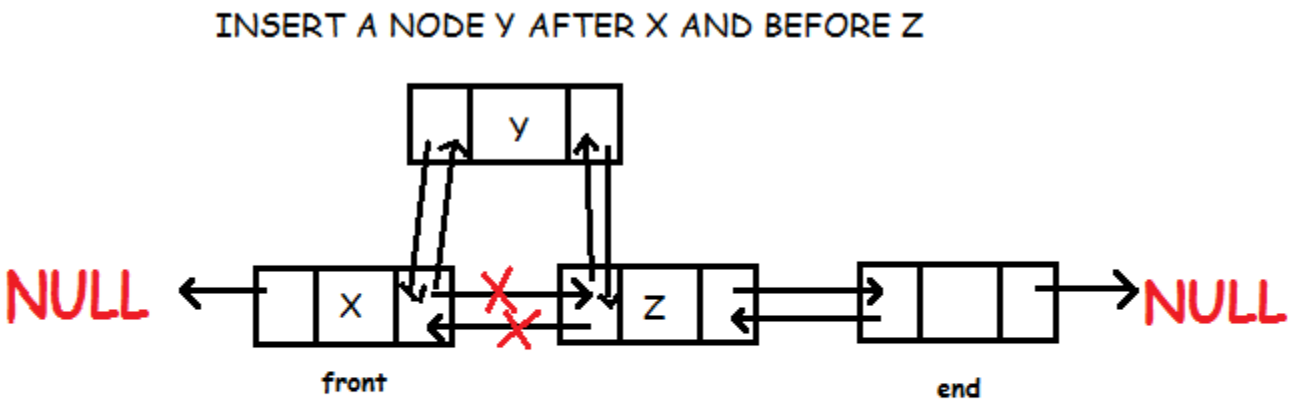
1. WE MAKE THE CURRENT FRONT NODE'S PREV POINT TO NEW NODE.
2. MAKE NEXT OF NEW NODE POINT TO CURRENT FRONT AND PREV OF NEW NODE POINT TO NULL.
3. WE CHANGE FRONT NODE TO THE NEW NODE.

```
void Doubly_Linked_List :: add_front(int d)

{

     // Creating new node

     node *temp;

     temp = new node();

     temp->data = d;

     temp->prev = NULL;

     temp->next = front;



     // List is empty

     if(front == NULL)

          end = temp;


     else

          front->prev = temp;


     front = temp;

}
```

Copy

Insert Data before a Node

Let’s say we are inserting node X before Y. Then X’s next pointer will point to Y and X’s prev pointer will point the node Y’s prev pointer is pointing. And Y’s prev pointer will now point to X. We need to make sure that if Y is the first node of list then after adding X we make front point to X.




```
void Doubly_Linked_List :: add_before(node *n, int d)

{

     node *temp;

     temp = new node();

     temp->data = d;

     temp->next = n;

     temp->prev = n->prev;

     n->prev = temp;



     //if node is to be inserted before first node

     if(n->prev == NULL)

          front = temp;

}
```

Copy

Insert Data after a Node

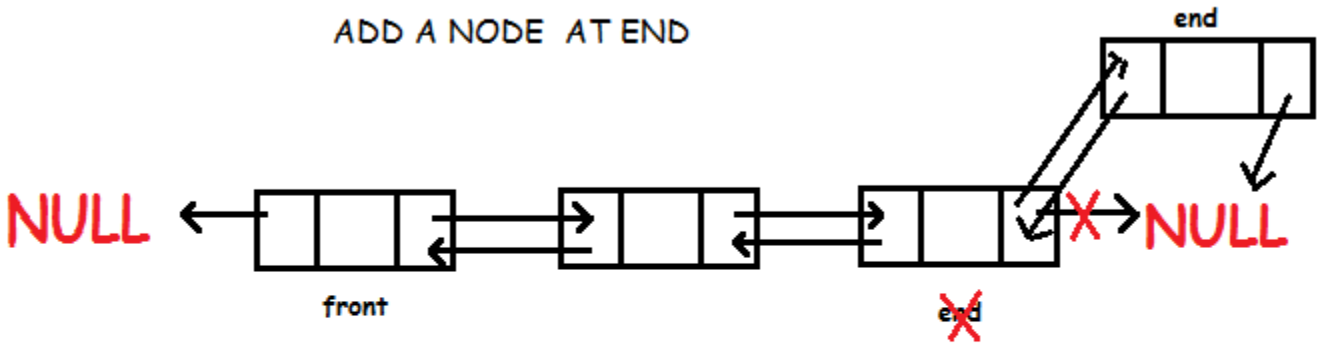
Let’s say we are inserting node Y after X. Then Y’s prev pointer will point to X and Y’s next pointer will point the node X’s next pointer is pointing. And X’s next pointer will now point to Y. We need to make sure that if X is the last node of list then after adding Y we make end point to Y.

```
void Doubly_Linked_List :: add_after(node *n, int d)

{

     node *temp;

     temp = new node();

     temp->data = d;

     temp->prev = n;

     temp->next = n->next;

     n->next = temp;



     //if node is to be inserted after last node

     if(n->next == NULL)

          end = temp;

}
```

Copy

Insert Data in the end

1. The **next** pointer of last node will always be NULL and **prev** will point to end.
2. If the node is inserted is the first node of the list then we make front and end point to this node.
3. Else we only make end point to this node.



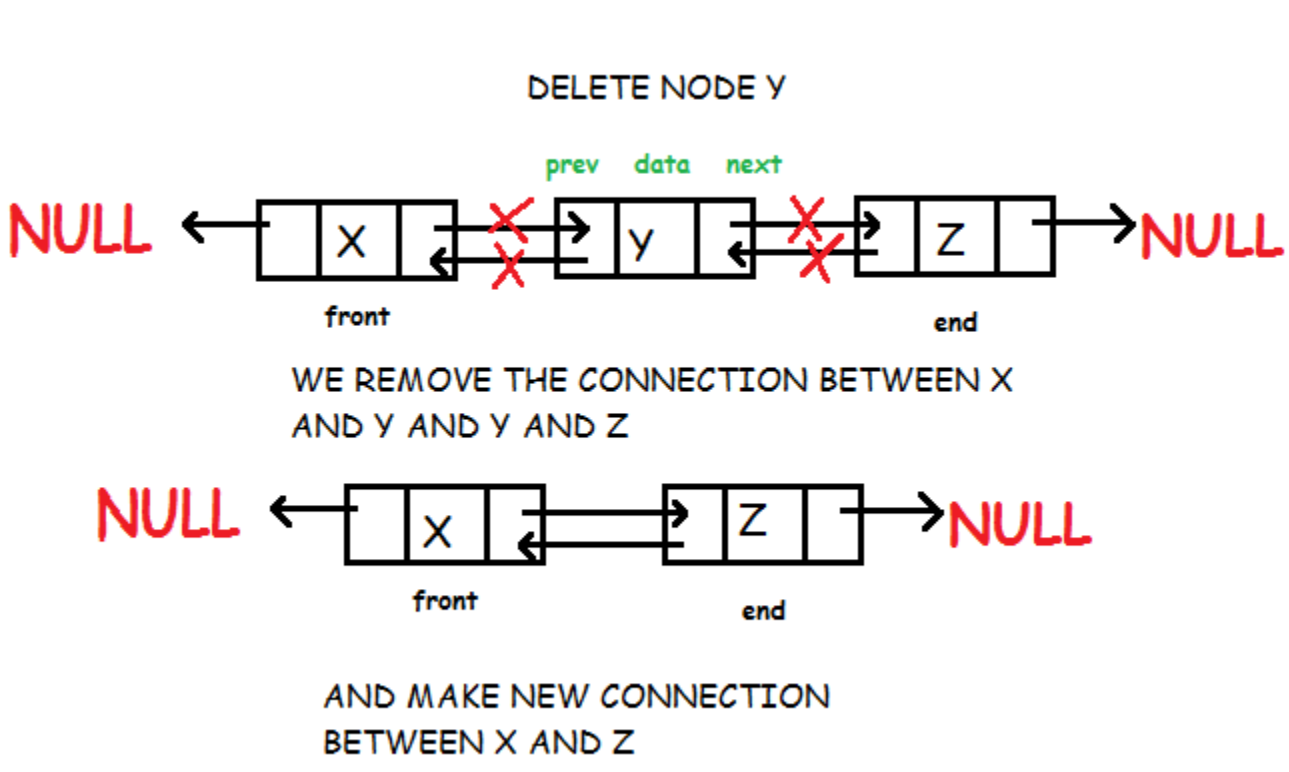

```
void Doubly_Linked_List :: add_end(int d)

{
```

```
    // create new node

    node *temp;

    temp = new node();

    temp->data = d;

    temp->prev = end;

    temp->next = NULL;


    // if list is empty

    if(end == NULL)

         front = temp;

    else

         end->next = temp;

    end = temp;

}
```

Copy

Remove a Node

Removal of a node is quite easy in Doubly linked list but requires special handling if the node to be deleted is first or last element of the list. Unlike singly linked list where we require the previous node, here only the node to be deleted is needed. We simply make the next of the previous node point to next of current node (node to be deleted) and prev of next node point to prev of current node. Look code for more details.




```
void Doubly_Linked_List :: delete_node(node *n)
```

```
{
     // if node to be deleted is first node of list
     if(n->prev == NULL)
     {
          front = n->next; //the next node will be front of list
          front->prev = NULL;
     }
     // if node to be deleted is last node of list
     else if(n->next == NULL)
     {
          end = n->prev;   // the previous node will be last of list
          end->next = NULL;
     }
     else
     {
          //previous node's next will point to current node's next
          n->prev->next = n->next;
          //next node's prev will point to current node's prev
          n->next->prev = n->prev;
     }
     //delete node
          delete(n);
}
```

Copy

Forward Traversal

Start with the front node and visit all the nodes untill the node becomes NULL.

```
void Doubly_Linked_List :: forward_traverse()
{
     node *trav;
```

```
      trav = front;

      while(trav != NULL)

      {

            cout<<trav->data<<endl;

            trav = trav->next;

      }

}
```

Copy

Backward Traversal

Start with the end node and visit all the nodes until the node becomes NULL.

```
void Doubly_Linked_List :: backward_traverse()

{

      node *trav;

      trav = end;

      while(trav != NULL)

      {

            cout<<trav->data<<endl;

            trav = trav->prev;

      }

}
```

Copy

Now that we have learned about the Doubly Linked List, you can also check out the other types of Linked list as well:

- Linear Linked List
- Circular Linked List

Introduction To Binary Trees

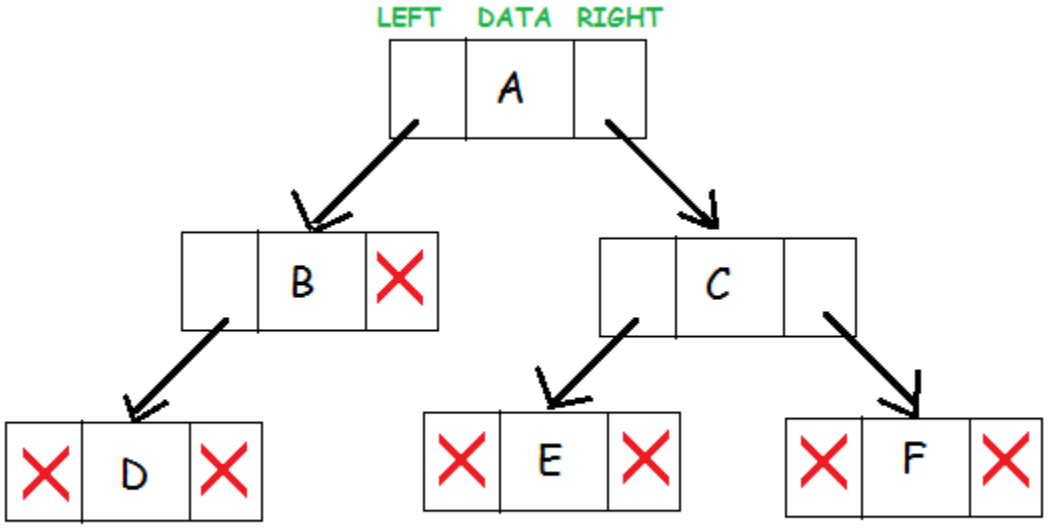
A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

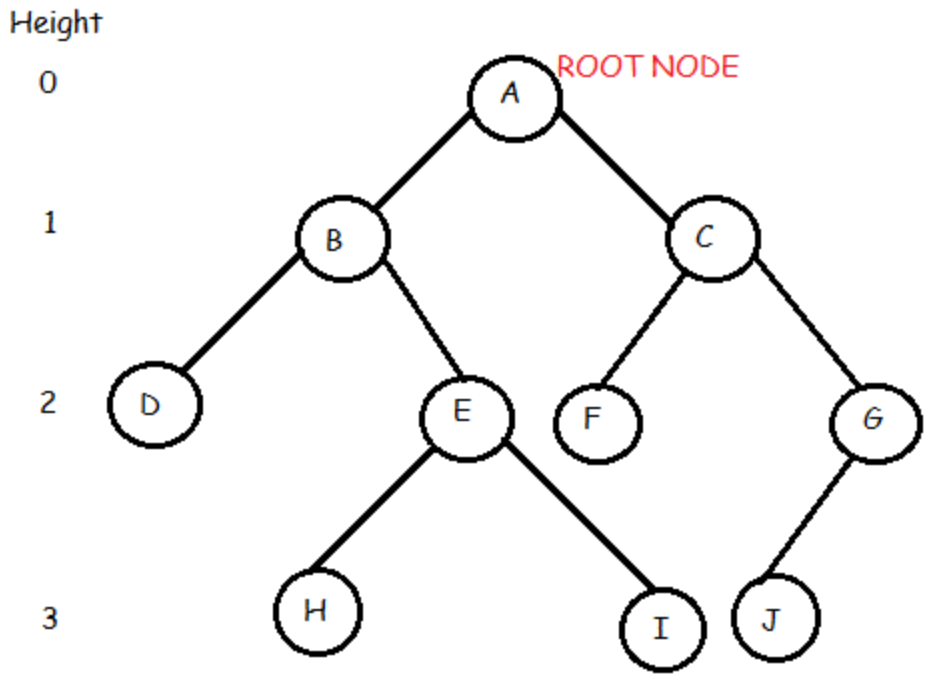
The topmost node in the tree is called the root. An empty tree is represented by **NULL** pointer.

A representation of binary tree is shown:



Binary Tree: Common Terminologies

- **Root:** Topmost node in a tree.
- **Parent:** Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent.
- **Child:** A node directly connected to another node when moving away from the root.
- **Leaf/External node:** Node with no children.
- **Internal node:** Node with atleast one children.
- **Depth of a node:** Number of edges from root to the node.
- **Height of a node:** Number of edges from the node to the deepest leaf. Height of the tree is the height of the root.



In the above binary tree we see that root node is **A**. The tree has 10 nodes with 5 internal nodes, i.e, **A,B,C,E,G** and 5 external nodes, i.e, **D,F,H,I,J**. The height of the tree is 3. **B** is the parent of **D** and **E** while **D** and **E** are children of **B**.

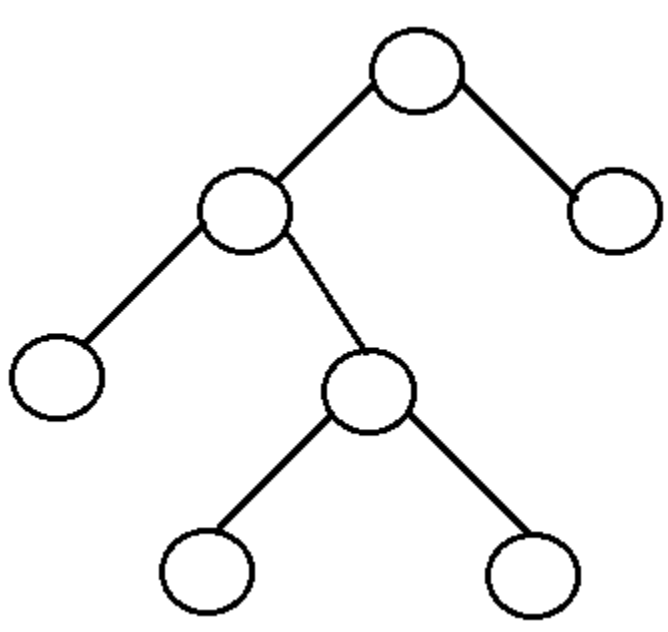
Advantages of Trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minumum effort.

Types of Binary Trees (Based on Structure)

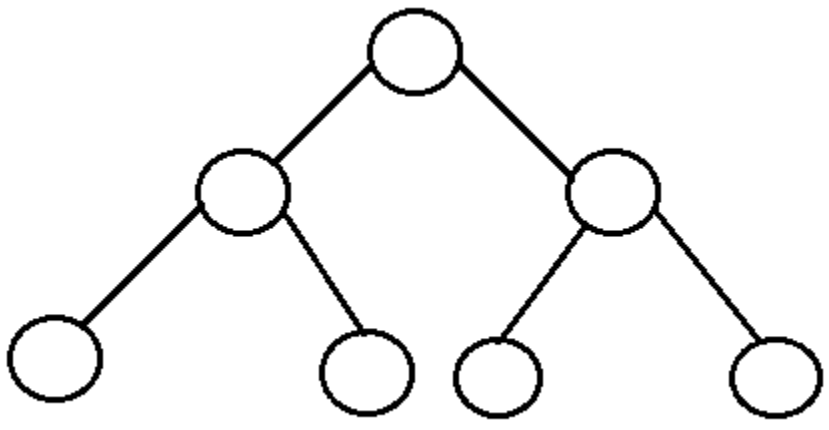
- **Rooted binary tree:** It has a root node and every node has atmost two children.
- **Full binary tree:** It is a tree in which every node in the tree has either 0 or 2 children.



  - The number of nodes, *n*, in a full binary tree is atleast n = 2h – 1, and atmost $n = 2^{h+1} - 1$, where *h* is the height of the tree.

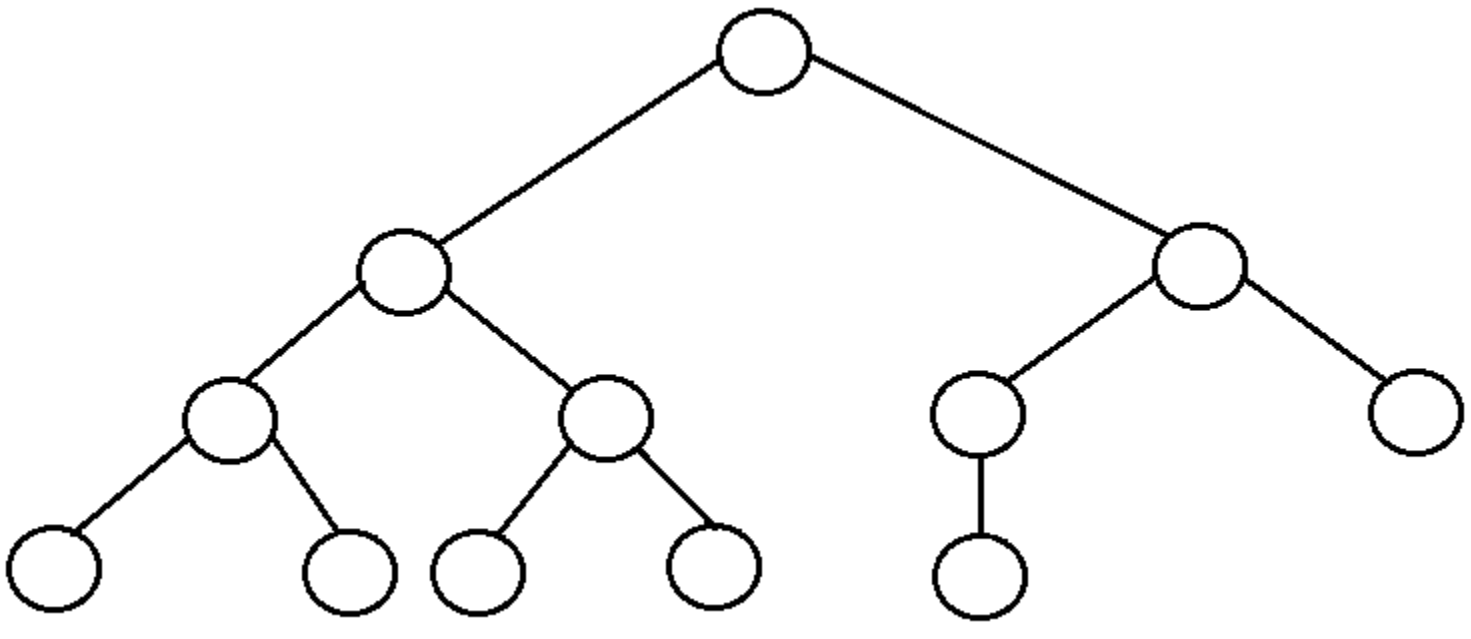
    - The number of leaf nodes *l*, in a full binary tree is number, *L* of internal nodes + 1, i.e, *l = L+1*.

- **Perfect binary tree:** It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



    - A perfect binary tree with *l* leaves has *n = 2l-1* nodes.
    - In perfect full binary tree, *l = 2h* and *n = 2h+1 - 1* where, *n* is number of nodes, *h* is height of tree and *l* is number of leaf nodes

- **Complete binary tree:** It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

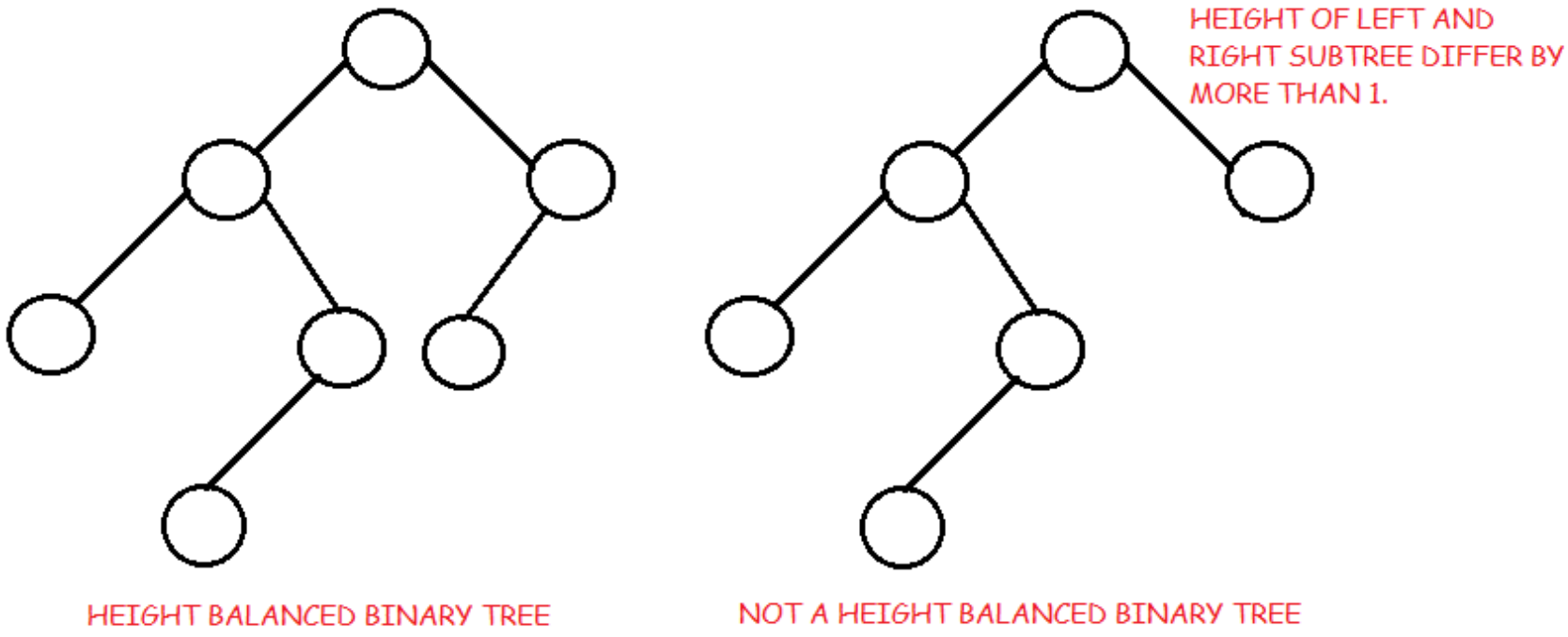


    - The number of internal nodes in a complete binary tree of n nodes is *floor(n/2)*.

- **Balanced binary tree:** A binary tree is height balanced if it satisfies the following constraints:

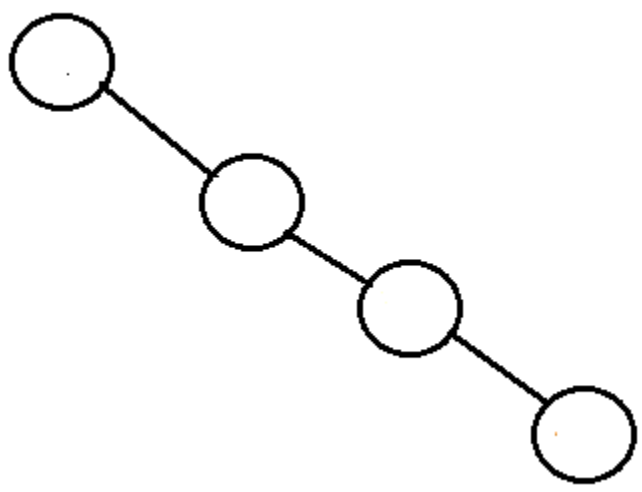
    1. The left and right subtrees' heights differ by at most one, AND
    2. The left subtree is balanced, AND
    3. The right subtree is balanced

    An empty tree is height balanced.



- The height of a balanced binary tree is O(*Log n*) where *n* is number of nodes.

- **Degenarate tree:** It is a tree is where each parent node has only one child node. It behaves like a linked list.



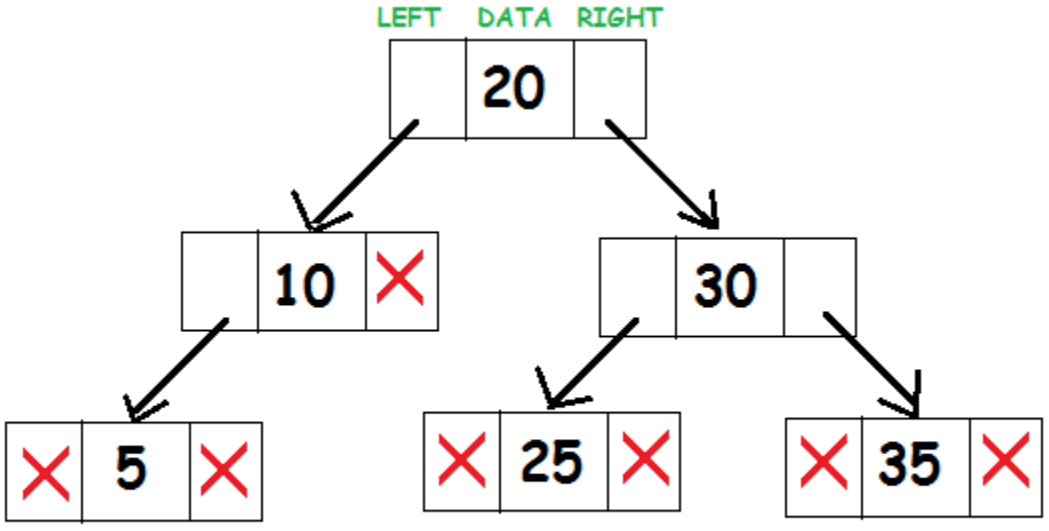
Binary Search Tree

A binary search tree is a useful data structure for fast addition and removal of data.

It is composed of nodes, which stores data and also links to upto two other child nodes. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure.

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right subtree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.

A representation of binary search tree looks like the following:



Consider the root node 20. All elements to the left of subtree(10, 5) are less than 20 and all elements to the right of subtree(25, 30, 35) are greater than 20.

Implementation of BST

First, define a struct as `tree_node`. It will store the data and pointers to left and right subtree.

```
struct tree_node

{

     int data;

     tree_node *left, *right;

};
```

Copy

The node itself is very similar to the node in a linked list. A basic knowledge of the code for a linked list will be very helpful in understanding the techniques of binary trees.

It is most logical to create a binary search tree class to encapsulate the workings of the tree into a single area, and also making it reusable. The class will contain functions to insert data into the tree, search if the data is present and methods for traversing the tree.

```
class BST

{

     tree_node *root;

     void insert(tree_node* , int );

     bool search(int , tree_node* );

     void inorder(tree_node* );

     void preorder(tree_node* );

     void postorder(tree_node* );



     public:

     BST()

     {

          root = NULL;

     }

     void insert(int );

     bool search(int key);

    void inorder();

    void preorder();

    void postorder();

};
```

Copy

It is necessary to initialize root to NULL for the later functions to be able to recognize that it does not exist.

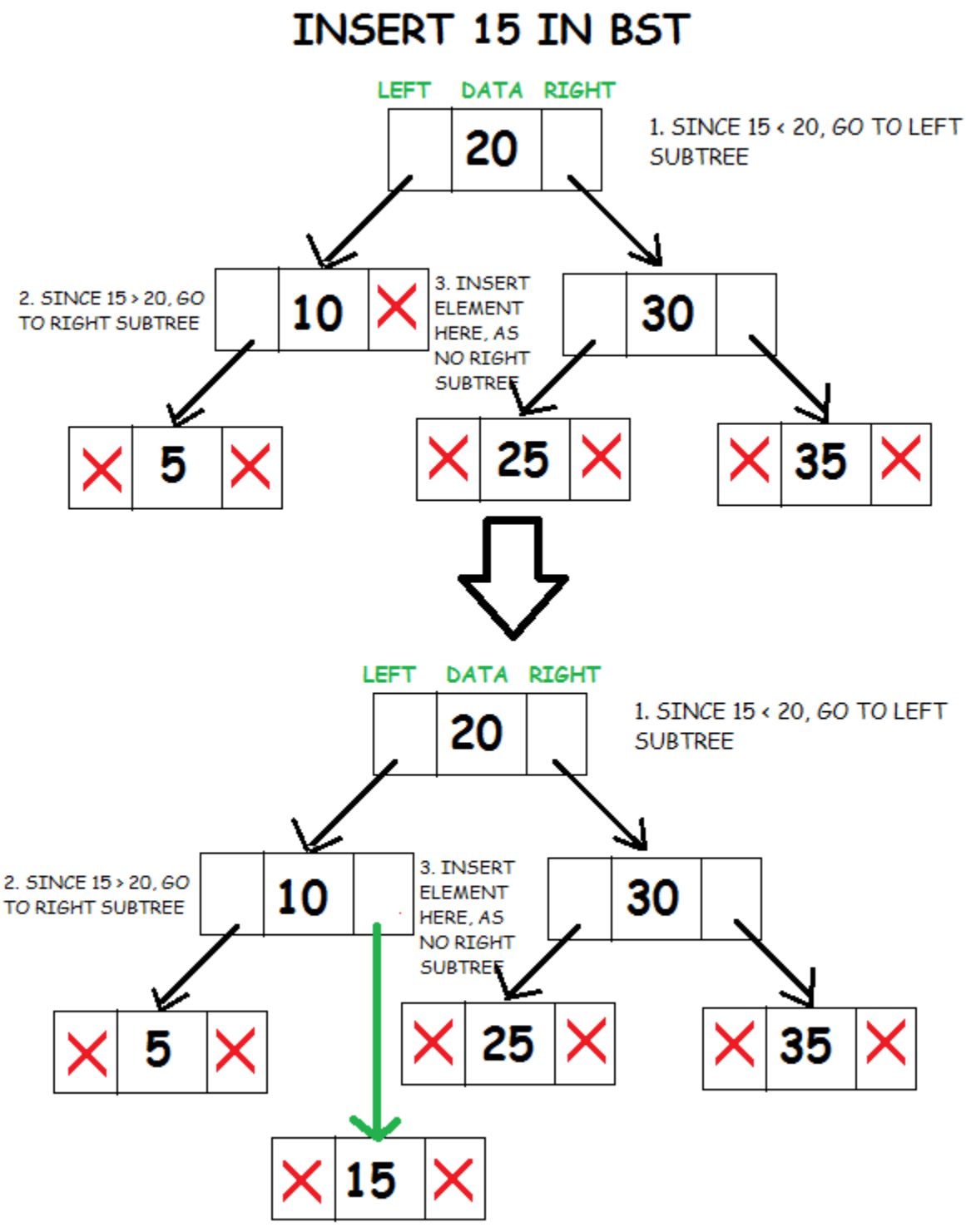
All the public members of the class are designed to allow the user of the class to use the class without dealing with the underlying design. The functions which will be called recursively are the ones which are private, allowing them to travel down the tree.

Insertion in a BST

To insert data into a binary tree involves a function searching for an unused node in the proper position in the tree in which to insert the key value. The insert function is generally a recursive function that continues moving down the levels of a binary tree until there is an unused leaf in a position which follows the following rules of placing nodes.

- Compare data of the root node and element to be inserted.

- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- Insert element as left child of current root.
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**.
- Else, insert element as right child of current root.




```
void BST :: insert(tree_node *node, int d)

{

     // element to be inserted is lesser than node's data

     if(d < node->data)

     {

          // if left subtree is present

          if(node->left != NULL)

               insert(node->left, d);
```

```
            // create new node
            else
            {
                  node->left = new tree_node;
                  node->left->data = d;
                  node->left->left = NULL;
                  node->left->right = NULL;
            }
      }

      // element to be inserted is greater than node's data
      else if(d >= node->data)
      {
            // if left subtree is present
            if(node->right != NULL)
                  insert(node->right, d);

            // create new node
            else
            {
                  node->right = new tree_node;
                  node->right->data = d;
                  node->right->left = NULL;
                  node->right->right = NULL;
            }
      }

}
```

Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to insert an element and also takes care of the case when root node is NULL.

```
void BST::insert(int d)

{

     if(root!=NULL)

                insert(root, d);

     else

    {

        root = new tree_node;

        root->data = d;

        root->left = NULL;

        root->right = NULL;

    }

}
```

Copy

Searching in a BST

The search function works in a similar fashion as insert. It will check if the key value of the current node is the value to be searched. If not, it should check to see if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node, or if it is greater than the value of the node, it should be recursively called on the right child node.

- Compare data of the root node and the value to be searched.
- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**. Else,
- If the value to be searched is equal to the data of root node, return true.
- Else, return false.

```
bool BST::search(int d, tree_node *node)

{

     bool ans = false;



     // node is not present
```

```
     if(node == NULL)
          return false;


     // Node’s data is equal to value searched
    if(d == node->data)
          return true;


     // Node’s data is greater than value searched
    else if(d < node->data)
          ans = search(d, node->left);


      // Node’s data is lesser than value searched
    else
          ans = search(d, node->right);


    return ans;
}
```

Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to search an element and also takes care of the case when root node is NULL.

```
bool BST::search(int d)
{
     if(root ==  NULL)
          return false;
     else
          return  search(d, root);
}
```

Copy

Traversing in a BST

There are mainly three types of tree traversals:

1. Pre-order Traversal:
In this technique, we do the following :

- Process data of root node.
- First, traverse left subtree completely.
- Then, traverse right subtree.

```
void BST :: preorder(tree_node *node)

{

     if(node !=  NULL)

    {

           cout<<node->data<<endl;

           preorder(node->left);

           preorder(node->right);

    }

}
```

Copy

2. Post-order Traversal
In this traversal technique we do the following:

- First traverse left subtree completely.
- Then, traverse right subtree completely.
- Then, process data of node.

```
void BST :: postorder(tree_node *node)

{

     if(node !=  NULL)

     {

        postorder(node->left);

        postorder(node->right);

         cout<<node->data<<endl;

     }

}
```

Copy

3. In-order Traversal

In in-order traversal, we do the following:

- First process left subtree.
- Then, process current root node.
- Then, process right subtree.

```
void BST :: inorder(tree_node *node)

{

     if(node !=  NULL)

     {

        inorder(node->left);

         cout<<node->data<<endl;

        inorder(node->right);

     }

}
```

Copy

The in-order traversal of a binary search tree gives a sorted ordering of the data elements that are present in the binary search tree. This is an important property of a binary search tree.

Since the root node is a private member, we also write public member functions which is available to non-members of the class. It calls the private recursive function to traverse the tree and also takes care of the case when root node is NULL.

```
void BST :: preorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";

     else

          preorder(root);

}


void BST :: postorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";
```

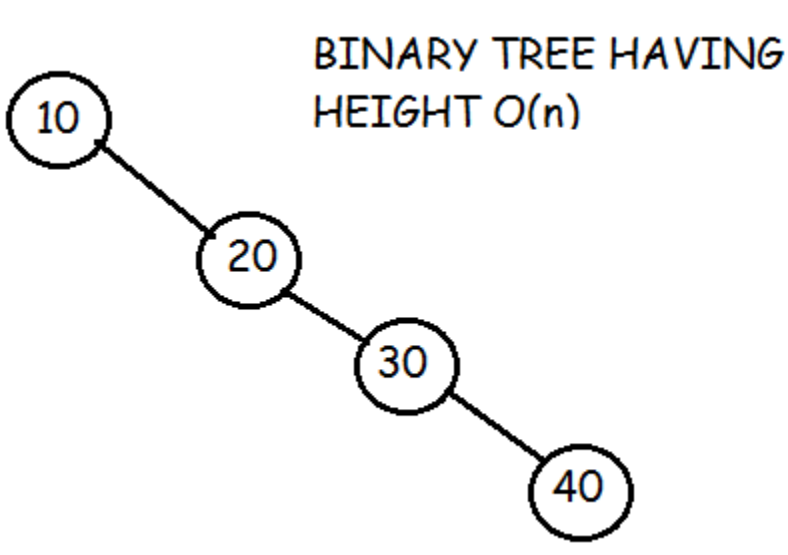
```
     else
         postorder(root);
}


void BST :: inorder()
{
     if(root == NULL)
          cout<<"TREE IS EMPTY\n";
     else
          inorder(root);
}
```

Copy

Complexity Analysis

ALGORITHM	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Insert	O(log n)	O(n)
Search	O(log n)	O(n)
Traverse	O(n)	O(n)

The time complexity of search and insert rely on the height of the tree. On average, binary search trees with n nodes have **O(log n)** height. However in the worst case the tree can have a height of **O(n)** when the unbalanced tree resembles a linked list. For example in this case :



Traversal requires **O(n)** time, since every node must be visited.

Introduction to Searching Algorithms

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it, that whenever a user asks for some data, computer has to search it's memory to look for the data and make it available to the user. And the computer has it's own techniques to search through it's memory fast, which you can learn more about in our Operating System tutorial series.

What if you have to write a program to search a given number in an array? How will you do it?

Well, to search an element in a given array, there are two popular algorithms available:

1. Linear Search
2. Binary Search

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return `-1`.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

We will implement the Linear Search algorithm in the next tutorial.

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence

on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. We will discuss this in details in the Binary Search tutorial.
3. It has a simple implementation.

Linear Search Algorithm

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

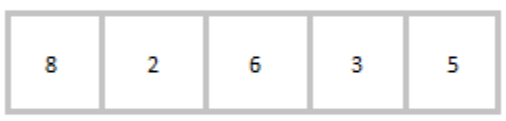
As we learned in the previous tutorial that the time complexity of Linear search algorithm is **O(n)**, we will analyse the same and see why it is **O(n)** after implementing it.

Implementing Linear Search

Following are the steps of implementation that we will be following:

1. Traverse the array using a `for` loop.
2. In every iteration, compare the `target` value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return `-1`.

To search the number **5** in the array given below, linear search will go step by step in a sequential order starting from the first element in the given array.




```
/*
    below we have implemented a simple function
    for linear search in C


    - values[] => array with all the values
    - target => value to be found
    - n => total number of elements in the array
*/
int linearSearch(int values[], int target, int n)
{
    for(int i = 0; i < n; i++)
    {
```

```
        if (values[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Copy

Some Examples with Inputs

```
Input: values[] = {5, 34, 65, 12, 77, 35}
       target = 77

Output: 4

Input: values[] = {101, 392, 1, 54, 32, 22, 90, 93}
       target = 200

Output: -1 (not found)
```

Final Thoughts

We know you like Linear search because it is so damn simple to implement, but it is not used practically because binary search is a lot faster than linear search. So let's head to the next tutorial where we will learn more about binary search.

Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return `-1`

```
/*
    function for carrying out binary search on given array
    - values[] => given sorted array
    - len => length of the array
    - target => value to be searched
*/
int binarySearch(int values[], int len, int target)
{
    int max = (len - 1);
    int min = 0;

    int guess;  // this will hold the index of middle elements
    int step = 0;  // to find out in how many steps we completed the search

    while(max >= min)
```

```
    {
        guess = (max + min) / 2;
        // we made the first guess, incrementing step by 1
        step++;

        if(values[guess] ==  target)
        {
            printf("Number of steps required for search: %d \n", step);
            return guess;
        }
        else if(values[guess] >  target)
        {
            // target would be in the left half
            max = (guess - 1);
        }
        else
        {
            // target would be in the right half
            min = (guess + 1);
        }
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int values[] = {13, 21, 54, 81, 90};
```

```
    int n = sizeof(values) / sizeof(values[0]);

    int target = 81;

    int result = binarySearch(values, n, target);

    if(result == -1)

    {

        printf("Element is not present in the given array.");

    }

    else

    {

        printf("Element is present at index: %d", result);

    }

    return 0;

}
```

Copy

We hope the above code is clear, if you have any confusion, post your question in our Q & A Forum.

Now let's try to understand, why is the time complexity of binary search **O(log n)** and how can we calculate the number of steps required to search an element from a given array using binary search without doing any calculations. It's super easy! Are you ready?

Time Complexity of Binary Search $O(\log n)$

When we say the time complexity is $\log n$, we actually mean $\log_2 n$, although the **base** of the log doesn't matter in asymptotic notations, but still to understand this better, we generally consider a base of 2.

Let's first understand what $\log_2(n)$ means.

```
Expression: log₂(n)

- - - - - - - - - - - - - - -

For n = 2:

log₂(2¹) = 1

Output = 1

- - - - - - - - - - - - - - -

For n = 4

log₂(2²) = 2

Output = 2

- - - - - - - - - - - - - - -

For n = 8
```

```
log₂(2³) = 3

Output = 3

- - - - - - - - - - - - - - -

For n = 256

log₂(2⁸) = 8

Output = 8

- - - - - - - - - - - - - - -

For n = 2048

log₂(2¹¹) = 11

Output = 11
```

Now that we know how $\log_2(n)$ works with different values of n, it will be easier for us to relate it with the time complexity of the binary search algorithm and also to understand how we can find out the number of steps required to search any number using binary search for any value of n.

Counting the Number of Steps

As we have already seen, that with every incorrect guess, binary search cuts down the list of elements into half. So if we start with 32 elements, after first unsuccessful guess, we will be left with 16 elements.

So consider an array with 8 elements, after the first unsuccessful, binary search will cut down the list to half, leaving behind 4 elements, then 2 elements after the second unsuccessful guess, and finally only 1 element will be left, which will either be the target or not, checking that will involve one more step. So all in all binary search needed at most 4 guesses to search the target in an array with 8 elements.

If the size of the list would have been 16, then after the first unsuccessful guess, we would have been left with 8 elements. And after that, as we know, we need atmost 4 guesses, add 1 guess to cut down the list from 16 to 8, that brings us to 5 guesses.

So we can say, as the number of elements are getting doubled, the number of guesses required to find the target increments by 1.

Seeing the pattern, right?

Generalizing this, we can say, for an array with n elements,

the number of times we can repeatedly halve, starting at n, until we get the value 1, plus one.

And guess what, in mathematics, the function $\log_2 n$ means exactly same. We have already seen how the **log** function works above, did you notice something there?

For $n = 8$, the output of $\log_2 n$ comes out to be 3, which means the array can be halved 3 times maximum, hence the number of steps(at most) to find the target value will be (3 + 1) = 4.

Question for you: What will be the maximum number of guesses required by Binary Search, to search a number in a list of **2,097,152** elements?

Now that we have learned the Binary Search Algorithms, you can also learn other types of Searching Algorithms and their applications:

- Linear Search
- Jump Search

Jump Search Algorithm

Jump Search Algorithm is a relatively new algorithm for searching an element in a sorted array.

The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm (which scans every element in the array to check if it matches with the element being searched or not). This can be done by skipping some fixed number of array elements or **jumping ahead by fixed number of steps** in every iteration.

Lets consider a sorted array $A[]$ of size n, with indexing ranging between **0** and **n-1**, and element **x** that needs to be searched in the array $A[]$. For implementing this algorithm, a block of size **m** is also required, that can be skipped or jumped in every iteration. Thus, the algorithm works as follows:

- **Iteration 1**: if ($x==A[0]$), then success, else, if ($x > A[0]$), then jump to the next block.
- **Iteration 2**: if ($x==A[m]$), then success, else, if ($x > A[m]$), then jump to the next block.
- **Iteration 3**: if ($x==A[2m]$), then success, else, if ($x > A[2m]$), then jump to the next block.
- At any point in time, if ($x < A[km]$), then a **linear search** is performed from index $A[(k-1)m]$ to $A[km]$

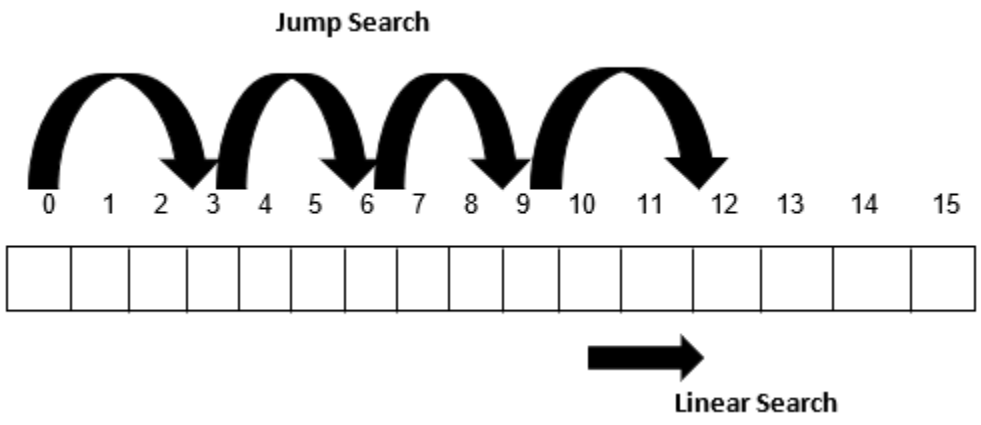


Figure 1: Jump Search technique

Optimal Size of m (Block size to be skipped)

The worst-case scenario requires:

- n/m jumps, and
- $(m-1)$ comparisons (in case of linear search if $x < A[km]$)

Hence, the total number of comparisons will be $(n/m+(m-1))$. This expression has to be minimum, so that we get the smallest value of m (block size).

On differentiating this expression with respect to m and equating it with **0**, we get:

```
n/-m2+1 = 0



n/m2 = 1



m = √n
```

Copy

Hence, the **optimal jump size** is **√n**, where **n** is the size of the array to be searched or the total number of elements to be searched.

Algorithm of Jump Search

Below we have the algorithm for implementing Jump search:

Input will be:

- Sorted array `A` of size `n`
- Element to be searched, say `item`

Output will be:

- A valid location of `item` in the array `A`

Steps for Jump Search Algorithms:

Step 1: Set `i=0` and `m = √n`.

Step 2: Compare `A[i]` with `item`. If `A[i] != item` and `A[i] < item`, then jump to the next block. Also, do the following:

1. Set **i = m**
2. Increment `m` by `√n`

Step 3: Repeat the step 2 till **m < n-1**

Step 4: If `A[i] > item`, then move to the beginning of the current block and perform a linear search.

1. Set **x = i**
2. Compare `A[x]` with `item`. If `A[x]== item`, then print `x` as the valid location else set `x++`
3. Repeat Step 4.1 and 4.2 till **x < m**

Step 5: Exit

Pictorial Representation of Jump Search with an Example

Let us trace the above algorithm using an example:

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77

Step 1: **m = √n = 4** (Block Size)

Step 2: Compare `A[0]` with `item`. Since **A[0] != item** and **A[0]<item**, skip to the next block

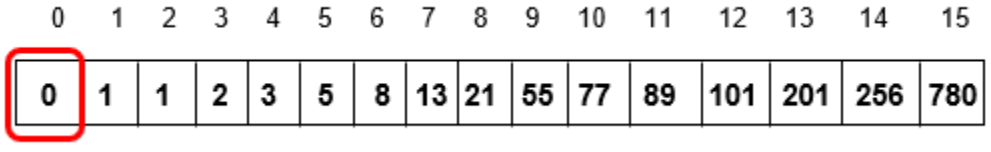


Figure 2: Comparing A[0] and item

Step 3: Compare A[3] with item. Since A[3] != itemand A[3]<item, skip to the next block

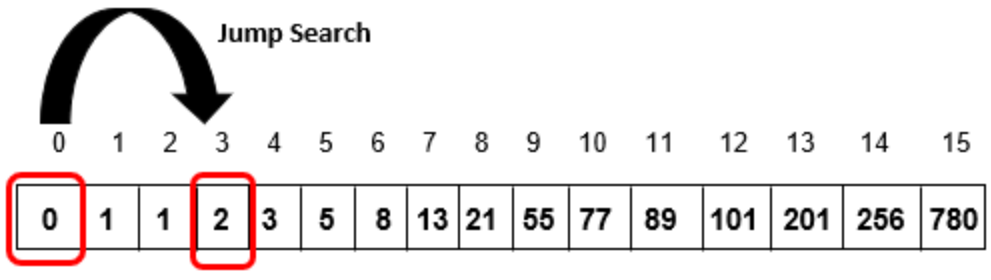


Figure 3: Comparing A[3] and item

Step 4: Compare A[6] with item. Since A[6] != itemand A[6]<item, skip to the next block

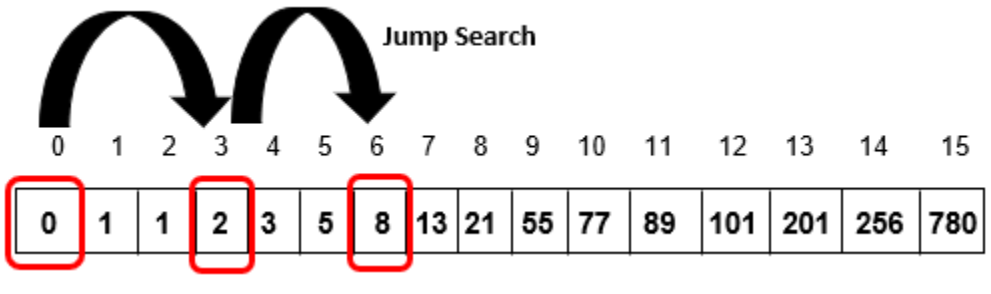


Figure 4: Comparing A[6] and item

Step 5: Compare A[9] with item. Since A[9] != itemand A[9]<item, skip to the next block

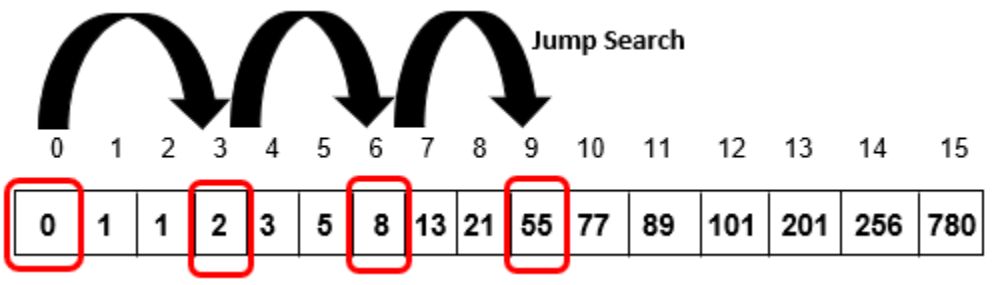


Figure 5: Comparing A[9] and item

Step 6: Compare A[12] with item. Since A[12] != item and A[12] >item, skip to A[9] (beginning of the current block) and perform a linear search.

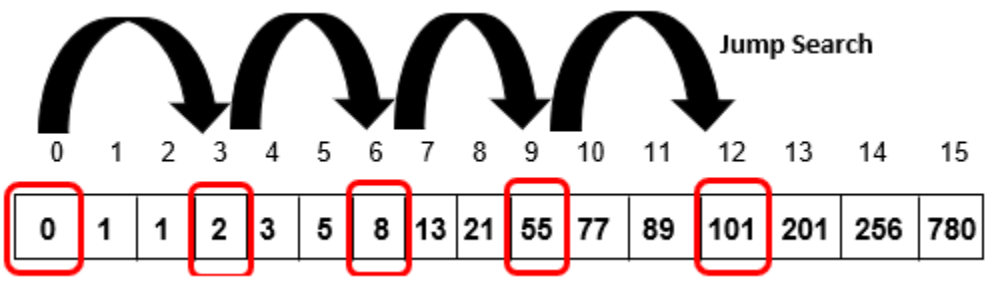


Figure 6: Comparing A[12] and item

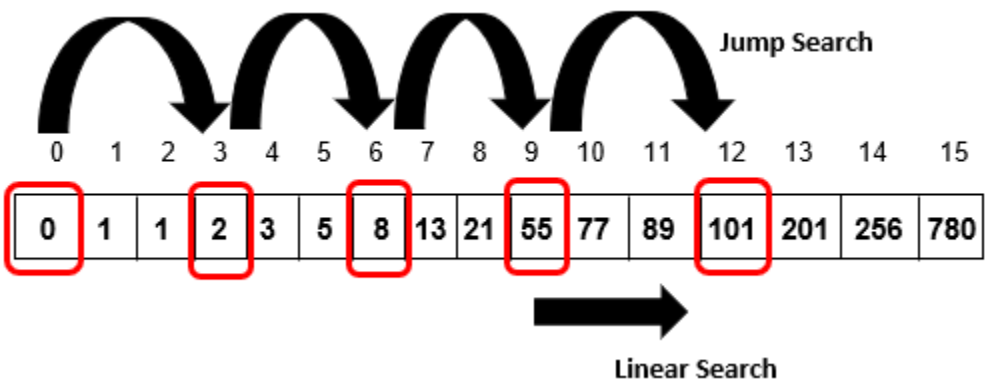


Figure 7: Comparing A[9] and item (Linear Search)

- Compare A[9] with item. Since A[9] != item, scan the next element
- Compare A[10] with item. Since A[10] == item, index 10 is printed as the valid location and the algorithm will terminate

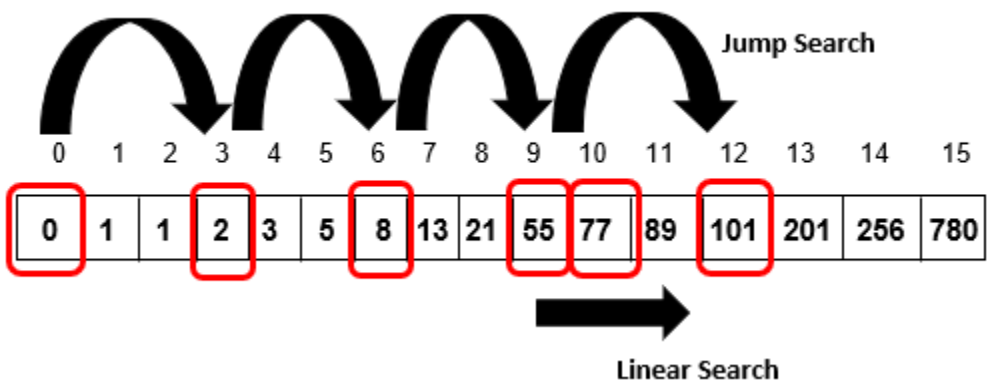


Figure 8: Comparing A[10] and item (Linear Search)

Implementation of Jump Search Algorithm

Following is the program in which we have implemented the Jump search algorithm in C++ language:

```
#include<iostream>

#include<cmath>



using namespace std;

int jump_Search(int a[], int n, int item) {

   int i = 0;

int m = sqrt(n); //initializing block size= √(n)


while(a[m] <= item && m < n) {

   // the control will continue to jump the blocks

   i = m;  // shift the block

   m += sqrt(n);

   if(m > n - 1)  // if m exceeds the array size

      return -1;

}


for(int x = i; x<m; x++) { //linear search in current block

   if(a[x] == item)

      return x; //position of element being searched

   }

   return -1;

}


int main() {

   int n, item, loc;

   cout << "\n Enter number of items: ";

   cin >> n;
```

```
   int arr[n]; //creating an array of size n

   cout << "\n Enter items: ";


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

      cin >> arr[i];

   }


   cout << "\n Enter search key to be found in the array: ";

   cin >> item;

   loc = jump_Search(arr, n, item);

   if(loc>=0)

      cout << "\n Item found at location: " << loc;

   else

      cout << "\n Item is not found in the list.";

}
```

Copy

The input array is the same as that used in the example:

```
Enter number of items: 16

Enter items: 0 1 1 2 3 5 8 13 21 55 77 89 101 201 256 780

Enter search key to be found in the array: 77

Item found at location: 10
```

Note: The algorithm can be implemented in any programming language as per the requirement.

Complexity Analysis for Jump Search

Let's see what will be the time and space complexity for the Jump search algorithm:

Time Complexity:

The while loop in the above C++ code executes n/m times because the loop counter increments by m times in every iteration. Since the optimal value of m= √n , thus, n/m=√n resulting in a time complexity of **O(√n)**.

Space Complexity:

The space complexity of this algorithm is **O(1)** since it does not requireany other data structure for its implementation.

Key Points to remember about Jump Search Algorithm

- This algorithm works only for sorted input arrays
- Optimal size of the block to be skipped is √n, thus resulting in the time complexity O(√n^2)
- The time complexity of this algorithm lies in between linear search (O(n)) and binary search (O(log n))
- It is also called block search algorithm

Advantages of Jump Search Algorithm

- It is faster than the linear search technique which has a time complexity of O(n) for searching an element

Disadvantages of Jump Search Algorithm

- It is slower than binary search algorithm which searches an element in O(log n)
- It requires the input array to be sorted

Introduction to Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it.

Well, thank god, computers don't work like this.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

Although it's easier to understand these sorting techniques, but still we suggest you to first learn about Space complexity, Time complexity and the searching algorithms, to warm up your brain for sorting algorithms.

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

NOTE: If you are not familiar with Sorting in data structure, you should first learn what is sorting to know about the basics of sorting.

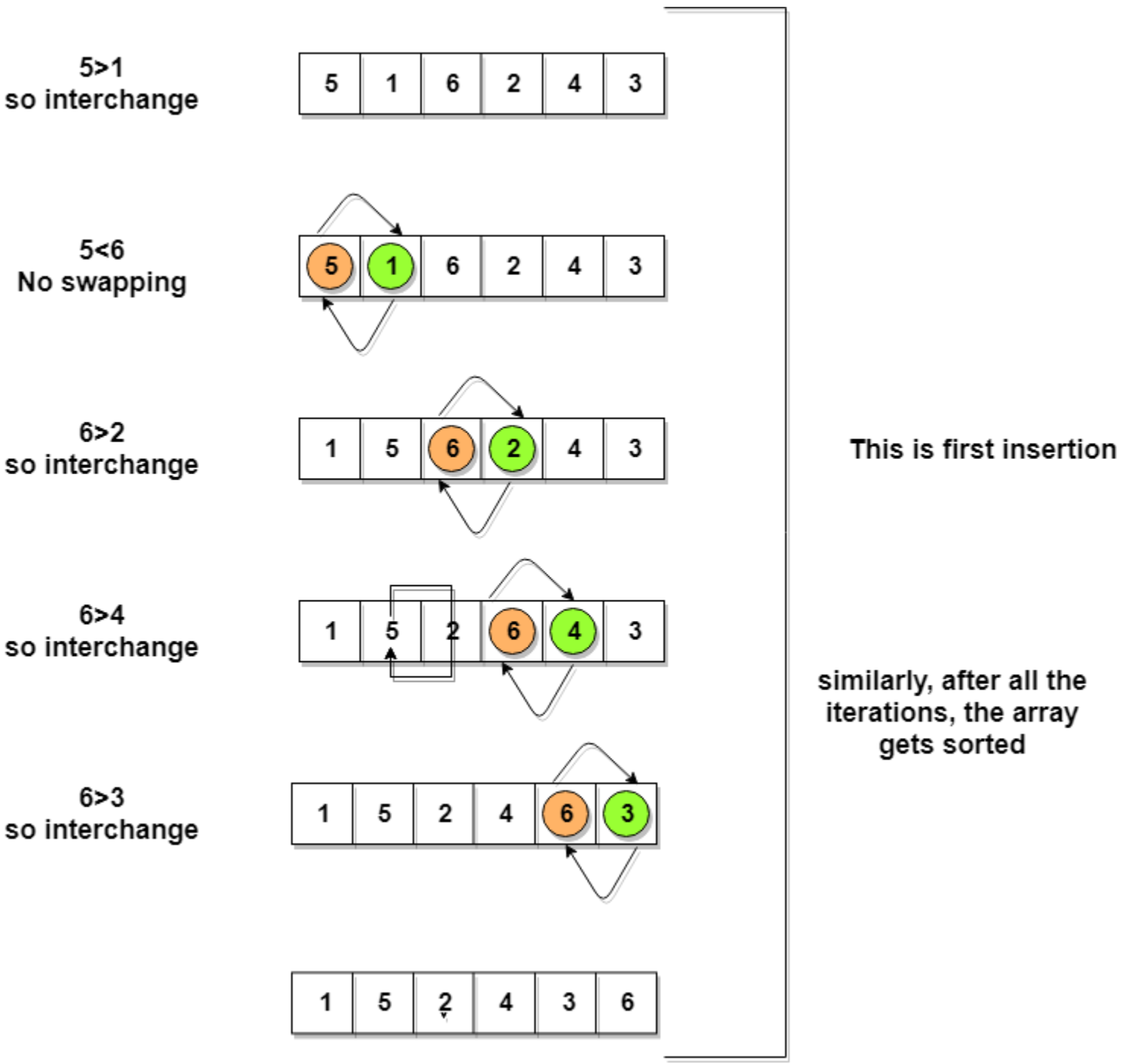
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

```c
// below we have a simple C program for bubble sort

#include <stdio.h>



void bubbleSort(int arr[], int n)

{

    int i, j, temp;

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

    {

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

    {

        if( arr[j] > arr[j+1])

        {

            // swap the elements

            temp = arr[j];

            arr[j] = arr[j+1];

            arr[j+1] = temp;

        }

    }

}



// print the sorted array

    printf("Sorted Array: ");

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

    {

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

    }

}



int main()

{

    int arr[100], i, n, step, temp;

    // ask user for number of elements to be sorted
```

```
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Copy

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.

So, we can clearly optimize our algorithm.

Optimized Bubble Sort Algorithm

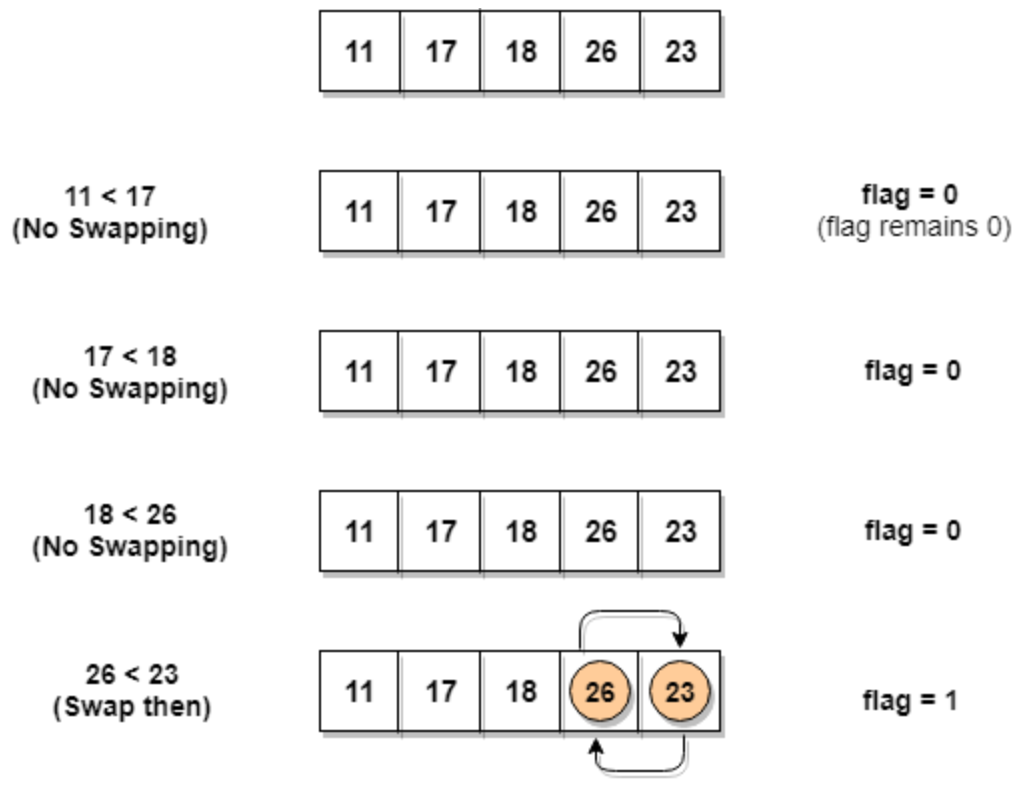
To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.

Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again. But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

```c
// below we have a simple C program for bubble sort

#include <stdio.h>



void bubbleSort(int arr[], int n)

{

    int i, j, temp, flag=0;

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

    {

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

        {

            // introducing a flag to monitor swapping

            if( arr[j] > arr[j+1])

            {

                // swap the elements

                temp = arr[j];

                arr[j] = arr[j+1];

                arr[j+1] = temp;

                // if swapping happens update flag to 1

                flag = 1;

            }

        }

        // if value of flag is zero after all the iterations of inner loop

        // then break out

        if(flag==0)

        {

            break;

        }

    }



    // print the sorted array

    printf("Sorted Array: ");

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

```
    {
        printf("%d  ", arr[i]);
    }
}


int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);


    return 0;
}
```

Copy

```
Enter the number of elements to be sorted: 5
Enter element no. 1: 2
Enter element no. 2: 3
Enter element no. 3: 34
Enter element no. 4: 22
Enter element no. 5: 11
Sorted Array: 2  3  11  22  34
```

In the above code, in the function bubbleSort, if for a single complete cycle of j iteration(inner for loop), no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
```

```
Sum = n(n-1)/2

i.e O(n²)
```

Hence the **time complexity** of Bubble Sort is **O(n²)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **O(n²)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n²)**
- Space Complexity: **O(1)**

Now that we have learned Bubble sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Quick Sort
- merge Sort
- Heap Sort
- Counting Sort

Insertion Sort Algorithm

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do?

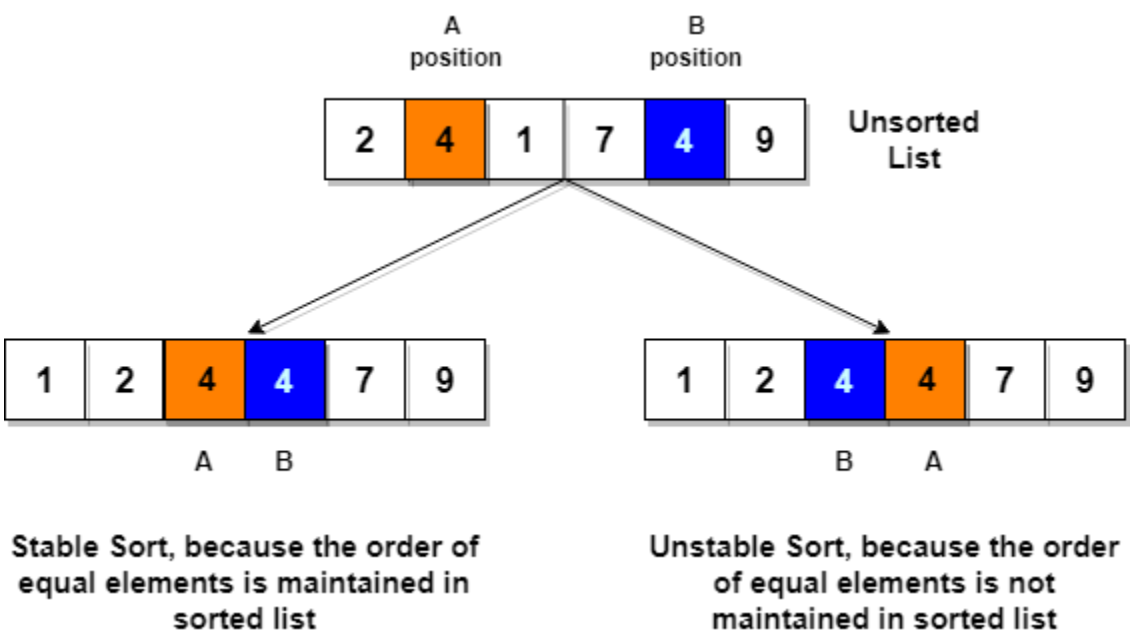
Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how **insertion sort** works. It starts from the index `1`(not `0`), and each index starting from index `1` is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.



How Insertion Sort Works?

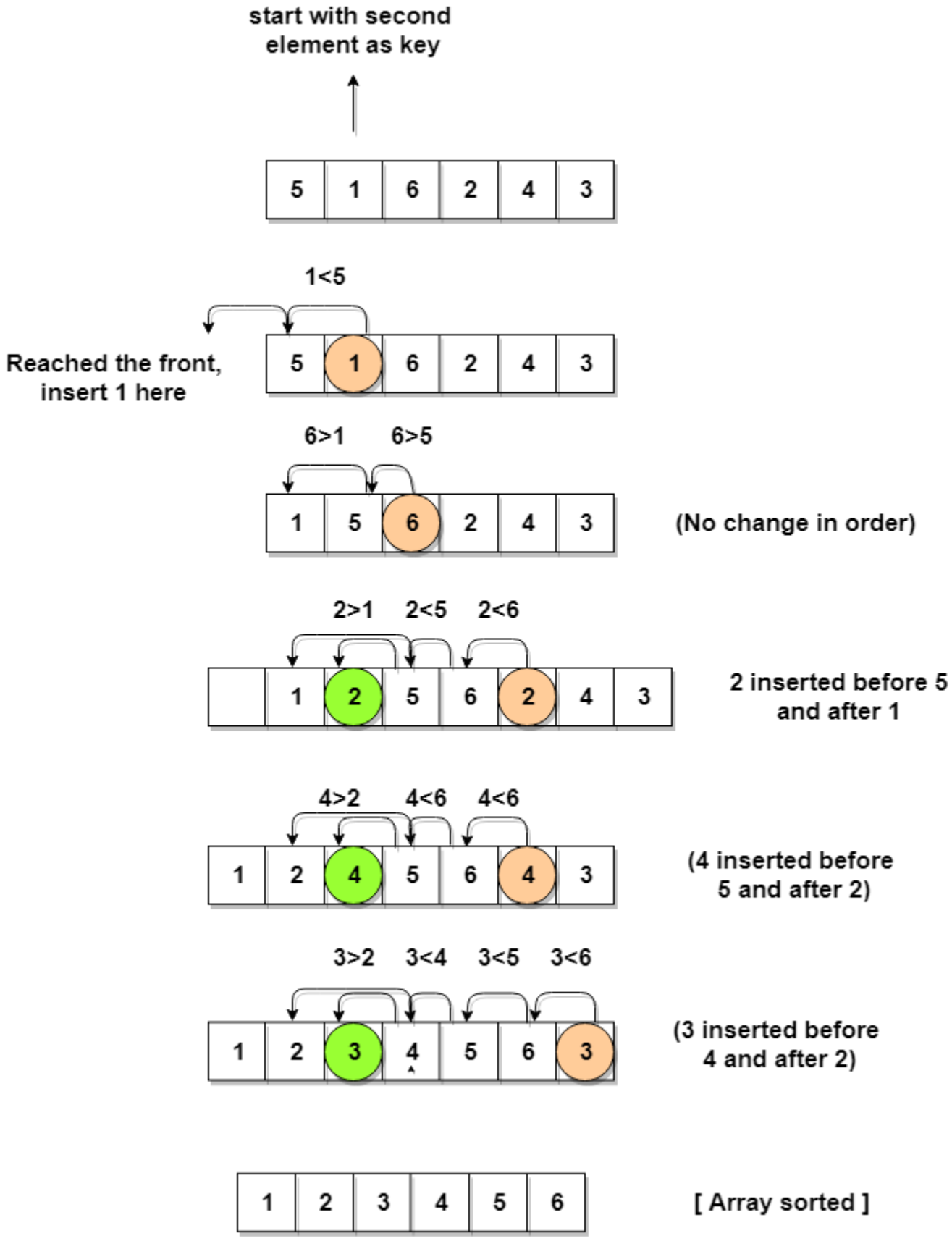
Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index `1`, the `key`. The `key` element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the `key` element with the element(s) before it, in this case, element at index `0`:

    - If the `key` element is less than the first element, we insert the `key` element before the first element.
    - If the `key` element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as `key` and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a `key`, we start iterating over the elements to the left of the `key`.

We continue to move towards left if the elements are greater than the `key` element and stop when we find the element which is less than the `key` element.

And, insert the `key` element after the element which is less than the `key` element.

Implementing Insertion Sort Algorithm

Below we have a simple implementation of Insertion sort in C++ language.

```
#include <stdlib.h>
#include <iostream>

using namespace std;

//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[], int size);

// main function
int main()
{
     int array[5] = {5, 1, 6, 2, 4, 3};
     // calling insertion sort function to sort the array
     insertionSort(array, 6);
     return 0;
}

void insertionSort(int arr[], int length)
{
     int i, j, key;
     for (i = 1; i < length; i++)
     {
          j = i;
          while (j > 0 && arr[j - 1] > arr[j])
          {
               key = arr[j];
```

```
                arr[j] = arr[j - 1];

                arr[j - 1] = key;

                j--;

            }

        }

        cout << "Sorted Array: ";

        // print the sorted array

        printArray(arr, length);

}


// function to print the given array

void printArray(int array[], int size)

{

        int j;

        for (j = 0; j < size; j++)

        {

                cout <<" "<< array[j];

        }

        cout << endl;

}
```

Copy

```
Sorted Array:  1 2 3 4 5 6
```

Now let's try to understand the above simple insertion sort algorithm.

We took an array with **6** integers. We took a variable `key`, in which we put each element of the array, during each pass, starting from the **second** element, that is `a[1]`.

Then using the `while` loop, we iterate, until `j` becomes equal to **zero** or we find an element which is greater than `key`, and then we **insert** the `key` at that position.

We keep on doing this, until `j` becomes equal to **zero**, or we encounter an element which is smaller than the `key`, and then we stop. The current `key` is now at the right position.

We then make the next element as `key` and then repeat the same process.

In the above array, first we pick **1** as `key`, we compare it with **5**(element before 1), **1** is smaller than **5**, we insert **1** before **5**. Then we pick **6** as `key`, and compare it with **5** and **1**, no shifting in

position this time. Then **2** becomes the `key` and is compared with **6** and **5**, and then **2** is inserted after **1**. And this goes on until the complete array gets sorted.

Complexity Analysis of Insertion Sort

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using `for` loops, but instead it uses one `while` loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer `for` loop, thereby requiring `n` steps to sort an already sorted array of `n` elements, which makes its **best case time complexity** a linear function of `n`.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

Selection Sort Algorithm

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

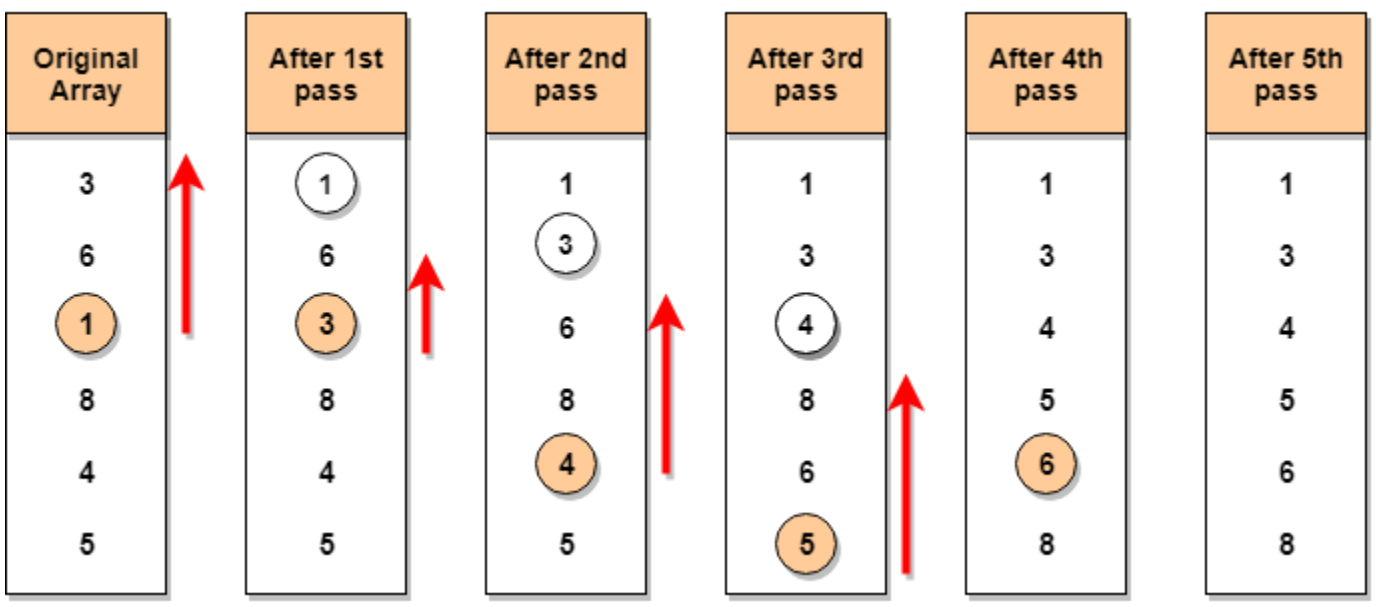
How Selection Sort Works?

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index `1`, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values `{3, 6, 1, 8, 4, 5}`

Below, we have a pictorial representation of how selection sort will sort the given array.



In the **first** pass, the smallest element will be `1`, so it will be placed at the first position.

Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get `3` as the smallest, so it will be then placed at the second position.

Then leaving `1` and `3`(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Finding Smallest Element in a subarray

In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?

Consider that you have an array with following values $\{3, 6, 1, 8, 4, 5\}$. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the **index** 2. Once the smallest number is found, it is swapped with the element at the first position.

Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?

If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.

So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.

Confused? Give it time to sink in.

After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

Then we will work on the subarray, starting from index 2 now, and again looking for the smallest element in this subarray.

Implementing Selection Sort Algorithm

In the C program below, we have tried to divide the program into small functions, so that it's easier fo you to understand which part is doing what.

There are many different ways to implement selection sort algorithm, here is the one that we like:

```
// C program implementing Selection Sort

# include <stdio.h>


// function to swap elements at the given index values

void swap(int arr[], int firstIndex, int secondIndex)

{

    int temp;

    temp = arr[firstIndex];

    arr[firstIndex] = arr[secondIndex];

    arr[secondIndex] = temp;

}
```

```
// function to look for smallest element in the given subarray
int indexOfMinimum(int arr[], int startIndex, int n)
{
    int minValue = arr[startIndex];
    int minIndex = startIndex;

    for(int i = minIndex + 1; i < n; i++) {
        if(arr[i] < minValue)
        {
            minIndex = i;
            minValue = arr[i];
        }
    }
    return minIndex;
}

void selectionSort(int arr[], int n)
{
    for(int i = 0; i < n; i++)
    {
        int index = indexOfMinimum(arr, i, n);
        swap(arr, i, index);
    }

}

void printArray(int arr[], int size)
{
    int i;
```

```
    for(i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}


int main()
{
    int arr[] = {46, 52, 21, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Copy

Note: Selection sort is an **unstable sort** i.e it might change the occurrence of two similar elements in the list while sorting. But it can also work as a stable sort when it is implemented using linked list.

Complexity Analysis of Selection Sort

Selection Sort requires two nested `for` loops to complete itself, one `for` loop is in the function `selectionSort`, and inside the first loop we are making a call to another function `indexOfMinimum`, which has the second(inner) `for` loop.

Hence for a given input size of `n`, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **$O(n^2)$**

Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

Quick Sort Algorithm

Quick Sort is one of the different Sorting Technique which is based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

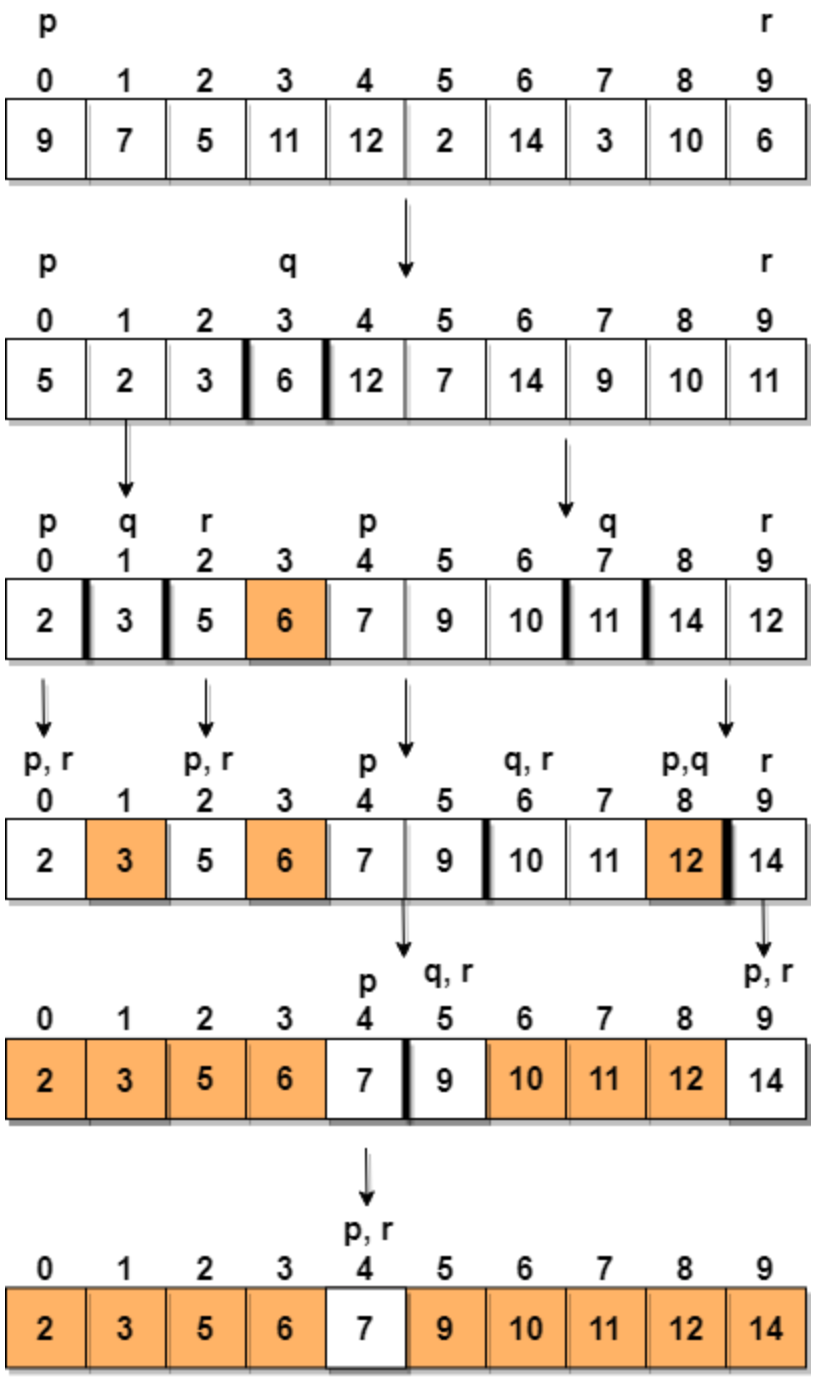
How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the **pivot**, which is `6` in this case, and call for `partitioning`, hence re-arranging the array in such a way that `6` will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose `3` as pivot for the left subarray and `11` as pivot for the right subarray.

And we again call for `partitioning`.

Implementing Quick Sort Algorithm

Below we have a simple C program implementing the Quick sort algorithm:

```
// simple C program for Quick Sort

#include <stdio.h>

int partition(int a[], int beg, int end);

void quickSort(int a[], int beg, int end);

void main()

{

    int i;

    int arr[10]={90,23,101,45,65,28,67,89,34,29};
```

```
    quickSort(arr, 0, 9);

    printf("\n The sorted array is: \n");

    for(i=0;i<10;i++)

    printf(" %d\t", arr[i]);

}

int partition(int a[], int beg, int end)

{

    int left, right, temp, loc, flag;

    loc = left = beg;

    right = end;

    flag = 0;

    while(flag != 1)

    {

        while((a[loc] <= a[right]) && (loc!=right))

        right--;

        if(loc==right)

        flag =1;

        else if(a[loc]>a[right])

        {

            temp = a[loc];

            a[loc] = a[right];

            a[right] = temp;

            loc = right;

        }

        if(flag!=1)

        {

            while((a[loc] >= a[left]) && (loc!=left))

            left++;

            if(loc==left)

            flag =1;
```

```
            else if(a[loc] < a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
    return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

Copy

```
The sorted array is:
23      28      29      34      45      65      67      89      90      101
```

Complexity Analysis of Quick Sort

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$

Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n*log n)**

As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

- Space required by quick sort is very less, only $O(n*\log n)$ additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Now that we have learned Quick sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge sort
- Heap Sort
- Counting Sort

Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

Before moving forward with Merge Sort, check these topics out first:

- Selection Sort
- Insertion Sort
- Space Complexity of Algorithms
- Time Complexity of Algorithms

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in $O(n*\log n)$ time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?

Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, **Divide and Rule**.

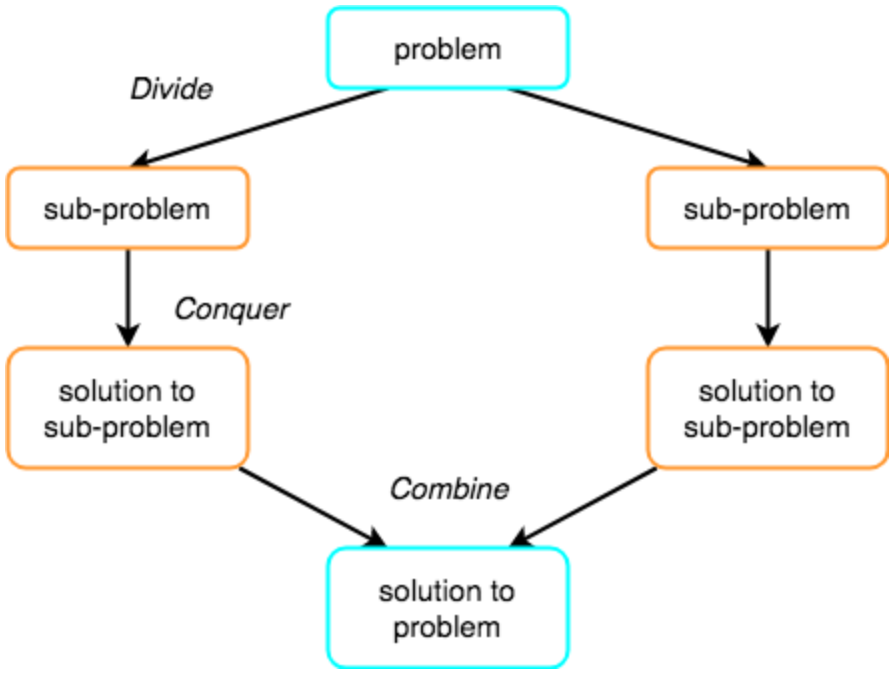
When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each.

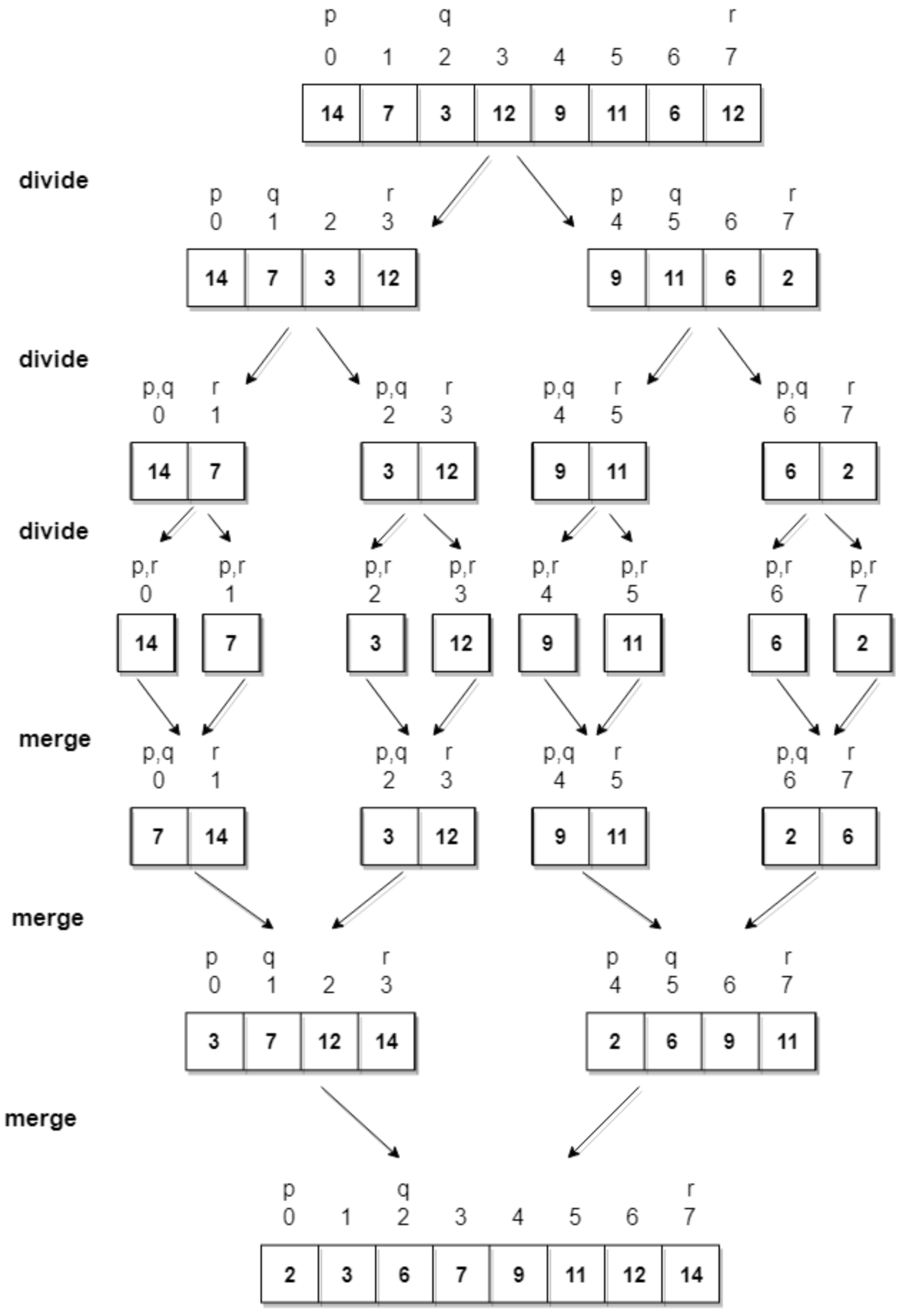
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

Let's consider an array with values $\{14, 7, 3, 12, 9, 11, 6, 12\}$

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula $(p + r)/2$ and mark the middle index as q, and break the array into two subarrays, from p to q and from $q + 1$ to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Implementing Merge Sort Algorithm

Below we have a C program implementing merge sort algorithm.

```
/*
```

```
    a[] is the array, p is starting index, that is 0,
    and r is the last index of array.
*/

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

// merge sort function
void mergeSort(int a[], int p, int r)
{
    int q;
    if(p < r)
    {
        q = (p + r) / 2;
        mergeSort(a, p, q);
        mergeSort(a, q+1, r);
        merge(a, p, q, r);
    }
}

// function to merge the subarrays
void merge(int a[], int p, int q, int r)
{
    int b[5];   //same size of a[]
    int i, j, k;
    k = 0;
    i = p;
    j = q + 1;
    while(i <= q && j <= r)
```

```
    {
        if(a[i] < a[j])
        {
            b[k++] = a[i++];     // same as b[k]=a[i]; k++; i++;
        }
        else
        {
            b[k++] = a[j++];
        }
    }


    while(i <= q)
    {
        b[k++] = a[i++];
    }


    while(j <= r)
    {
        b[k++] = a[j++];
    }


    for(i=r; i >= p; i--)
    {
        a[i] = b[--k];  // copying back the sorted list to a[]
    }
}


// function to print the array
void printArray(int a[], int size)
{
```

```
    int i;
    for (i=0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {32, 45, 67, 2, 7};
    int len = sizeof(arr)/sizeof(arr[0]);

    printf("Given array: \n");
    printArray(arr, len);

    // calling merge sort
    mergeSort(arr, 0, len - 1);

    printf("\nSorted array: \n");
    printArray(arr, len);
    return 0;
}
```

Copy

```
Given array:
32 45 67 2 7

Sorted array:
2 7 32 45 67
```

Complexity Analysis of Merge Sort

Merge Sort is quite fast, and has a time complexity of $O(n*\log n)$. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

In this section we will understand why the running time for merge sort is $O(n*\log n)$.

As we have already learned in Binary Search that whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is $\log n$ and the number of steps can be represented by $\log n + 1$(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. $O(1)$.

And to **merge** the subarrays, made by dividing the original array of n elements, a running time of $O(n)$ will be required.

Hence the total time for mergeSort function will become $n(\log n + 1)$, which gives us a time complexity of $O(n*\log n)$.

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

- Time complexity of Merge Sort is $O(n*\text{Log } n)$ in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.
- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.
- It is the best Sorting technique used for sorting **Linked Lists**.

Now that we have learned Insertion sorting algorithm, you can check out these other sorting algorithm and their applications aswell :

- Bubble Sort
- Insertion Sort
- Quick Sort
- Selection Sort
- Heap Sort
- Counting Sort

Heap Sort Algorithm

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

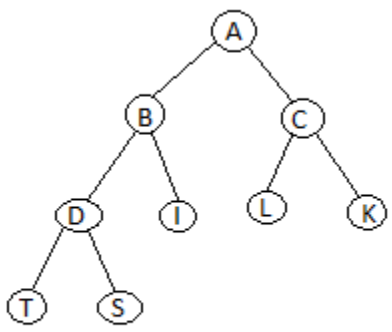
You must be wondering, how converting an array of numbers into a heap data structure will help in sorting the array. To understand this, let's start by understanding what is a Heap.

NOTE: If you are not familiar with Sorting in data structure, you should first learn what is sorting to know about the basics of sorting.

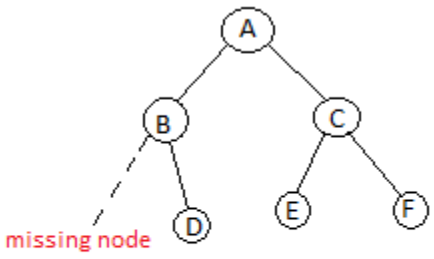
What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

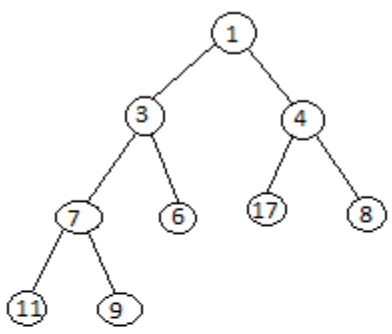


Complete Binary Tree



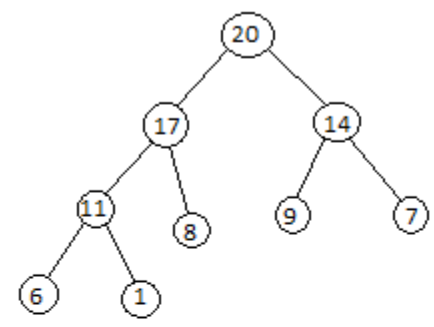
In-Complete Binary Tree

2. **Heap Property:** All nodes are either **greater than or equal to** or **less than or equal to** each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works?

Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially `heapsort()` function is called, which calls `heapify()` to build the heap.

Implementing Heap Sort Algorithm

Below we have a simple C++ program implementing the Heap sort algorithm.

```
/*  Below program is written in C++ language  */


#include <iostream>


using namespace std;


void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;


    // if left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;


    // if right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

```
    // if largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // one by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* function to print array of size n */
void printArray(int arr[], int n)
{
```

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



int main()
{
    int arr[] = {121, 10, 130, 57, 36, 17};
    int n = sizeof(arr)/sizeof(arr[0]);


    heapSort(arr, n);


    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Copy

Complexity Analysis of Heap Sort

Worst Case Time Complexity: **O(n*log n)**

Best Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

Space Complexity : **O(1)**

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.

Now that we have learned Heap sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Bubble Sort

- Merge sort
- Heap Sort
- Counting Sort

Counting Sort Algorithm

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

- **Input**: Unsorted array A[] of n elements
- **Output**: Sorted arrayB[]

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

a. Set i=0 and $t = A[i]$

b. Add t to B[v] such that $v = (count[t]-1)$.

c. Decrement count[t] by 1

d. Increment i by 1

Repeat steps **(a)** to **(d)** till **i = n-1**

Step 6: Display B since this is the sorted array

Pictorial Representation of Counting Sort with an Example

Let us trace the above algorithm using an example:

Consider the following input array A to be sorted. All the elements are in range 0 to 9

```
A[]= {1, 3, 2, 8, 5, 1, 5, 1, 2, 7}
```

Copy

Step 1: Initialize an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)

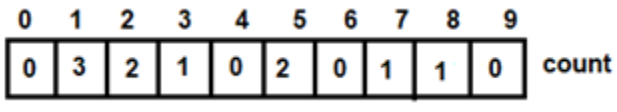


Figure 1: count array

Step 2: Using the formula, updated count array is -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Figure 2: Formula for updating count array

0	1	2	3	4	5	6	7	8	9	
0	3	5	6	6	8	8	9	10	10	count

Figure 3 : Updated count array

Step 3: Add elements of array A to resultant array B using the following steps:

- For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	0	0	0	0	0	B
0	2	5	6	6	8	8	9	10	10	count

Figure 4: For i=0

- For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	3	0	0	0	0	B
0	2	5	5	6	8	8	9	10	10	count

Figure 5: For i=1

- For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	0	B
0	2	4	5	6	8	8	9	10	10	count

Figure 6: For i=2

- For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	8	B
0	2	4	5	6	8	8	9	9	10	count

Figure 7: For i=3

- On similar lines, we have the following:

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	5	0	8	B
0	2	4	5	6	7	8	9	9	10	count

Figure 8: For i=4

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	0	5	0	8	B
0	1	4	5	6	7	8	9	9	10	count

Figure 9: For i=5

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	5	5	0	8	B
0	1	4	5	6	6	8	9	9	10	count

Figure 10: For i=6

0	1	2	3	4	5	6	7	8	9	
1	1	1	0	2	3	5	5	0	8	B
0	0	4	5	6	6	8	9	9	10	count

Figure 11: For i=7

0	1	2	3	4	5	6	7	8	9	
1	1	1	2	2	3	5	5	0	8	B
0	0	3	5	6	6	8	9	9	10	count

Figure 12: For i=8

Figure 13: For i=9

Thus, array B has the sorted list of elements.

Program for Counting Sort Algorithm

Below we have a simple program in C++ implementing the counting sort algorithm:

```
#include<iostream>

using namespace std;



int k=0;  // for storing the maximum element of input array



/* Method to sort the array */

void sort_func(int A[],int B[],int n)

{

     int count[k+1],t;

     for(int i=0;i<=k;i++)

     {

          //Initialize array count

          count[i] = 0;

     }

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

     {

         // count the occurrence of elements u of A

         // & increment count[u] where u=A[i]

          t = A[i];

          count[t]++;

     }

     for(int i=1;i<=k;i++)

     {
```

```
        // Updating elements of count array
        count[i] = count[i]+count[i-1];
    }
    for(int i=0;i<n;i++)
    {
        // Add elements of array A to array B
        t = A[i];
        B[count[t]] = t;
        // Decrement the count value by 1
        count[t]=count[t]-1;
        }
}
int main()
{
    int n;
    cout<<"Enter the size of the array :";
    cin>>n;

    // A is the input array and will store elements entered by the
user
    // B is the output array having the sorted elements
    int A[n],B[n];
    cout<<"Enter the array elements: ";
    for(int i=0;i<n;i++)
    {
        cin>>A[i];
        if(A[i]>k)
        {
            // k will have the maximum element of A[]
            k = A[i];
        }
```

```
    }


    sort_func(A,B,n);


    // Printing the elements of array B
    for(int i=1;i<=n;i++)
    {
         cout<<B[i]<<" ";
    }

         cout<<"\n";
         return 0;
}
```

Copy

The input array is the same as that used in the example:

```
Enter the size of the array :10
Enter the array elements: 1 3 2 8 5 1 5 1 2 7
1 1 1 2 2 3 5 5 7 8


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 14: Output of Program

Note: The algorithm can be mapped to any programming language as per the requirement.

Time Complexity Analysis

For scanning the input array elements, the loop iterates n times, thus taking $O(n)$ running time. The sorted array B[] also gets computed in n iterations, thus requiring O(n) running time. The count array also uses k iterations, thus has a running time of O(k). Thus the total running time for counting sort algorithm is $O(n+k)$.

Key Points:

- The above implementation of Counting Sort can also be extended to sort negative input numbers
- Since counting sort is suitable for sorting numbers that belong to a well-defined, finite and small range, it can be used asa subprogram in other sorting algorithms like radix sort which can be used for sorting numbers having a large range
- Counting Sort algorithm is efficient if the range of input data (k) is not much greater than the number of elements in the input array (n). It will not work if we have 5 elements to sort in the range of 0 to 10,000

- It is an integer-based sorting algorithm unlike others which are usually comparison-based. A comparison-based sorting algorithm sorts numbers only by comparing pairs of numbers. Few examples of comparison based sorting algorithms are **quick sort, merge sort, bubble sort, selection sort, heap sort, insertion sort**, whereas algorithms like radix sort, bucket sort and comparison sort fall into the category of non-comparison based sorting algorithms.

Advantages of Counting Sort:

- It is quite fast
- It is a stable algorithm

Note: For a sorting algorithm to be stable, the order of elements with equal keys (values) in the sorted array should be the same as that of the input array.

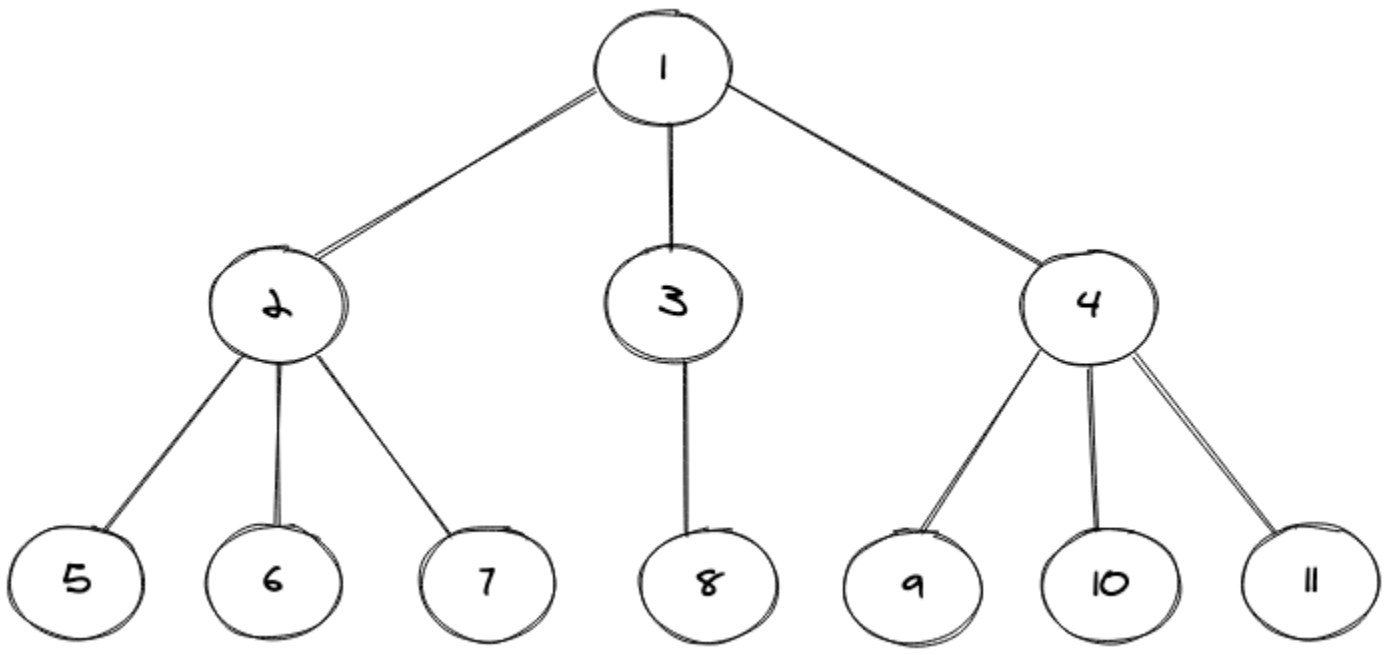
Disadvantages of Counting Sort:

- It is not suitable for sorting large data sets
- It is not suitable for sorting string values

N-ary Tree Data Structure

The N-ary tree is a tree that allows us to have `n` number of children of a particular node, hence the name **N-ary**, making it **slightly complex than the very common binary trees** that allow us to have at most 2 children of a particular node.

A pictorial representation of an N-ary tree is shown below:



In the N-ary tree shown above, we can notice that there are **11 nodes** in total and **some nodes have three children** and some have had one only. In the case of a binary tree, it was easier to store these children nodes as we can assign two nodes (i.e. left and right) to a particular node to mark its children, but here it's not that simple. To store the children nodes of any tree node we make use of another data structure, mainly vector in C++ and LinkedList in Java.

Implementation of N-ary Tree

The first thing that we do when we are dealing with non-linear data structures is to create our own structure (constructors in Java) for them. Like in the case of a binary tree, we make use of a class `TreeNode` and inside that class, we create our constructors and have our class-level variables.

Consider the code snippet below:

```
public static class TreeNode{

        int val;

        List<TreeNode> children = new LinkedList<>();


        TreeNode(int data){

            val = data;

        }


        TreeNode(int data,List<TreeNode> child){

            val = data;

            children = child;
```

```
        }

}
```

Copy

In the above code snippet, we have a class named `TreeNode` which in turn contains two constructors with the same name, but they are overloaded (same method names but with different parameters) in nature. We also have two identifiers, one of them is the val that stores the value of any particular node, and then we have a List to store the children nodes of any node of the tree.

The above snippet contains the basic structure of our tree, and now are only left with making one and then later we will see how to print the tree using level order traversal. To build a tree we will make use of the constructors that we have defined in the above classes.

Consider the code snippet below:

```
public static void main(String[] args) {

        // creating an exact replica of the above pictorial N-ary Tree

        TreeNode root = new TreeNode(1);

        root.children.add(new TreeNode(2));

        root.children.add(new TreeNode(3));

        root.children.add(new TreeNode(4));

        root.children.get(0).children.add(new TreeNode(5));

        root.children.get(0).children.add(new TreeNode(6));

        root.children.get(0).children.add(new TreeNode(7));

        root.children.get(1).children.add(new TreeNode(8));

        root.children.get(2).children.add(new TreeNode(9));

        root.children.get(2).children.add(new TreeNode(10));

        root.children.get(2).children.add(new TreeNode(11));

        printNAryTree(root);

}
```

Copy

First thing first, we created the **root node of our N-ary Tree**, then we have to assign some children to this root node, we do this by making use of the dot(.) operator and accessing the children property of the root node and then adding different children to the root nodes by using the add method provided by the `List` interface. Once we have added all the children of the root node, it is time to add children of each of the new level nodes, we do that by first accessing that node by using the get method provided by the list interface and then adding the respective children nodes to that node.

Lastly, we print this N-ary Tree, we did it by calling the `printNAryTree` method.

Now, since printing a tree is not as simple as looping through a collection of items, we have different techniques (algorithms precisely) at our disposal. These are mainly:

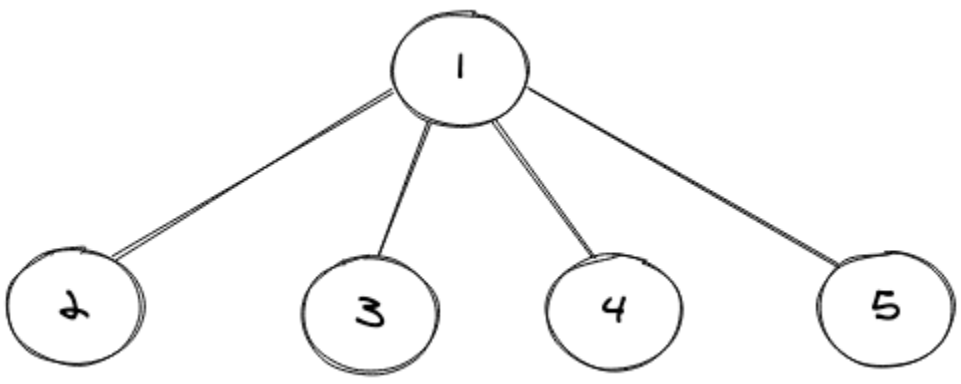
- **Inorder Traversal**
- **Preorder Traversal**
- **PostOrder Traversal**
- **Level Order Traversal**

For the sake of this tutorial, we will use the Level Order Traversal approach, as it is easier to understand, provided if you have seen how it works on a binary tree before.

Level Order Traversal (Printing the N-ary Tree)

The Level Order traversal of any tree takes into the fact that we want to print the nodes at a root level first, then move on to the next level and keep repeating this process until we are at the last level. We make use of the Queue data structure to store the nodes at a particular level.

Consider a simple N-ary Tree shown below:



Level Order Traversal for the above tree will look like this:

```
1

2 3 4 5
```

Copy

Consider the code snippet below:

```
private static void printNAryTree(TreeNode root){

        if(root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty()) {

            int len = queue.size();

            for(int i=0;i<len;i++) { // so that we can reach each level

                TreeNode node = queue.poll();

                System.out.print(node.val + " ");

                for (TreeNode item : node.children) { // for-Each loop
to iterate over all childrens

                    queue.offer(item);
```

```
                }

            }

            System.out.println();

        }

}
```

Copy

The entire code looks something like this:

```
import java.util.LinkedList;

import java.util.List;

import java.util.Queue;


public class NAryTree {

    public static class TreeNode{

        int val;

        List<TreeNode> children = new LinkedList<>();


        TreeNode(int data){

            val = data;

        }


        TreeNode(int data,List<TreeNode> child){

            val = data;

            children = child;

        }

    }


    private static void printNAryTree(TreeNode root){

        if(root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty()) {
```

```
            int len = queue.size();
            for(int i=0;i<len;i++) {
                TreeNode node = queue.poll();
                assert node != null;
                System.out.print(node.val + " ");
                for (TreeNode item : node.children) {
                    queue.offer(item);
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.children.add(new TreeNode(2));
        root.children.add(new TreeNode(3));
        root.children.add(new TreeNode(4));
        root.children.get(0).children.add(new TreeNode(5));
        root.children.get(0).children.add(new TreeNode(6));
        root.children.get(0).children.add(new TreeNode(7));
        root.children.get(1).children.add(new TreeNode(8));
        root.children.get(2).children.add(new TreeNode(9));
        root.children.get(2).children.add(new TreeNode(10));
        root.children.get(2).children.add(new TreeNode(11));
        printNAryTree(root);
    }
}
```

Copy

The output of the above code is:

```
1

2 3 4

5 6 7 8 9 10 11
```

We can compare this output to the pictorial representation of the N-ary tree we had in the starting, each level node contains the same values.

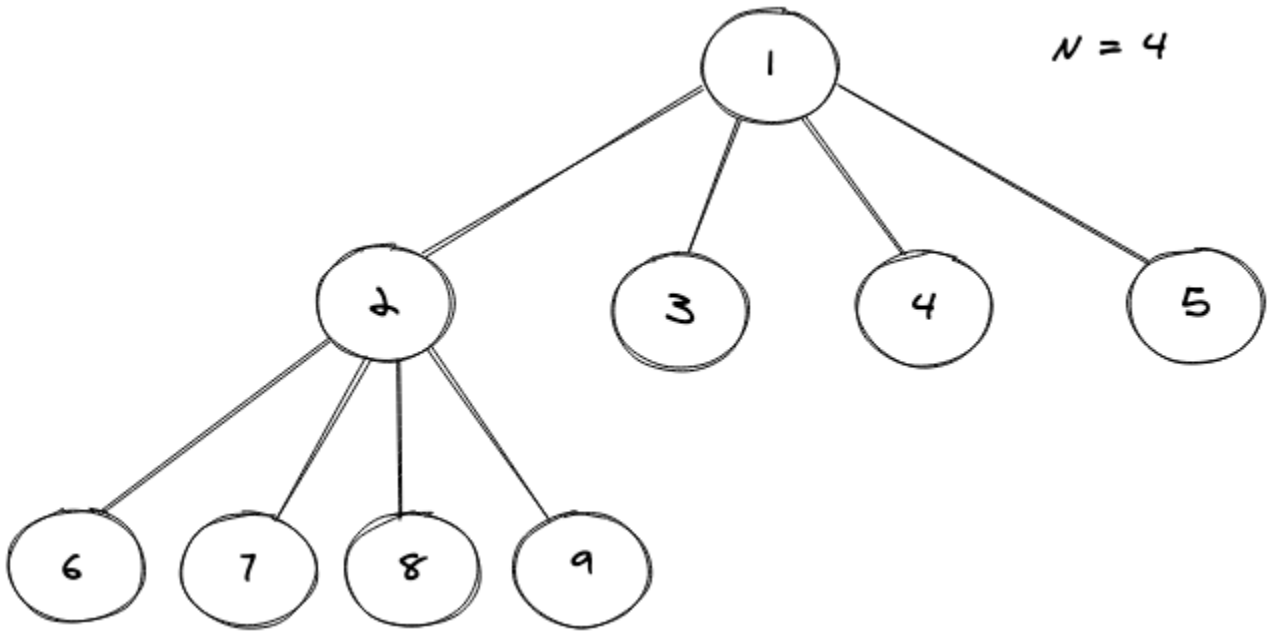
Types of N-ary Tree

Following are the types of N-ary tree:

1. Full N-ary Tree

A Full N-ary Tree is an N-ary tree that allows each node to have either 0 or N children.

Consider the pictorial representation of a full N-ary tree shown below:

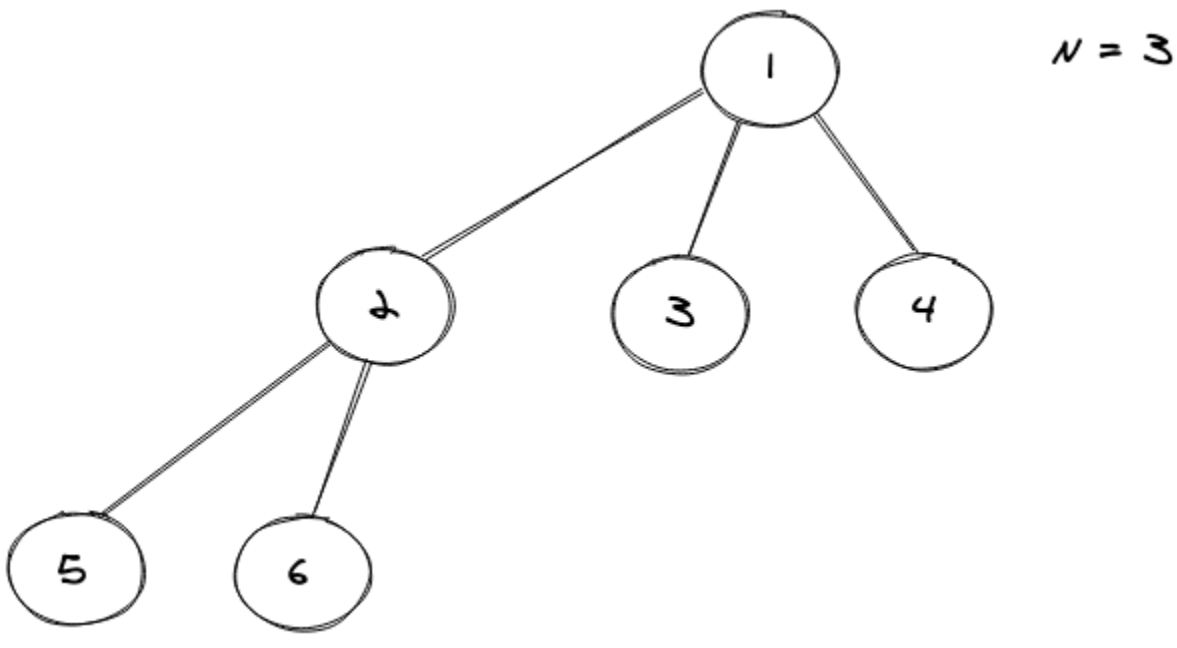


Notice that all the nodes of the above N-ary have either 4 children or 0, hence satisfying the property.

2. Complete N-ary Tree

A complete N-ary tree is an N-ary tree in which the nodes at each level of the tree should be complete (should have exactly **N children**) except the last level nodes and if the last level nodes aren't complete, the nodes must be "as left as possible".

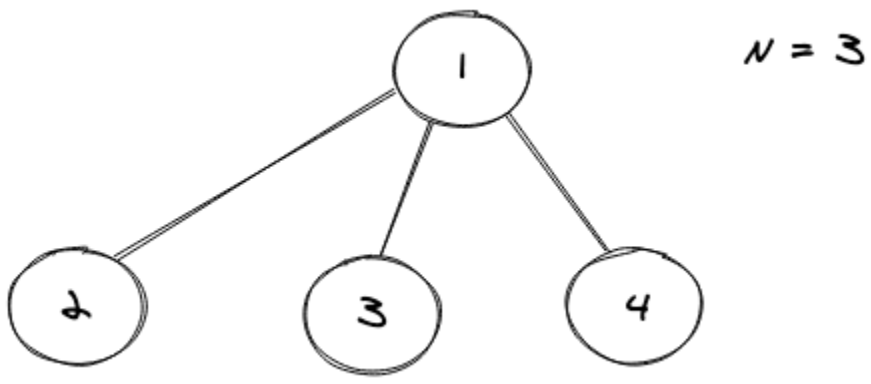
Consider the pictorial representation of a complete N-ary tree shown below:



3. Perfect N-ary Tree

A perfect N-ary tree is a full N-ary tree but the level of the leaf nodes must be the same.

Consider the pictorial representation of a perfect N-ary tree shown below:



Conclusions

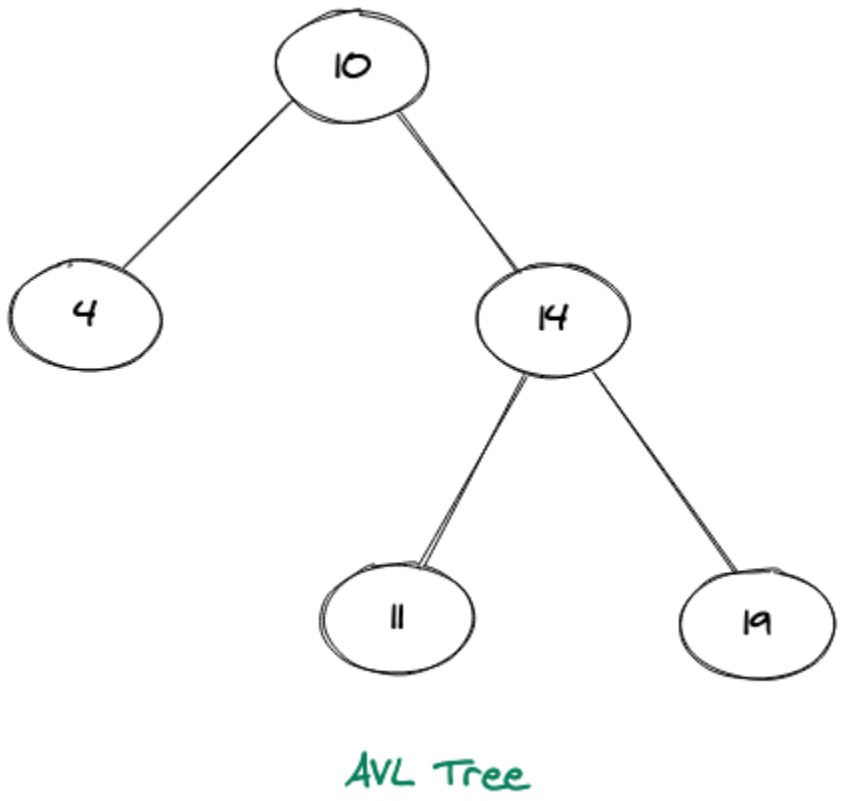
- We learned what an N-ary tree is.
- We also learned how to implement an N-ary tree in Java(via level order traversal).
- Then we learned what different types of N-ary trees are there in total.

AVL Tree Data Structure

An AVL tree is another special tree that has several properties that makes it special. These are:

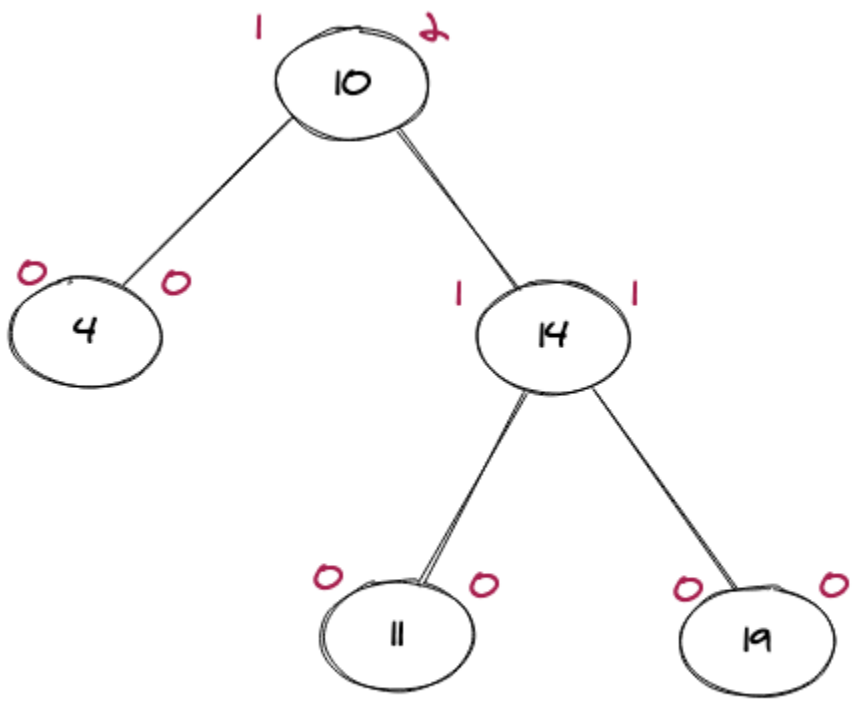
- It is a BST that is balanced in nature.
- It is self-balanced.
- It is not perfectly balanced.
- Every sub-tree is also an AVL Tree.

A pictorial representation of an AVL Tree is shown below:



AVL Tree

When explained, we can say that it is a Binary Search Tree that is height-balanced in nature. **Height balance** means that the difference between the left subtree height and the right subtree height for each node cannot be greater than 1(i.e **height(left) - height(right) <= 1**).



AVL Tree with Balance Factor

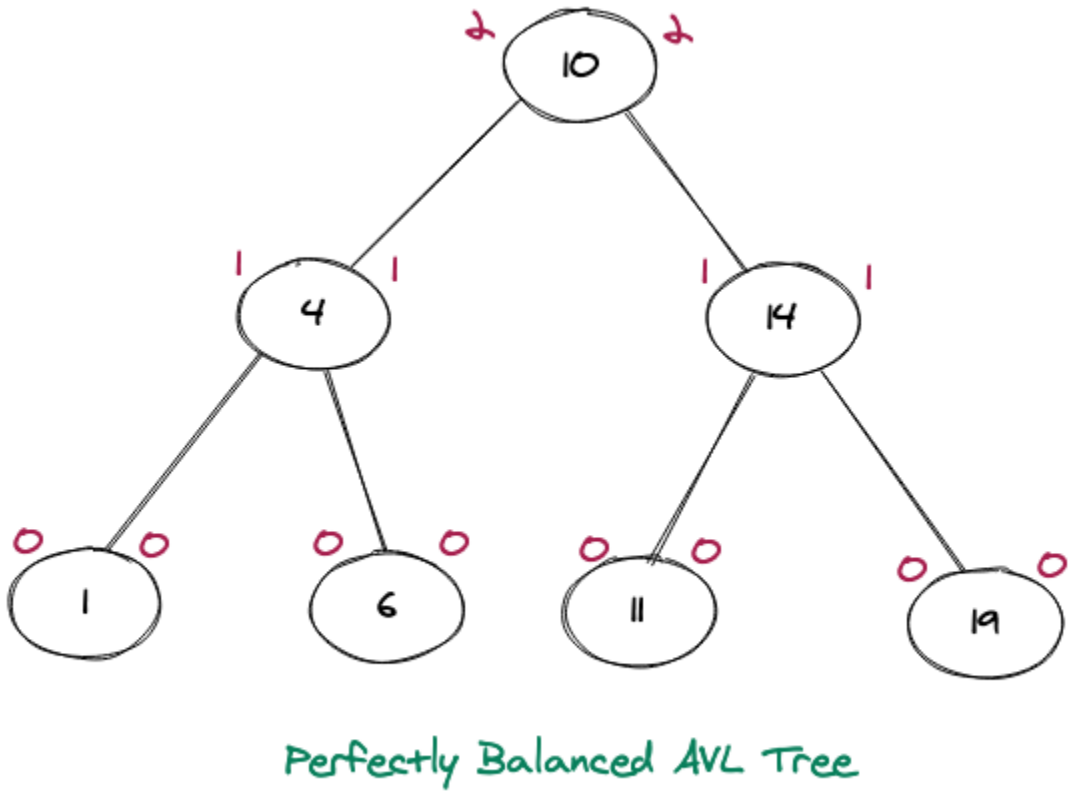
Notice that each node in the above BST contains two values, the left value denotes the height of the left subtree of that node and the right value denotes the height of the right subtree of that node, and it can be seen that at not a single node we have a value difference that is greater than 1, hence making it a balanced AVL Tree.

A **self-balanced tree** means that when we try to insert any element in this BST and if it violates the balancing factor of any node, then it dynamically rotates itself accordingly to make itself self-

balanced. It is also well known that AVL trees were the first well known dynamically balanced trees that were proposed.

It is not perfectly balanced, where the perfect balance implies the fact that the **height between the right subtree and the left subtree** of each node is equal to **0**.

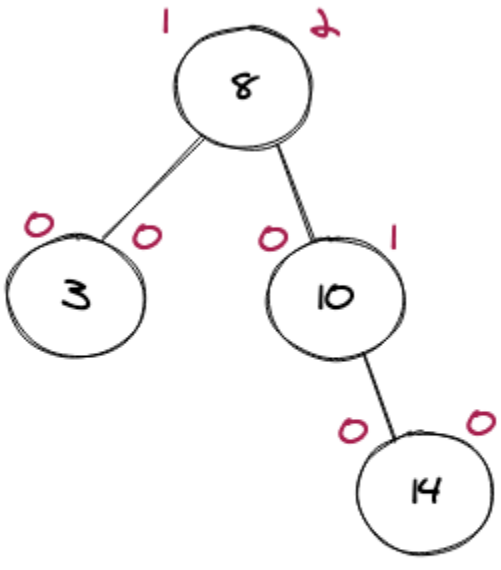
A pictorial representation of a Perfectly Balanced AVL Tree is shown below:



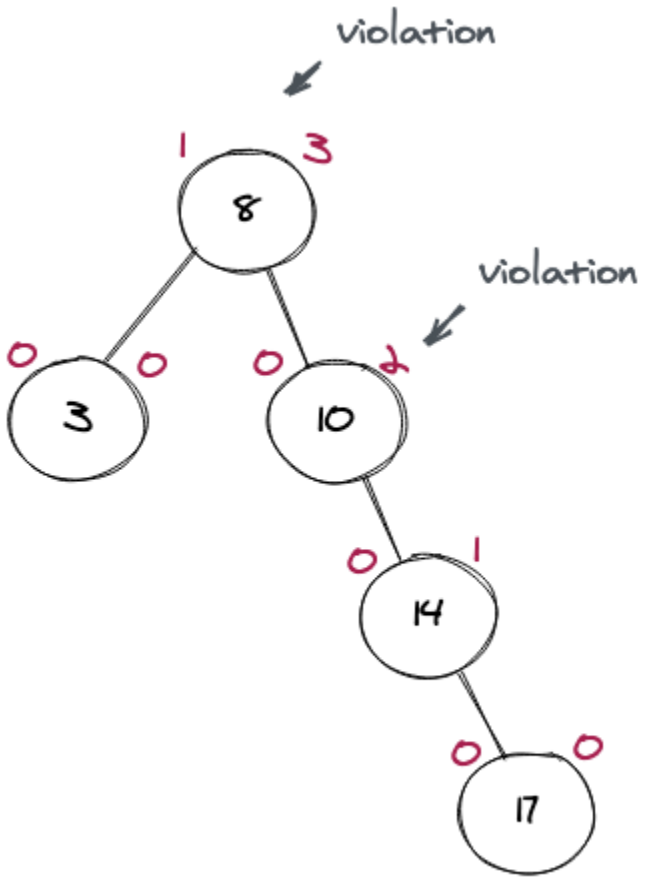
Perfectly Balanced AVL Tree

All the nodes of the above AVL Tree have the balance factor equal to 0(**height(left) - height(right) = 0**) making it a perfect AVL tree. It should also be noted that a perfect BST is the same as a perfect AVL tree. If we take a closer look at all the pictorial representations above, we can clearly see that each subtree of the AVL tree is also an AVL tree.

Now let's take a look at two more pictorial representations, where one of them is an AVL tree and one isn't.



AVL Tree



NOT an AVL Tree

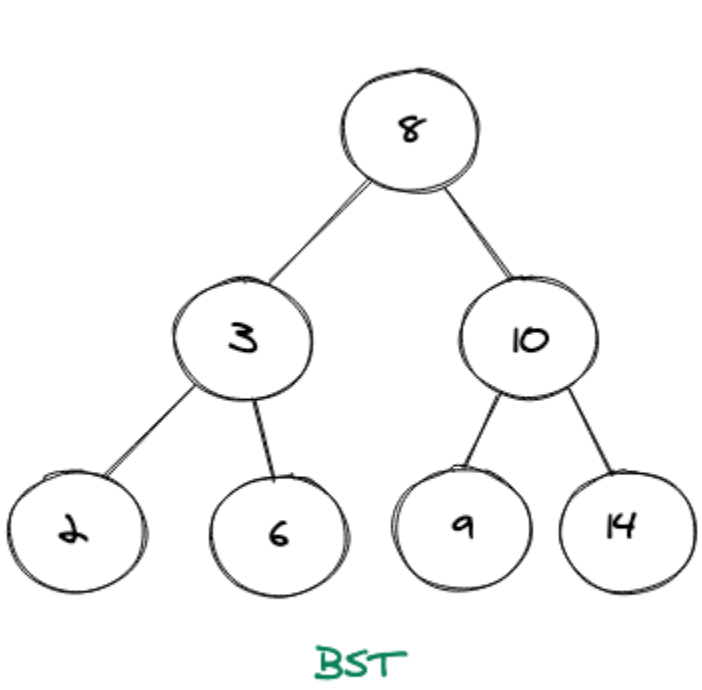
Though both the above trees are BST, only the left BST in the above representation is an AVL one as it height-balanced. In right BST, we have two violation nodes (i.e 8 and 10) where the balance

factor is more than 2 in both the cases. It should be noted that this tree can be rotated in a certain manner to make it an AVL tree.

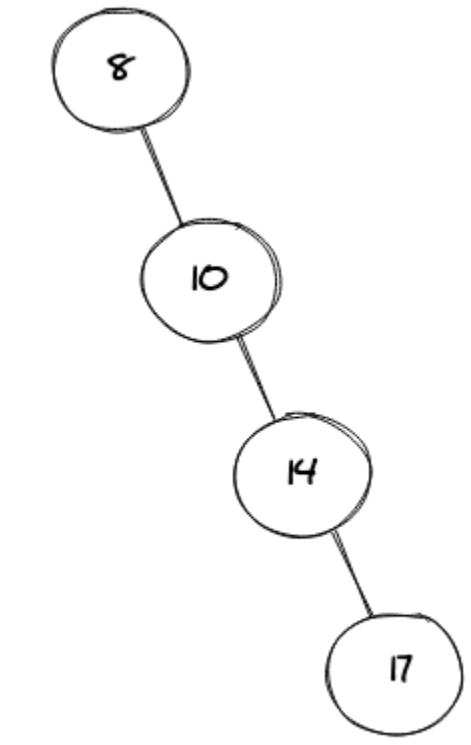
Why AVL Tree?

When we already had a similar data structure(i.e. BST's), why would we need a complex version of it?. The answer lies in the fact that BST has some limitations in certain scenarios that make some operations like searching, inserting costly (as costly as $O(n)$).

Consider the pictorial representation of two BST's below:



BST



Right Skewed BST

The height of the left BST is $O(\log n)$ and the height of the right BST is $O(n)$, thus the search operation time complexity for the left BST is $O(\log n)$ and for the right-skewed is $O(n)$ which is its worst case too. Hence, if we have such a skewed tree then there's no benefit of using a BST, as it is just like a linked list when it comes to time and space complexity.

That's why we needed a balanced BST's so that all the basic operations guarantee a time complexity of O(logN).

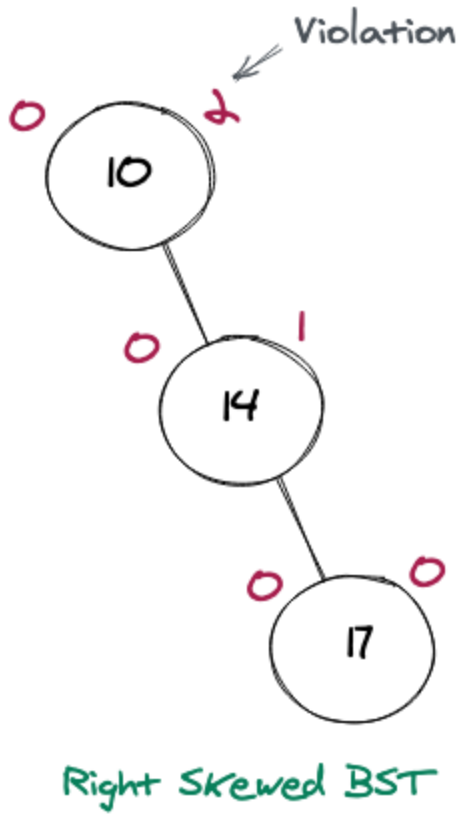
AVL Tree Rotations

If the AVL tree encounters any **violation of the balance factor** then it tries to do some rotations to make the tree balanced again. There are four rotations in total, we will look at each one of them. There mainly are:

- Left Rotation
- Right Rotation
- Left - Right Rotation
- Right - Left Rotation

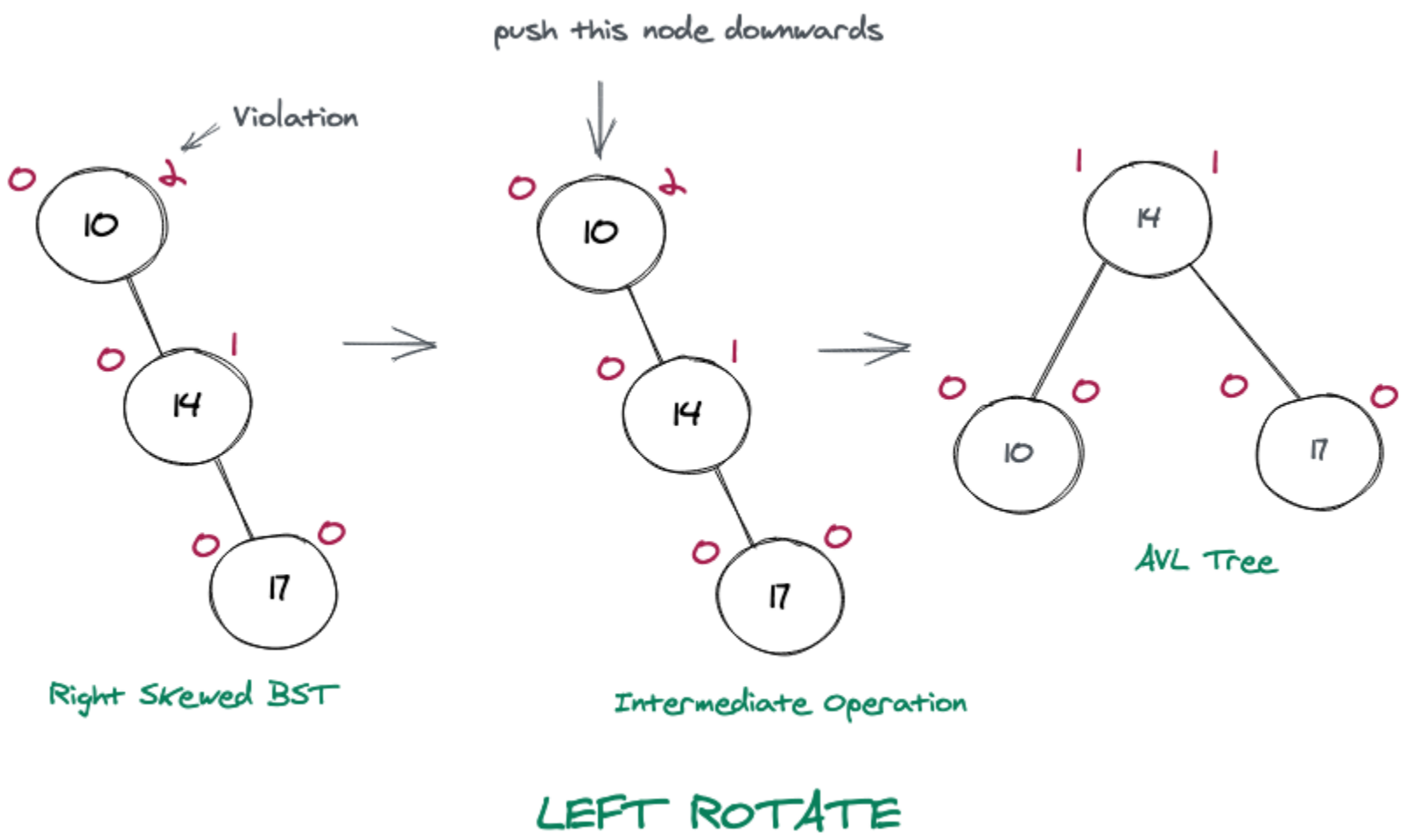
1. Left Rotation

This rotation is performed when the violation occurs in the left subtree of the left child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



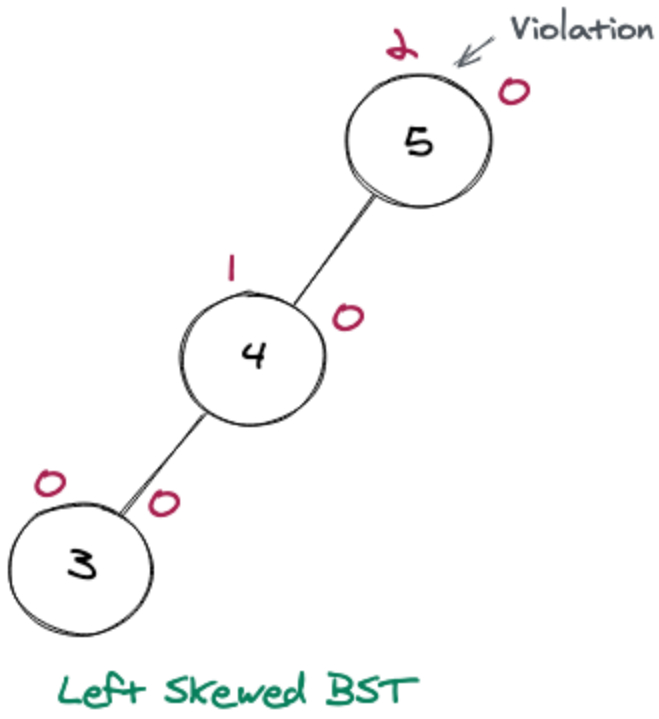
Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The tree is heavy on the right and the node that causes the issue is 17 as if we remove that one, we would have a balanced BST (i.e. AVL). Since it is heavily skewed on the right, we will do a left-rotation to make it balanced.

Consider the pictorial representation of making the above right skewed tree into an AVL tree.



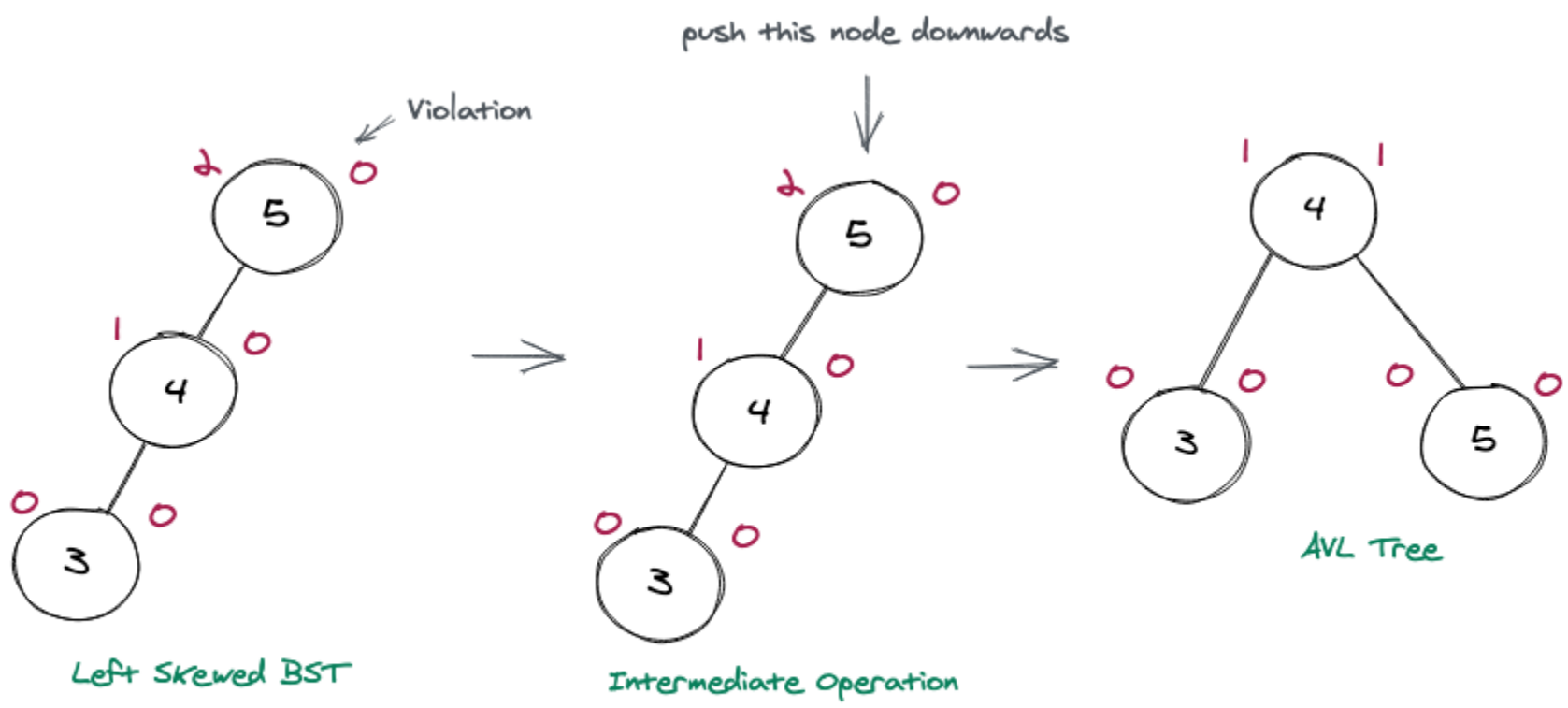
2. Right Rotation

This rotation is performed when the violation occurs in the right subtree of the right child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The tree is heavy on the left and the node that causes the issue is 3 as if we remove that one, we would have a balanced BST (i.e. AVL). Since it is heavily skewed on the left, we will do a right-rotation to make it balanced.

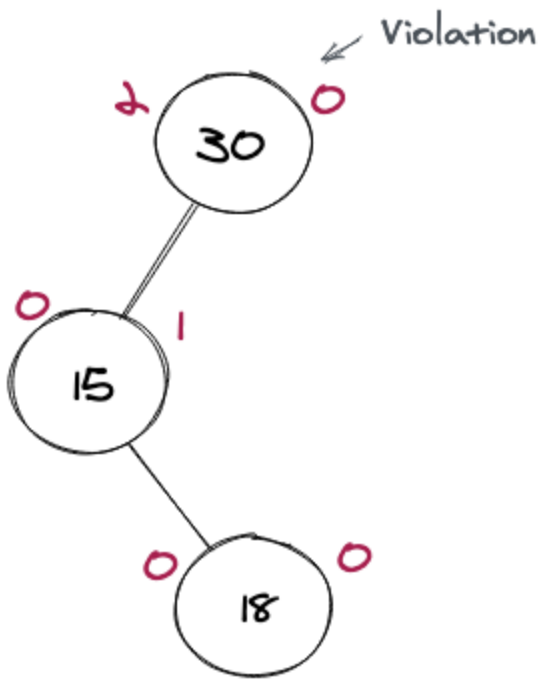
Consider the pictorial representation of making the above left-skewed tree into an AVL tree.



RIGHT ROTATE

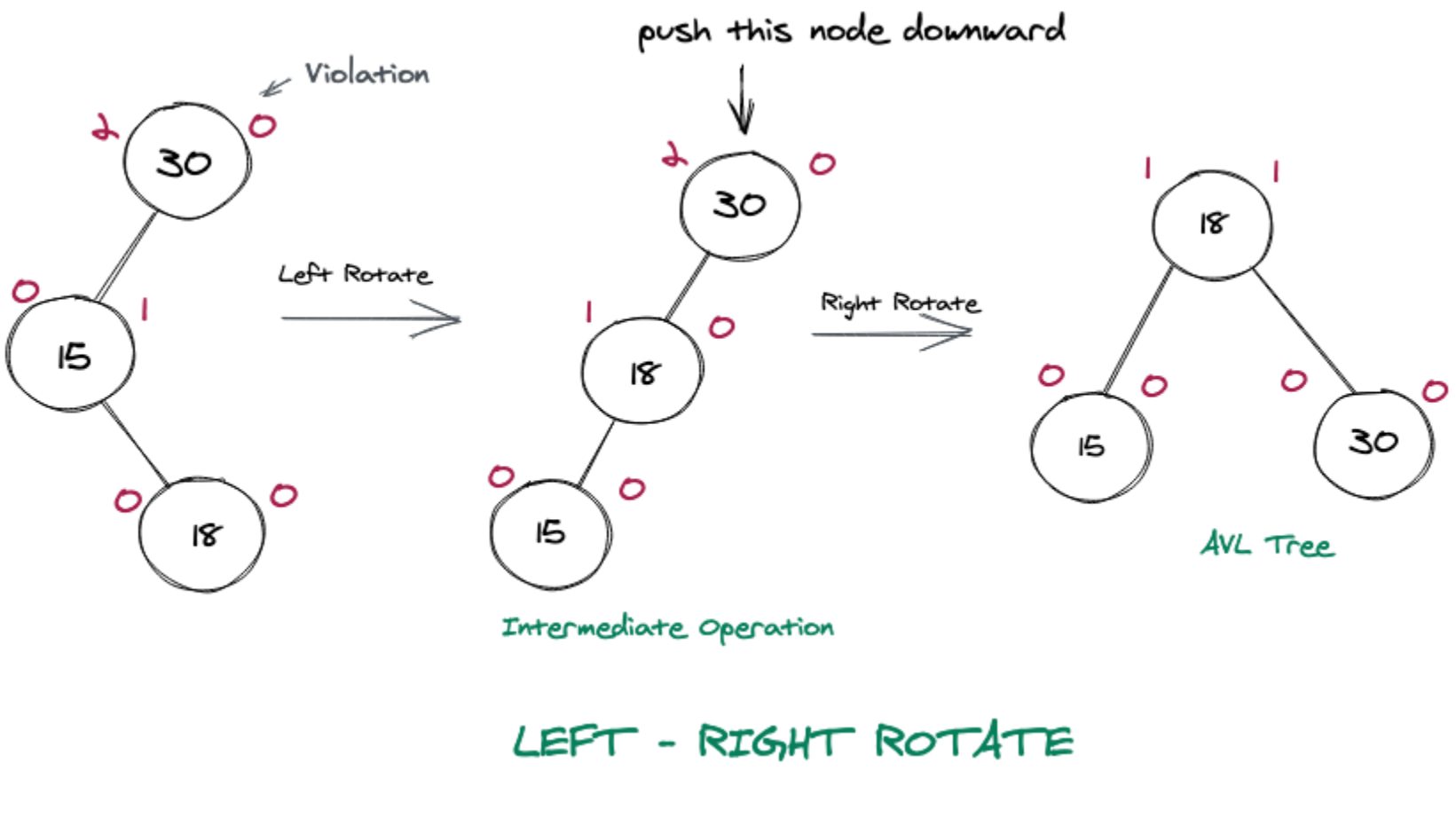
3. Left - Right Rotation

This rotation is performed when the violation occurs in the right subtree of the left child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



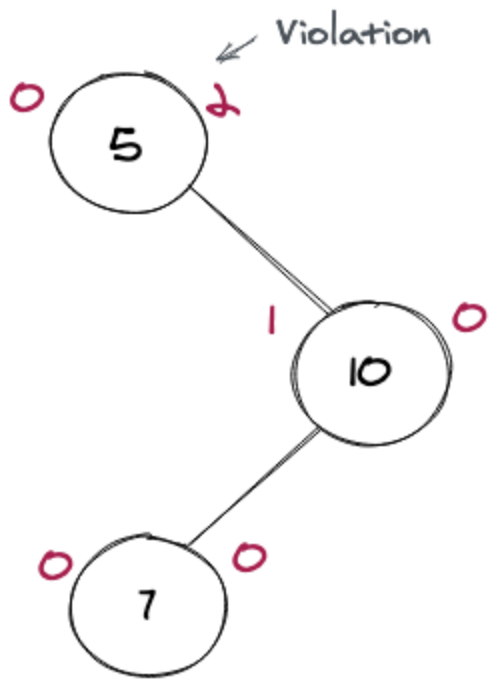
Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The node that causes the issue is 18 as if we remove that one, we would have a balanced BST (i.e. AVL). Since the issue is because of the right child of the left subtree, we will do a Left-Right rotation to make it balanced.

Consider the pictorial representation of making the above tree into an AVL tree.



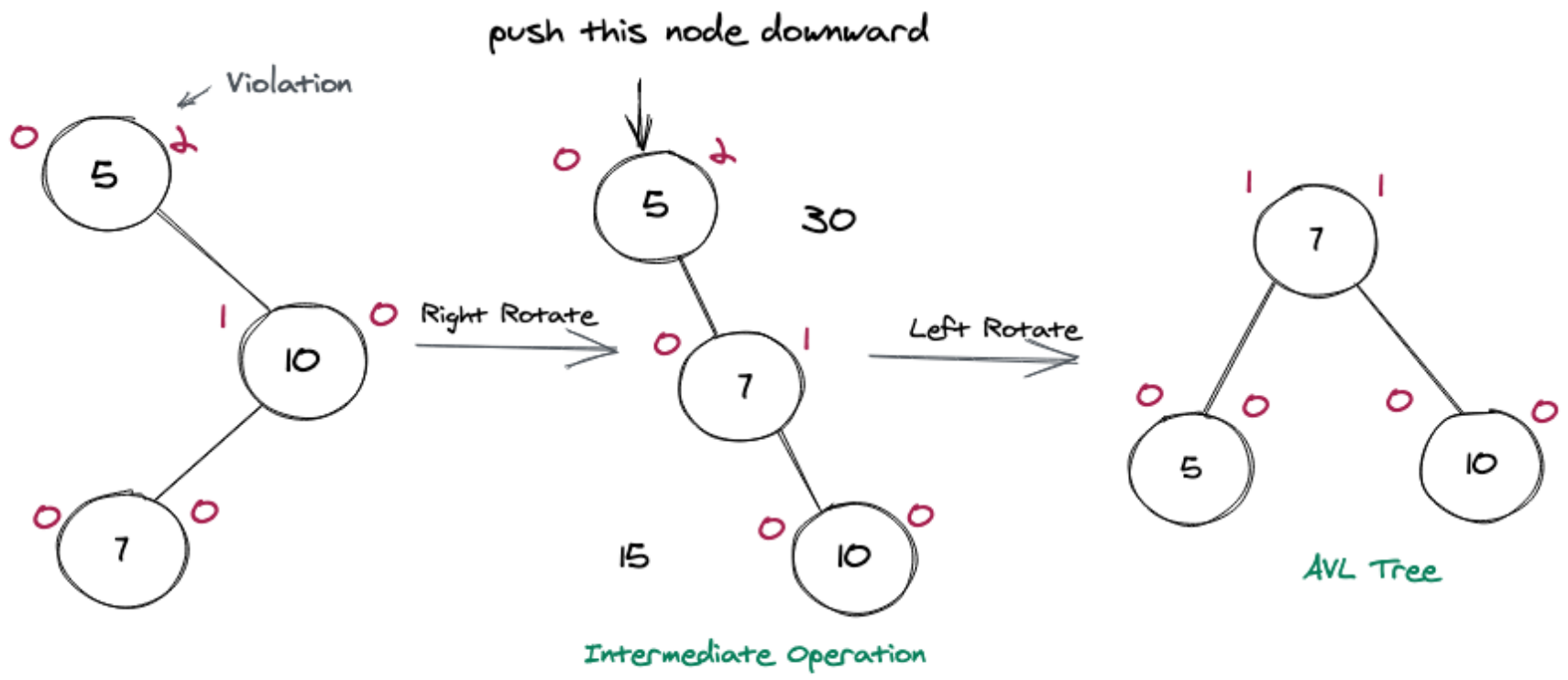
4. Right - Left Rotation

This rotation is performed when the violation occurs in the left subtree of the right child of the violating node. Consider the pictorial representation of a BST below that is not balanced, in order to make it so, we will do some rotations.



Clearly this is not an AVL tree, but consider the node where the violation occurs, we take this node as a pivot to do our rotations. The node that causes the issue is 7 as if we remove that one, we would have a balanced BST (i.e. AVL). Since the issue is because of the left child of the right subtree, we will do a Right-Left rotation to make it balanced.

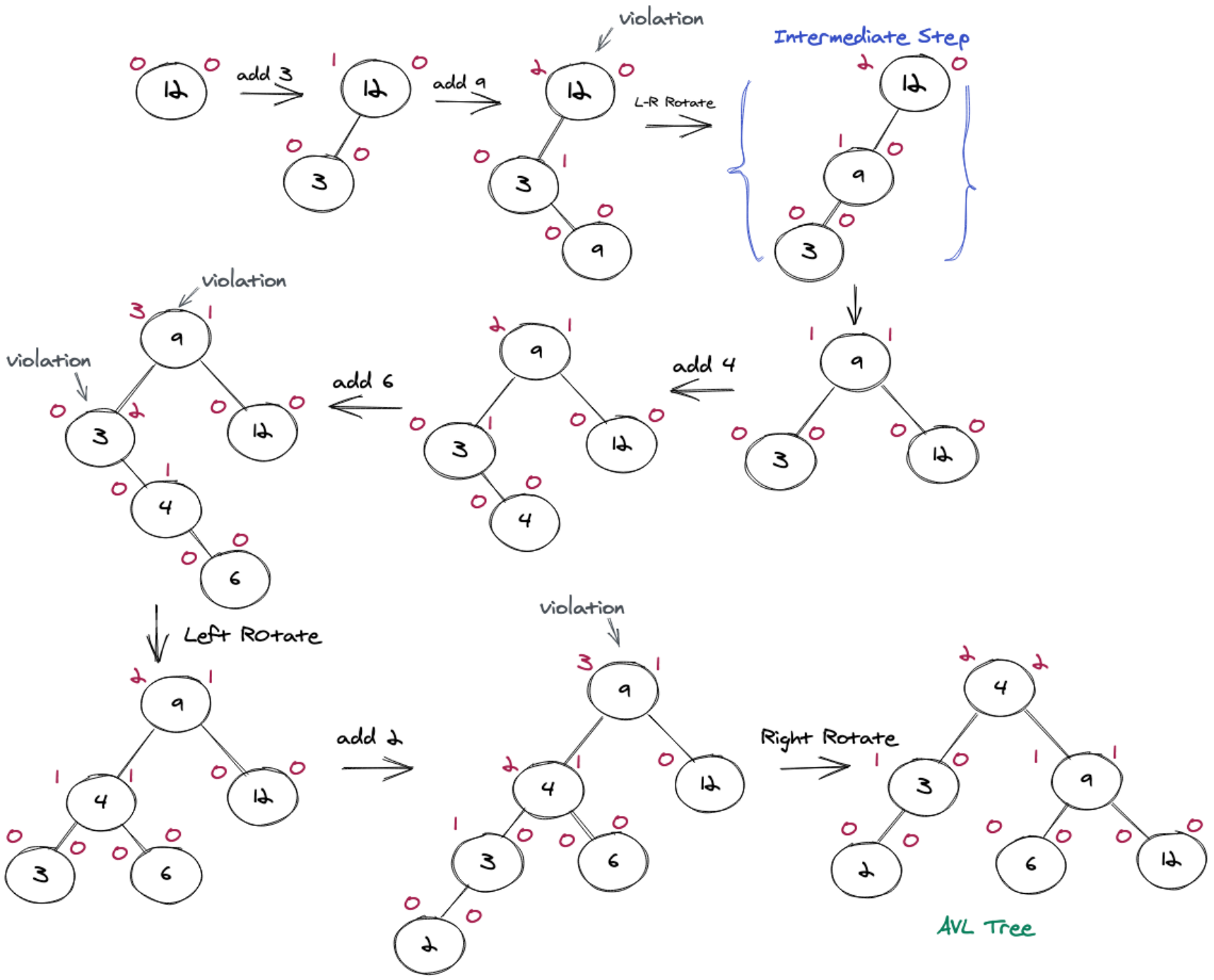
Consider the pictorial representation of making the above tree into an AVL tree.



AVL Tree Insertion

Let's take one array set of elements and build an AVL tree of these elements. Let, $\text{nums} = [12, 3, 9, 4, 6, 2]$

The complete step by step process of making an AVL tree (with rotations) is shown below.



INSERTING, IN AVL TREE

Key Points:

- An AVL tree with N number of nodes can have a minimum height of floor(logN) base 2.
- The height of an AVL tree with N number of nodes cannot exceed 1.44(logN) base 2.
- The maximum number of nodes in an AVL tree with height H can be : 2^H+1 - 1
- Minimum number of nodes with height h of an AVL tree can be represented as : N(h) = N(h-1) + N(h-2) + 1 for n>2 where N(0) = 1 and N(1) = 2.

Conclusions

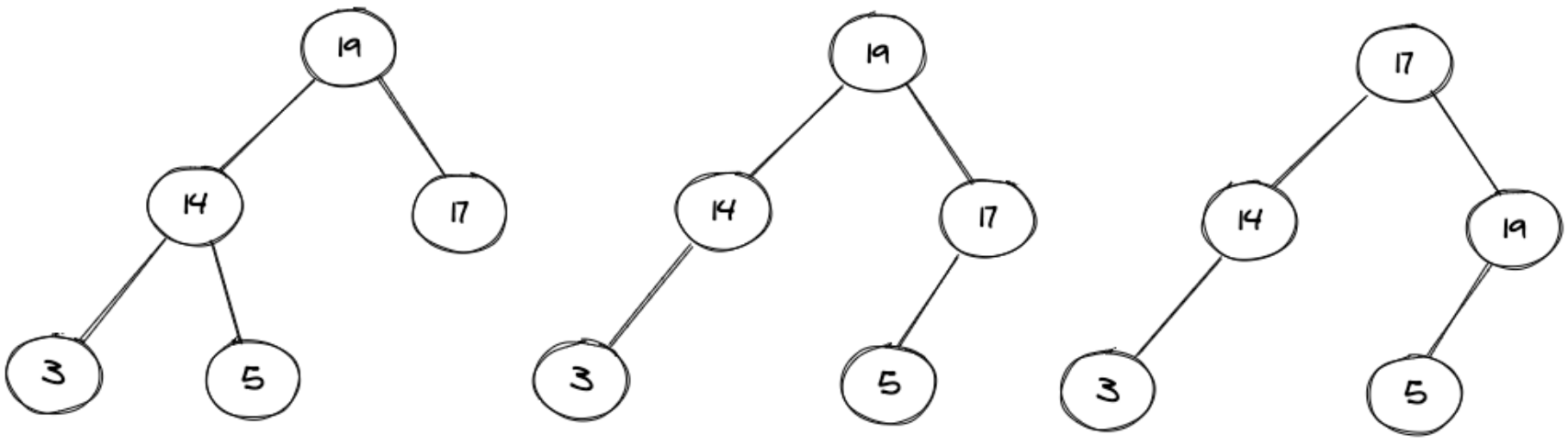
- We learned what an AVL tree is and why we need it.
- Then we learned the different type of Rotations that are possible in AVL tree to make it a balanced one.
- Followed by the rotations, we also did an insertion example in an AVL Tree.
- Lastly, we talked about different key points that we should remember with AVL

Heap Data Structure

A Heap is a special type of tree that follows two properties. These properties are :

- All leaves must be at `h` or `h-1` levels for some `h > 0`(complete binary tree property).
- The value of the node must be `>=` (or `<=`) the values of its children nodes, known as the heap property.

Consider the pictorial representation shown below:



In the pictures shown above, the leftmost tree denotes a heap (Max Heap) and the two tree to its right aren't heap as the middle tree violates the first heap property(not a complete binary tree) and the last tree from the left violates the second heap property(17 < 19).

Types of Heap

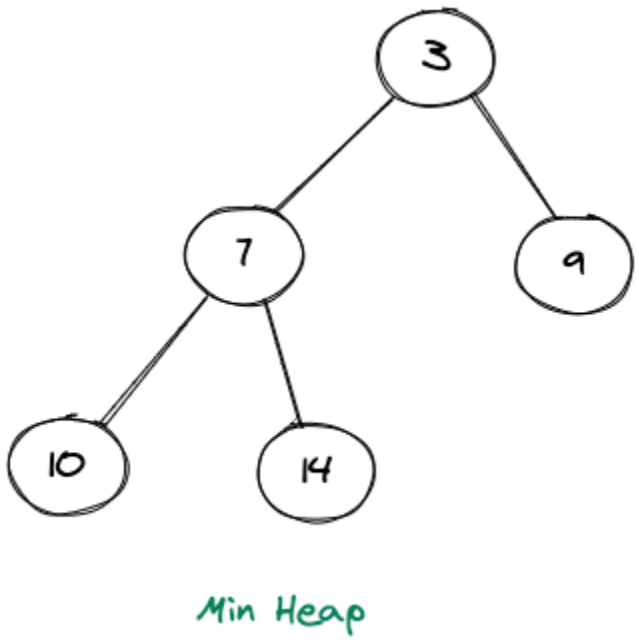
If we consider the properties of a heap, then we can have two types of heaps. These mainly are:

- Min Heap
- Max Heap

Min Heap

In this heap, the value of a node must be less than or equal to the values of its children nodes.

Consider the pictorial representation of a Min Heap below:

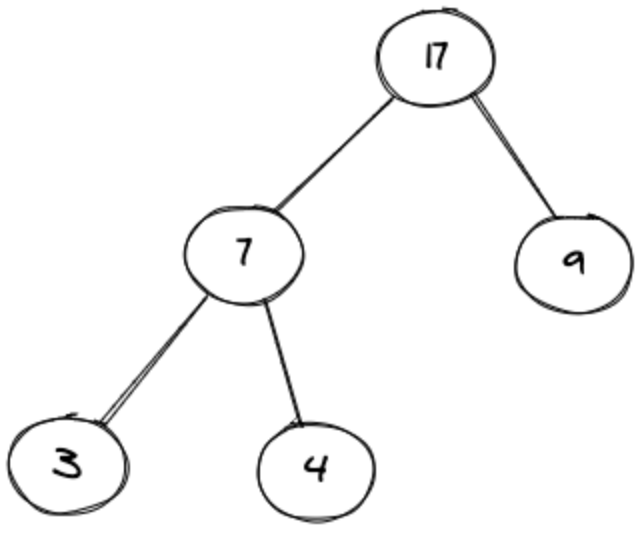


It can be clearly seen that the value of any node in the above heap is always less than the value of its children nodes.

Max Heap

In this heap, the value of a node must be greater than or equal to the values of its children nodes.

Consider the pictorial representation of a Max Heap below:



Max Heap

It can be clearly seen that the value of any node in the above heap is always greater than the value of its children nodes.

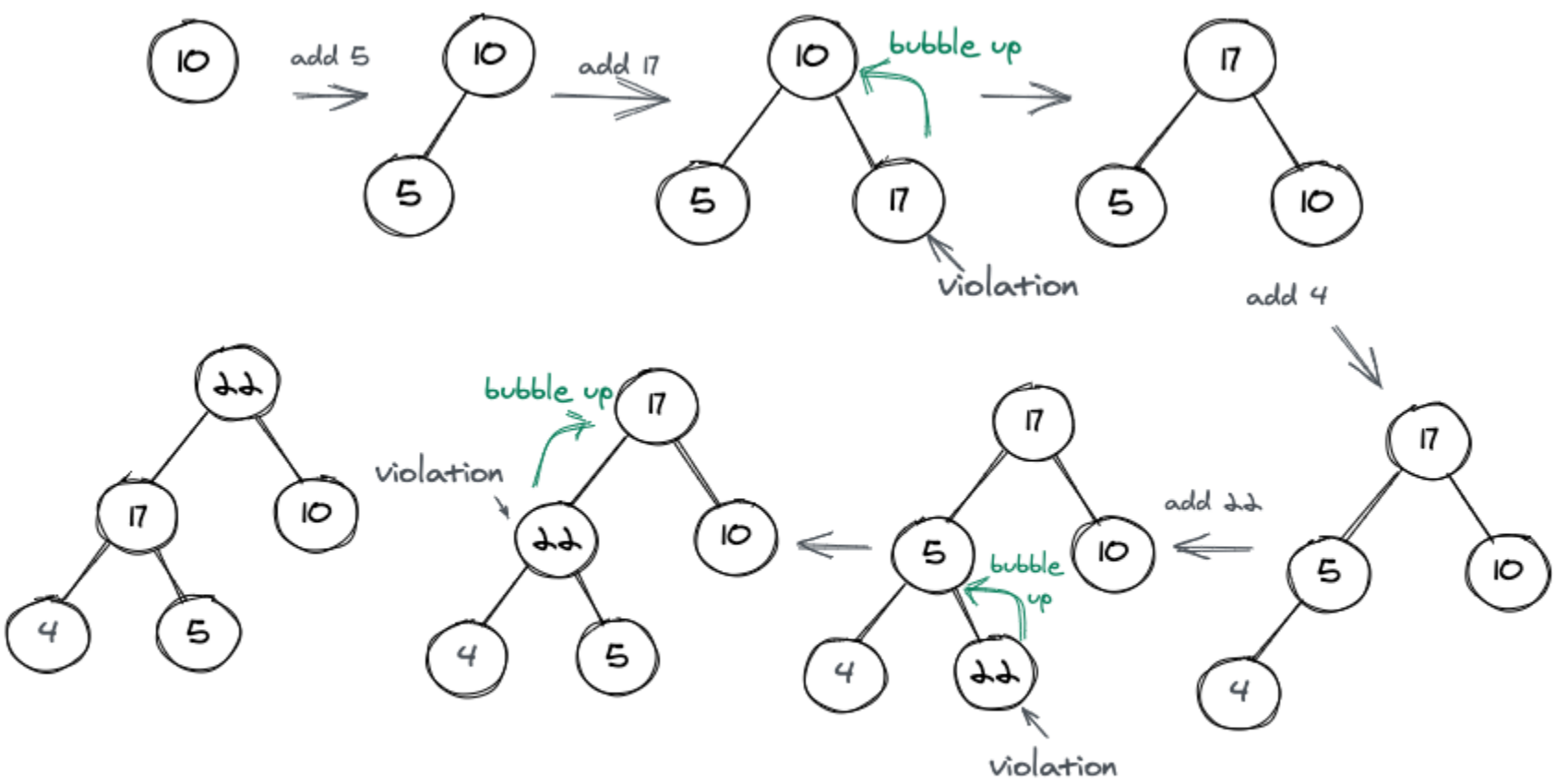
Building a Heap

Let's look at some operations like inserting an element in a heap or deleting an element from a heap.

1. Inserting an Element :

- First increase the heap size by 1.
- Then insert the new element at the available position(leftmost position ? Last level).
- Heapify the element from the bottom to the top(bubble up).

Building a heap includes adding elements into the heap. Let us consider an array of elements, namely nums = [10,5,17,4,22]. We want to make a Max Heap out of these elements, and the way we do that is shown in the pictorial representation below.

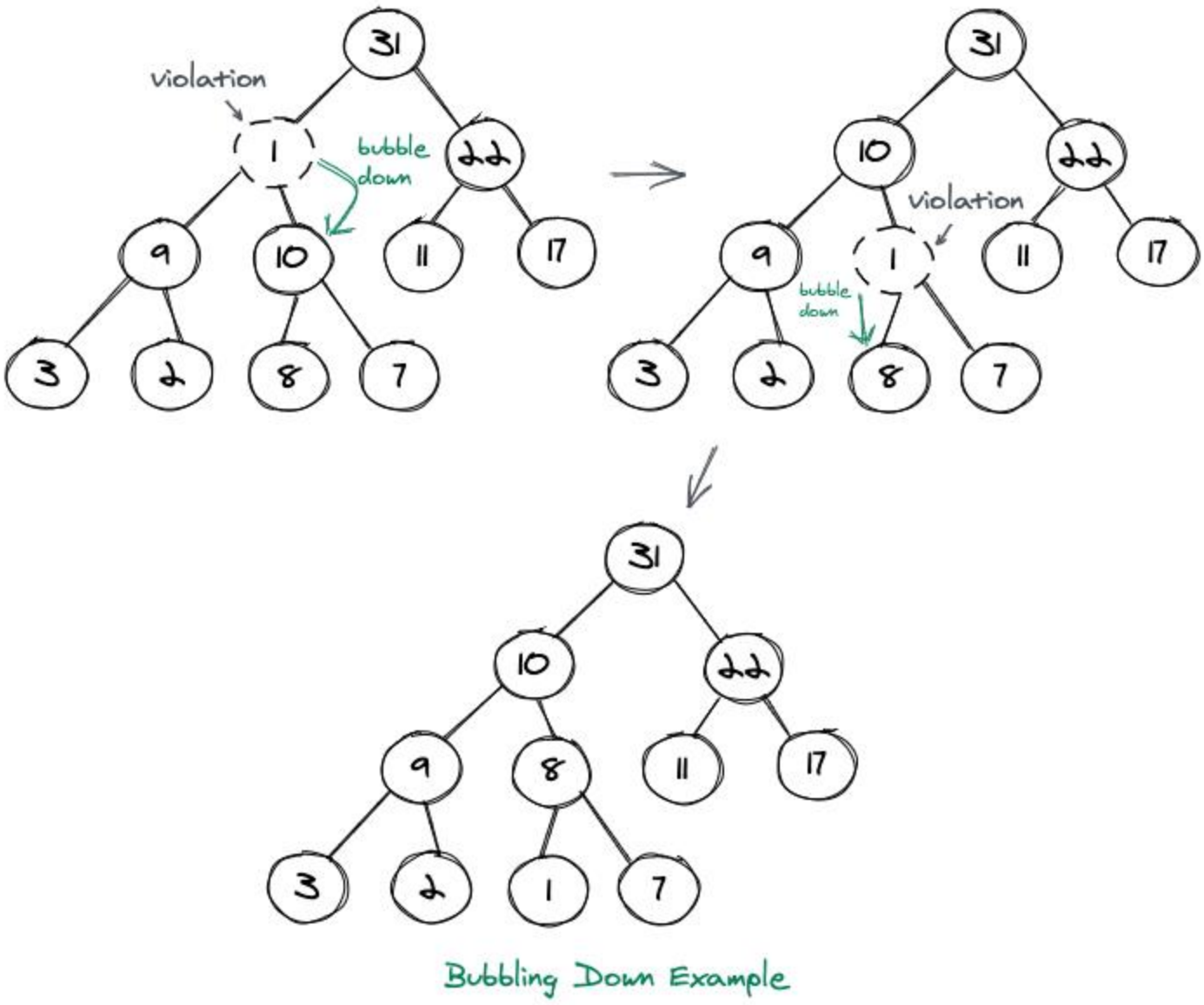


Building a Heap

Whenever we add an element while building a heap, it is quite possible that it will violate one of the properties. We take care of the first heap property by simply filling each level with maximum nodes and then when we move on to the next level, we fill the left child first and then the right child. But it is still possible that we have a violation regarding the second heap property, in that case we simply bubble up the current element that we have inserted and try to find the right position of it in the heap. Bubbling up includes swapping the current element with its parent till the heap is a max heap. In case of a min heap, we do the same procedure while adding an element to the heap.

The above representation shows three violations in total(17, 22, 22) and in all these cases we have basically swapped the current node with the parent node, hence bubbling up. It can also be noted that this process of bubbling up is also known as sift up.

Now let us look at another example, where we bubble down. Consider the pictorial representation of a tree(not heap) shown below:



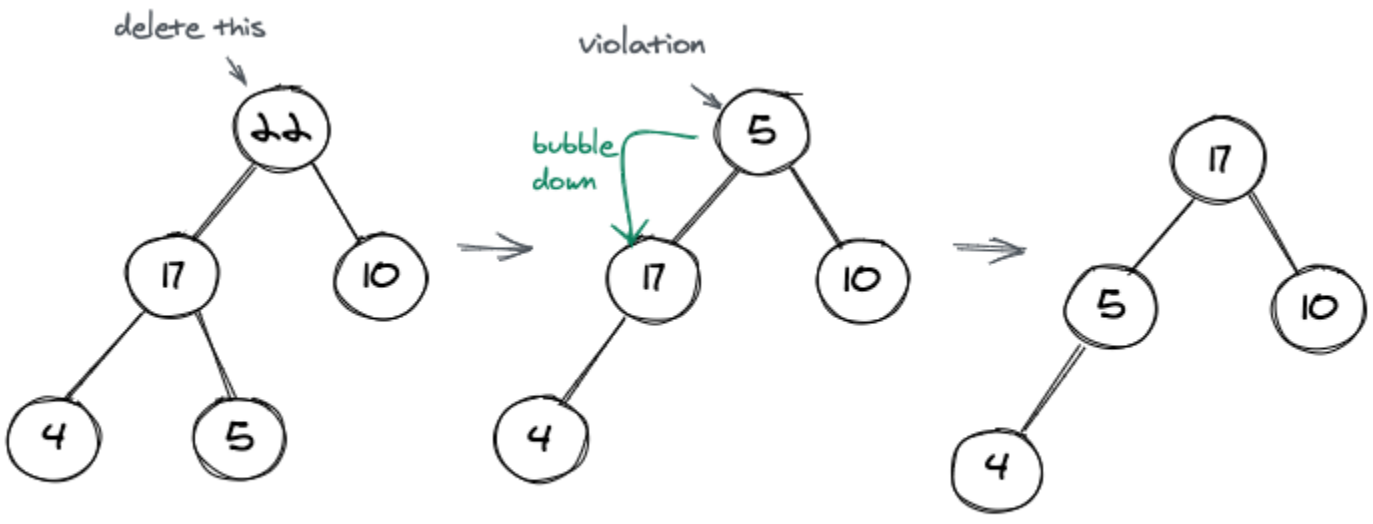
Bubbling Down Example

2. Deleting an Element :

- Copy the first element of the heap(root) into some variable
- Place the last element of the heap in the root's position
- Bubble Down to make it a valid heap

Whenever we are deleting an element, we simply delete the root element and replace it with the last element of the heap(the rightmost child of last level). We do that as we simply want to maintain the first property, as if we take any other element out of the heap we won't have a valid complete binary tree. We then place this node at the root, and it might be possible that this node will not satisfy the second property of the heap, and hence we bubble down to make it a valid heap. It can also be noted that this process of bubbling down is also known as sift down.

Consider the pictorial representation shown below:



Deletion in Heap

Applications of Binary Heaps

- Binary heaps are used in a famous sorting algorithm known as Heap sort.
- Binary heaps are also the main reason of implementing priority queues, as because of them the several priority queue operations like add(), remove() etc gets a time complexity of O(n).
- They are also the most preferred choice for solving Kth smallest / Kth Largest element questions.

Heap: Time Complexity Analysis

Let's see the time complexity for various operations of heap.

1. Inserting an Element:

Inserting an element in heap includes inserting it at the leaf level and then bubbling it up if it is somehow violating any property of the heap. We know that the heap is a complete binary tree, and the height of a complete binary tree is ($\log N$) where N represents the number of elements in the tree. So, if we consider the worst case scenario where we might have to swap this newly inserted node to the very top, we will have 1 swap at each level of the tree, hence we will require log N swaps. Hence, the worst case time complexity of Inserting an element in a binary heap is: O(log N)

2. Deleting an Element:

Deleting an element from a heap includes removing the root node and then swapping it with the last node of the last level, and then if this new root node violates any heap property, we need to swap it with the child node, until the tree is a valid binary heap. Since, in worst case scenarios we might have to swap this new root node with node at the lower levels to the very bottom(leaf level), which in turn means the height of the tree, the time complexity of deleting the node from the binary heap thus in turn is: $O(\log N)$.

3. Get Min/Max Element:

Getting the max(or min) element in a binary heap is simply a constant time operation, as we know that if it is a min heap the minimum will be the root node, and similarly in case of a max heap the maximum element will also be the root node. So, time complexity of extracting Min/Max is: $O(1)$.

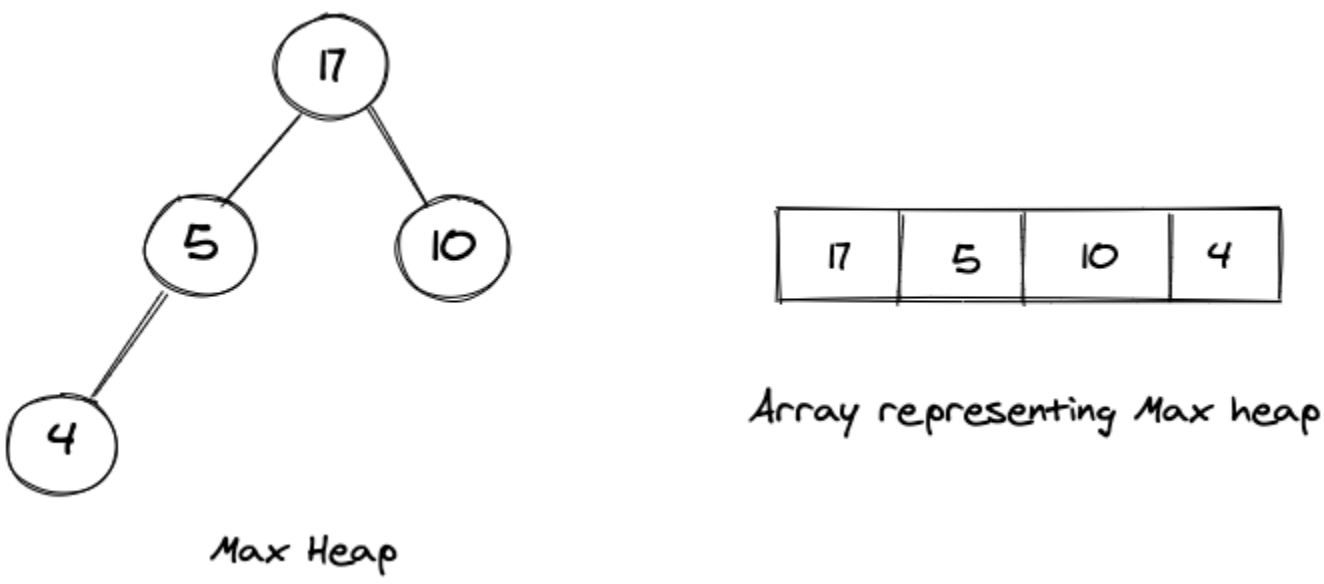
Conclusions

- We learned what a heap is, followed by the explanation of what min/max heaps are.
- Then we learned how to do insertion into a heap, followed by deletion of an element from the heap.
- Then we talked about the applications of binary heaps.
- Finally, we discussed the time complexity of heaps.

Implementing Heaps

In the previous article we learned all about heaps. It is time to implement them. Arrays are a preferred choice for implementing heaps because of the heap's complete binary tree property, we can be assured that no wastage of array locations will be there.

Let us consider a binary heap shown below, this heap can be represented by an array also shown in the pictorial representation below.



Implementing Heap

Let's see how we can implement a Heap data structure.

1. Structure of Heap

We need an array to store the elements of the heap, besides that we will also have a variable that represents the size of the heap. Consider the code snippet below:

```
private int[] items = new int[10];

private int size;
```

Copy

The initial capacity can be dynamic also, but here I have chosen a fixed length of 10 to represent the same.

2. Inserting into a Heap

When we want to insert an item into a heap, we will simply insert it into the array and then do a `bubbleUp()` operation if it doesn't satisfy any of the heap properties. The code will look something like this:

```
private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();
```

```
}
```

Copy

A case might arrive where the heap is full, and then if we try to insert any more items into it, it should return an Exception. The `isFull()` method looks like this:

```
private boolean isFull(){ return size == items.length; }
```

Copy

After this `isFull()` call returns false, we move ahead and insert the item into our heap, and increase the size of the array as well. Now, the most important part is to correct the position of this element that we have inserted in the heap. We do this using `bubbleUp()` method. The code of `bubbleUp()` method is shown below:

```
private void bubbleUp(){

        int index = size - 1;

        while(index > 0 && items[index] > items[parent(index)]){

            swap(index,parent(index));

            index = parent(index);

        }

}
```

Copy

In the above method we are taking the index of the element we just inserted and then swapping it with the parent element in case this element is bigger than the parent element, also we are updating the index so that we don't pick the wrong element. The `swap()` method looks like this:

```
private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

}
```

Copy

It should also be noted that we are calling a parent method in our `bubbleUp()` method, and that parent method returns us the parent of a current index. The code of this method looks like this:

```
private int parent(int index){ return (index - 1)/2; }
```

Copy

The above methods are all we need to insert an element into a binary heap.

3. Removing from a Heap

Removing an element from a heap is a bit tricky process as when compared to inserting into a heap, as in this we need to `bubbleDown()` and this process of bubbling down involves checking the

parent element about its validity, also methods to insert into the tree as left as possible to maintain the complete binary tree property.

The `remove()` method looks like this:

```
private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

}
```

Copy

The first check is to make sure that we are not trying to remove anything from an empty heap. Then we insert the element at the start, and then `bubbleDown()` the element to it's correct position. The `bubbleDown()` method looks like this:

```
private void bubbleDown(){

        int index = 0;

        while(index <= size && !isValidParent(index)){

            int largerChildIndex = largerChildIndex(index);

            swap(index,largerChildIndex);

            index = largerChildIndex;

        }

}
```

Copy

We are checking that the index we have should be less than the size and also making sure that the parent node should not be a valid (should be less than the child values and other checks). The `largerChildIndex()` method return us the children item with which we will replace this current item while bubbling downwards. The code for `largerChildIndex()` looks like this:

```
private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

}
```

Copy

The code for isValidParent() method is shown below:

```
private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

}
```

Copy

In the above code snippet we are just making sure that the parent is valid, and also if the left child is null then we return true. The leftChild(), rightChild(), hasLeftChild(), hasRightChild() methods looks like this:

```
private int leftChild(int index){

        return items[leftChildIndex(index)];

}



private int rightChild(int index){

        return items[rightChildIndex(index)];

}



private boolean hasLeftChild(int index){

        return leftChildIndex(index) <= size;

}



private boolean hasRightChild(int index){

        return rightChildIndex(index) <= size;

}
```

Copy

The above method calls the methods to get the left and right children of a binary heap element, these are:

```
private int leftChildIndex(int index){

        return 2*index + 1;

}



private int rightChildIndex(int index){

        return 2*index + 2;

}
```

Copy

Now we are done with all the methods we need to insert or delete an element from the heap. The complete code is given below:

```
public class Heap {


    private int[] items = new int[10];

    private int size;


    private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();

    }


    private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

    }
```

```
    private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

    }



    private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

    }


    private int leftChild(int index){

        return items[leftChildIndex(index)];

    }


    private int rightChild(int index){

        return items[rightChildIndex(index)];

    }


    private int leftChildIndex(int index){

        return 2*index + 1;

    }


    private int rightChildIndex(int index){

        return 2*index + 2;

    }
```

```
    private boolean hasLeftChild(int index){
        return leftChildIndex(index) <= size;
    }

    private boolean hasRightChild(int index){
        return rightChildIndex(index) <= size;
    }

    private boolean isFull(){
        return size == items.length;
    }

    private boolean isEmpty(){
        return size == 0;
    }

    private void bubbleUp(){
        int index = size - 1;
        while(index > 0 && items[index] > items[parent(index)]){
            swap(index,parent(index));
            index = parent(index);
        }
    }

    private void bubbleDown(){
        int index = 0;
        while(index <= size && !isValidParent(index)){
            int largerChildIndex = largerChildIndex(index);
            swap(index,largerChildIndex);
```

```
            index = largerChildIndex;

        }

    }


    private int parent(int index){

        return (index - 1)/2;

    }


    private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

    }


    private int getMax(){

        return items[0];

    }


    public static void main(String[] args) {

        Heap heap = new Heap();

        heap.insert(10);

         heap.insert(5);

         heap.insert(17);

         heap.insert(4);

         heap.insert(22);

        /// heap.remove();

         System.out.println("Done!");

    }

}
```

Copy

Conclusion

- We learned how to implement heaps, mainly the **bubbleUp()** and **bubbleDown()** methods.

Trie Data Structure - Explained with Examples

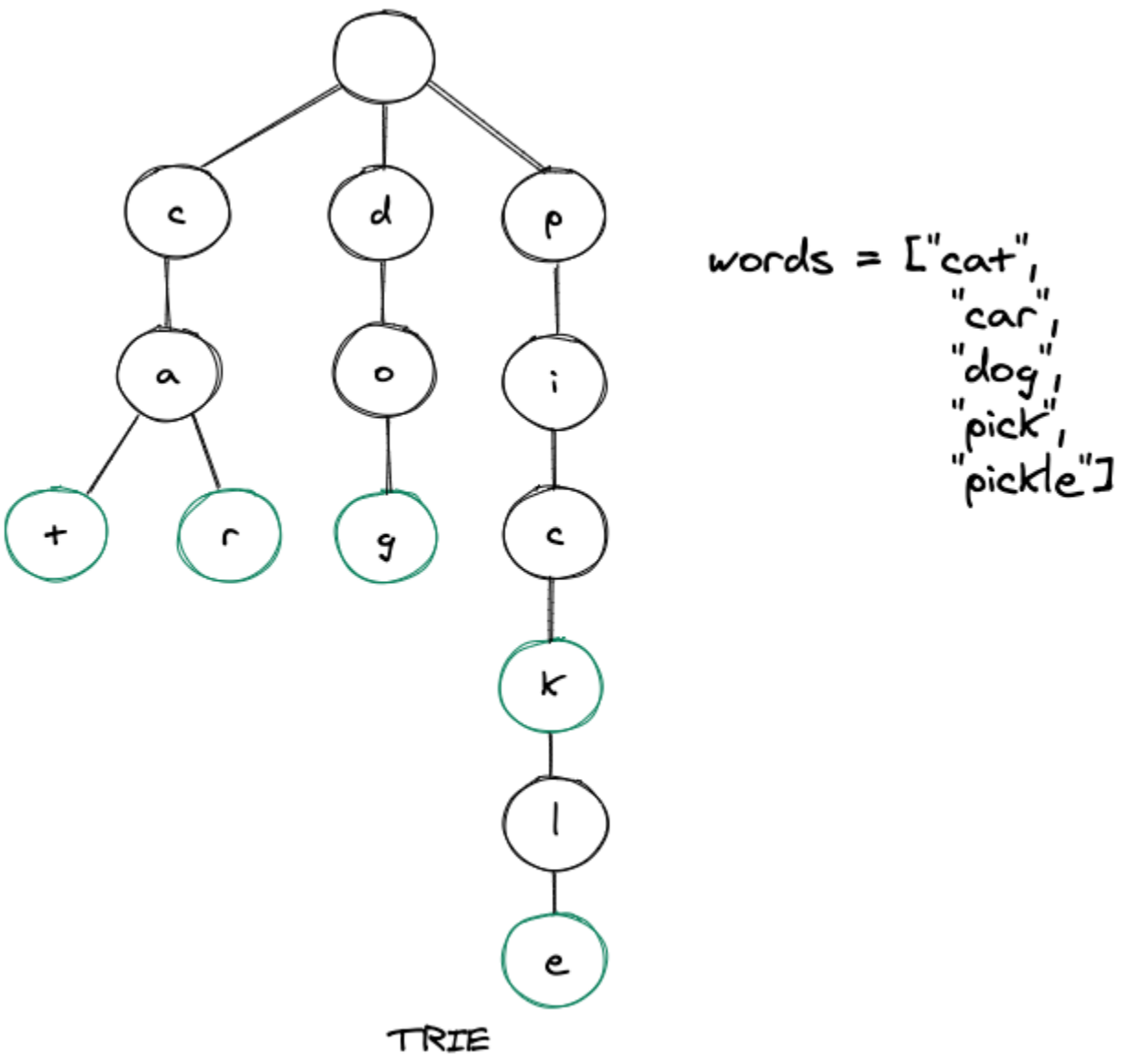
A **Trie is an advanced data structure** that is sometimes also known as **prefix tree or digital tree**. It is a tree that stores the data in an ordered and efficient way. We generally **use trie's to store strings**. Each node of a trie can have as many as 26 references (pointers).

Each node of a trie consists of two things:

- A character
- A boolean value is used to implement whether this character represents the end of the word.

Tries in general are used to store English characters, hence each character can have 26 references. Nodes in a trie do not store entire keys, instead, they store a part of the key(usually a character of the string). When we traverse down from the root node to the leaf node, we can build the key from these small parts of the key.

Let's build a trie by inserting some words in it. Below is a pictorial representation of the same, we have 5 words, and then we are inserting these words one by one in our trie.



As it can be seen in the image above, the key(words) can be formed as we traverse down from the root node to the leaf nodes. It can be noted that the green highlighted nodes, represents the `endOfWord` boolean value of a word which in turn means that this particular word is completed at this node. Also, the root node of a trie is empty so that it can refer to all the members of the alphabet the trie is using to store, and the children nodes of any node of a trie can have at most 26 references. Tries are not balanced in nature, unlike AVL trees.

Why use Trie Data Structure?

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally comes to our mind**. Though very efficient in nature but still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

- There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is `O(k)` where **k = length of the word**.
- It can take even less than `O(k)` time when the word is not there in a trie.

Implementing Tries:

We will implement the Trie data structure in Java language.

Trie Node Declaration:

```
class TrieNode {

    boolean isEndOfWord;

    TrieNode children[];



    public TrieNode(){

        isEndOfWord = false;

        children = new TrieNode[26];

    }

}
```

Copy
Note that we have two fields in the above **TrieNode** class as explained earlier, the boolean isEndOfWord keyword and an array of Trie nodes named children. Now let's initialize the root node of the trie class.

```
TrieNode root;



public Trie() {

    root = new TrieNode();

}
```

Copy
There are two key functions in a trie data structure, these are:

- **Search**
- **Insert**

Insert in trie:

When we insert a character(part of a key) into a trie, we start from the root node and then search for a reference, which corresponds to the first key character of the string whose character we are trying to insert in the trie. Two scenarios are possible:

- A reference exists, if, so then we traverse down the tree following the reference to the next children level.
- A reference does not exist, then we create a new node and refer it with parents reference matching the current key character. We repeat this step until we get to the last character of the key, then we mark the current node as an end node and the algorithm finishes.

Consider the code snippet below:

```
public void insert(String word) {

    TrieNode node = root;



    for (char c : word.toCharArray()) {

        if (node.children[c-'a'] == null) {

            node.children[c-'a'] = new TrieNode();

        }

        node = node.children[c-'a'];

    }
```

```
    node.isEndOfWord = true;

}
```

Copy

Search in trie:

A key in a trie is stored as a path that starts from the root node and it might go all the way to the leaf node or some intermediate node. If we want to search a key in a trie, we start with the root node and then traverses downwards if we get a reference match for the next character of the key we are searching, then there are two cases:

1. A reference of the next character exists, hence we move downwards following this link, and proceed to search for the next key character.
2. A reference does not exist for the next character. If there are no more characters of the key present and this character is marked as `isEndOfWord = true`, then we return **true**, implying that we have found the key. Otherwise, two more cases are possible, and in each of them we return **false**. These are:
    1. There are key characters left in the key, but we cannot traverse down as the path is terminated, hence the key doesn't exist.
    2. No characters in the key are left, but the last character is not marked as `isEndOfWord = false`. Therefore, the search key is just the prefix of the key we are trying to search in the trie.

Consider the code snippet below:

```
public boolean search(String word) {

    return isMatch(word, root, 0, true);

}



public boolean startsWith(String prefix) {

    return isMatch(prefix, root, 0, false);

}



public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {

    if (node == null)

        return false;



    if (index == s.length())

        return !isFullMatch || node.isEndOfWord;



    return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);

}
```

Copy
The method `startsWith()` is used to find if the desired key prefix is present in the trie or not. Also, both the `search()` and `startsWith()` methods make use of `isMatch()` method.

Entire Code:

```
class Trie {

    class TrieNode {
```

```
        boolean isEndOfWord;
        TrieNode children[];

        public TrieNode(){
            isEndOfWord = false;
            children = new TrieNode[26];
        }
    }

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;

        for (char c : word.toCharArray()) {
            if (node.children[c-'a'] == null) {
                node.children[c-'a'] = new TrieNode();
            }
            node = node.children[c-'a'];
        }
        node.isEndOfWord = true;
    }

    public boolean search(String word) {
        return isMatch(word, root, 0, true);
    }

    public boolean startsWith(String prefix) {
        return isMatch(prefix, root, 0, false);
```

```
    }


    public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {
        if (node == null)
            return false;

        if (index == s.length())
            return !isFullMatch || node.isEndOfWord;

        return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);
    }


    public static void main(String[] args){
        Trie trie = new Trie();
        trie.insert("cat");
        trie.insert("car");
        trie.insert("dog");
        trie.insert("pick");
        trie.insert("pickle");
        boolean isPresent = trie.search("cat");
        System.out.println(isPresent);
        isPresent = trie.search("picky");
        System.out.println(isPresent);
        isPresent = trie.startsWith("ca");
        System.out.println(isPresent);
        isPresent = trie.startsWith("pen");
        System.out.println(isPresent);


    }
}
```

Copy
The output of the above looks like this:

true

false

true

false

Trie Applications

- The most common use of tries in real world is the **autocomplete feature** that we get on a search engine(now everywhere else too). After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of strings are possible for the words we have typed in.
- Trie also helps in the case where we want to store additional information of a word, say the popularity of the word, which makes it so powerful. You might have seen that when you type **"foot"** on the search bar, you get **"football"** before anything say **"footpath"**. It is because **"football"** is a much popular word.
- Trie also has helped in checking the correct spellings of a word, as the path is similar for a slightly misspelled word.
- **String matching** is another case where tries to excel a lot.

Key Points

- The time complexity of creating a trie is `O(m*n)` where m = number of words in a trie and n = average length of each word.
- Inserting a node in a trie has a time complexity of `O(n)` where n = length of the word we are trying to insert.
- Inserting a node in a trie has a space complexity of `O(n)` where n = length of the word we are trying to insert.
- Time complexity for searching a key(word) in a trie is `O(n)` where n = length of the word we are searching.
- Space complexity for searching a key(word) in a trie is `O(1)`.
- Searching for a prefix of a key(word) also has a time complexity of `O(n)` and space complexity of `O(1)`.

Conclusions

- We learned what a Trie is, and why do we need one.
- We also implemented trie in Java, where insert and search were the main methods implemented.
- We then talked about the applications of Trie data structure in real world, followed by several key points that we should also remember about trie's.

B Trees (M-way Trees) Data Structure

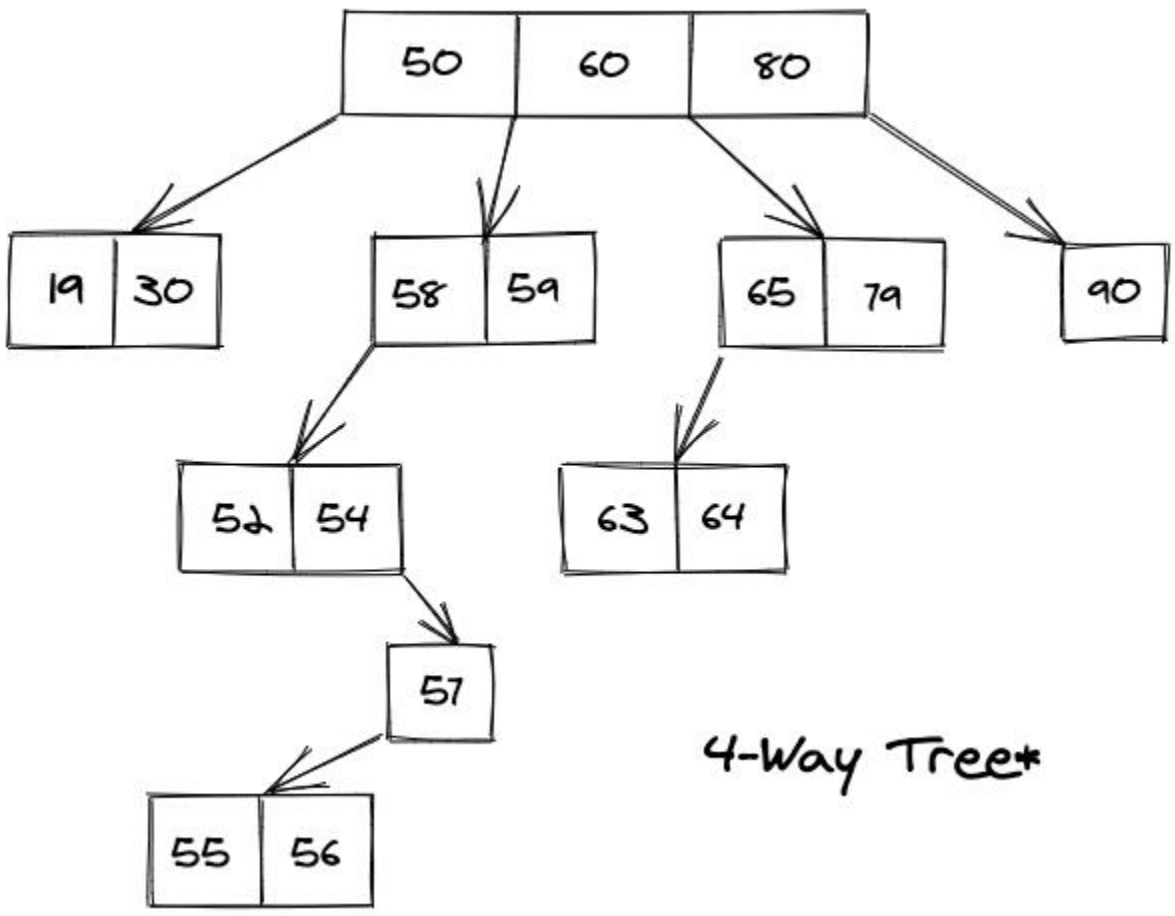
A B-Tree is a special type of M-way search tree.

M-way Trees

Before learning about B-Trees we need to know what M-way trees are, and how B-tree is a special type of M-way tree. An M-way(multi-way) tree is a tree that has the following properties:

- Each node in the tree can have at most **m** children.
- Nodes in the tree have at most **(m-1)** key fields and pointers(references) to the children.

Consider the pictorial representation shown below of an **M-way tree**.



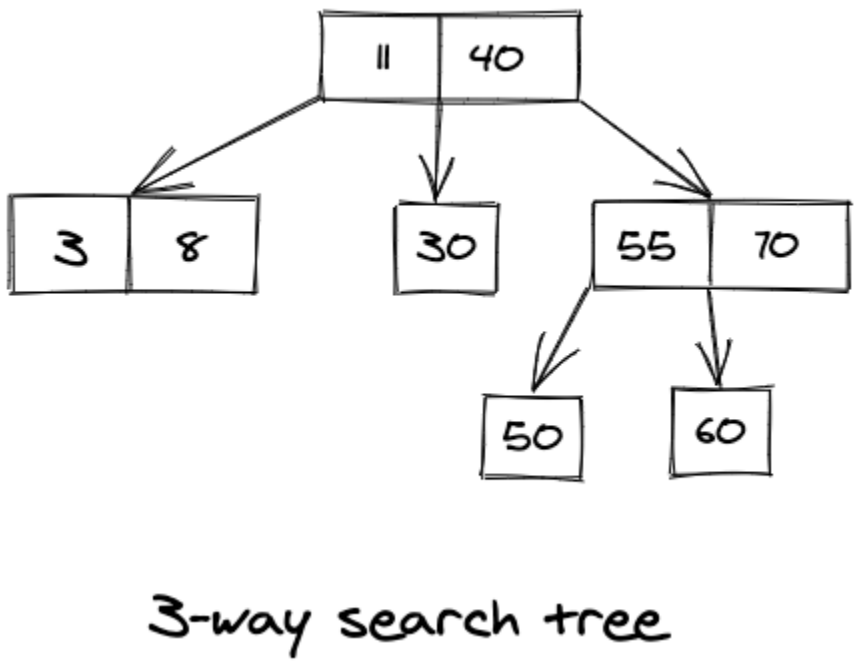
The above image shows a **4-way tree**, where each node can have at most **3(4-1) key fields** and at most **4** children. It is also a **4-way search tree**.

M-way Search Trees

An M-way search tree is a more constrained m-way tree, and these constrain mainly apply to the key fields and the values in them. The constraints on an **M-way** tree that makes it an M-way search tree are:

- Each node in the tree can associate with **m** children and **m-1** key fields.
- The keys in any node of the tree are arranged in a sorted order(**ascending**).
- The keys in the first **K** children are **less than** the **Kth** key of this node.
- The keys in the last **(m-K**) children are higher than the **Kth** key.

Consider the pictorial representation shown below of an M-way search tree:



M-way search trees have the same advantage over the M-way trees, which is making the search and update operations much more efficient. Though, they can become unbalanced which in turn leaves us to the same issue of searching for a key in a skewed tree which is not much of an advantage.

Searching in an M-way Search Tree:

If we want to search for a value say **X** in an M-way search tree and currently we are at a node that contains key values from **Y1, Y2, Y3,.....,Yk**. Then in total 4 cases are possible to deal with this scenario, these are:

- If **X < Y1**, then we need to recursively traverse the left subtree of **Y1**.
- If **X > Yk**, then we need to recursively traverse the right subtree of **Yk**.
- If **X = Yi**, for some **i**, then we are done, and can return.
- Last and only remaining case is that when for some **i** we have **Yi < X < Y(i+1)**, then in this case we need to recursively traverse the subtree that is present in between **Yi** and **Y(i+1)**.

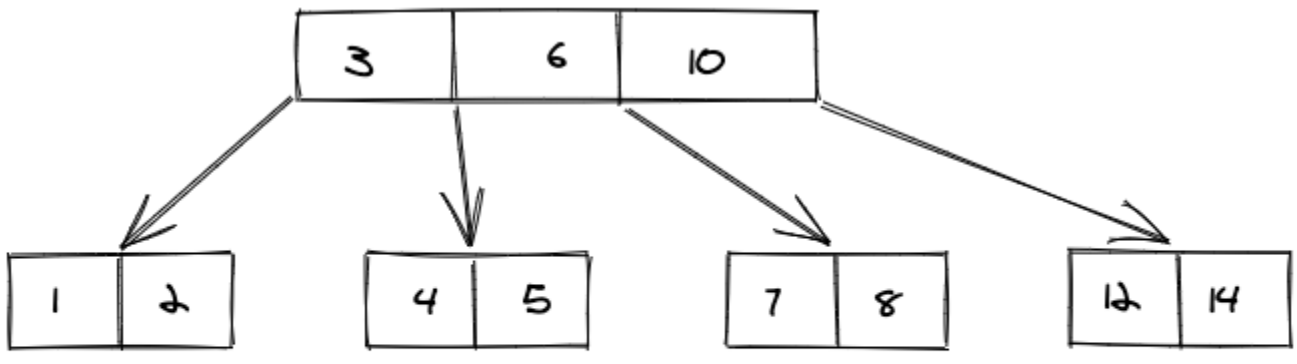
For example, consider the 3-way search tree that is shown above, say, we want to search for a node having key(X) equal to 60. Then, considering the above cases, for the root node, the second condition applies, and **(60 > 40)** and hence we move on level down to the right subtree of **40**. Now, the last condition is valid only, hence we traverse the subtree which is in between the **55** and **70**. And finally, while traversing down, we have our value that we were looking for.

B Trees Data Structure:

A B tree is an extension of an M-way search tree. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:

- All the leaf nodes in a B tree are at the same level.
- All internal nodes must have **M/2** children.
- If the root node is a non leaf node, then it must have at least two children.
- All nodes except the root node, must have at least **[M/2]-1** keys and at most **M-1** keys.

Consider the pictorial representation of a B tree shown below:



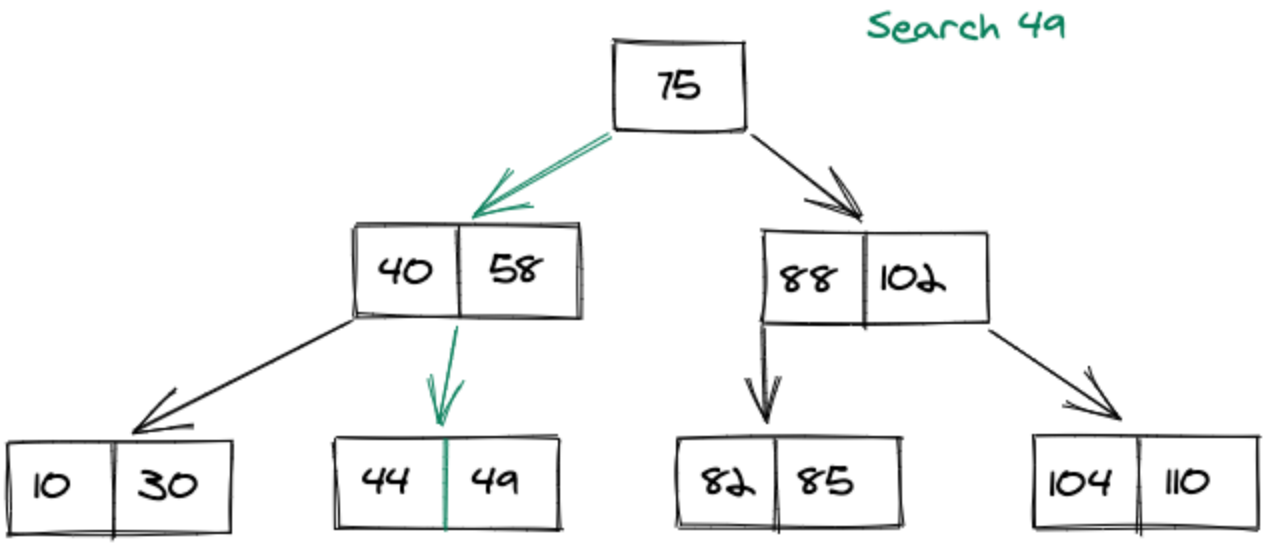
B tree

Searching in a B Tree:

Searching for a key in a B Tree is exactly like searching in an M-way search tree, which we have seen just above. Consider the pictorial representation shown below of a B tree, say we want to search for a key 49 in the below shown B tree. We do it as following:

- Compare item **49 with root node 75**. Since **49 < 75** hence, move to its left sub-tree.
- Since, **40<49<58**, traverse right sub-tree of 40.
- **49>44**, move to right. Compare **49**.
- We have found 49, hence returning.

Consider the pictorial representation shown below:

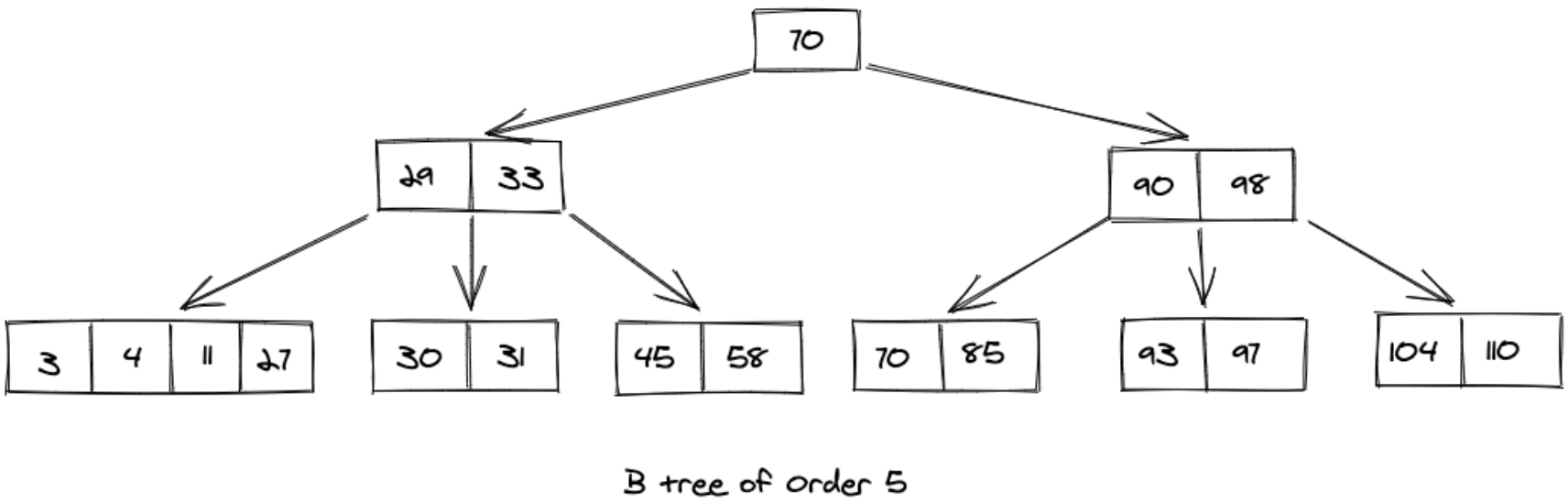


Inserting in a B Tree:

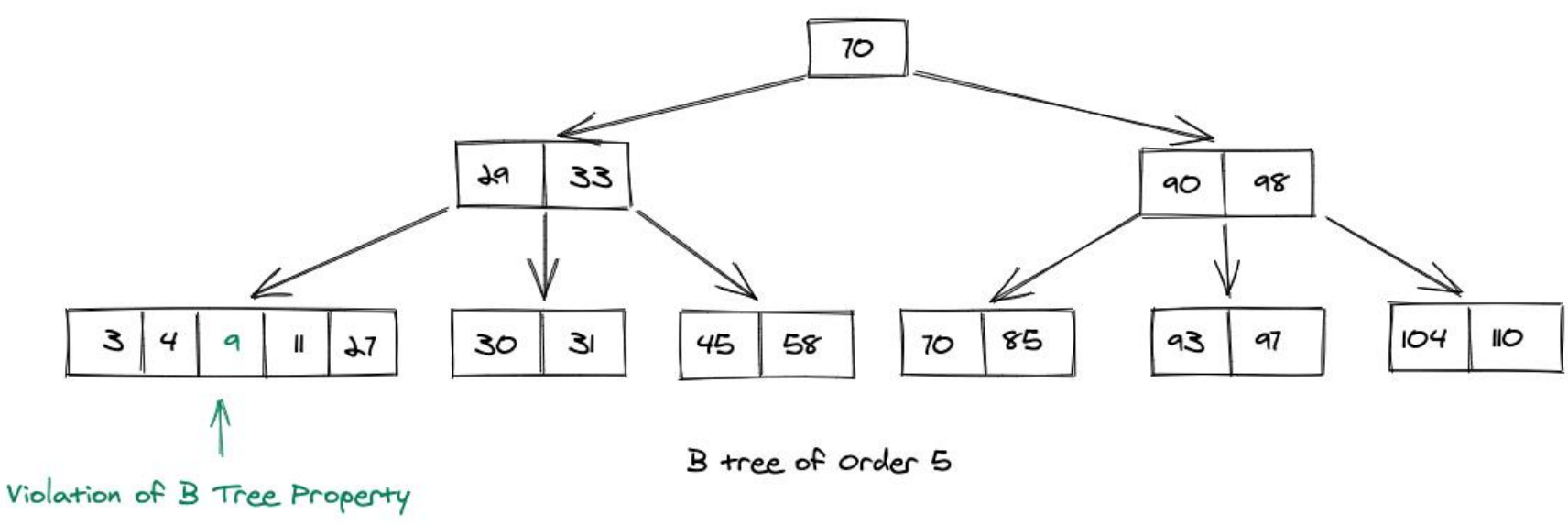
Inserting in a B tree is done at the leaf node level. We follow the given steps to make sure that after the insertion the B tree is valid, these are:

- First, we traverse the B tree to find the appropriate node where the to be inserted key will fit.
- If that node contains less than **M-1** keys, then we insert the key in an increasing order.
- If that node contains exactly **M-1** keys, then we have two cases ? Insert the new element in increasing order, split the nodes into two nodes through the median, push the median element up to its parent node, and finally if the parent node also contains **M-1 keys**, then we need to repeat these steps.

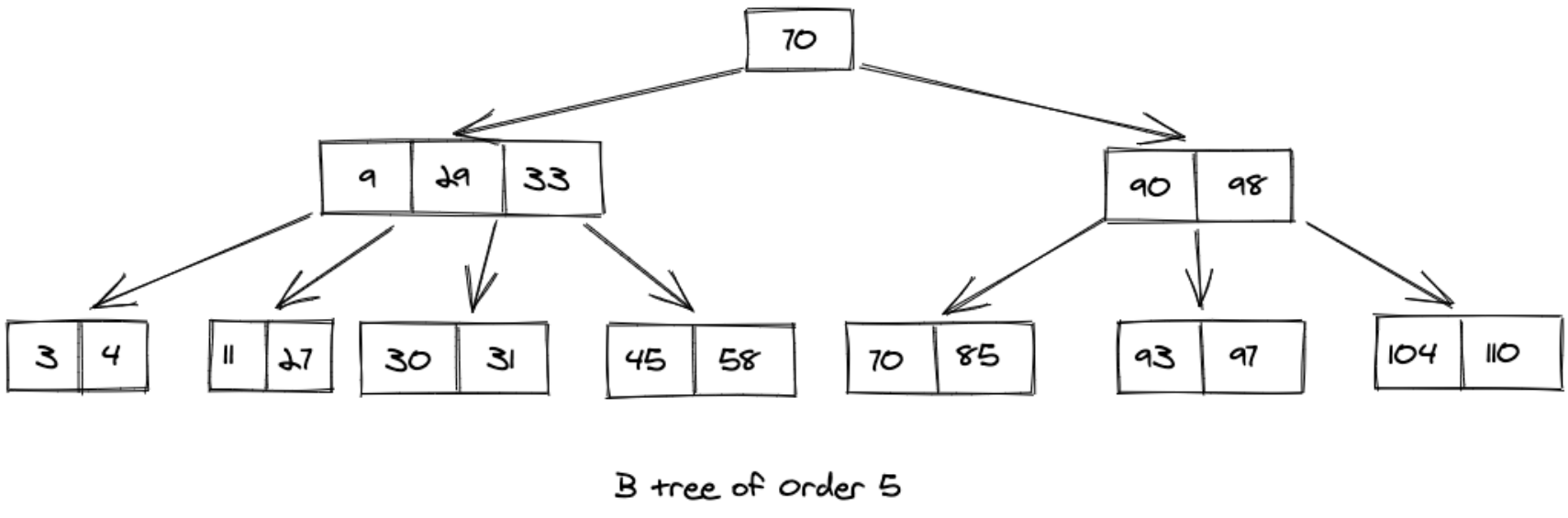
Consider the pictorial representation shown below:



Now, consider that we want to insert a key 9 into the above shown B tree, the tree after inserting the key 9 will look something like this:



Since, a violation occurred, we need to push the median node to the parent node, and then split the node in two parts, hence the final look of B tree is:



B tree of order 5

Deletion in a B Tree:

Deletion of a key in a B tree includes two cases, these are:

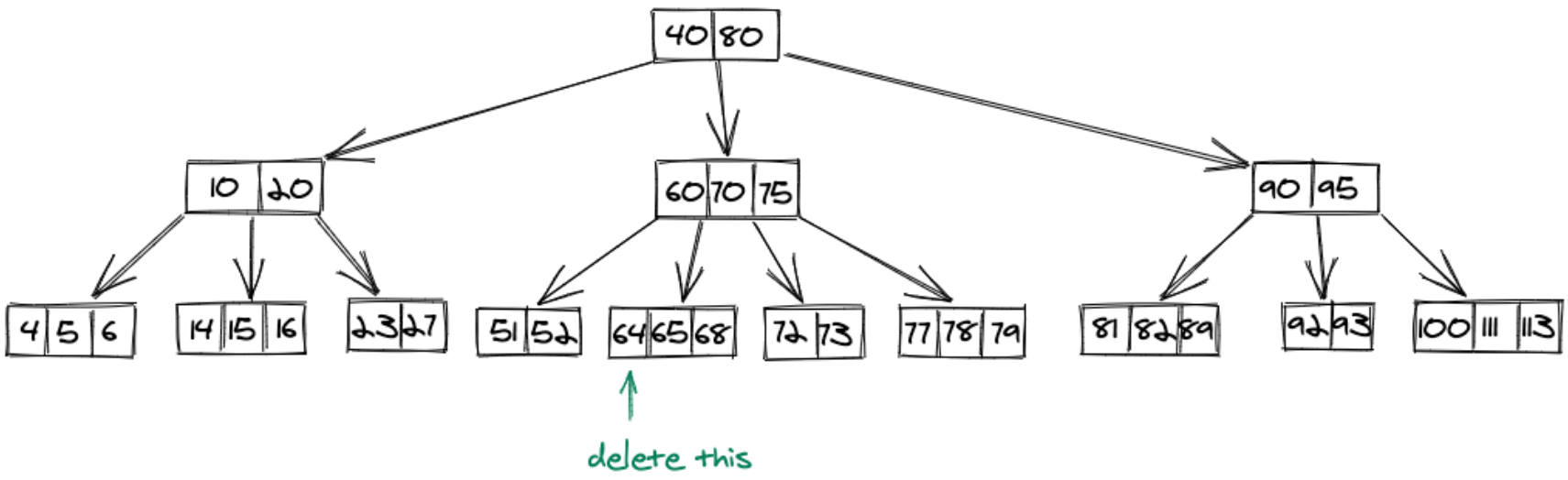
- **Deletion of key from a leaf node**
- **Deletion of a key from an Internal node**

Deletion of Key from a leaf node:

If we want to delete a key that is present in a leaf node of a B tree, then we have two cases possible, these are:

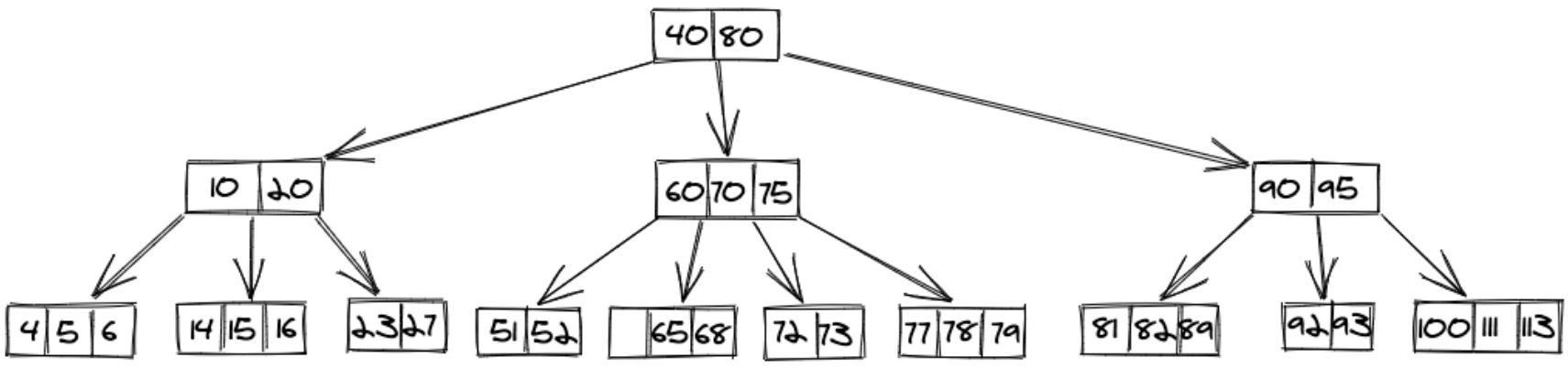
- If the node that contains the key that we want to delete, in turn **contains more than the minimum number of keys required** for the valid B tree, then we can simply delete that key.

Consider the pictorial representation shown below:



Say, we want to delete the key 64 and the node in which 64 is present, has more than minimum number of nodes required by the B tree, which is 2. So, we can simply delete this node.

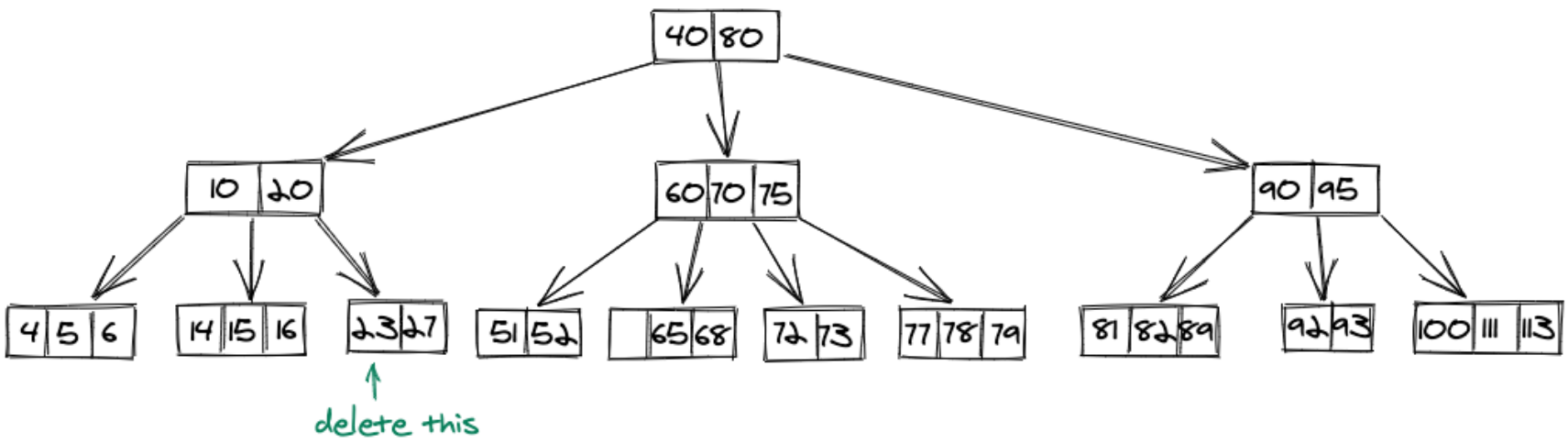
The final tree after deletion of 64 will look like this:



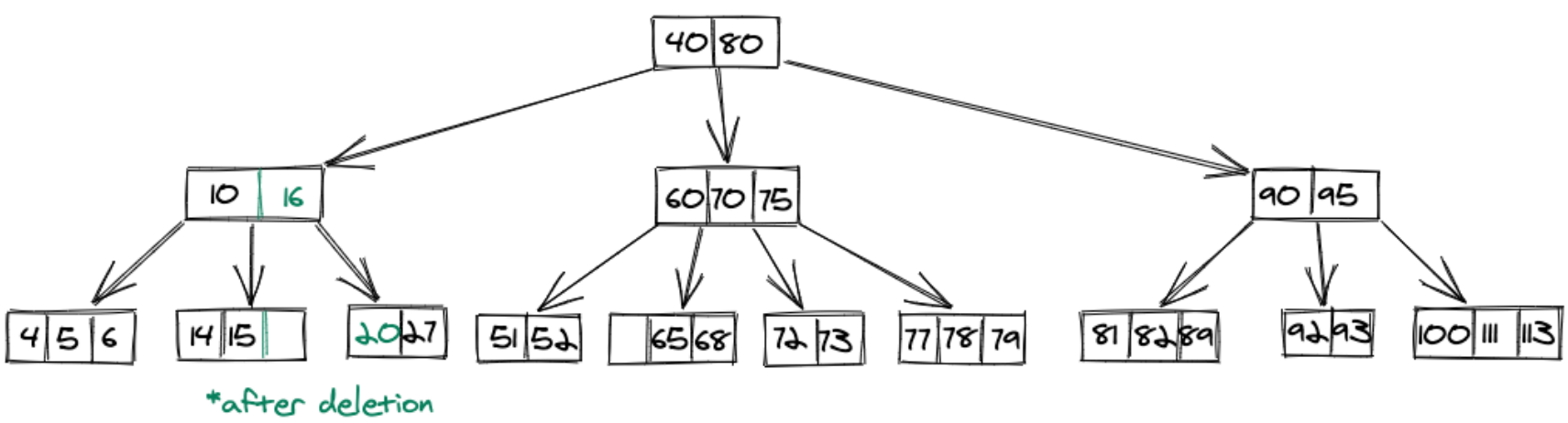
- If the node that contains the key that we want to delete, in turn **contains the minimum number of keys required** for the valid B tree, then three cases are possible:
  - In order to delete this key from the B Tree, we can borrow a key from the immediate left node(left sibling). The process is that we move the highest value key from the left sibling to the parent, and then the highest value parent key to the node from which we just deleted our key.
  - In another case, we might have to borrow a key from the immediate right node(right sibling). The process is that we move the lowest value key from the right sibling to the parent node, and then the highest value parent key to the node from which we just deleted our key.

- Last case would be that neither the left sibling or the right sibling are in a state to give the current node any value, so in this step we will do a merge with either one of them, and the merge will also include a key from the parent, and then we can delete that key from the node.

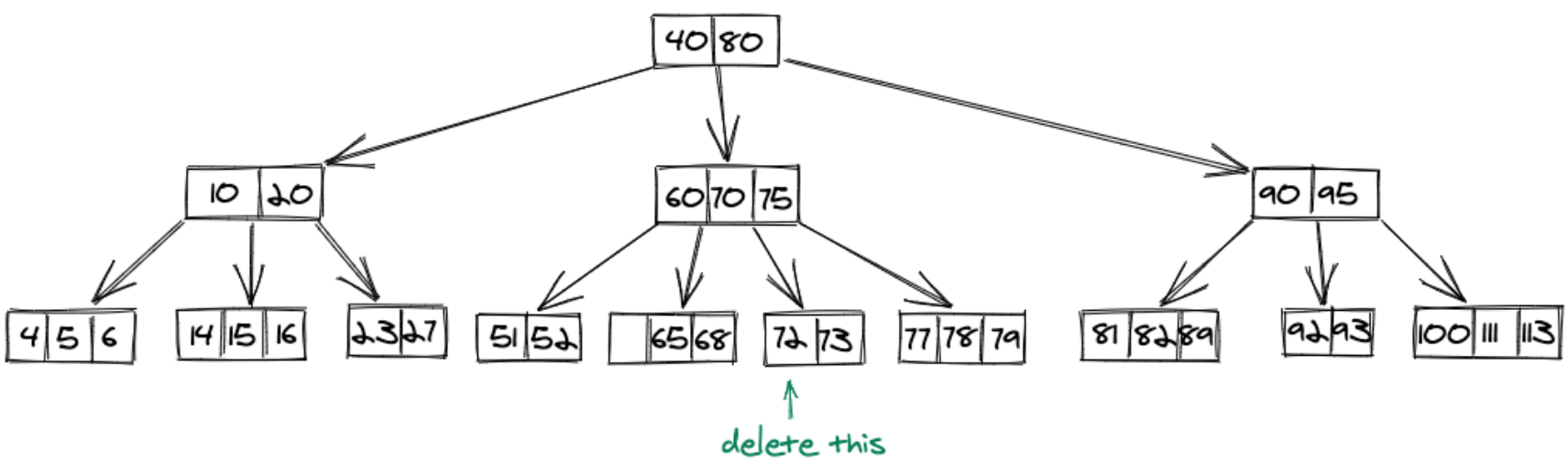
Case 1 pictorial representation:



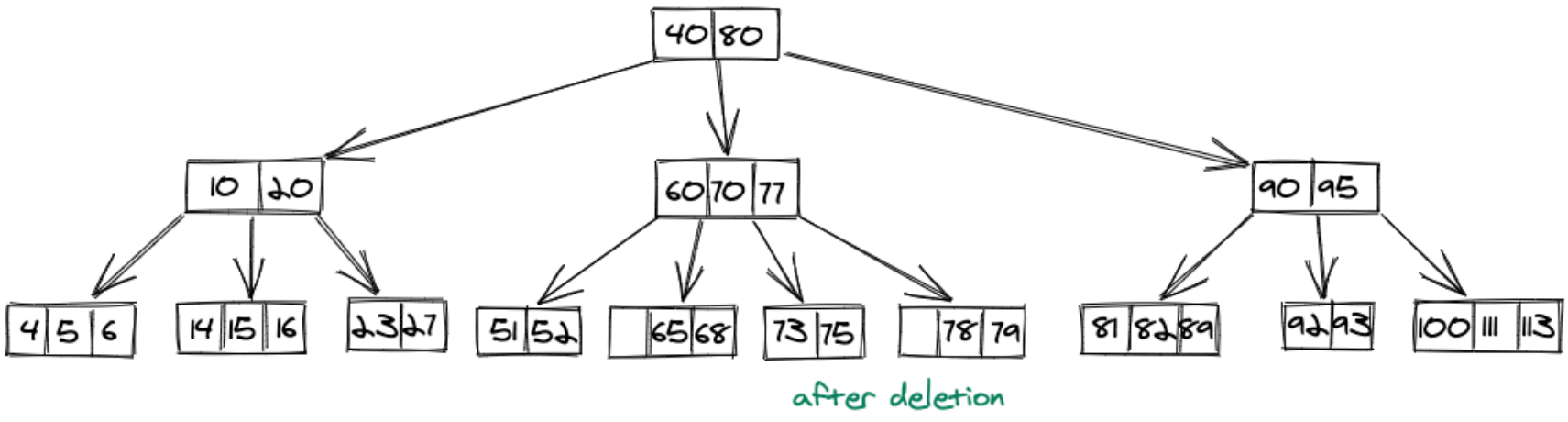
After we delete 23, we ask the left sibling, and then move 16 to the parent node and then push 20 downwards, and the resultant B tree is:



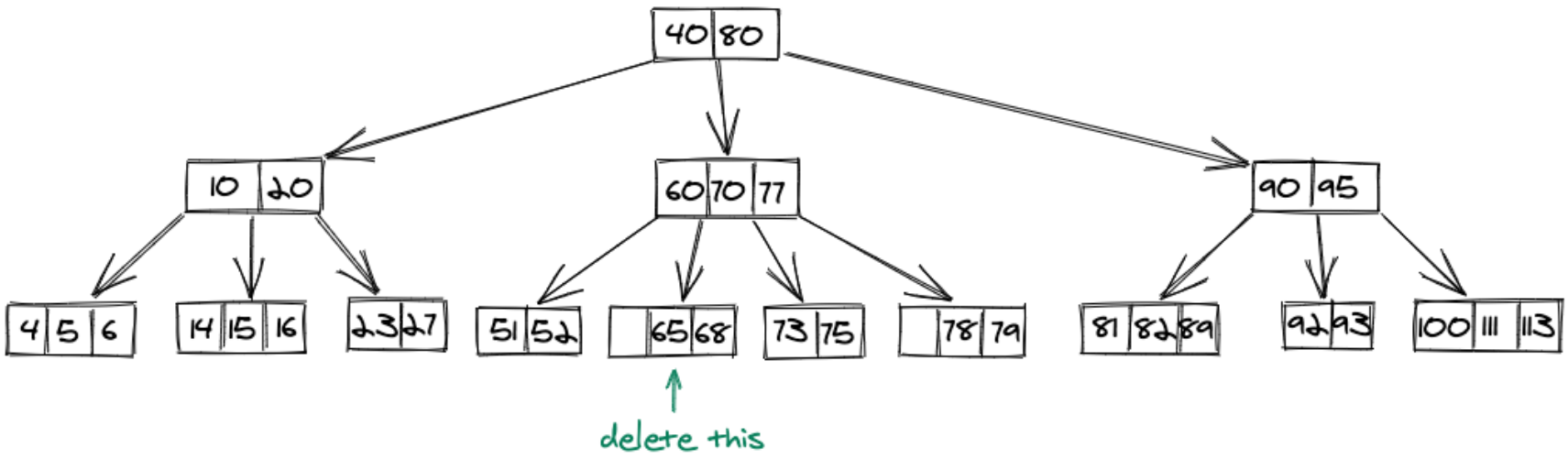
Case 2 pictorial representation:



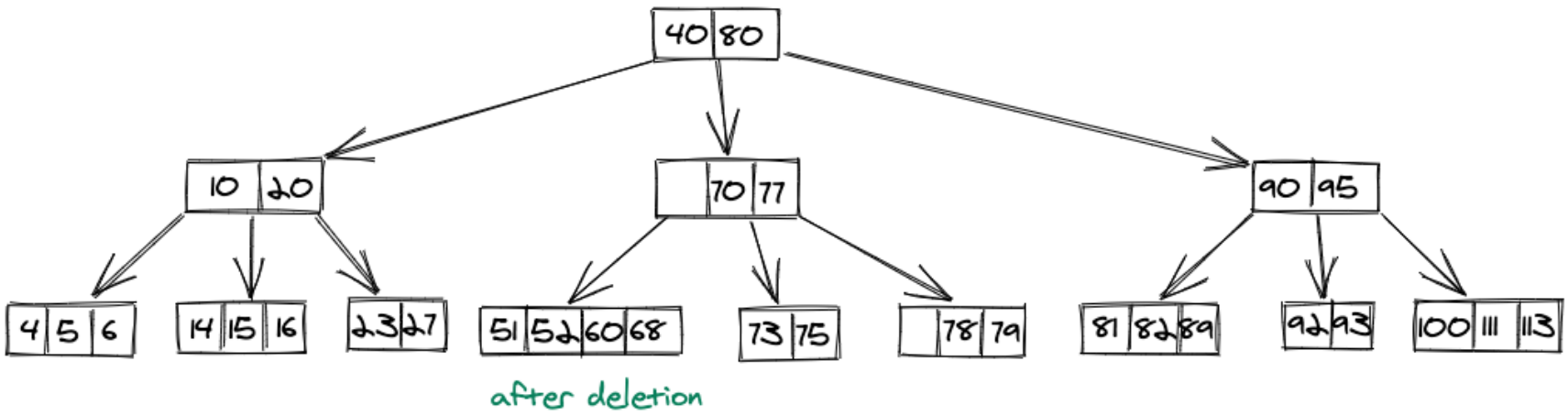
After we delete 72, we ask the right sibling, and then move the 77 to the parent node and then push the 75 downwards, and the resultant B tree is:



Case 3 pictorial representation:



After deleting 65 from the leaf node, we will have the final B tree as:

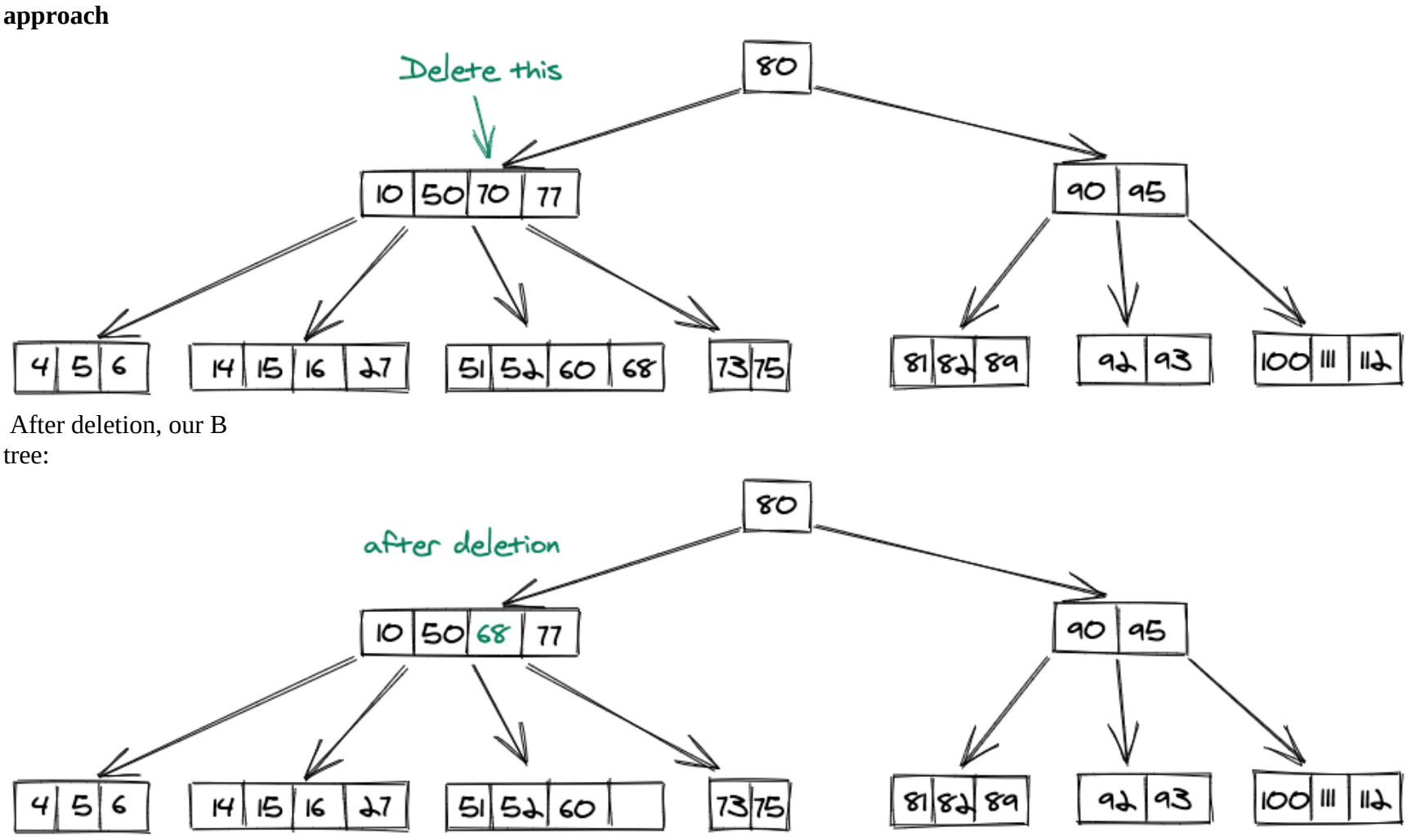


Deletion of Key from an Internal node:

- If we want to delete a key that is present in an internal node, then we can either take the value which is in order **predecessor** of this key or if taking that inorder predecessor violates the B tree property we can take the **inorder successor** of the key.
- In the inorder predecessor approach, we extract the highest value in the left children node of the node where our key is present.
- In the inorder successor approach, we extract the lowest value in the right children node of the node where our key is present.

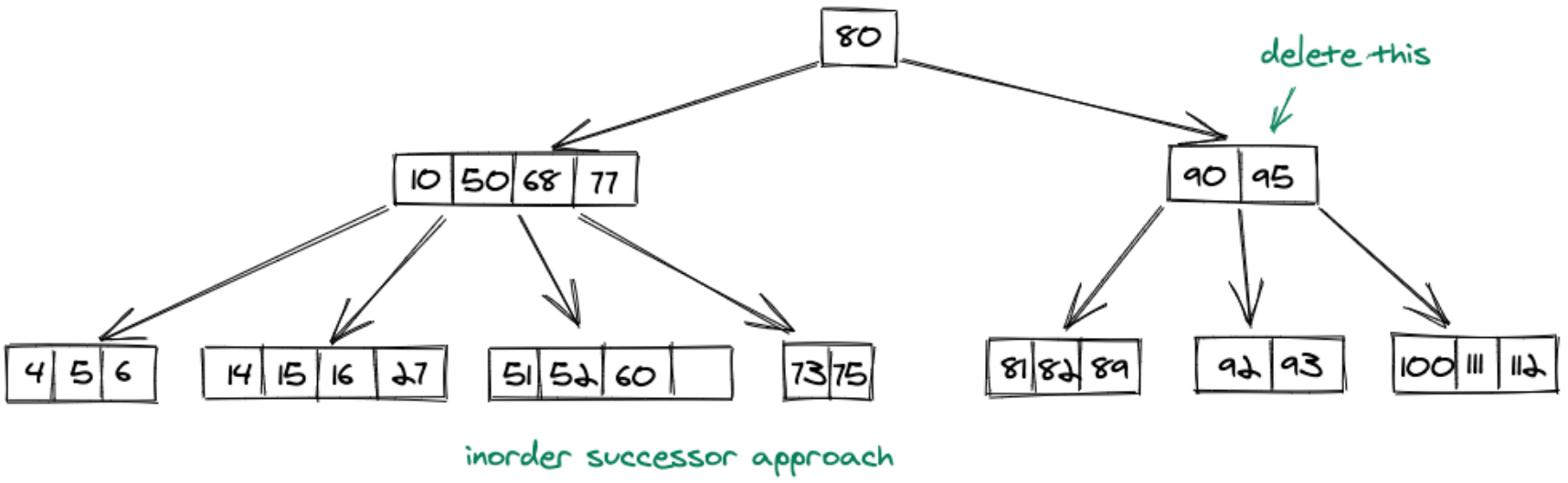
Pictorial Representation of the above cases:

- **Internal predecessor approach**

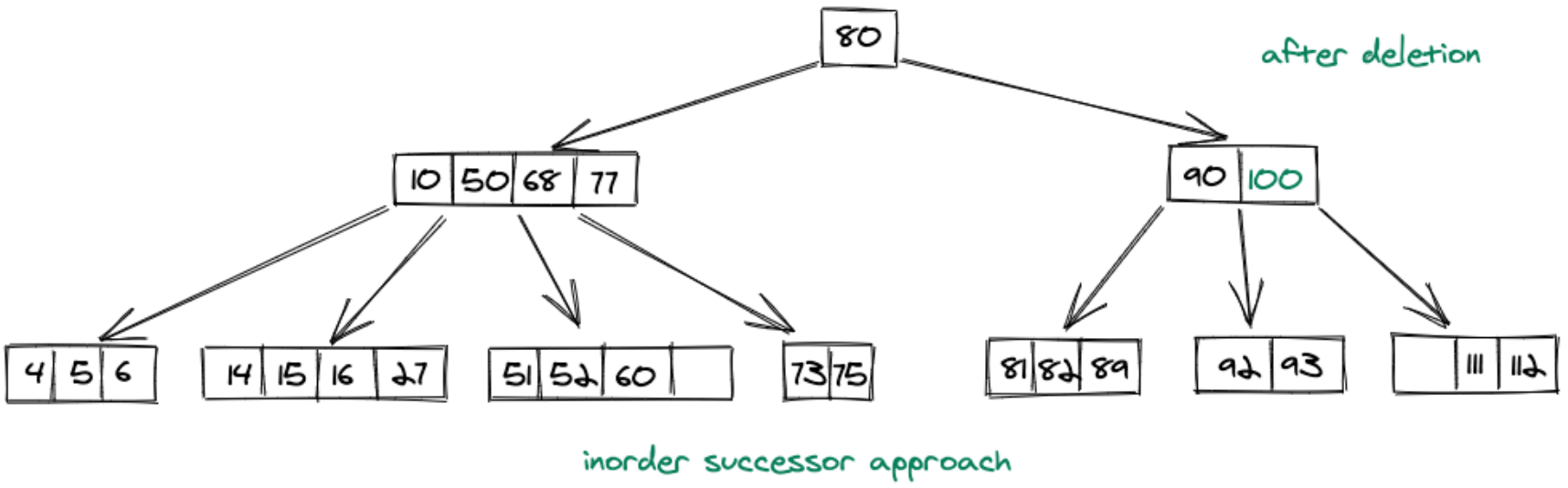


After deletion, our B tree:

- **Internal successor approach**



After deletion of 95, our tree will look like this:



Key Points:

- The time complexity for search, insert and delete operations in a B tree is **O(log n).**
- The minimum number of keys in a B tree should be **[M/2] - 1.**
- The maximum number of keys in a B tree should be **M-1.**
- All the leaf nodes in a B tree should be at the same level.
- All the keys in a node in a binary tree are in increasing order.
- B Trees are used in SQL to improve the efficiency of queries.
- Each node in a B Tree can have at most **M** children.

Conclusion:

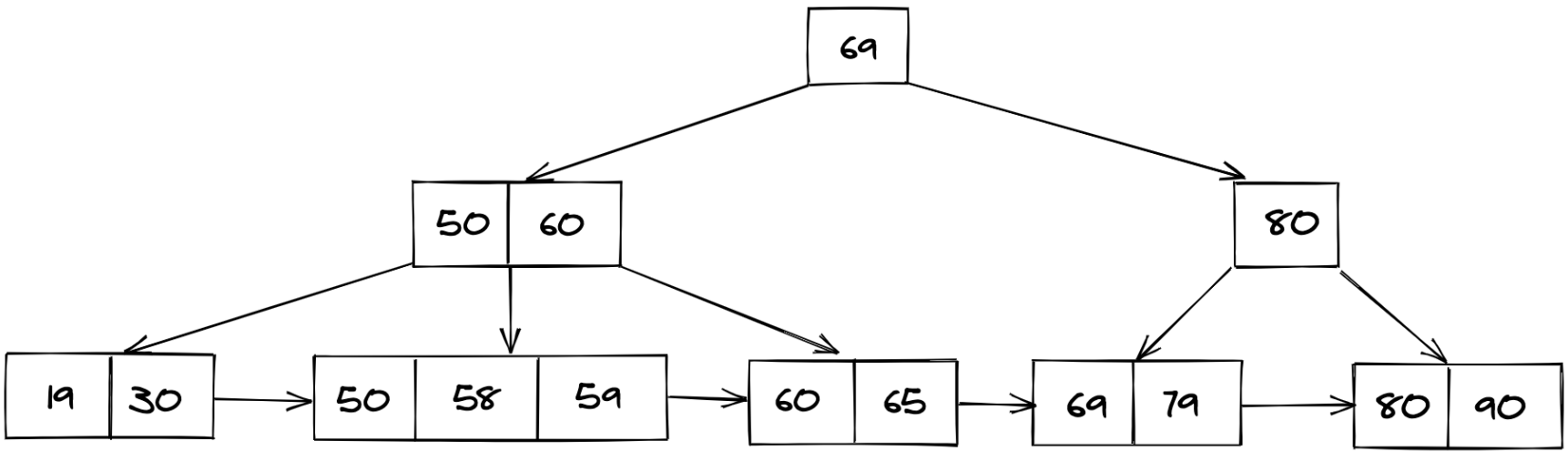
In this article, we learned what an M-way tree is, what are the differences between the M-way tree and M-way search tree. Also, we learned the constraints that are applied to an M-way tree to make it a B tree. Then we learned about the search, insert and delete operations.

B+ Trees Data Structure

A **B+ tree** is an extension of a B tree which makes the search, insert and delete operations more efficient. We know that B trees allow both the data pointers and the key values in internal nodes as well as leaf nodes, this certainly becomes a drawback for B trees as the ability to insert the nodes at a particular level is decreased thus increase the node levels in it, which is certainly of no good. B+ trees reduce this drawback by simply **storing the data pointers at the leaf node level** and only storing the key values in the internal nodes. It should also be noted that the nodes at the leaf level are linked with each other, hence making the traversal of the data pointers easy and more efficient.

B+ trees come in handy when we want to store **a large amount of data** in the main memory. Since we know that the size of the main memory is not that large, so make use of the B+ trees, whose internal nodes that store the key(to access the records) are stored in the main memory whereas, the leaf nodes that contain the data pointers are actually stored in the secondary memory.

A pictorial representation of a B+ tree is given below:



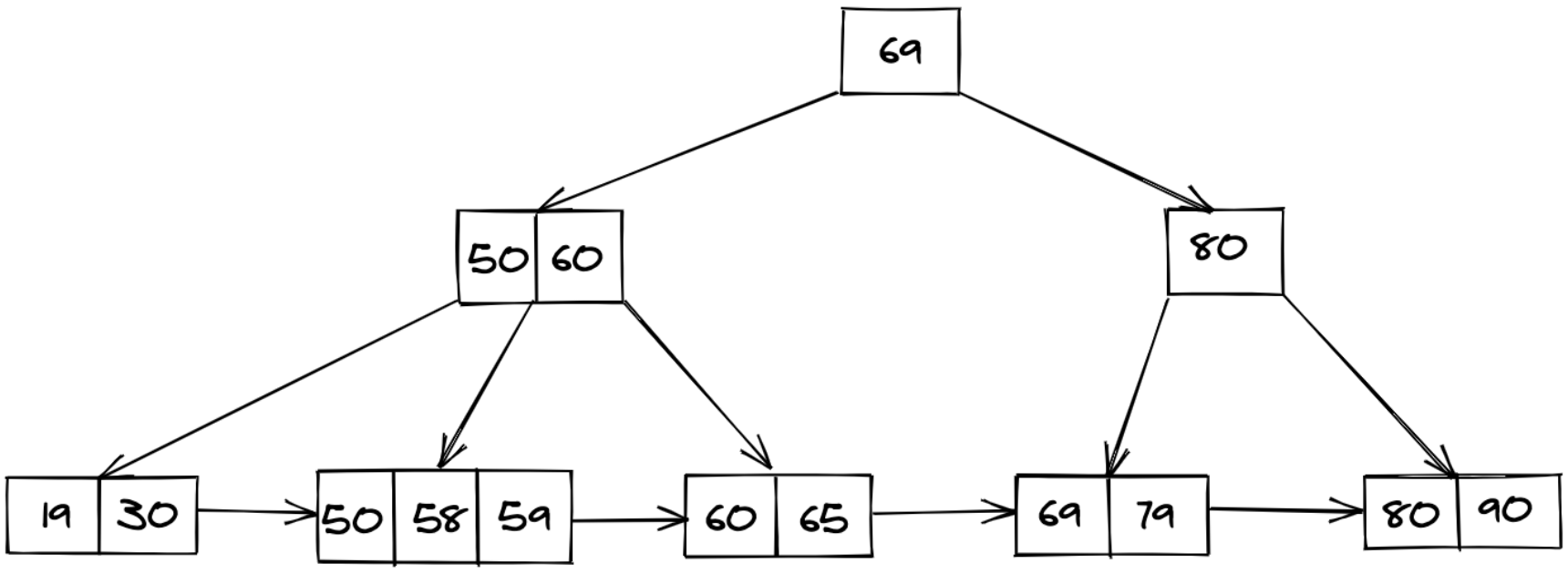
Why B+ trees?

- B+ trees store the records which later can be fetched in an equal number of disk accesses.
- The height of the B+ tree **remains balanced** and very less as when compared to B trees even if the number of records store in them is the same.
- Having less number of levels makes the accessing of records very easy.
- As the leaf nodes are connected with each other **like a linked list**, we can easily search elements in sequential manners.

Inserting in B+ tree

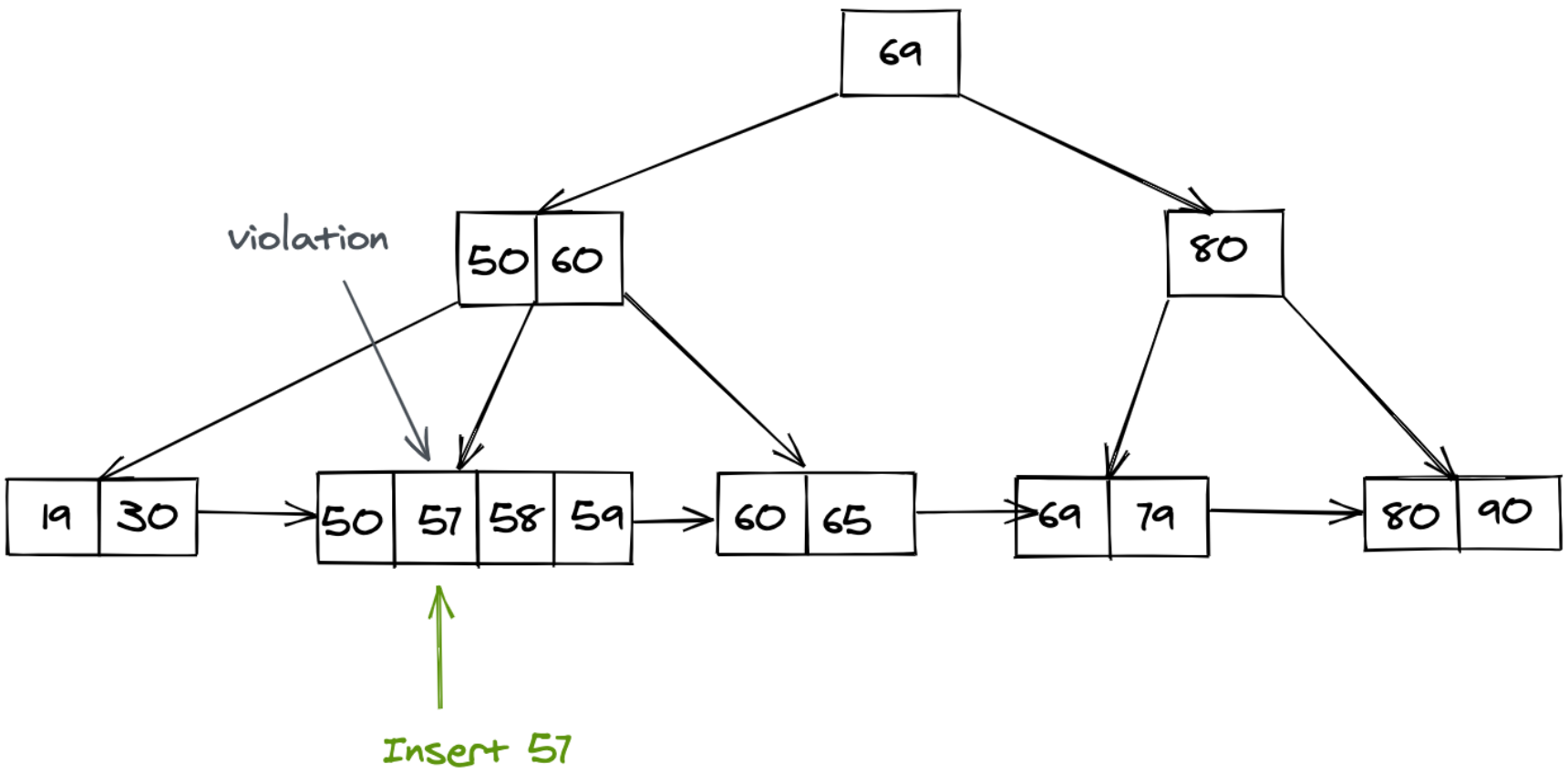
- Perform a search operation in the B+ tree to check the **ideal bucket location** where this new node should go to.
- If the bucket is not full(does not violate the B+ tree property), then add that node into this bucket.
- Otherwise split the nodes into two nodes and push the middle node(median node to be precise) to the parent node and then insert the new node.
- Repeat the above steps if the parent node is there and the current node keeps getting full.

Consider the pictorial representations shown below to understand the Insertion operation in the B+ tree:

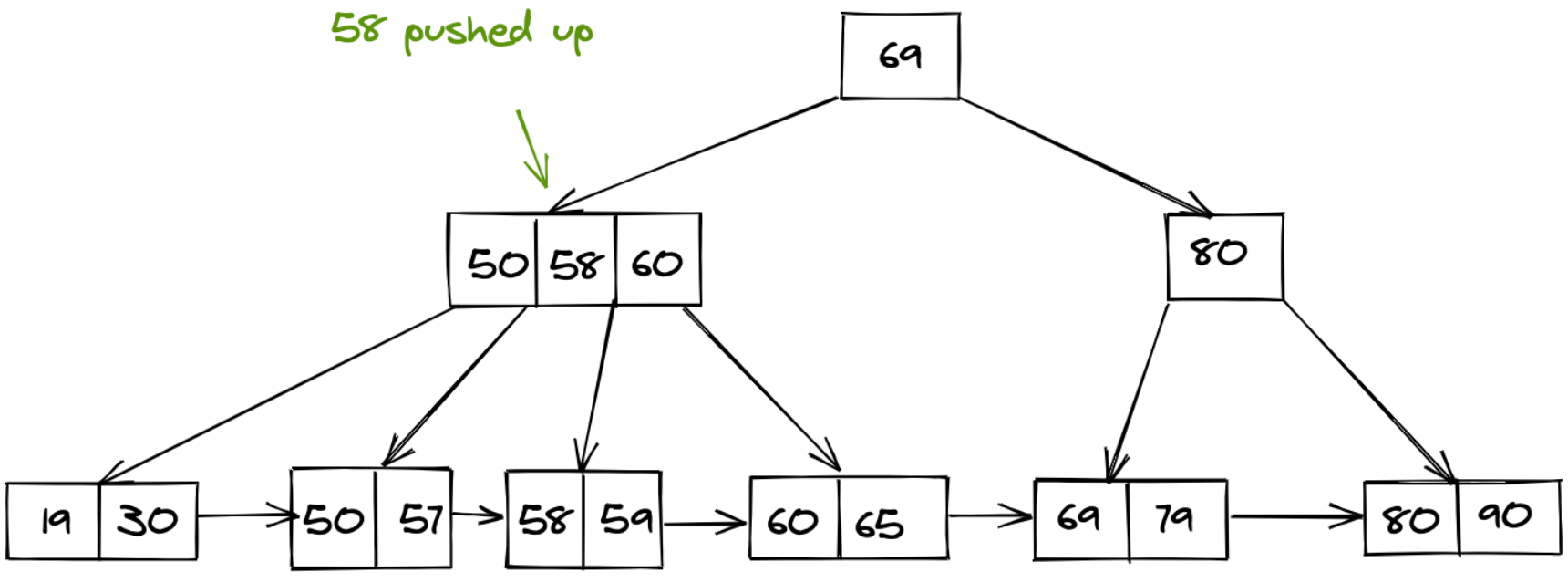


B+ tree of order 4

Let us try to insert 57 inside the above-shown B+ tree, the resultant B+ tree will look like this:



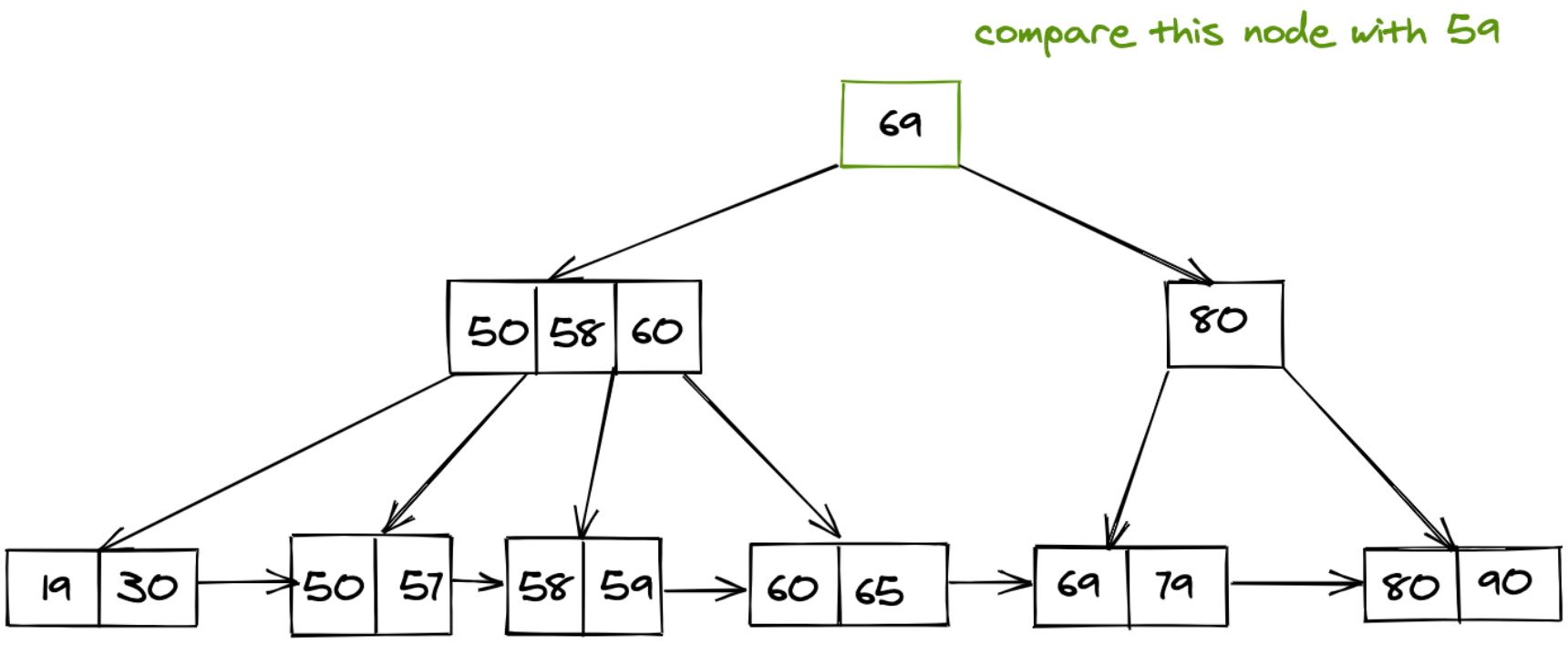
We know that the bucket(node) where we inserted the key with value 57 is now violating the property of the B+ tree, hence we need to split this node as mentioned in the steps above. After splitting we will push the median node to the parent node, and the resulting B+ tree will look like this:



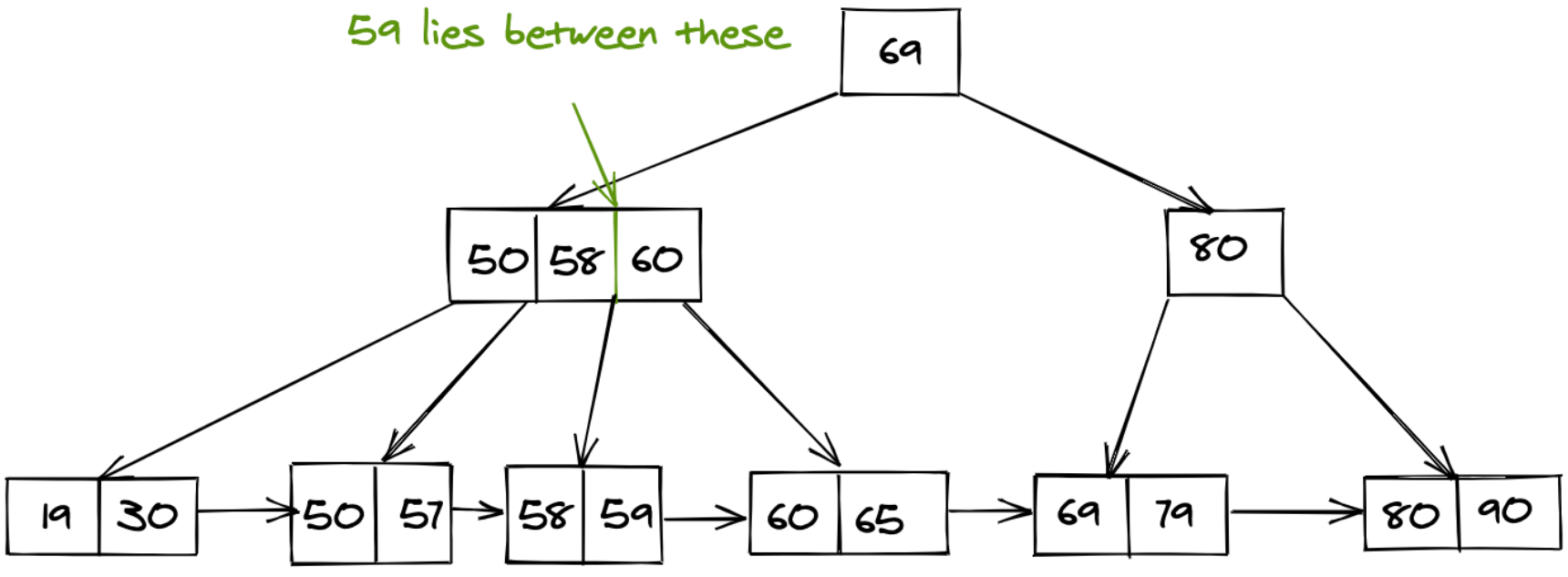
Searching in B+ tree:

Searching in a B+ tree is similar to searching in a BST. If the current value is less than the searching key, then traverse the left subtree, and if greater than first traverse this current bucket(node) and then check where the ideal location is.

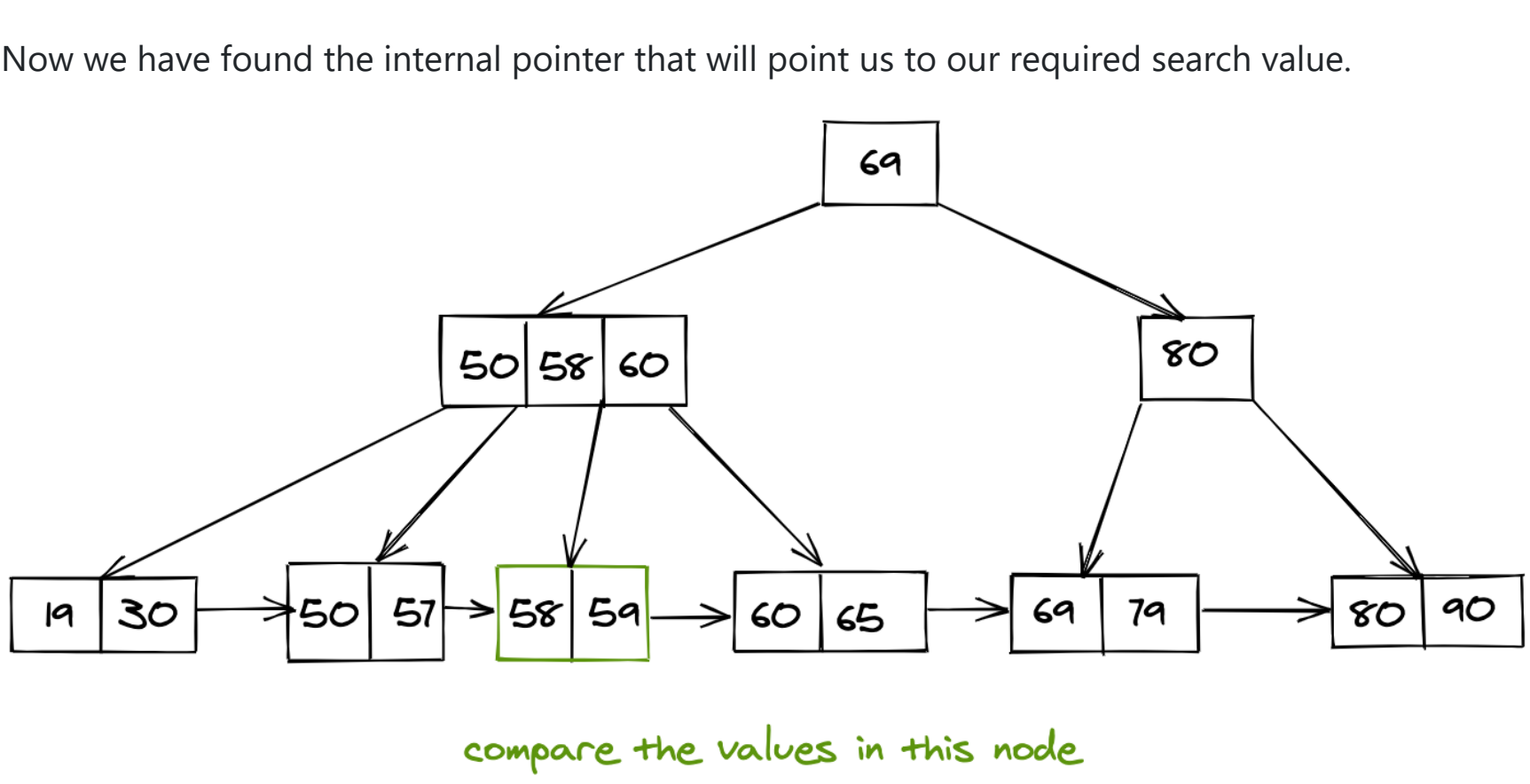
Consider the below representation of a B+ tree to understand the searching procedure. Suppose we want to search for a key with a value equal to 59 in the below given B+ tree.



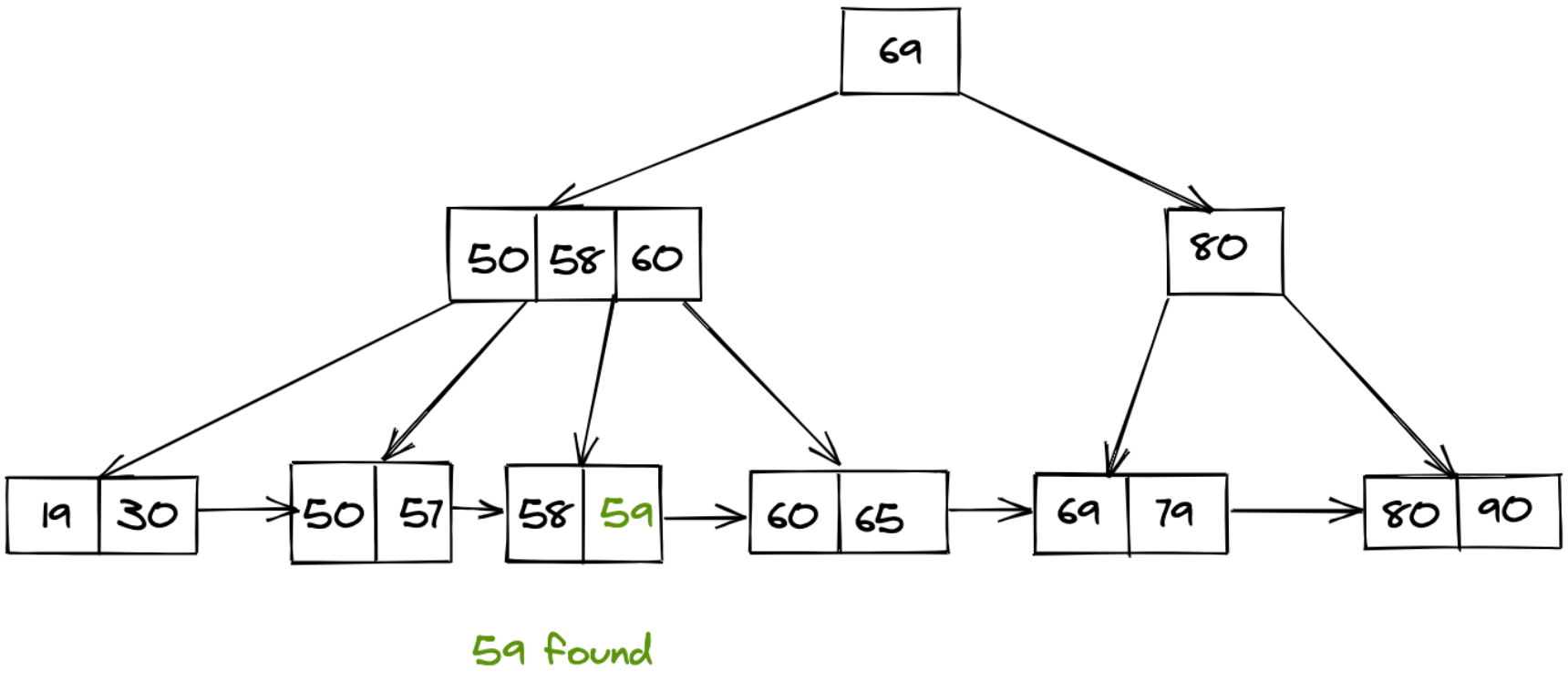
Now we know that 59 < 69, hence we traverse the left subtree.



Now we have found the internal pointer that will point us to our required search value.



Finally, we traverse this bucket in a linear fashion to get to our required search value.



Deletion in B+ tree:

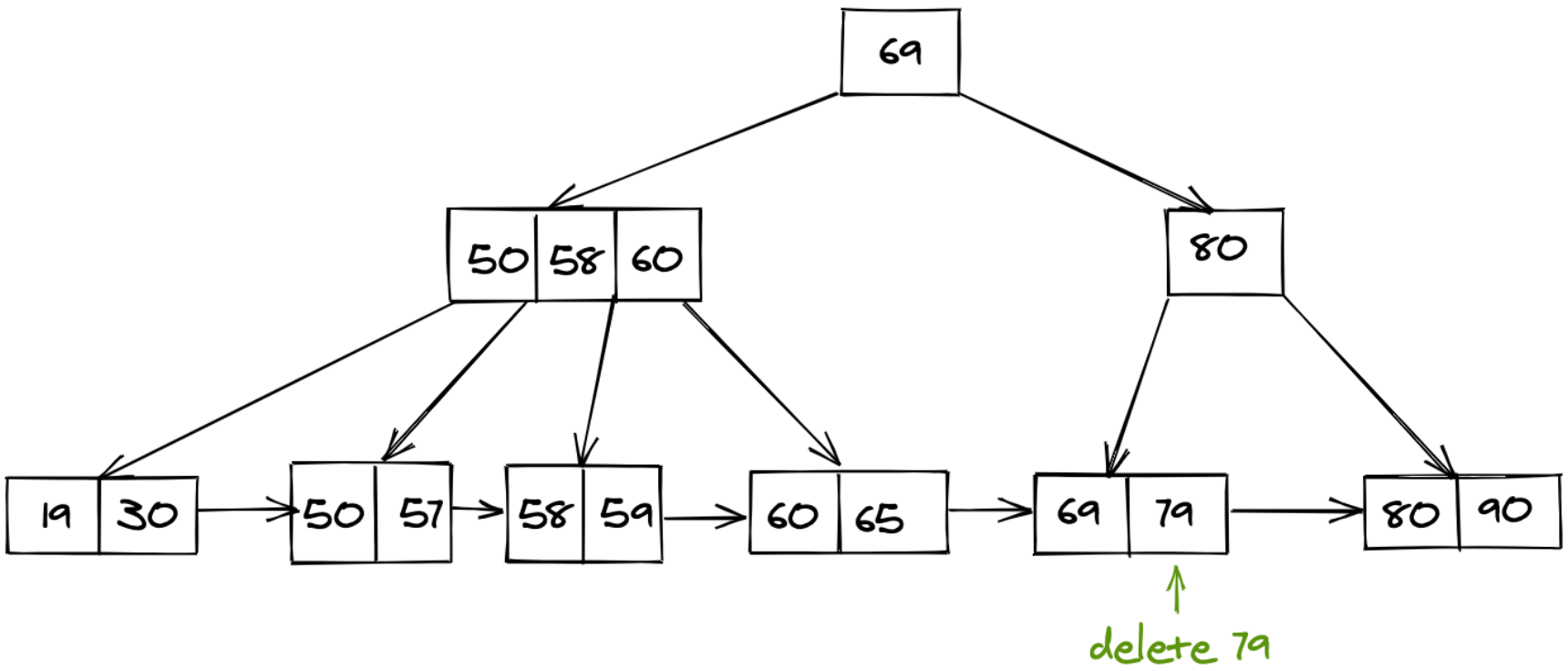
Deletion is a bit complicated process, two case arises:

- It is only present at the **leaf level**
- or, it contains a pointer from an internal node also.

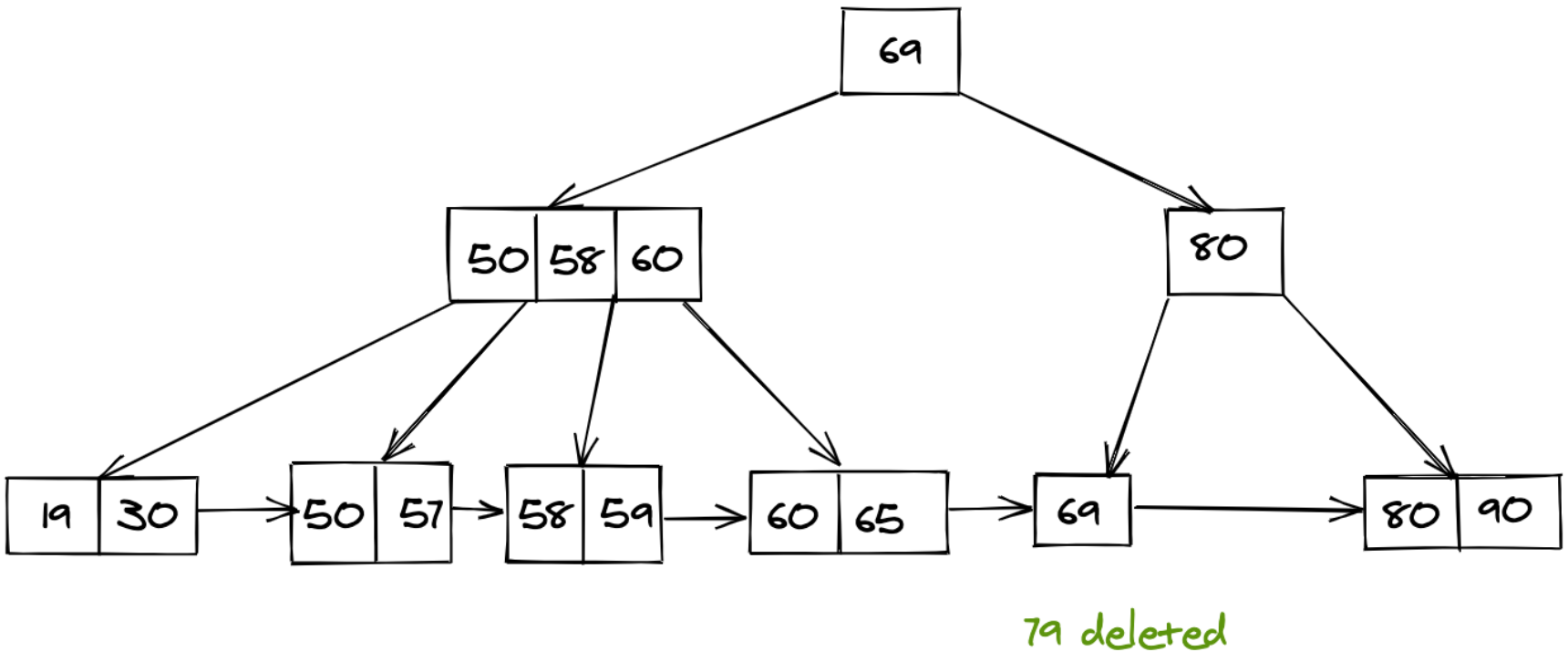
Deletion of only the leaf-node:

If it is present only as a **leaf node position**, then we can simply delete it, for which we first do the search operation and then delete it.

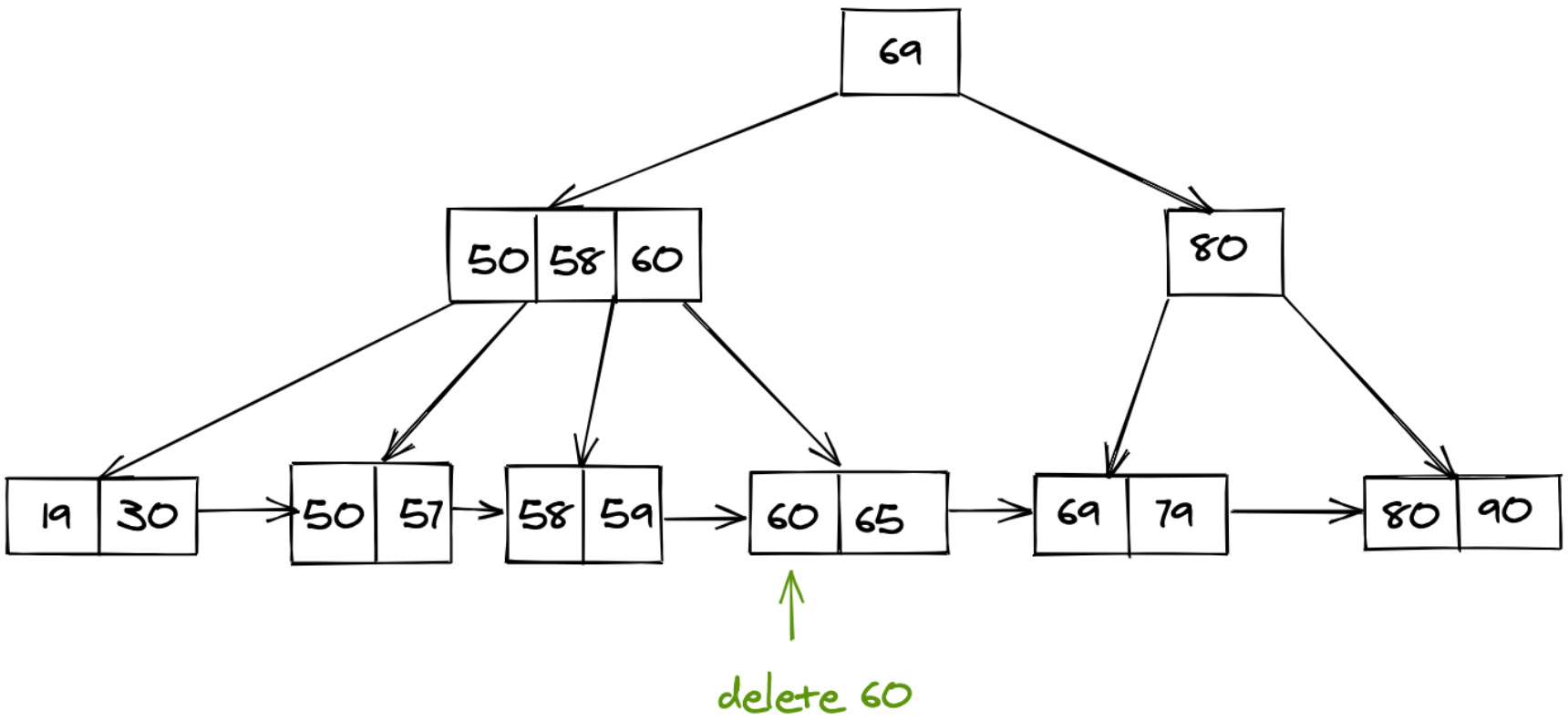
Consider the pictorial representation shown below:



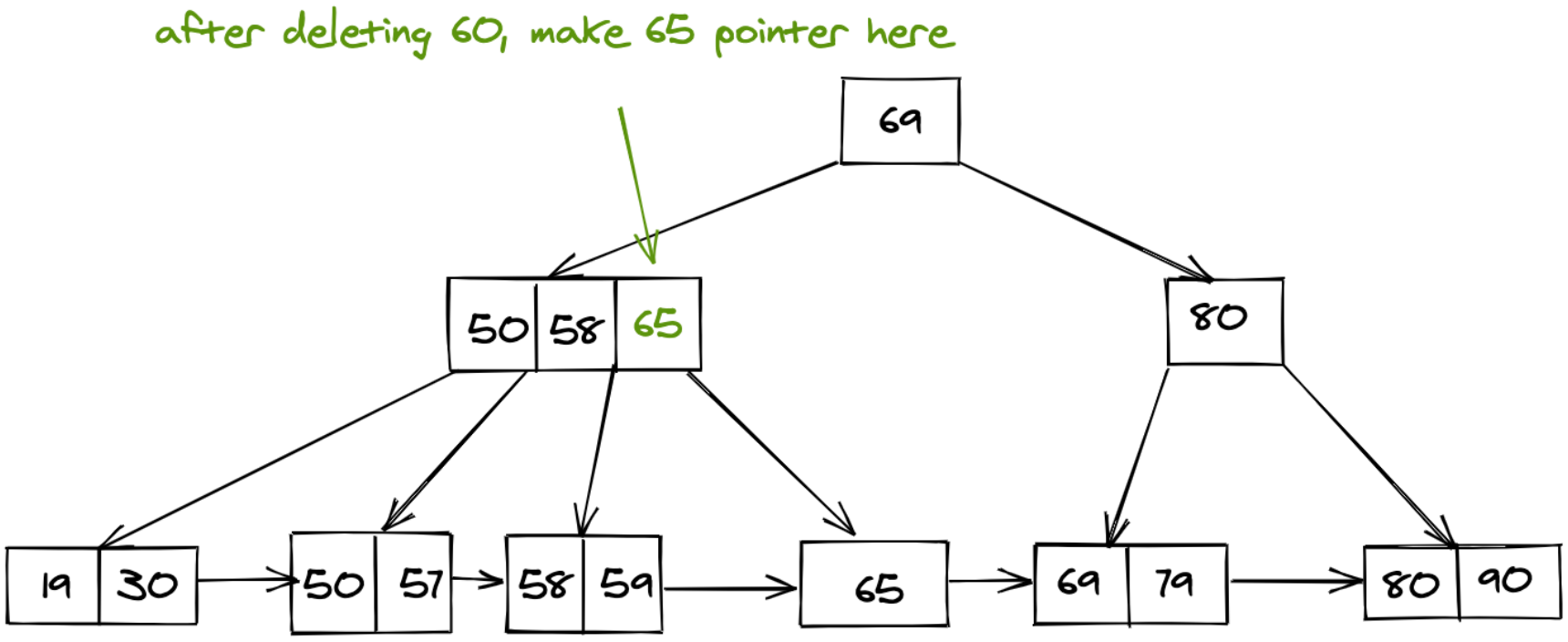
After the deletion of 79, we are left with the following B+ tree.



Deletion if a pointer to a leaf node is there:



After we locate the node we want to delete we must also delete the internal pointer that points to this node, and then we finally need to move the next node pointer to move to the parent node.



Conclusion:

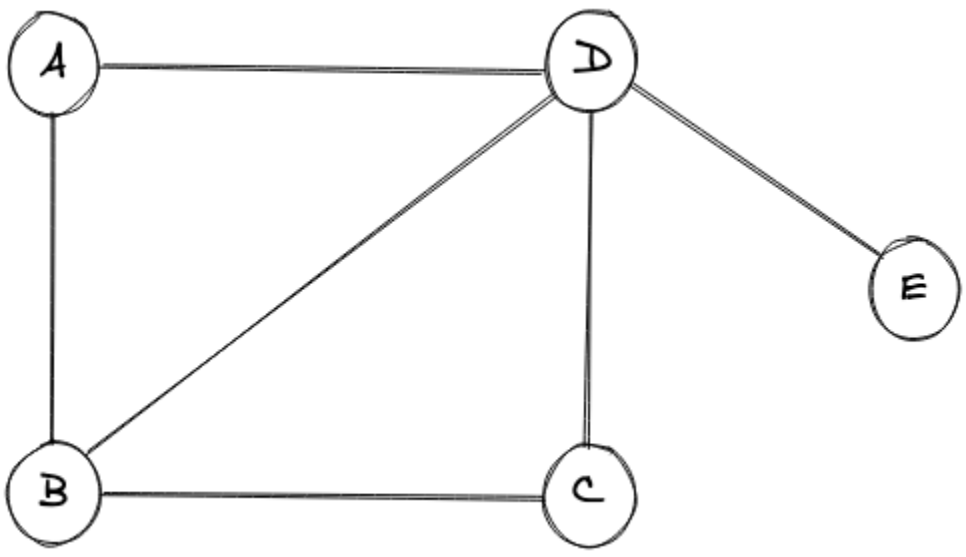
- We learned what a B+ tree is, and how it is different from a B tree.

- Then we learned why we need a B+ tree.
- Lastly, we learned different operations that we do on a B+ tree-like searching, Insertion, and deletion of a record(key).

Introduction to Graphs

A **graph is an advanced data structure** that is used to organize items in an **interconnected network**. Each item in a graph is known as a **node**(or **vertex**) and these nodes are connected by **edges**.

In the figure below, we have a simple graph where there are five nodes in total and six edges.

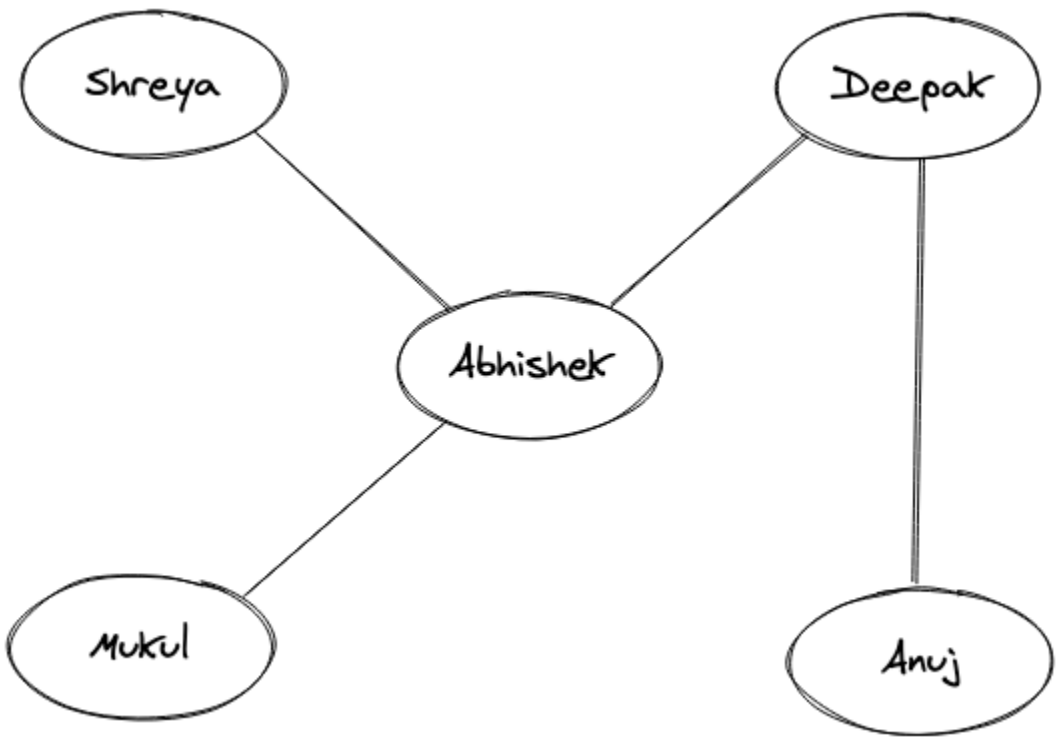


The nodes in any graph can be referred to as **entities** and the edges that connect different nodes define the **relationships between these entities**. In the above graph we have a set of nodes **{V} = {A, B, C, D, E}** and a set of edges, **{E} = {A-B, A-D, B-C, B-D, C-D, D-E}** respectively.

Real-World Example

A very good example of graphs is a **network of socially connected people**, connected by a simple connection which is whether they know each other or not.

Consider the figure below, where a pictorial representation of a social network is shown, in which there are five people in total.



A line in the above representation between two people mean that they know each other. If there's no line in between the names, then they simply don't know each other. The names here are equivalent to the nodes of a graph and the lines that define the relationship of "knowing each other" is simply the equivalent of an edge of a graph. It should also be noted that the relationship of knowing each other goes both ways like "Abhishek" knows "Mukul" and "Mukul" knows "Abhishek".

The social network depicted above is nothing but a graph.

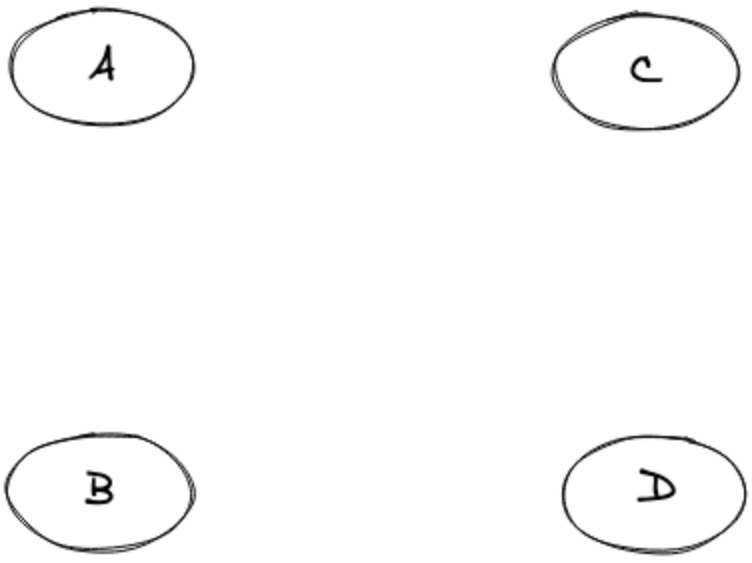
Types of Graphs

Let's cover various different types of graphs.

1. Null Graphs

A graph is said to be null if there are no edges in that graph.

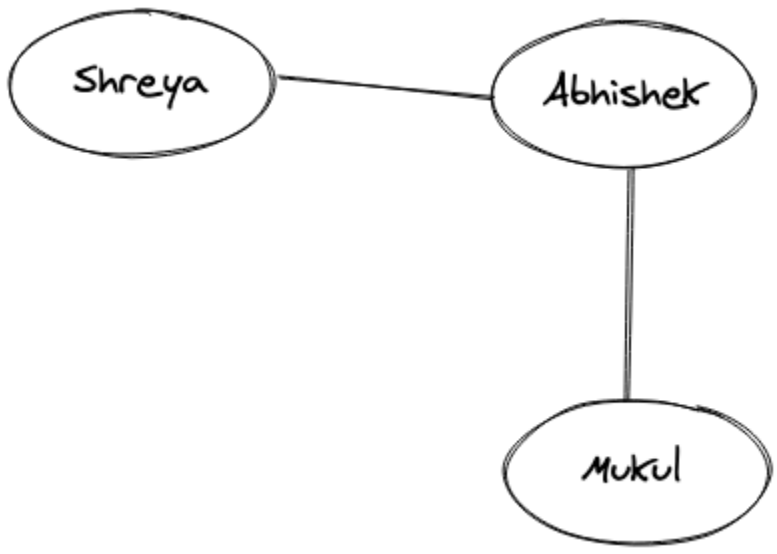
A pictorial representation of the null graph is given below:



2. Undirected Graphs

If we take a look at the pictorial representation that we had in the Real-world example above, we can clearly see that different nodes are connected by a link (i.e. edge) and that edge doesn't have any kind of direction associated with it. For example, the edge between "Anuj" and "Deepak" is bi-directional and hence the relationship between them is two ways, which turns out to be that "Anuj" knows "Deepak" and "Deepak" also knows about "Anuj". These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs.

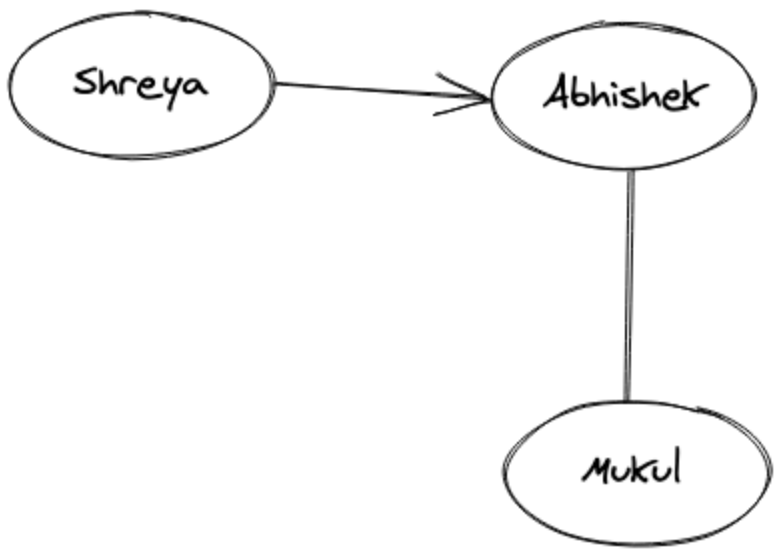
A pictorial representation of another undirected graph is given below:



3. Directed Graphs

What if the relation between the two people is something like, "Shreya" know "Abhishek" but "Abhishek" doesn't know "Shreya". This type of relationship is one-way, and it does include a direction. The edges with arrows basically denote the direction of the relationship and such graphs are known as directed graphs.

A pictorial representation of the graph is given below:

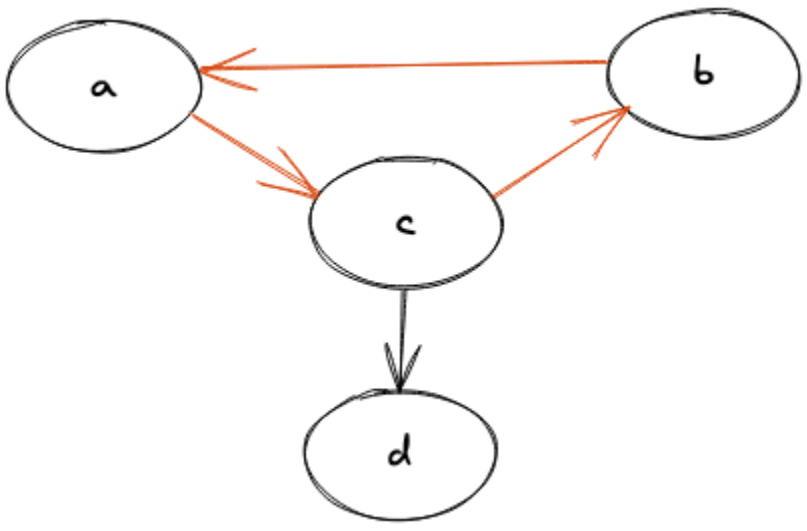


It can also be noted that the edge from "Shreya" to "Abhishek" is an outgoing edge for "Shreya" and an incoming edge for "Abhishek".

4. Cyclic Graph

A graph that contains at least one node that traverses back to itself is known as a cyclic graph. In simple words, a graph should have at least one cycle formation for it to be called a cyclic graph.

A pictorial representation of a cyclic graph is given below:

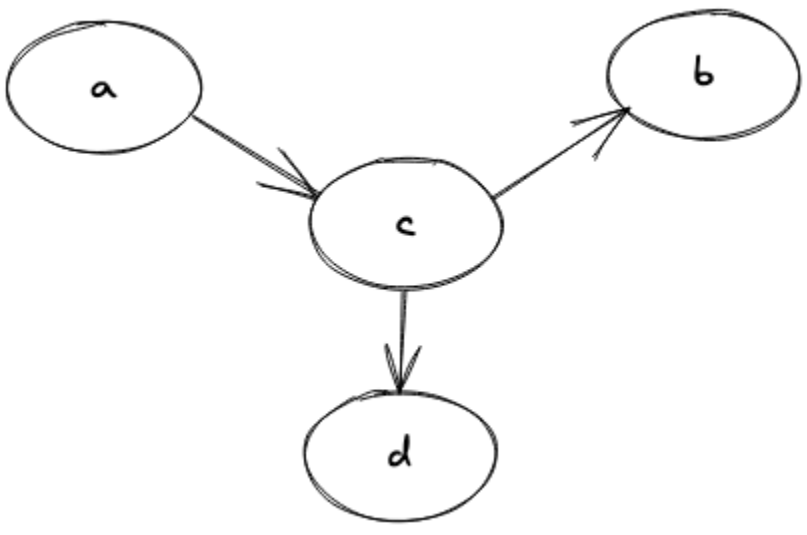


It can be easily seen that there exists a cycle between the nodes (a, b, c) and hence it is a cyclic graph.

5. Acyclic Graph

A graph where there's no way we can start from one node and can traverse back to the same one, or simply doesn't have a single cycle is known as an acyclic graph.

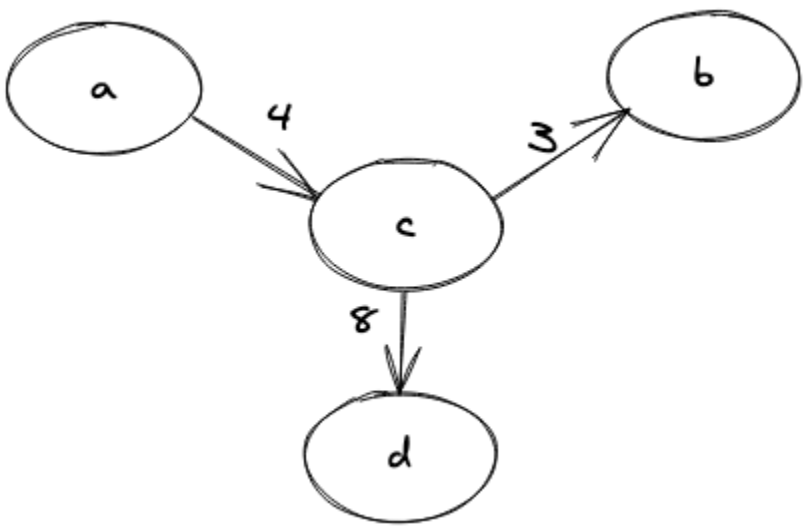
A pictorial representation of an acyclic graph is given below:



6. Weighted Graph

When the edge in a graph has some weight associated with it, we call that graph as a weighted graph. The weight is generally a number that could mean anything, totally dependent on the relationship between the nodes of that graph.

A pictorial representation of the weighted graph is given below:

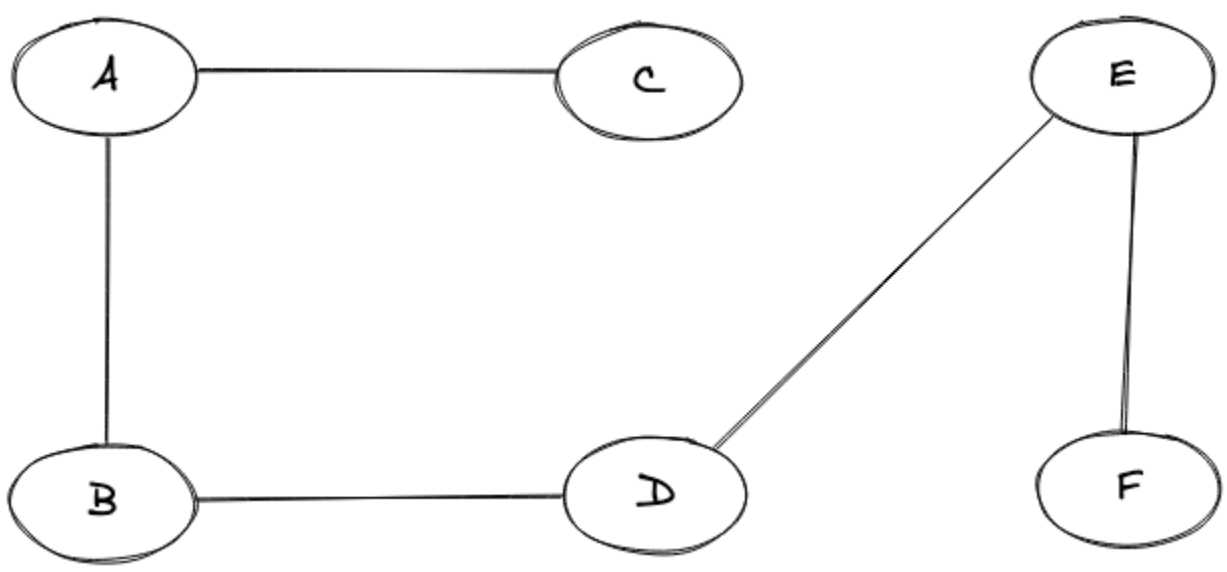


It can also be noted that if any graph doesn't have any weight associated with it, we simply call it an unweighted graph.

7. Connected Graph

A graph where we have a path between every two nodes of the graph is known as a connected graph. A path here means that we are able to traverse from a node "A" to say any node "B". In simple terms, we can say that if we start from one node of the graph we will always be able to traverse to all the other nodes of the graph from that node, hence the connectivity.

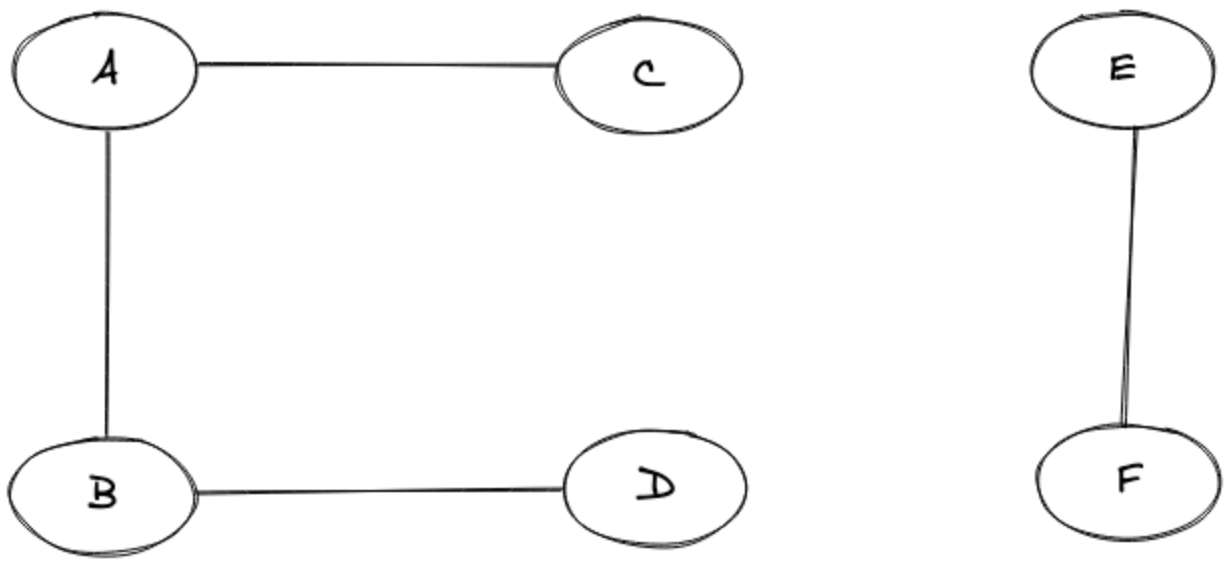
A pictorial representation of the connected graph is given below:



8. Disconnected Graph

A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.

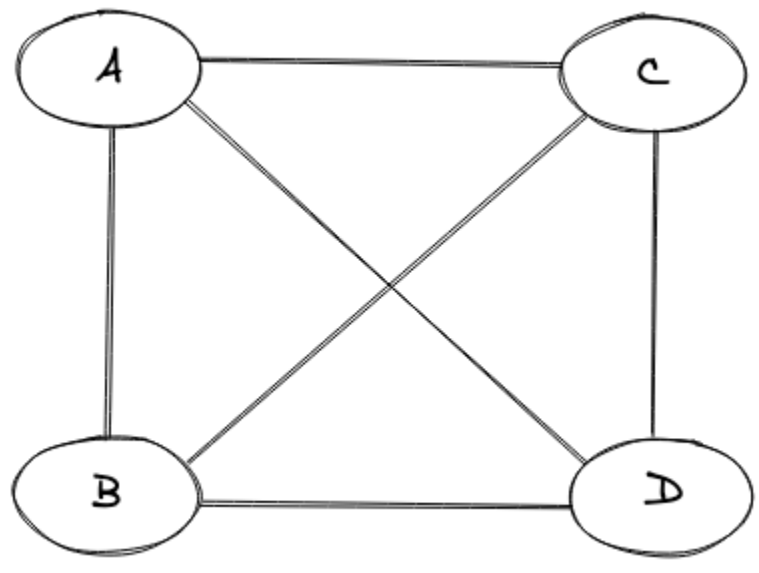
A pictorial representation of the disconnected graph is given below:



9. Complete Graph

A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph.

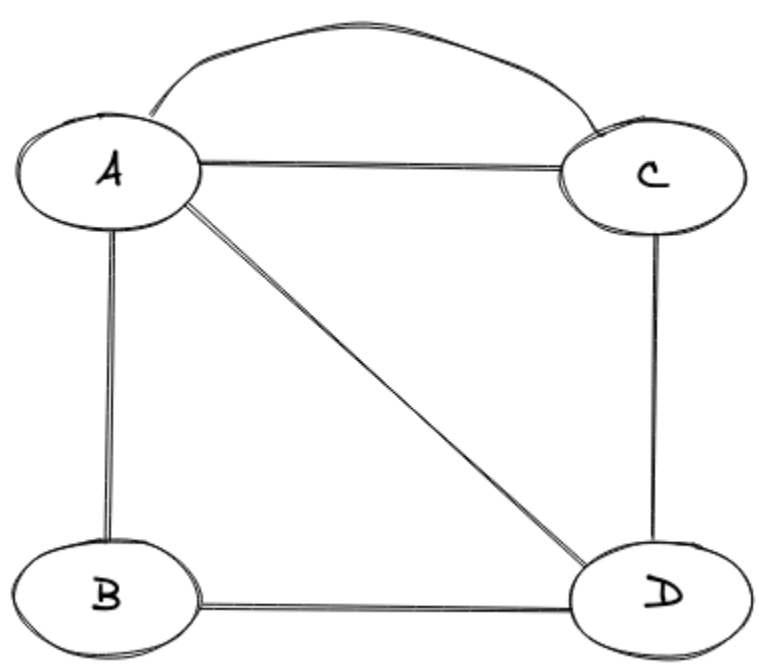
A pictorial representation of the complete graph is given below:



10. Multigraph

A graph is said to be a multigraph if there exist two or more than two edges between any pair of nodes in the graph.

A pictorial representation of the multigraph is given below:



Commonly Used Graph Terminologies

- **Path** - A sequence of alternating nodes and edges such that each of the successive nodes are connected by the edge.

- **Cycle** - A path where the starting and the ending node is the same.
- **Simple Path** - A path where we do not encounter a vertex again.
- **Bridge** - An edge whose removal will simply make the graph disconnected.
- **Forest** - A forest is a graph without cycles.
- **Tree** - A connected graph that doesn't have any cycles.
- **Degree** - The degree in a graph is the number of edges that are incident on a particular node.
- **Neighbour** - We say vertex "A" and "B" are neighbours if there exists an edge between them.

Conclusions

- We learned about what a graph is with a help of an undirected graph between different people.
- We learned how many types of graphs are there in total.
- We learned about the common terminologies that we use while talking about graphs and their subparts.

Graph Representations - Adjacency Matrix and List

There are two ways in which we represent graphs, these are:

- Adjacency Matrix
- Adjacency List

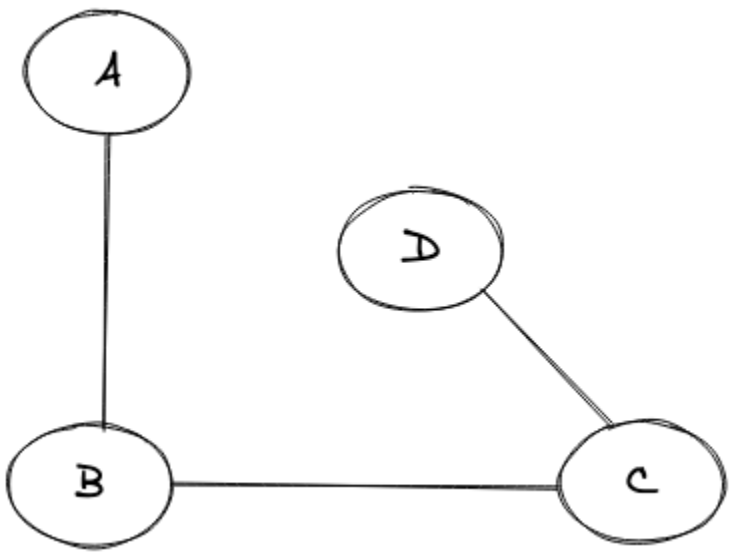
Both these have their advantages and disadvantages. In this tutorial, we will cover both of these graph representation along with how to implement them.

Adjacency Matrix

Adjacency matrix representation makes use of a matrix (table) where the **first row and first column of the matrix denote the nodes** (vertices) of the graph. The **rest of the cells contains either 0 or 1** (can contain an associated **weight w** if it is a weighted graph).

Each **row X column** intersection points to a cell and the value of that cell will help us in determining that whether the vertex denoted by the row and the vertex denoted by the column are connected or not. If the value of the cell for v1 X v2 is equal to 1, then we can conclude that these two vertices v1 and v2 are connected by an edge, else they aren't connected at all.

Consider the given graph below:

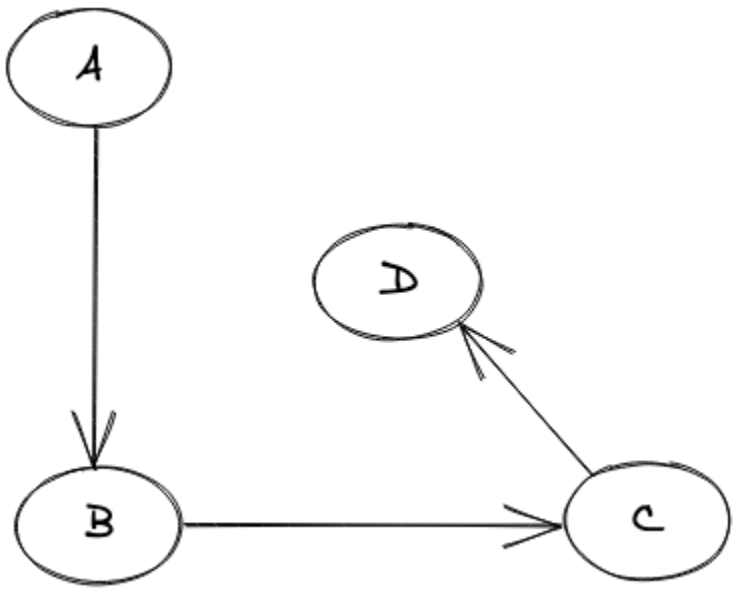


The graph shown above is an undirected one and the adjacency matrix for the same looks as:

-	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

The above matrix is the adjacency matrix representation of the graph shown above. If we look closely, we can see that the matrix is symmetric. Now let's see how the adjacency matrix changes for a directed graph.

Consider the given graph below:



For the directed graph shown above the adjacency matrix will look something like this:

-	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	1
D	0	0	0	0

Implementation of Adjacency Matrix

The structure (constructor in Java) for the adjacency matrix will look something like this:

```
public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

}
```

Copy

It should also be noted that we have two class-level variables, like:

```
int vertex;

int[][] matrix;
```

Copy

We have a constructor above named AdjacencyMatrix which takes the count of the number of the vertices that are present in the graph and then assigns our global vertex variable that value and also creates a 2D matrix of the same size. Now since our structure part is complete, we are simply left with adding the edges together, and the way we do that is:

```
public void addEdge(int start,int destination){

        matrix[start][destination] = 1;

        matrix[destination][start] = 1; // for un-directed graph

}
```

Copy

In the above `addEdge` function we also assigned 1 for the direction from the destination to the start node, as in this code we looked at the example of the undirected graph, in which the relationship is a two-way process. If it had been a directed graph, then we can simply make this value equal to 0, and we would have a valid adjacency matrix.

Now the only thing left is to print the graph.

```
public void printGraph(){

        System.out.println("Adjacency Matrix : ");

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

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

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

}
```

Copy

The entire code looks something like this:

```
public class AdjacencyMatrix {

    int vertex;

    int[][] matrix;


    // constructor

    public AdjacencyMatrix(int vertex){

        this.vertex = vertex;

        matrix = new int[vertex][vertex];

    }


    public void addEdge(int start,int destination){
```

```
        matrix[start][destination] = 1;

        matrix[destination][start] = 1;

    }


    public void printGraph(){

        System.out.println("Adjacency Matrix : ");

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

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

                System.out.print(matrix[i][j]+ " ");

            }

            System.out.println();

        }

    }


    public static void main(String[] args) {

        AdjacencyMatrix adj = new AdjacencyMatrix(4);

        adj.addEdge(0,1); // 0 as the array is 0-indexed

        adj.addEdge(1,2);

        adj.addEdge(2,3);

        adj.printGraph();

    }

}
```

Copy

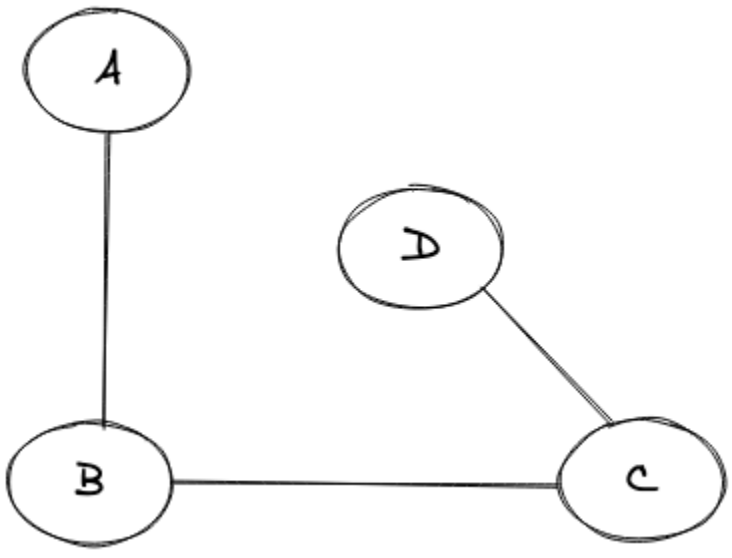
The output of the above looks like:

```
Adjacency Matrix :

0 1 0 0

1 0 1 0

0 1 0 1

0 0 1 0
```

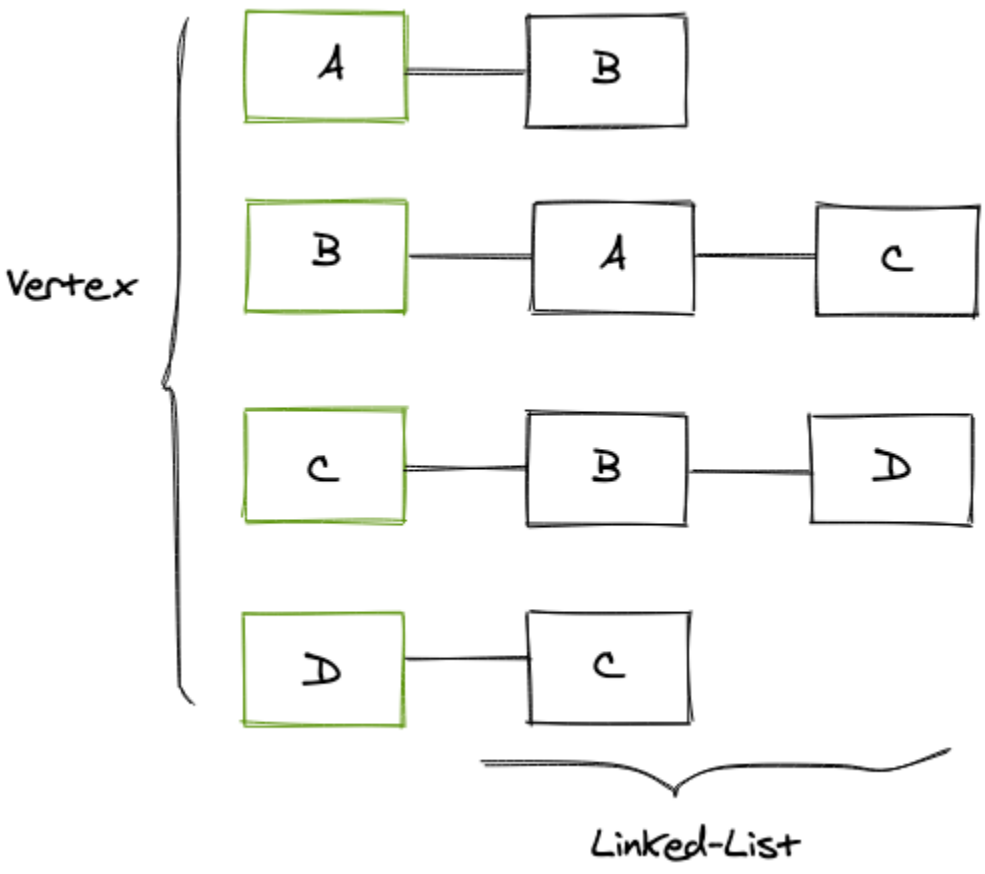
Adjacency List

In the adjacency list representation, we have an array of linked-list where the size of the array is the number of the vertex (nodes) present in the graph. Each vertex has its own linked-list that contains the nodes that it is connected to.

Consider the graph shown below:

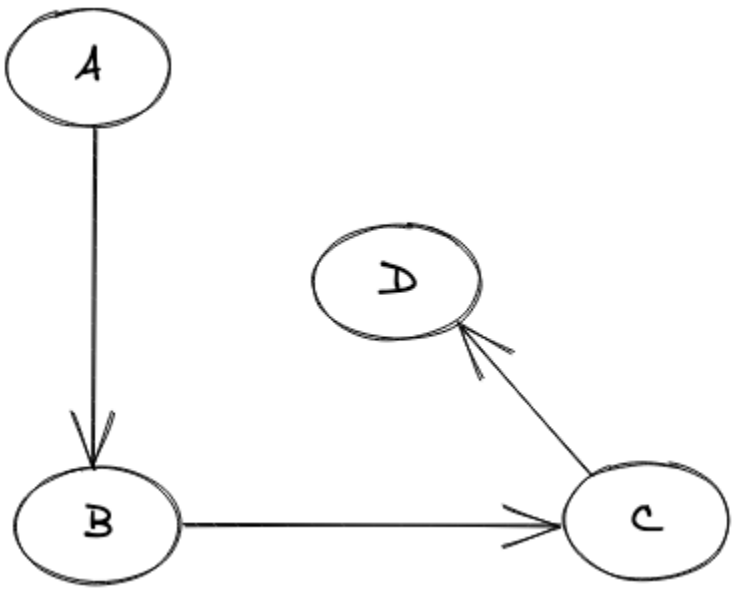


The above graph is an undirected one and the Adjacency list for it looks like:

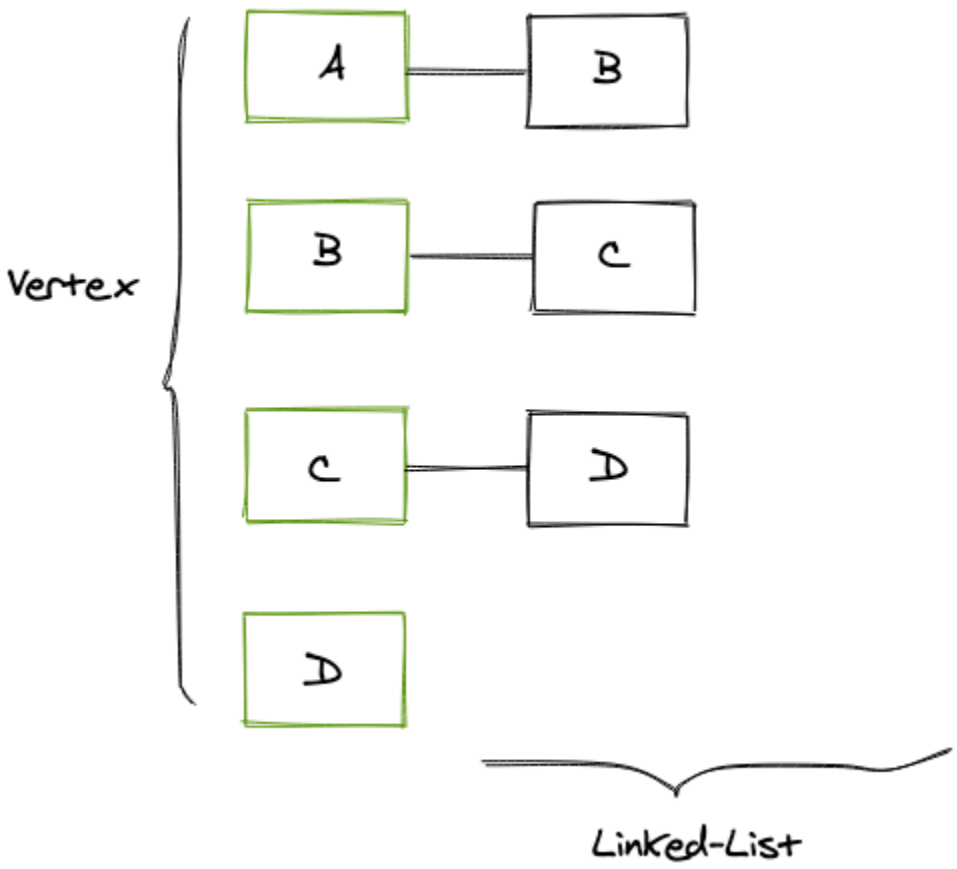


The first column contains all the vertices we have in the graph above and then each of these vertices contains a linked list that in turn contains the nodes that each vertex is connected to. For a directed graph the only change would be that the linked list will only contain the node on which the incident edge is present.

Consider the graph shown below:



The above graph is a directed one and the Adjacency list for this looks like:



Implementation of Adjacency List

The structure (constructor in Java) for the adjacency list will look something like this:

```
public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

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

            list[i] = new LinkedList<Integer>();

        }

}
```

Copy

The above constructor takes the number of vertices as an argument and then assigns the class level variable this value, and then we create an array of LinkedList of the size of the vertices present in the graph. Finally, we create an empty LinkedList for each item of this array of LinkedList.

It should also be noted that we have two class-level variables, like:

```
int vertex;

LinkedList<Integer> []list;
```

Copy

Now we have laid the foundations and the only thing left is to add the edges together, we do that like this:

```
public void addEdge(int start,int destination){

        list[start].addFirst(destination);
```

```
        list[destination].addFirst(start); // un-directed graph

}
```

Copy

We are taking the vertices from which an edge starts and ends, and we are simply inserting the destination vertex in the LinkedList of the start vertex and vice-versa (as it is for the undirected graph).

Now the only thing left is to print the graph.

```
public void printGraph(){

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

            if(list[i].size()>0) {

                System.out.print("Node " + i + " is connected to: ");

                for (int j = 0; j < list[i].size(); j++) {

                    System.out.print(list[i].get(j) + " ");

                }

                System.out.println();

            }

        }

}
```

Copy

The entire code looks something like this:

```
import java.util.LinkedList;


public class AdjacencyList {

    int vertex;

    LinkedList<Integer> []list;


    public AdjacencyList(int vertex){

        this.vertex = vertex;

        list = new LinkedList[vertex];

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

            list[i] = new LinkedList<Integer>();

        }
```

```
    }


    public void addEdge(int start,int destination){
        list[start].addFirst(destination);
        list[destination].addFirst(start); // un-directed graph
    }


    public void printGraph(){
        for (int i = 0; i < vertex ; i++) {
            if(list[i].size()>0) {
                System.out.print("Node " + i + " is connected to: ");
                for (int j = 0; j < list[i].size(); j++) {
                    System.out.print(list[i].get(j) + " ");
                }
                System.out.println();
            }
        }
    }


    public static void main(String[] args) {
        AdjacencyList adl = new AdjacencyList(4);
        adl.addEdge(0,1);
        adl.addEdge(1,2);
        adl.addEdge(2,3);
        adl.printGraph();
    }


}
```

Copy

The output of the above looks like:

```
Node 0 is connected to: 1

Node 1 is connected to: 2 0

Node 2 is connected to: 3 1

Node 3 is connected to: 2
```

Conclusion

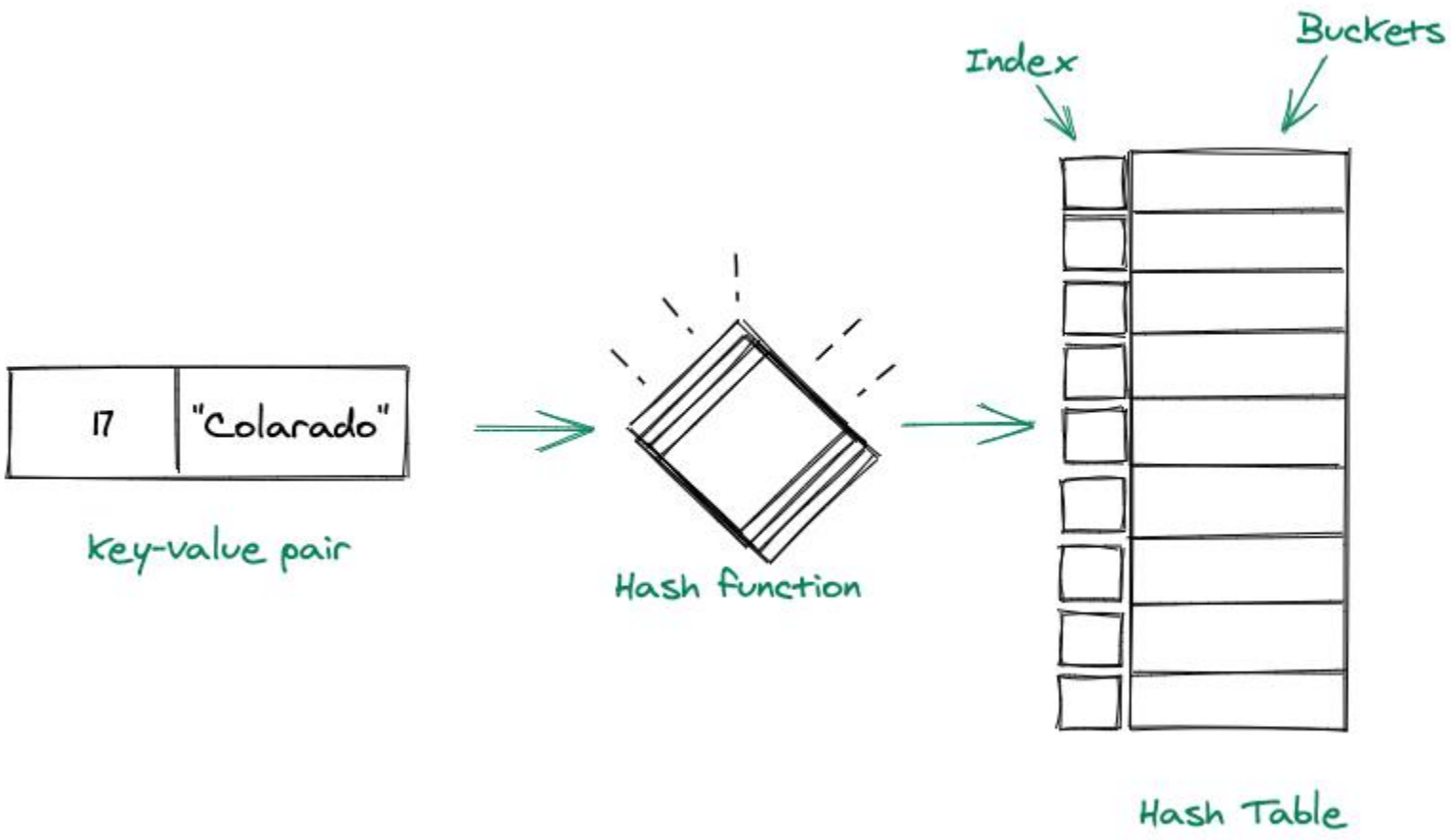
- We learned how to represent the graphs in programming, via adjacency matrix and adjacency lists.

Hash Table Data Structure

A Hash table is a data structure that is used to store the data in key-value pairs.

- Each key in the hash table is mapped to a value that is present in the hash table.
- The key present in the hash table are used to index the values, as we take this key and pass it to a hash function which in turn performs some arithmetic operation on it. The resulting value we get after the operation is the index of the hash table where we store the key-value pairs in the table.
- Internally a hash table contains buckets, and the location of which bucket to use for a particular key is determined by the key's hash function.

Consider the visual representation below:



Components of a Hash Table

A hash table comprises two components in total, these are:

Hash function:

- A hash function is used to determine the index of the key-value pair.
- It is always recommended that we should choose a good hash function for creating a good hash table.
- Besides a good hash function, it should be a one-way function, i.e. we should be able to get the hash value from the key and not vice versa.
- Also, it should avoid producing the same hash for different keys.

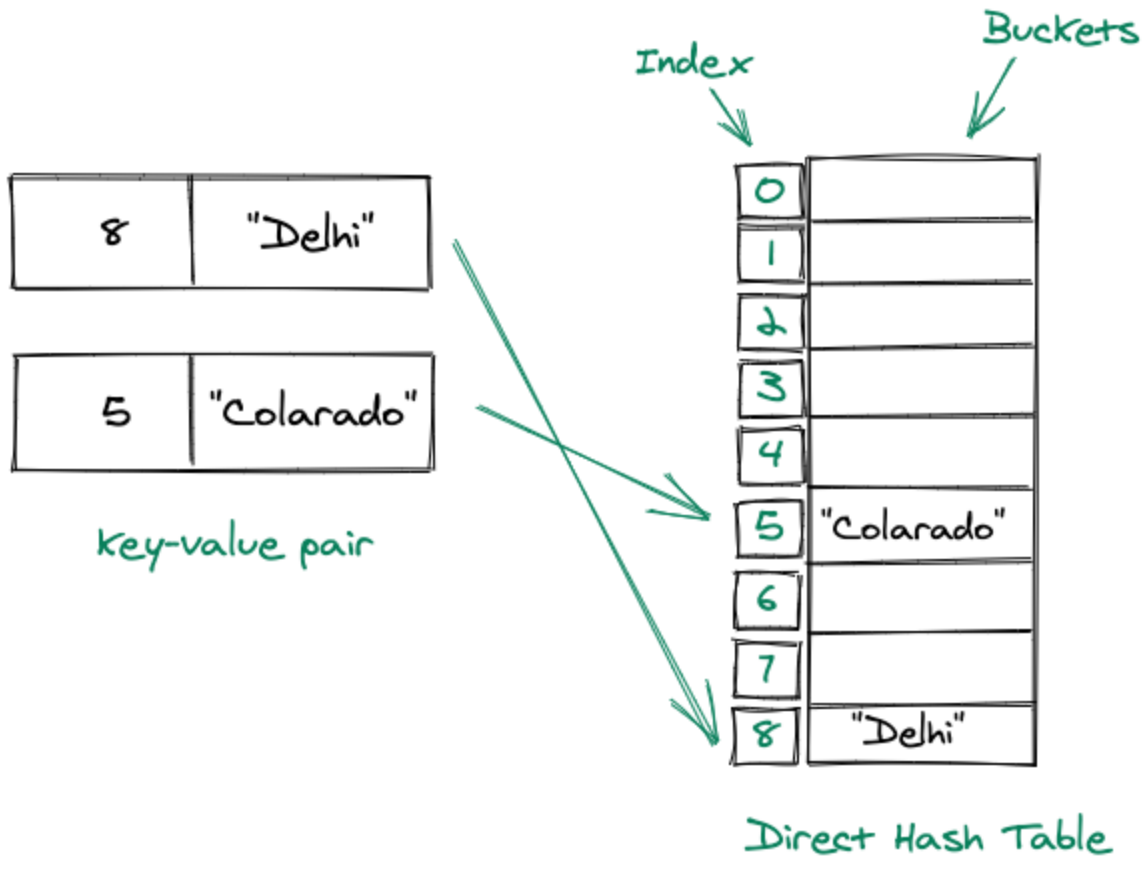
Arrays:

- The array(buckets) are used to hold the key-value pairs.
- The size of the array should be set accordingly to the number of key-value pairs we will have.

Direct Addressing

It is a technique in which we make use of direct address tables to map the values with their keys. It uses the keys as indexes of the buckets and then store the values at those bucket locations. Though direct addressing does facilitate fast searching, fast inserting and fast deletion operations, but at a cost.

Consider the pictorial representation below:



Advantages of Direct Address Table:

- **Insertion in O(1) time:** Inserting an element in a direct address table is the same as inserting an element in an array, hence we can do that in **O(1) time** as we already know the index(via key).
- **Deletion in O(1) time:** Deleting an element from a direct address table is the same as deleting from an array, hence the **O(1) time**.
- **Searching in O(1) time:** Searching an element takes linear time**(O(1))** as we can easily access an element in an array in linear time if we already know the index of that element.

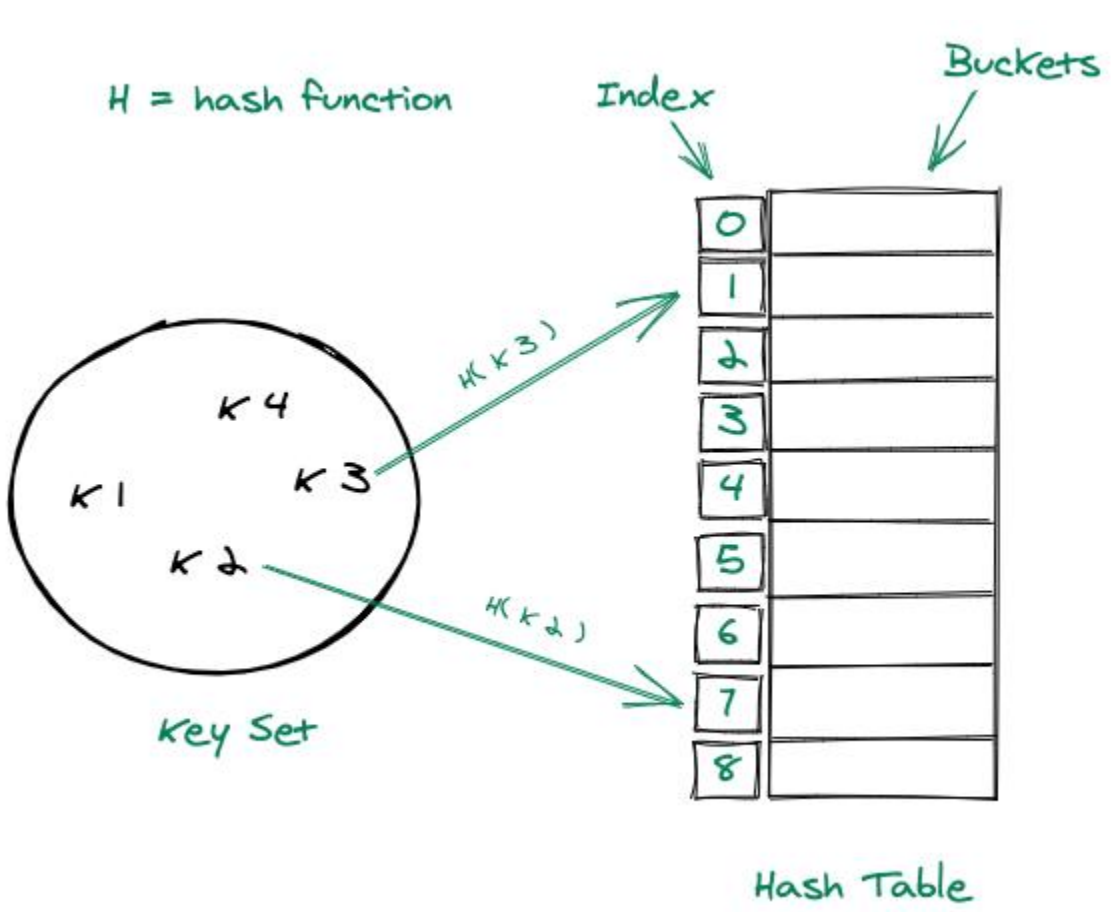
Disadvantages of Direct Address Table:

- It is not recommended using the direct address table if the key values are **very large**.
- It cannot handle the case where two keys are equal and contain different data.

Hash Table

Direct addressing has serious disadvantages, making it not suitable for the practical usage of current world scenarios, which is why we make use of Hash Tables. Unlike a direct address table where we take the key as indices of the address table, we use keys in different way. We process these keys via a hash function and the result we get from that is basically the location of the bucket where we store our data. A Hash Table has different implementations in different languages, like in Java we have **Hash Table**, **HashMap** and many more, in python we have **dictionaries**, but no matter how they are implemented the core functionality remains the same.

Consider the pictorial representation below:



Advantages of Hash Table:

- While the advantages of hash table is same when we talk about **insertion**, **deletion** or **searching** of an element, there's a huge advantage that hash table has over address table, which is that it maintains the size constraint. Let us consider a **key = 7898789**, which in turn is a large number, if we insert this in a direct address table, then we are **wasting too much space** as we will have to find this location(key) and then insert the value at that location, but in case of a hash table we can process this **key via a hash function**, say it yields us = 17, now we are only left with inserting at position(17) of the hash table.

Disadvantages of Hash Table:

- A situation might arise when we get the same bucket location for different keys via our hash function, this situation is known as collision. Though we can improve the hash quality, but we can't guarantee that collisions won't take place.

Handling Collisions in Hash Table:

There are multiple ways with which we can handle collisions. Some of them are:

A good hash function:

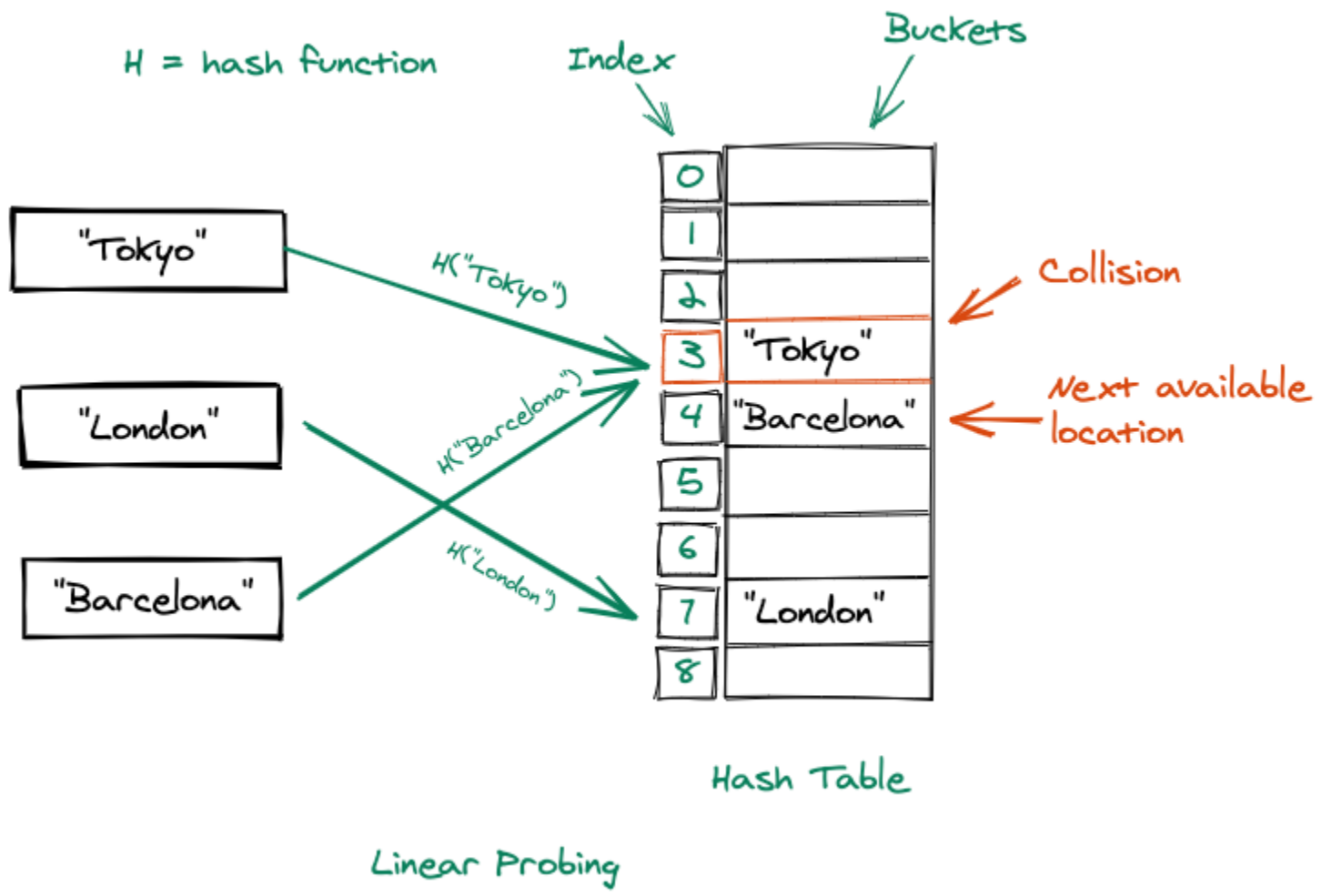
A good hash function can mean many things at the same time, but we can conclude that a hash function is considered as a good hash function, if:

- It minimizes the amount of collisions as much as possible.
- It should not generate the bucket locations that are larger than the hash table, in that case we will be wasting too much space.
- The bucket locations generated should neither be too far apart and too much closer.

Linear Probing:

It is a technique that makes sure that if we have a collision at a particular bucket, then we check for next available bucket location(just below it), and insert our data in that bucket.

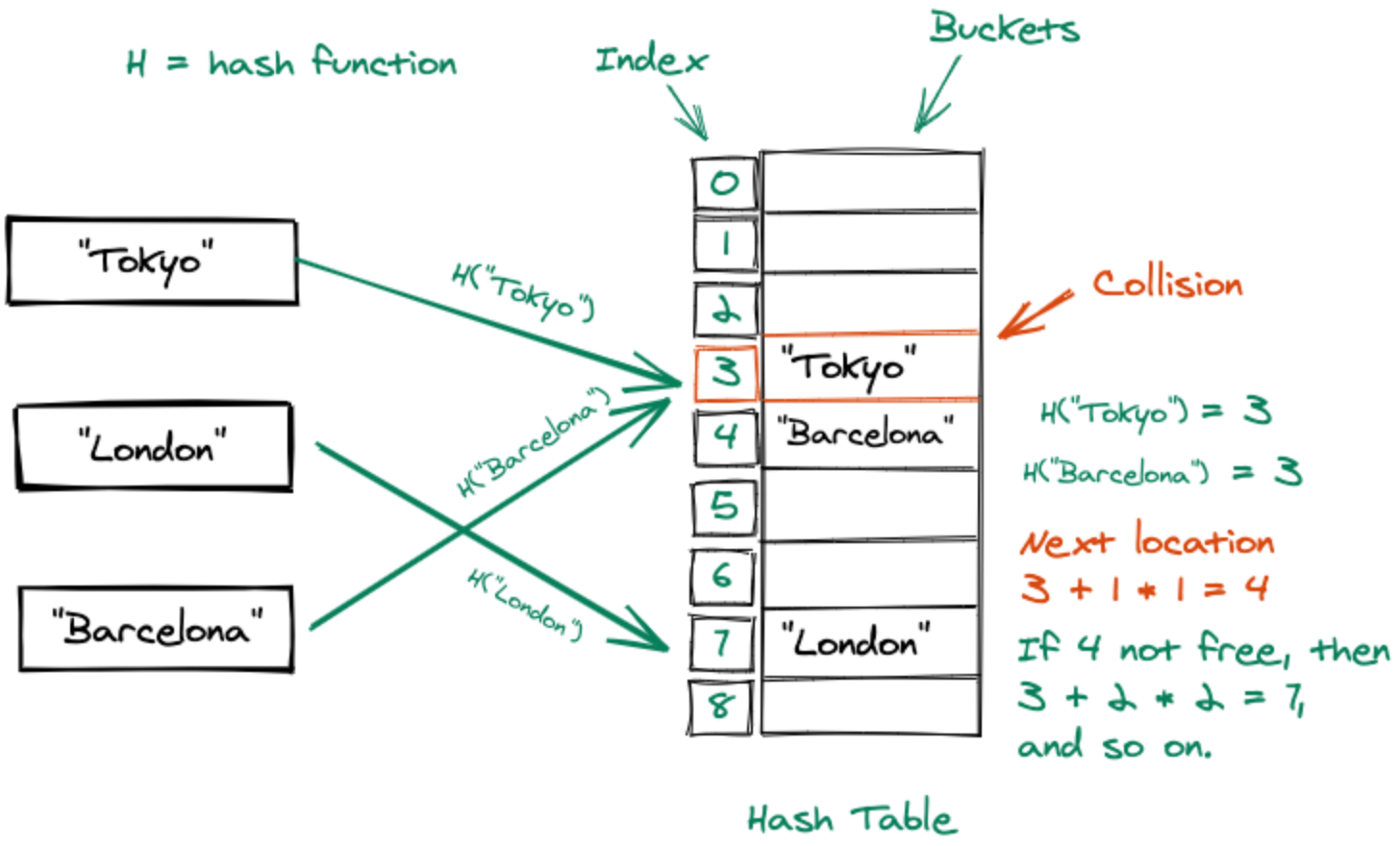
Consider the pictorial representation shown below:



Quadratic Probing:

In quadratic probing the next bucket location is decided using the formula: **h(k, i) = (h?(k) + x*i + y*i^2)** where x and y are constants. Another way to look at this to say that the distance between the next location is increased quadratically.

Consider the pictorial representation shown below:

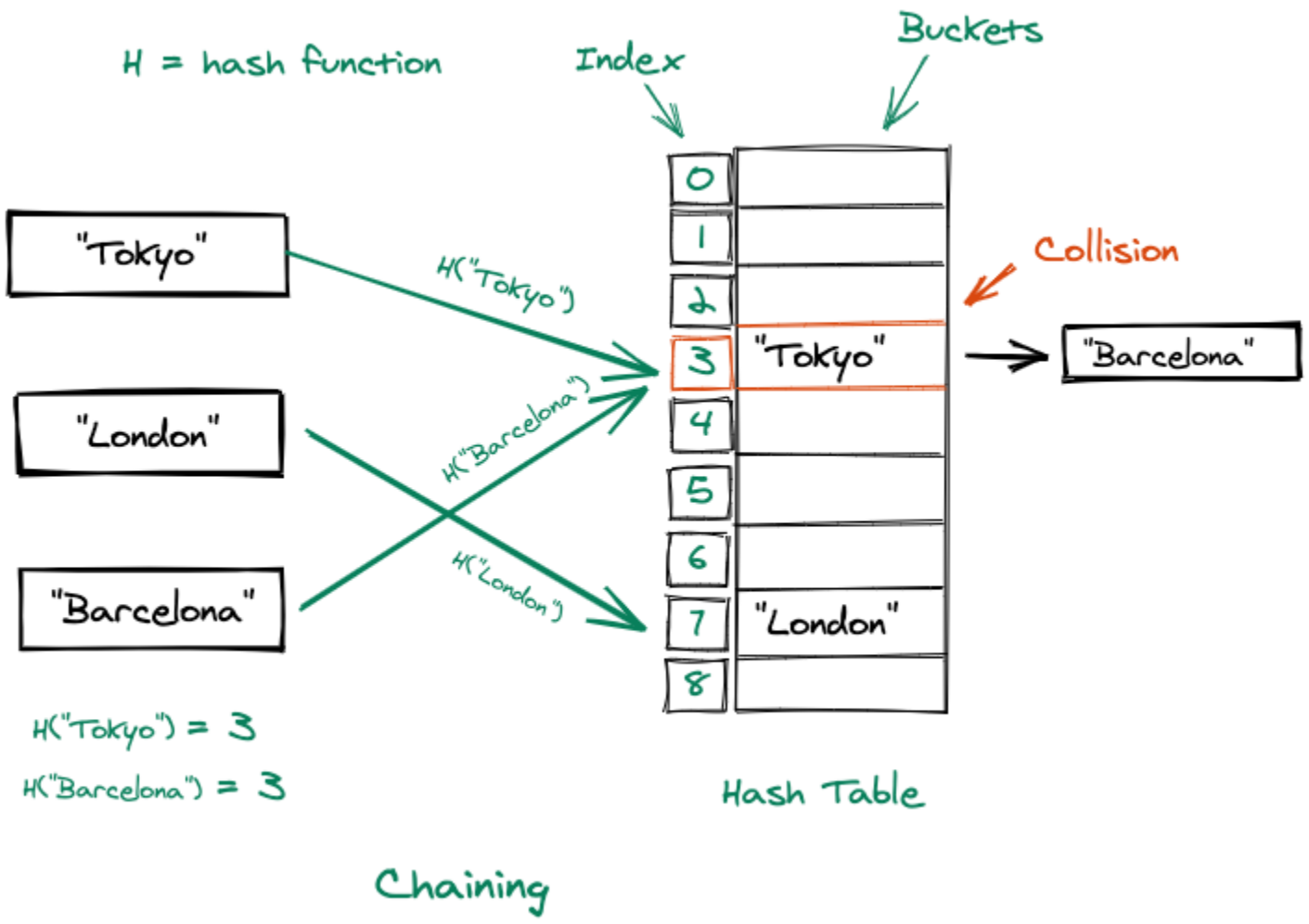


Quadratic Probing

Chaining:

In this technique of collision handling, if at a particular bucket location a collision occurs then at that location we simply create a linked list, and the value will be added as the next node of the list. Hash table becomes an array of linked list.

Consider the pictorial representation shown below:



Chaining

Applications of Hash Tables:

Pattern Matching:

Hash Tables search time complexity makes it a perfect candidate for finding a pattern in a pool of strings.

Compilers:

Compilers make use of hash tables to store the keywords and other identifiers to store the values of a programming language.

File Systems:

The mapping between the name of the file in our file system and the path of that file is stored via a map that intern makes use of a hash table.

Conclusions

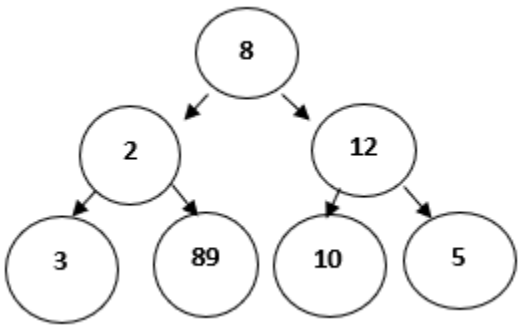
- We learned what a Hash table is, what are its main components.
- Then we learned about direct address tables, followed by a detailed explanation of hash tables.
- Then we talked about different techniques of handling collisions in hash tables, followed by the real-world applications of a hash table.

Greedy Approach or Technique

As the name implies, this is a simple approach which tries to find the **best** solution at every step. Thus, it aims to find the local optimal solution at every step so as to find the global optimal solution for the entire problem.

Consider that there is an **objective function** that has to be optimized (maximized/ minimized). This approach makes greedy choices at each step and makes sure that the objective function is optimized.

The greedy algorithm has only one chance to compute the optimal solution and thus, cannot go back and look at other alternate solutions. However, in many problems, this strategy fails to produce a global optimal solution. Let's consider the following binary tree to understand how a basic greedy algorithm works:



For the above problem the objective function is:

To find the path with largest sum.

Since we need to maximize the objective function, Greedy approach can be used. Following steps are followed to find the solution:

Step 1: Initialize **sum = 0**

Step 2: Select the root node, so its value will be added to sum, **sum = 0+8 = 8**

Step 3: The algorithm compares nodes at next level, selects the largest node which is **12**, making the **sum = 20**.

Step 4: The algorithm compares nodes at the next level, selects the largest node which is **10**, making the **sum = 30**.

Thus, using the greedy algorithm, we get **8-12-10** as the path. But this is not the optimal solution, since the path **8-2-89** has the largest sum ie **99**.

This happens because the algorithm makes decision based on the information available at each step without considering the overall problem.

When to use Greedy Algorithms?

For a problem with the following properties, we can use the greedy technique:

- **Greedy Choice Property**: This states that a globally optimal solution can be obtained by locally optimal choices.
- **Optimal Sub-Problem**: This property states that an optimal solution to a problem, contains within it, optimal solution to the sub-problems. Thus, a globally optimal solution can be constructed from locally optimal sub-solutions.

Generally, **optimization problem**, or the problem where we have to find maximum or minimum of something or we have to find some optimal solution, greedy technique is used.

An optimization problem has two types of solutions:

- **Feasible Solution**: This can be referred as approximate solution (subset of solution) satisfying the objective function and it may or may not build up to the optimal solution.
- **Optimal Solution**: This can be defined as a feasible solution that either maximizes or minimizes the objective function.

Key Terminologies used in Greedy Algorithms

- **Objective Function**: This can be defined as the function that needs to be either maximized or minimized.
- **Candidate Set**: The global optimal solution is created from this set.
- **Selection Function**: Determines the best candidate and includes it in the solution set.
- **Feasibility Function**: Determines whether a candidate is feasible and can contribute to the solution.

Standard Greedy Algorithm

This algorithm proceeds step-by-step, considering one input, say `x`, at each step.

- If `x` gives a local optimal solution (`x` is feasible), then it is included in the partial solution set, else it is discarded.
- The algorithm then goes to the next step and never considers `x` again.
- This continues until the input set is finished or the optimal solution is found.

The above algorithm can be translated into the following pseudocode:

```
Algorithm Greedy(a, n)    // n defines the input set
{
    solution= NULL;       // initialize solution set
     for i=1 to n do
     {
          x = Select(a); // Selection Function
          if Feasible(solution, x) then  // Feasibility solution
               solution = Union (solution, x);    // Include x in the
solution set
     }
     return solution;
}
```

Copy

Advantages of Greedy Approach/Technique

- This technique is easy to formulate and implement.
- It works efficiently in many scenarios.
- This approach minimizes the time required for generating the solution.

Now, let's see a few disadvantages too,

Disadvantages of Greedy Approach/Technique

- This approach does not guarantee a global optimal solution since it never looks back at the choices made for finding the local optimal solution.

Although we have already covered that which type of problem in general can be solved using greedy approach, here are a few popular problems which use greedy technique:

1. Knapsack Problem
2. Activity Selection Problem
3. Dijkstra’s Problem
4. Prim’s Algorithmfor finding Minimum Spanning Tree
5. Kruskal’s Algorithmfor finding Minimum Spanning Tree
6. Huffman Coding
7. Travelling Salesman Problem

Conclusion

Greedy Technique is best suited for applications where:

- Solution is required in real-time.
- Approximate solution is sufficient.

Activity Selection Problem

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have `n` activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

- It might not be possible to complete all the activities, since their timings can collapse.
- Two activities, say **i** and **j**, are said to be non-conflicting if `si >= fj` or `sj >= fi` where `si` and `sj` denote the starting time of activities **i** and **j** respectively, and `fi` and `fj` refer to the finishing time of the activities **i** and **j** respectively.
- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- `act[]` array containing all the activities.
- `s[]` array containing the starting time of all the activities.
- `f[]` array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- `sol[]` array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array `act[]` and add it to `sol[]` array.

Step 3: Repeat steps 4 and 5 for the remaining activities in `act[]`.

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the `sol[]` array.

Step 5: Select the next activity in `act[]` array.

Step 6: Print the `sol[]` array.

Activity Selection Problem Example

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) > f(a2)), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.

C. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

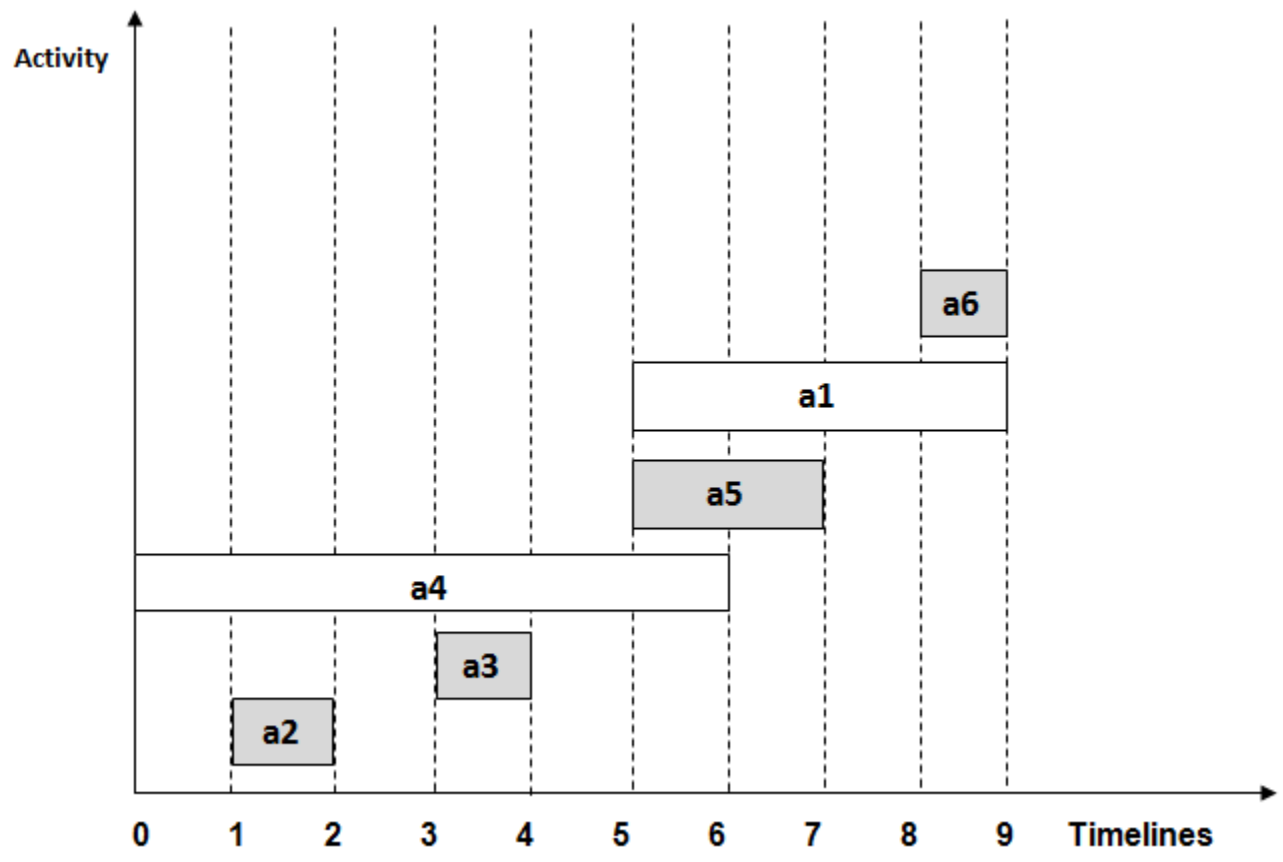
D. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5). Thus **sol = {a2, a3, a5, a6}**.

Step 6: At last, print the array sol[]

Hence, the execution schedule of maximum number of non-conflicting activities will be:

```
(1,2)
(3,4)
(5,7)
(8,9)
```



In the above diagram, the selected activities have been highlighted in grey.

Implementation of Activity Selection Problem Algorithm

Now that we have an overall understanding of the activity selection problem as we have already discussed the algorithm and its working details with the help of an example, following is the C++ implementation for the same.

Note: The algorithm can be easily written in any programming language.

```
#include <bits/stdc++.h>


using namespace std;

#define N 6          // defines the number of activities


// Structure represents an activity having start time and finish time.
struct Activity
{
    int start, finish;
};


// This function is used for sorting activities according to finish
time
bool Sort_activity(Activity s1, Activity s2)
{
    return (s1.finish< s2.finish);
}


/*   Prints maximum number of activities that can
     be done by a single person or single machine at a time.
*/
void print_Max_Activities(Activity arr[], int n)
{
    // Sort activities according to finish time
     sort(arr, arr+n, Sort_activity);


     cout<< "Following activities are selected \n";


    // Select the first activity
    int i = 0;
```

```
     cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";


    // Consider the remaining activities from 1 to n-1
    for (int j = 1; j < n; j++)
    {
     // Select this activity if it has start time greater than or equal
     // to the finish time of previously selected activity
          if (arr[j].start>= arr[i].finish)
          {
               cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ")
\n";
               i = j;
          }
    }
}


// Driver program
int main()
{
    Activity arr[N];
     for(int i=0; i<=N-1; i++)
     {
          cout<<"Enter the start and end time of "<<i+1<<" activity
\n";
          cin>>arr[i].start>>arr[i].finish;
    }


     print_Max_Activities(arr, N);
    return 0;
}
```

Copy

The program is executed using same inputs as that of the example explained above. This will help in verifying the resultant solution set with actual output.

```
Enter the start and end time of 1 activity
5 9
Enter the start and end time of 2 activity
1 2
Enter the start and end time of 3 activity
3 4
Enter the start and end time of 4 activity
0 6
Enter the start and end time of 5 activity
5 7
Enter the start and end time of 6 activity
8 9
Following activities are selected
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Time Complexity Analysis

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

- **Case 1**: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be $O(n)$
- **Case 2**: When a given set of activities is unsorted, then we will have to use the `sort()` method defined in **bits/stdc++** header file for sorting the activities list. The time complexity of this method will be $O(nlogn)$, which also defines complexity of the algorithm.

Real-life Applications of Activity Selection Problem

Following are some of the real-life applications of this problem:

- Scheduling multiple competing events in a room, such that each event has its own start and end time.
- Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
- Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.

Prim's Minimum Spanning Tree

In this tutorial we will cover another algorithm which uses greedy approach/technique for finding the solution.

Let's start with a real-life scenario to understant the premise of this algorithm:

1. A telecommunications organization, has offices spanned across multiple locations around the globe.

Figure 1

2. It has to use leased phone lines for connecting all these offices with each other.
3. The cost(in units) of connecting each pair of offices is different and is shown as follows:

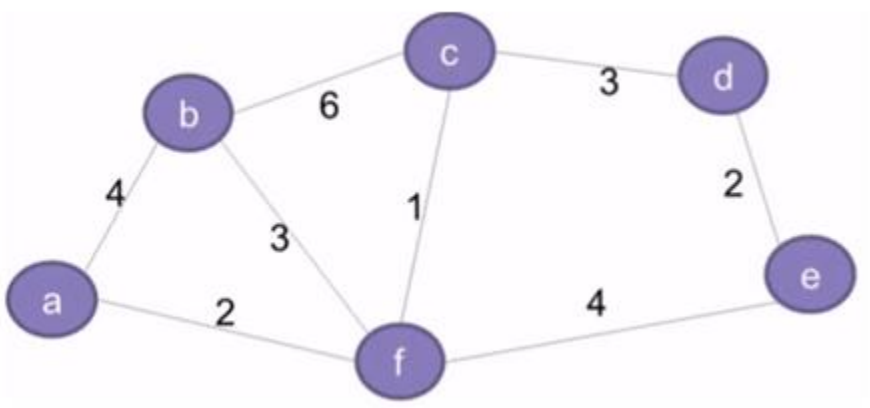


Figure 2

4. The organization, thus, wants to use minimum cost for connecting all its offices. This requires that all the offices should be connected using minimum number of leased lines so as to reduce the effective cost.
5. The solution to this problem can be implemented by using the concept of **Minimum Spanning Tree**, which is discussed in the subsequent section.
6. This tutorial also details the concepts related to Prim's Algorithm which is used for finding the minimum spanning tree for a given graph.

What is a Spanning Tree?

The network shown in the second figure basically represents a graph **G = (V, E)** with a set of vertices **V = {a, b, c, d, e, f}** and a set of edges **E = { (a,b), (b,c), (c,d), (d,e), (e,f), (f,a), (b,f), (c,f) }**. The graph is:

- Connected (there exists a path between every pair of vertices)

- Undirected (the edges do no have any directions associated with them such that (a,b) and (b,a) are equivalent)
- Weighted (each edge has a weight or cost assigned to it)

A spanning tree $G' = (V, E')$ for the given graph G will include:

- All the vertices (V) of G
- All the vertices should be connected by minimum number of edges (E') such that $E' \subset E$
- G' can have maximum $n-1$ edges, where n is equal to the total number of edges in G
- G' should not have any cycles. This is one of the basic differences between a tree and graph that **a graph can have cycles, but a tree cannot**. Thus, a tree is also defined as an **acyclic graph**.

Following is an example of a spanning tree for the above graph. Please note that only the highlighted edges are included in the spanning tree,

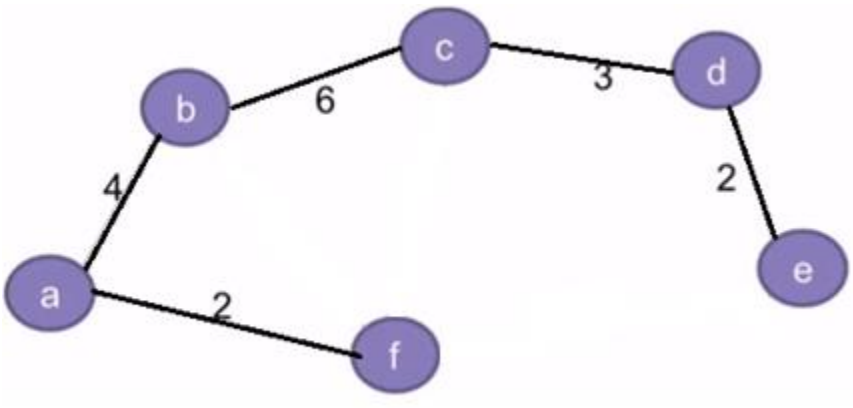


Figure 3

Also, there can be multiple spanning trees possible for any given graph. For eg: In addition to the spanning tree in the above diagram, the graph can also have another spanning tree as shown below:

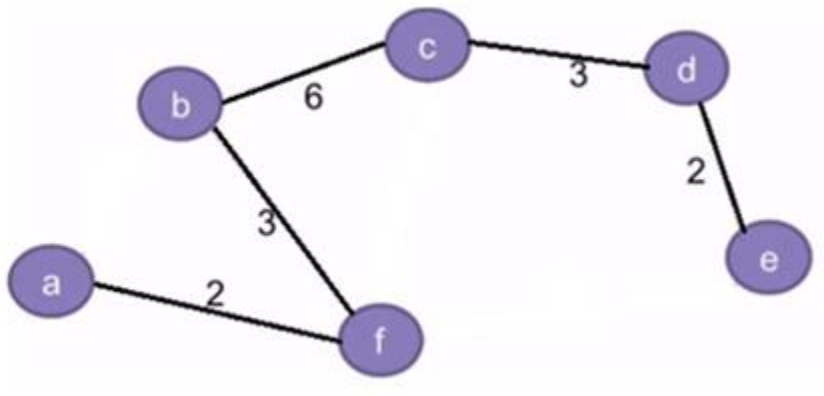


Figure 4

By convention, the total number of spanning trees for a given graph can be defined as:

${}^{n}C_{m} = n!/(m!*(n-m)!)$, where,

- n is equal to the total number of edges in the given graph
- m is equal to the total number of edges in the spanning tree such that $m <= (n-1)$.

Hence, the total number of spanning trees(S) for the given graph(second diagram from top) can be computed as follows:

- **n = 8**, for the given graph in Fig. 2
- **m = 5**, since its corresponding spanning tree can have only 5 edges. Adding a 6th edge can result in the formation of cycles which is not allowed.
- So, **$S = {}^{n}C_{m} = {}^{8}C_{5} = 8!/ (5! * 3!) = 56$**, which means that **56** different variations of spannig trees can be created for the given graph.

What is a Minimum Spanning Tree?

The cost of a spanning tree is the total of the weights of all the edges in the tree. For example, the cost of spanning tree in Fig. 3 is **(2+4+6+3+2) = 17** units, whereas in Fig. 4 it is **(2+3+6+3+2) = 16** units.

Since we can have multiple spanning trees for a graph, each having its own cost value, the objective is to find the spanning tree with minimum cost. This is called a **Minimum Spanning Tree(MST)**.

Note: There can be multiple minimum spanning trees for a graph, if any two edges in the graph have the same weight. However, if each edge has a distinct weight, then there will be only one minimum spanning tree for any given graph.

Problem Statement for Minimum Spanning Tree

Given a weighted, undirected and connected graph **G**, the objective is to find the minimum spanning tree **G'** for G.

Apart from the Prim's Algorithm for minimum spanning tree, we also have Kruskal's Algorithm for finding minimum spanning tree.

However, this tutorial will only discuss the fundamentals of **Prim's Algorithm.**

Since this algorithm aims to find the spanning tree with minimum cost, it uses **greedy approach** for finding the solution.

As part of finding the or creating the minimum spanning tree fram a given graph we will be following these steps:

- Initially, the tree is empty.
- The tree starts building from a random source vertex.
- A new vertex gets added to the tree at every step.
- This continues till all the vertices of graph are added to the tree.

Input Data will be:

A **Cost Adjacency Matrix** for out graph **G**, say `cost`

Output will be:

A Spanning tree with minimum total cost

Algorithm for Prim's Minimum Spanning Tree

Below we have the complete logic, stepwise, which is followed in prim's algorithm:

Step 1: Keep a track of all the vertices that have been visited and added to the spanning tree.

Step 2: Initially the spanning tree is empty.

Step 3: Choose a random **vertex**, and add it to the spanning tree. This becomes the **root node**.

Step 4: Add a new vertex, say **x**, such that

a. **x** is not in the already built spanning tree.

b. **x** is connected to the built spanning tree using minimum weight edge. (Thus, **x** can be adjacent to any of the nodes that have already been added in the spanning tree).

c. Adding **x** to the spanning tree should not form cycles.

Step 5: Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.

Step 6: Print the total cost of the spanning tree.

Example for Prim's Minimum Spanning Tree Algorithm

Let's try to trace the above algorithm for finding the Minimum Spanning Tree for the graph in Fig. 2:

Step A:

a. Define `key[]` array for storing the key value(or cost) of every vertex. Initialize this to ∞(infinity) for all the vertices

b. Define another array `booleanvisited[]` for keeping a track of all the vertices that have been added to the spanning tree. Initially this will be **0** for all the vertices, since the spanning tree is empty.

c. Define an array `parent[]` for keeping track of the parent vertex. Initialize this to **-1** for all the vertices.

d. Initialize minimum cost, **minCost = 0**

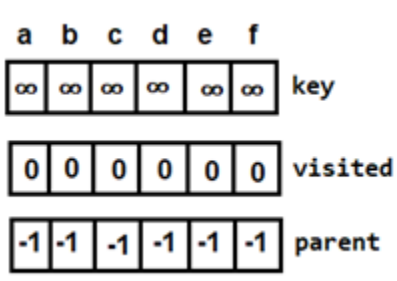


Figure 5: Initial arrays when tree is empty

Step B:

Choose any random vertex, say **f** and set `key[f]=0`.

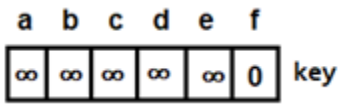


Figure 6: Setting key value of root node

Since its key value is minimum and it is not visited, add **f** to the spanning tree.

Figure 7: Root Node

Also, update the following:

- `minCost = 0 + key[f] = 0`
- This is how the `visited[]` array will look like:

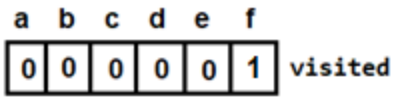


Figure 8: visited array after adding the root node

- Key values for all the adjacent vertices of **f** will look like this(key value is nothing but the cost or the weight of the edge, for **(f,d)** it is still infinity because they are not directly connected):

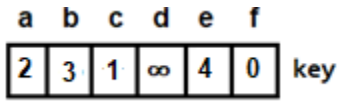


Figure 9: key array after adding the root node

Note: There will be no change in the `parent[]` because **f** is the root node.

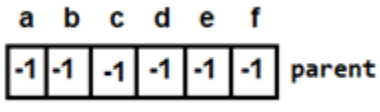


Figure 10: parent array after adding the root node

Step C:

The arrays `key[]` and `visited[]` will be searched for finding the next vertex.

- **f** has the minimum key value but will not be considered since it is already added (`visited[f]==1`)
- Next vertex having the minimum key value is **c**. Since `visited[c]==0`, it will be added to the spanning tree.

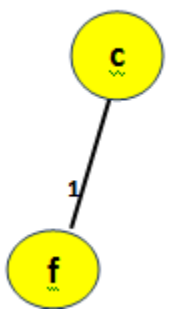


Figure 11: Adding vertex c

Again, update the following:

- `minCost = 0 + key[c] = 0 + 1 = 1`
- This is how the `visited[]` array will look like:

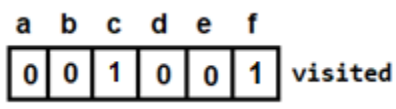


Figure 12: visited array after adding vertex c

- And, the `parent[]` array (**f** becomes the parent of **c**):

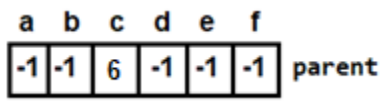


Figure 13: parent array after adding the root node

- For every adjacent vertex of **c**, say **v**, values in `key[v]` will be updated using the formula:

  `key[v] = min(key[v], cost[c][v])`

  Thus the `key[]` array will become:

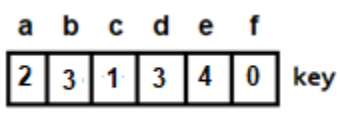


Figure 14: key array after adding the root node

Step D:

Repeat the Step C for the remaining vertices.

- Next vertex to be selected is **a**. And minimum cost will become `minCost=1+2=3`

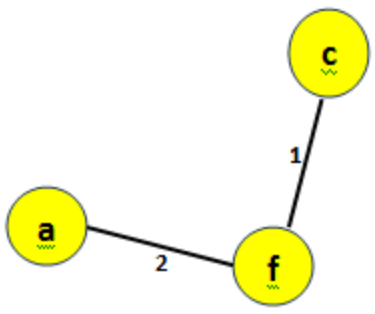


Figure 15: Adding vertex a

a	b	c	d	e	f	
1	0	1	0	0	1	visited
6	-1	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 16: Updated arrays after adding vertex a

- Next, either **b** or **d** can be selected. Let's consider **b**. Then the minimum cost will become $\text{minCost}=3+3=6$

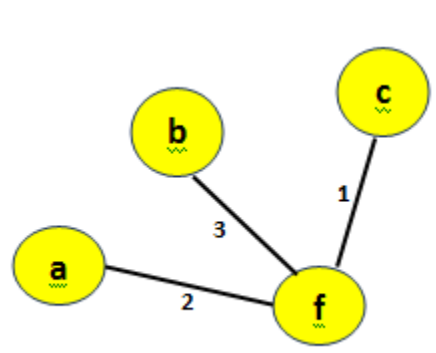


Figure 17: Adding vertex b to minimum spanning tree

a	b	c	d	e	f	
1	1	1	0	0	1	visited
6	6	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 18: Updated arrays after adding vertex b

- Next vertex to be selected is **d**, making the minimum cost $\text{minCost}=6+3=9$

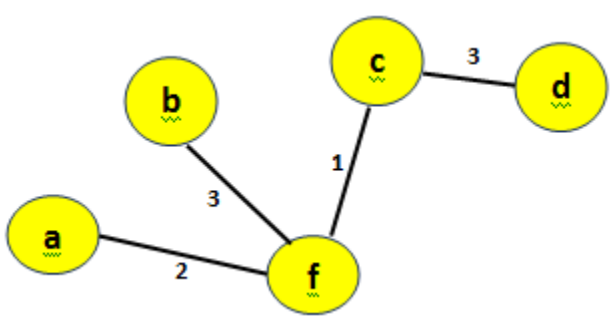


Figure 19: Adding vertex d

a	b	c	d	e	f	
1	1	1	1	0	1	visited
6	6	6	3	-1	-1	parent
2	3	1	3	2	0	key

Figure 20: Updated arrays after adding vertex d

- Then, **e** is selected and the minimum cost will become, $\text{minCost}=9+2=11$

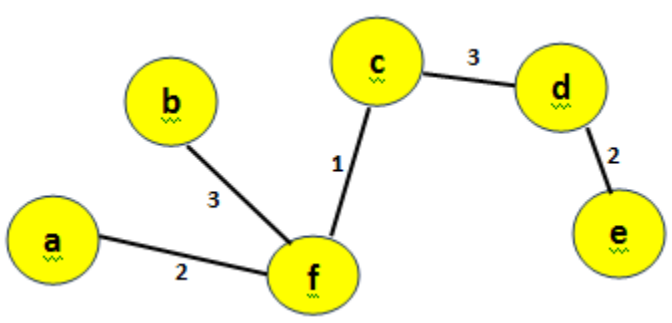


Figure 21: Adding vertex e. This is the final minimum spanning tree

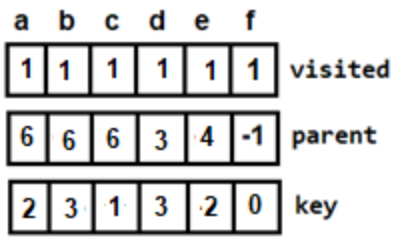


Figure 22: Updated arrays after adding vertex e (final arrays)

- Since all the vertices have been visited now, the algorithm terminates.
- Thus, Fig. 21 represents the Minimum Spanning Tree with total **cost=11**.

Implementation of Prim's Minimum Spanning Tree Algorithm

Now it's time to write a program in C++ for the finding out minimum spanning tree using prim's algorithm.

```
#include<iostream>


using namespace std;


// Number of vertices in the graph
const int V=6;


// Function to find the vertex with minimum key value
int min_Key(int key[], bool visited[])
{
    int min = 999, min_index;  // 999 represents an Infinite value


    for (int v = 0; v < V; v++) {
        if (visited[v] == false && key[v] < min) {
          // vertex should not be visited
            min = key[v];
                min_index = v;
        }
    }
    return min_index;
}
```

```
// Function to print the final MST stored in parent[]
int print_MST(int parent[], int cost[V][V])
{
    int minCost=0;
     cout<<"Edge \tWeight\n";
    for (int i = 1; i< V; i++) {
          cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";
          minCost+=cost[i][parent[i]];
    }
     cout<<"Total cost is"<<minCost;
}

// Function to find the MST using adjacency cost matrix representation
void find_MST(int cost[V][V])
{
    int parent[V], key[V];
    bool visited[V];

    // Initialize all the arrays
    for (int i = 0; i< V; i++) {
        key[i] = 999;    // 99 represents an Infinite value
        visited[i] = false;
        parent[i]=-1;
    }

    key[0] = 0;  // Include first vertex in MST by setting its key vaue
to 0.
    parent[0] = -1; // First node is always root of MST

    // The MST will have maximum V-1 vertices
```

```
    for (int x = 0; x < V - 1; x++)

    {

        // Finding the minimum key vertex from the

        //set of vertices not yet included in MST

        int u = min_Key(key, visited);


        visited[u] = true;  // Add the minimum key vertex to the MST


        // Update key and parent arrays

        for (int v = 0; v < V; v++)

        {

            // cost[u][v] is non zero only for adjacent vertices of u

            // visited[v] is false for vertices not yet included in MST

            // key[] gets updated only if cost[u][v] is smaller than
key[v]

            if (cost[u][v]!=0 && visited[v] == false && cost[u][v] <
key[v])

            {

                parent[v] = u;

                key[v] = cost[u][v];

            }

        }


    }


    // print the final MST

     print_MST(parent, cost);

}


// main function

int main()

{
```

```
    int cost[V][V];

     cout<<"Enter the vertices for a graph with 6 vetices";

    for (int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

                cin>>cost[i][j];

        }

    }

     find_MST(cost);



        return 0;

}
```

Copy

The input graph is the same as the graph in Fig 2.

```
Enter the vertices for a graph with 6 vetices
0 4 0 0 0 2
4 0 6 0 0 3
0 6 0 3 0 1
0 0 3 0 2 0
0 0 0 2 0 4
2 3 1 0 4 0
Edge    Weight
5 - 1   3
5 - 2   1
2 - 3   3
3 - 4   2
0 - 5   2
Total cost is11
```

Figure 23: Output of the program

Time Complexity Analysis for Prim's MST

Time complexity of the above C++ program is **O(V2)** since it uses adjacency matrix representation for the input graph. However, using an adjacency list representation, with the help of binary heap, can reduce the complexity of Prim's algorithm to **O(ElogV)**.

Real-world Applications of a Minimum Spanning Tree

Finding an MST is a fundamental problem and has the following real-life applications:

1. Designing the networks including computer networks, telecommunication networks, transportation networks, electricity grid and water supply networks.

2. Used in algorithms for approximately finding solutions to problems like Travelling Salesman problem, minimum cut problem, etc.
    - The objective of a **Travelling Salesman problem** is to find the shortest route in a graph that visits each vertex only once and returns back to the source vertex.
    - A **minimum cut problem** is used to find the minimum number of cuts between all the pairs of vertices in a planar graph. A graph can be classified as planar if it can be drawn in a plane with no edges crossing each other. For example,

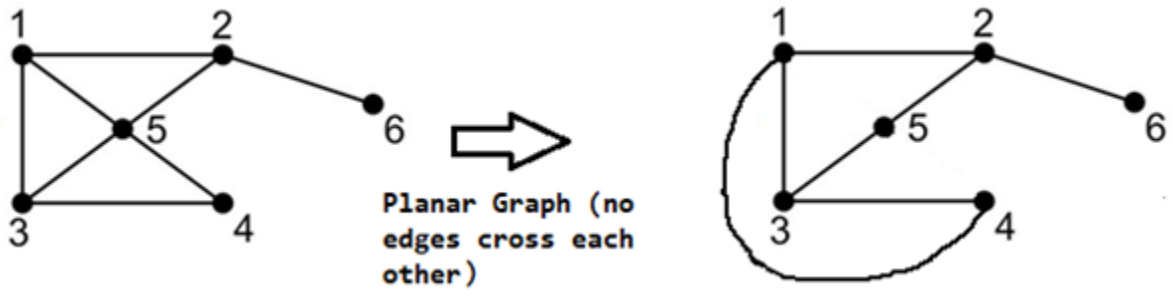


Figure 24: Planar Graph

3. Also, a cut is a subset of edges which, if removed from a planar graph, increases the number of components in the graph

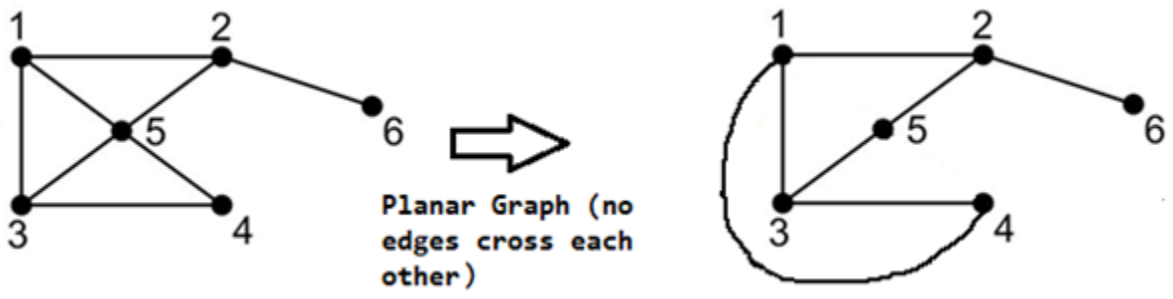


Figure 25: Cut-Set in a Planar Graph

4. Analysis of clusters.
5. Handwriting recognition of mathematical expressions.
6. Image registration and segmentation

Huffman Coding Algorithm

Every information in computer science is **encoded** as strings of **1s and 0s**. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

Fixed-Length encoding - Every character is assigned a binary code using same number of bits. Thus, a string like "aabacdad" can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like "aabacdad", frequency of characters 'a', 'b', 'c' and 'd' is 4,1,1 and 2 respectively. Since 'a' occurs more frequently than 'b', 'c' and 'd', it uses least number of bits, followed by 'd', 'b' and 'c'. Suppose we randomly assign binary codes to each character as follows-

a 0 b 011 c 111 d 11

Thus, the string "aabacdad" gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string 00011011111011, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11) aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11) aabbcb (0 | 0 | 011 | 011 | 111 | 011)

... and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the **"prefix rule"** which states that no binary code should be a prefix of another code. This will produce uniquely **decodable codes**. The above codes for 'a', 'b', 'c' and 'd' do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous **decodable codes.**

Lets reconsider assigning the binary codes to characters 'a', 'b', 'c' and 'd'.

a 0 b 11 c 101 d 100

Using the above codes, string **"aabacdad"** gets encoded to 0011010011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100). Now, we can decode it back to string **"aabacdad"**.

Problem Statement-

Input: Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output: Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-
Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child.
Also $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$

Step 5- Add z to min heap

Step 6- Last node in the heap is the root of Huffman tree

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15

i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

- **Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

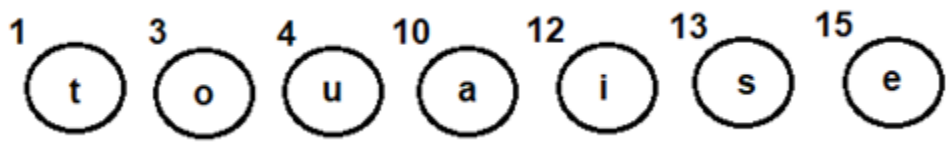


Fig 1: Leaf nodes for each character

Step B- Repeat the following steps till heap has more than one nodes

- **Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap
- **Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)
- **Step 5**- Add z to min heap

i. Extract and Combinenode u with an internal node having 4 as the frequency

ii. Add the new internal node to priority queue-

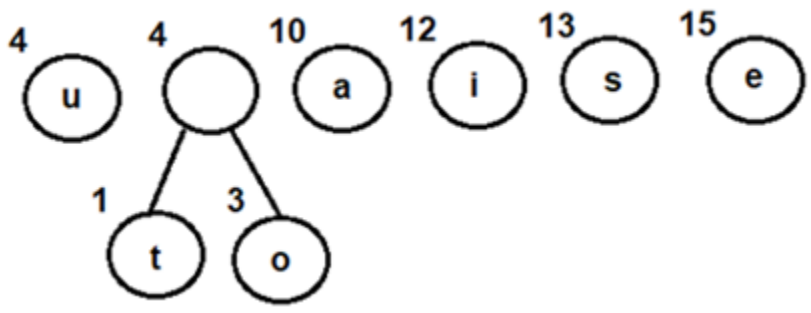


Fig 2: Combining nodes o and t

i. Extract and Combine node awith an internal node having 8 as the frequency

ii. Add the new internal node to priority queue-

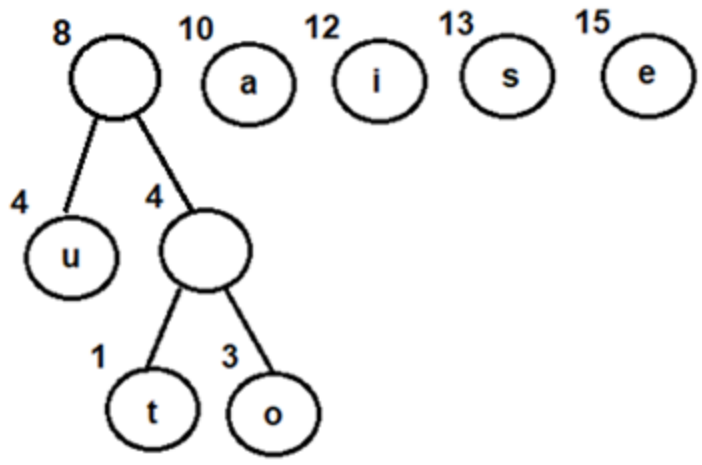


Fig 3: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s

ii. Add the new internal node to priority queue-

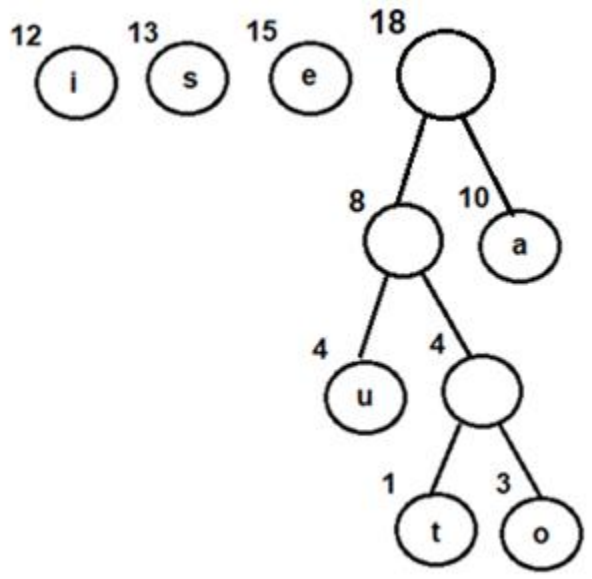


Fig 4: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s

ii. Add the new internal node to priority queue-

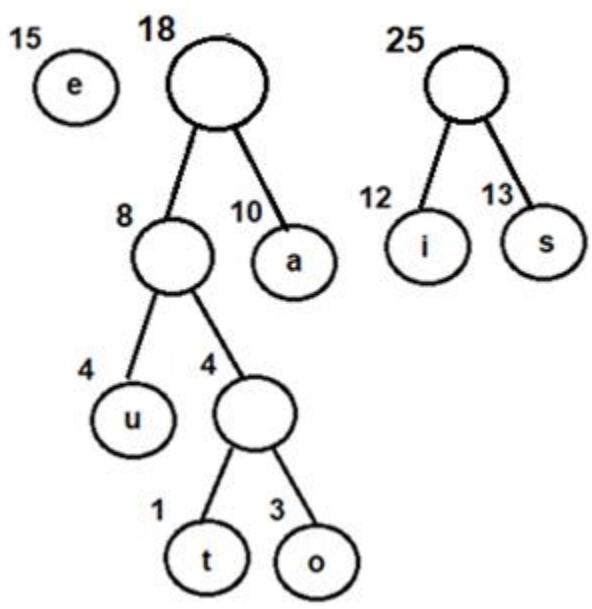


Fig 5: Combining nodes i and s

i. Extract and Combine node ewith an internal node having 18 as the frequency

ii. Add the new internal node to priority queue-

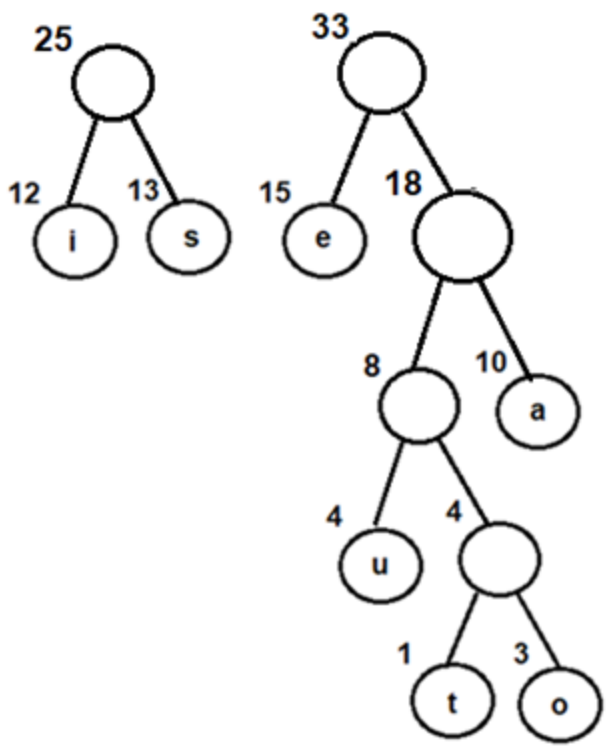


Fig 6: Combining node e with an internal node having 18 as frequency

i. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency

ii. Add the new internal node to priority queue-

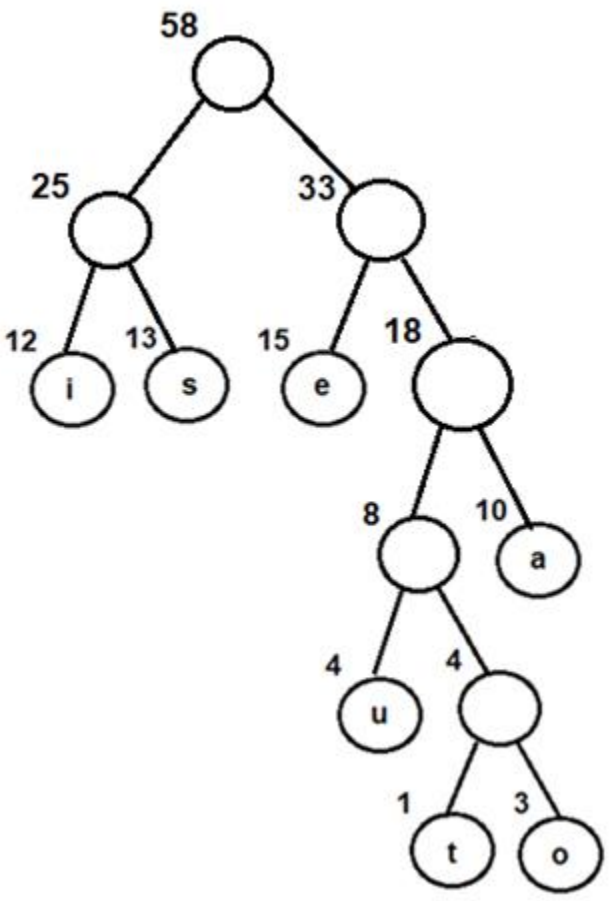


Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to arraywhile traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

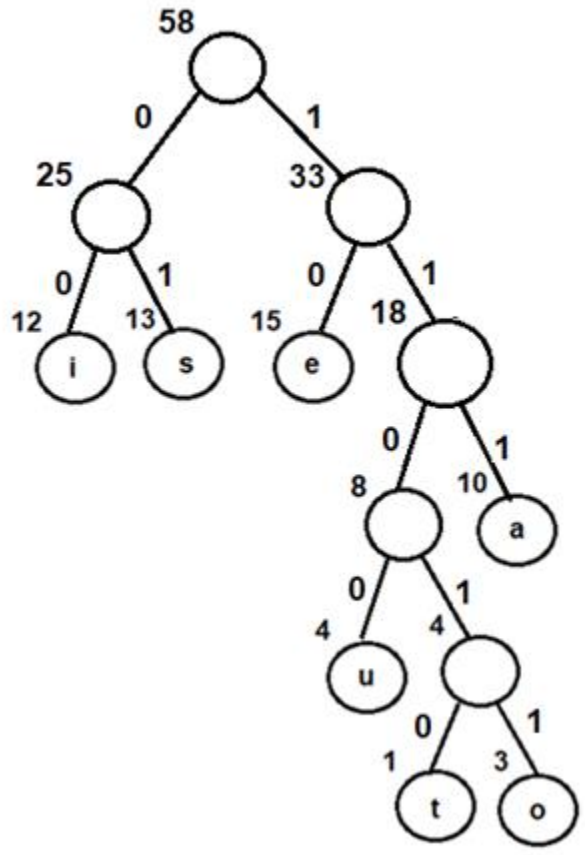


Fig 8: Assigning binary codes to Huffman tree

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

Using the above binary codes-

Suppose the string “staeiout” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to **“01110101111000110111110011010” (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010)** at the sender side.

Once received at the receiver’s side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the

string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Thus for the above bit stream

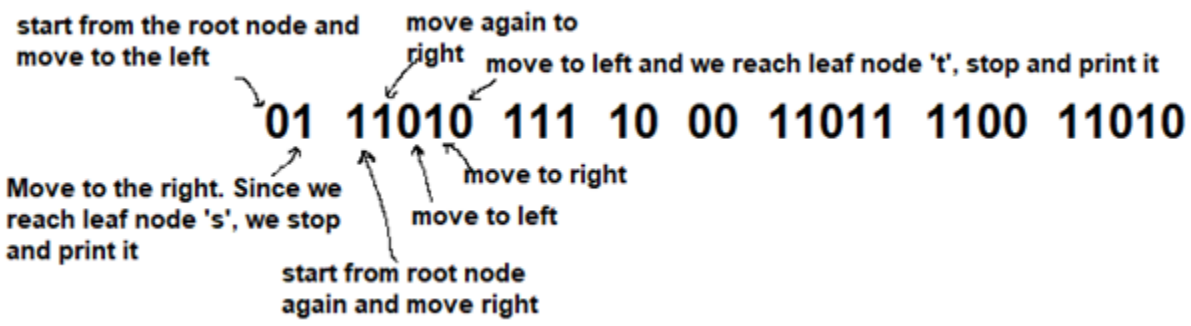


Fig 9: Decoding the bit stream

On similar lines-

- 111 gets decoded to ‘a’
- 10 gets decoded to ‘e’
- 00 gets decoded to ‘i’
- 11011 gets decoded to ‘o’
- 1100 gets decoded to ‘u’
- And finally, 11010 gets decoded to ‘t’, thus returning the string “staeiout” back

Implementation-

Following is the C++ implementation of Huffman coding. The algorithmcan be mapped to any programming language as per the requirement.

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>

using namespace std;

class Huffman_Codes
{
 struct New_Node
 {
  char data;
  size_t freq;
  New_Node* left;
```

```
  New_Node* right;

  New_Node(char data, size_t freq) : data(data),

                                     freq(freq),

left(NULL),

right(NULL)

                                     {}

 ~New_Node()

 {

   delete left;

   delete right;

 }

 };


 struct compare

 {

  bool operator()(New_Node* l, New_Node* r)

  {

    return (l->freq > r->freq);

  }

};


New_Node* top;


void print_Code(New_Node* root, string str)

{

if(root == NULL)

   return;


 if(root->data == '$')

 {
```

```
 print_Code(root->left, str + "0");

 print_Code(root->right, str + "1");

}



if(root->data != '$')

{

  cout << root->data <<" : " << str << "\n";

  print_Code(root->left, str + "0");

  print_Code(root->right, str + "1");

}

 }

}



public:

  Huffman_Codes() {};

  ~Huffman_Codes()

  {

    delete top;

  }

  void Generate_Huffman_tree(vector<char>& data, vector<size_t>& freq,
size_t size)

  {

   New_Node* left;

   New_Node* right;

   priority_queue<New_Node*, vector<New_Node*>, compare > minHeap;



for(size_t i = 0; i < size; ++i)

   {

      minHeap.push(new New_Node(data[i], freq[i]));

   }



while(minHeap.size() != 1)
```

```
    {
      left = minHeap.top();
minHeap.pop();

      right = minHeap.top();
minHeap.pop();

      top = new New_Node('$', left->freq + right->freq);
      top->left  = left;
      top->right = right;
      minHeap.push(top);
     }
    print_Code(minHeap.top(), "");
 }
};

 int main()
 {
  int n, f;
  char ch;
  Huffman_Codes set1;
  vector<char> data;
  vector<size_t> freq;
  cout<<"Enter the number of elements \n";
  cin>>n;
  cout<<"Enter the characters \n";
  for (int i=0;i<n;i++)
  {
      cin>>ch;
data.insert(data.end(), ch);
```

```
  }
  cout<<"Enter the frequencies \n";
  for (int i=0;i<n;i++)
  {
      cin>>f;
freq.insert(freq.end(), f);
  }

  size_t size = data.size();
  set1.Generate_Huffman_tree(data, freq, size);

  return 0;
}
```

Copy

The program is executed using same inputs as that of the example explained above. This will help in verifying the resultant solution set with actual output.

```
Enter the number of elements
7
Enter the characters
a e i o u s t
Enter the frequencies
10 15 12 3 4 13 1
i : 00
s : 01
e : 10
u : 1100
t : 11010
o : 11011
a : 111
```

Fig 10: Output

Time Complexity Analysis-

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

- Building a min heap takes $O(nlogn)$ time (Moving an element from root to leaf node requires $O(logn)$ comparisons and this is done for n/2 elements, in the worst case).
- Building a min heap takes $O(nlogn)$ time (Moving an element from root to leaf node requires $O(logn)$ comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

Advantages of Huffman Encoding-

- This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length
- It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string
- The binary codes generated are prefix-free

Disadvantages of Huffman Encoding-

- Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.
- Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
- Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

Real-life applications of Huffman Encoding-

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.

Dijkstra's Algorithm

Dijkstra's algorithm, published in 1959, is named after its discoverer Edsger Dijkstra, who was a Dutch computer scientist. This algorithm aims to find the shortest-path in a directed or undirected graph with non-negative edge weights.

Before, we look into the details of this algorithm, let’s have a quick overview about the following:

- **Graph**: A graph is a non-linear data structure defined as G=(V,E) where V is a finite set of vertices and E is a finite set of edges, such that each edge is a line or arc connecting any two vertices.
- **Weighted graph**: It is a special type of graph in which every edge is assigned a numerical value, called weight
- **Connected graph**: A path exists between each pair of vertices in this type of graph
- **Spanning tree** for a graph G is a subgraph G’ including all the vertices of G connected with minimum number of edges. Thus, for a graph G with n vertices, spanning tree G’ will have n vertices and maximum n-1 edges.

Problem Statement

Given a weighted graph G, the objective is to find the shortest path from a given source vertex to all other vertices of G. The graph has the following characteristics-

- Set of vertices `V`
- Set of weighted edges `E` such that `(q,r)` denotes an **edge** between **vertices** q and r and cost(q,r) denotes its weight

Dijkstra's Algorithm:

- This is a single-source shortest path algorithm and aims to find solution to the given problem statement
- This algorithm works for both directed and undirected graphs
- It works only for connected graphs
- The graph should not contain negative edge weights
- The algorithm predominantly follows Greedy approach for finding locally optimal solution. But, it also uses Dynamic Programming approach for building globally optimal solution, since the previous solutions are stored and further added to get final distances from the source vertex
- The main logic of this algorithm is basedon the following formula- `dist[r]=min(dist[r], dist[q]+cost[q][r])`

This formula states that distance vertex r, which is adjacent to vertex q, will be updated if and only if the value of `dist[q]+cost[q][r] is less than dist[r]`. Here-

- dist is a 1-D array which, at every step, keeps track of the shortest distance from source vertex to all other vertices, and
- cost is a 2-D array, representing the cost adjacency matrix for the graph

- This formula uses both Greedy and Dynamic approaches. The Greedy approach is used for finding the minimum distance value, whereas the Dynamic approach is used for combining the previous solutions (**dist[q]** is already calculated and is used to calculate **dist[r]**)

Algorithm-

Input Data-

- Cost Adjacency Matrix for Graph G, say cost
- Source vertex, say s

Output Data-

- Spanning tree having shortest path from s to all other vertices in G

Following are the steps used for finding the solution-

Step 1; Set dist[s]=0, S=ϕ // s is the source vertex and S is a 1-D array having all the visited vertices

Step 2: For all nodes v except s, set dist[v]= ∞

Step 3: find q not in S such that dist[q] is minimum // vertex q should not be visited

Step 4: add q to S // add vertex q to S since it has now been visited

Step 5: update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited $dist[r]=min(dist[r], dist[q]+cost[q][r])$ //Greedy and Dynamic approach

Step 6: Repeat Steps 3 to 5 until all the nodes are in S // repeat till all the vertices have been visited

Step 7: Print array dist having shortest path from the source vertex u to all other vertices

Step 8: Exit

Let's try and understand the working of this algorithm using the following example-

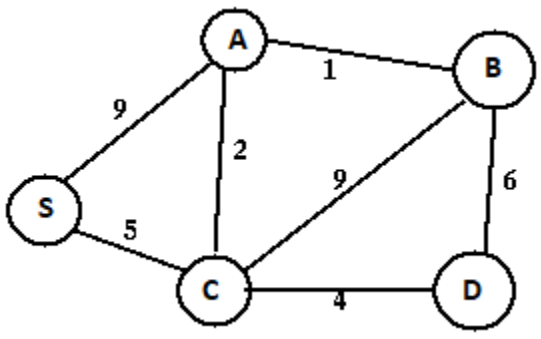


Fig 1: Input Graph (Weighted and Connected)

Given the above weighted and connected graph and source vertex s, following steps are used for finding the tree representing shortest path between s and all other vertices-

Step A- Initialize the distance array (dist) using the following steps of algorithm –

- **Step 1**- Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the visited vertices
- **Step 2**- For all nodes v except s, set dist[v]= ∞

Set of visited vertices (S)	S	A	B	C	D

	0	∞	∞	∞	∞

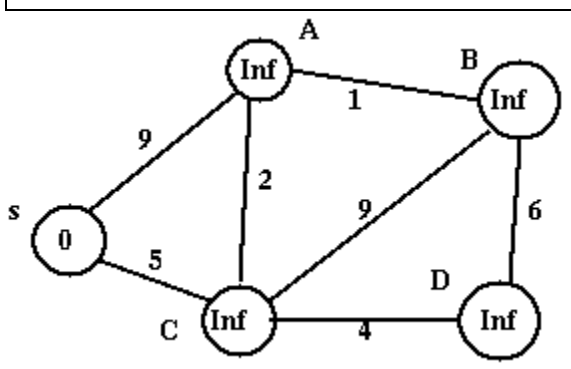


Fig 2: Graph after initializing dist[]

Step B- a)Choose the source vertex s as dist[s] is minimum and s is not in S.

Step 3- find q not in S such that dist[q] is minimum // vertex should not be visited

Visit s by adding it to S

Step 4- add q to S // add vertex q to S since it has now been visited

Step c) For all adjacent vertices of s which have not been visited yet (are not in S) i.e A and C, update the distance array using the following steps of algorithm -

Step 5- update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited $dist[r]=min(dist[r], dist[q]+cost[q][r])$ //Greedy and Dynamic approach

$dist[A]= min(dist[A], dist[s]+cost(s, A)) = min(\infty, 0+9) = 9$ $dist[C] = min(dist[C], dist[s]+cost(s, C)) = min(\infty, 0+5) = 5$

Thus dist[] gets updated as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. Updating the distance array for adjacent vertices of C i.e. A, B and D

Step 6- Repeat Steps 3 to 5 until all the nodes are in S

$dist[A]=min(dist[A], dist[C]+cost(C,A)) = min(9, 5+2)= 7$

$dist[B]= min(dist[B], dist[C]+cost(C,B)) = min(\infty, 5+9)= 14$

$dist[D]= min(dist[D], dist[C]+cost(C,D))= min((\infty,5+4)=9$

This updates dist[] as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

[s,C]	0	7	14	5	9

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

The last updation of dist[] gives the shortest path values from s to all other vertices

The resultant shortest path spanning tree for the given graph is as follows-

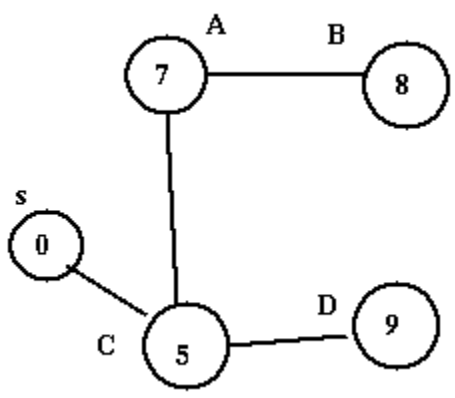


Fig 3: Shortest path spanning tree

Note-

- There can be multiple shortest path spanning trees for the same graph depending on the source vertex

Implementation-
Following is the C++ implementation for Dijkstra's Algorithm

Note : The algorithm can be mapped to any programming language as per the requirement.

```
#include<iostream>

using namespace std;

#define V 5  //Defines total number of vertices in the graph

#define INFINITY 999
```

```
int min_Dist(int dist[], bool visited[])

//This method used to find the vertex with minimum distance and is not
yet visited

{

     int min=INFINITY,index;                    //Initialize min with
infinity

     for(int v=1;v<=V;v++)

     {

          if(visited[v]==false &&dist[v]<=min)

          {

               min=dist[v];

               index=v;

          }

     }

     return index;

}

void Dijkstra(int cost[V][V],int src) //Method to implement shortest
path algorithm

{

     int dist[V];

     bool visited[V];

     for(int i=1;i<=V;i++)                      //Initialize dist[] and
visited[]

     {

          dist[i]=INFINITY;

          visited[i]=false;

     }

     //Initialize distance of the source vertec to zero

     dist[src]=0;

     for(int c=2;c<=V;c++)

     {

          //u is the vertex that is not yet included in visited and is
having minimum
```

```
          int u=min_Dist(dist,visited);                distance

          visited[u]=true;                              //vertex u is now
visited

          for(int v=1;v<=V;v++)

//Update dist[v] for vertex v which is not yet included in visited[]
and

//there is a path from src to v through u that has smaller distance
than

// current value of dist[v]

          {

               if(!visited[v] && cost[u][v]
&&dist[u]+cost[u][v]<dist[v])

                    dist[v]=dist[u]+cost[u][v];

               }

     }

      //will print the vertex with their distance from the source

     cout<<"The shortest path  "<<src<<" to all the other vertices is:
\n";

     for(int i=1;i<=V;i++)

     {

        if(i!=src)

        cout<<"source:"<<src<<"\t destination:"<<i<<"\t MinCost
is:"<<dist[i]<<"\n";

     }

}

int main()

{

     int cost[V][V], i,j, s;

     cout<<"\n Enter the cost matrix weights";

     for(i=1;i<=V;i++)       //Indexing ranges from 1 to n

          for(j=1;j<=V;j++)

          {

cin>>cost[i][j];
```

```
                //Absence of edge between vertices i and j is
represented by INFINITY

              if(cost[i][j]==0)

                cost[i][j]=INFINITY;

            }

cout<<"\n Enter the Source Vertex";

cin>>s;



     Dijkstra(cost,s);

     return 0;

}
```

Copy

The program is executed using same input graph as in Fig.1.This will help in verifying the resultant solution set with actual output.

```
 Enter the cost matrix weights
0 9 999 5 999
9 0 1 2 999
999 1 0 9 6
5 2 9 0 4
999 999 6 4 0

 Enter the Source Vertex1
The shortest path from source 1 to all the other vertices is:
source: 1        destination: 2  MinCost is: 7
source: 1        destination: 3  MinCost is: 8
source: 1        destination: 4  MinCost is: 5
source: 1        destination: 5  MinCost is: 9
```

Fig 4: Output

Time Complexity Analysis-

Following are the cases for calculating the time complexity of Dijkstra's Algorithm-

- **Case1**- When graph G is represented using an adjacency matrix -This scenario is implemented in the above **C++** based program. Since the implementation contains two nested for loops, each of complexity **O(n)**, the complexity of Dijkstra's algorithm is **O(n2)**. Please note that n here refers to total number of vertices in the given graph
- **Case 2**- When graph G is represented using an adjacency list - The time complexity, in this scenario reduces to **O(|E| + |V| log |V|)** where |E|represents number of edges and |V| represents number of vertices in the graph

Disadvantages of Dijkstra's Algorithm-

Dijkstra's Algorithm cannot obtain correct shortest path(s)with weighted graphs having negative edges. Let's consider the following example to explain this scenario-

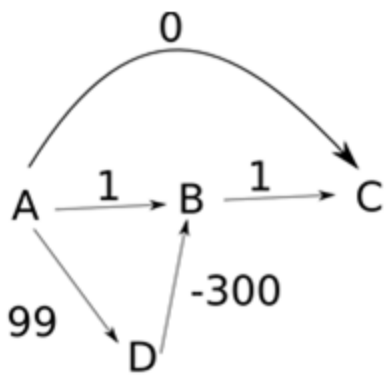


Fig 5: Weighted graph with negative edges

Choosing source vertex as A, the algorithm works as follows-

Step A- Initialize the distance array (dist)-

Set of visited vertices (S)	A	B	C	D
	0	∞	∞	∞

Step B- Choose vertex A as dist[A] is minimum and A is not in S. Visit A and add it to S. For all adjacent vertices of A which have not been visited yet (are not in S) i.e C, B and D, update the distance array

$dist[C] = \min(dist[C], dist[A]+cost(A, C)) = \min(\infty, 0+0) = 0$

$dist[B] = \min(dist[B], dist[A]+cost(A, B)) = \min(\infty, 0+1) = 1$

$dist[D] = \min(dist[D], dist[A]+cost(A, D)) = \min(\infty, 0+99) = 99$

Thus dist[] gets updated as follows-

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. The distance array does not get updated since there are no adjacent vertices of C

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99
[A, C]	0	1	0	99

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	A	B	C	D

[A]	0	1	0	99
[A, C]	0	1	0	99
[A, C, B]	0	1	0	99
[A, C, B, D]	0	1	0	99

Thus, following are the shortest distances from A to B, C and D-

A->C = 0 A->B = 1 A->D = 99

But these values are not correct, since we can have another path from **A** to **C, A->D->B->C** having **total cost= -200** which is smaller than 0. This happens because once a vertex is visited and is added to the set S, it is never "looked back" again. Thus, Dijkstra's algorithm does not try to find a shorter path to the vertices which have already been added to S.

- It performs a blind search for finding the shortest path, thus, consuming a lot of time and wasting other resources

Applications of Dijkstra's Algorithm-

- Traffic information systems use Dijkstra's Algorithm for tracking destinations from a given source location
- **Open Source Path First (OSPF)**, an Internet-based routing protocol, uses Dijkstra's Algorithm for finding best route from source router to other routers in the network
- It is used by **Telephone and Cellular networks** for routing management
- It is also used by **Geographic Information System (GIS)**, such as Google Maps, for finding shortest path from point A to point B
