Course:

Data structure & Algorithm

Synthesis Report

Part 1

Learned Knowledges

Guide teachers

Lê Ngọc Thành

Tạ Việt Phương

Phạm Trọng Nghĩa

Academic year

2020-2021

Thank you

Introduction members:

Full name	Student ID
Nguyễn Quốc Huy	20127518
Lại Minh Thông	20127635
Nguyễn Lê Hoàng Thông	20127078
Trần Anh Kiệt	20127545
Đào Đại Hải	20127016

Contents

Topic 1: Abstract Data Types

Contents:

I. Introduction

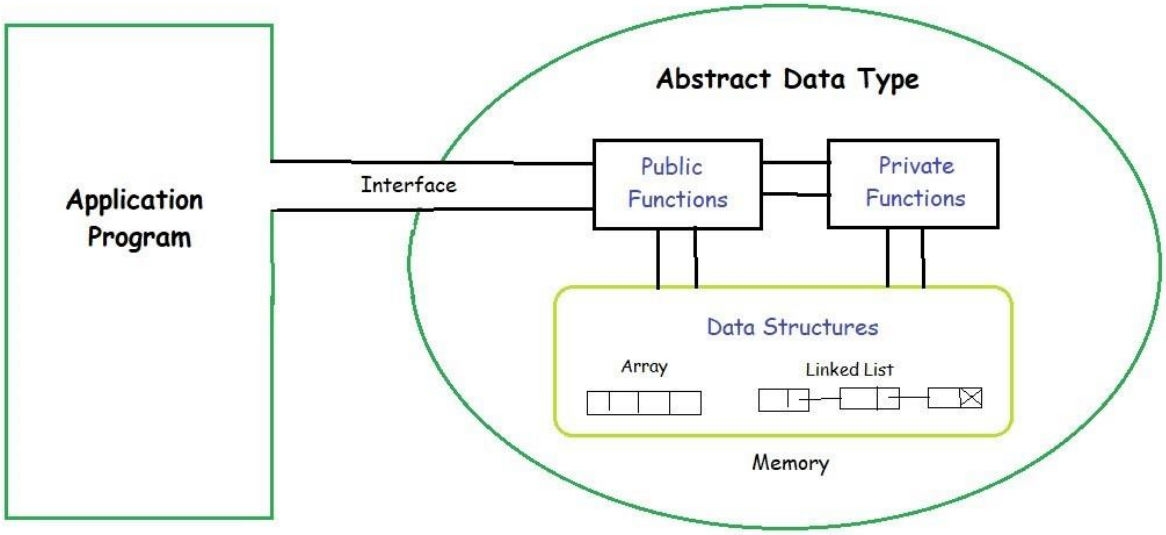
Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Abstract data types include:

- Values – V
- Operations - O



The user of data type does not need to know how that data type is implemented. Think of ADT as a black box which hides the inner structure and design of the data type.

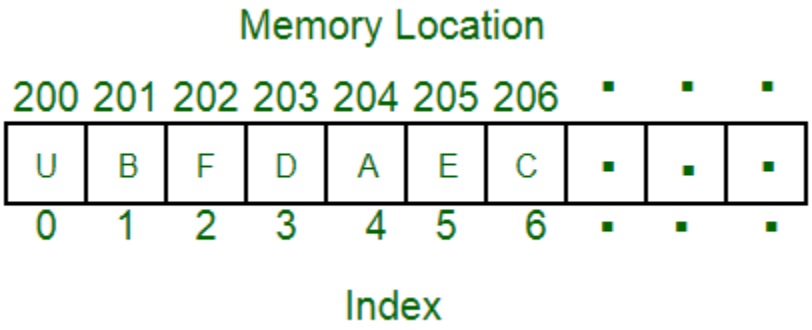
For example, we will describe an ADT that is a singly linked list used to manage some student data.

```
struct Student {
     string name;
     string studentID;
     float gpa;
};

struct Node {
     Student info;
     Node* pNext;
};
```

II. Some Abstract Data Types

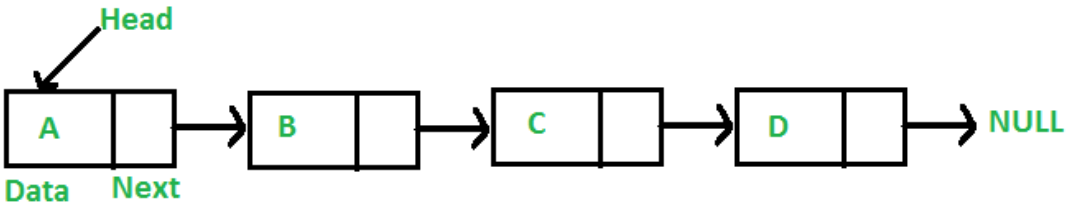
1. Array

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together This makes it easier to calculate the position of each element by simply adding an offset to a base value. The memory location of the first element of the array.



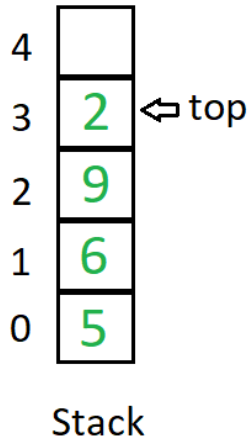
2. Linked Lists

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location. The elements are linked using pointers.



3. Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

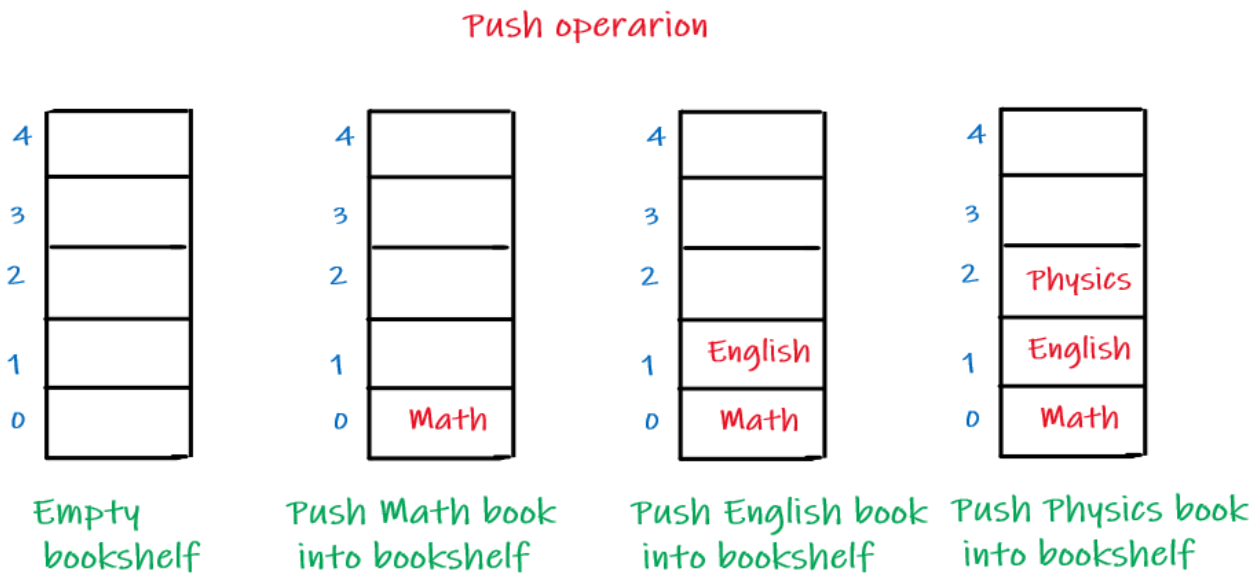


Mainly the following three basic operations are performed in the stack:

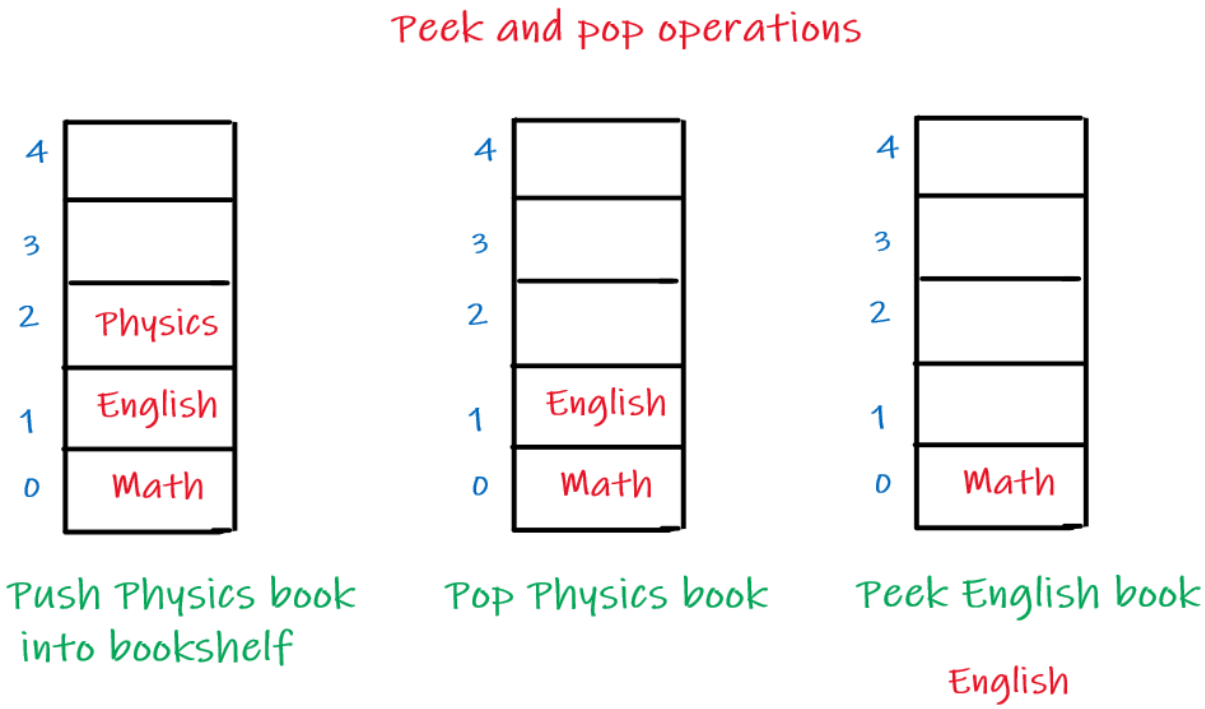
- **Push**: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop**: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek** or **Top**: Returns top element of the stack.
- **isEmpty**: Returns true if the stack is empty, else false.

Example to illustrate for stack: We're going arrange the stack of books

Push operation: Now we put books one by one on the bookshelf.



Peek and Pop operations: Assumption that we want to take English book to read

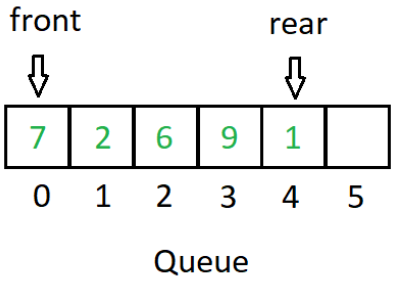


4. Queue

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Mainly the following four basic operations are performed on queue:

- **Enqueue**: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue**: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front**: Get the front item from the queue.
- **Rear**: Get the last item from the queue.

The image above is a real example of a queue, first come first served.

Why Abstract Data Type became a necessity?

- Earlier if a programmer wanted to read a file, the whole code was written to read the physical file device. So that is how Abstract Data Type (ADT) came into existence.
- The code to read a file was written and placed in a library and made available for everyone's use. This concept of ADT is being used in the modern languages nowadays.

Topic 2: Analysis of Algorithms

Contents:

I. Introduction

- **What is an algorithm?**

  An algorithm is the list of instructions and rules that a computer needs to do to complete a task.

  Ex: Find the maximum value in array A

  S1: Max ← A[0], i ← 0

  S2: i ← i + 1

  S3: If i > n, move to step 5

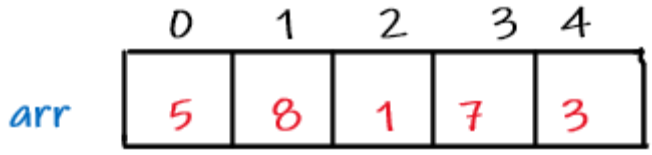
  S4: If A[i] > Max, Max ← A[i], go back to step 2

  S5: end.

- **Why performance analysis?**
  - Algorithm analysis helps determine which algorithms are effective in terms of space and time, …
  - When we work with large programs, we will have a good sense of where major slowdowns are likely to cause bottlenecks, and where more attention should be paid to get the largest improvements.

II. Three cases to analyze an algorithm

Let us consider the following implementation of Linear Search




```
int linearSearch(int arr[], int n, int x){
     for (int i = 0; i < n; i++) {
          if (arr[i] == x)
               return i;
     }
     return -1;
}
```

1. **The Worst Case (O)**

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

For *Linear Search*, the worst case happens when the element to be searched is not present in the array (x in the above code). When x is not present, the **search**() functions compare it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(n).

2. **The Average Case (Θ)**

In average case analysis, we take all possible inputs and calculate computing time for all the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know distribution of cases.

For *Linear Search* problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1).

$$\frac{\sum_{i=1}^{n+1} \theta(i)}{n+1} = \frac{\theta\left((n+1) * \frac{n+2}{2}\right)}{n+1} = \theta(n)$$

3. The Best Case (Ω)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

In *Linear Search* problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1).

Lower bound (Ω) ≤ Average time (Θ) ≤ Upper Bound (O)

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

For insertion sort

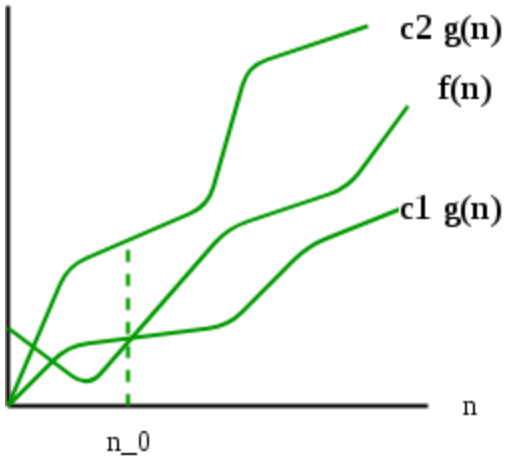
0	*1*	*2*	*3*	*4*	*5*	*6*
30	50	57	78			

The worst case occurs when the array is reverse sorted. For example, when we add the number **10** into the array, all elements of the array move backward in one position.

The best case occurs when the array is sorted in the same order as output. For example, when we add the number **90** into the array without any change.

III. Asymptotic Notations

1. Θ Notation



f(n) = theta(g(n))

The **Theta** notation bounds a function from above and below, so it defines exact asymptotic behavior.

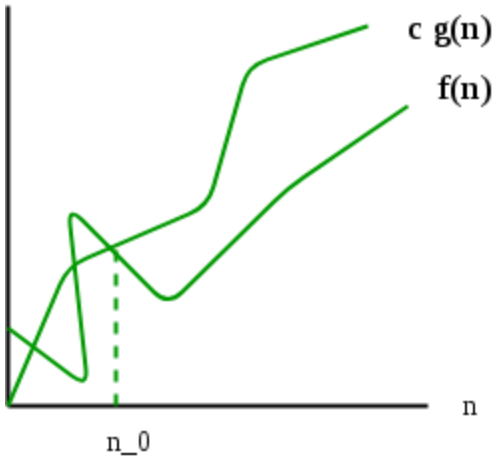
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.

For example, consider the following expression.

$$3n^2 + n^2 + 4500 = \Theta(n^3)$$

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that $0 \leq c1*g(n) \leq f(n) \leq c2*g(n)$ for all $n \geq n0$}

2. Notation:



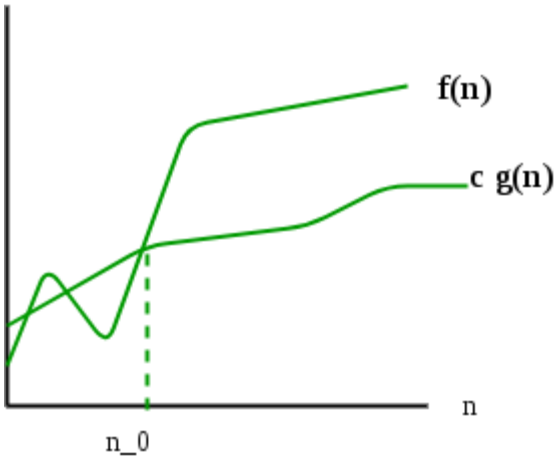
f(n) = O(g(n))

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

O(g(n)) = {f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq c*g(n)$ for all $n \geq n0$}

For example, consider the case of Insertion Sort. It takes quadratic time in worst case. We can say that the time complexity of Insertion sort is $O(n^2)$.

3. Ω Notation



f(n) = Omega(g(n))

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n0 such that $0 \leq c*g(n) \leq f(n)$ for all $n \geq n0\}$

IV. Properties of Asymptotic Notations:

Some important properties of those notations

1. **General properties:**

If f(n) is O(g(n)) then k*f(n) is also O(g(n)); where k is a

Example: $f(n) = 3n^2 + 5$ is $O(n^2)$

Then we have $7*f(n) = 7(3n^2 + 5) = 21n^2 + 35$ is also $O(n^2)$

Similarly, this property satisfies for both Θ and Ω notation.

2. **Transitive properties:**

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n))

Example: if f(n) = n, g(n) = n2 and h(n) = n3

n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

Similarly, this property satisfies for both Θ and Ω notation.

3. **Reflexive properties:**

If f(n) is given, then f(n) is O(f(n)). Since max value of f(n) will be f(n) itself.

Example: $f(n) = n^2$; $O(n^2)$ that is O(f(n))

Similarly, this property satisfies for both Θ and Ω notation.

4. Symmetric properties:

If f(n) is Θ(g(n)) then g(n) is Θ(f(n))

Example: f(n) = n^2 and g(n) = n^2

then f(n) = Θ(n^2) and g(n) = Θ(n^2)

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties

If f(n) is O(g(n)) then g(n) is Ω(f(n))

Example: f(n) = n, g(n) = n^2

then n is O(n^2) and n^2 is Ω(n)

This property only satisfies for O and Ω notations.

Topic 3: Search Algotithms

Lesson 1: Basic search algorithm

Contents:

I. Introduction.

- Searching algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.
- Based on the type of search operation, these algorithms are generally classified into two categories:
  - ***Sequential search***.
  - ***Interval search***.
- Depend on the case of the data to use the right one of the two algorithms above.
- To be easier for illustration, the object that contains a individual that we are going to search in this topic is the a consecutive sequence of integers (array of integers)
- There are also different methods to search for a element like *Hashing* or using *Search Trees* which also will be researched deeper in different topics.

II. Search Algorithms.

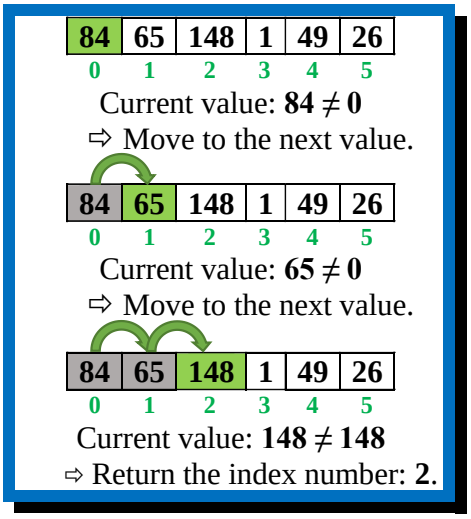
1. Liner search. (Sequential search)

- This is the simplest method to searching for an element.
- The array is tranversed through every single element while making a comparison between the current element in the array with the element we are looking for.
- Time Complexity: O(n).

- Example:

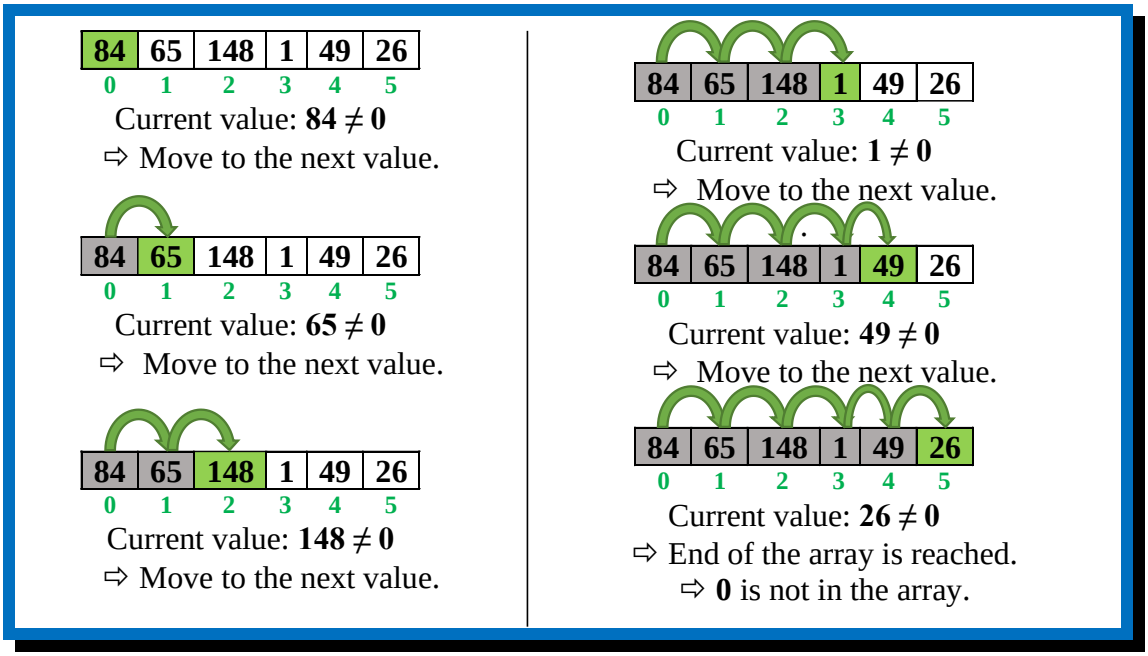
Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

- Search for the key = 148:



Example 1 - 1

- Search for key = 0:



Example 1 - 2

- Advantage: Easy to implement.
- Disadvantage: Take very long time.

2. Binary search. (*Interval search)*

- This algorithm is only able to be used on sorted structures.
- The main action of this algorithm is the division half of the current's total of elements by eliminating all the elements before or after the middle element depend on what value of the element is looking for and the middle element is considering in the current elements.
- Time Complexity: O(log n).
- Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

- Search for the key = 148:

84	65	148	1	49	26
0	1	2	3	4	5

Step 1: Sort the array.

1	26	49	65	84	148
0	1	2	3	4	5

Step 2: Use Binary Search.

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**5**+**0**)/2 = **2**

Middle value: **49** < **148**

⇨ Eliminate all elements before index **2**.

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**5**+**3**)/2 = **4**

Middle value: **84** < **148**

⇨ Eliminate all elements before index **4**.

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**5**+**5**)/2 = **10**

Middle value: **148** = **148**

⇨ Return the index number: **4**

Example 2 - 1

- Search for the key = 55:

1	26	49	65	84	148
0	1	2	3	4	5

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**5**+**0**)/2 = **2**

Middle value: **49** < **55**

⇨ Eliminate all elements before index **4**.

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**5**+**3**)/2 = **4**

Middle value: **84** > **55**

⇨ Eliminate all elements after index **4**.

1	26	49	65	84	148
0	1	2	3	4	5

Middle index: (**3**+**3**)/2 = **3**

Middle value: **65** > **55**

⇨ End of the array is reached

⇨ **55** is not in the array

Example 2 - 2

Lesson 2: Hashing

Contents:

I. Introduction

Suppose that we want to design a system for storing student records keyed using Student ID and we want following queries to be performed efficiently:

1. Insert a Student ID and corresponding information.
2. Search a Student ID and fetch the information.
3. Delete a Student ID and related information.

For example, to insert a Student ID, we create a record with details of given Student ID, use Student ID as index and store the pointer to the created record in table. But this solution has many practical limitations. The problem with this solution is extra space required is huge. For example, in my university, the student ID has 8 digits, we need $O(m * 10^8)$ space for table where m is size of pointer to record. So, hashing appeared to solve this problem

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to data structures like Array, Linked List, … in practice. With hashing we get O(1) search time on average and O(n) in worst case.

Hash function:

The idea is to use hash function that converts a given big number or string to a smaller integer and uses the small number as index in a table called hash table.

A good hash function should have following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys.

Hash Table:

An array that stores the pointers to records corresponding to a given element.

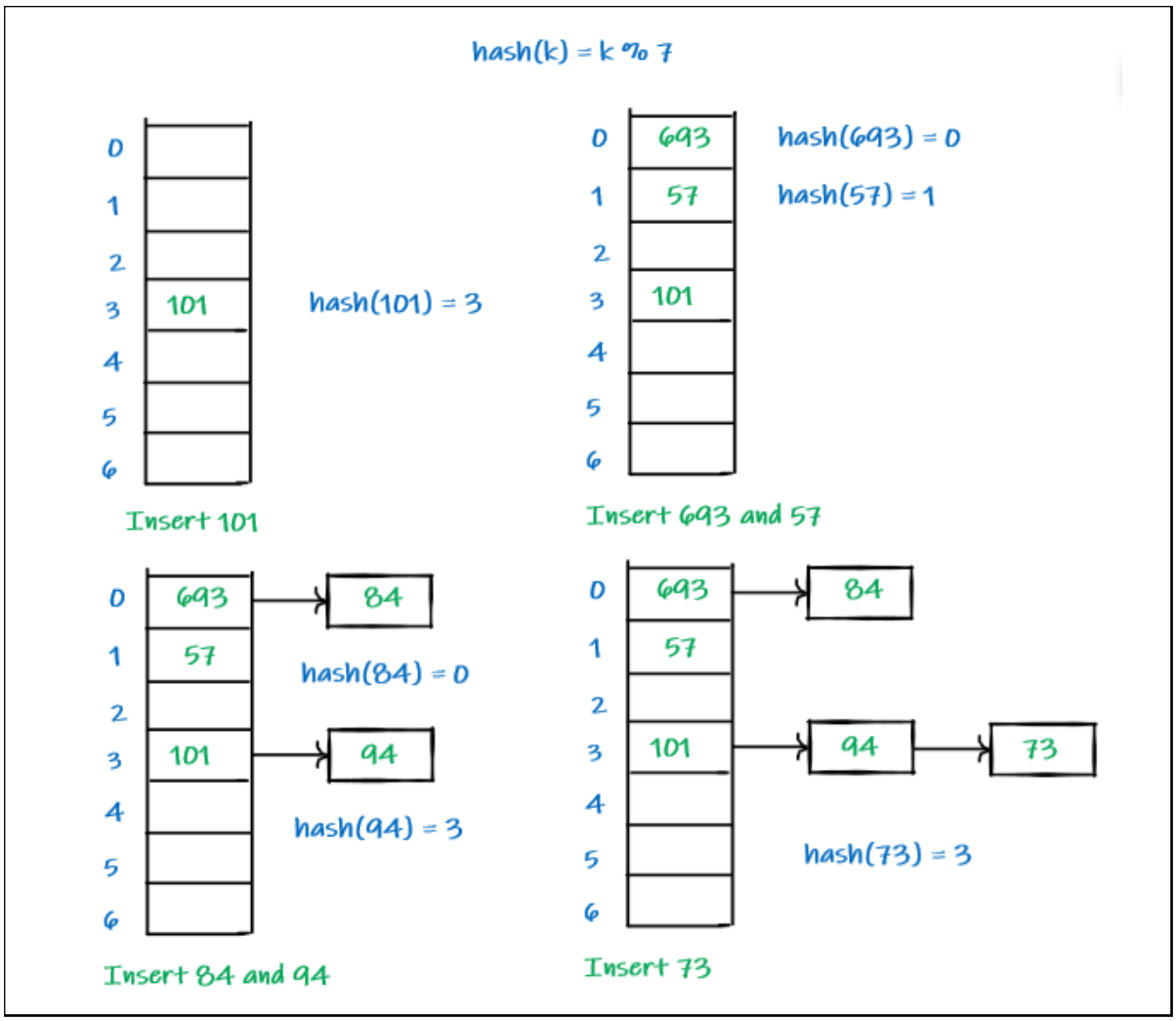
Collision Handling:

Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called *collision* and must be handled using some collision handling technique. We are going to discuss them in part II, III.

II. Separate Chaining

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 101, 693, 57, 84, 94, 73.



Advantages:

1. Simple to implement.

2. Hash table never fills up, we can always add more elements to the chain.

3. It is mostly used when it is unknown how many keys may be inserted or deleted.

Disadvantages:

1. Cache performance of chaining is not good as keys are stored using a linked list.

2. Wastage of Space (Some Parts of hash table are never used).

3. If the chain becomes long, then search time can become O(n) in the worst case.

4. Uses extra space for links.

III. Open Addressing

Open addressing is also a method for handling collisions. In this method, all elements are stored in the hash table itself. So, at any point, the size of the table must be greater than or equal to the total number of keys.

The idea of Open Addressing:

- Insert(k): Keep probing until an empty slot is found. After an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): If we simply delete a key, then the search may fail. So, slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

1. **Linear probing**

If the current position has been taken, we will find another position. In linear probing, we linearly probe for next slot.

h (k, i) = (hash(k) + i) mod S

Where:

i is the order of the attempt (i = 0, 1, 2, …)

hash(k): hash function

S: size of the array

Let **hash(x)** be the slot index computed using a *hash function* and **S** be the table size:

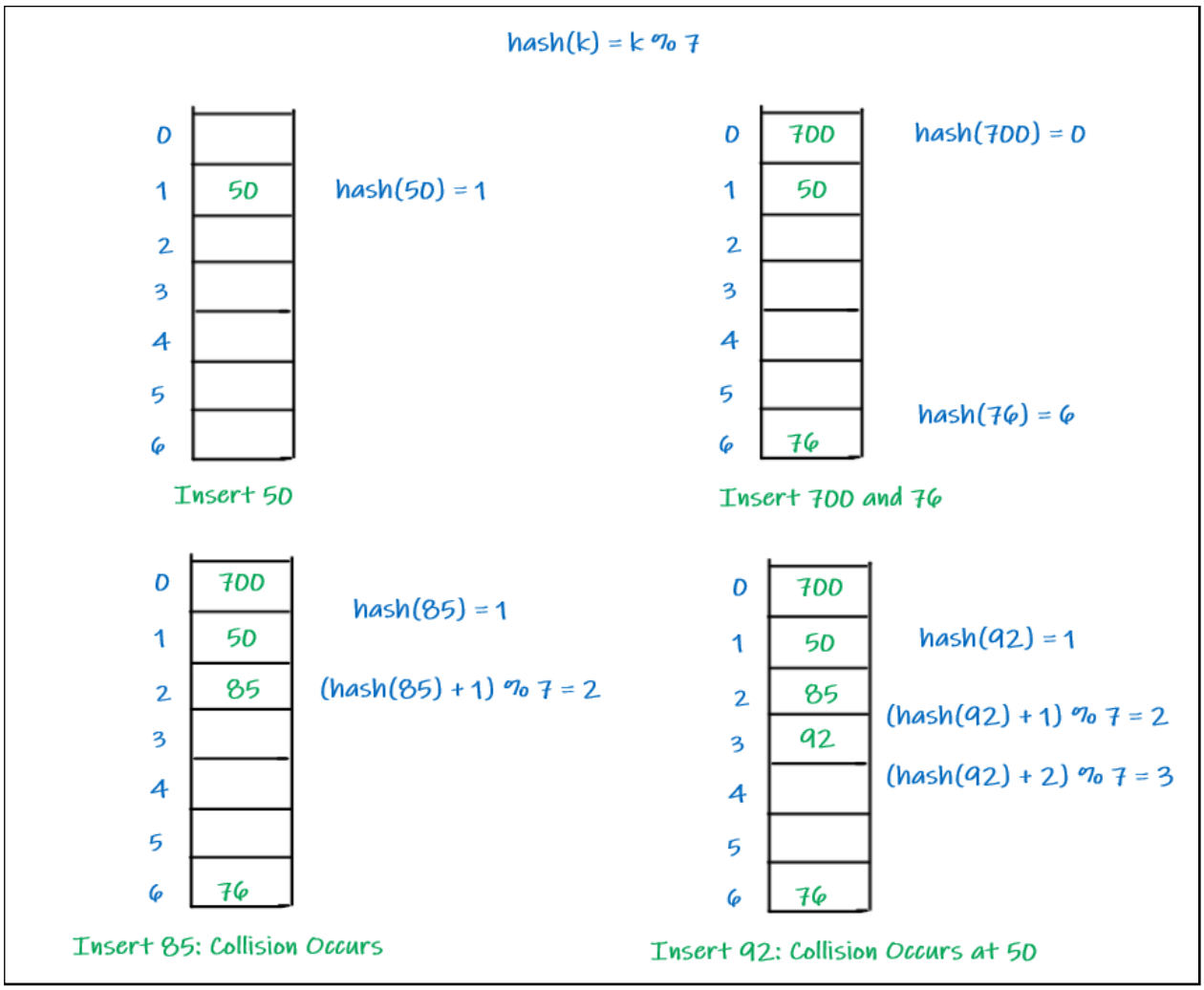
If slot [hash(x) % S] is full, then we try [(hash(x) + 1) % S]

If [(hash(x) + 1) % S] is also full, then we try [(hash(x) + 2) % S]

If [(hash(x) + 2) % S] is also full, then we try [(hash(x) + 3) % S]

……

Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92.



Some problems with **Linear Probing**:

1. Many consecutive elements form groups, and it starts taking time to find a free slot or to search for an element.

2. Two records only have the same collision chain if their initial position is the same.

2. Quadratic Probing

Like linear probing, but we use quadratic function.

$$h(k, i) = (hash(k) + i^2) \bmod S$$

Where:

i: the order of the attempt (i = 0, 1, 2, …)

hash(k): hash function

S: size of the array

Let **hash(x)** be the slot index computed using hash function.

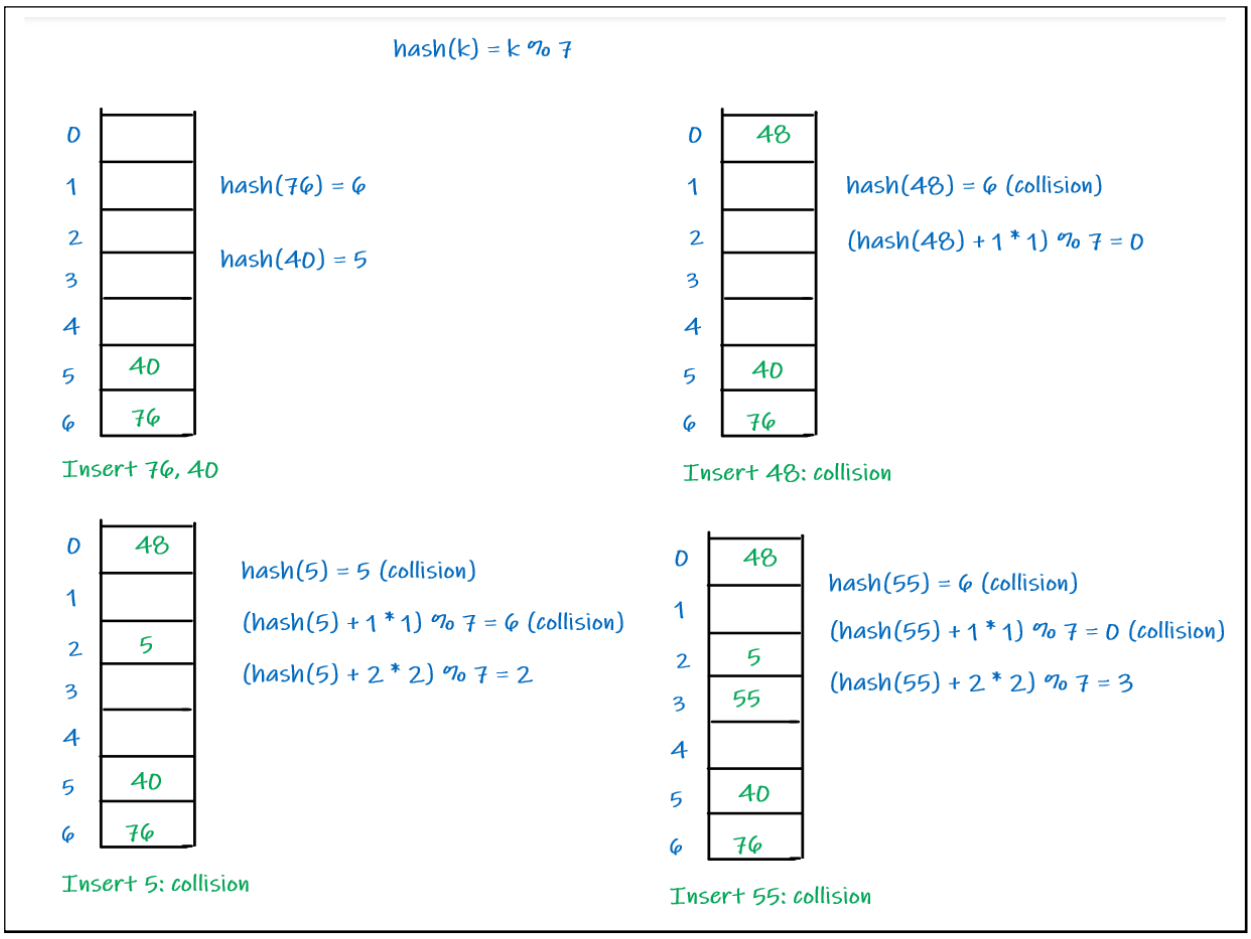
If slot [hash(x) % S] is full, then we try [(hash(x) + 1 * 1) % S]

If [(hash(x) + 1 * 1) % S] is also full, then we try [(hash(x) + 2 * 2) % S]

If [(hash(x) + 2 * 2) % S] is also full, then we try [(hash(x) + 3 * 3) % S]

……

Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 76, 40, 48, 5, 55.



3. Double Hashing

Double hashing uses the idea of applying a second hash function to key when a collision occurs.

$$h(k, i) = (hash1(k) + i * hash2(k)) \bmod S$$

Where:

i: the order of the attempt (i = 0, 1, 2, …)

hash1(k) and hash2(k): hash functions

S: size of the array

let hash(x) be the slot index computed using hash function.

If slot [hash(x) % S] is full, then we try [(hash(x) + 1*hash2(x)) % S]

If [(hash(x) + 1*hash2(x)) % S] is also full, then we try [(hash(x) + 2*hash2(x)) % S]

If [(hash(x) + 2*hash2(x)) % S] is also full, then we try [(hash(x) + 3*hash2(x)) % S]

.....

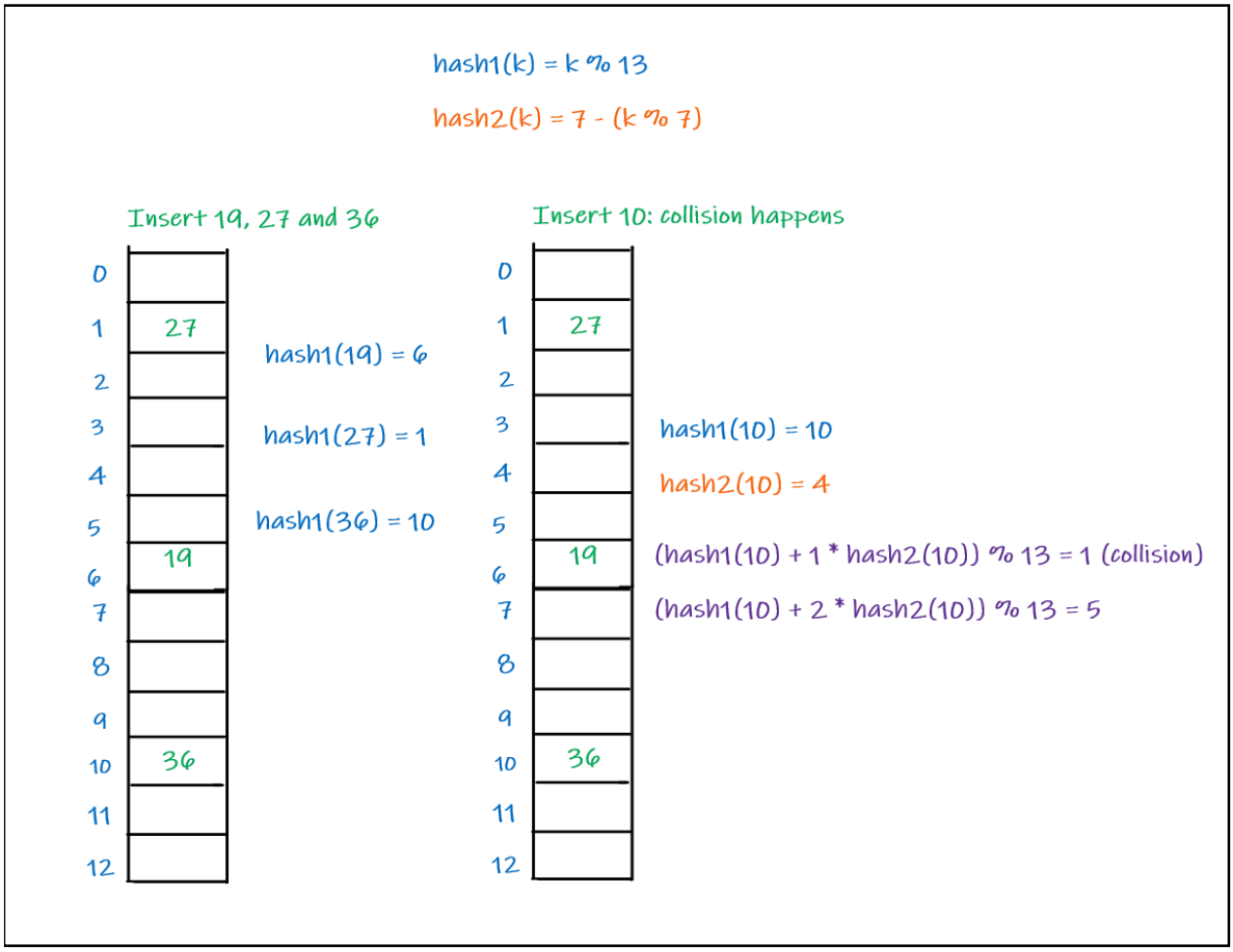
First hash function is typically hash1(k) = k % S

A popular second hash function is: hash2(k) = PRIME – (k % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

A good second Hash function is:

- It must never evaluate to zero.
- Must make sure that all cells can be probed.

Let us consider 2 simple hash functions as "**key mod 13**" and "**key mod 7**", and a sequence of keys as 19, 27, 36, 10.



Topic 4: Sort Algorithms

Contents:

- **Introduction**
- **Describe Sort Algorithms**

I. Introduction.

- Sorting algorithms are used in many data structures and applications to reorder the members of the list based on the comparison between elements.
- There are a variety of different sorting algorithms which will come with different time and space complexity and depend on the case of the data to use the suitable sorting algorithm for it.
- To be easier for illustration, the structure that we are going to sort in this topic is a consecutive sequence of integers (array of integers)
- They can also be classified in many ways like in-place, recursion, non-recursion, stability, …

II. Sort Algorithms.

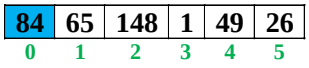
1. Selection Sort.

- Selection Sort proceeds by finding the smallest (or largest, depending on sorting order) element in the right unsorted subarray, exchanging it with the leftmost unsorted element (putting it in sorted order), and expand the sublist boundaries one element to the right.
- Selection sort is an in-place comparison sorting algorithm.
- Time Complexity: $O(n^2)$
- Advantages:
    - It takes the elements to their final positions in the sorted array.
    - Selection sort is typically used in situations where the auxiliary memory is limited because it is a in-place sorting algorithm.
    - Easy to implement.

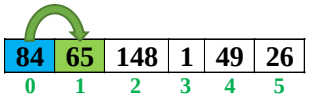
- Disadvantages: High time complexity.
- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in ascending order

Step 1: Find the smallest value's index.

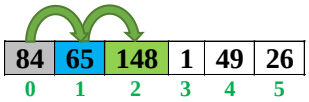


⇨ Initialize **min** with 0 (**84**).



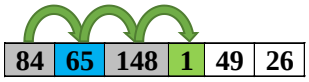
Current **min** index's value: **84 > 65**

⇨ Assign **min** index with **1**.



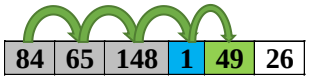
Current **min** index's value: **65 < 148**

⇨ Move to the next value.



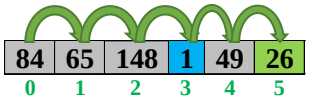
Current **min** index's value: **65 > 1**

⇨ Assign **min** index with **3**.



Current **min** index's value: **1 < 49**

⇨ Move to the next value.

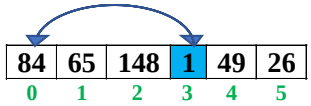


Current **min** index's value: **1 < 26**

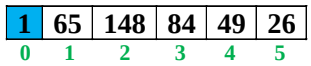
⇨ End of the array is reached.

⇨ Smallest value's index is **min** = **3**.

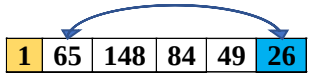
Step 2: Swap the index of the smallest value with the right-most index of the left sorted subarray.



⇨ Swap values of index **0** and **3**.



⇨ Repeat from step 1 until the array is sorted:

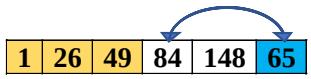


Current smallest value's index: **5**

⇨ Swap values of index **5** and **1**.

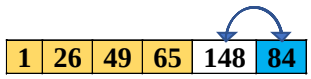
Current smallest value: **4**

⇨ Swap values of index **4** and **2**.



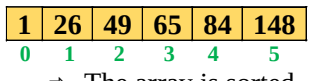
Current smallest value: **5**

⇨ Swap values of index **5** and **3**.



Current smallest value: **5**

⇨ Swap values of index **5** and **4**.

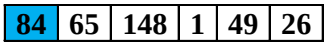


⇨ The array is sorted.

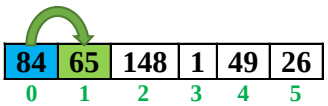
Example 1 - 3

Sort in descending order

Step 1: Find the largest value's index.

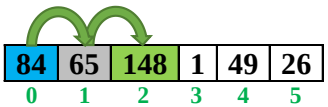


⇨ Initialize **max** with 0 (**84**).



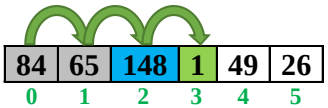
Current **max** index's value: **84** > **65**

⇨ Move to the next value.



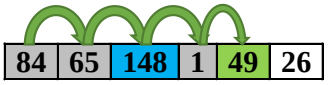
Current **max** index's value: **84** < **148**

⇨ Assign **max** index with **2**.



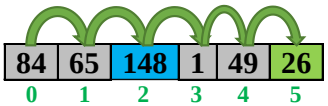
Current **max** index's value: **148** > **1**

⇨ Move to the next value.



Current **max** index's value: **148** > **49**

⇨ Move to the next value.

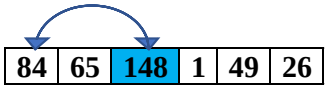


Current **max** index's value: **148** > **26**

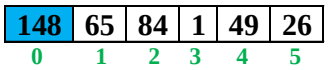
⇨ End of the array is reached.

⇨ Largest value's index is **max** = **2**.

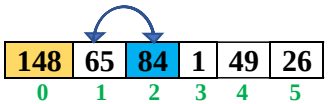
Step 2: Swap the index of the largest value with the right-most index of the left sorted subarray.



⇨ Swap values of index **0** and **3**.

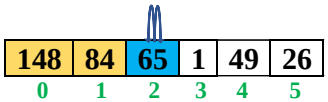


⇨ Repeat from step 1 until the array is sorted:



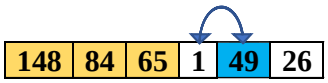
Current largest value: **2**

⇨ Swap values of index **2** and **1**.



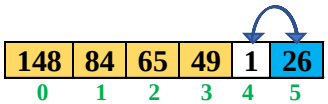
Current largest value: **2**

⇨ Swap values of index **2** and **2**.



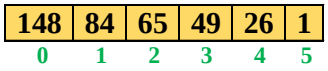
Current largest value: **4**

⇨ Swap values of index **4** and **3**.



Current largest value: **5**

⇨ Swap values of index **5** and **4**.



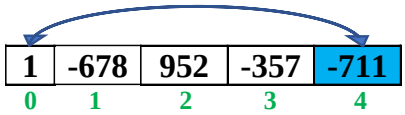
⇨ The array is sorted.

Example 1 - 4

Array B = {**1, -678, 952, -357, -711**}.

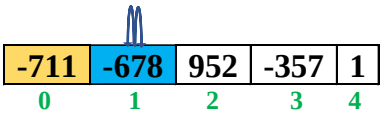
Sort in ascending order

Find the smallest value's index and swap the index of the smallest value with the right-most index of the left sorted subarray.



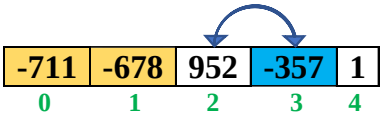
Current smallest value's index: **4** (**-711**)

⇨ Swap values of index **4** and **0**.



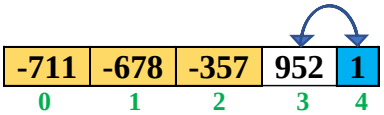
Current smallest value's index: **1** (**-678**)

⇨ Swap values of index **1** and **1**.



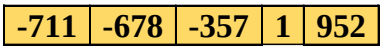
Current smallest value's index: **3** (**-357**)

⇨ Swap values of index **3** and **2**.



Current smallest value's index: **4** (**1**)

⇨ Swap values of index **4** and **3**.



⇨ The array is sorted.

Example 1 - 5

2. Insertion Sort.

- Insertion Sort is a simple sorting algorithm. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part by making some comparisons: if the element is smaller than its predecessor, compare it with the elements before. Move the greater elements one position up to make space for the swapped element.
- Insertion Sort is an in-place and stable comparison sorting algorithm.
- Advantages:
    - Real-time sorting, data may be incomplete or coming, but the array is still sortable.
    - Efficient for small data sets, much like other quadratic sorting algorithms and more efficient than most other simple quadratic algorithms such as *Selection Sort* or *Bubble Sort*
    - Easy to implement.
- Disadvantages:
    - High time complexity.
    - It is much less efficient on large lists than more advanced algorithms such as *Quick Sort*, *Heap Sort*, or *Merge Sort*.

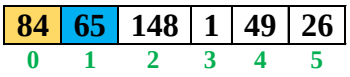
- Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

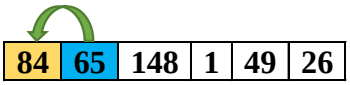
Sort in ascending order

Step 1: Find the niche position for the left-most element from right unsorted subarray.

While traversing, move the value of the current index to the right by 1 element.

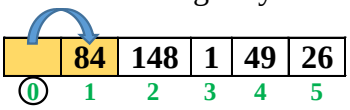


Current left-most value from the unsorted subarray is **v** = **65**.



Current **v** value: **65** < **84**

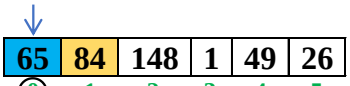
⇨ Move the value of index **0** to the right by 1.



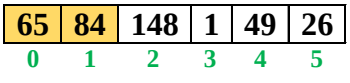
⇨ End of the sorted subarray.

⇨ The niche index for **v** is **0**.

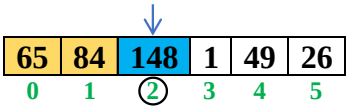
Step 2: And assign the current left-most element from unsorted subarray to that niche position.



Assign **v** = **65** to index **0**

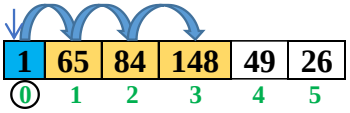


⇨ Repeat from step 1 until the array is sorted:



Current **v** value: **148** > **84**

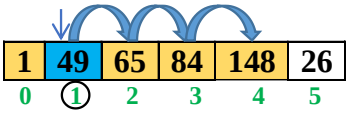
⇨ The niche index for **v** is **2**.



Current **v** value: **1** < **65**

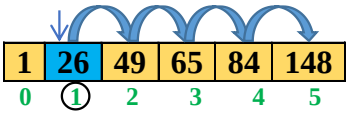
⇨ End of the sorted subarray.

⇨ The niche index for **v** is **0**.



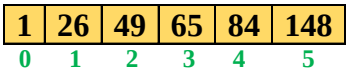
Current **v** value: **49** > **1**

⇨ The niche index for **v** is **1**.



Current **v** value: **26** > **1**

⇨ The niche index for **v** is **1**.



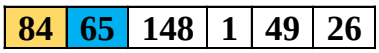
⇨ The array is sorted.

Example 2 - 3

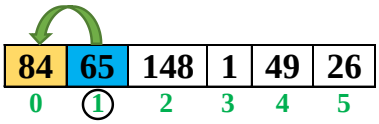
Sort in descending order

Step 1: Find the niche position for the left-most element from right unsorted subarray.

While traversing, move the value of the current index to the right by 1 element.



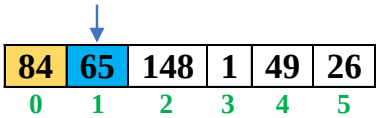
Current left-most value from the unsorted subarray is **v** = **65**.



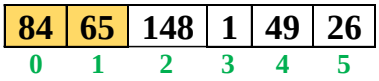
Current **v** value: **65 < 84**

⇨ The niche index for **v** is **1**.

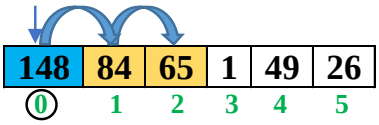
Step 2: And assign the current left-most element from unsorted subarray to that niche position.



Assign **v** = **65** to index **1**



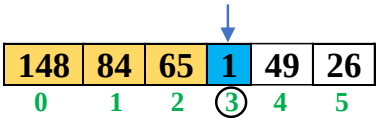
⇨ Repeat from step 1 until the array is sorted:



Current **v** value: **148 > 84**

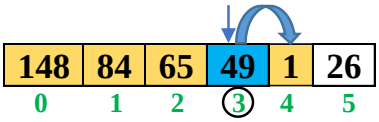
⇨ End of the sorted subarray.

⇨ The niche index for **v** is **0**.



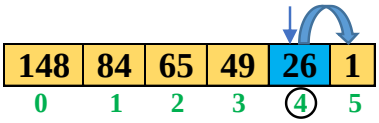
Current **v** value: **1 < 65**

⇨ The niche index for **v** is **3**.



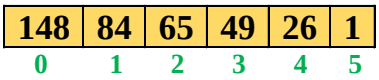
Current **v** value: **49 < 65**

⇨ The niche index for **v** is **3**.



Current **v** value: **26 < 49**

⇨ The niche index for **v** is **4**.



⇨ The array is sorted.

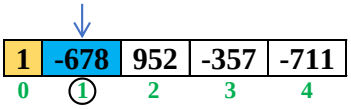
Example 2 - 4

Array B = {**1, -678, 952, -357, -711**}.

Sort in descending order

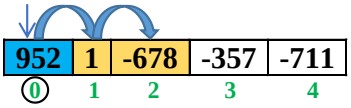
Find the niche position for the left-most element from right unsorted subarray. While traversing, move the value of the current index to the right by 1 element.

Then, assign the current left-most element from unsorted subarray to that niche position.



Current **v** value: **-678 < 1**

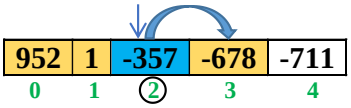
⇨ The niche index for **v** is **1**.



Current **v** value: **952 > 1**

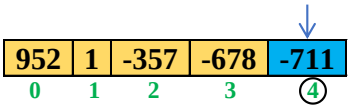
⇨ End of the sorted subarray.

⇨ The niche index for **v** is **1**.



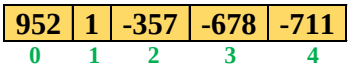
Current **v** value: **-357 < 1**

⇨ The niche index for **v** is **2**.



Current **v** value: **-711 < -678**

⇨ The niche index for **v** is **4**.



⇨ The array is sorted.

Example 2 - 5

3. Binary Insertion Sort.

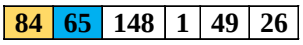
- Binary insertion Sort is a variant of the *Insertion Sort*. Because of finding the insert position is sequential in *Insertion Sort*. For a better way, we use the *Binary Search* method to find this position.
- Binary insertion Sort employs a binary search to determine the correct location to insert new elements, and therefore performs $O(\log_2 n)$ comparisons in the worst case. When each element in the array is searched for and inserted this is O(nlogn).
- But because of the series of swaps required for each insertion. The algorithm as a whole still has a time complexity of $O(n^2)$ on average.
- Advantages: The number of swaps is reduced more than a normal *Insertion Sort*.

- Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in ascending order

Step 1: Find the niche position for the left-most element from right unsorted subarray by using *Binary Search* algorithm.



Current left-most value from the unsorted subarray is **v** = **65**.

Sorted subarray:

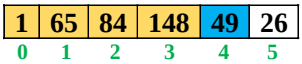
Middle index: (**0**+**0**)/2 = **0**

Middle value: **84** > **v** **(65)**

⇨ End of the sorted subarray.
⇨ The niche index for **v** is **0**.

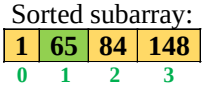
***

Another example of Binary Search:



Current left-most value from the unsorted subarray is **v** = **49**.

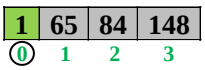
Sorted subarray:



Middle index: (**0**+**3**)/2 = **1**

Middle value: **65** > **v** **(49)**

⇨ Eliminate all elements after index **1**.



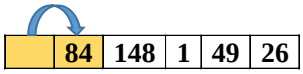
Middle index: (**0**+**0**)/2 = **0**

Middle value: **1** < **v** **(49)**

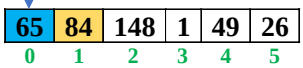
⇨ End of the sorted subarray.
⇨ The niche index for **v** is **0**.

****As you can see here. It is way faster to using Binary Search to find the right position.**

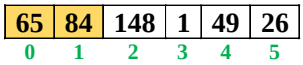
Step 2: Transfer all the element after the position just found to the right by 1.
Then, assign the current left-most element from unsorted subarray to that niche position.



Move all the elements from the index to the right.



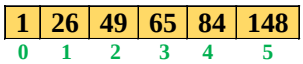
Assign **v** = **65** to index **0**



⇨ Repeat from step 1 until the array is sorted:

****The main different procedure of this variant of insertion sort with normal Insertion Sort is that it uses *Binary Search* to find the index instead of *Liner Search*. Otherwise, the rest of the algorithm is the same concept.**

***You can find all the iteration of this example in the *Insertion Sort* category's example.**



⇨ The array is sorted.

Example 3 - 1

Sort in ascending order

Step 1: Find the niche position for the left-most element from right unsorted subarray by using *Binary Search* algorithm.

84	65	148	1	49	26
0	1	2	3	4	5

Current left-most value from the unsorted subarray is **v** = **65**.

Sorted subarray:

84
0

Middle index: (**0**+**0**)/2 = **0**

Middle value: **84** > **v** **(65)**

⇨ End of the sorted subarray.
⇨ The niche index for **v** is **1**.

***

Another example of Binary Search:

148	84	65	1	49	26
0	1	2	3	4	5

Current left-most value from the unsorted subarray is **v** = **49**.

Sorted subarray:

148	84	65	1
0	1	2	3

Middle index: (**0**+**3**)/2 = **1**

Middle value: **84** > **v** **(49)**

⇨ Eliminate all elements before index **1**.

148	84	65	1
0	1	2	3

Middle index: (**2**+**3**)/2 = **2**

Middle value: **65** > **v** **(49)**

⇨ Eliminate all elements before index **2**.

148	84	65	1
0	1	2	3

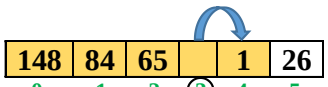
Middle index: (**3**+**3**)/2 = **3**

Middle value: **1** < **v** **(49)**

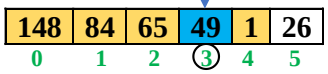
⇨ End of the sorted subarray.
⇨ The niche index for **v** is **3**.

****As you can see here. It is way faster to using Binary Search to find the right position.**

Step 2: Transfer all the element after the position just found to the right by 1.
Then, assign the current left-most element from unsorted subarray to that niche position.



Move all the elements after the index to the right.



Assign **v** = **49** to index **3**

148	84	65	49	1	26
0	1	2	3	4	5

⇨ Repeat from step 1 until the array is sorted:

****The main different procedure of this variant of insertion sort with normal Insertion Sort is that it uses *Binary Search* to find the index instead of *Liner Search*. Otherwise, the rest of the algorithm is the same concept.**

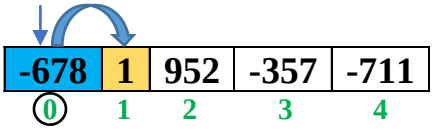
Example 3 - 2

Array B = {**1, -678, 952, -357, -711**}.

Sort in ascending order

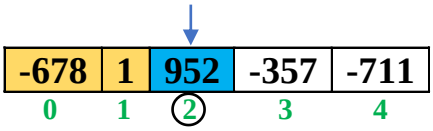
Find the niche position for the left-most element from right unsorted subarray by using *Binary Search* algorithm.

Transfer all the element after the position just found to the right by 1. Then, assign the current left-most element from unsorted subarray to that niche position.



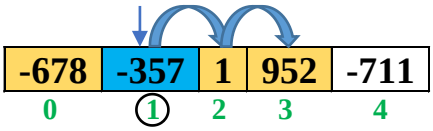
Current left-most value from the unsorted subarray is **v** = **-678**.

⇨ Use *Binary Seach* to find the niche index for **v** is **0**.



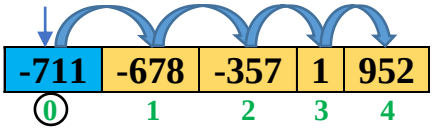
Current left-most value from the unsorted subarray is **v** = **-952**.

⇨ Use *Binary Seach* to find the niche index for **v** is **2**.



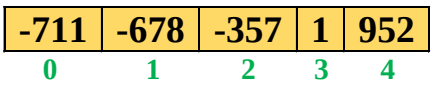
Current left-most value from the unsorted subarray is **v** = **-357**.

⇨ Use *Binary Seach* to find the niche index for **v** is **1**.



Current left-most value from the unsorted subarray is **v** = **-711**.

⇨ Use *Binary Seach* to find the niche index for **v** is **0**.



⇨ The array is sorted.

Example 3 - 3

4. Interchange Sort.

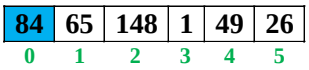
- Interchange Sort works by comparing the first element with all elements behind it, swapping where needed, it then proceeds to do the same for the second element, and so on.
- Interchange Sort is an in-place comparison sort.
- Time complexity: $O(n^2)$
- Advantages:
    - Like any simple $O(n^2)$ sort it can be reasonably fast over very small data sets, though in general insertion sort will be faster.
    - Easy to implement.
- Disadvantages:
    - High time complexity.
    - It is much less efficient on large lists.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

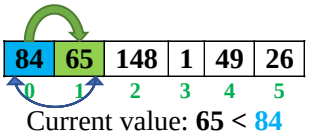
Sort in ascending order

Split the array into 2 subarray the left sorted subarray and right unsorted subarray.

Compare the left most element from unsorted subarray with all the rest elements of it and swap where needed.

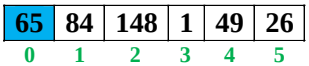


Current left most value from unsorted subarray: **84**.

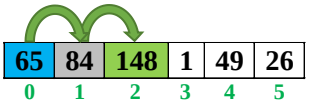


Current value: **65** < **84**

⇨ Swap it with the left most value from unsorted subarray.

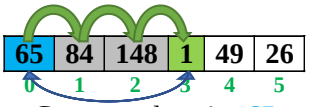


⇨ Check the next value.



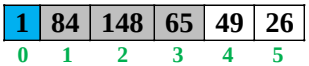
Current value: **148** > **65**

⇨ Check the next value.

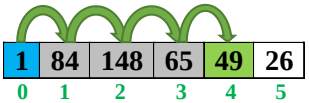


Current value: **1** < **65**

⇨ Swap it with the left most value from unsorted subarray.

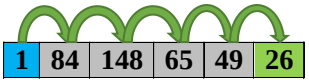


⇨ Check the next value.



Current value: **49** < **1**

⇨ Check to the next value.

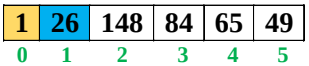


Current value: **26** < **1**

⇨ End of the unsorted subarray.

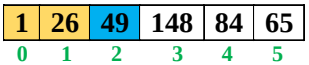
Reduce the right unsorted subarray by 1 element from the left and repeat the action until the array is sorted.

Iteration 2:



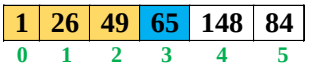
Final left most value from unsorted subarray: **26**.

Iteration 3:



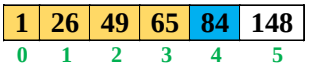
Final left most value from unsorted subarray: **49**.

Iteration 4:



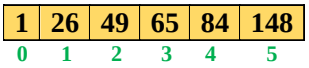
Final left most value from unsorted subarray: **65**.

Iteration 5:



Final left most value from unsorted subarray: **84**.

⇨ End of the array.



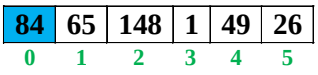
⇨ The array is sorted.

Example 4 - 1

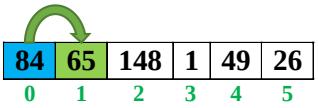
Sort in descending order

Split the array into 2 subarray the left sorted subarray and right unsorted subarray.

Compare the left most element from unsorted subarray with all the rest elements of it and swap where needed.

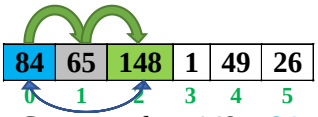


Current left most value from unsorted subarray: **84**.



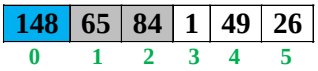
Current value: **65** < **84**

⇨ Check the next value.

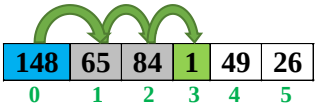


Current value: **148** > **84**

⇨ Swap it with the left most value from unsorted subarray.

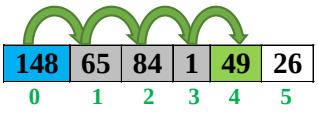


⇨ Check the next value.



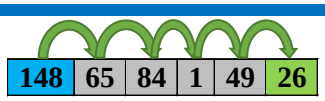
Current value: **1** < **148**

⇨ Check the next value.



Current value: **49** < **148**

⇨ Check the next value.

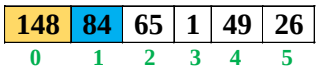


Current value: **26** < **148**

⇨ End of the unsorted subarray.

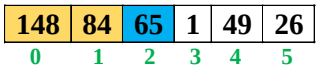
Reduce the right unsorted subarray by 1 element from the left and repeat the action until the array is sorted.

Iteration 2:



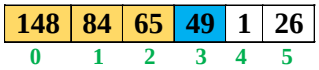
Final left most value from unsorted subarray: **84**.

Iteration 3:



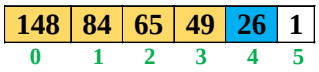
Final left most value from unsorted subarray: **65**.

Iteration 4:



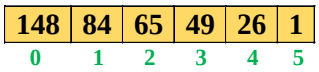
Final left most value from unsorted subarray: **49**.

Iteration 5:



Final left most value from unsorted subarray: **26**.

⇨ End of the array.



⇨ The array is sorted.

Example 4 - 2

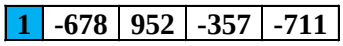
Array B = {**1, -678, 952, -357, -711**}.

Sort in descending order

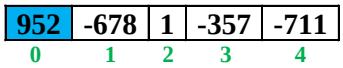
Split the array into 2 subarray the left sorted subarray and right unsorted subarray.Compare the left most element from unsorted subarray with all the rest elements of it and swap where needed.

Then, reduce the right unsorted subarray by 1 element from the left and repeat the action until the array is sorted.

Iteration 1:

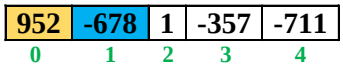


⇨ Current left most value from unsorted subarray: **1**.

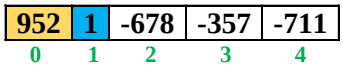


⇨ Final left most value from unsorted subarray: **952**.

Iteration 2:



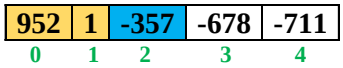
⇨ Current left most value from unsorted subarray: **-678**.



⇨ Final left most value from unsorted subarray: **1**.

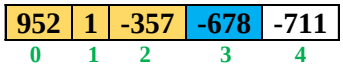
Iteration 3:

⇨ Current left most value from unsorted subarray: **-678**.

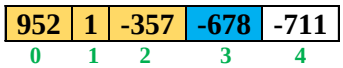


⇨ Final left most value from unsorted subarray: **-357**.

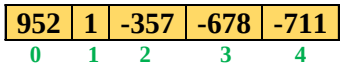
Iteration 4:



⇨ Current left most value from unsorted subarray: **-678**.



⇨ Final left most value from unsorted subarray: **-678**.



⇨ The array is sorted.

Example 4 - 3

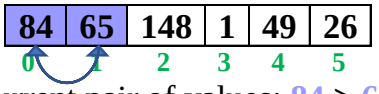
5. Bubble Sort.

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.
- Bubble Sort is an in-place comparison sort.
- Time complexity: $O(n^2)$
- Advantages:
    - Easy to implement.
- Disadvantages:
    - High time complexity.
    - It is much less efficient on large lists.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

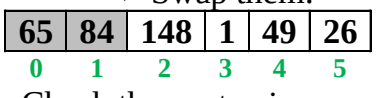
Sort in ascending order

Bubble up the largest element to the surface by comparing two consecutive elements and swap them where needed, then continue with the next one until hit a surface's element.

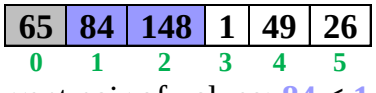


Current pair of values: 84 > 65.

⇨ Swap them.

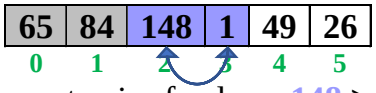


⇨ Check the next pair.



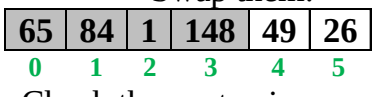
Current pair of values: 84 < 148.

⇨ Check the next pair.

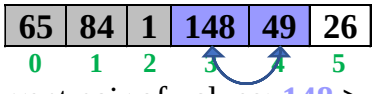


Current pair of values: 148 > 1.

⇨ Swap them.

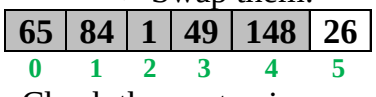


⇨ Check the next pair.

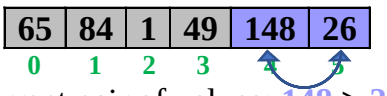


Current pair of values: 148 > 49.

⇨ Swap them.

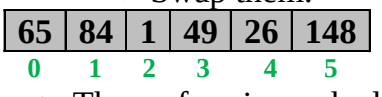


⇨ Check the next pair.



Current pair of values: 148 > 26.

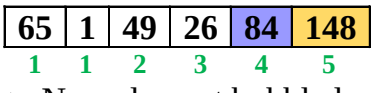
⇨ Swap them.



⇨ The surface is reached.

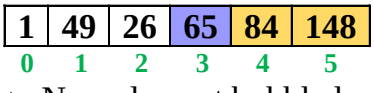
Now the new element just bubbled up will be the newly added surface's element. Repeat the action until the array is sorted.

Iteration 2:



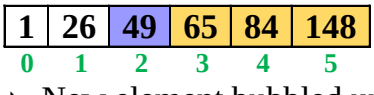
⇨ New element bubbled up to surface: 84.

Iteration 3:



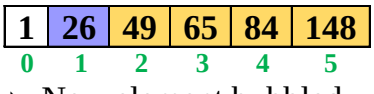
⇨ New element bubbled up to surface: 65.

Iteration 4:

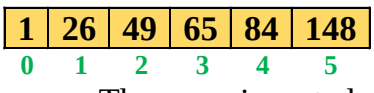


⇨ New element bubbled up to surface: 49.

Iteration 5:



⇨ New element bubbled up to surface: 26.

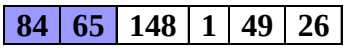


⇨ The array is sorted.

Example 5 - 1

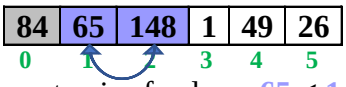
Sort in descending order

Bubble up the smallest element to the surface by comparing two consecutive elements and swap them where needed, then continue with the next one until hit a surface's element.



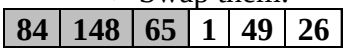
Current pair of values: 84 > 65.

⇨ Check the next pair.



Current pair of values: 65 < 148.

⇨ Swap them.



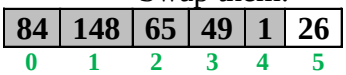
⇨ Check the next pair.

Current pair of values: 65 < 1.

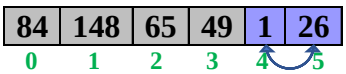
⇨ Check the next pair.

Current pair of values: 1 < 49.

⇨ Swap them.

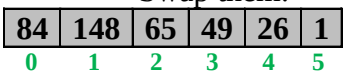


⇨ Check the next pair.



Current pair of values: 1 < 26.

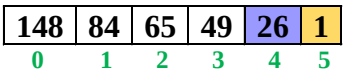
⇨ Swap them.



⇨ The surface is reached.

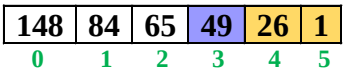
Now the new element just bubbled up will be the newly added surface's element. Repeat the action until the array is sorted.

Iteration 2:



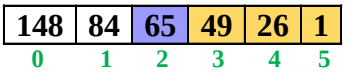
⇨ New element bubbled up to surface: 26.

Iteration 3:



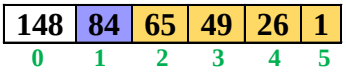
⇨ New element bubbled up to surface: 49.

Iteration 4:

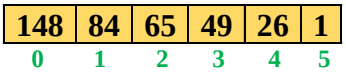


⇨ New element bubbled up to surface: 65.

Iteration 5:



⇨ New element bubbled up to surface: 84.



⇨ The array is sorted.

Example 5 - 2

Array B = {**1, -678, 952, -357, -711**}.

Sort in descending order

Bubble up the smallest element to the surface by comparing two consecutive elements and swap them where needed, then continue with the next one until hit a surface's element.

Now the new element just bubbled up will be the newly added surface's element. Repeat the action until the array is sorted.

-678	1	952	-357	-711
0	1	2	3	4

Iteration 1:

-678	1	952	-357	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -711.

Iteration 2:

1	952	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -678.

Iteration 3:

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -357.

Iteration 4:

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: 1.

952	1	-357	-678	-711
0	1	2	3	4

⇨ The array is sorted.

Example 5 - 3

6. Shaker Sort.

- Shaker Sort is an extension of *Bubble Sort*.
- The algorithm extends *Bubble Sort* by operating in two directions. While it improves on *Bubble Sort* by more quickly moving items to the beginning of the list.
- Shaker Sort is an in-place comparison sort.
- Time complexity: $O(n^2)$
- Advantages:
    - If the list is mostly ordered before applying the sorting algorithm it becomes closer to O(n) time complexity.
- Disadvantages:
    - High time complexity.
    - It is much less efficient on large lists.
    - Even it is an extension of *Bubble Sort* but it still not be better than *Insertion Sort*.

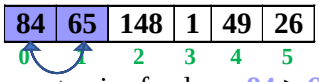
- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in ascending order

Bubble up the largest element to the surface and sink down the smallest one to the bottom by comparing two consecutive elements and swap them where needed, continue with the next one until hit a surface's element.

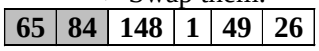
Do the same with the opposite direction until hit a bottom's element.

Phase 1: Bubble up

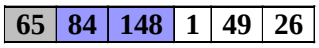


Current pair of values: 84 > 65.

⇨ Swap them.

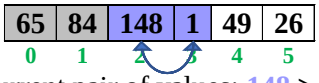


⇨ Check the next pair.



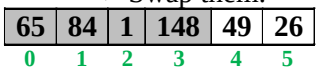
Current pair of values: 84 < 148.

⇨ Check the next pair.

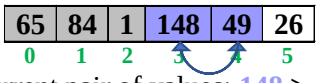


Current pair of values: 148 > 1.

⇨ Swap them.

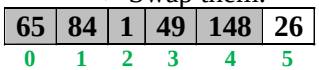


⇨ Check the next pair.

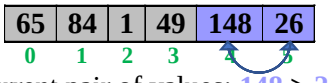


Current pair of values: 148 > 49.

⇨ Swap them.

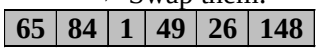


⇨ Check the next pair.



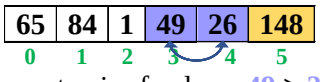
Current pair of values: 148 > 26.

⇨ Swap them.



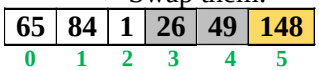
⇨ The surface is reached.

Phase 2: Sink down

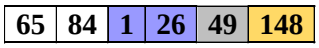


Current pair of values: 49 > 26.

⇨ Swap them.



⇨ Check the next pair.

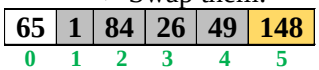


Current pair of values: 1 < 26.

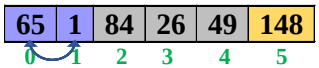
⇨ Check the next pair.

Current pair of values: 84 > 1.

⇨ Swap them.

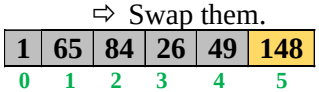


⇨ Check the next pair.



Current pair of values: 84 > 1.

⇨ Swap them.



⇨ The bottom is reached.

Example 6 - 1.1

Now the element just bubbled up will be the newly added surface's element and the element just sinked down will be the newly added bottom's element. Repeat the action until the array is sorted.

Iteration 2:

1	65	26	49	84	148
0	1	2	3	4	5

⇨ New element bubbled up to surface: 84.

1	26	65	49	84	148
0	1	2	3	4	5

⇨ New element sinked down to bottom: 26.

Iteration 3:

1	26	49	65	84	148
0	1	2	3	4	5

⇨ New element bubbled up to surface: 65.

1	26	49	65	84	148
0	1	2	3	4	5

⇨ New element sinked down to bottom: 49.

1	26	49	65	84	148
0	1	2	3	4	5

⇨ The array is sorted.

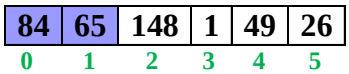
Example 6 - 1.2

Sort in descending order

Bubble up the smallest element to the surface and sink down the largest one to the bottom by comparing two consecutive elements and swap them where needed, continue with the next one until hit a surface's element.

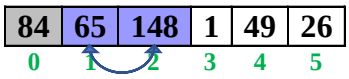
Do the same with the opposite direction until hit a bottom's element.

Phase 1: Bubble up



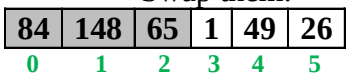
Current pair of values: 84 > 65.

⇨ Check the next pair.

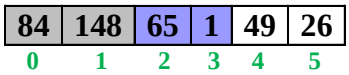


Current pair of values: 65 < 148.

⇨ Swap them.



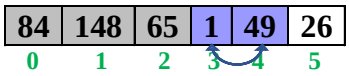
⇨ Check the next pair.



Current pair of values: 65 > 1.

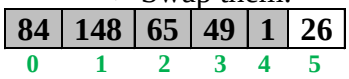
⇨ Check the next pair.

⇨

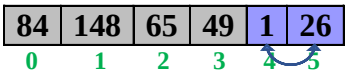


Current pair of values: 1 < 49.

⇨ Swap them.

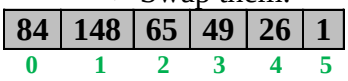


⇨ Check the next pair.



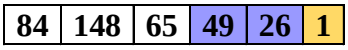
Current pair of values: 1 < 26.

⇨ Swap them.



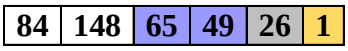
⇨ The surface is reached.

Phase 2: Sink down



Current pair of values: 49 > 26.

⇨ Check the next pair.

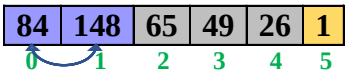


Current pair of values: 65 > 49.

⇨ Check the next pair.

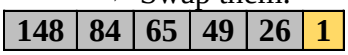
Current pair of values: 148 > 65.

⇨ Check the next pair.



Current pair of values: 84 < 148.

⇨ Swap them.

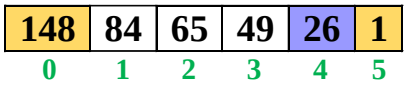


⇨ The bottom is reached.

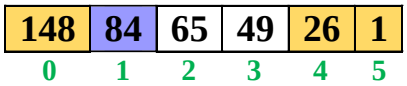
Example 6 – 2.1

Now the element just bubbled up will be the newly added surface's element and the element just sinked down will be the newly added bottom's element. Repeat the action until the array is sorted.

Iteration 2:

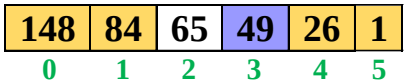


⇨ New element bubbled up to surface: 26.

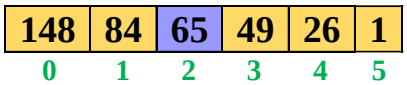


⇨ New element sinked down to bottom: 84.

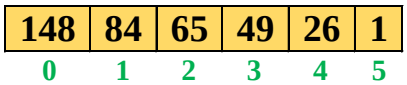
Iteration 3:



⇨ New element bubbled up to surface: 49.



⇨ New element sinked down to bottom: 65.



⇨ The array is sorted.

Example 6 – 2.2

Array B = {**1, -678, 952, -357, -711**}.

Sort in descending order

Now the element just bubbled up will be the newly added surface's element and the element just sinked down will be the newly added bottom's element. Repeat the action until the array is sorted.

1	-678	952	-357	-711
0	1	2	3	4

Iteration 1:

1	952	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -711.

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element sinked down to bottom: 952.

Iteration 2:

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -678.

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element sinked down to bottom: 1.

Iteration 3:

952	1	-357	-678	-711
0	1	2	3	4

⇨ New element bubbled up to surface: -357.

952	1	-357	-678	-711
0	1	2	3	4

⇨ The array is sorted.

Example 6 – 3

7. Shell Sort.

- Shell Sort is a modified version of *Insertion Sort* that allows the exchange of items that are far apart.
- The sorting algorithm compares elements separated and using *Insertion Sort* methods but by a distance and decreases on each pass.
- Shell Sort is an in-place comparison sort.
- Time complexity: $O(n^2)$
- Advantages:
    - Shell Sort has distinctly improved running times more than *Insertion Sort.*
- Example:

Note that: You can freely choose the distance apart of the elements (gap**). In these examples, The **gap** will start with **n/3** and will be reduced by dividing it by **2** in each pass.

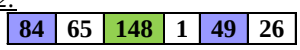
Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in ascending order

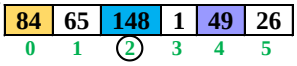
Start with gap = n/3 = 2. Traversing through the array and put the current element into niche position just like the *Insertion Sort* algorithm.

Start with index = gap = 2

Index 2:



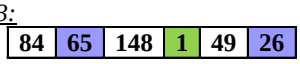
These are the elements corresponding with **gap = 2** for element with index = **2**.



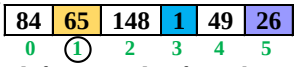
Current left-most value from the unsorted subarray is **v** = **148** and the niche index for **v** is **2.**

⇨ Check the next index.

Index 3:

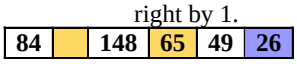


These are the elements corresponding with **gap = 2** for element with index = **3**.

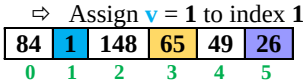


Current left-most value from the unsorted subarray is **v** = **1** and the niche index for **v** is **1.**

⇨ Move all values after index **1** to the right by 1.

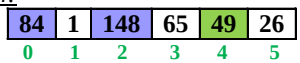


⇨ Assign **v** = **1** to index **1**

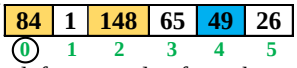


⇨ Check the next index.

Index 4:

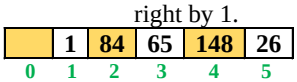


These are the elements corresponding with **gap = 2** for element with index = **4**.

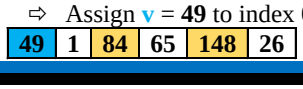


Current left-most value from the unsorted subarray is **v** = **49** and the niche index for **v** is **0.**

⇨ Move all values after index **0** to the right by 1.

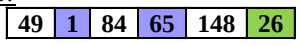


⇨ Assign **v** = **49** to index **0**

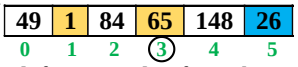


⇨ Check the next index.

Index 5:

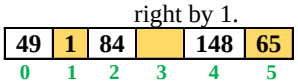


These are the elements corresponding with **gap = 2** for element with index = **5**.

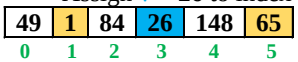


Current left-most value from the unsorted subarray is **v** = **26** and the niche index for **v** is **3.**

⇨ Move all values after index **0** to the right by 1.

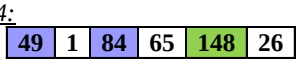


⇨ Assign **v** = **26** to index **3**

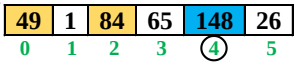


⇨ Check the next index.

Index 4:



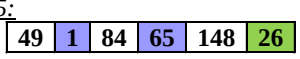
These are the elements corresponding with **gap = 2** for element with index = **4**.



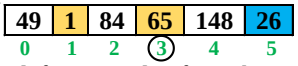
Current left-most value from the unsorted subarray is **v** = **148** and the niche index for **v** is **4.**

⇨ Check the next index.

Index 5:

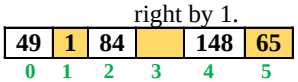


These are the elements corresponding with **gap = 2** for element with index = **4**.

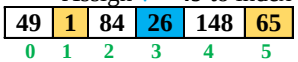


Current left-most value from the unsorted subarray is **v** = **26** and the niche index for **v** is **3.**

⇨ Move all values after index **3** to the right by 1.



⇨ Assign **v** = **49** to index **0**

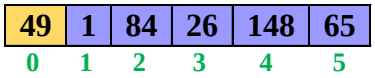


⇨ End of the array.

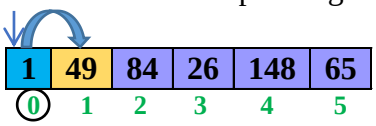
Example 7 - 1.1

Repeat the action but reduce the gap by 2 until gap = 0

gap = n/6 = 1:

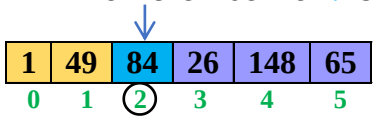


These are the elements corresponding with **gap = 1**



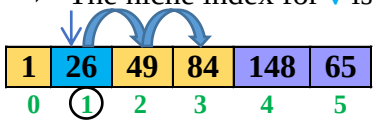
Current left-most value from the unsorted subarray is **v** = **1**

⇨ The niche index for **v** is **0.**



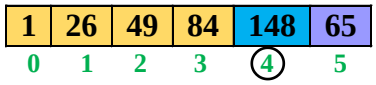
Current left-most value from the unsorted subarray is **v** = **84**

⇨ The niche index for **v** is **2.**



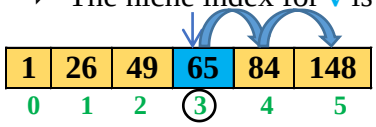
Current left-most value from the unsorted subarray is **v** = **26**

⇨ The niche index for **v** is **1.**



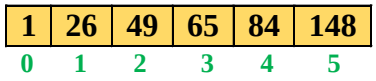
Current left-most value from the unsorted subarray is **v** = **148**

⇨ The niche index for **v** is **4.**



Current left-most value from the unsorted subarray is **v** = **65**

⇨ The niche index for **v** is **3.**



⇨ The array is sorted.

Example 7 - 1.2

Sort in descending order

Start with gap = n/3 = 2. Traversing through the array and put the current element into niche position just like the *Insertion Sort* algorithm.

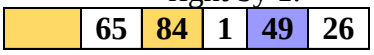
Start with index = gap = 2

Index 2:

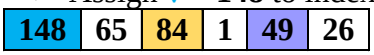
These are the elements corresponding with **gap = 2** for element with index = **2**.

Current left-most value from the unsorted subarray is **v** = **148** and the niche index for **v** is **0.**

⇨ Move all values after index **0** to the right by 1.



⇨ Assign **v** = **148** to index **0**



⇨ Check the next index.

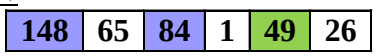
Index 3:

These are the elements corresponding with **gap = 2** for element with index = **3**.

Current left-most value from the unsorted subarray is **v** = **1** and the niche index for **v** is **3.**

⇨ Check the next index.

Index 4:



These are the elements corresponding with **gap = 2** for element with index = **4**.

Current left-most value from the unsorted subarray is **v** = **49** and the niche index for **v** is **4.**

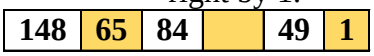
⇨ Check the next index.

Index 5:

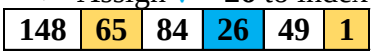
These are the elements corresponding with **gap = 2** for element with index = **5**.

Current left-most value from the unsorted subarray is **v** = **26** and the niche index for **v** is **3.**

⇨ Move all values after index **0** to the right by 1.



⇨ Assign **v** = **26** to index **3**

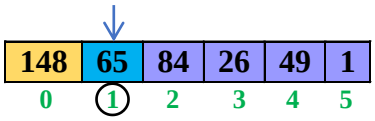


⇨ End of the array.

Example 7 - 2.1

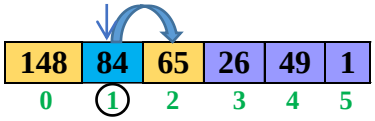
Repeat the action but reduce the gap by 2 until gap = 0

gap = n/6 = 1:



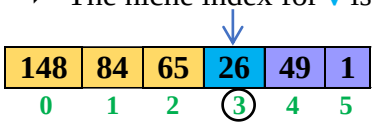
Current left-most value from the unsorted subarray is **v** = **65**

⇨ The niche index for **v** is **1.**



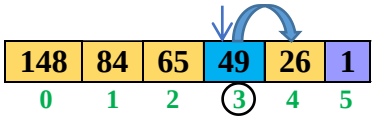
Current left-most value from the unsorted subarray is **v** = **84**

⇨ The niche index for **v** is **1.**



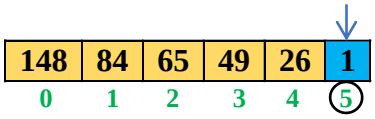
Current left-most value from the unsorted subarray is **v** = **26**

⇨ The niche index for **v** is **3.**



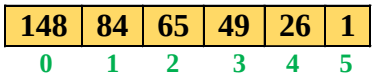
Current left-most value from the unsorted subarray is **v** = **49**

⇨ The niche index for **v** is **3.**



Current left-most value from the unsorted subarray is **v** = **1**

⇨ The niche index for **v** is **5.**



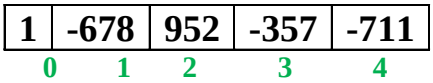
⇨ The array is sorted.

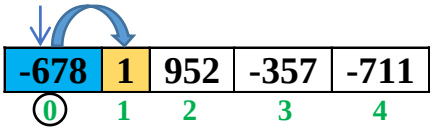
Example 7 – 2.2

Array B = {**1, -678, 952, -357, -711**}.

Sort in ascending order

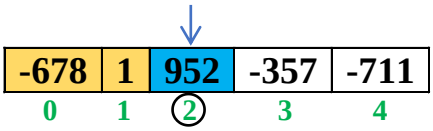
Start with gap = n/3 = 2. Traversing through the array and put the current element into niche position just like the Insertion Sort algorithm. Repeat the action but reduce the gap by 2 until gap = 0





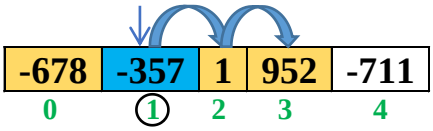
Current left-most value from the unsorted subarray is **v** = **65**

⇨ The niche index for **v** is **0.**



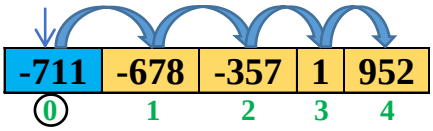
Current left-most value from the unsorted subarray is **v** = **952**

⇨ The niche index for **v** is **2.**



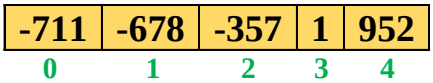
Current left-most value from the unsorted subarray is **v** = **-357**

⇨ The niche index for **v** is **1.**



Current left-most value from the unsorted subarray is **v** = **-711**

⇨ The niche index for **v** is **0.**



⇨ The array is sorted.

Example 7 - 3

8. Heap Sort.

- Heap Sort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.
- The concept of it seems like it is very similar with *Selection Sort*, but Heap Sort does not waste time with a linear-time scan of the unsorted region. Rather, Heap Sort maintains the unsorted region in a *Heap Data Structure* or *Binary Heap* to more quickly find the largest element in each step.
- A *Binary Heap* is a complete *Binary Tree* where items are stored in a special order such that the value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called *Max Heap* and the latter is called *Min Heap*. The heap can be represented by a binary tree or array.
- Heap sort is an in-place comparison sorting algorithm, but it is not a stable sort.
- Time complexity: O(nlogn)
- Advantages:
    - Low time complexity.
- Disadvantages:
    - The implementation of the *Quick Sort* is still better.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

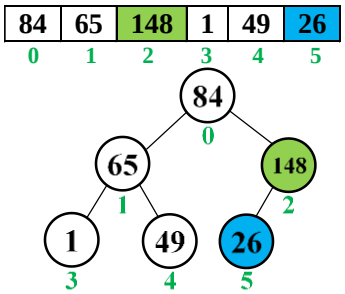
Sort in ascending order

Create a heap tree which have the principles:

- **The parents elements will have the index *i* <= *n/2 – 1*.**
- **A parent element will have 2 children whose index are 2*i + 1 and 2*i + 2.**

Make the heap tree become a max-heap tree by swapping the parent node with their children whose value is greater than its parent. Traverse down to the child just have swapped. If no swap is occurred traverse up. (Heapify)

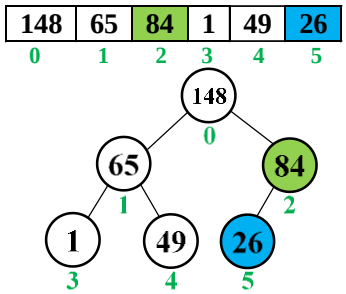
Start with the last parent element/node whose index is n/2 – 1 = 2



⇨ Parrent: 148 and Child: 26

⇨ 26 < 148

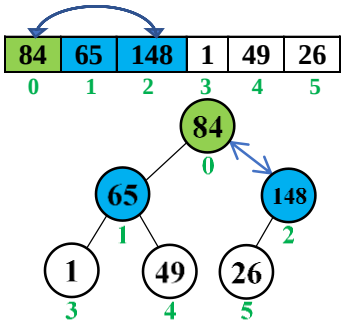
⇨ No swap is needed, traverse up.



⇨ Parrent: 84 and Child: 26

⇨ 26 < 84

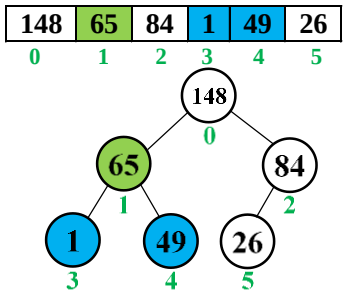
⇨ No swap is needed, traverse up.



⇨ Parrent: 84 and Children: 65, 148

⇨ 148 > 84

⇨ Swap them, traverse down.



⇨ Parrent: 84 and Children: 1, 49

⇨ 1 < 49 < 65

⇨ No swap is needed, End of tree.

Example 8 - 1.1

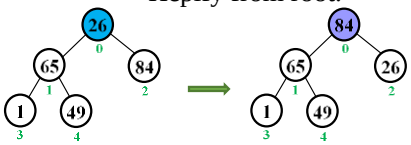
Swap the root of the max-heap tree with the last element of the array and remove it from the tree. Heapify from the root node again and swap until the array is sorted.

148	65	84	1	49	26
0	1	2	3	4	5

⇨ Swap the first and last element.

26	65	84	1	49	148
0	1	2	3	4	5

⇨ Hepify from root.

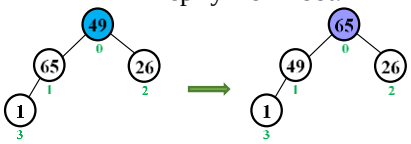


84	65	26	1	49	148
0	1	2	3	4	5

⇨ Swap the first and last element.

49	65	26	1	84	148
0	1	2	3	4	5

⇨ Hepify from root.

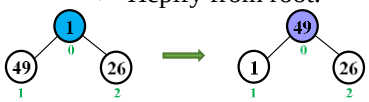


65	49	26	1	84	148
0	1	2	3	4	5

⇨ Swap the first and last element.

1	49	26	65	84	148
0	1	2	3	4	5

⇨ Hepify from root.

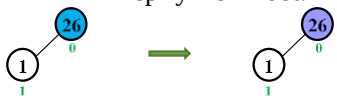


49	1	26	65	84	148
0	1	2	3	4	5

⇨ Swap the first and last element.

26	1	49	65	84	148
0	1	2	3	4	5

⇨ Hepify from root.



26	1	49	65	84	148
0	1	2	3	4	5

⇨ Swap the first and last element.

1	26	49	65	84	148
0	1	2	3	4	5

⇨ End of Array.

1	26	49	65	84	148
0	1	2	3	4	5

⇨ The array is sorted.

Example 8 – 1.2

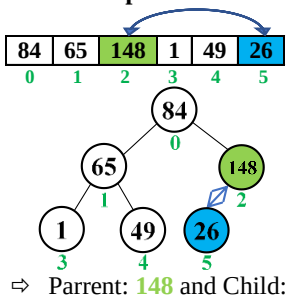
Sort in descending order

Create a heap tree which have the principles:

- **The parents elements will have the index *i <= n/2 – 1*.**
- **A parent element will have 2 children whose index are 2*i + 1 and 2*i + 2.**

Make the heap tree become a min-heap tree by swapping the parent node with their children whose value is lower than its parent. Traverse down to the child just have swapped. If no swap is occurred traverse up. (Heapify)

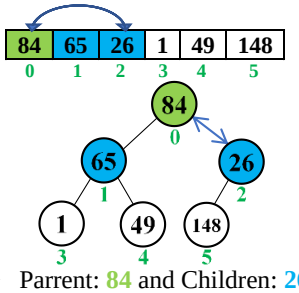
Start with the last parent element/node whose index is n/2 – 1 = 2



⇨ Parrent: 148 and Child: 26

⇨ 26 < 148

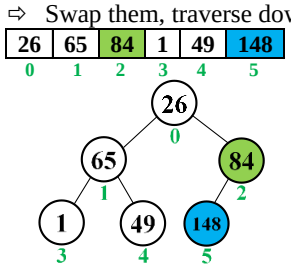
⇨ Swap them, the bottom is reached.



⇨ Parrent: 84 and Children: 26, 65

⇨ 26 < 65 < 84

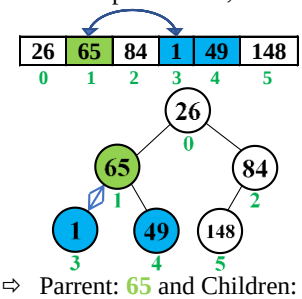
⇨ Swap them, traverse down.



⇨ Parrent: 84 and Children: 148

⇨ 148 > 84

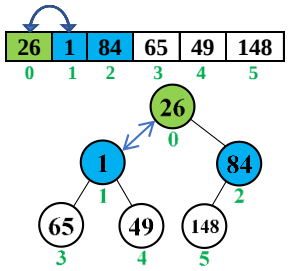
⇨ No swap is needed, traverse up.



⇨ Parrent: 65 and Children: 1, 49

⇨ 1 < 49 < 65

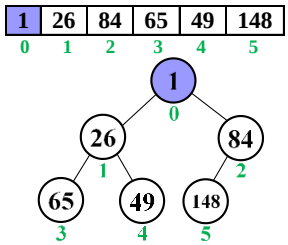
⇨ Swap them, traverse up.



⇨ Parrent: 65 and Children: 1, 49

⇨ 1 < 26 < 84

⇨ Swap them, End of Tree.



Example 8 - 2.1

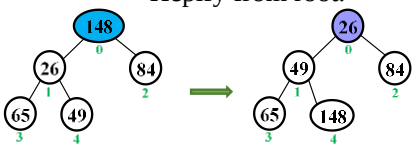
Swap the root of the max-heap tree with the last element of the array and remove it from the tree. Heapify from the root node again and swap until the array is sorted.

1	26	84	65	49	148
0	1	2	3	4	5

⇨ Swap the first and last element.

148	26	84	65	49	1
0	1	2	3	4	5

⇨ Hepify from root.

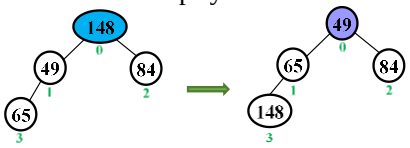


26	49	84	65	148	1
0	1	2	3	4	5

⇨ Swap the first and last element.

148	49	84	65	26	1
0	1	2	3	4	5

⇨ Hepify from root.

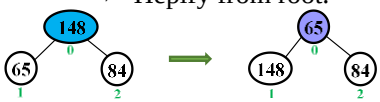


49	65	84	148	26	1
0	1	2	3	4	5

⇨ Swap the first and last element.

148	65	84	49	26	1
0	1	2	3	4	5

⇨ Hepify from root.

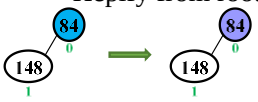


65	148	84	49	26	1
0	1	2	3	4	5

⇨ Swap the first and last element.

84	148	65	49	26	1
0	1	2	3	4	5

⇨ Hepify from root.



84	148	65	49	26	1
0	1	2	3	4	5

⇨ Swap the first and last element.

148	84	65	49	26	1
0	1	2	3	4	5

⇨ End of Array.

148	84	65	49	26	1
0	1	2	3	4	5

⇨ The array is sorted.

Example 8 – 2.2

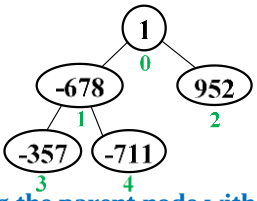
Array B = {**1, -678, 952, -357, -711**}.

Sort in ascending order

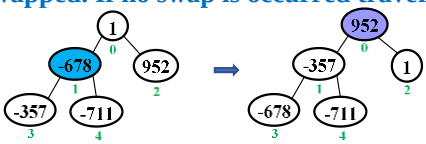
Create a heap tree which have the principles:

- **The parents elements will have the index *i* <= *n/2 – 1*.**
- **A parent element will have 2 children whose index are 2*i + 1 and 2*i + 2.**

1	-678	952	-357	-711
0	1	2	3	4



Make the heap tree become a min-heap tree by swapping the parent node with their children whose value is lower than its parent. Traverse down to the child just have swapped. If no swap is occurred traverse up. (Heapify)

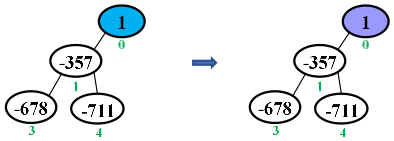


952	-357	-678	-711	1
0	1	2	3	4

⇨ Swap the first and last element.

1	-357	-678	-711	952
0	1	2	3	4

⇨ Hepify from root.

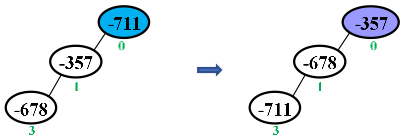


1	-357	-678	-711	952
0	1	2	3	4

⇨ Swap the first and last element.

-711	-357	-678	1	952
0	1	2	3	4

⇨ Hepify from root.



-357	-678	-711	1	952
0	1	2	3	4

⇨ Swap the first and last element.

-711	-678	-357	1	952
0	1	2	3	4

⇨ Hepify from root.

-678	-711	-357	1	952
0	1	2	3	4

⇨ Swap the first and last element.

-711	-678	-711	1	952
0	1	2	3	4

⇨ End of Array.

-711	-678	-711	1	952
0	1	2	3	4

⇨ The array is sorted.

Example 8 - 3

9. Quick Sort.

- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place.
- Quick Sort is an in-place and unstable sorting algorithm and also is a divide and conquer algorithm.
- Time complexity: O(nlogn)
- Advantages:
    - Low time complexity. Faster than *Merge Sort* and about 2 or 3 times faster than *Heap Sort*.
- Disadventages:
    - Requiring small additional amounts of memory to perform the sorting.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in ascending order

Choose a element as pivot. In this example, the middle element will be a pivot.
Step 1: Start from left, traverse to the right to find a element greater than(>=) pivot and from the right, find an element smaller than(<=) pivot. Then, swap them and continue unless all elements are traversed.
pivot** = middle value = **148
Phase 1: Find the greater element.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **84** < **pivot** = **148**.
⇨ Check the next value.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **65** < **pivot** = **148**.
⇨ Check the next value.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **148** = **pivot** = **148**.
⇨ The greater element: **148**.

Phase 2: Find the smaller element.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **26** < **pivot** = **148**.
⇨ The smaller element: **26**.

Phase 3: Swap.

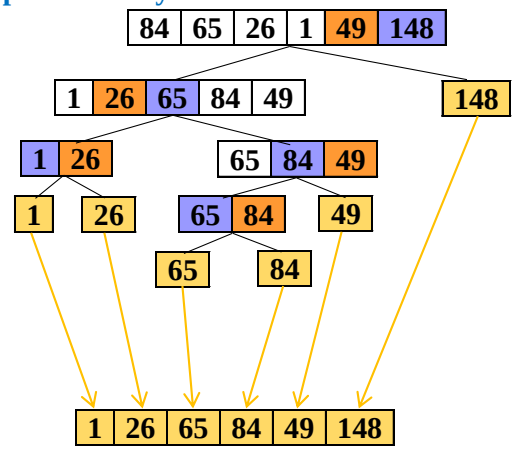
84	65	148	1	49	26
0	1	2	3	4	5

⇨ Swap the 2 elements.

84	65	26	1	49	148
0	1	2	3	4	5

⇨ Repeat the actions.

Step 2: Do the partition. Divide the array into 2 parts:
+ Part 1: From left to the lastest smaller element.
+ Part 2: From right to the lastest greater element.
Do the same with these 2 parts from the step 1 until the divided part has only 1 element.



⇨ Merge back all the part.
⇨ The array is sorted.

Example 9 - 1

Sort in descending order

Choose a element as pivot. In this example, the middle element will be a pivot.
Step 1: Start from left, traverse to the right to find a element smaller than(>=) pivot and from the right, find an element greater than(<=) pivot. Then, swap them and continue unless all elements are traversed.

pivot = middle value = 148

Phase 1: Find the smaller element.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **84** < **pivot** = **148**.
⇨ The smaller element: **84**.

Phase 2: Find the greater element.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **26** < **pivot** = **148**.
⇨ Check the next value.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **49** < **pivot** = **148**.
⇨ Check the next value.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **1** < **pivot** = **148**.
⇨ Check the next value.

84	65	148	1	49	26
0	1	2	3	4	5

Current value: **148** = **pivot** = **148**.
⇨ The greater element: **148**.

Phase 3: Swap.

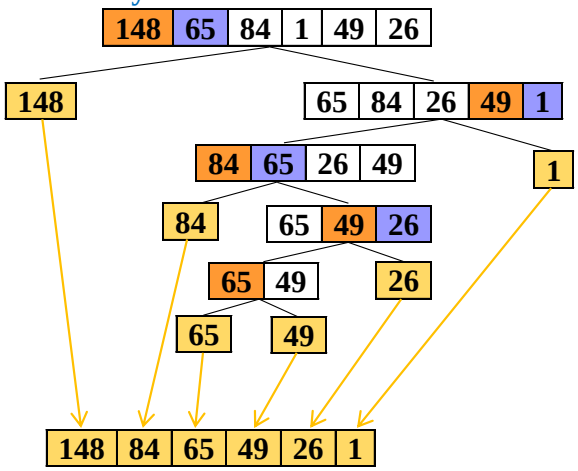
84	65	148	1	49	26
0	1	2	3	4	5

⇨ Swap the 2 elements.

148	65	84	1	49	26
0	1	2	3	4	5

⇨ Repeat the actions.

Step 2: Do the partition. Divide the array into 2 parts:
+ Part 1: From left to the lastest greatest element.
+ Part 2: From right to the lastest smallest element.
Do the same with these 2 parts from the step 1 until the divided part has only 1 element.



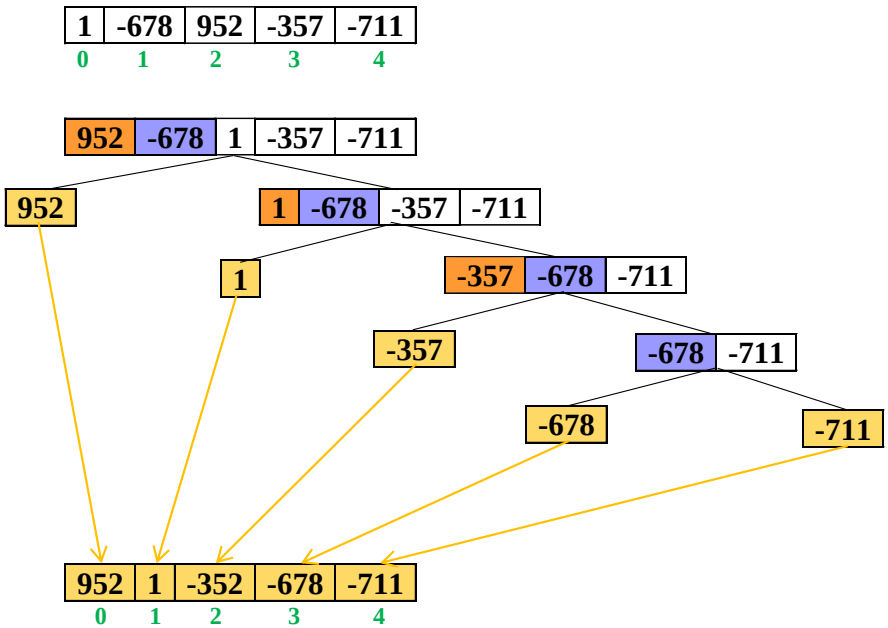
⇨ Merge back all the part.
⇨ The array is sorted.

Example 9 - 2

Array B = {**1, -678, 952, -357, -711**}.

Sort in descending order

Choose a element as pivot. In this example, the middle element will be a pivot.
Step 1: Start from left, traverse to the right to find a element smaller than(>=) pivot and from the right, find an element greater than(<=) pivot. Then, swap them and continue unless all elements are traversed.
Step 2: Do the partition. Divide the array into 2 parts:
+ Part 1: From left to the lastest greatest element.
+ Part 2: From right to the lastest smallest element.
Do the same with these 2 parts from the step 1 until the divided part has only 1 element.



⇨ Merge back all the parts.
⇨ The array is sorted.

Example 9 - 3

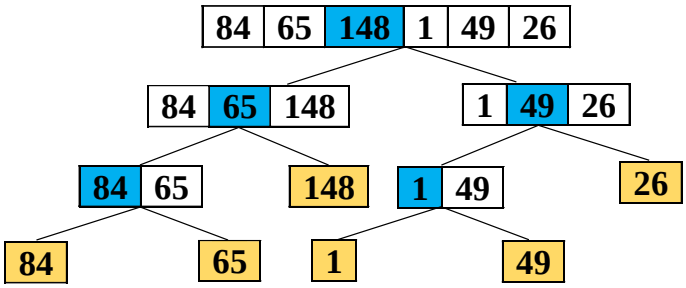
10.Merge Sort.

- Merge Sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
- Merge Sort is an in-place and stable comparison sorting algorithm and also is a divide and conquer algorithm.
- Time complexity: O(nlogn)
- Advantages:
    - Low time complexity.
- Disadventages:
    - Requiring small additional amounts of memory to perform the sorting.
    - Goes through the whole process even if the list is sorted.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

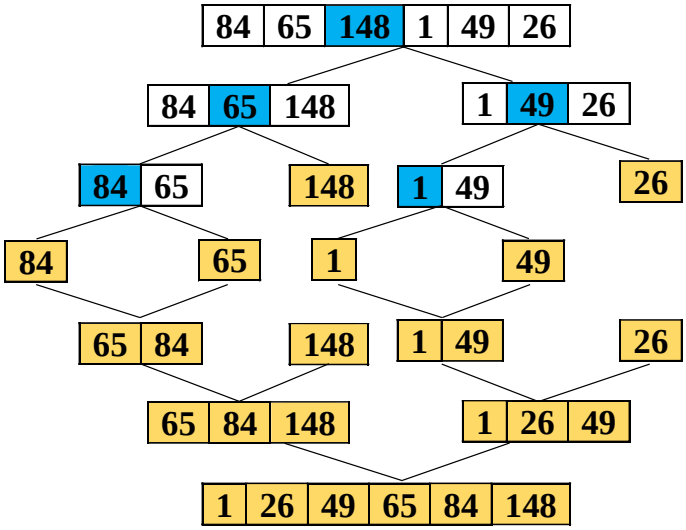
Sort in ascending order

First, separate the array in half and iterate the same with its two subarrays until the subarray has only one element.



Second, merge all the subarray into a big array but in the ascending order by taking out one by one an element which is smaller than the element from the other subarray and put in the new merged array from those two.

The 2 subarrays are always in ascending order (due to the recursion, the 2 subarrays actually used to be another merged array). So you can traverse from left to right through them to find the appropriate element.

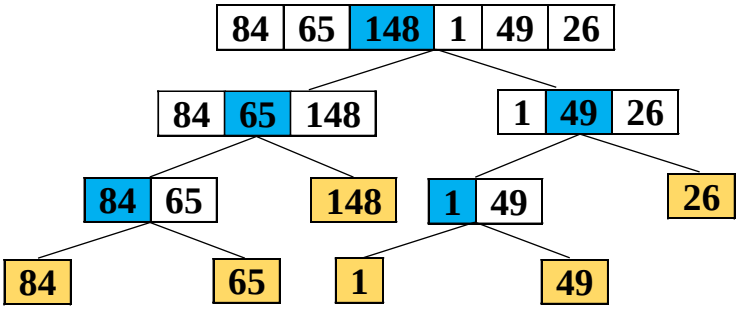


⇨ The array is sorted.

Example 10 - 1

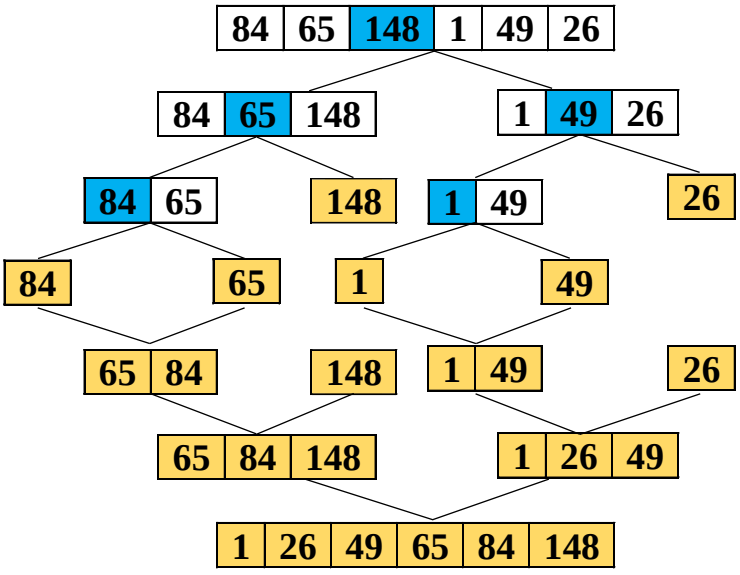
Sort in descending order

First, separate the array in half and iterate the same with its two subarrays until the subarray has only one element.



Second, merge all the subarray into a big array but in the descending order by taking out one by one an element which is greater than the element from the other subarray and put in the new merged array from those two.

The 2 subarrays are always in descending order (due to the recursion, the 2 subarrays actually used to be another merged array). So you can traverse from left to right through them to find the appropriate element.

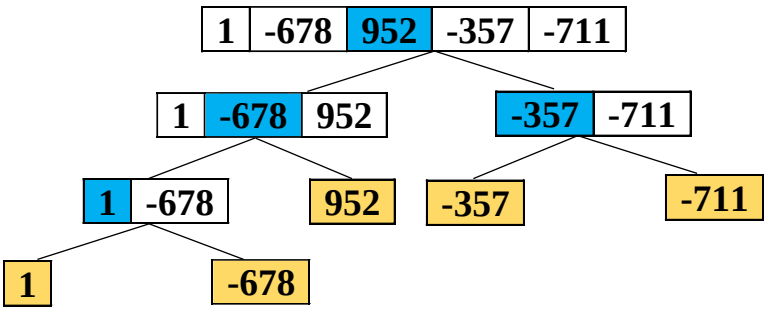


⇨ The array is sorted.

Example 10 - 2

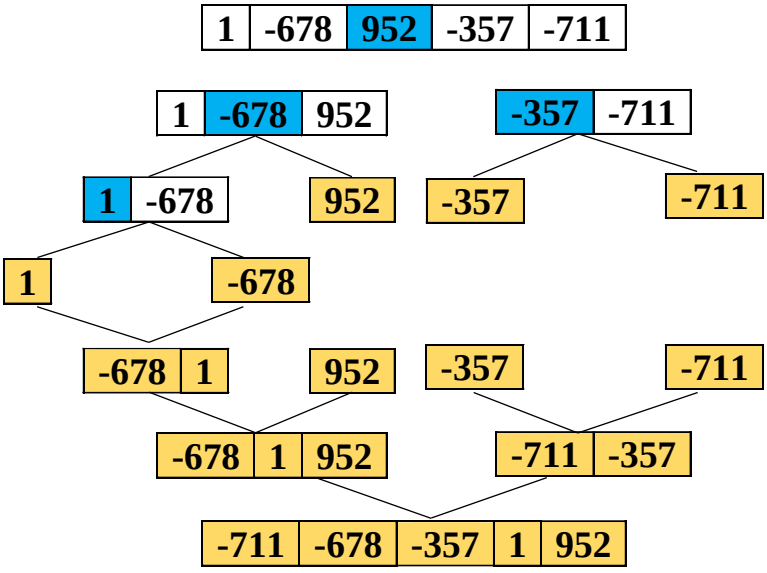
Array B = {**1, -678, 952, -357, -711**}.
Sort in ascending order

First, separate the array in half and iterate the same with its two subarrays until the subarray has only one element.



Second, merge all the subarray into a big array but in the ascending order by taking out one by one an element which is smaller than the element from the other subarray and put in the new merged array from those two.

The 2 subarrays are always in ascending order (due to the recursion, the 2 subarrays actually used to be another merged array). So you can traverse from left to right through them to find the appropriate element.



⇨ The array is sorted.

Example 10 - 3

11.Radix Sort.

- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
- Radix Sort is a non-comparative stable sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.
- Time complexity: O(n*k) (k is the length of the data)
- Advantages:
    - Fast when the keys are short or when the range of the array elements is less.
- Disadvantages:
    - Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. So, for every different type of data it needs to be rewritten.
    - It takes more space compared to *Quick Sort* which is inplace sorting.

- Example: Array A = {**84**, **65**, **148**, **1**, **49**, **26**}

Sort in ascending order

Start from the least significant digit of the elements, traverse and put all the elements into the bucket of digits respectively with their digits.

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **8**4. Put in the **4** section.

				84					
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **6**5. Put in the **5** section.

				84	65				
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **14**8. Put in the **8** section.

				84	65			148	
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **14**8. Put in the **8** section.

	1			84	65			148	
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **14**8. Put in the **8** section.

	1			84	65			148	49
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **14**8. Put in the **8** section.

	1			84	65	26		148	49
0	1	2	3	4	5	6	7	8	9

Sort in digits by taking values out from the lowest digit to highest, from the bottom to top of the bucket. Then, put it into the array from the beginning to the end.

	1			84	65	26		148	49
0	1	2	3	4	5	6	7	8	9

1					
0	1	2	3	4	5

⇨ Put **1** into index **0**.

				84	65	26		148	49
0	1	2	3	4	5	6	7	8	9

1	84				
0	1	2	3	4	5

⇨ Put **84** into index **1**.

					65	26		148	49
0	1	2	3	4	5	6	7	8	9

1	84	65			
0	1	2	3	4	5

⇨ Put **65** into index **2**.

						26		148	49
0	1	2	3	4	5	6	7	8	9

1	84	65	26		
0	1	2	3	4	5

⇨ Put **26** into index **3**.

								148	49
0	1	2	3	4	5	6	7	8	9

1	84	65	26	148	
0	1	2	3	4	5

⇨ Put **148** into index **4**.

									49
0	1	2	3	4	5	6	7	8	9

1	84	65	26	148	49
0	1	2	3	4	5

⇨ Put **148** into index **4**.

⇨ Stack is empty.

Example 11 - 1.1

Iterate with the preceding digits until all the element's digits is sorted.

1	84	65	26	148	49
0	1	2	3	4	5

Second digit (tens):

01	84	65	26	148	49
0	1	2	3	4	5

⇨ Put all values into the bucket by the tens digits of the elements.

				49					
1		26		148		65		84	
0	1	2	3	4	5	6	7	8	9

1	26	148	49	65	84
0	1	2	3	4	5

⇨ Put all values back into the array from the lowest digit to highest.

Third digit (hundreds):

001	026	148	049	065	084
0	1	2	3	4	5

⇨ Put all values into the bucket by the tens digits of the elements.

84									
65									
49									
26									
1	148								
0	1	2	3	4	5	6	7	8	9

1	26	49	65	84	148
0	1	2	3	4	5

⇨ Put all values back into the array from the lowest digit to highest.

1	26	49	65	84	148
0	1	2	3	4	5

⇨ The array is sorted.

Example 11 – 1.2

Sort in descending order

Start from the least significant digit of the elements, traverse and put all the elements into the bucket of digits respectively with their digits.

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **84**. Put in the **4** section.

				84					
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **65**. Put in the **5** section.

				84	65				
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **148**. Put in the **8** section.

				84	65			148	
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **148**. Put in the **8** section.

	1			84	65			148	
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **148**. Put in the **8** section.

	1			84	65			148	49
0	1	2	3	4	5	6	7	8	9

84	65	148	1	49	26
0	1	2	3	4	5

⇨ Current digit: **148**. Put in the **8** section.

	1			84	65	26		148	49
0	1	2	3	4	5	6	7	8	9

Sort in digits by taking values out from the highest digit to lowest, from the bottom to top of the bucket. Then, put it into the array from the beginning to the end.

	1			84	65	26		148	49
0	1	2	3	4	5	6	7	8	9

49					
0	1	2	3	4	5

⇨ Put **49** into index **0**.

	1			84	65	26		148	
0	1	2	3	4	5	6	7	8	9

49	148				
0	1	2	3	4	5

⇨ Put **148** into index **1**.

	1			84	65	26			
0	1	2	3	4	5	6	7	8	9

49	148	26			
0	1	2	3	4	5

⇨ Put **26** into index **2**.

	1			84	65				
0	1	2	3	4	5	6	7	8	9

49	148	26	65		
0	1	2	3	4	5

⇨ Put **65** into index **2**.

	1			84					
0	1	2	3	4	5	6	7	8	9

49	148	26	65	84	
0	1	2	3	4	5

⇨ Put **84** into index **2**.

	1								
0	1	2	3	4	5	6	7	8	9

49	148	26	65	84	1
0	1	2	3	4	5

⇨ Put **1** into index **2**.

⇨ Stack is empty.

Example 11 - 2.1

Iterate with the preceding digits until all the element's digits is sorted.

49	148	26	65	84	1
0	1	2	3	4	5

Second digit (tens):

49	148	26	65	84	01
0	1	2	3	4	5

⇨ Put all values into the bucket by the tens digits of the elements.

				148					
1		26		49		65		84	
0	1	2	3	4	5	6	7	8	9

84	65	49	148	26	1
0	1	2	3	4	5

⇨ Put all values back into the array from the highest digit to lowest.

Third digit (hundreds):

084	065	049	148	026	001
0	1	2	3	4	5

⇨ Put all values into the bucket by the tens digits of the elements.

1									
26									
49									
65									
84	148								
0	1	2	3	4	5	6	7	8	9

148	84	65	49	26	1
0	1	2	3	4	5

⇨ Put all values back into the array from the highest digit to lowest.

148	84	65	49	26	1
0	1	2	3	4	5

⇨ The array is sorted.

Example 11 – 2.2

Array B = {**1, 678, 952, 357, 711**}.

Sort in ascending order:

Iterate with the preceding digits until all the element's digits is sorted.

1	678	952	357	711
0	1	2	3	4

Second digit (tens):

01	678	952	357	711
0	1	2	3	4

⇨ Put all values into the bucket by the tens digits of the elements.

					357				
1		711			952		678		
0	1	2	3	4	5	6	7	8	9

1	711	952	357	678
0	1	2	3	4

⇨ Put all values back into the array from the lowest digit to highest.

Third digit (hundreds):

001	711	952	357	678
0	1	2	3	4

⇨ Put all values into the bucket by the tens digits of the elements.

1			357			678	711		952
0	1	2	3	4	5	6	7	8	9

1	357	678	711	952
0	1	2	3	4

⇨ Put all values back into the array from the lowest digit to highest.

1	357	678	711	952
0	1	2	3	4

⇨ The array is sorted.

Example 11 - 3

Topic 5: Tree

Lesson 1: Tree

Contents:

- Introduction
- Properties of Tree

I. Introduction:

* A tree <T> (Tree) is:

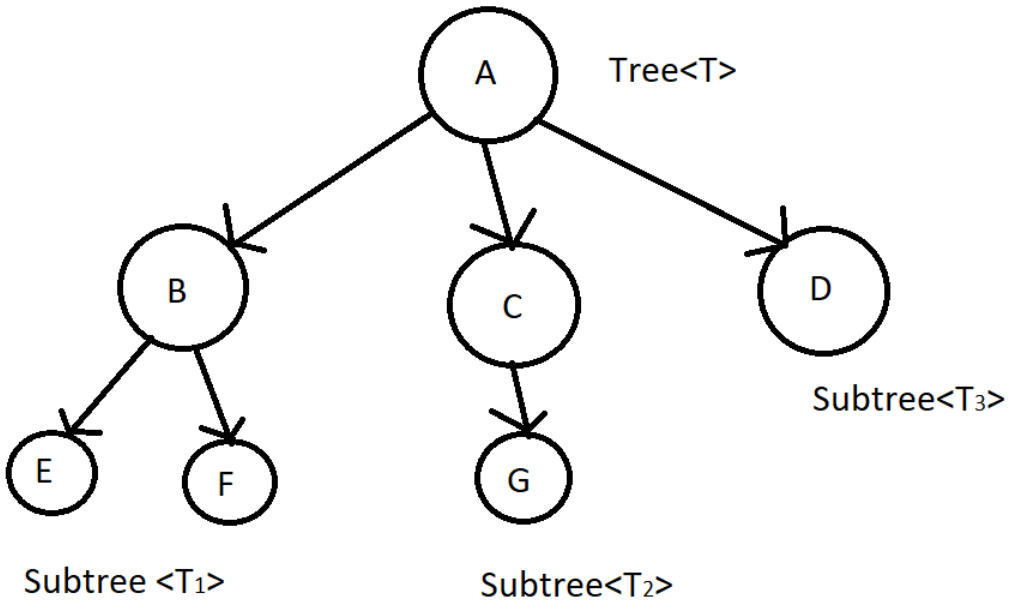
+A set of values of the same type, elements of the same value are stored by a node, called node p1, p2, ...

+ If a tree has no elements, it is called an empty tree.

+If the tree is not empty, there are some caveats:

* Each tree has only 1 root node. The root node is the starting node of the tree.

* The remaining nodes are divided into non-intersection sets (T_i ,$T_{i+1,...}$). Each <T_i> is called a subtree of <T> tree. Example:

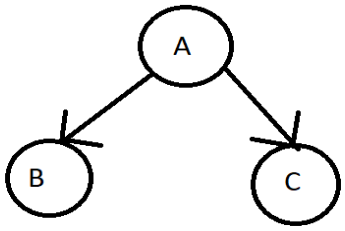


II. Properties of Tree:

+ The root node has no parent node.

+ Leaf node (external node):is a node that does not have any child nodes.

+ Parent node: is the node that we will be processed to do something before reaching another node that it can go to. Child node: It is the node that when the parent node has been processed, it comes to it. Example:



A is the node A is the node of B and C .B and C are child nodes of A.

+ Each node has only one node parent.

+ Each node can have multiple children.

+ Trees do not count cycles. That is, when a node 1 node goes to a child node, the child node cannot have any path in the tree to go against the node above. If possible, it is called a graph.

+ Node: is an element in tree. Node can contain any data. Each node can contain many types of data. For example, 1 node can contain 1 data as name, another data as age, ... And importantly, all nodes must be the same in terms of data but may differ in the values in the data that the node holds. That's why we say a tree is a data structure that collects elements of the same type.

+ Branch: link 2 nodes together.

+ Sibling nodes: is a node with the same parent node.

+ Degree of node: number of children of a node.

+ Internal node: is a node that has both a parent node and a child node.

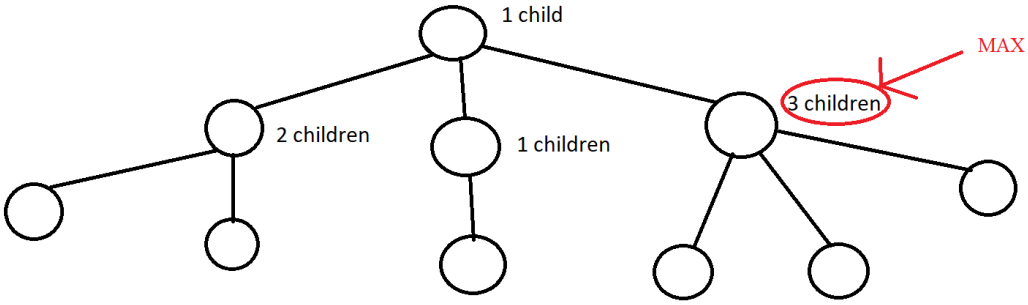
+ Subtree: is a subtree of the original tree. Also starts with a node.

+ Degree of tree is the maximum value of the degree of node in the tree.

When we interpret it with a formula, we can write:

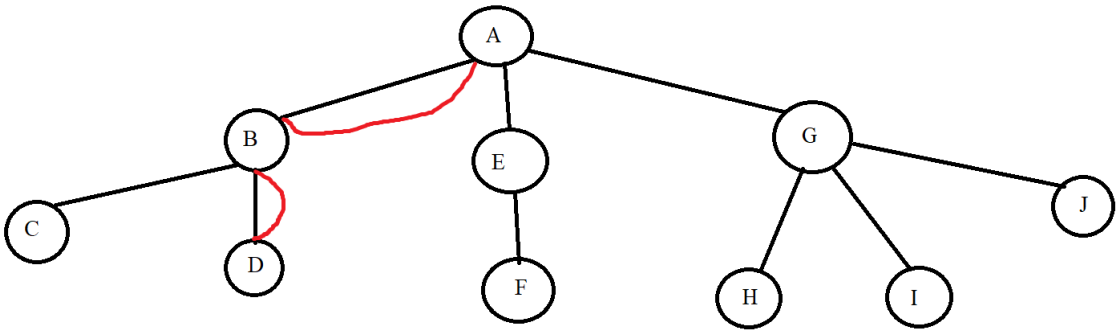
$$\text{Degree}(<T>) = \max\{\text{degree}(p_i) / p_i \in <T>\}$$

Example:



Degree of tree is 3

+ Path between node p_i to node p_j: is a series of nodes from pi to pj and on those nodes there must be branches to connect 2 nodes together. Example:



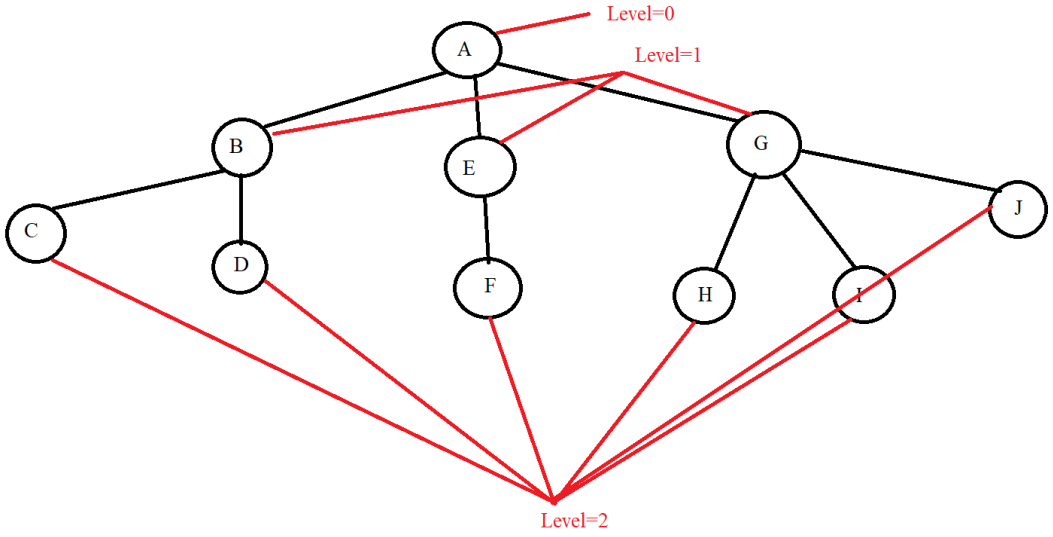
Path(A,D)=3

+ Level: Just like how to calculate the floor of a whole tower, we apply it to trees as well. Where the ground floor is the root with level =0 and the upper floors are considered the bottom nodes of a node, the level is calculated as the parent node's level plus 1. From this we can deduce the formula:

Level(p) = 0 if p = root

Level(p) = 1 + level(parent(p)) if p! = Root

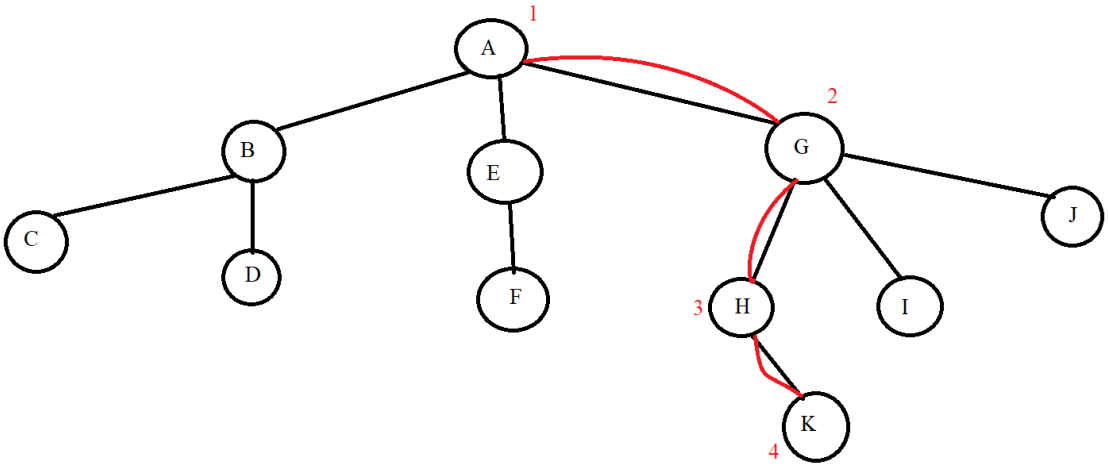
Example:



+ Height of tree (h_T): Maximum value of path from root to leaf node. We have the mathematical formula:

$$h_T = \max \{Path (root, p_i) \mid p_i \text{ is the leaf node} \in <T>\}$$

Example:



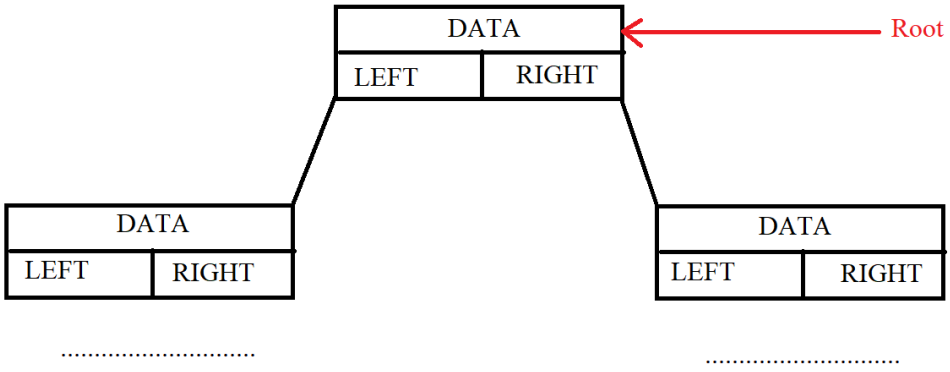
$H_T=4$

Lesson 2: Binary tree

Contents:

I. Introduction

- A tree <T> (Tree) is a binary tree when the degree of the tree is 2.

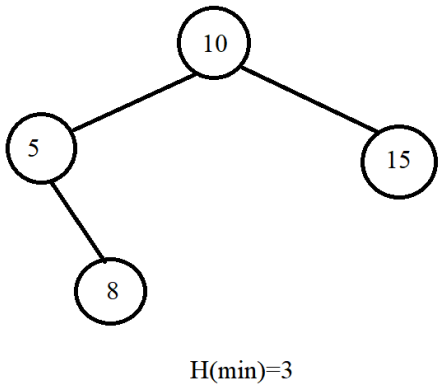


- The height of a binary tree has N nodes:

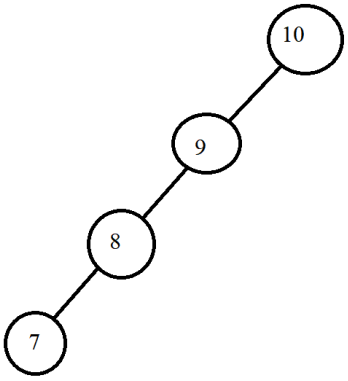
$$h_T(max) = N$$

$$h_T(min) = \log_2 N + 1$$

Example: with N=4



H(min)=3



H(max)=4

- There are 2 ways to organize a binary tree:

+ Array.

+ Structure pointers.

- Here it is best to use pointers because it is easier to use pointers to delete, insert and search. Pointers make it easier to know the child node of a node.

```
typedef struct tagBT_NODE
{
     int Data;//
     tagBT_NODE* pLeft;//pointer to hold the left node
     tagBT_NODE* pRight;//pointer to hold the right node
} BT_NODE;

typedef struct BIN_TREE
{
     int Count;//Number of nodes in the tree
     BT_NODE* pRoot;//The pointer holds the root node of the
tree
};
```

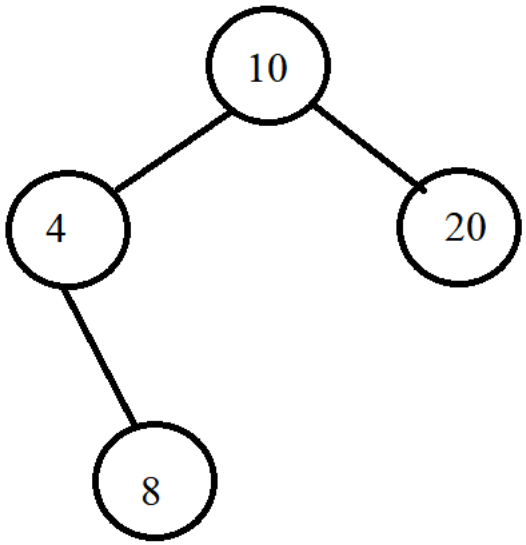
II. Operations on Binary Tree

- Traverse in Tree: We have many ways of traversing such as NLR, LRN, RNL, ... but we only consider 3 basic traversals: NLR, LRN and LRN.

+ Pre-Order (NLR): It will process the current node and then recursively call the left node and finally the right node. Put the case we want in the buttons in we have the following examples

```
void NLR(BT_NODE* pCurr)
{
     if (pCurr != NULL)
     {
          cout << pCurr->Data << " ";
          NLR(pCurr->pLeft);
          NLR(pCurr->pRight);
     }
}
```

Example:

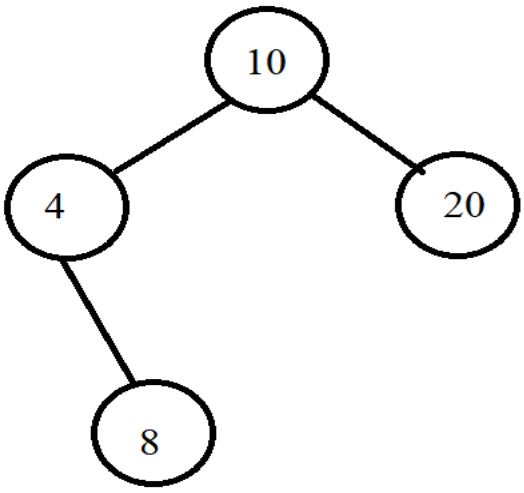


=> 10 , 4 , 8 , 20

+ In-Order (LNR): It will recursively call the left node. Then it will process the current node and finally recursively call the right node.

```
void LNR(BT_NODE* pCurr)
{
     if (pCurr != NULL)
     {
         LNR(pCurr->pLeft);
          cout << pCurr->Data << " ";
          LNR(pCurr->pRight);
     }
}
```

Example:

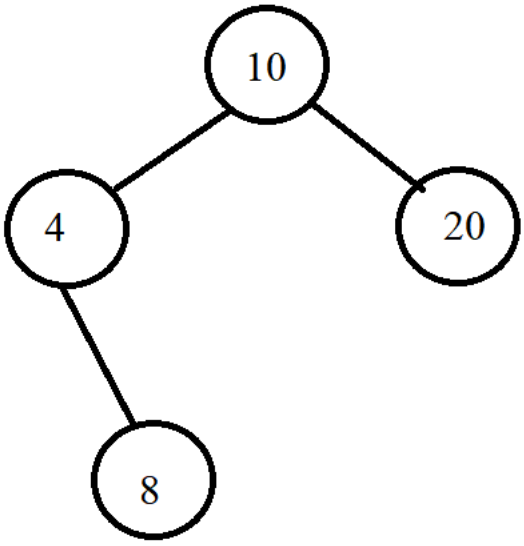


=> 4 , 8 , 10 , 20

+ Post-Order (LRN): We will recursively call the left node, go to the right node, and finally process the current node.

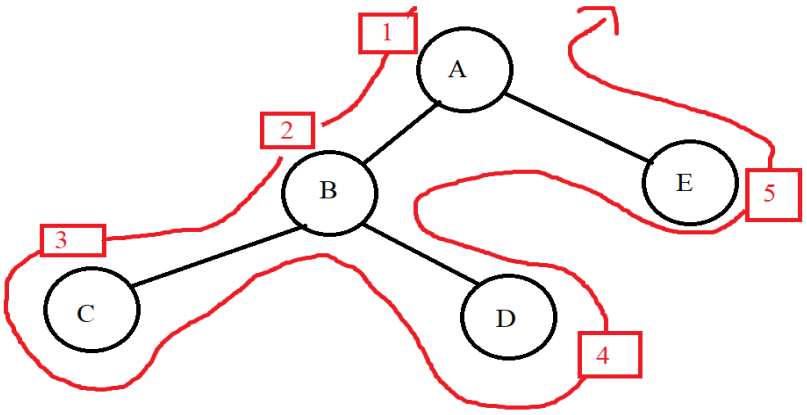
```
void LRN(BT_NODE* pCurr)
{
     if (pCurr != NULL)
     {
          LRN(pCurr->pLeft);
          LRN(pCurr->pRight);
        cout << pCurr->Data << " ";
     }
}
```

Example:

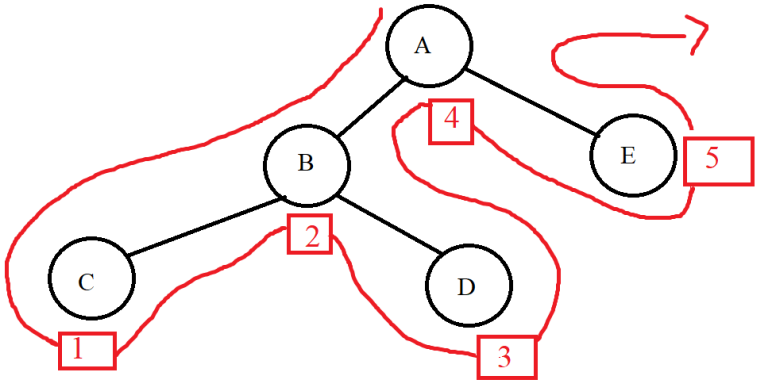


=> 8 , 4 , 20 , 10

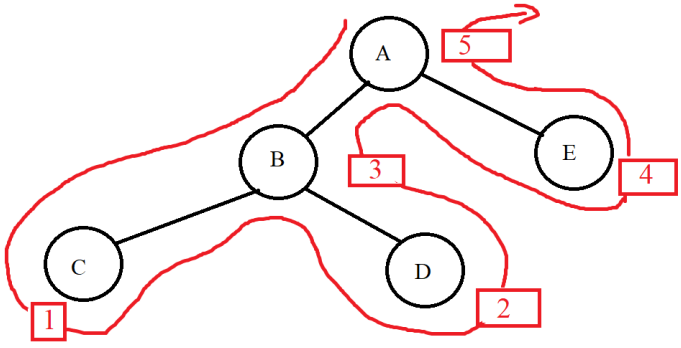
*Note: In addition to printing nodes like the examples above, we can change the place (cout<< pCurr->Data << " ") to whatever command we want to handle the query, the subject we want. Below is an illustration of the traversal of the three methods above.



NLR



LNR



LRN

Lesson 3: Binary Search Tree (BST)

Contents:

I. Introduction

- The binary search tree is:

+ A binary tree: That is, each node has at most 2 child nodes.

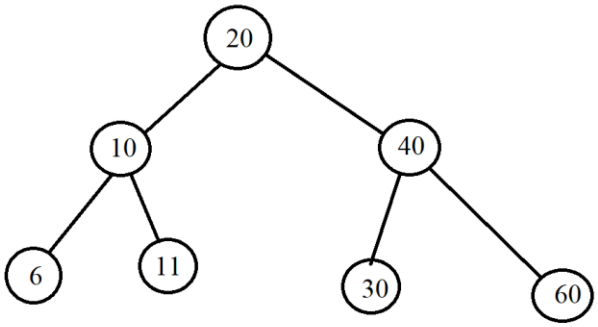
+ Each value node exists only once in the tree.

+ Each node must satisfy the condition:

- All nodes of the left subtree must be less than the current node.
- All nodes of the right subtree must be less than the current node.
- Assuming the current node is node (p), then we have:

∀L ∈ p-> pLeft: L-> Data <p-> Data
∀R ∈ p-> pRight: R-> Data> p-> Data

Example:



II. Operations on Binary Search Tree:

+ Create an empty tree.

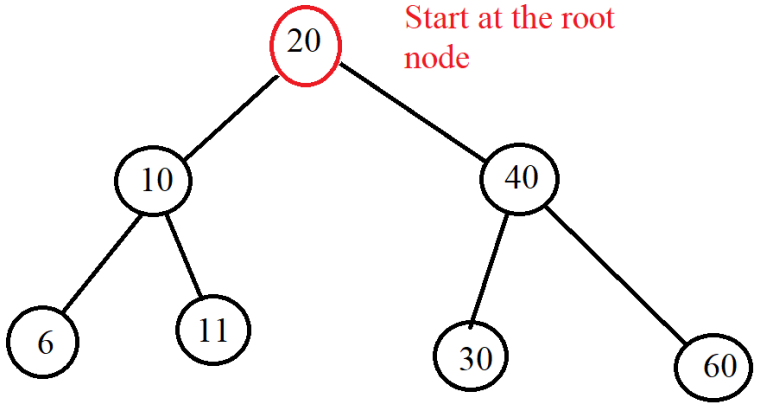
```
void BSTCreate(BIN_TREE& t)
{
     t.Count = 0;
     t.pRoot = NULL;
}
```

+ Check the empty tree.

```
int BSTIsEmpty(BIN_TREE t)
{
     if (t.pRoot == NULL)
          return 1;
     return 0;
}
```

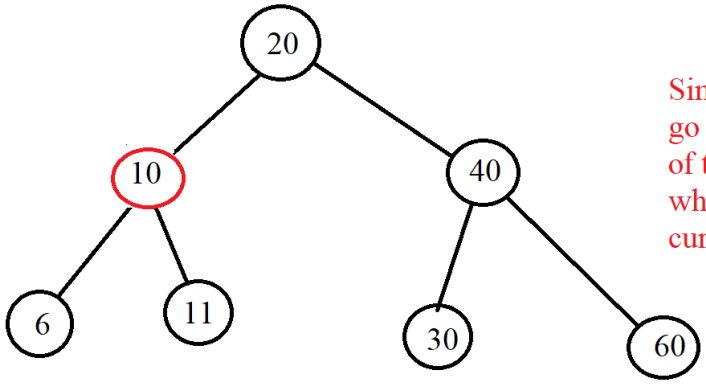
+ Find an element: Using the same condition for each node in the BST that the left child is less than the current node and the right node is greater than the current node, we will apply it to finding the element in the BST tree. Accordingly, if the current node that the tree is traversing has data larger than the element we are looking for, we will recursively go to the left node and vice versa, we will recursively go to the right. If we find the value we want to find, we stop and if we can't find it, the node is Null. Example:

FIND 11



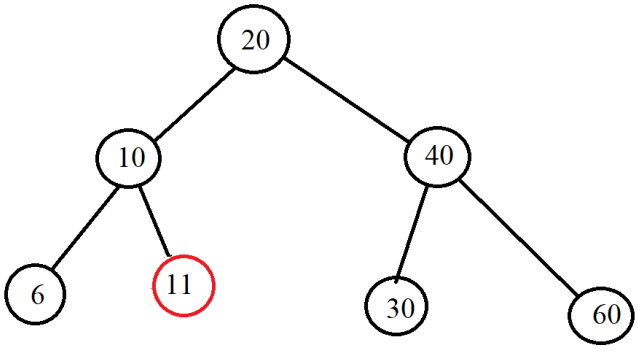
Start at the root node

FIND 11



Since 11 < 20, we go to the left node of the current node which is 10 of the current 20 node

FIND 11

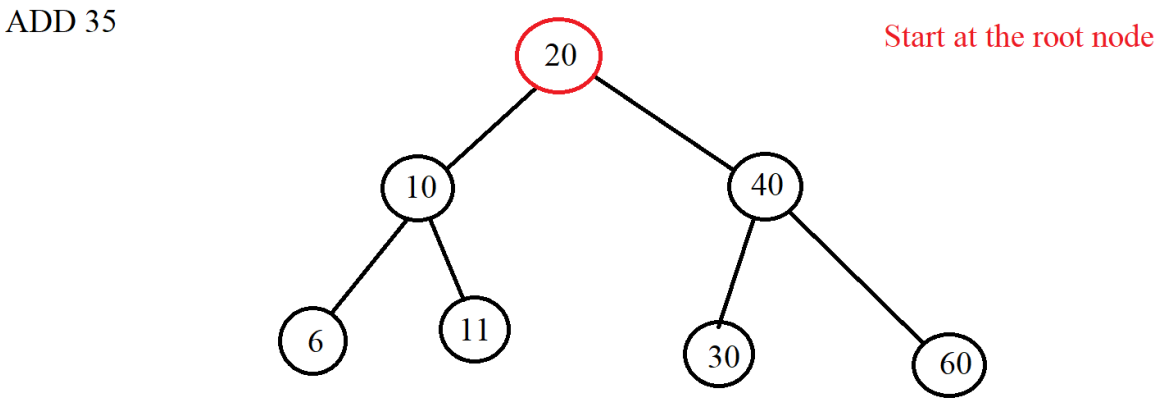


Since 11 > 10, we go to the right node of the current node which is 11 of the current node 10. Here, we see the current node value is 11 => Right number is looking, so we stop the search.

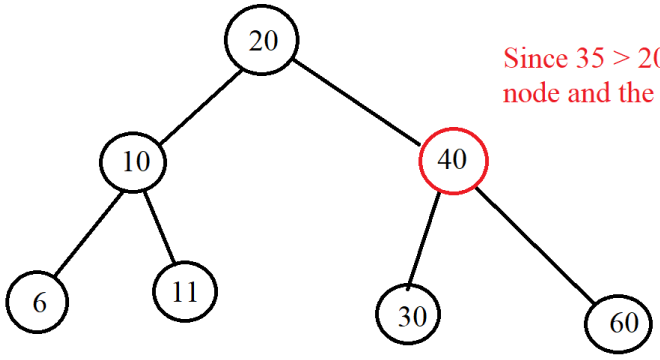
```
BT_NODE* BSTSearch(BT_NODE* pCurr, int Key)
{
     if (pCurr == NULL) return NULL;
     if (pCurr->Data == Key) return pCurr;
     else if (pCurr->Data > Key)

          return BSTSearch(pCurr->pLeft, Key);

     else
          return BSTSearch(pCurr->pRight, Key);
}
```

+ Add 1 element: Just like the search is based on conditions, we will start at the root node and will traverse left if the inserted element is greater than the current node and vice versa. We will do this until the current node is a NULL node and finally, we replace the NULL node with the node that we initialize with the value we want to insert.

Example:

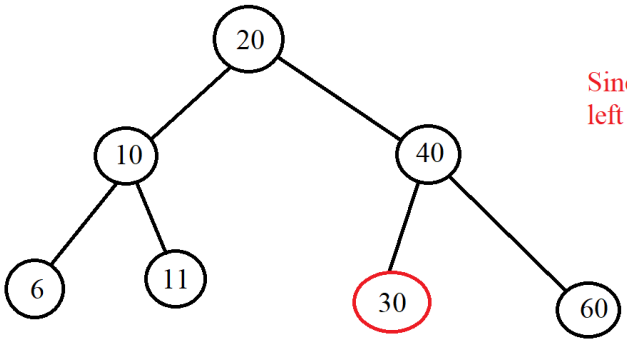


ADD 35



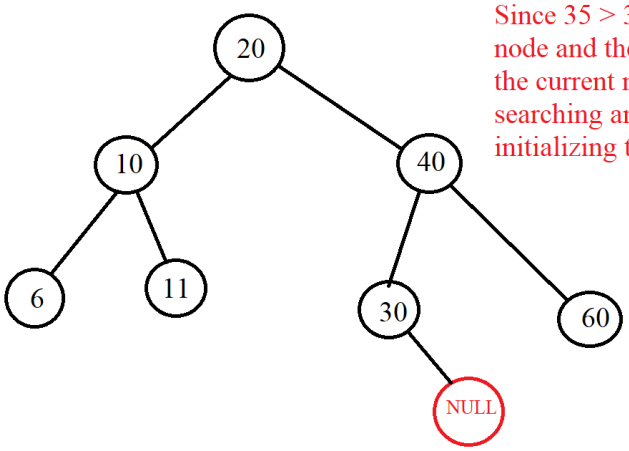
Since 35 > 20, we browse to the right node and the current node is 40

ADD 35



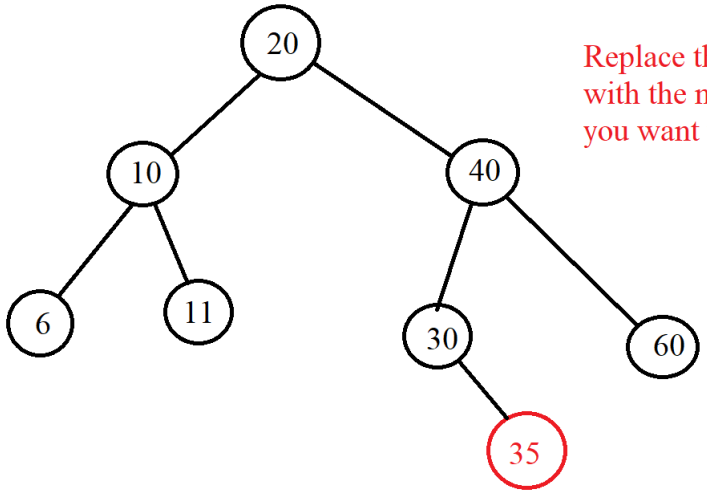
Since 35 < 40, we browse to the left node and the current node is 30

ADD 35



Since 35 > 30, we browse to the right node and the current node is NULL. Since the current node is NULL, we will stop searching and replace the NULL node by initializing the node we want to insert

ADD 35



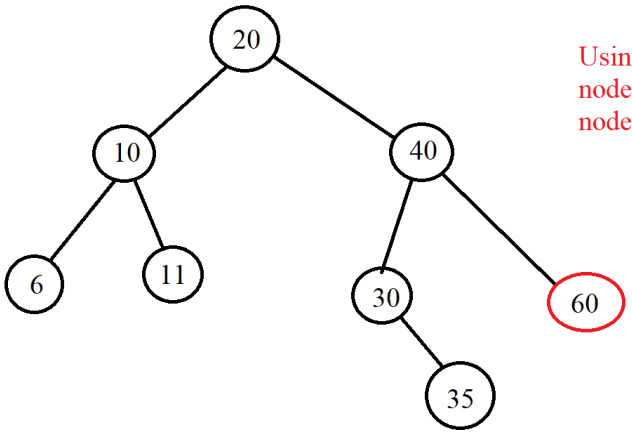
Replace the NULL node with the node whose value you want to insert is 35

```
int BSTInsert(BT_NODE*& pCurr, int newKey)
{
     if (pCurr == NULL) {
          pCurr = new BT_NODE;
          pCurr->Data = newKey;
          pCurr->pLeft = pCurr->pRight = NULL;
          return 1;
     }
     if (pCurr->Data > newKey)
          return BSTInsert(pCurr->pLeft, newKey);
     else if (pCurr->Data < newKey)
          return BSTInsert(pCurr->pRight, newKey);
     else return 0;
}
```

+ Delete 1 element: We use the above tree search algorithm to find the element we want to delete. If you can't find it, it means there is no element in the tree. And if there is, then we divide into 3 cases to delete.

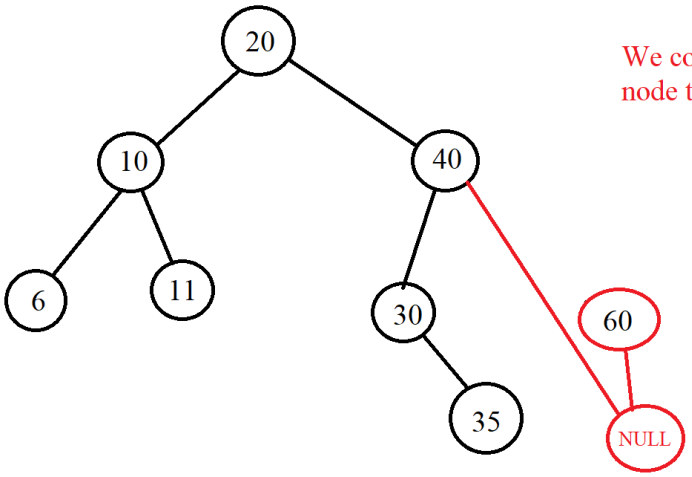
Case 1: Delete node without any child node => We just need to delete that node and let the parent node of the node to be deleted connect to the NULL node. Example:

DELETE 60



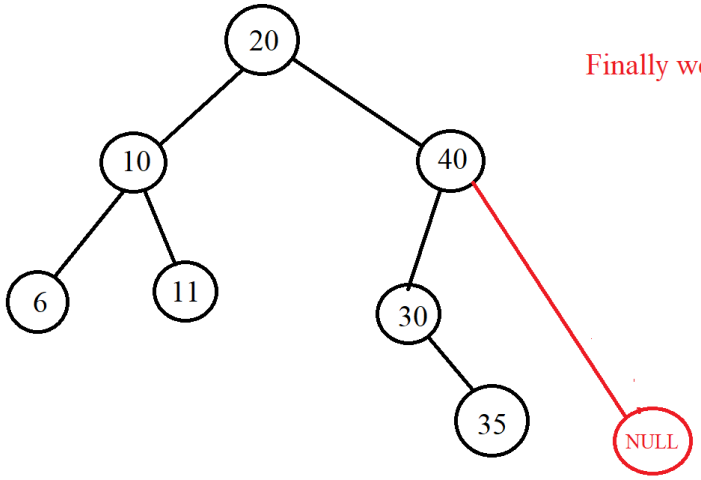
Using algorithm to search 1 node in the tree to find the node to delete is node 60.

DELETE 60



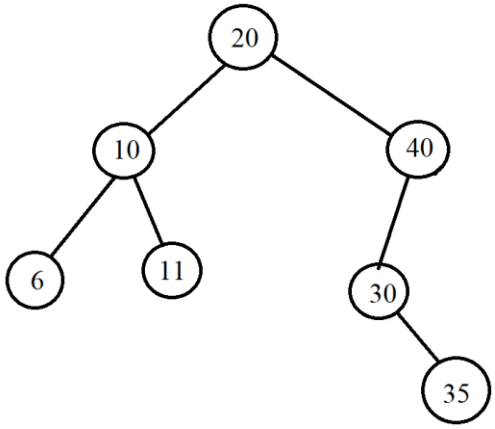
We connect the parent node to the NULL node

DELETE 60



Finally we delete node 60

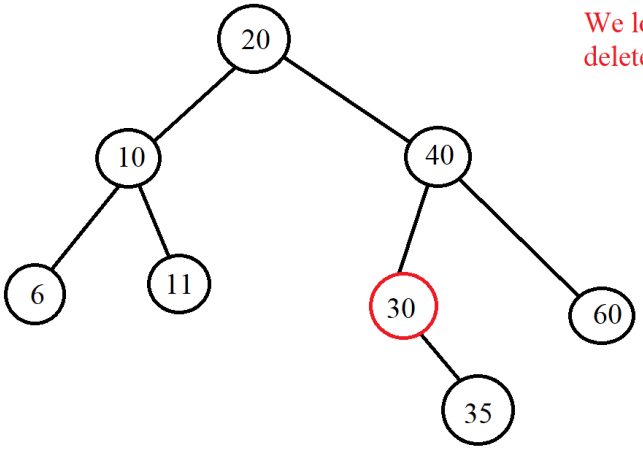
DELETE 60



Since the node is NULL, we treat it as an empty node, so when we present it, we don't need to draw it.

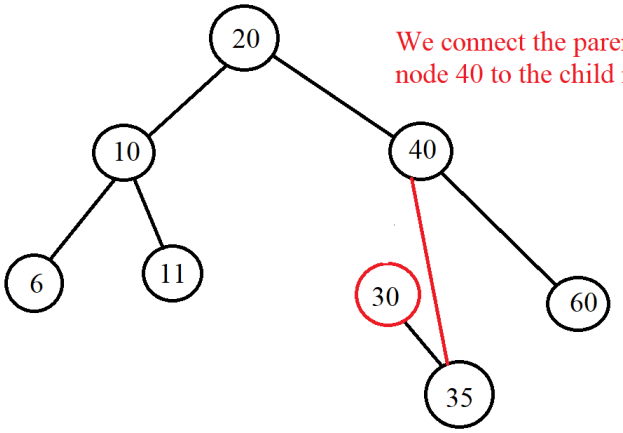
Case 2: Delete node with 1 child node => When we want to delete the element to be deleted in this case, we must connect the parent node of the node to be deleted with the child node of that node because that way the new tree will not be broken. When the connection is complete, we just need to delete it. When connecting the parent node of the node to be deleted with its child, we need to pay attention that if the left child node of that node is a NULL node, we will connect to the right child node and vice versa. In fact, this case is similar to case 1, but only the difference is that a child node is NULL and here is a valid node. Example we delete node 30 with 35 as the right child:

DELETE 30



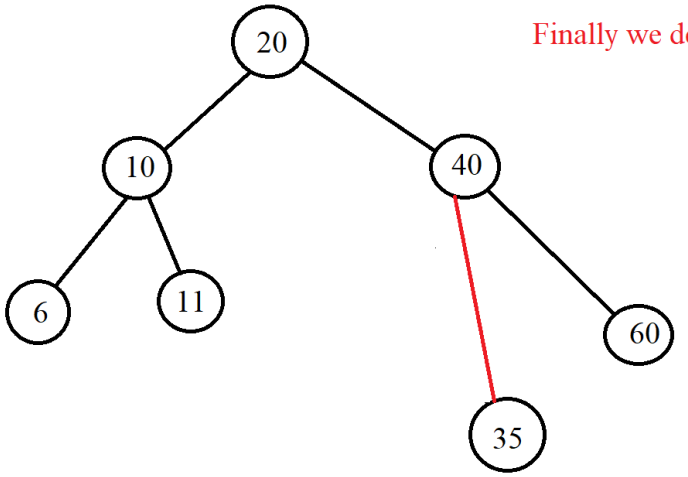
We look for the node to be deleted is node 30.

DELETE 30



We connect the parent node of node 30 which is node 40 to the child node of node 30 which is 35

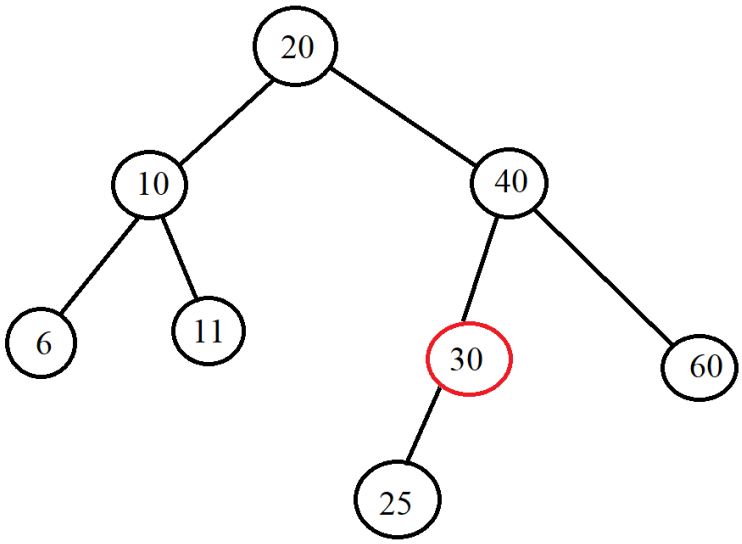
DELETE 30



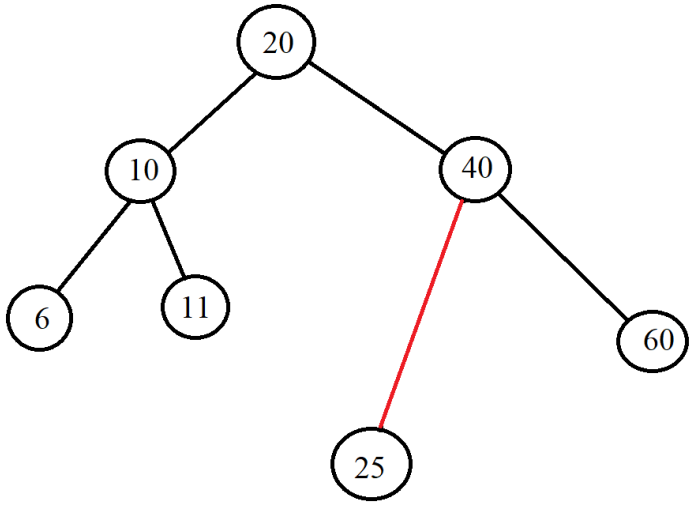
Finally we delete node 30

Similarly, when we delete 30 and 25 are left child nodes:

DELETE 30



DELETE 30

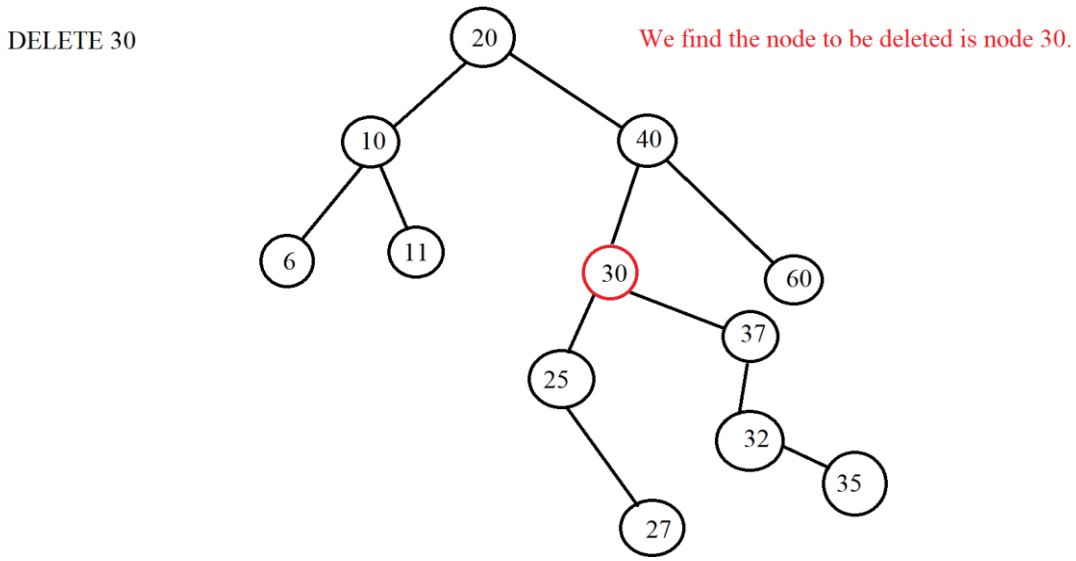


Case 3: Delete node with 2 children => Instead of deleting that node, we delete the node for which we get the value to replace the node that needs to be deleted. Then the node that we want to delete will be replaced with another value. Therefore, the node so that we can substitute the value for the node we want to delete is the node instead of the node we want to delete must satisfy the condition of the BST tree. From there, we have 2 ways to do it:

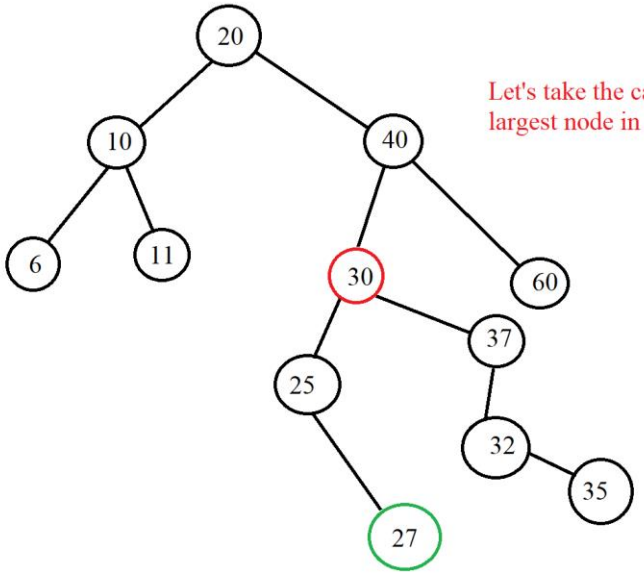
*Node 1: We will choose the node with the largest value of the left subtree of the replaced node.

*Node 2: We choose the smallest node of the right subtree.

When the replacement is complete, we delete the node that has been taken to replace the node we want to delete. Here is the largest node of the left subtree or the smallest node of the right subtree. If the replacement node is deleted, it will fall into one of the two cases 1 or 2 above. Example:

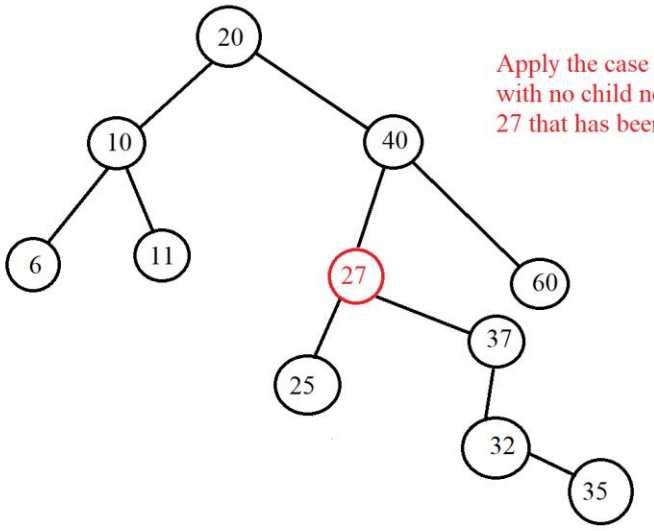


DELETE 30



Let's take the case that we choose the largest node in the left subtree.

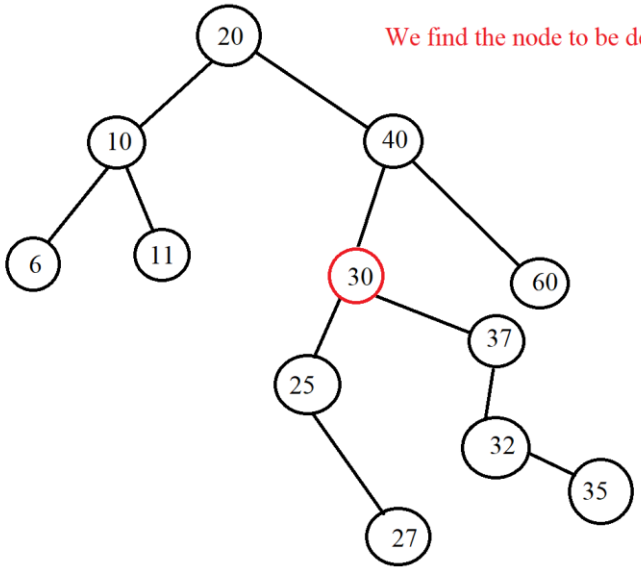
DELETE 30



Apply the case of deleting a node with no child nodes to delete node 27 that has been replaced.

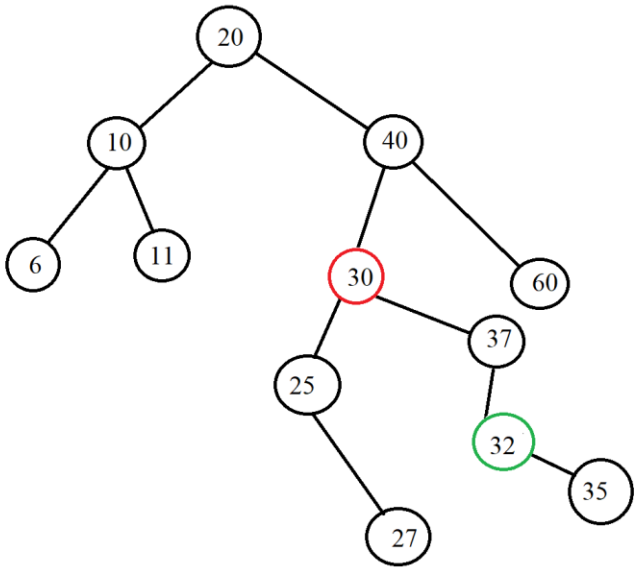
(Let's say we want to get the smallest node of the right subtree instead)

DELETE 30

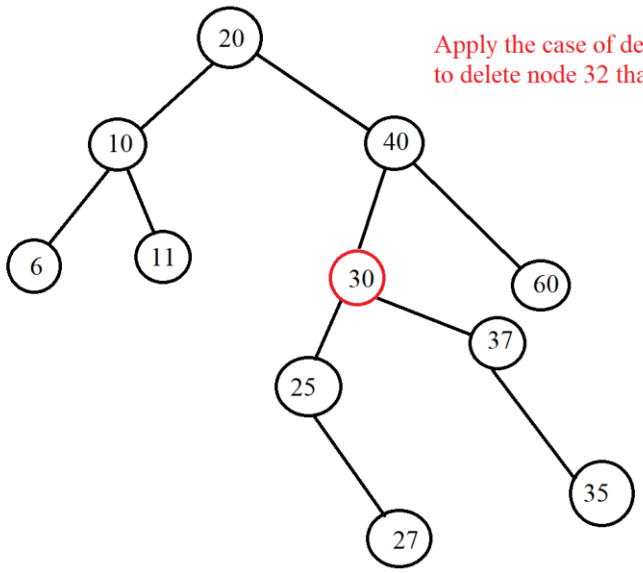


We find the node to be deleted is node 30.

DELETE 30



DELETE 30



Apply the case of deleting a node with 1 child to delete node 32 that has been replaced

```
// The function that performs the delete operation
void _Delete(BT_NODE*& pCurr)
{
     BT_NODE* pTemp = pCurr;
     // node has only left child node
     if (pCurr->pRight == NULL)
          pCurr = pCurr->pLeft;
     // the case of a node with a right child node or a
node without a child node
     else if (pCurr->pLeft == NULL)
          pCurr = pCurr->pRight;
     // node has 2 children
     else
          //_SearchStandFor(pCurr->pRight, pCurr) if we
want to replace it with the smallest node on the right
          pTemp = _SearchStandFor(pCurr->pLeft, pCurr);

     //Delete node
     delete pTemp;
}
```

```
//Verification function to find the node to delete
int BSTDelete(BT_NODE*& pCurr, int Key)
{
     if (pCurr == NULL) return 0;
     if (pCurr->Data > Key)
          return BSTDelete(pCurr->pLeft, Key);
     else if (pCurr->Data < Key)
          return BSTDelete(pCurr->pRight, Key);

     _Delete(pCurr);
     return 1;
}
```

III. Illustration examples

Example 1:

- **INPUT:**

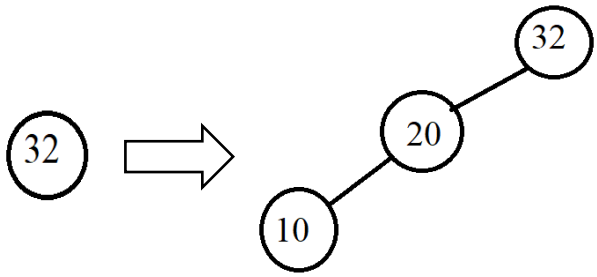
  -Insert:32, 20, 10, 15, 40, 25, 27, 26, 5, 13, 14.

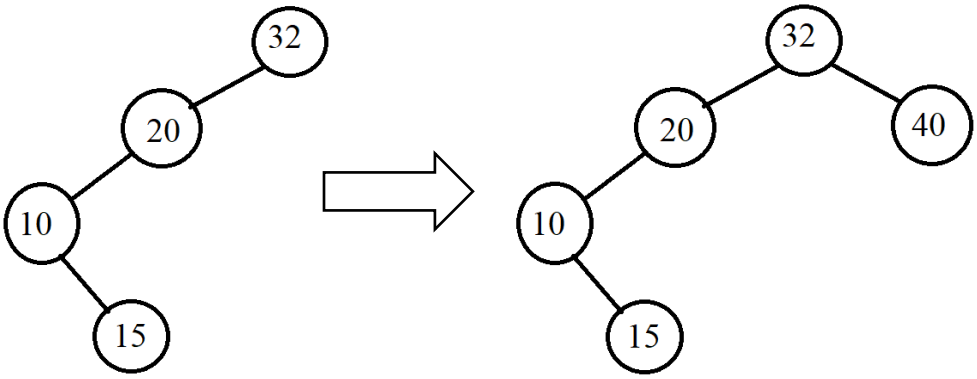
  -Find: 15, 29

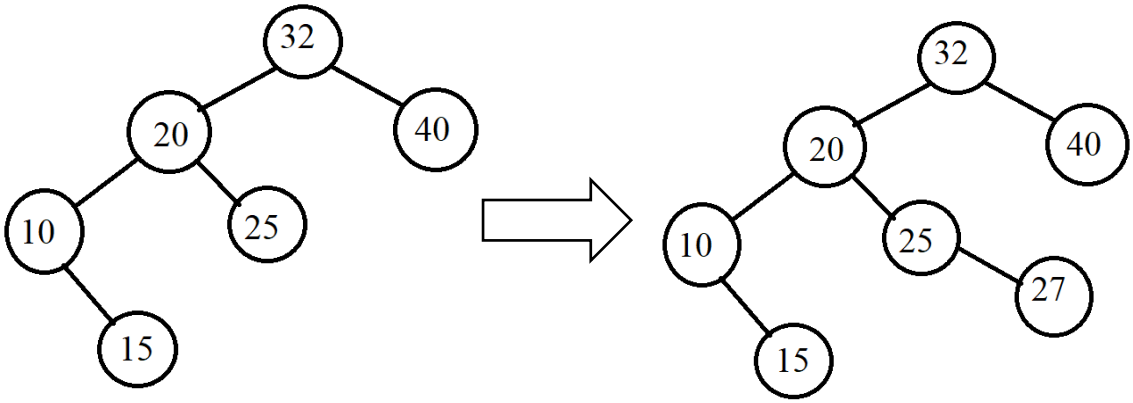
  -Delete: 25, 40, 20

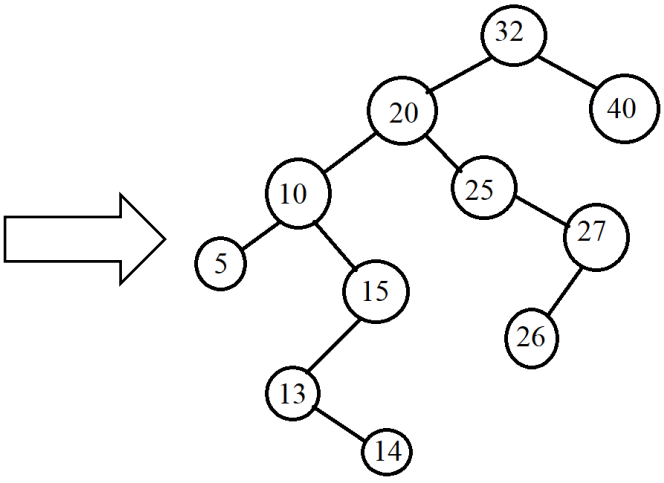
- **OUTPUT:**

- Insert:



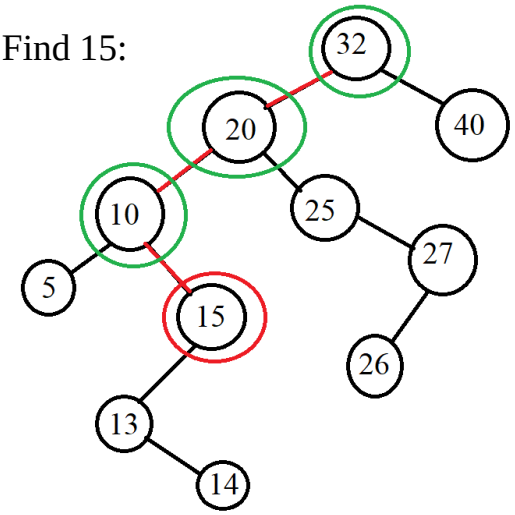

32
20
10
15
32
20
40
10
15


32
20
40
10
25
15
32
20
40
10
25
27
15

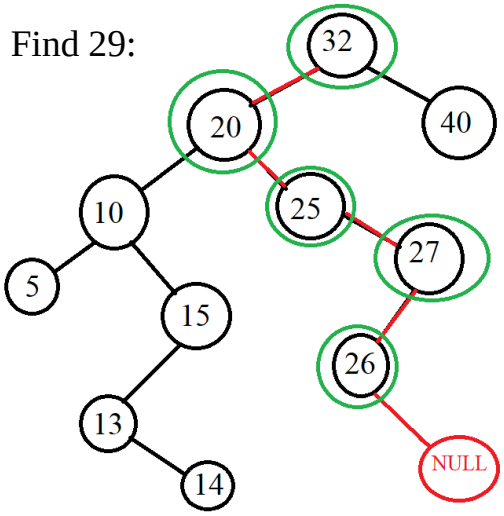

32
20
40
10
25
27
5
15
26
13
14

- Find:

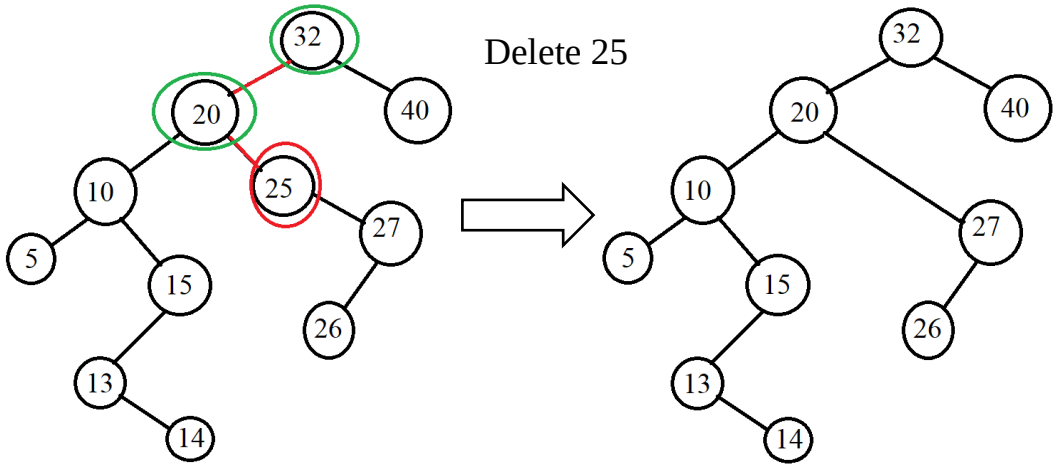
Find 15:

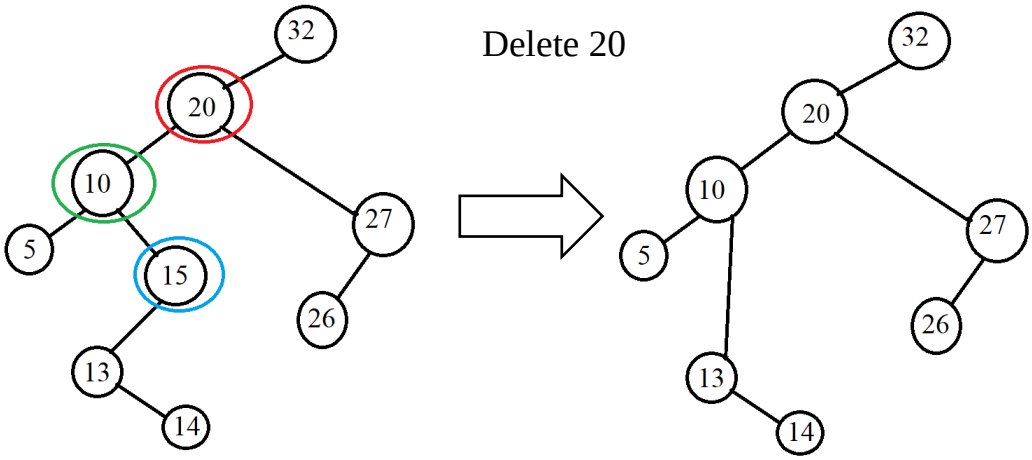


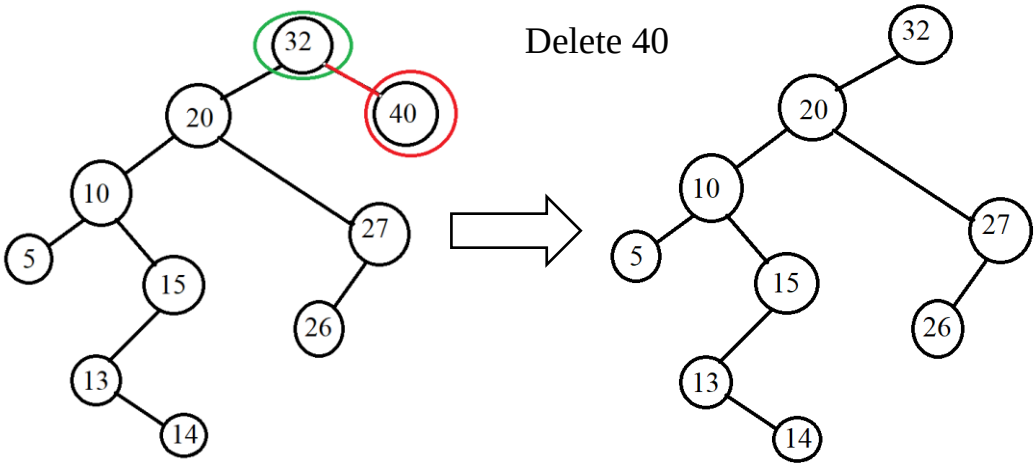
Find 29:



- Delete:


Delete 25
32
20
40
10
25
27
5
15
26
13
14


Delete 20
32
20
10
27
5
15
26
13
14


Delete 40
32
20
40
10
27
5
15
26
13
14

Example 2:

- **INPUT:**

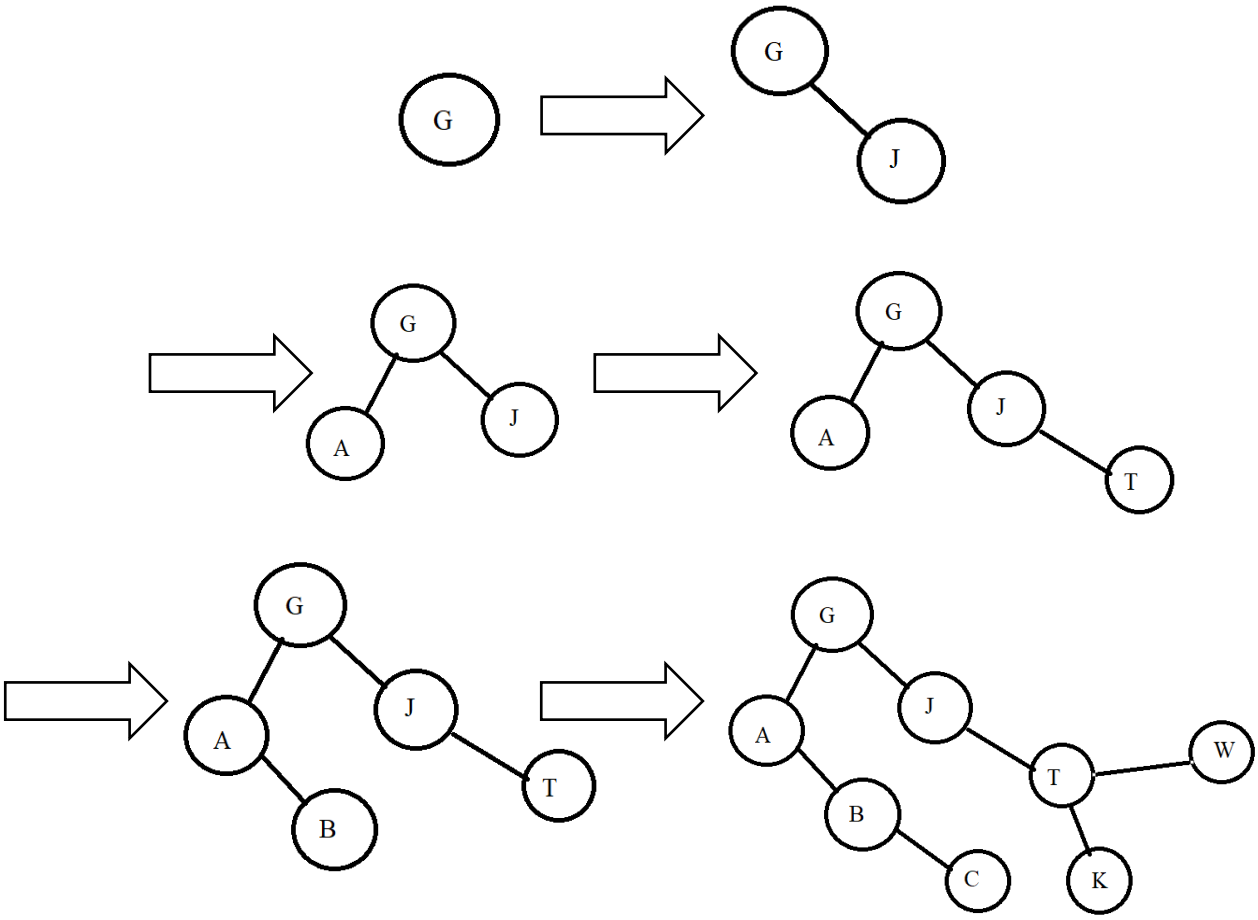
-Insert: G, J, A, T, B, W, C, K.

-Find: A, Z

-Delete: J, A

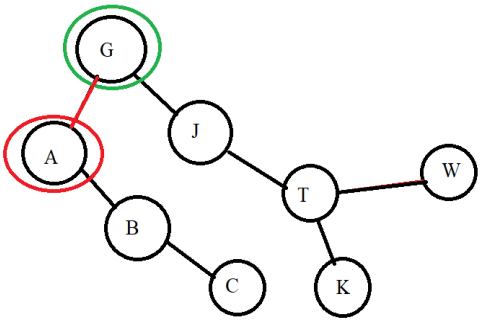
- **OUTPUT:**

-Insert:

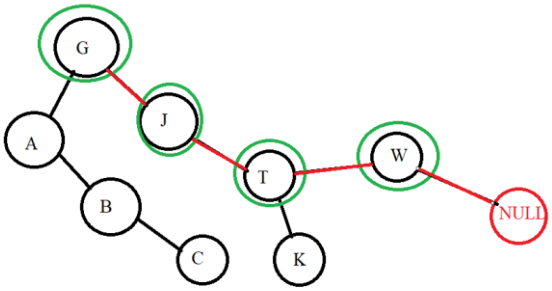


- Find:

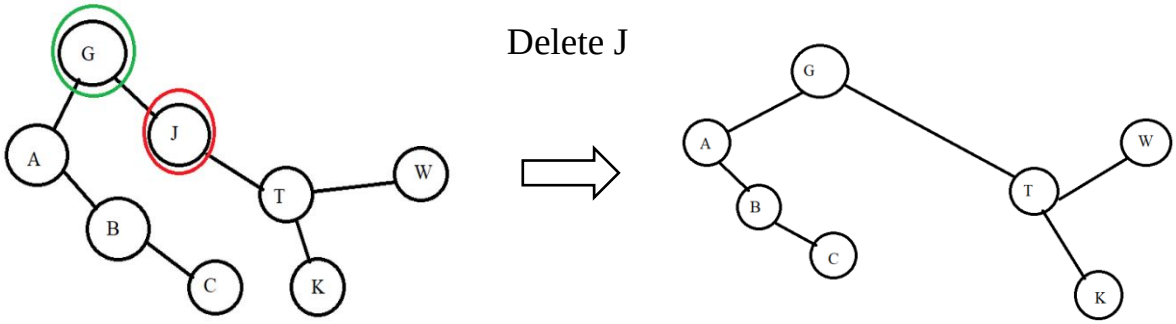
Find A

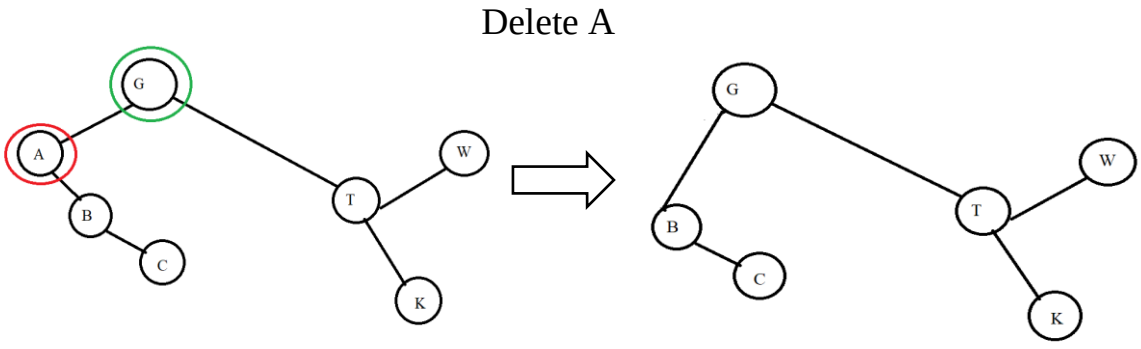


Find Z



- Delete:





Example 3:

- **INPUT:** input is a data pair of the form (x, y) where x is an integer and y are a letter. We will compare the number first if the number is equal, we compare the letter.

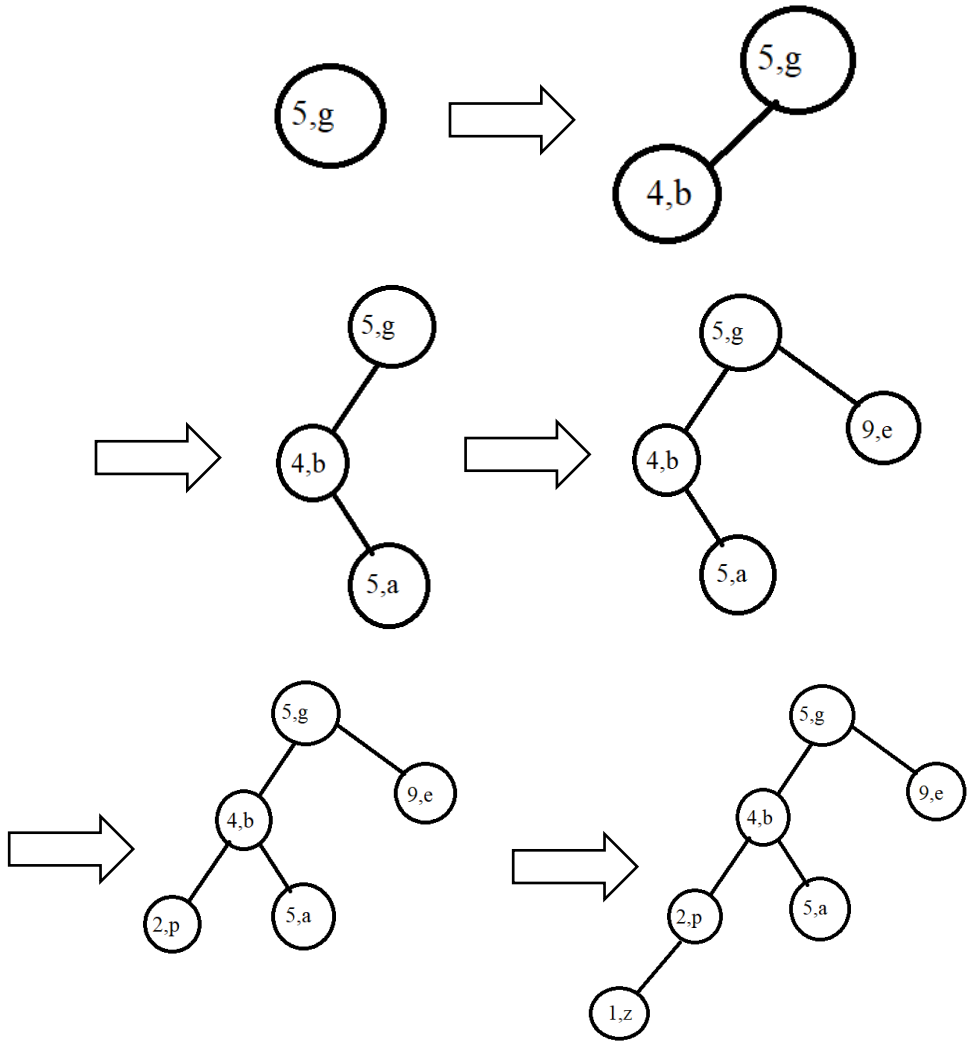
- Insert: (5, g), (4, b), (5, a), (9, e), (2, p), (1, z)

- Find: (9, e), (7, a)

- Delete: (5, g), (2, p)

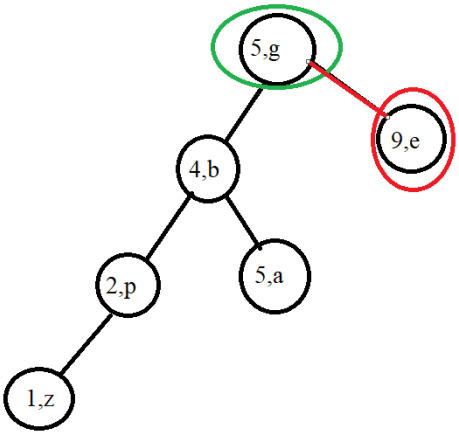
- **OUTPUT:**

- Insert:

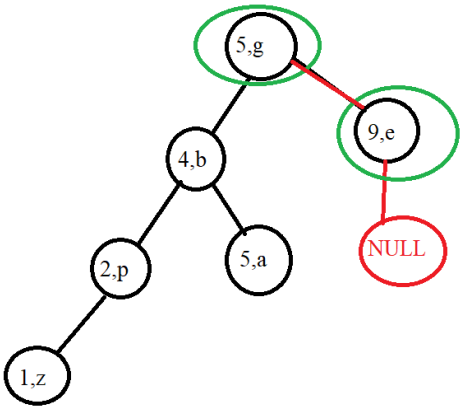


- Find:

Find (9, e)

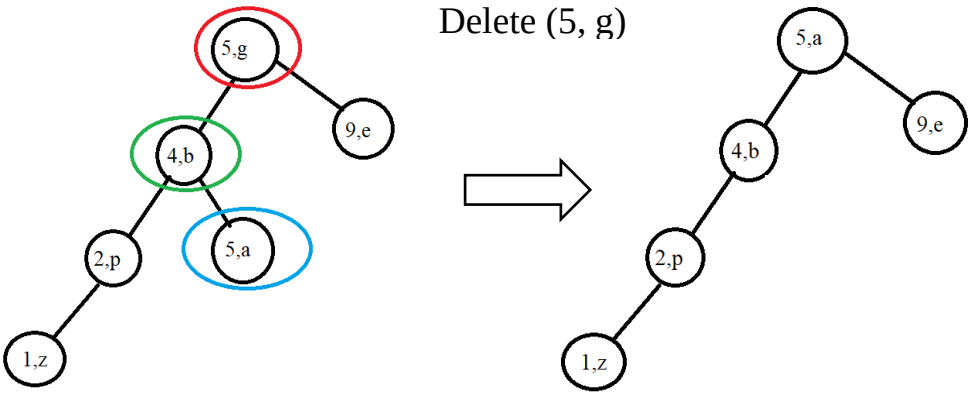


Find (7, a)

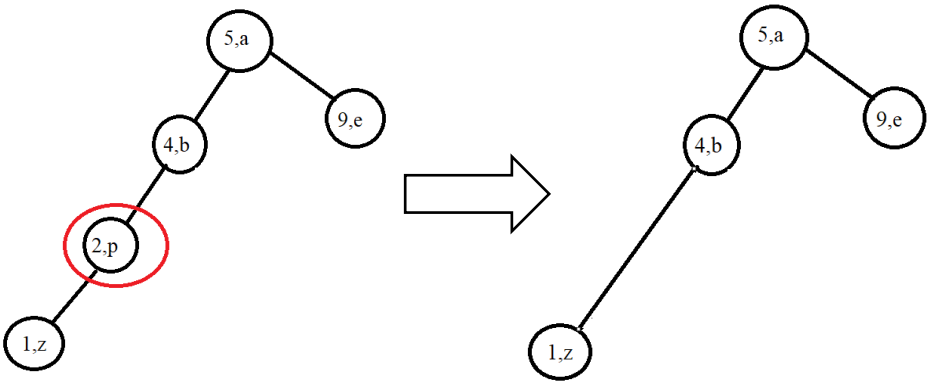


- Delete:

Delete (5, g)



Delete (2, p))



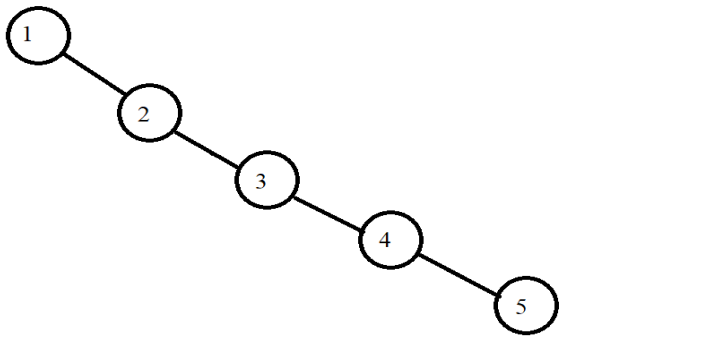
Lesson 4: Balanced Tree (AVL)

Contents:

I. Introduction

***** Question:

- If we keep inserting elements into the BST tree in ascending order of 1,2,3, 4… or descending, for example, 1000, 999,… then when we insert millions of numbers or the whole input ratio in that order, we will create a tree that leans left or right. Example:



.....

* The main content of the balanced tree:

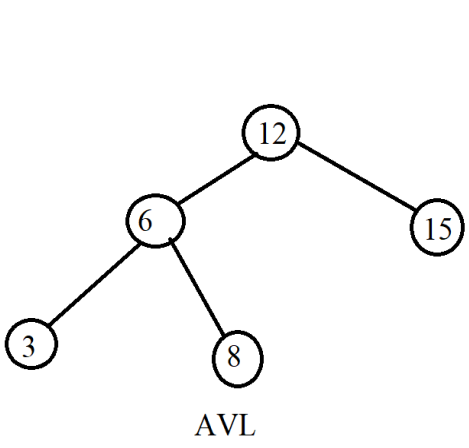
- The AVL is:

+ A search binary tree: That is, it also has only 2 children in each node, where the left child node is smaller than the parent node and the right child node is larger than the parent node.

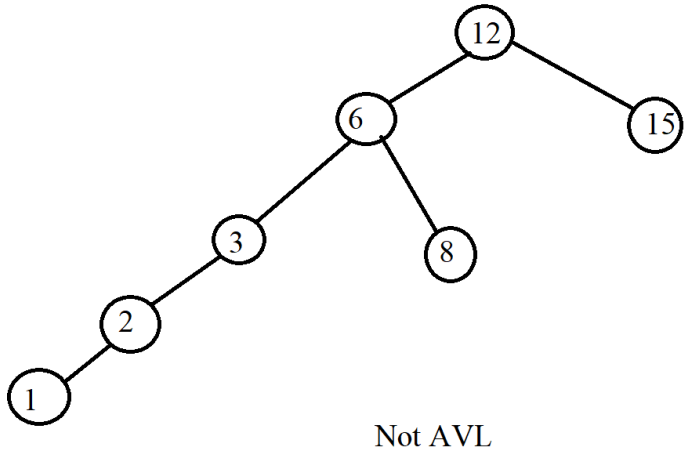
+ The key to a balanced tree is: At each node, the difference between left and right subtrees is less than or equal to 1. Note here that the concept of equilibrium is not that the left and right subtrees are equal, but that their difference is less than or equal to 1 because in the case of two equal left and right subtrees, it is too good to achieve. I have the formula:

$$\forall p \in T_{AVL} : (h_p -> Left - h_p -> Right) \leq 1$$

Example:



AVL



Not AVL

II. Operations on AVL Tree

How to declare in a node:

```
struct Node
{
    int key;
    Node* left;
    Node* right;
    int height;
};
```

- Balance: We call getBalance a function that tells us whether it is unbalanced or unbalanced. If it is out of balance, it will tell us which side it is unbalanced.

+ If getBalance = 1: Right child node is more than left child node.

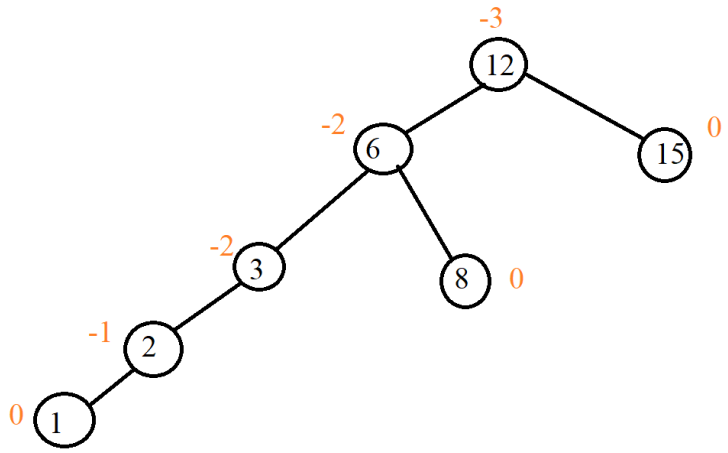
+ If getBalance = -1: Left child node is more than right child node

+ The rest is in the case of a balanced tree.

```
// A utility function to get height
// of the tree
int height(Node* N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
```

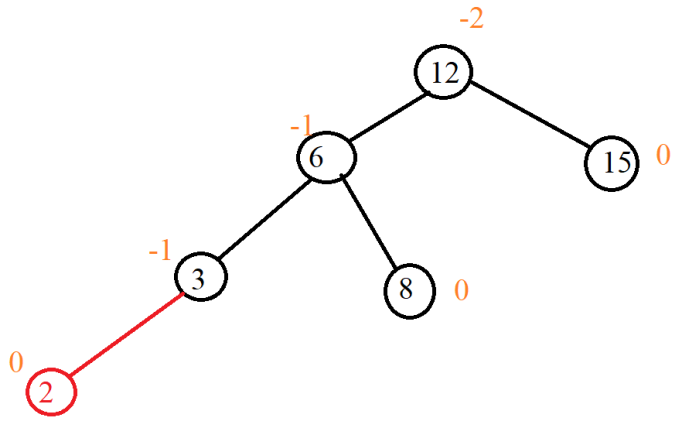
```
// Get Balance factor of node N
int getBalance(Node* N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

Example:

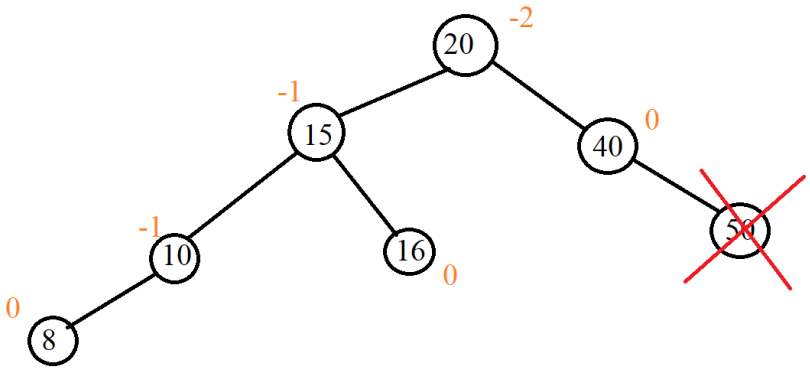


- Adjust the tree to balance: There are two operations on the tree that make the tree unbalanced: insertion and deletion because then we have changed the state of the tree. In the case of inserting an element into the tree, we have the ability to make the parent node of the node we just inserted unbalanced or the nodes on it also fall into the unbalanced case and in the case of deletion, it will too.

Example:



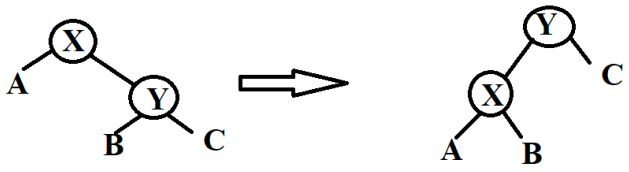
In case when adding node 2, it causes node -2 to be left unbalanced



When we delete node 50, it will make node 20 unbalanced to the left

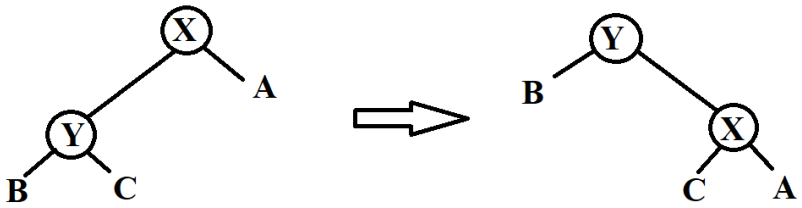
So, the question is, what should we do when we fall into that situation? The answer is that we use tree correction.

- Before going into editing the tree, we must see how to rotate 2 nodes because when editing the tree, we rotate the node for it to become an AVL tree.



This is the case of left rotation at node X

```
Node* leftRotate(Node* x)
{
    Node* y = x->right;
    Node* T2 = y->left;
    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left),
        height(x->right)) + 1;
    y->height = max(height(y->left),
        height(y->right)) + 1;
    // Return new root
    return y;
}
```

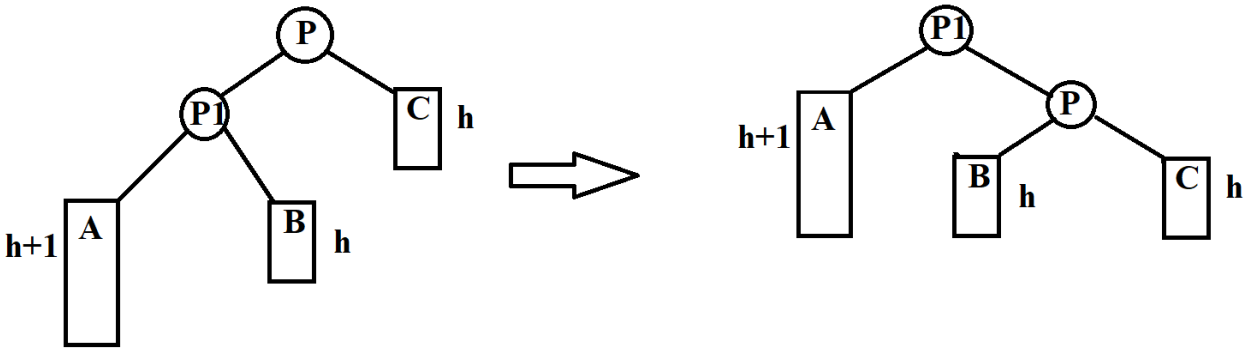


This is the case of right rotation at node X

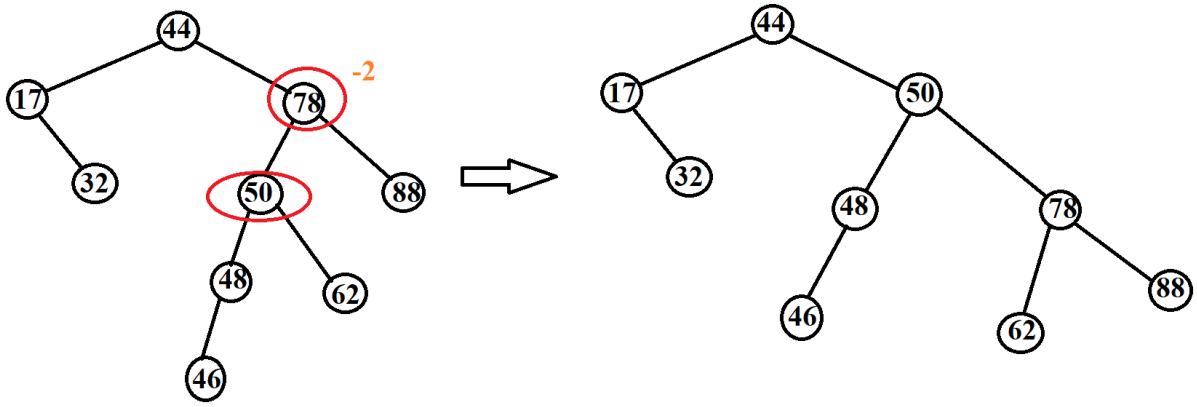
```
Node* rightRotate(Node* y)
{
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
        height(y->right)) + 1;
    x->height = max(height(x->left),
        height(x->right)) + 1;

    // Return new root
    return x;
}
```

- We have 4 cases to edit the tree which are Left-Left Case, Right-Right Case, Left-Right Case and Right-Left Case. But we only need to know 2 cases, Left Left Case and Left Right Case because the other 2 cases we do the same but on different sides.

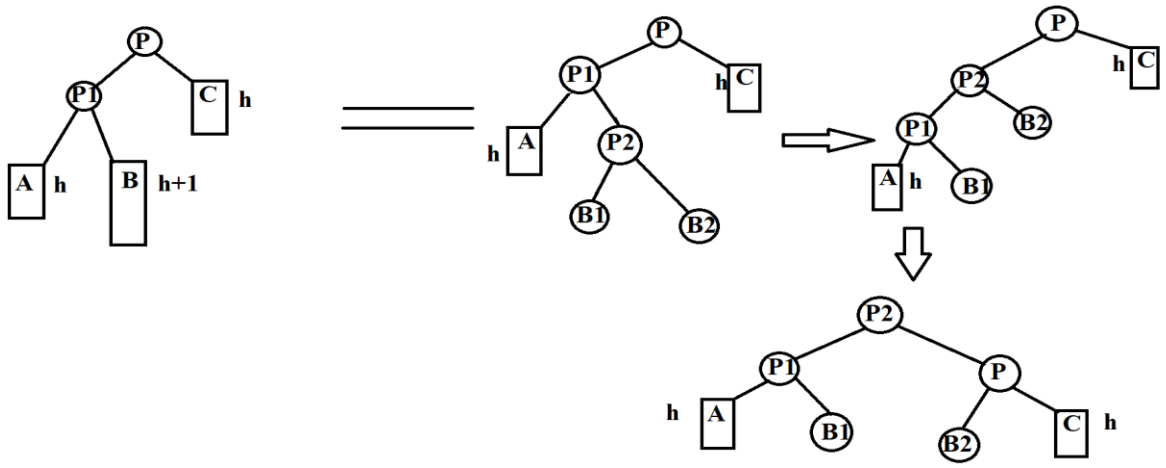
+ Left-Left Case (L-L): In this case we remember that the node is left unbalanced, the left child of that node is also unbalanced to the left.



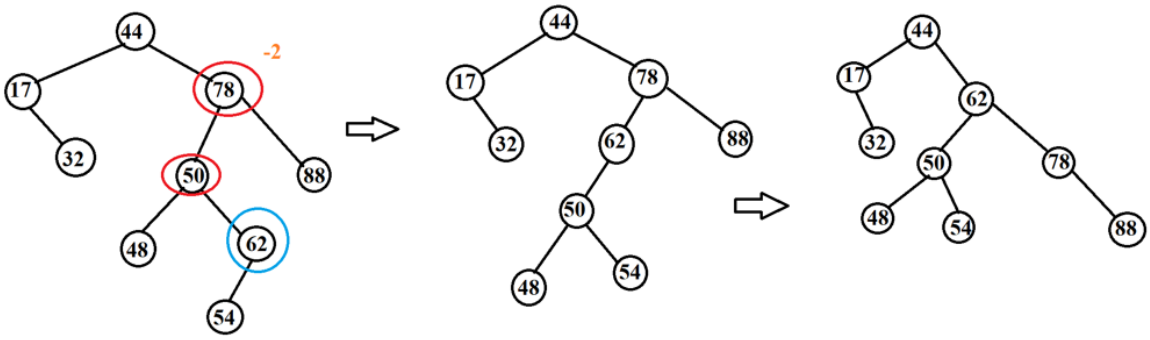
Example:



+ Left_Right Case (L-R): In this case we remember that the left node is unbalanced, the left child of that node is also unbalanced on the right.



Example:



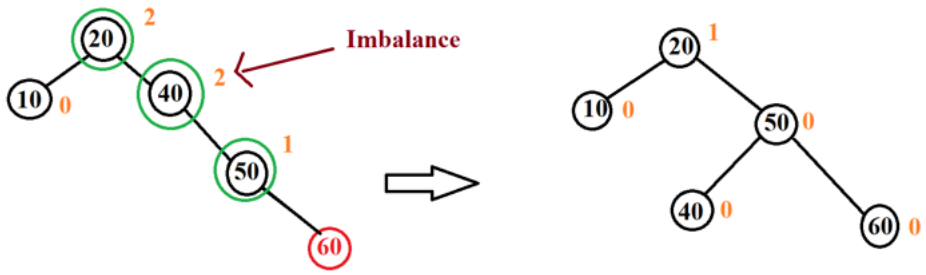
- Insert, delete: How tree-like insert and delete BST but the other is in step when done, we must calibrate the balance tree (we only balance when the length of the left subtree minus the length of the right subtree and we take abs, the result is large more than 1).

+ Insert: When it is finished, we will go back to the previously reviewed nodes to see if that node is unbalanced or not.

```
Node* insert(Node*node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
    {
        //node = newNode(key);
        return(newNode(key));
    }

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left), height(node->right));

    /* 3. Get the balance factor of this
        ancestor node to check whether
        this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
```

```
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}
```

Example: when we add 60



We see that when we go back to the reviewed nodes, node 40 we see that it is unbalanced and will perform a tree correction. When editing the tree is complete, it will continue to go back up on the nodes that were traversed at the beginning. It will execute until there are no more nodes to go back to. In the above case, it will go up again in the order of 50, 40 and 20

+ Delete: When it is finished, we will go back to the previously reviewed nodes to see if that node is unbalanced or not.

```
Node* deleteNode(Node* root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);

    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else
    {
        // node with only one child or no child
        if ((root->left == NULL) ||
            (root->right == NULL))
        {
            Node* temp = root->left ? root->left : root->right;
            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of
                               // the non-empty child
            free(temp);
        }
        else
        {
            Node* temp = minValueNode(root->right);
            root->key = temp->key;
            // Delete the inorder successor
            root->right = deleteNode(root->right,
                temp->key);
        }
    }
    if (root == NULL)
        return root;
    root->height = 1 + max(height(root->left),
        height(root->right));
```

```
    int balance = getBalance(root);
    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
```

- Complexity and tree height:

+Tree height:

. $h_{AVL} < 1.44 \log_2(N + 1)$.

. AVL tree is 44% higher than optimal BST tree.

+ Complexity:

. Search: $O(\log_2 N)$.

. Adding an element: $O(\log_2 N)$, including tree search $O(\log_2 N)$ and editing $O(\log_2 N)$.

. Deleting element O(log2N), including tree search $O(\log_2 N)$ and editing $O(\log_2 N)$.

III. Illustration examples

Example 1:

INPUT:

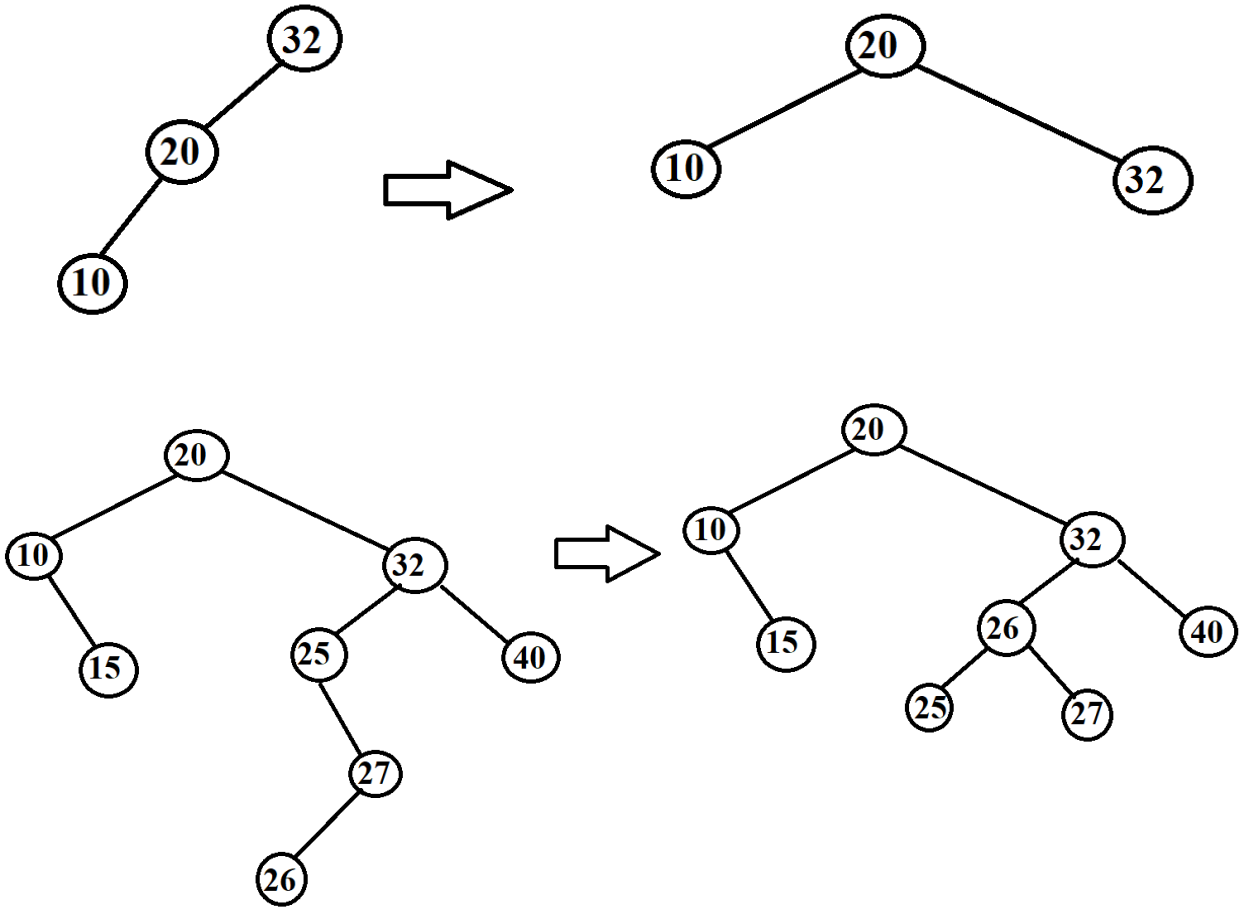
-Insert:32, 20, 10, 15, 40, 25, 27, 26, 13, 14.

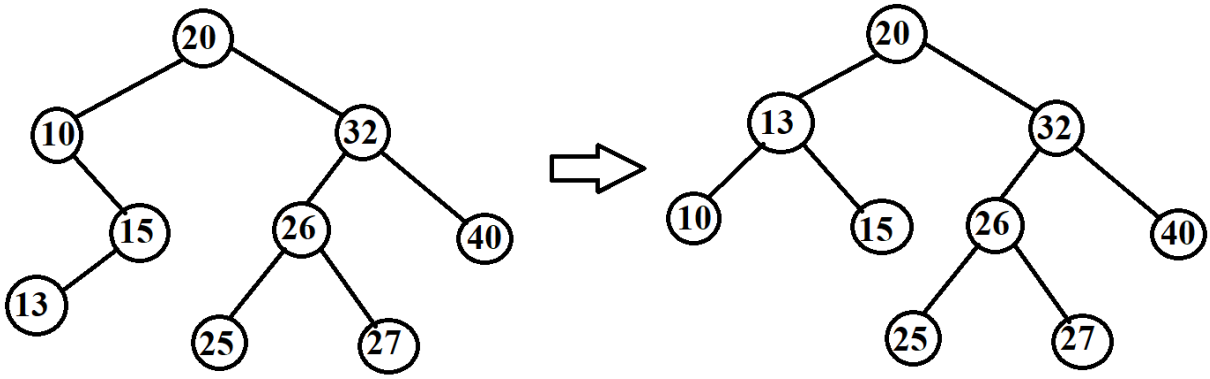
-Find: 15, 29

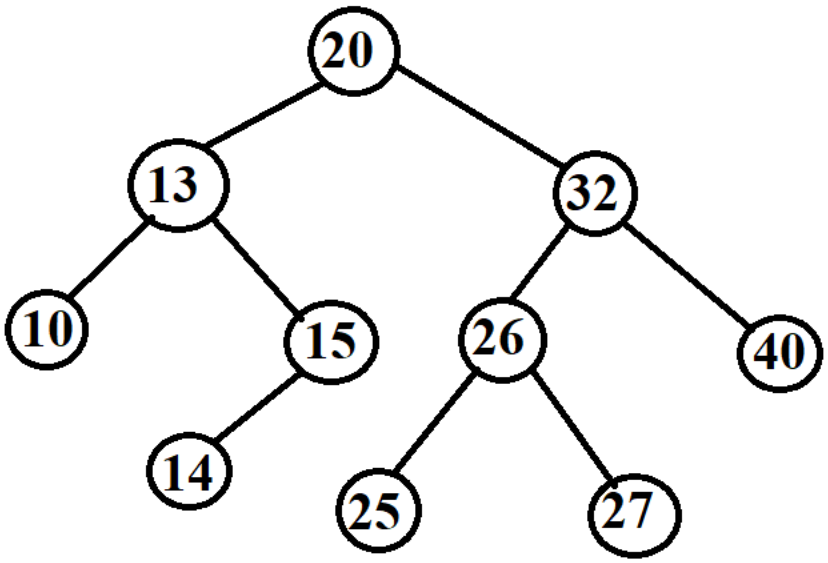
-Delete: 25, 40, 20

OUTPUT:

-Insert:

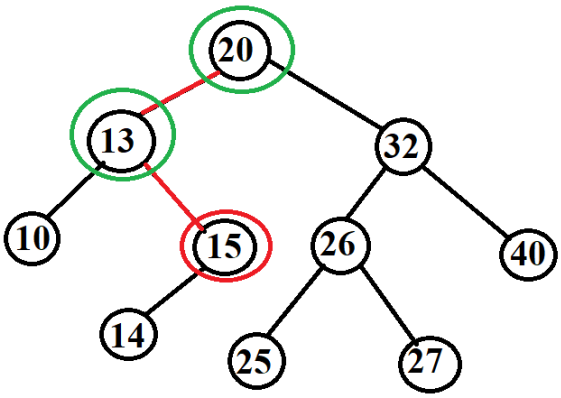



20
10
32
15
26
40
13
25
27
20
13
32
10
15
26
40
25
27

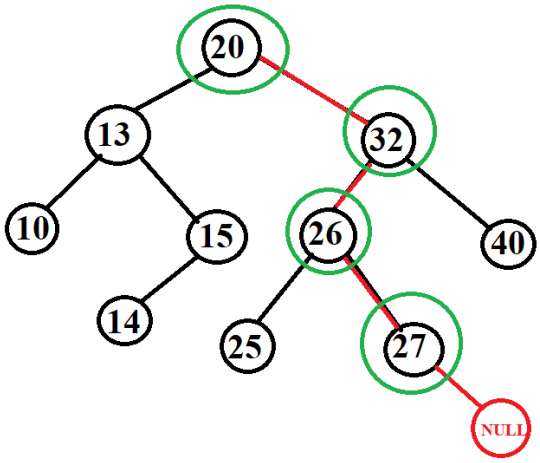

20
13
32
10
15
26
40
14
25
27

- Find:

Find 15

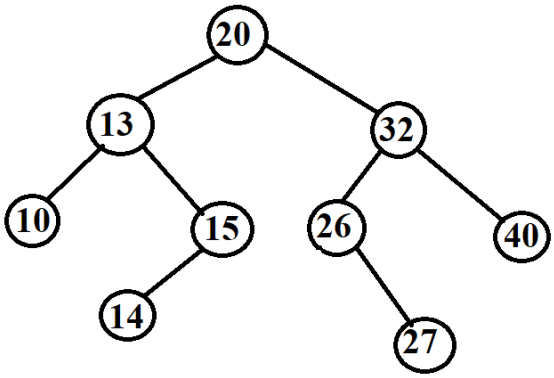


Find 29

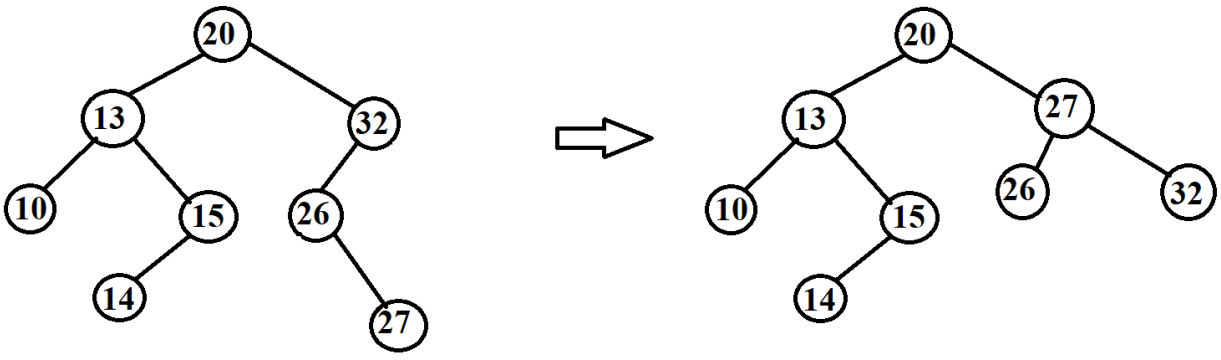


- Delete:

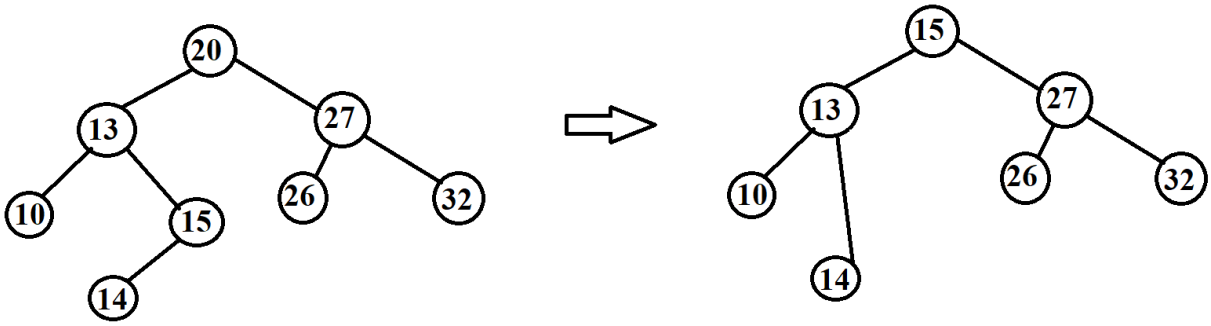
Delete 25



Delete 40



Delete 20



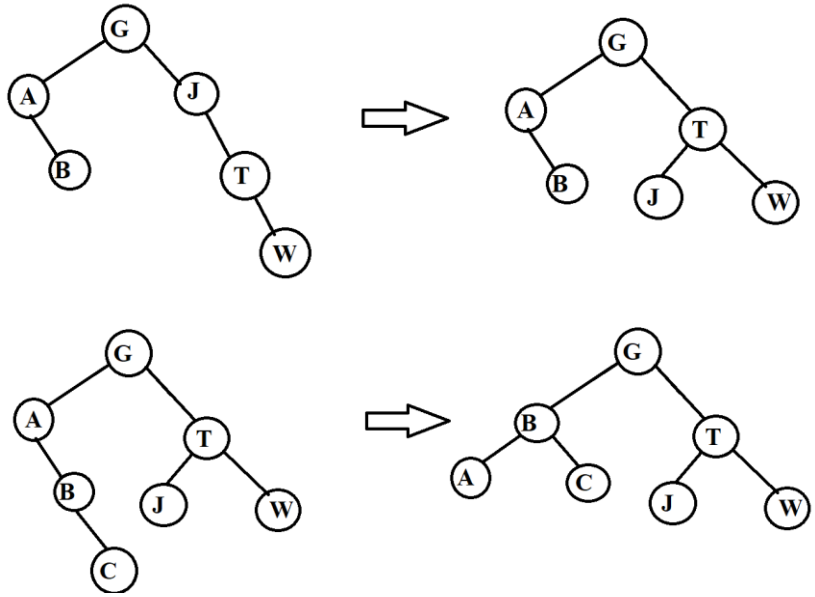
Example 2:

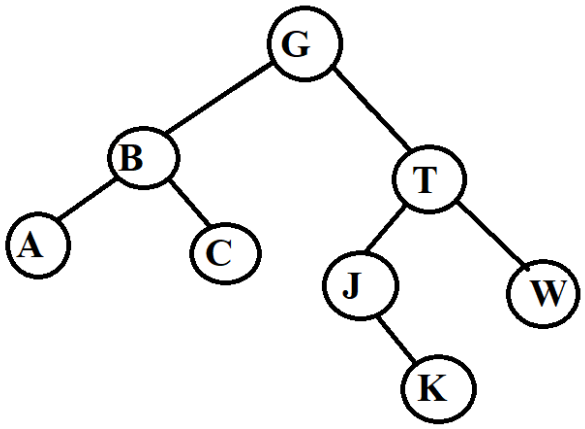
INPUT:

- Insert: G, J, A, T, B, W, C, K.
- Find: A, Z
- Delete: J, A

OUTPUT:

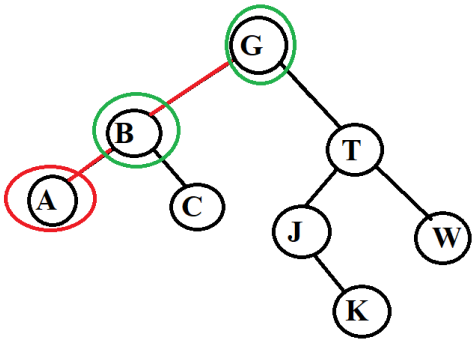
- Insert:



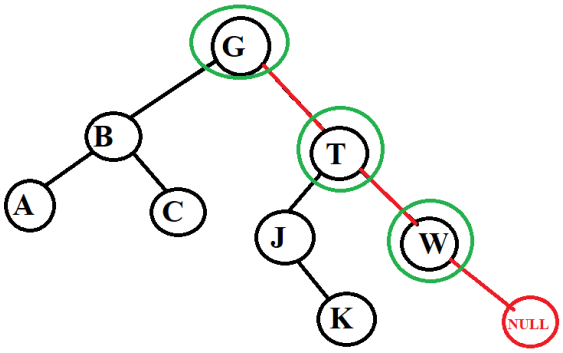


- Find:

Find A

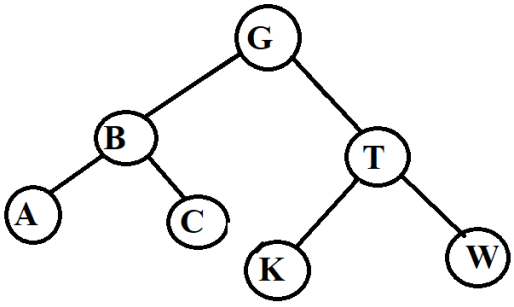


Find Z

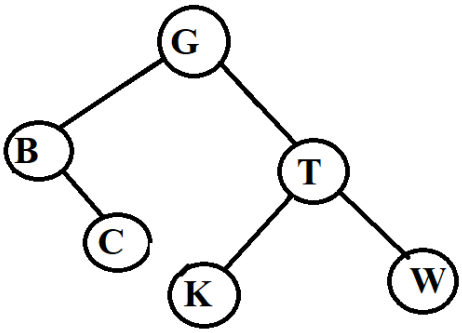


- Delete:

Delete J



Delete A



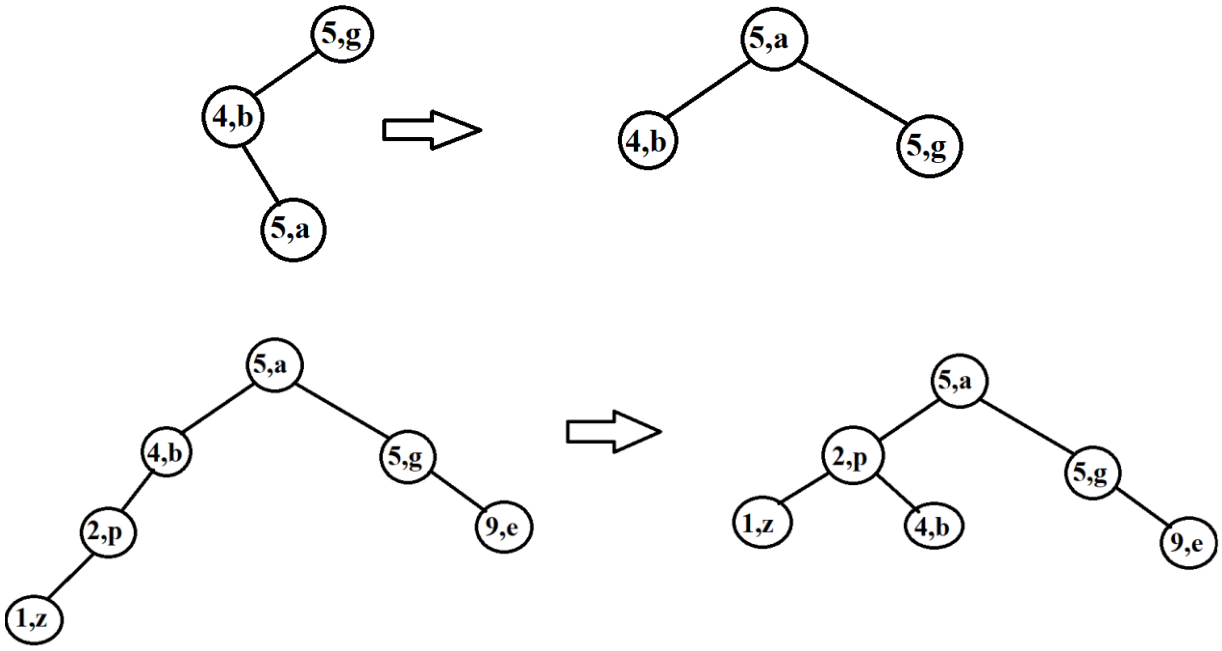
Example 3:

INPUT: input is a data pair of the form (x, y) where x is an integer and y is a letter. We will compare the number first if the number is equal, we compare the letter.

- Insert: (5, g), (4, b), (5, a), (9, e), (2, p), (1, z)
- Find: (9, e), (7, a)
- Delete: (5, g), (2, p)

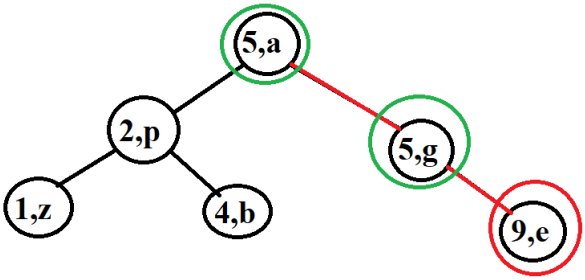
OUTPUT:

- Insert:

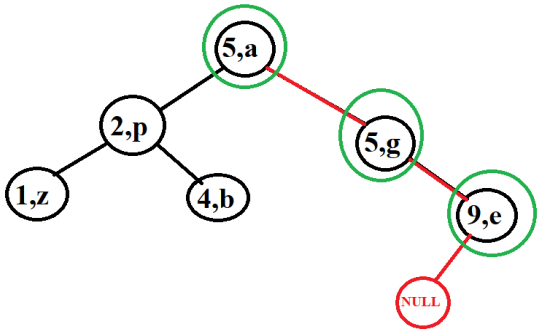


- Find:

Find (9, e)

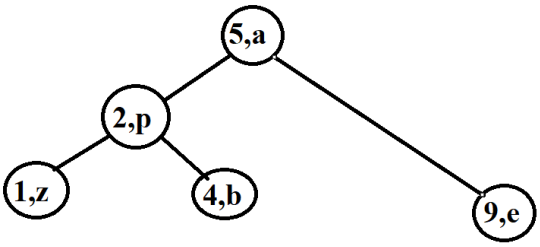


Find (7, a)

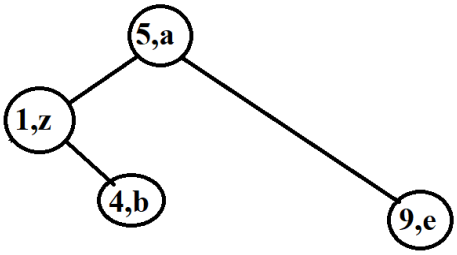


- Delete:

Delete (5, g)



Delete (2, p)



Lesson 5: Red Black Tree (RB Tree)

Contents:

I. Introduction

***** Question:

-Is there a balanced search tree that has fewer structural changes in the tree but is faster to find, delete and insert than an AVL tree in practice? The answer is yes, it is a red black tree as fewer rotations are done due to relatively relaxed balancing.

* The main content of the Red-Back tree:

- Red-Black Tree is a binary search tree.

- It must satisfy 5 principles:

[1] Each node is red or black.

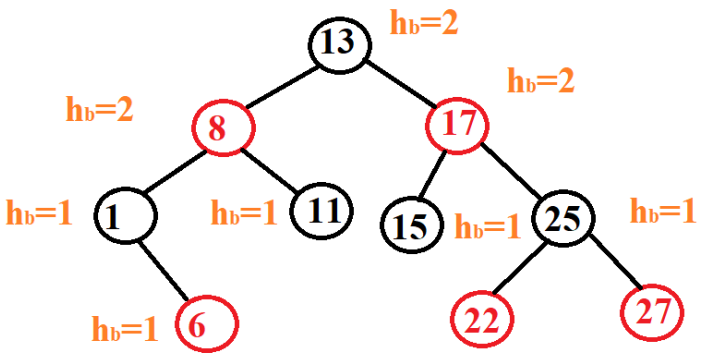
[2] The root node is always black.

[3] Node NULL is always black. In a B Tree, we will call the NULL node NIL because it has the unique properties of the RB Tree such as color.

[4] If that node is red then its child node is black. That means there is no case of 2 red nodes in a row.

[5] If all patches originate from any 1 node (including the root node), the number of black nodes in its path to the NULL node must be equal. From there, we will denote the number of black nodes as $h_b(x)$. This black node calculation does not count node x if that node x is a black node.

Example:



II. Operations on Red Black Tree

- How to declare node in RB tree:

```
#define RED 1
#define BLACK 0
typedef bool colour;
struct node
{
    node *parent,*left,*right;
    int key;
    colour color;
};
#define NIL &sentinel           /* all leafs are sentinels */
node sentinel = {NIL,NIL, NIL, 0, BLACK};
```

- Insert a new element:

+ The inserted node is always red because then our rule number 5 will not be normalized but agree that rule number 4 can be violated but then it will be easier for us to modify the tree.

+ We also do the same as the BST tree in the first step, find the insertion location.

+ The next step we will check to see if it is in any of the cases after inserting:

. If the parent node is Black => it doesn't matter.

. If the parent node is Red => violates parent-child are Red => adjust to balance tree. The tree correction here only takes place when there are 2 consecutive red nodes, not like the AVL tree, which has to check the length of 2 subtrees of the node. After done, if the tree meets the equilibrium condition, then it is correct.

```
void insert(int key)
{
    if(root==NIL)
    {
        root=new node;
        root->left=NIL;
        root->right=NIL;
        root->parent=NULL;
        root->color=BLACK;
        root->key=key;
        return;
    }
    node *parent,*current,*x;
    parent=NULL;
    current=root;
    while(current!=NIL)
    {
        if(key==current->key) return;
        parent=current;
        if(key>current->key) current=current->right;
        else current=current->left;
    }
    x=new node;
    x->parent=parent;
    x->left=NIL;
    x->right=NIL;
    x->color=RED;
    x->key=key;

    if(key>parent->key) parent->right=x;
    else parent->left=x;

    fixinsert(x);
    return;
}
```

Before going into how to edit the tree, let's talk about rotation in RB Tree. Basically, the rotation is no different from the rotation of a node in AVL, but in the code we have to modify the parent node of that node because then the parent node of that node is already connected to the child node of the rotated node.

```
void rotateLeft(node *x)
{
    if(x->right==NIL) return;
    node *y=x->right;
    x->right=y->left;
    if(y->left!=NIL) y->left->parent=x;
    y->parent = x->parent;
    if(x!=root)
    {
        if(x==x->parent->left)
            x->parent->left=y;
        else x->parent->right=y;
    }
    else root=y;
    y->left=x;
    x->parent=y;
}
```

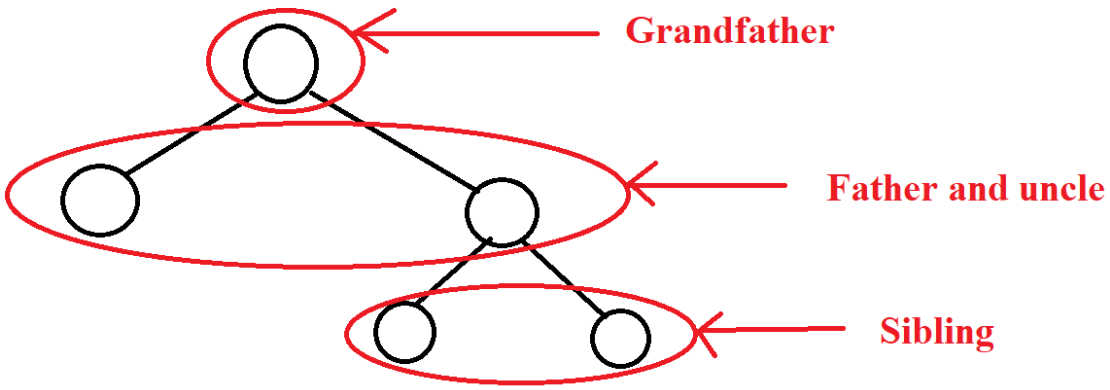
```
void rotateRight(node *x)
{
    if(x->left==NIL) return;
    node *y=x->left;
    x->left=y->right;
    if(y->right!=NIL) y->right->parent=x;
    y->parent = x->parent;
    if(x!=root)
    {
        if(x==x->parent->left)
            x->parent->left=y;
        else x->parent->right=y;
    }
    else root=y;
    y->right=x;
    x->parent=y;
}
```

+ How to edit a tree when the parent node is a red node: The algorithms have mainly two cases.

. If the uncle is red, we do recolour.

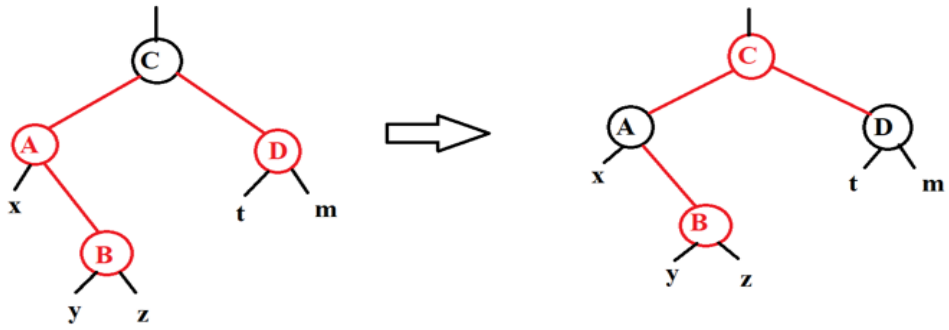
. If the uncle is black, we do rotations and /or recolouring.

Note: uncle is the same node as the parent node of the node under consideration.

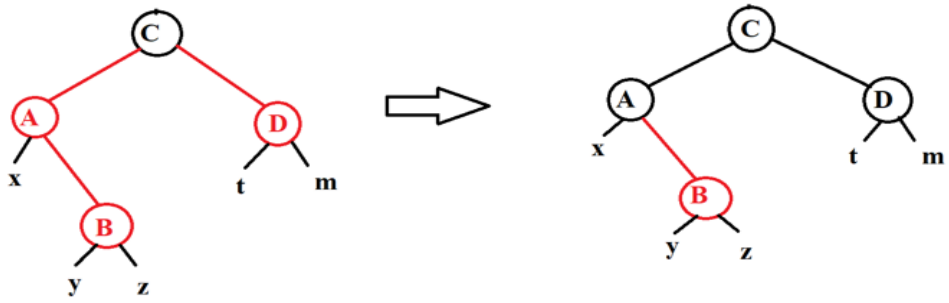


We list 6 smaller cases for easy control. But we only need to find the first 3 cases where the newly inserted node is located in the left subtree of the grandfather node and the last 3 cases do the same as above but only differ in direction.

Case 1: If the parent node is red, the uncle is red, then we only need to change the color of the parent node, uncle and grandfather node. But notice that if he is a root node, we won't change the color but keep it black.

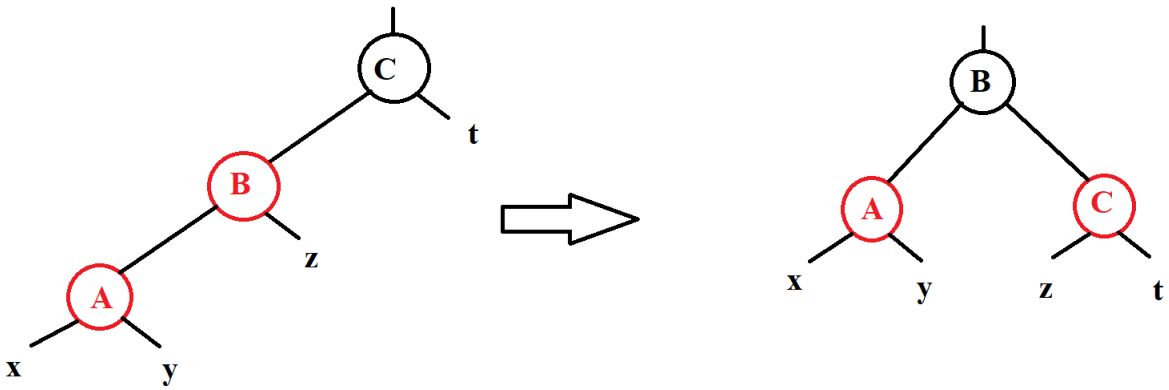


If C is not root

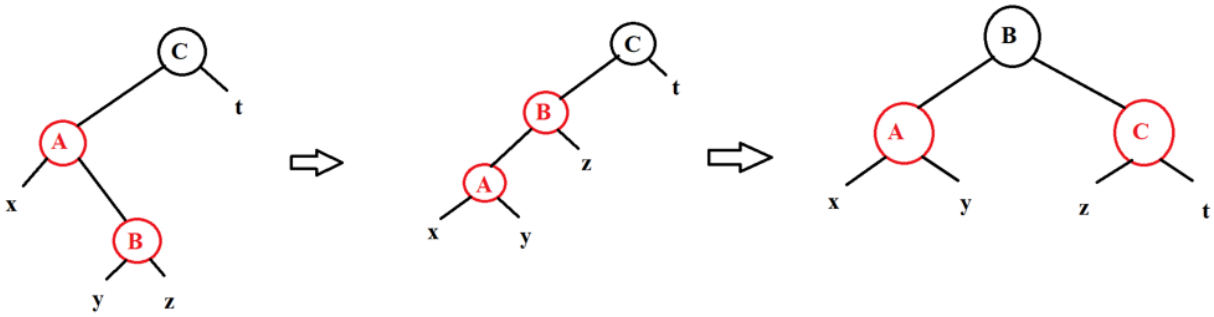


If C is root

Case 2: If the parent node is red, the uncle is black and the inserted node is the left child of the parent node, we change the color of the parent node and the grandfather node, then we will rotate right at the grandfather node.



Case 3: If the red parent node, the black uncle node and the newly inserted node are the right child nodes of the parent node, then we take the parent node to turn left and right rotate the grandfather node. Finally, we change the color of the parent node and the grandfather node.



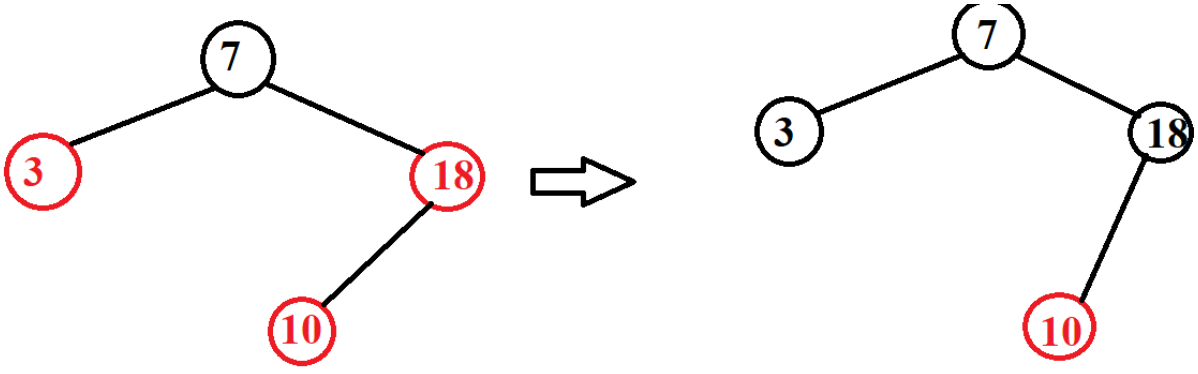
Case 4, Case 5, Case 6: We also do the same operation as above, but only in the opposite direction, this time the node is added it is in the right subtree of his node.

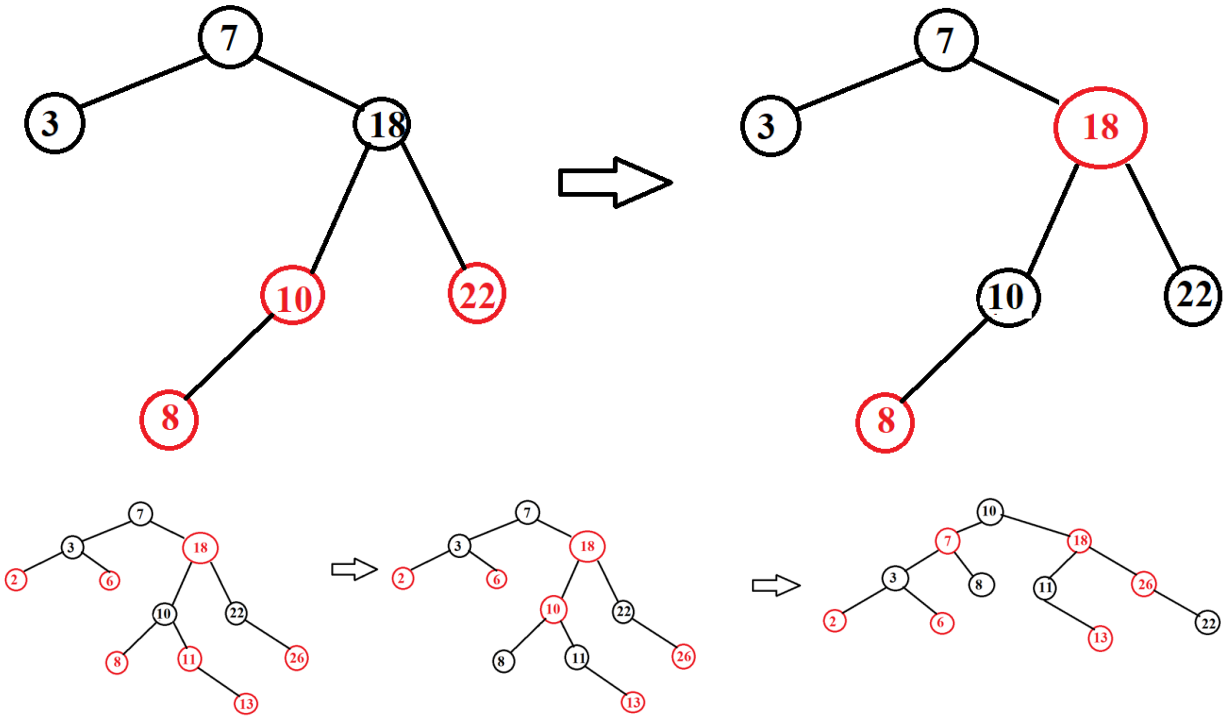
+ At the last step of the insert, we just do the same as the AVL tree, which is to go against the above nodes that have been approved to see if there is a violation of the attribute of 2 consecutive red nodes.

```
void fixinsert(node* x)
{
    while(x!=root&&x->parent->color==RED)
    {
        if(x->parent==x->parent->parent->left)
        {
            node *y=x->parent->parent->right;
            if(y->color==RED)
            {
                x->parent->color=BLACK;
                y->color=BLACK;
                x->parent->parent->color=RED;
                x=x->parent->parent;
            }
            else{
                if(x==x->parent->right)
                {
                    x=x->parent;
                    rotateLeft(x);
                }
                x->parent->color=BLACK;
                x->parent->parent->color=RED;
                rotateRight(x->parent->parent);
            }
        }
        else
        {
            node *y=x->parent->parent->left;
            if(y->color==RED)
            {
```

```
                x->parent->color=BLACK;
                y->color=BLACK;
                x->parent->parent->color=RED;
                x=x->parent->parent;
            }

            else
            {
                if(x==x->parent->left)
                {
                    x=x->parent;
                    rotateRight(x);
                }
                x->parent->color=BLACK;
                x->parent->parent->color=RED;
                rotateLeft(x->parent->parent);
            }
        }
    }
    root->color=BLACK;
}
```

Example: Add 7, 3, 18, 10, 22, 8, 11, 26, 2, 6, 13





- Delete an element:

+ The first step we do the same as BST is to find and delete the node to be deleted. Note when in the case of deleting node 2 children, we only replace the lice without changing the color.

+ Just like inserting we also have 2 cases:

. The actually deleted node is the red node. In this case, we just need to delete the node normally or in other words, treat it as a BST tree to delete. Because it doesn't violate any rules in the RB tree.

. The actually deleted node is the black node. This is the most difficult case because deleting it will cause many cases of violating the rules of the tree. For example, we will be able to make the root node turn red because it can delete root and replace it with a red node. It may violate the rule that 2 consecutive child nodes cannot be red because deleting that node may cause the child node and parent node of the deleted node to be red and connected. Finally, we can

change the number of black nodes of a node to the NIL node because deleting can reduce the black node.

+ As mentioned above, the case of deleting a black node is the most difficult case we need to consider.

. After finding the node that needs to be deleted, we don't need to delete it, but we have to do an action called removing that node from the tree. That is, if the node is less than 2 children, then we connect the parent node with the child and then we will edit the tree immediately the child node of the node is deleted, and in the case of deleting a node with 2 child nodes, we will immediately deal with the child node at the node whose replacement price is taken.

. After editing the tree, we will delete the node to delete.

```
void erase(int key)
{
    node *x,*y,*current;
    current=root;
    while(current!=NIL)
    {
        if(current->key==key) break;
         if(key>current->key) current=current->right;
        else current=current->left;
    }
    if(current==NIL) return;
    if(current->left==NIL||current->right==NIL)
    {
        y=current;
    }
    else
    {
        y=current->left;
        while(y->right!=NIL) y=y->right;
    }
    if(y->left!=NIL) x=y->left;
    else x=y->right;

    x->parent=y->parent;
    if(y->parent!=NIL)
    {
        if(y==y->parent->left)
            y->parent->left=x;
        else y->parent->right=x;
    }
    else
    {
        root = x;
    }
```

```
    current->key=y->key;
    if(y->color==BLACK)
        fixerase(x);
    delete y;
    return;
}
```

- There are 2 cases in case of deleting the black node:

Case 1: The child of the delete black node is in red: we just need to change the color of child node to black.

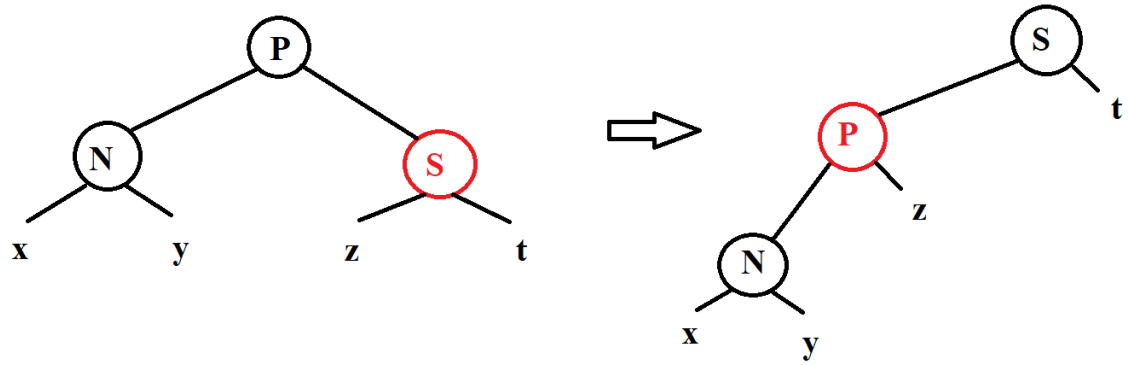
Case 2: the child of the deleted black node is in black: we will do the steps below (We only need to consider the case that the node to be deleted is in the left subtree of its parent node, in the other case we do the same but only do the opposite.) We will call the node that is deleted as node n, the sibling node as node s and the parent node is p.

- Step 1: If n has no parent (n is the root): end. Otherwise, we go to step 2.
- Step 2: n is black, s is red.

- We change color and rotate left at parent node.

- We change the color of node p.

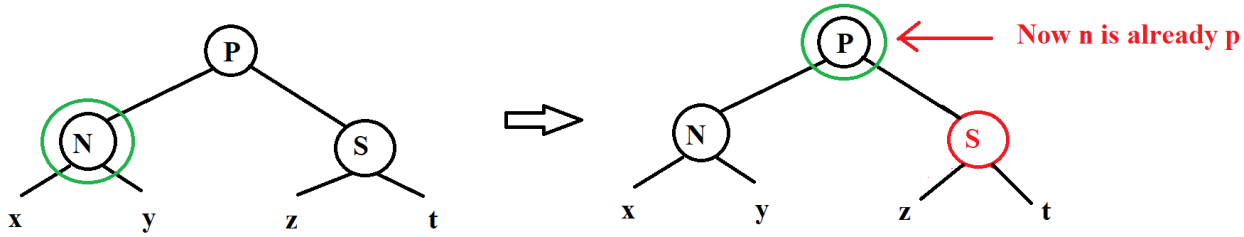
- Go to step 3.



- Step 3: n is black, s is black, children of s are black, p is black.

  - Change node s color to red.

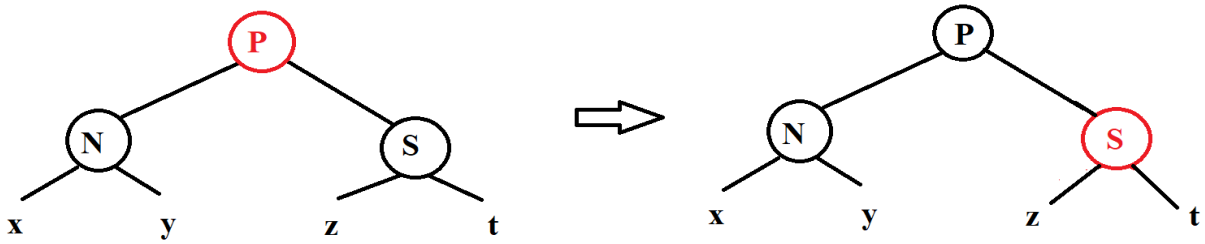
  - Go back to step 1 with n being its parent node.



- Step 4: n is black, s is black, children of s is black, p is red.

- Change node p to black and node s to red.

- End the process of editing the tree when deleting here. Otherwise, if we do not fall into this step, we will go to step 5.

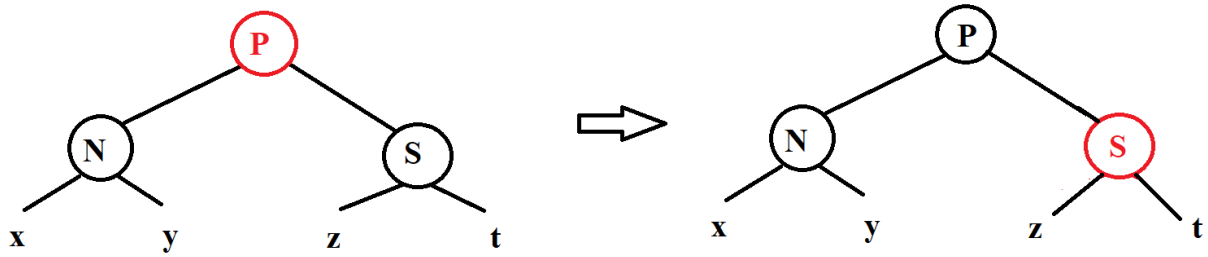


- Step 5: n is black, s is black, left child of s is red, right child of s is black.

- We change the color of node s and the left child of node s.

- We turn right at node s.

- Go to step 6.



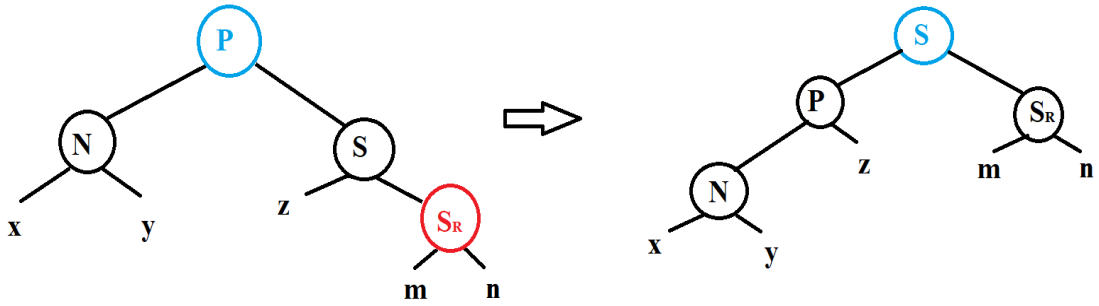
- Step 6: n is black, s is black, right child of s is red.

  -We change the color of node p and s for each other.

  -We change the color of the right child node of node s to black.

  -Turn left at node p.

  -We end the tree editing process here.

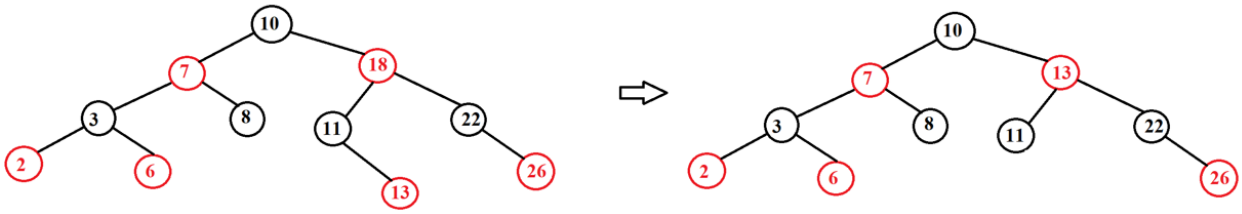


```
void fixerase(node *x)
{
    while (x != root && x->color == BLACK)
    {
        if (x == x->parent->left)
        {
            node* w = x->parent->right;
            if (w->color == RED)
            {
                w->color = BLACK;
                x->parent->color = RED;
                rotateLeft (x->parent);
                w = x->parent->right
            }
            if (w->left->color == BLACK && w->right->color == BLACK)
            {

                w->color = RED;
                x = x->parent;
            }
            else
            {
                if (w->right->color == BLACK)
                {
                    w->left->color = BLACK;
                    w->color = RED;
                    rotateRight(w);
                    w = x->parent->right;
                }

                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                rotateLeft(x->parent);
                x = root;
            }
```

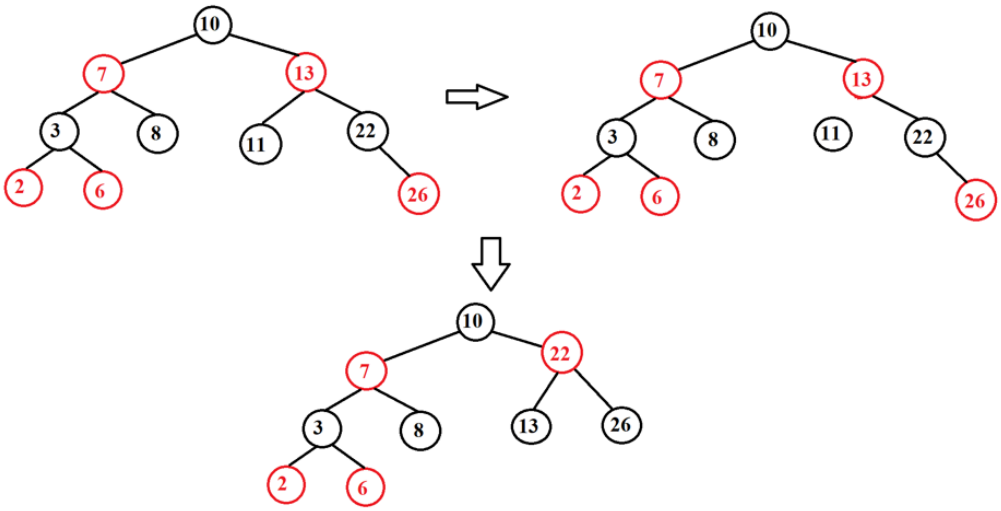
```
        }
        else
        {
            node *w = x->parent->left;
            if (w->color == RED)
            {
                w->color = BLACK;
                x->parent->color = RED;
                rotateRight (x->parent);
                w = x->parent->left;
            }
            if (w->right->color == BLACK && w->left->color == BLACK)
            {
                w->color = RED;
                x = x->parent;
            }
            else
            {
                if (w->left->color == BLACK)
                {
                    w->right->color = BLACK;
                    w->color = RED;
                    rotateLeft (w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                rotateRight (x->parent);
                x = root;
            }
        }
    }
    x->color = BLACK;
}
```

Example: We have an RB tree 7, 3, 18, 10, 22, 8, 11, 26, 2, 6, 13. We will delete the elements 18, 11.

Delete 18



Delete 11



- Complexity and tree height:

+ Complexity:

. Search O($\log_2 N$)

. Insert O($\log_2 N$)

. Delete O($\log_2 N$)

. Minimum O($\log_2 N$)

. Maximum O($\log_2 N$)

III. Illustration examples

Example 1:

- **INPUT:**

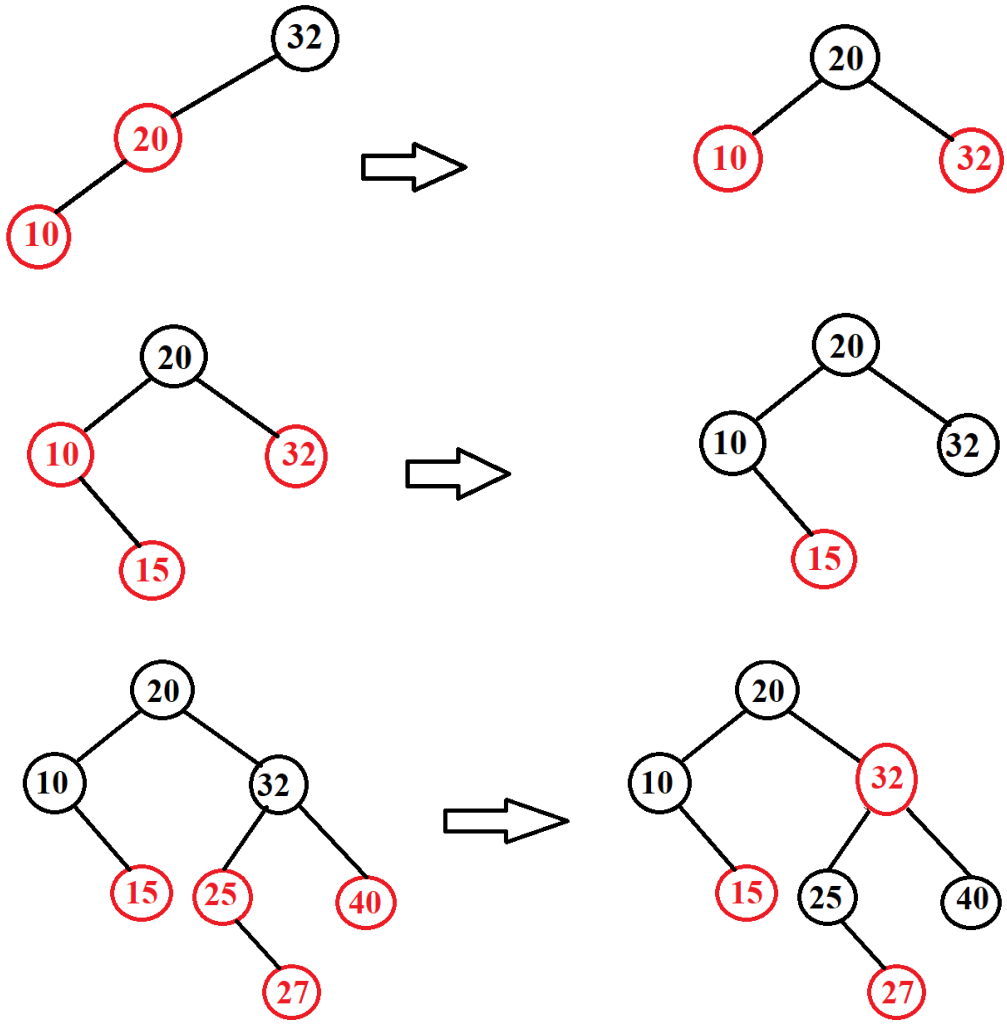
- Insert:32, 20, 10, 15, 40, 25, 27, 26, 13, 14.

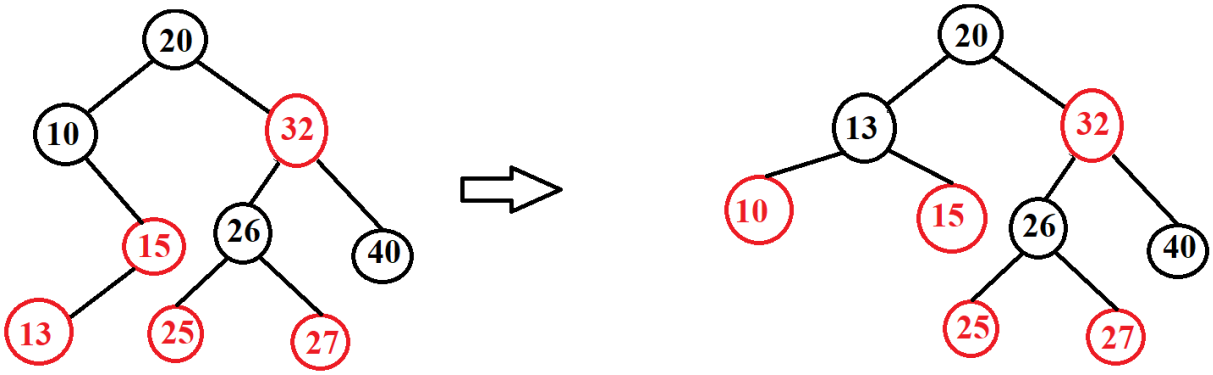
- Find: 15, 29

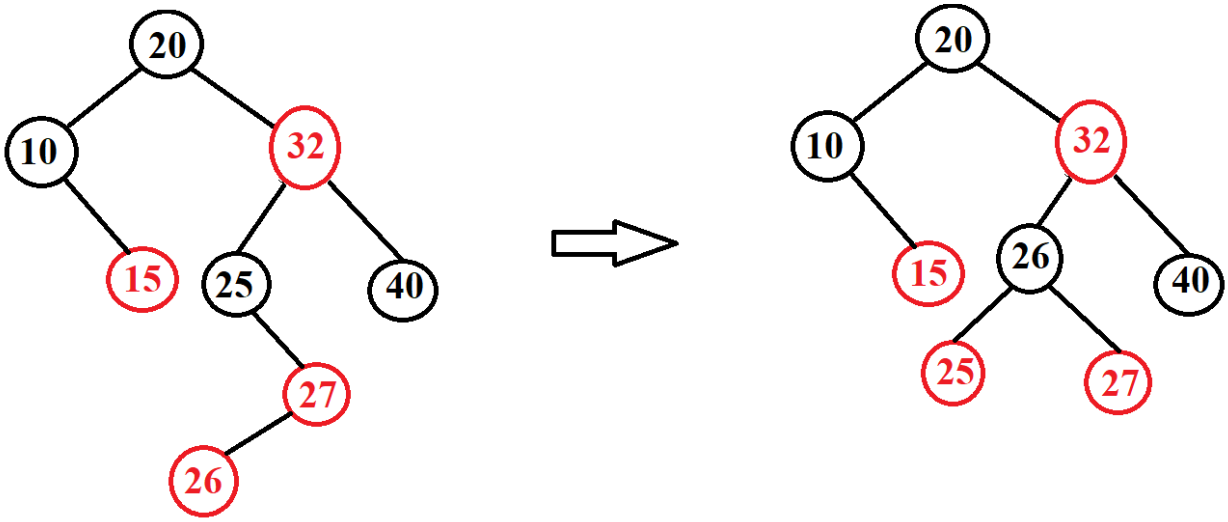
- Delete: 25, 40, 20

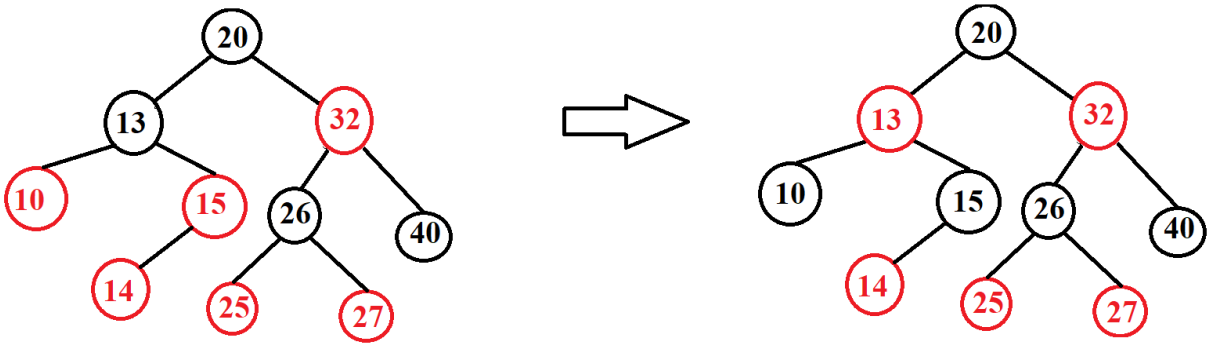
- **OUTPUT:**

- Insert:



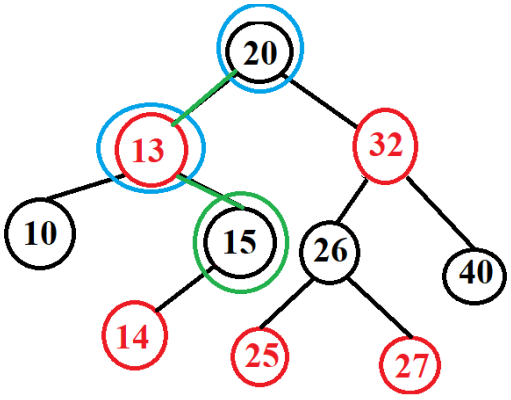

20
10
32
15
26
40
13
25
27
20
13
32
10
15
26
40
25
27


20
10
32
15
25
40
27
26
20
10
32
15
26
40
25
27

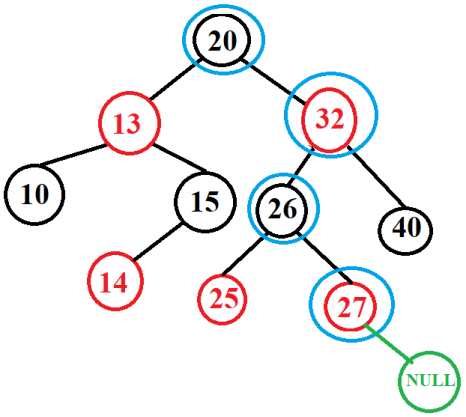

20
13
32
10
15
26
40
14
25
27
20
13
32
10
15
26
40
14
25
27

- Find:

Find 15

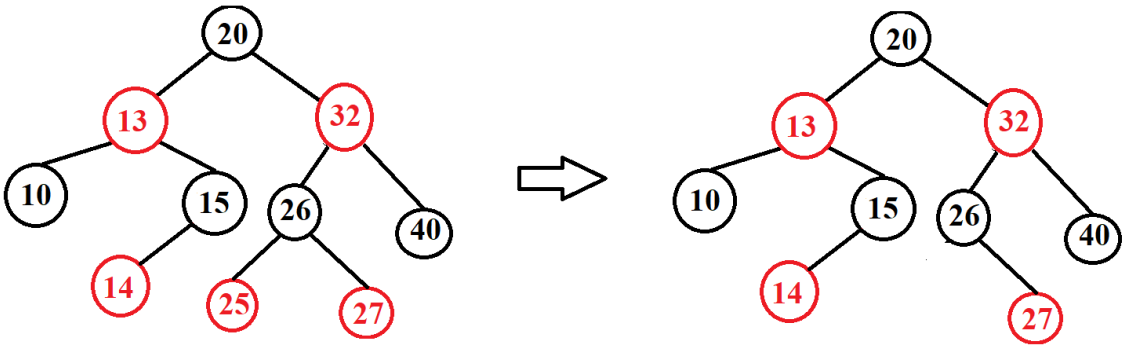


Find 29

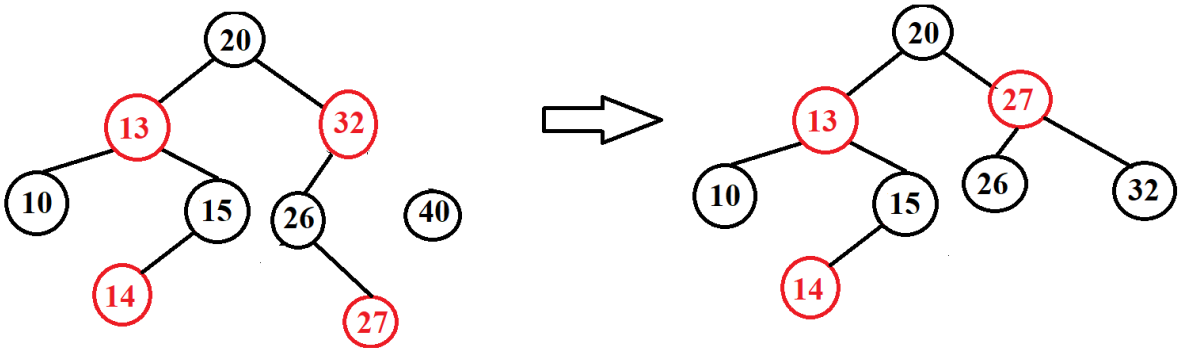


- Delete:

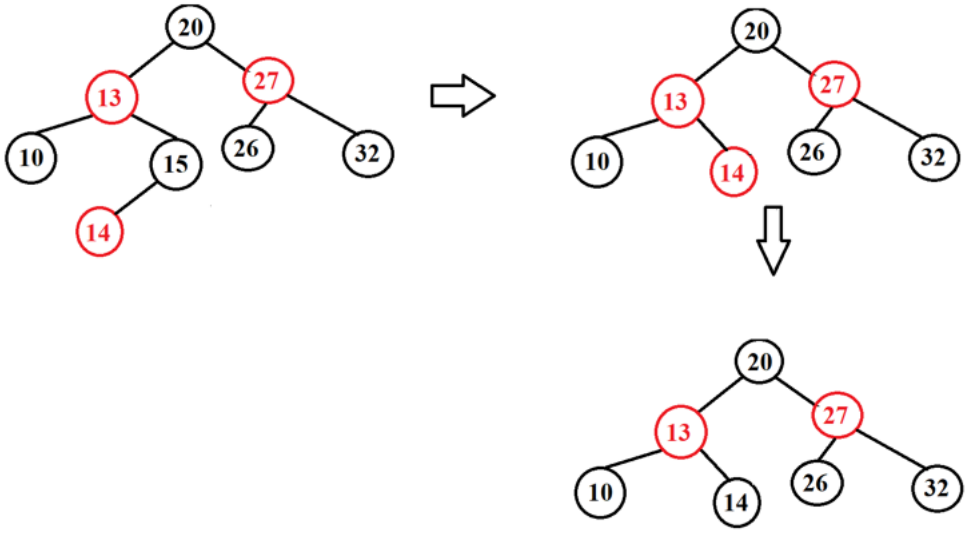
Delete 25



Delete 40



Delete 20



Example 2:

- **INPUT:**

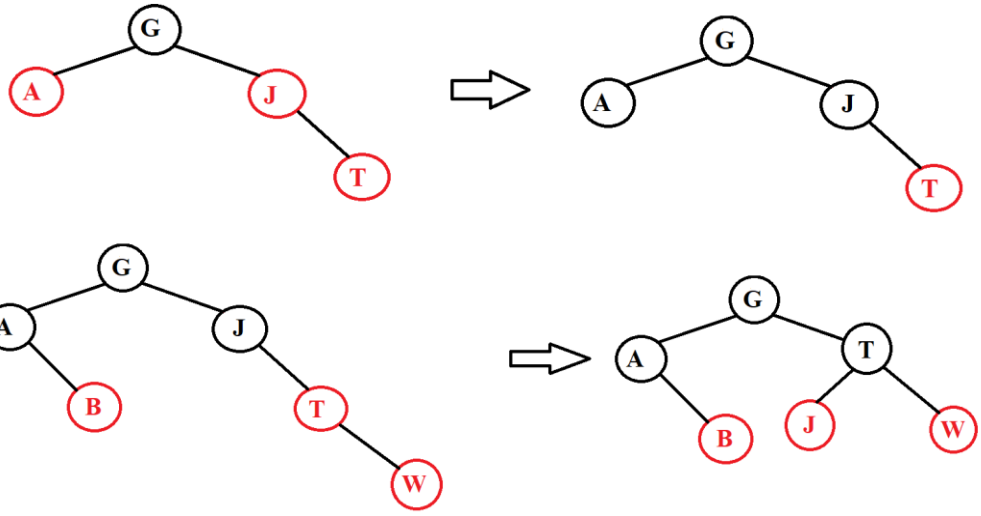
- Insert: G, J, A, T, B, W, C, K.

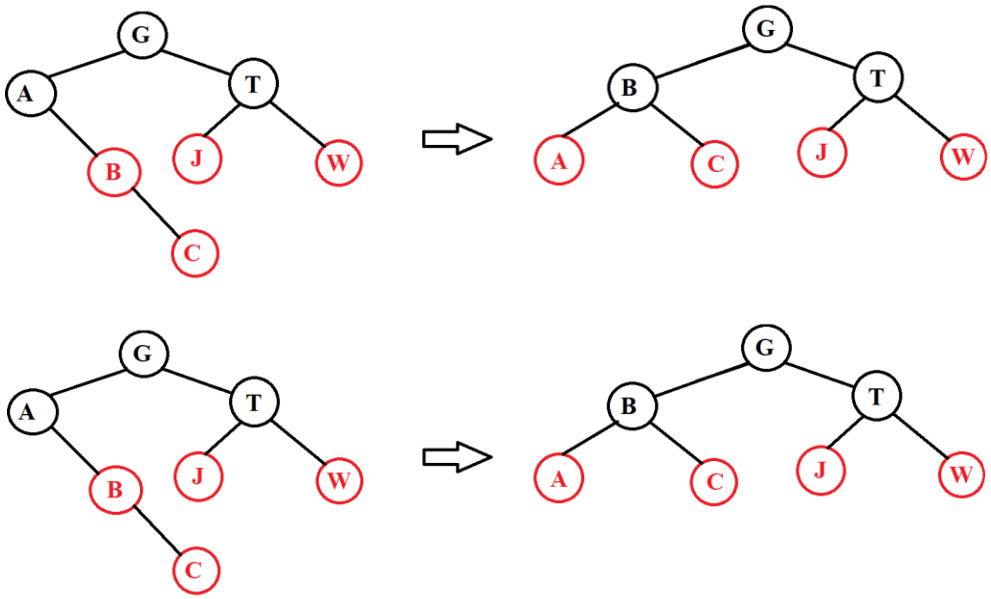
- Find: A, Z

- Delete: J, A

- **OUTPUT:**

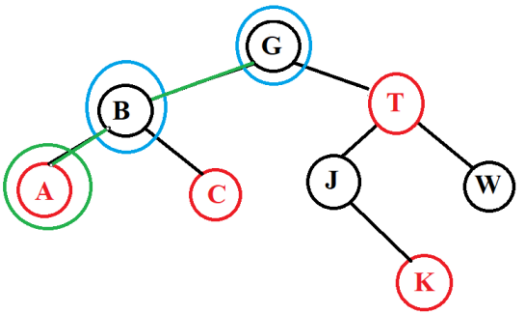
- Insert:



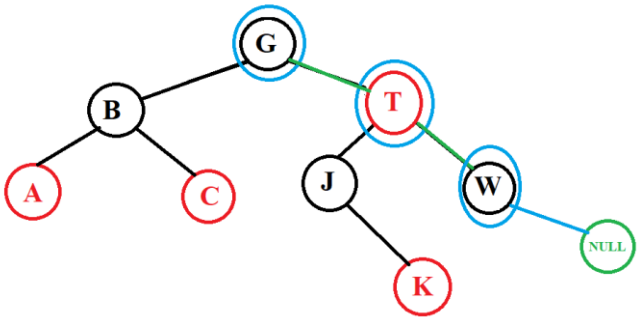


- Find:

Find A

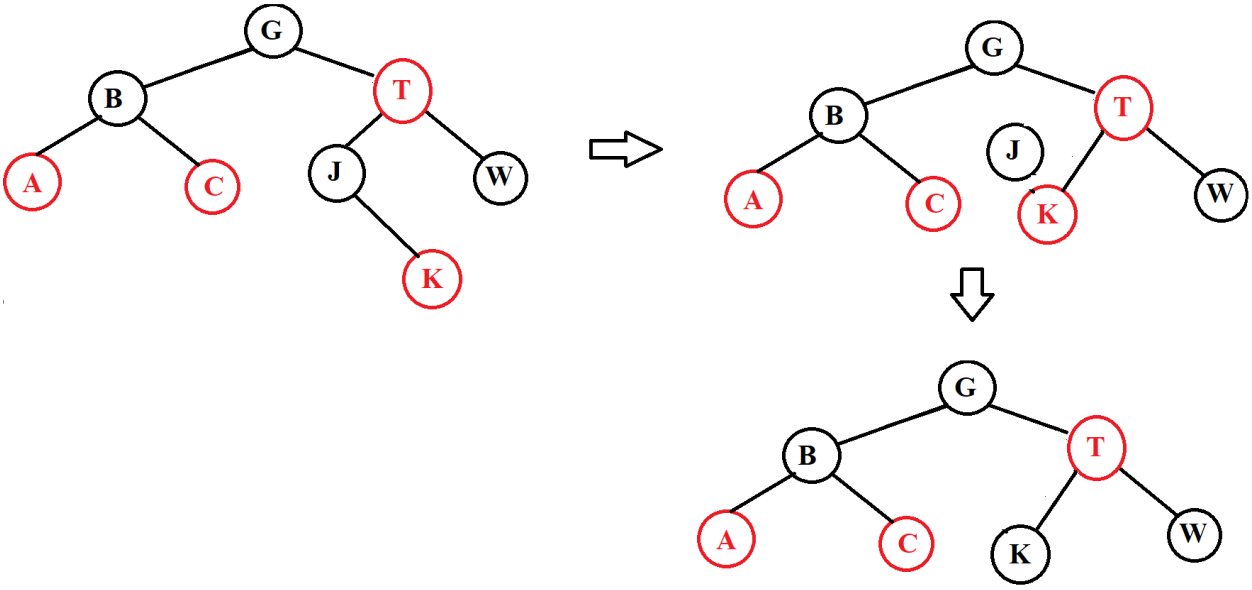


Find Z

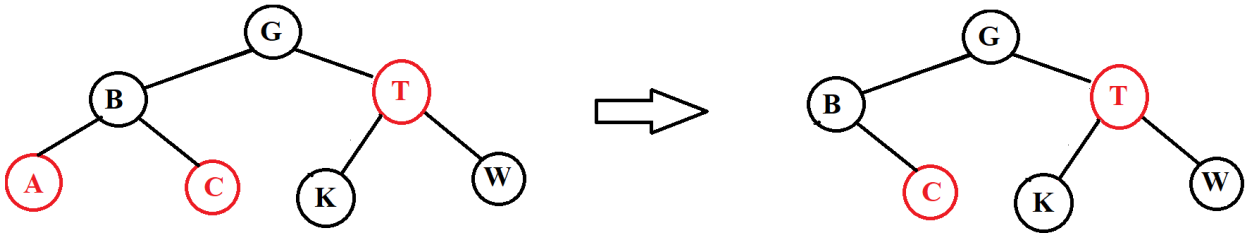


- Delete:

Delete J



Delete A



Example 3:

- **INPUT:** input is a data pair of the form (x, y) where x is an integer and y are a letter. We will compare the number first if the number is equal, we compare the letter.

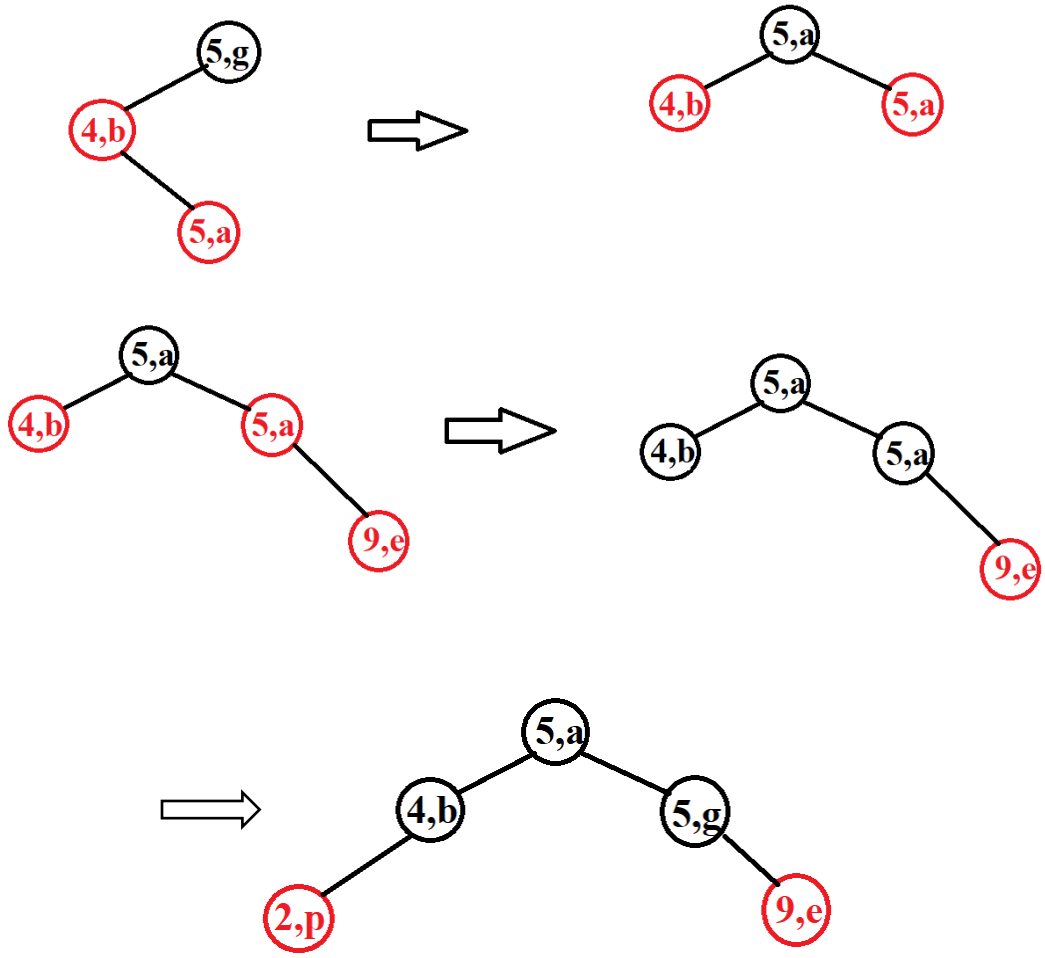
-Insert: (5, g), (4, b), (5, a), (9, e), (2, p), (1, z)

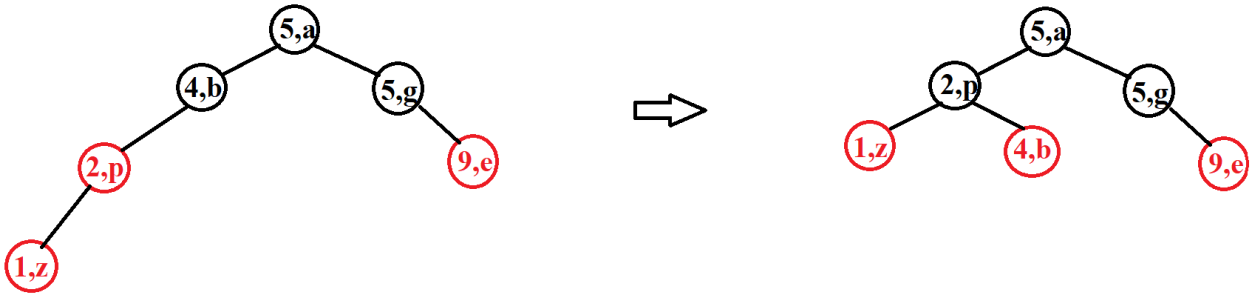
-Find: (9, e), (7, a)

-Delete: (5, g), (2, p)

- **OUTPUT:**

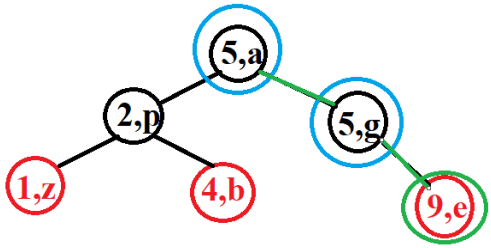
- Insert:



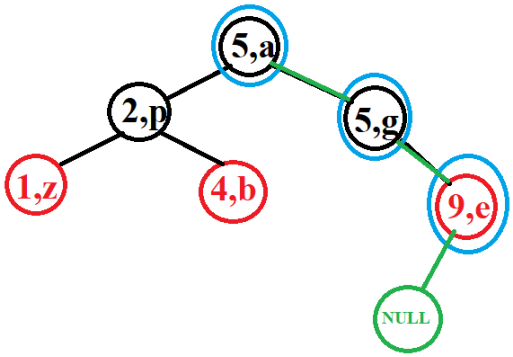


- Find:

Find (9, e)

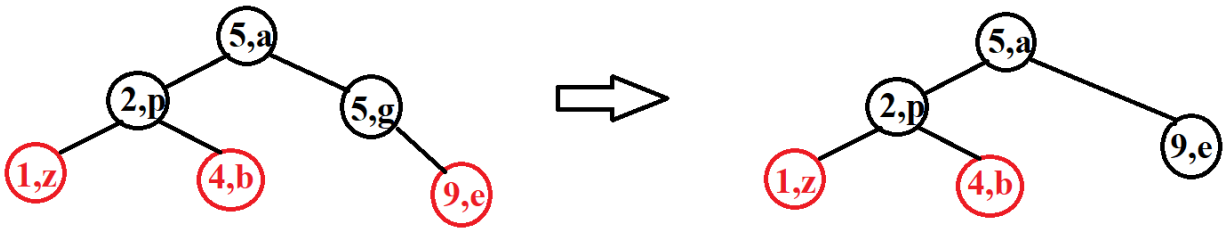


Find (7, e)

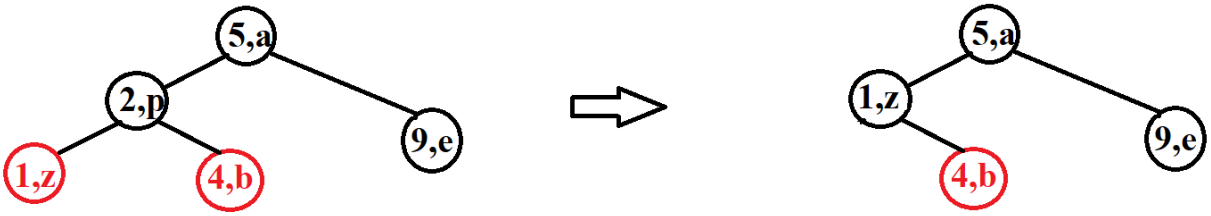


- Delete:

Delete (5, g)



Delete (2, p)



IV. Compare AVL and RB tree:

- Search operations of AVL trees are faster than those of RB trees because they are more tightly balanced. Because it checks for it to correct the tree occurs with more frequency than AVL, for example in insert we see it will check 2 child nodes in a row so it will rotate more, leading to a tighter equalization tree.

- Red-black tree has faster insertion and deletion than AVL tree because RB tree has a tighter balance, so those two operations make RB tree faster than AVL.

- Storing in 1 node of AVL tree is less than RB tree because it will store 1 more data which is color.

- the rotations for an AVL tree are harder to implement and debug than that for a Red-Black tree.

- Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++.

Lesson 6: Multi-way Tree

Contents:

I. Introduction:

A *multi-way tree* or *m-way tree* is a generalized version of the binary search tree. We know that in a binary search tree (BST), each node contains a distinct key (value) and two pointers point to its left and right child nodes.

Each node in m-way tree:

- Can contain at most m – 1 keys and m addresses of m child nodes.

- The keys are distinct and in ascending order.

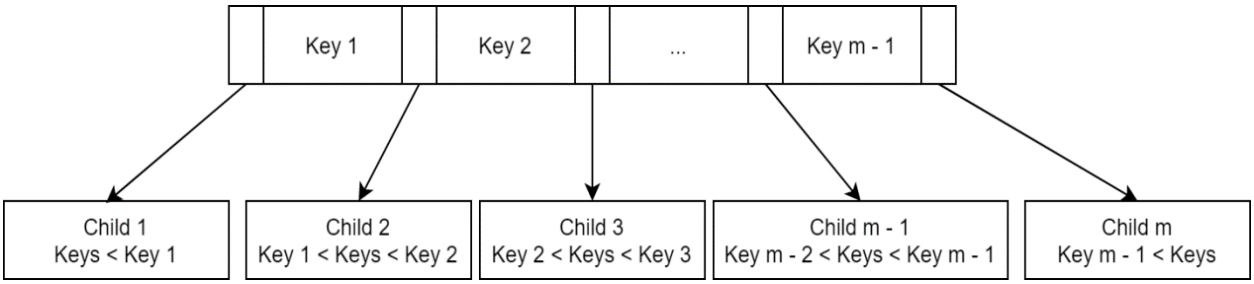
- The keys in the 1^{st} child node must be smaller than the 1^{st} key.

- The keys in the m^{th} child node must be larger than the $(m – 1)^{th}$ key.

- The keys in the i^{th} child node must be smaller than the i^{th} key and larger than the $(i – 1)^{th}$ key ($2 \leq i \leq m - 1$).

- The child nodes can be empty.

Imagine a node in this form:

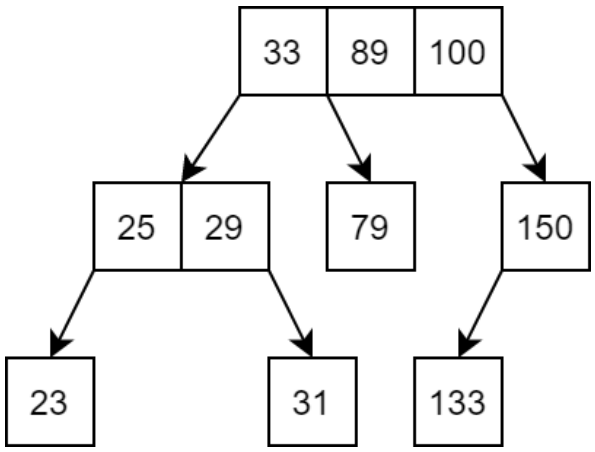


We call child[i] left child of key[i], and child [i +1] its right child.

A BST is basically a 2-way tree. But an unbalanced BST can waste a lot of memory, and even if balanced, the height of the tree can still be very high if the number of keys is large. Also, the implementation of a balanced BST is very complex. The m-way tree reduces height of the tree but take more time in traversal, in other word it slows down the data accessing process because the branching system is not as simple as in BST. In particular, we only make a 2-way branching decision at each node in BST, but in m-way tree, we must make a multiway branching decision.

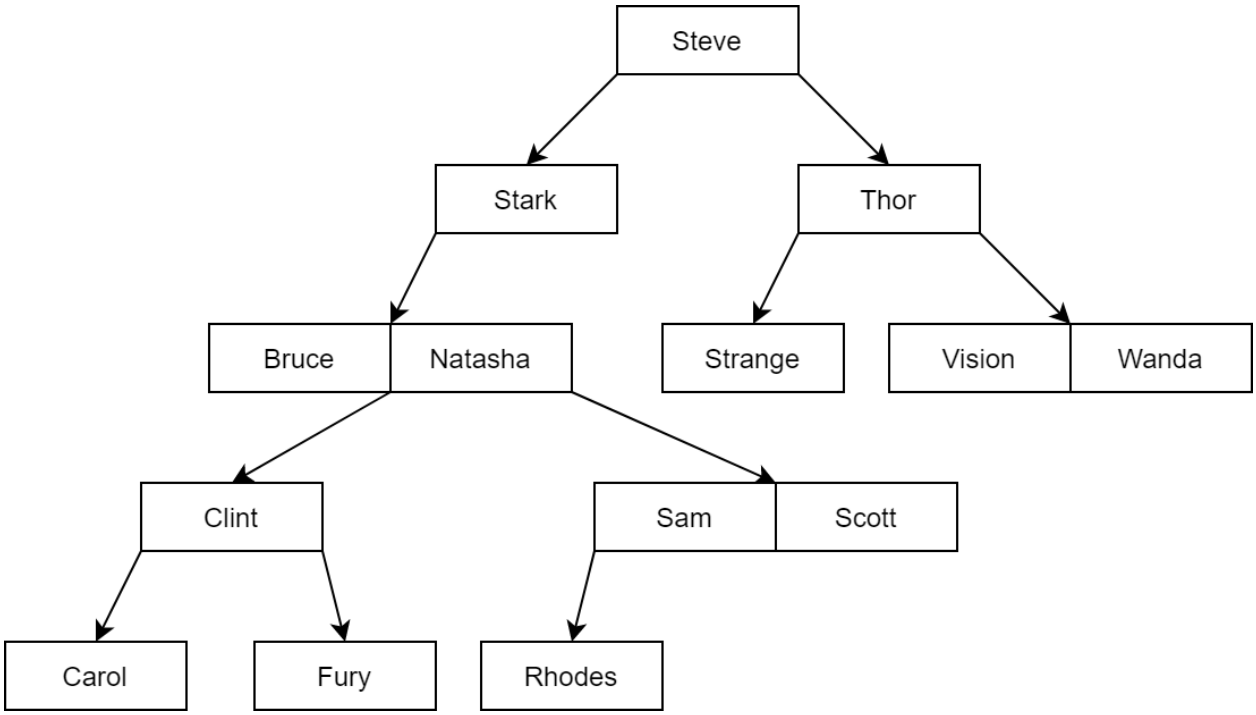
***Example 1:**

4-way tree example (integer keys):



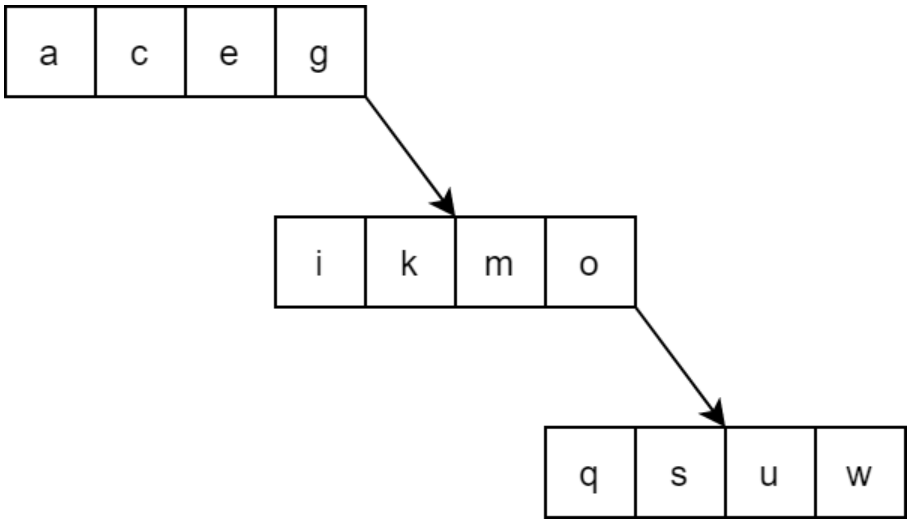
***Example 2:**

3-way tree example (string keys): using string compared to



***Example 3:**

5-way tree example (character keys):



This is an example of an unbalanced m-way tree. This case usually happens when we add keys to the tree in sorted order.

Let WAY be the maximum number of branches in a m-way tree node, the node structure will be defined below:

```
struct node
{
   int count;
   int value[WAY – 1];
   struct node* child[WAY];
};
```

Here, *count* store the number of values stored in the current node. All values in current node are stored in the array *value*, and the array *child* store addresses of every child node of the current node.

Let's also define a function to check if a node is full and another to create a new node, which will help later:

```
bool isFull(node* curr)
{
   return (curr->count == WAY - 1) ? 1 : 0;
}
```

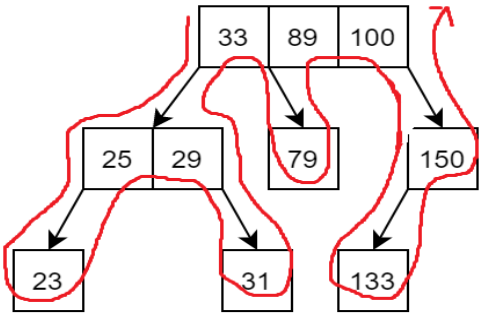
```
node* createnode()
{
   node* newnode = new node;
   newnode->count = 0;
   for (int i = 0; i < WAY; i++) newnode->child[i] = NULL;
   return newnode;
}
```

II. Operations on Multi-way Tree:

1. Traversing and searching operations:

Traversing an m-way tree is similar to in-order traversing in BST. We start at the left-most child, then the key and the next child, repeat for all remaining keys and child.

***Example:**



Searching in an m-way tree consist of two processes: traversing between nodes and searching in node. Traversing between nodes is similar to BST traversal, the difference is now we have m option to branch instead of two. Searching in node is to consider every value in the current nodes.

We consider a second function called *search_in_node*, which will determine if the key we are looking for is in the current node or not, and for both cases, return value *pos* which is either the position of key in current node if found, or position of the child node that might contain key. For example, *pos* will be i if value [i – 1] < key < value[i], or i if value[i] = key. The main search function will then call recursive to search the corresponding child node.

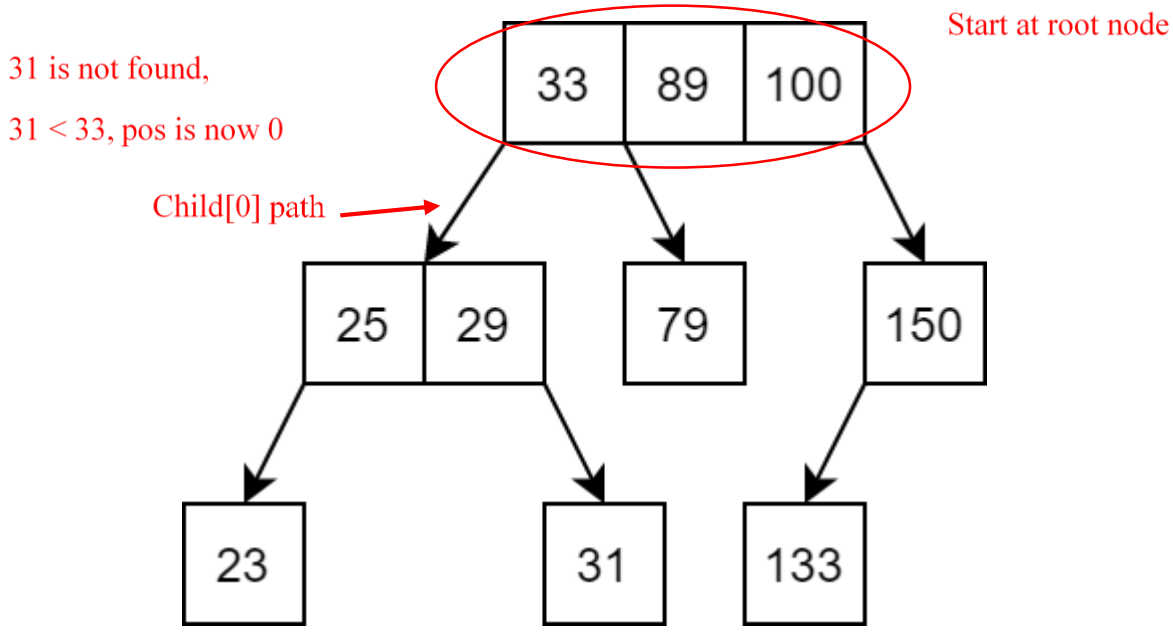
```
bool search_in_node(int key, node* curr, int& pos)
{
   pos = 0;
   // find the first key in node that is >= key,
   while ((pos < curr->count) && (key > curr->value[pos])) pos++;
   // if key > all values in node return 0, pos is position of the last child in node
   if (pos == curr->count) return 0;
   // if key is found, return 1,pos is now position of key in current node;
   if (key == curr->value[pos]) return 1;
   // else return 0, pos is now position of the child node that might contain key
   return 0;
}
```

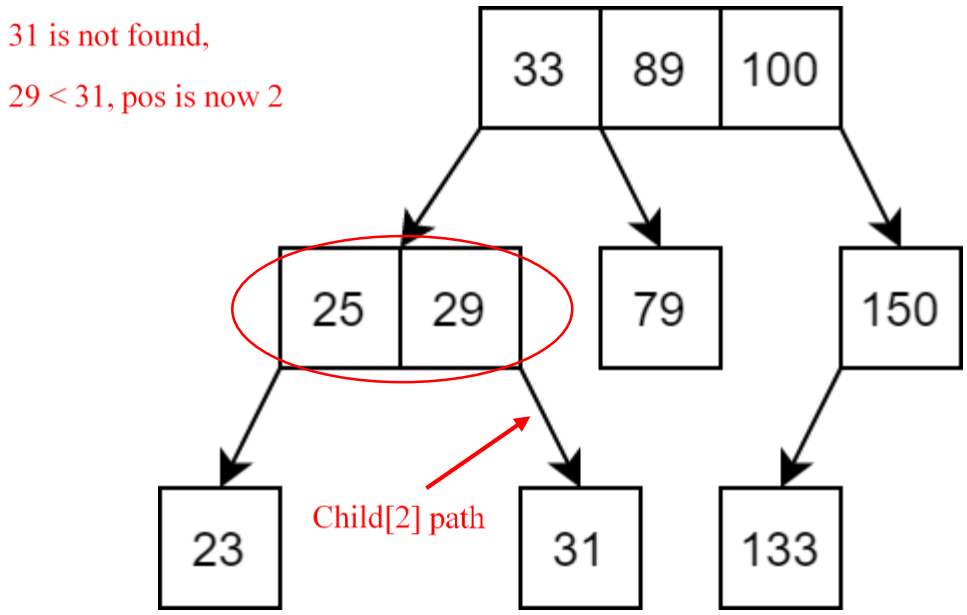
```
node* search(int key, node* curr, int& pos)
{
   // reach NULL node mean the key is not found in the tree, return NULL
   if (curr == NULL) return NULL;
   // if key is found in the current node return the current node
   // pos will be position of key in current node
   if (search_in_node(key, curr, pos)) return curr;
   // child[pos] is now the possible child to contain key, call recursive to this child
   return search(key, curr->child[pos]);
}
```

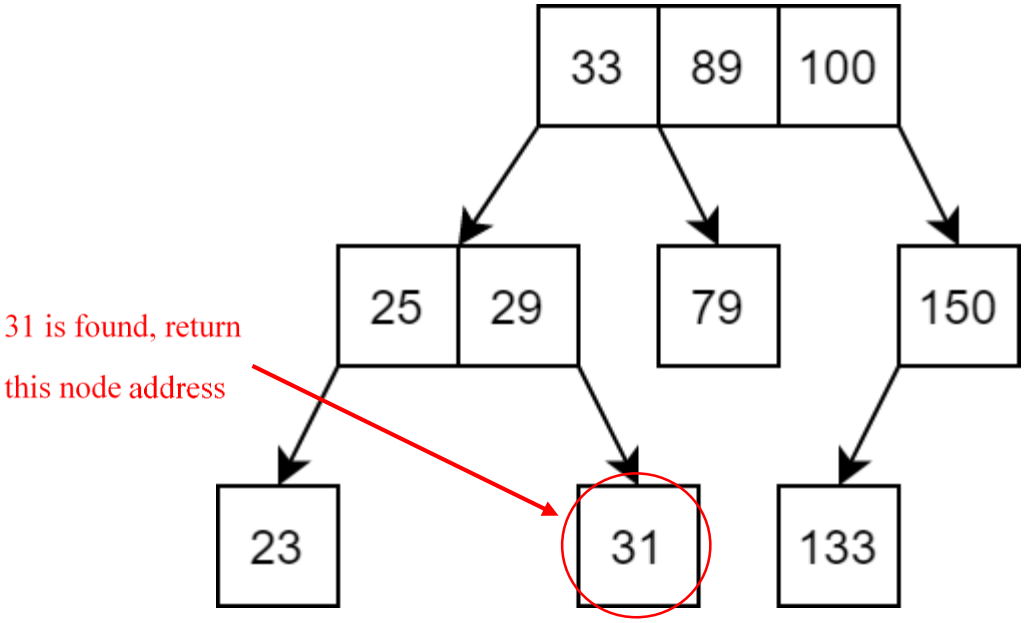
Calling the search function for the root node of a m-way tree will return address of the node contain the key being look for, or NULL if the key is not found.

Note that if the searching algorithm above is used, it can only return the node holding the key, not its exact position in that node.

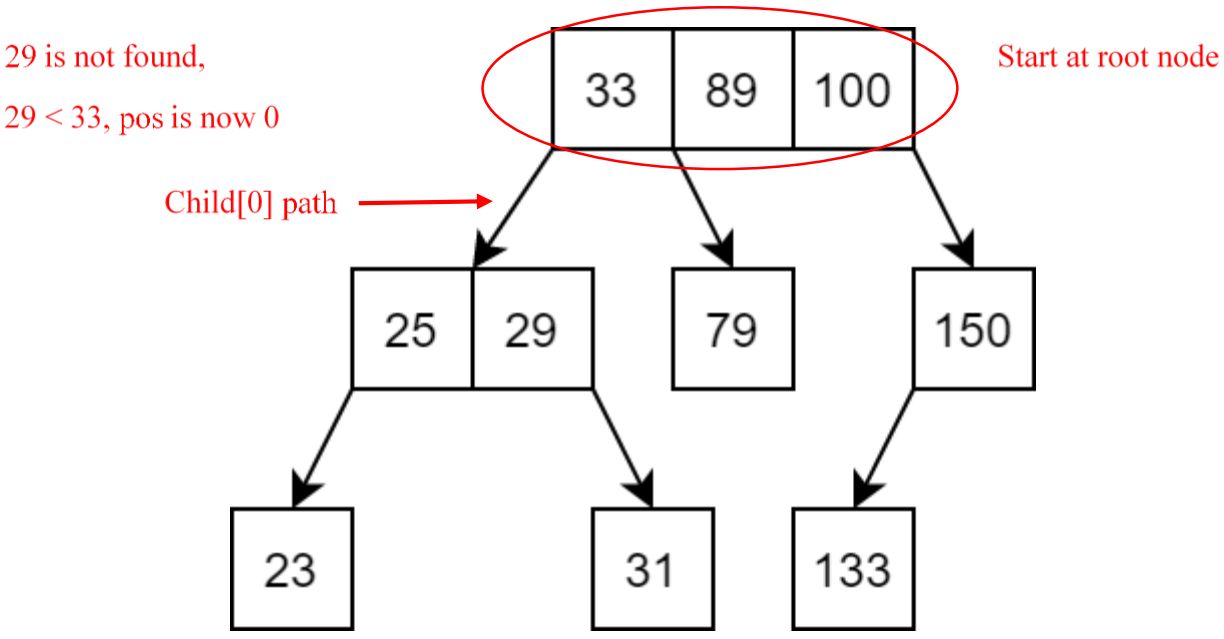
***Example:** Let's search for 31 in example 1.

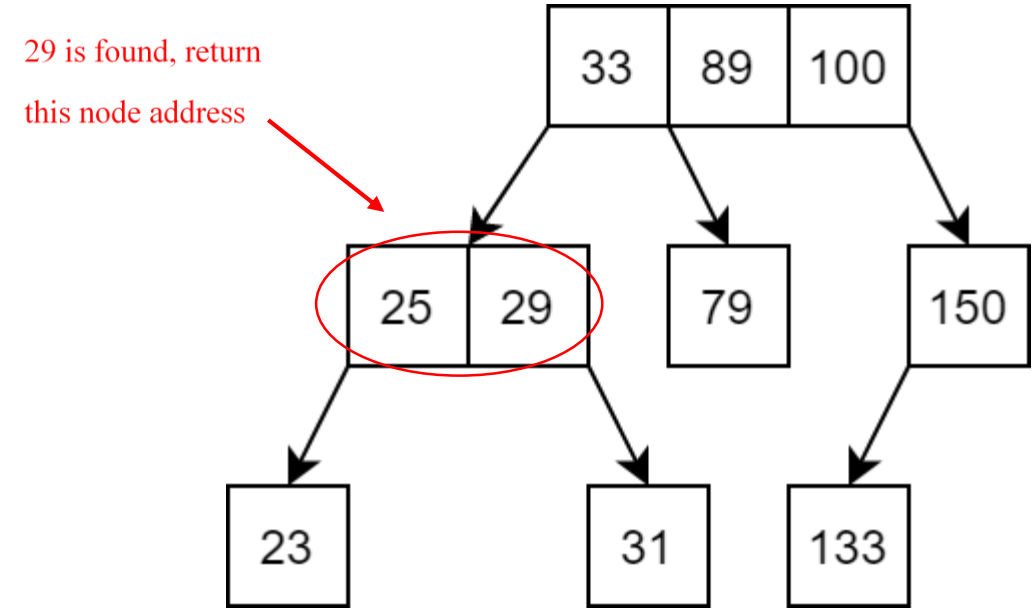




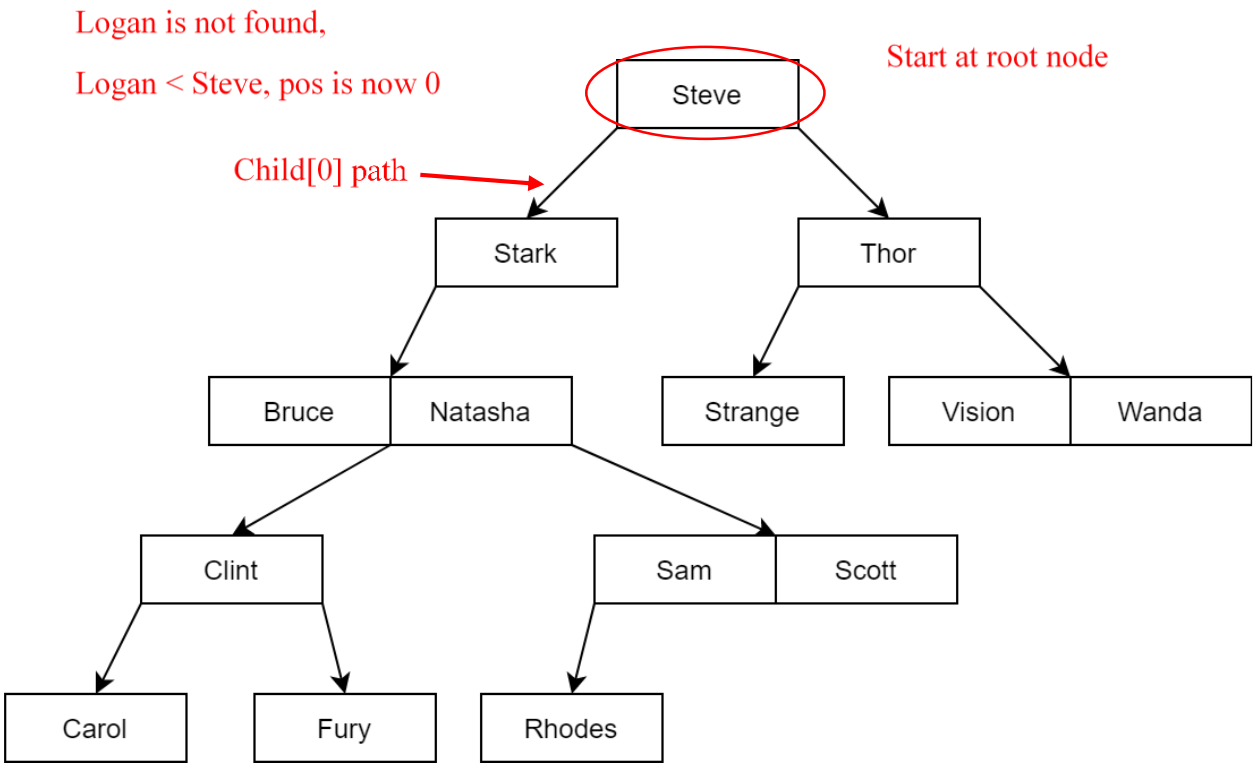


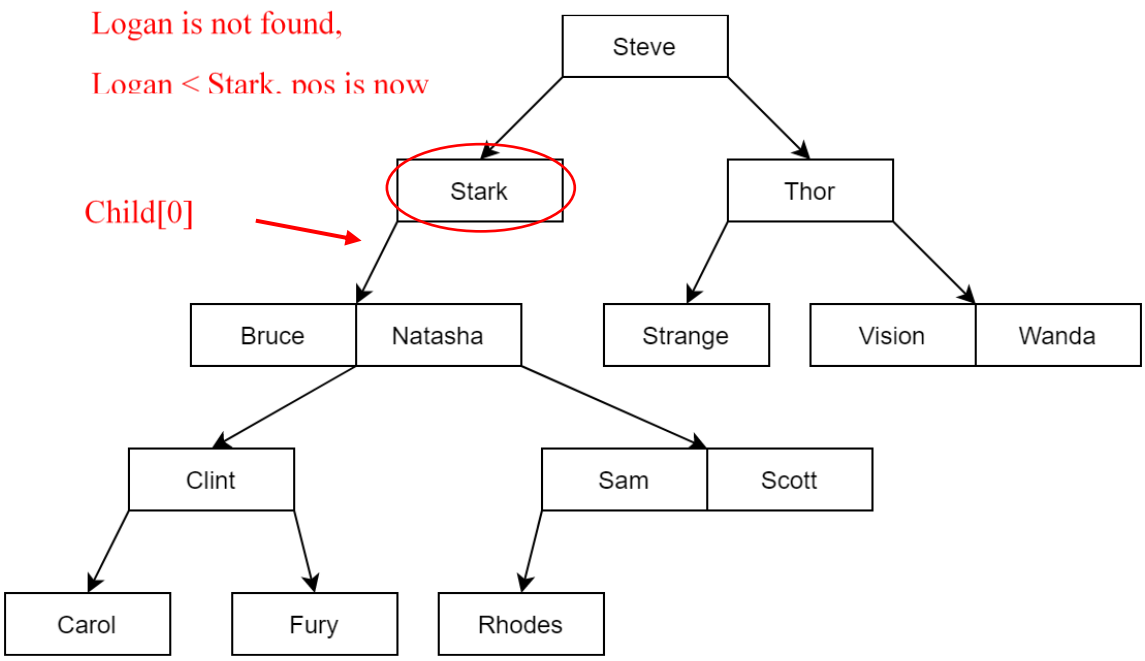
Let's try again with 29.

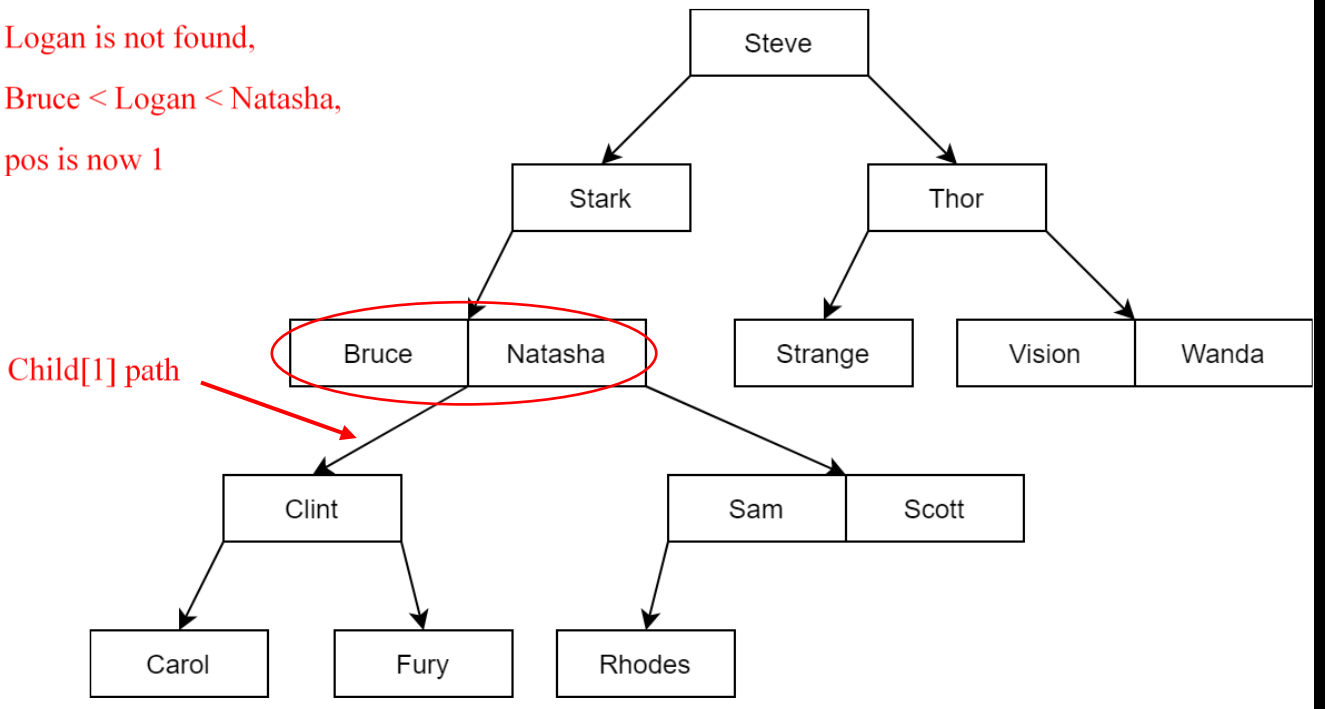


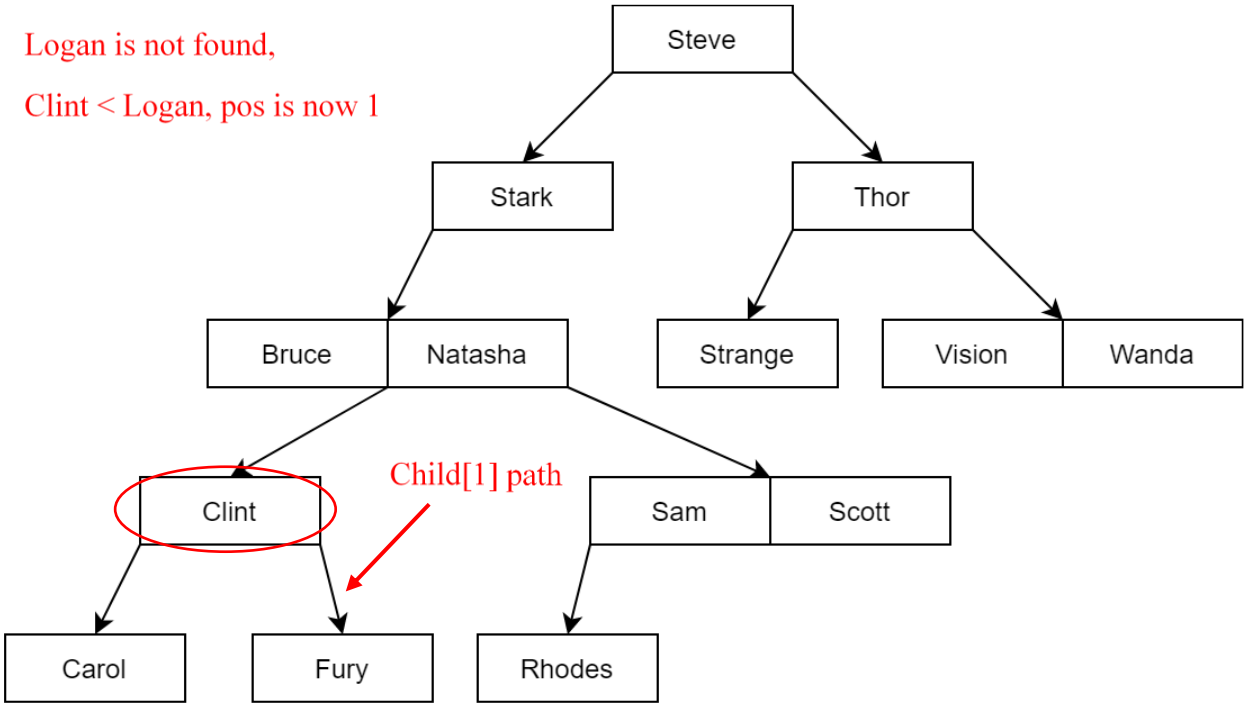


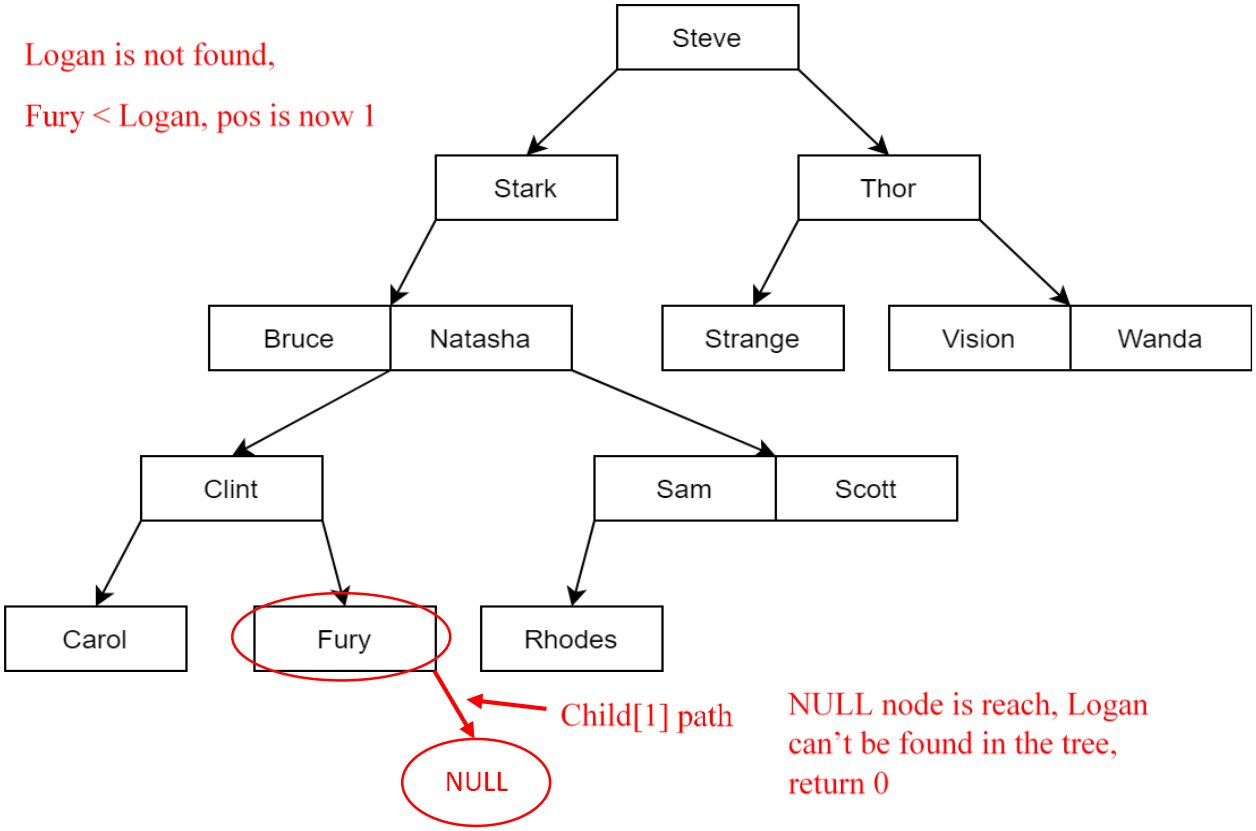
Now let's look for Logan in example 2:




Logan is not found,
Logan < Stark, pos is now
Child[0]
Steve
Stark
Thor
Bruce
Natasha
Strange
Vision
Wanda
Clint
Sam
Scott
Carol
Fury
Rhodes


Logan is not found,
Bruce < Logan < Natasha,
pos is now 1
Child[1] path
Steve
Stark
Thor
Bruce
Natasha
Strange
Vision
Wanda
Clint
Sam
Scott
Carol
Fury
Rhodes


Logan is not found,
Clint < Logan, pos is now 1
Steve
Stark
Thor
Bruce
Natasha
Strange
Vision
Wanda
Child[1] path
Clint
Sam
Scott
Carol
Fury
Rhodes


Logan is not found,
Fury < Logan, pos is now 1
Steve
Stark
Thor
Bruce
Natasha
Strange
Vision
Wanda
Clint
Sam
Scott
Carol
Fury
Rhodes
Child[1] path
NULL
NULL node is reach, Logan can't be found in the tree, return 0

2. Insertion operation:

To insert a new value into an m-way tree, we follow these steps:

- Traverse the tree until reaching a NULL node. We reuse the *search_in_node* function and the recursive method in searching algorithm.

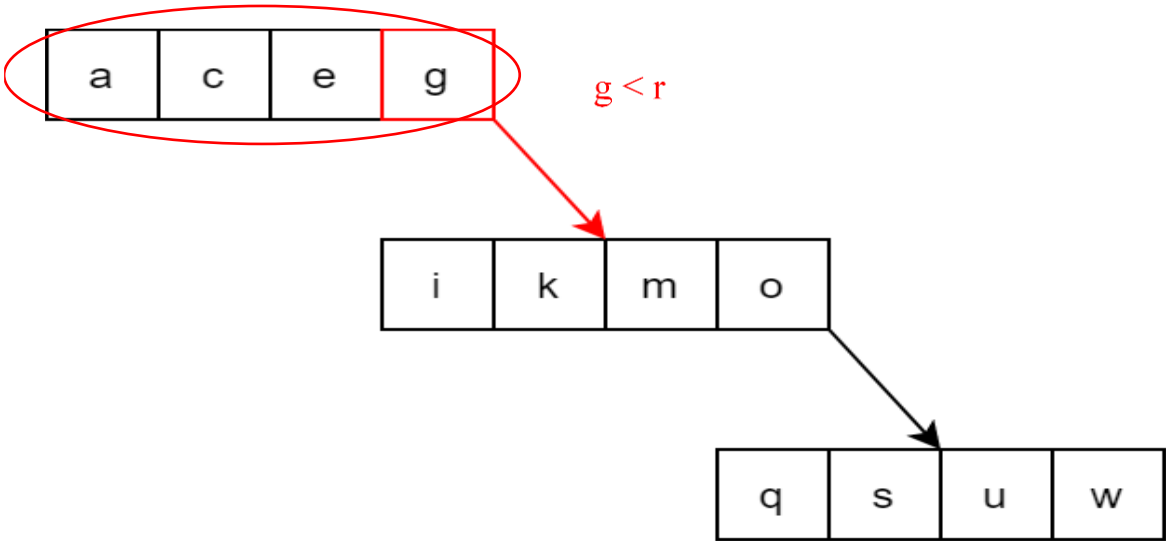
- If the parent of the NULL node is not full, insert the new key into the parent node.

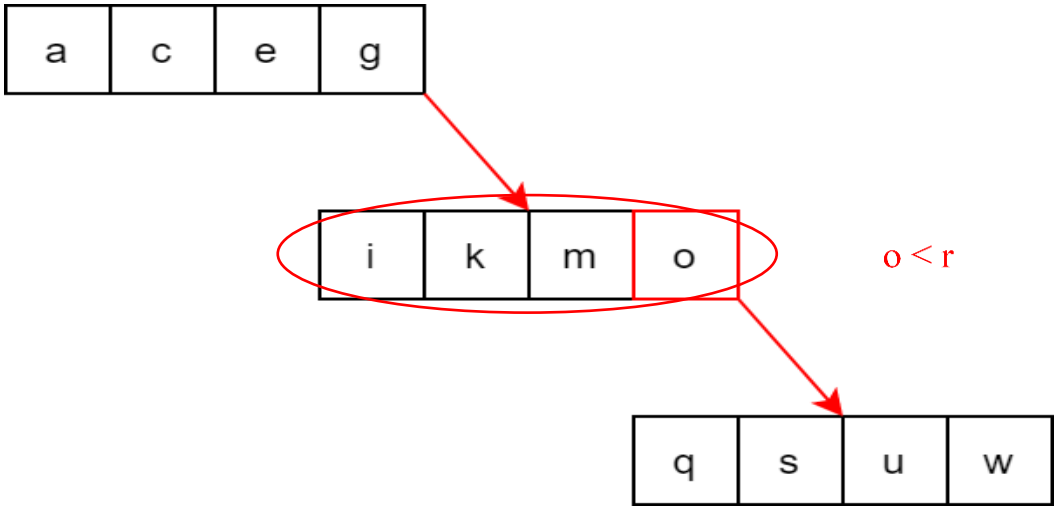
- Otherwise, create a new node and insert the new key to this new node.

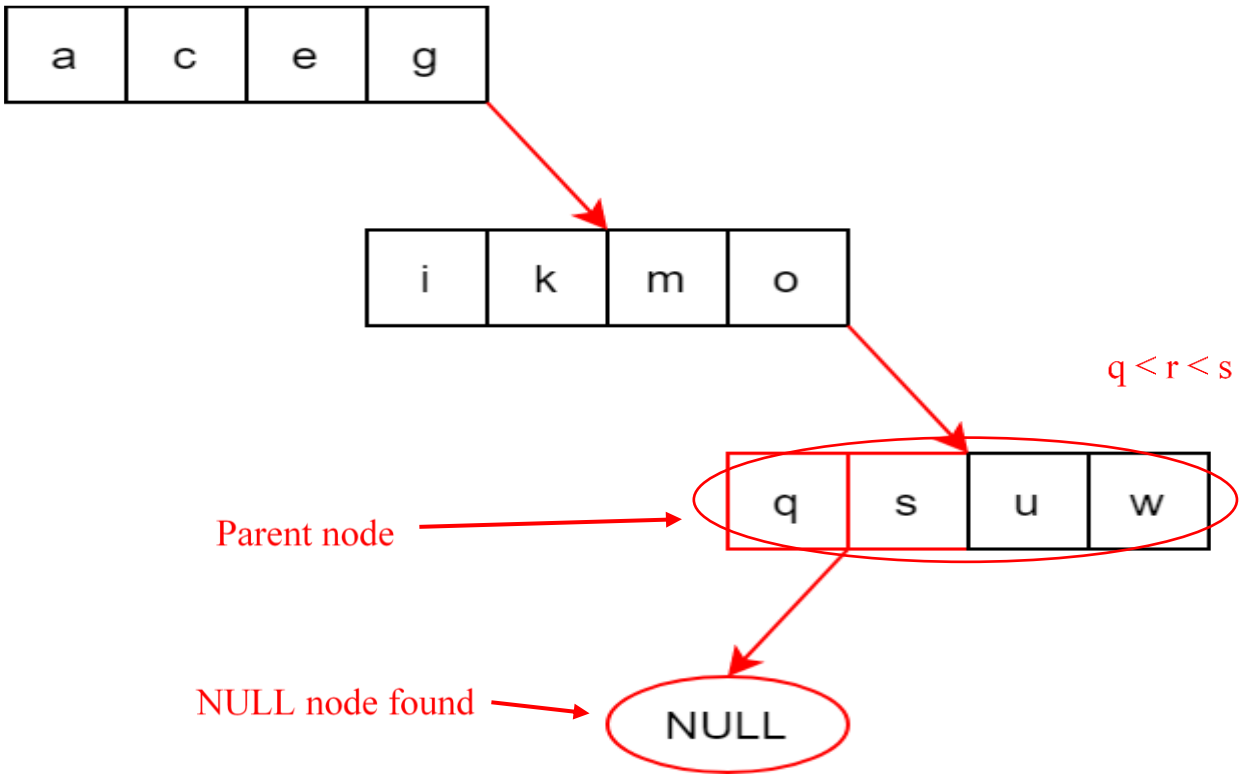
***Example:**

Let's try inserting r into example 3:

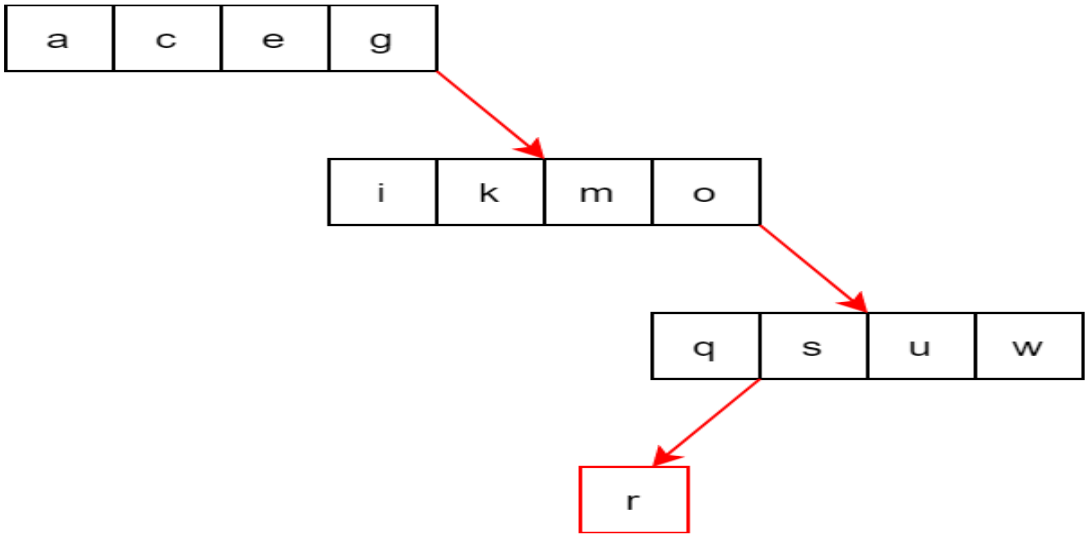
First, we traverse the tree to find a NULL node. This step is done by "searching" for r in the tree. With that, we traverse the tree following the red path:




a
c
e
g
i
k
m
o
o < r
q
s
u
w

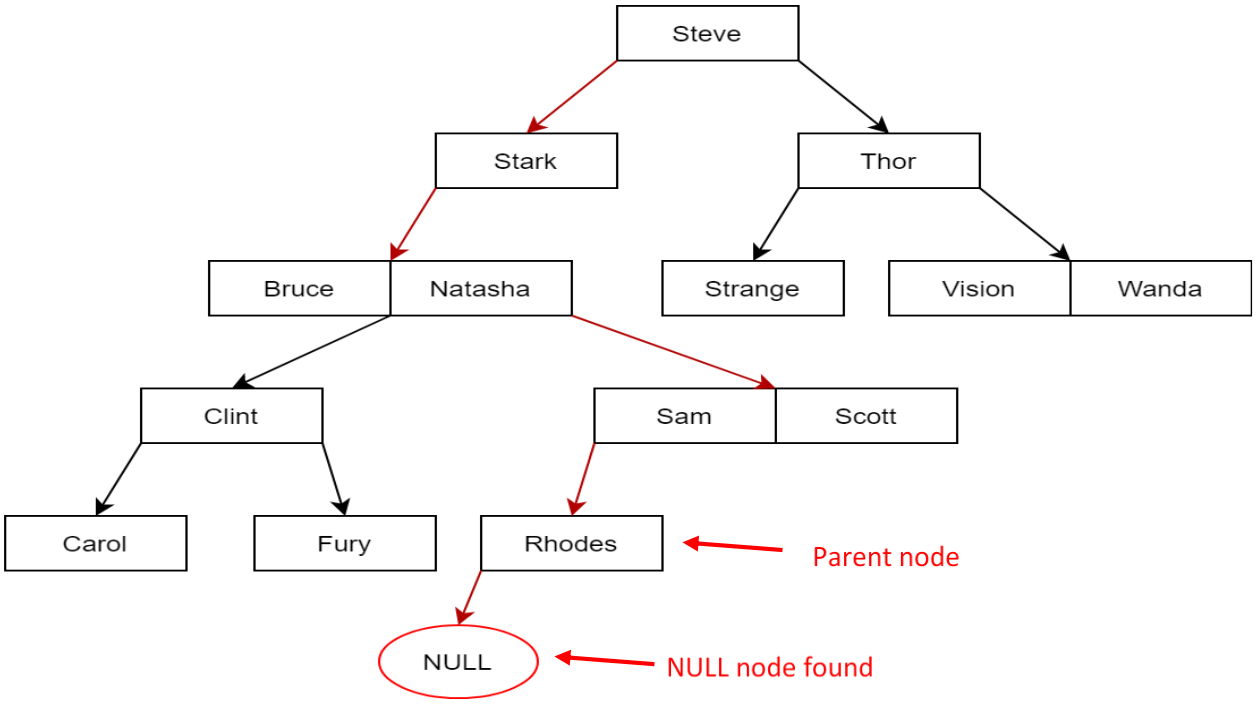

a
c
e
g
i
k
m
o
q < r < s
Parent node
q
s
u
w
NULL node found
NULL

The parent node is full, so we create a new node in position of the NULL node and add r to it. The new node only has one value.

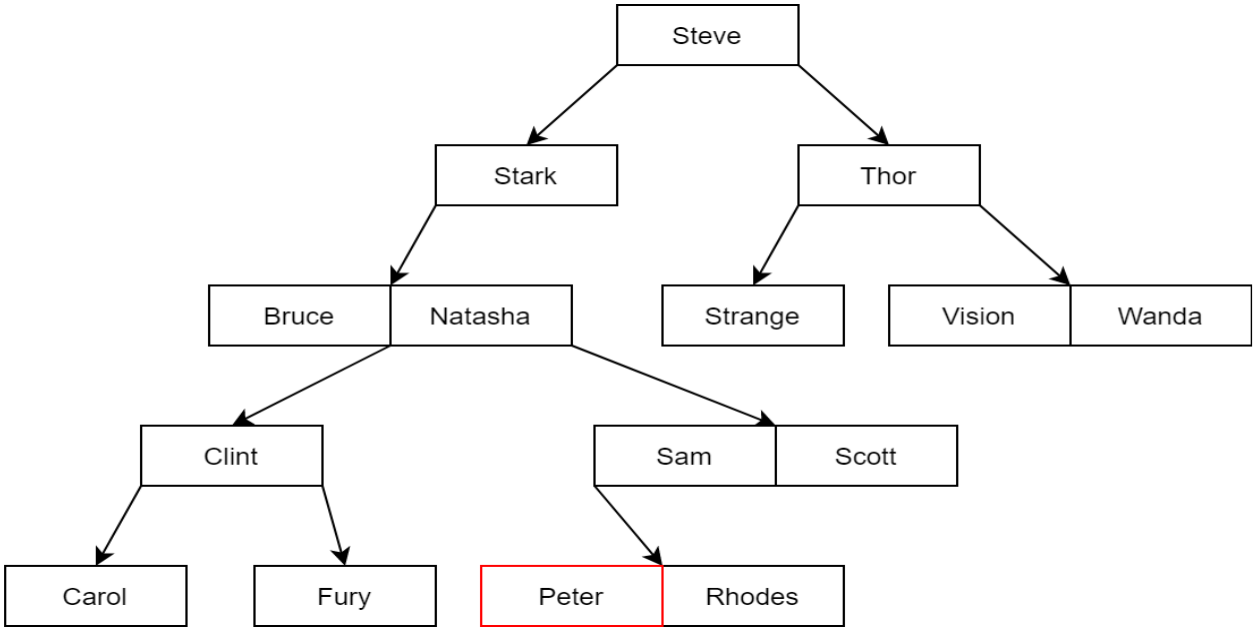


Let's add Peter to example 2:

First traverse the tree following the red path to reach a NULL node.

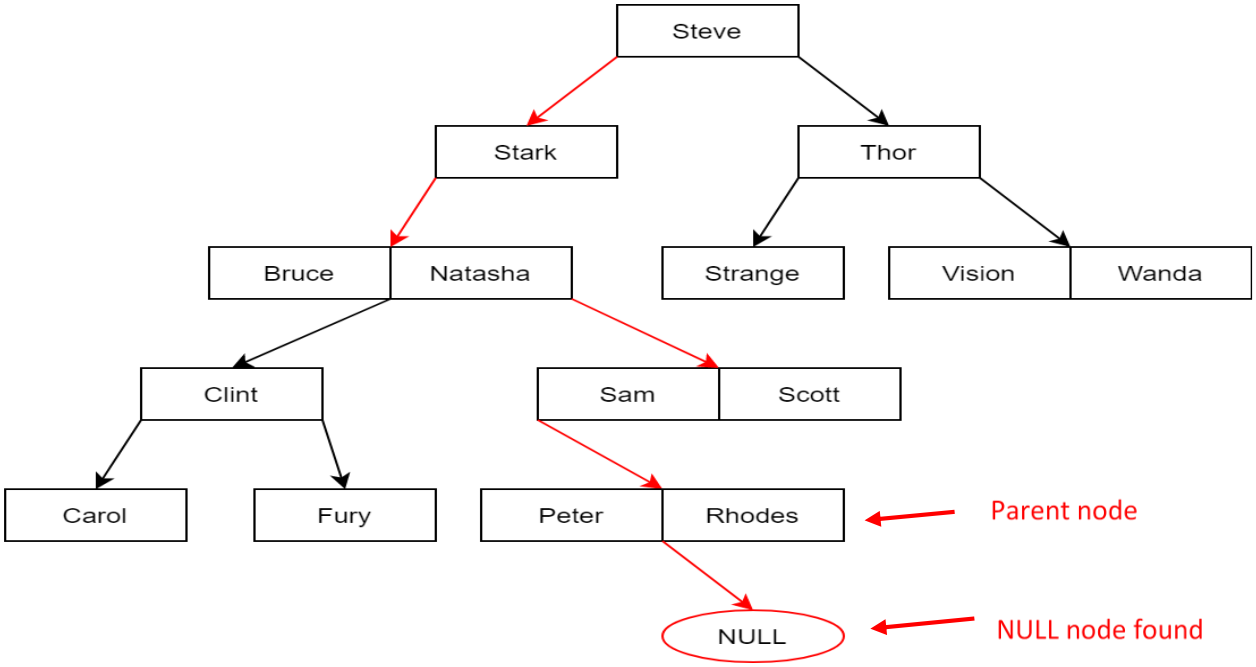


Now we have reached a NULL node, but its parent node is not full, so we insert Peter into the parent node. Peter < Rhodes, so he is inserted in the left of Rhodes.

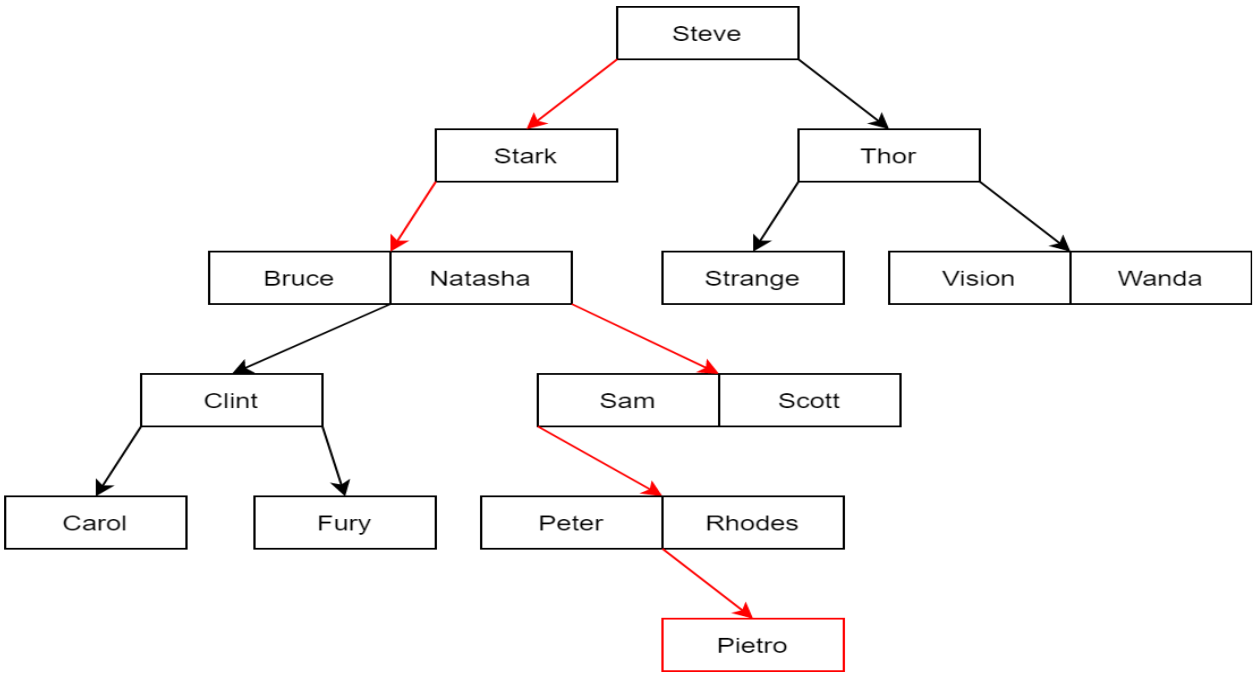


Also add Pietro to example 2:

Again, we traverse the tree to reach a NULL node.



The NULL node is found at the same level with the same parent node in the previous example, but now, the parent node is full, so we must create a new node for Pietro.



The insertion operation is implemented in C++ below:

```
bool insert_node(node*& curr, int key)
{
   // if tree is empty
   if (curr == NULL)
   {
      curr = createnode();
      curr->value[0] = key;
      curr->count = 1;
   }

   int pos;
   // if the new key is already in current node, which mean in the tree, do not insert
   if (search_in_node(key, curr, pos)) return 0;

   // reach a NULL node
   if (curr->child[pos] == NULL)
   {
      if (isFull(curr))
      {
         // if parent node is full
         curr->child[pos] = createnode();
         curr->child[pos]->count = 1;
         curr->child[pos]->value[0] = key;
      }
      else
      {
         // otherwise insert key into parent node
         // shift all value > key to the right to create a slot
         for (int i = curr->count; i > pos; i--)
         {
            curr->value[i] = curr->value[i - 1];
            curr->child[i + 1] = curr->child[i];
         }
         curr->child[pos + 1] = curr->child[pos];
         // insert key into created slot
         curr->value[pos] = key;
         curr->child[pos] = NULL;

         // update count value
         curr->count++;
      }
      // successfully insert
      return 1;
```

```
    }
    else
    {
        // call recursive to the child node
        return insert_node(curr->child[pos], key);
    }
}
```

3. Deletion operation:

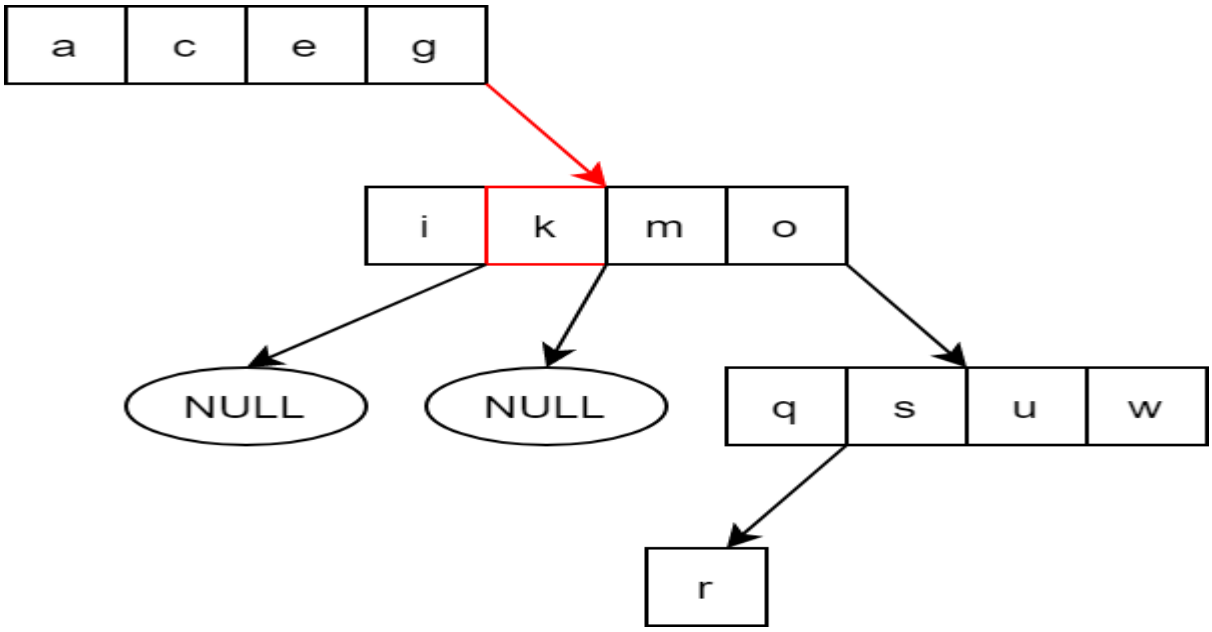
To remove a key from an m-way tree, we follow these steps:

- Traverse the tree to find the node holding key.
- If key doesn't have both left and right child, we remove key from current node by moving all value after key and their child one slot forward.
- If key has either left or right child, we replace key with the largest value in its left sub tree, or the smallest value in its right sub tree. Then, we call recursive to remove the replaced value from its node.
- If a node become empty, we delete it.

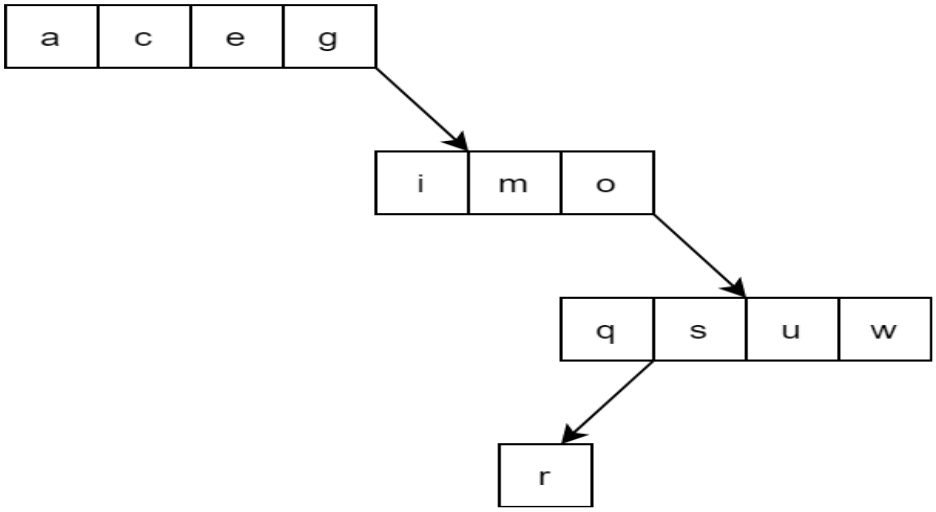
***Example:**

Let's remove k from example 3:

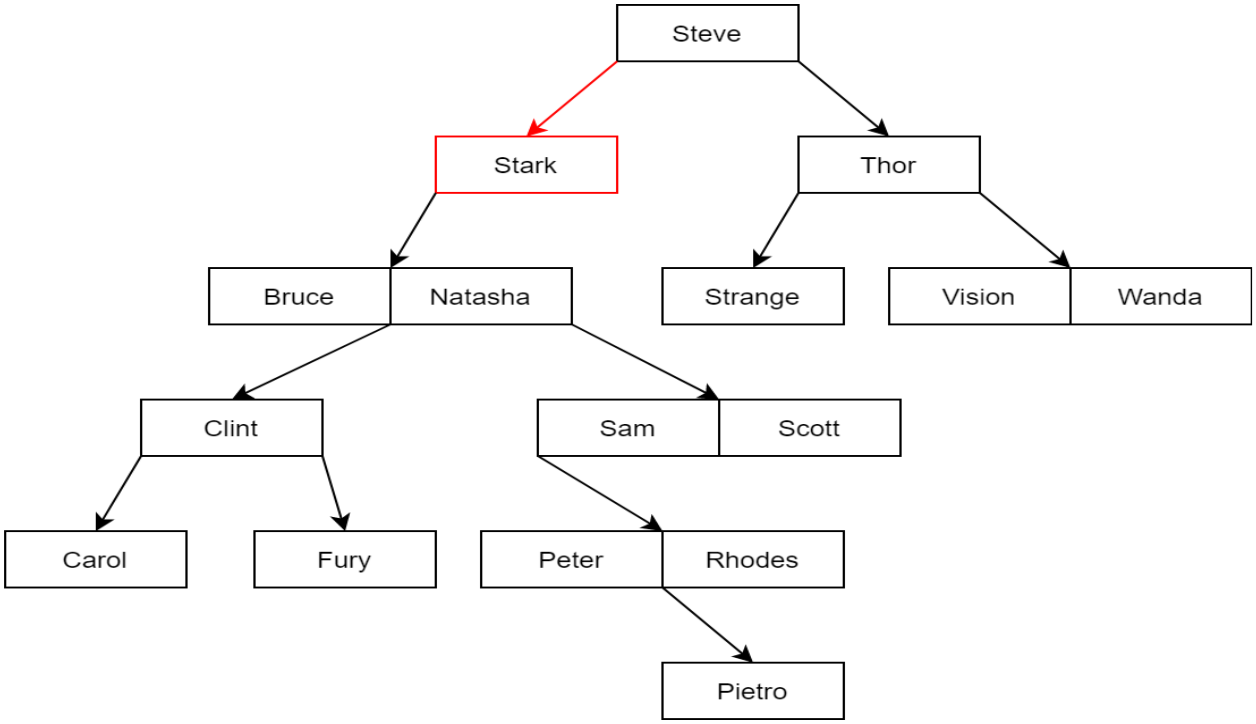
First, we find the node holding k:



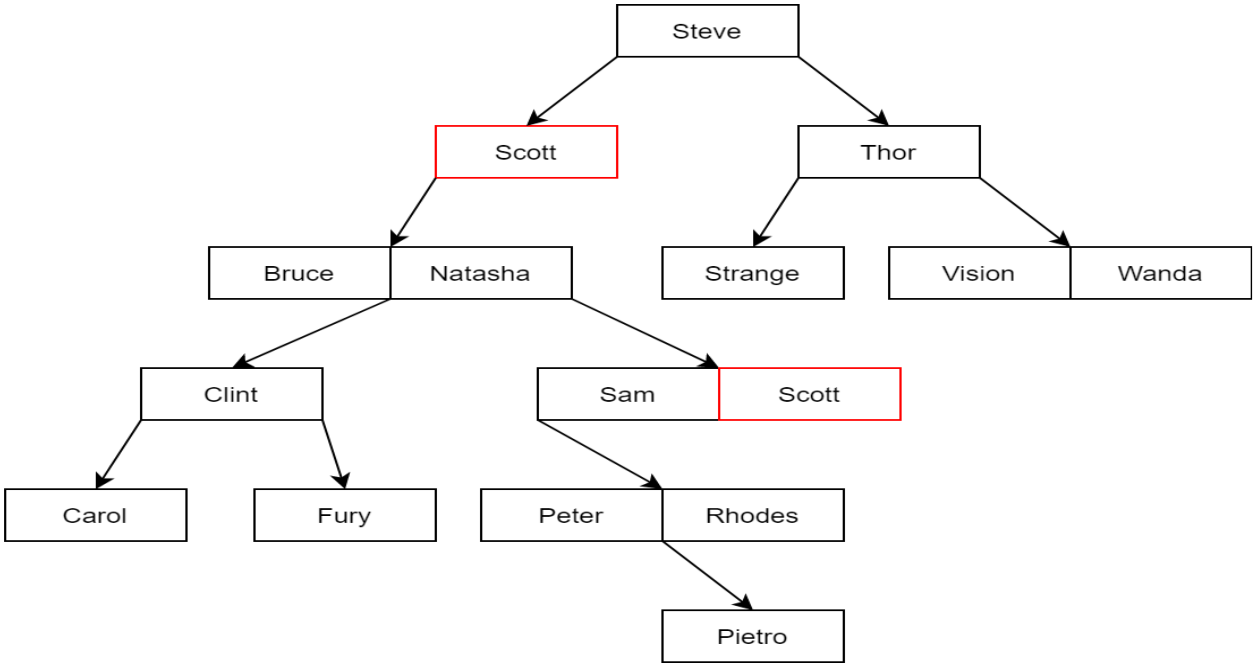
We see that k doesn't have both left and right child, so we just remove it from its node.



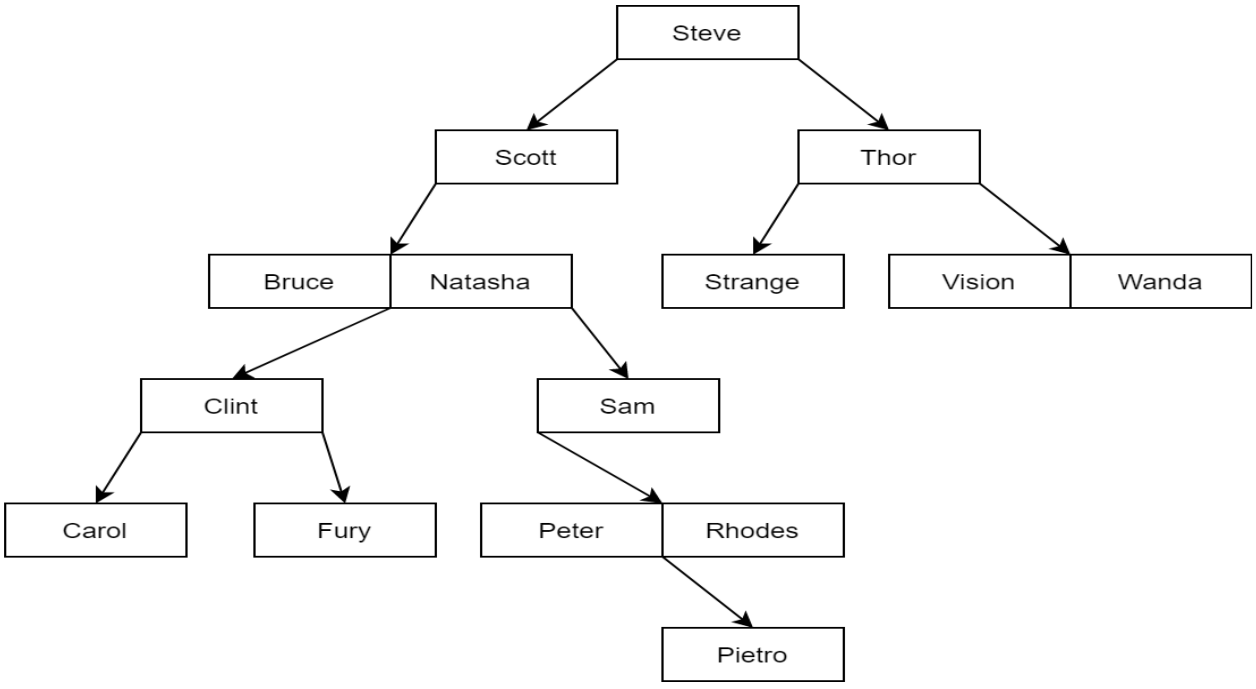
Remove Stark from example 2: First we find Stark.



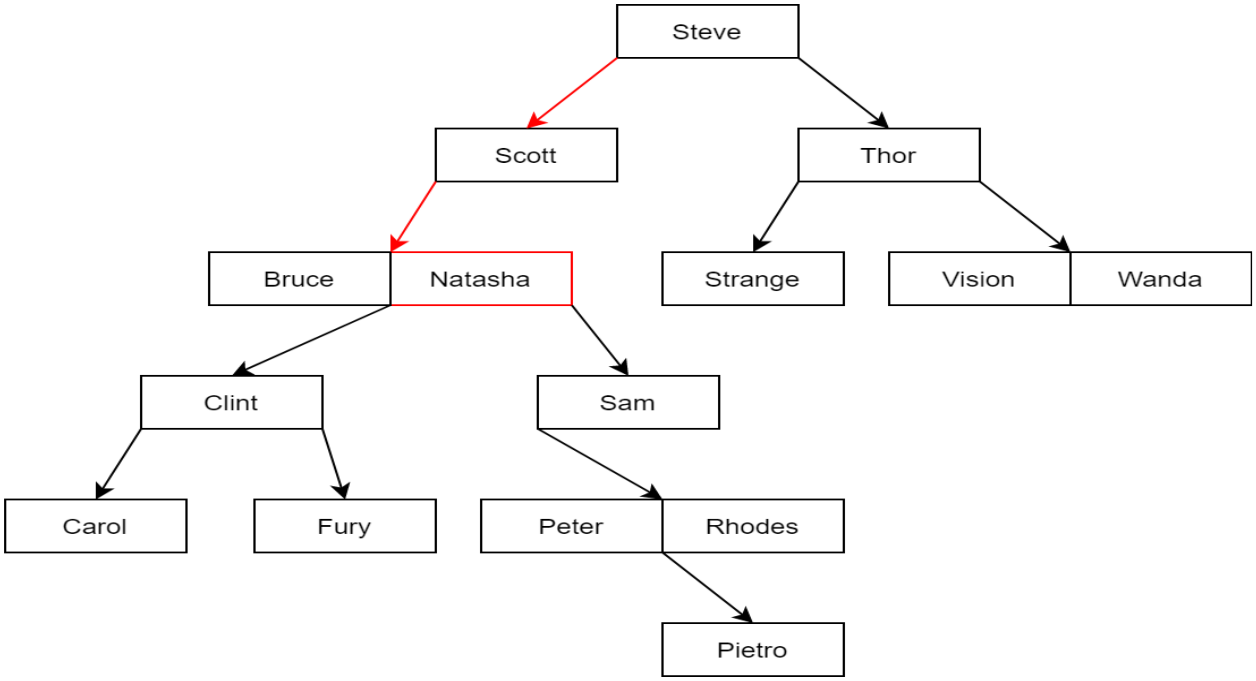
We notice that Stark has left child, so we replace Stark with the largest value in the left sub tree. Here, that value is Scott.



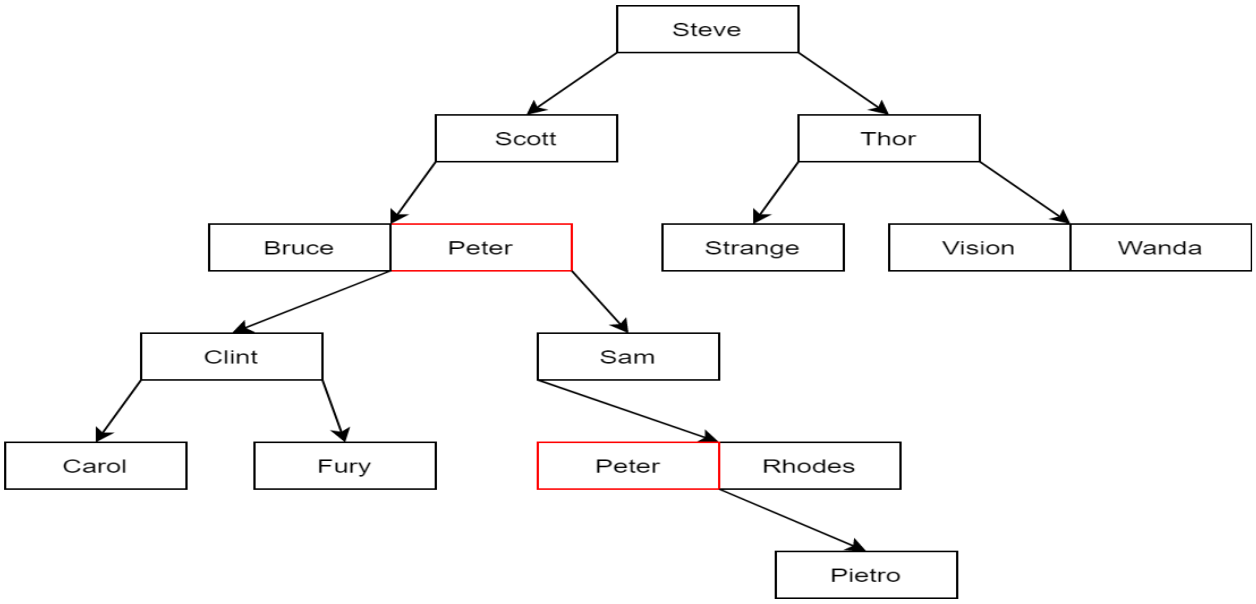
Now that Scott has been moved to Stark position, we remove Scott from his old position. Old Scott doesn't have any child, so we just remove him.



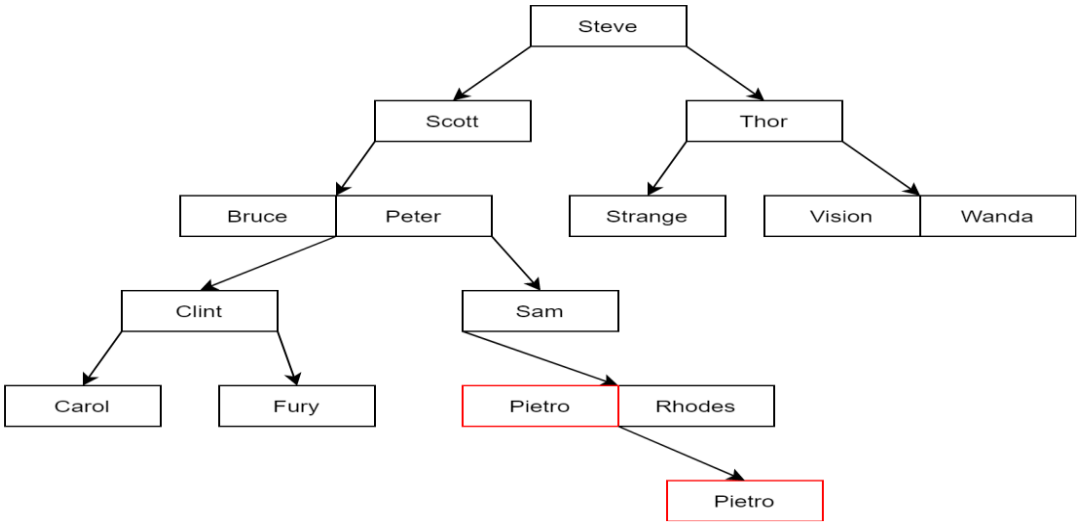
Let's try again with Natasha.



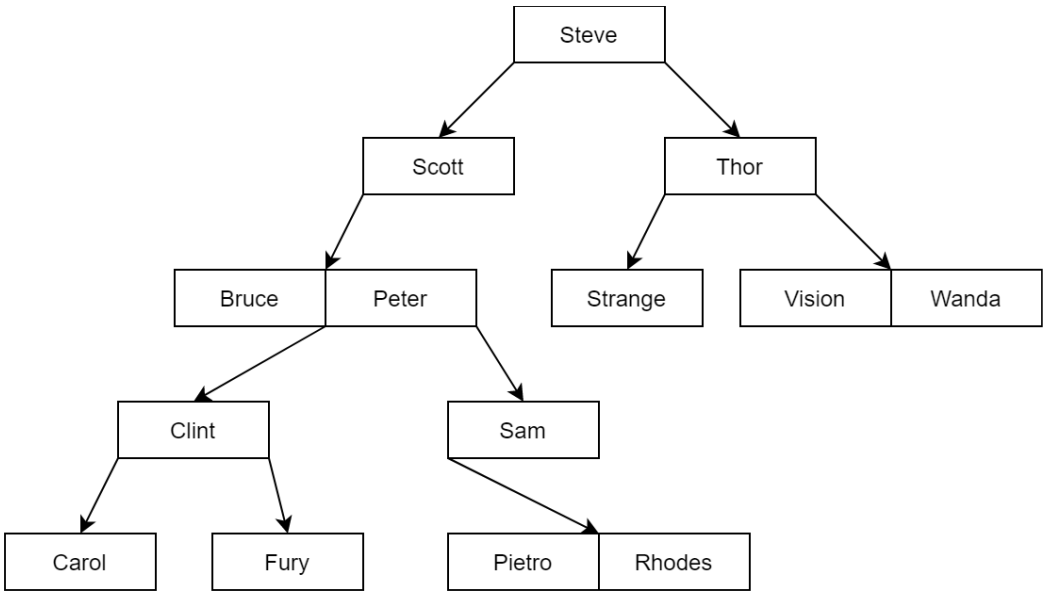
Natasha has both left and right child, we can choose either to replace her with the largest value in left sub tree, or the smallest one in right sub tree. This time, let follow the right path. The smallest value in the right sub tree is Peter.



Now we have to remove old Peter, but old Peter still has a right child, so we repeat and replace old Peter which the smallest and also the only child in right sub tree, Pietro.



Moving to remove old Pietro, he doesn't have any child and is also the only value in his node. After removing him, his node become empty, so we delete it.



The deleting algorithm will be implemented in C++ below:

```
node* left_replace(node *curr,node *&parent, int &replacekey)
{
   // find the largest value in this sub tree
   // get the position of the largest value in current node
   int pos = curr->count - 1;

   // if value[pos] still have right child, it is not the largest value
   if (curr->child[pos + 1] != NULL)
   {
      parent = curr;
      return left_replace(curr->child[pos + 1],parent,replacekey);
   }
   else
   {
      // else return the current node
      replacekey = curr->value[pos];
      return curr;
   }
}

node* right_replace(node* curr,node *&parent, int& replacekey)
{
   // the first value is the smallest value in current node
   // if the first value still have left child, it is not the smallest value
   if (curr->child[0] != NULL)
   {
      parent = curr;
      return right_replace(curr->child[0],parent, replacekey);
   }
   else
   {
      // else return the current node
      replacekey = curr->value[0];
      return curr;
   }
}


bool delete_node(node*& curr, int key)
{
   // if the function reach a NULL node, mean the key being look for is not in the tree,
   // or this is an empty tree
   if (curr == NULL) return 0;
```

```
int pos;
// if key is not in current node, call recursive to child node
if (!search_in_node(key, curr, pos))
{
   return delete_node(curr->child[pos], key);
}

// if key is found, value[pos] is now holding key
if (curr->child[pos] != NULL)
{
   // if key has a left child, replace key with the largest value in left sub tree
   int replacekey;
   node* parent = curr;
   node* alter_node = left_replace(curr->child[pos], parent, replacekey);
   // replace the current key need to be deleted
   curr->value[pos] = replacekey;
   // remove the replaced key in the alternative node
   bool temp = delete_node(alter_node, replacekey);
   // if alter node is empty, delete it
   if (alter_node == NULL)
   {
      if (parent == curr) parent->child[pos] = NULL;
      else parent->child[parent->count - 1] = NULL;
      delete alter_node;
   }
   return temp;
}
else if (curr->child[pos + 1] != NULL)
{
   // if key has a right child, replace key with the smallest value in right sub trees
   int replacekey;
   node* parent = curr;
   node* alter_node = right_replace(curr->child[pos + 1],parent, replacekey);
   // replace the current key need to be deleted
   curr->value[pos] = replacekey;
   // remove the replaced key in the alternative node
   bool temp = delete_node(alter_node, replacekey);
   // if alter node is empty, delete it
   if (alter_node == NULL)
   {
      if (parent == curr) parent->child[pos + 1] = NULL;
      else parent->child[0] = NULL;
      delete alter_node;
   }
```

```
        return temp;
    }
    else
    {
        // if key doesn't have both left and right child, remove key from current node
        // shift all value after key to the left
        while (pos < curr->count - 1)
        {
            curr->value[pos] = curr->value[pos + 1];
            curr->child[pos] = curr->child[pos + 1];
            pos++;
        }
        curr->child[pos] = curr->child[pos + 1];
        curr->child[pos + 1] = NULL;

        // update count value
        curr->count--;

        if (curr->count == 0) curr = NULL;
        // successfully delete
        return 1;
    }
}
```

Lesson 7: B-tree

Contents:

I. Introduction:

We know that implementing a balanced binary search tree is complex, and an m-way tree can still be unbalanced. To solve the problem, we will discuss about a balanced m-way tree.

B-tree is a self-balancing m-way tree. Let WAY be the maximum number of branches in a node. To keep itself balanced, a B-tree needs to satisfy:

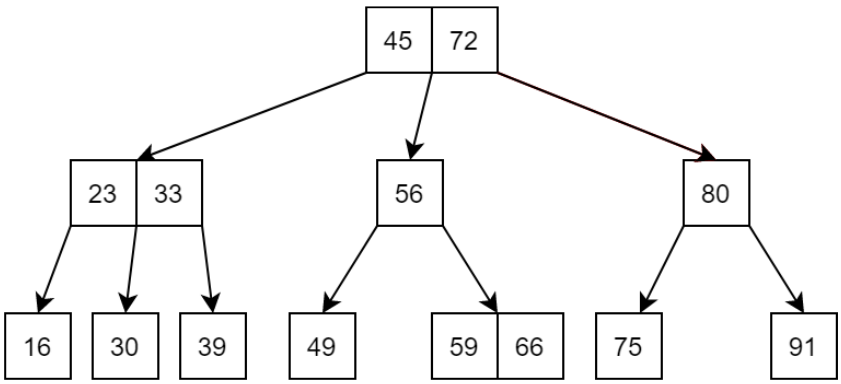
- The root node must have at least one key.

- All nodes except root node must contain at least [(WAY – 1)/2] key, or in other word, must have at least [(WAY – 1)/2] + 1 child nodes.

- All leaves nodes are at the same level, or we can say all NULL node are at the same level.

***Example:**

A 3-way B tree with integer values:



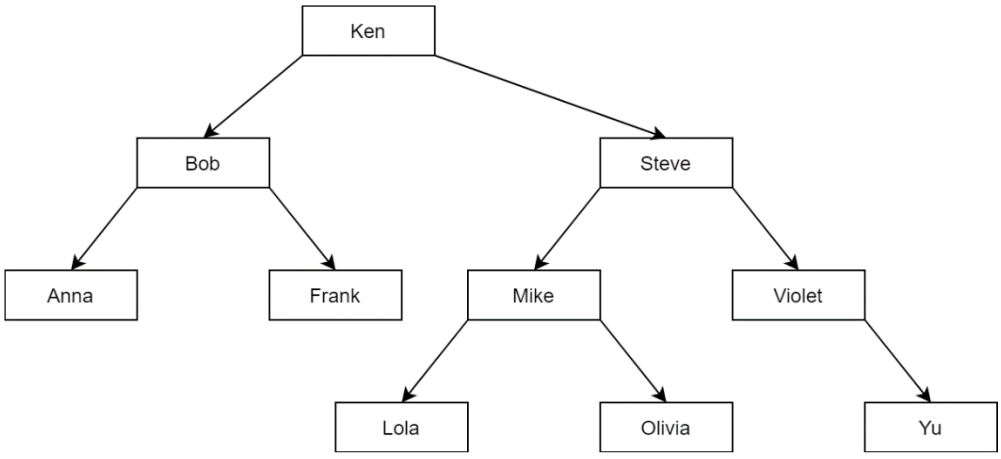
This tree sastifies:

- Root node has at least 1 key. ✓

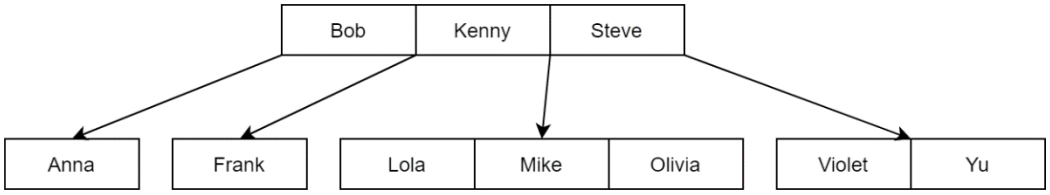
- All nodes except root node must have at least [(3 - 1)/2] = 1 key; [(3 – 1)/2] + 1 = 2 childs. ✓

- All leaves nodes are at the same level. ✓

A list of names organized as a balanced BST:



With only 10 values, the BST has grown to the height of 4. When with the same number of values, a 4-way B-tree reduce the height to 2.



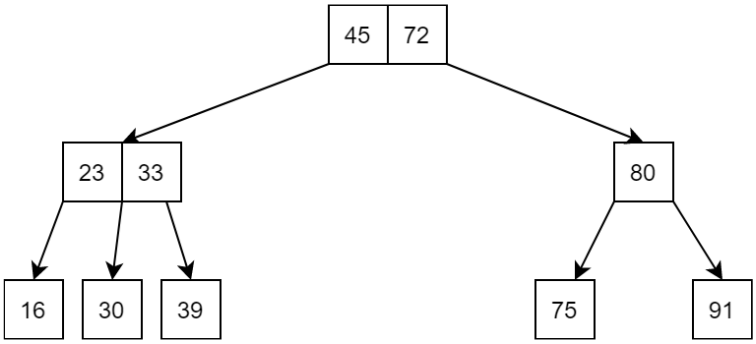
This tree sastifies:

- Root node has at least 1 key. ✓

- All nodes except root node must have at least [(4 - 1)/2] = 1 key; [(4 – 1)/2] + 1 = 2 childs. ✓

- All leaves nodes are at the same level. ✓

A 3-way B-tree?

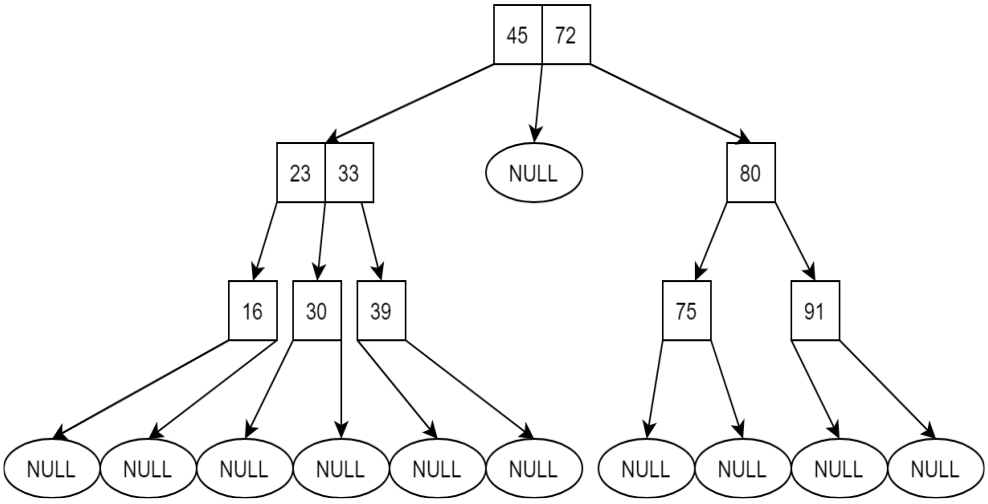


This tree sastifies:

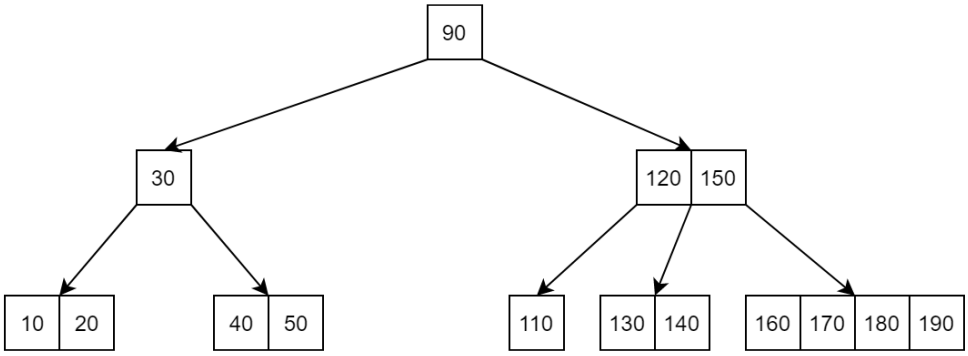
- Root node has at least 1 key. ✓

- All nodes except root node must have at least [(3 - 1)/2] = 1 key; [(3 – 1)/2] + 1 = 2 childs. ✓

- All leaves nodes are at the same level. ✗. The NULL nodes are not at the same level.



A 5-way B-tree?

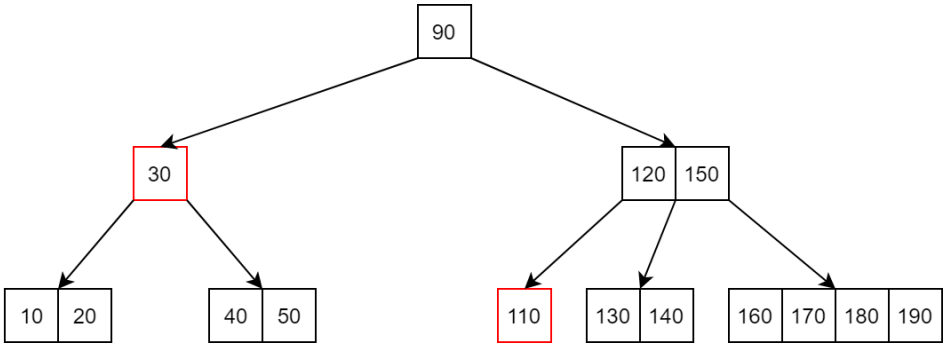


This tree sastifies:

- Root node has at least 1 key. ✓

- All leaves nodes are at the same level. ✓

- All nodes except root node must have at least [(5 - 1)/2] = 2 key; [(5 – 1)/2] + 1 = 3 childs. ✗. There are nodes with not enough keys and childs.



A B-tree node can be defined like an m-tree node, but let's add a boolean value to determine if a node is leaf node or not, and a parent node address for easy traversing. Note that this is not the best way to define the structure of a B-tree node, it is only used to clearly explain the idea of the algorithms later. In this structure, we add one extra slot to the *value* and *child* array, which we will used later.

```
#define WAY 3 // maximum number of branches in a node
#define MIN ((WAY - 1)/2) // minimum number of keys in a node

struct node
{
   int count;
   int value[WAY];
   struct node* child[WAY + 1];
   struct node* parent;
   bool isLeaf;
};
```

II. Operations on B-tree:

1. Traversing and searching operations:

Because B-tree is an m-way tree, the traversing and searching method is the same as m-way tree. So, we bring back the *search_in_node* function introducted in the m-way tree section. This function returns a boolean value and an integer value *pos*, they are either 1 and position of key if found in current node, or 0 and position of the child node that might contain key.

```
bool search_in_node(int key, node* curr, int& pos)
{
   pos = 0;
   // find the first key in node that is >= key,
   while ((pos < curr->count) && (key > curr->value[pos])) pos++;
   // if key > all values in node return 0, pos is position of the last child in node
   if (pos == curr->count) return 0;
   // if key is found, return 1, pos is now position of key in current node;
   if (key == curr->value[pos]) return 1;
   // else return 0, pos is now position of the child node that might contain key
   return 0;
}
```

```
node* search(int key, node* curr, int& pos)

{
   // reach NULL node mean the key is not found in the tree, return NULL
   if (curr == NULL) return NULL;

   // if key is found in the current node return the current node
   // pos will be position of key in current node
   if (search_in_node(key, curr, pos)) return curr;
   // child[pos] is now the possible child to contain key, call recursive to this child
   return search(key, curr->child[pos],pos);
}
```

2. Insertion operation:

A new key always added to a B-tree at a leaf node.

To add a new key to the B-tree we follow these steps:

- Traverse the tree to reach a suitable leaf node.

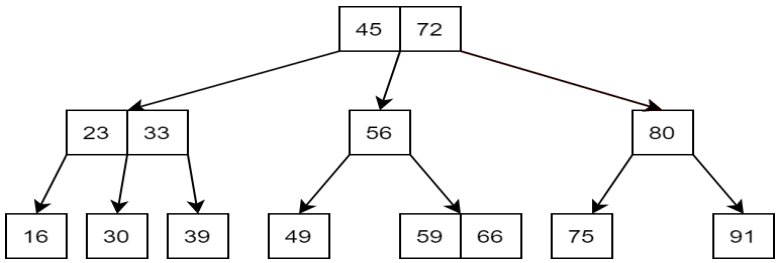
- Add new key to the leaf node.

- If the leaf node is full, splits it in half and moves the middle value onto its parent node.

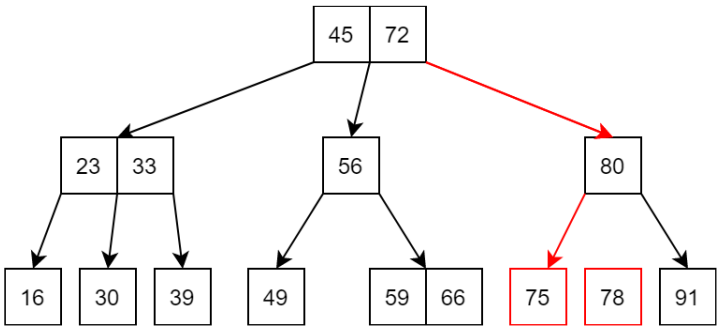
- Call recursive to check the parent node and split it if it is full until we reach a non-full node or root node.

- If the root node is full, splits it in half and creates a new root that will contain the middle value. This is the reason B-tree grow upward, unlike the BST which grow downward.

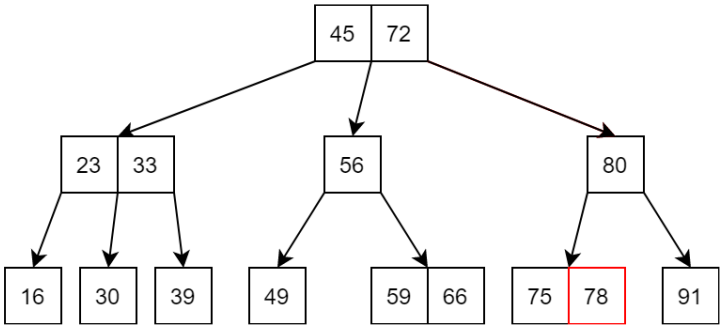
*** Example:** Add 78 to this 3-way B-tree:



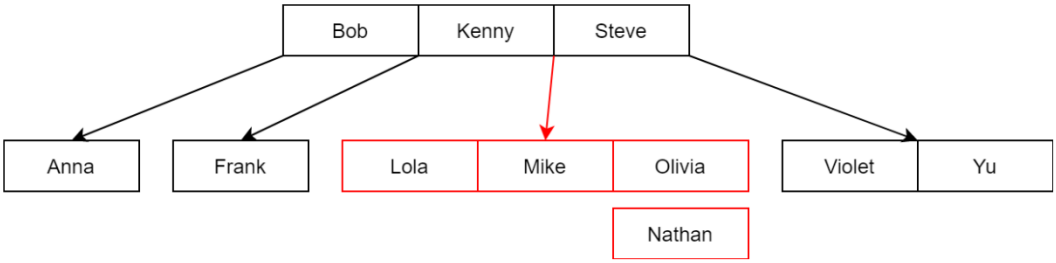
Start from the root node, we traverse the tree to reach a leaf node that will have 78 inserts to.



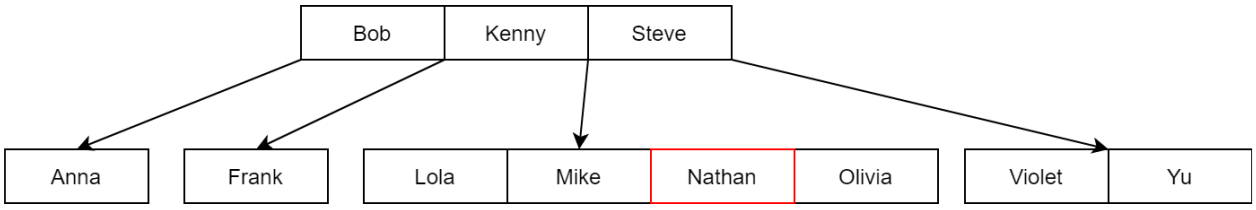
75 < 78, so we add 78 after 75. The leaf node is not full, no adjustment needed.



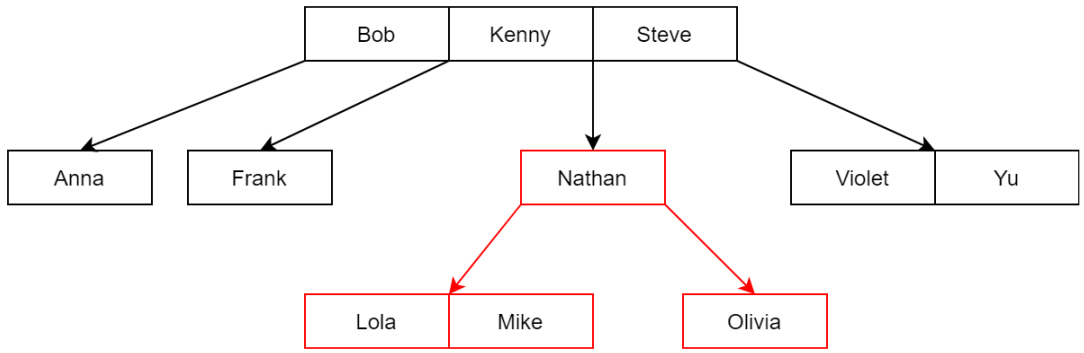
Now let's add a new name, Nathan, to this 4-way B-tree.



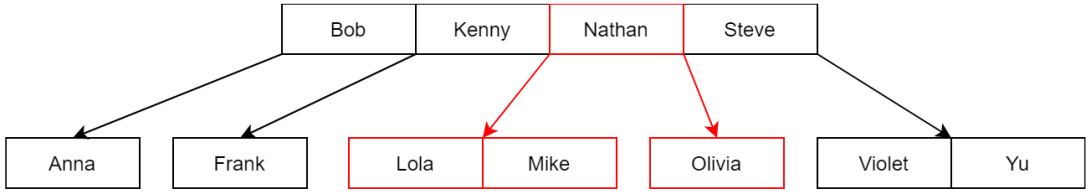
Mike < Nathan < Olivia, so Nathan is added to the middle.



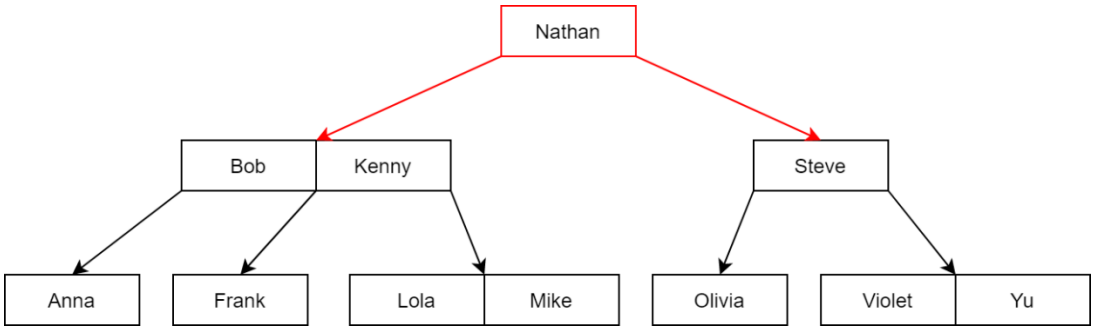
The leaf node is full, we split it in half and move the middle value, Nathan to the parent node, or in other word, insert Nathan to the parent node. Note that in this case, the middle value can be either Mike or Nathan, depend on how we design the algorithms.



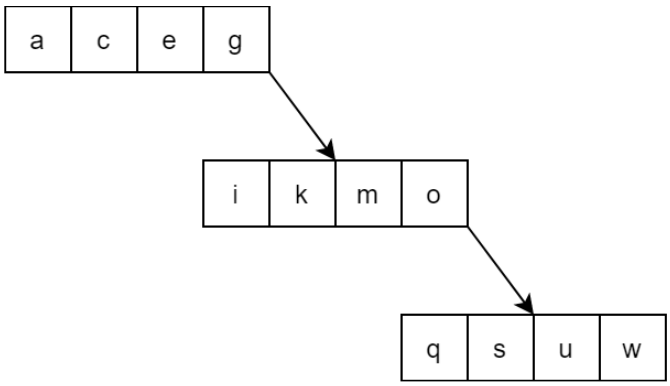
Kenny < Nathan < Steve



Now the root node is also full, we split it in half and the middle value, Nathan, become the new root. We can see the tree grow upward.



In the m-way tree section, we introduced an unbalanced 5-way tree with characters values:

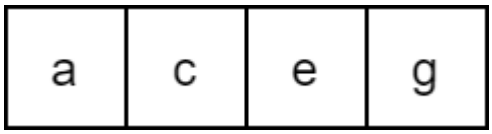


Now let's make a 5-way B-tree from these values.

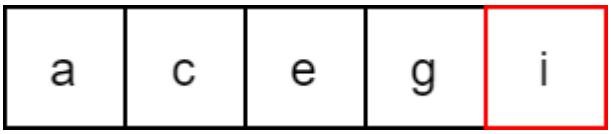
At first we have an empty tree, the first value will be added to the new-created root node, which is also the leaf node.

Continue on adding c, e and g. The root node will contain 4 keys.

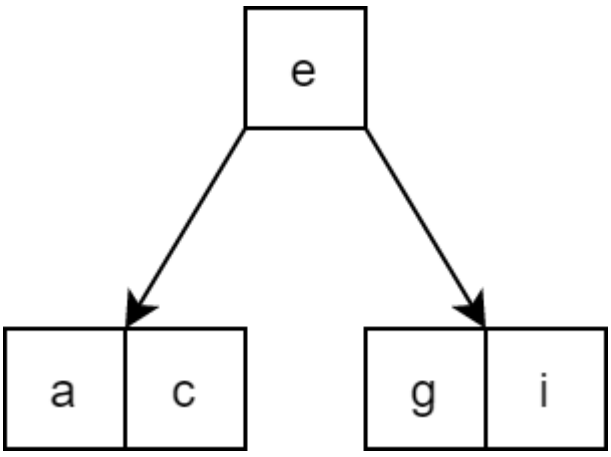
Now as we try to add i, we first add it to the root node.



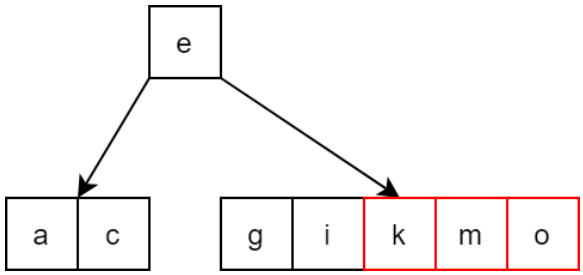
The root node is now full, we split it in half and e become the new root.



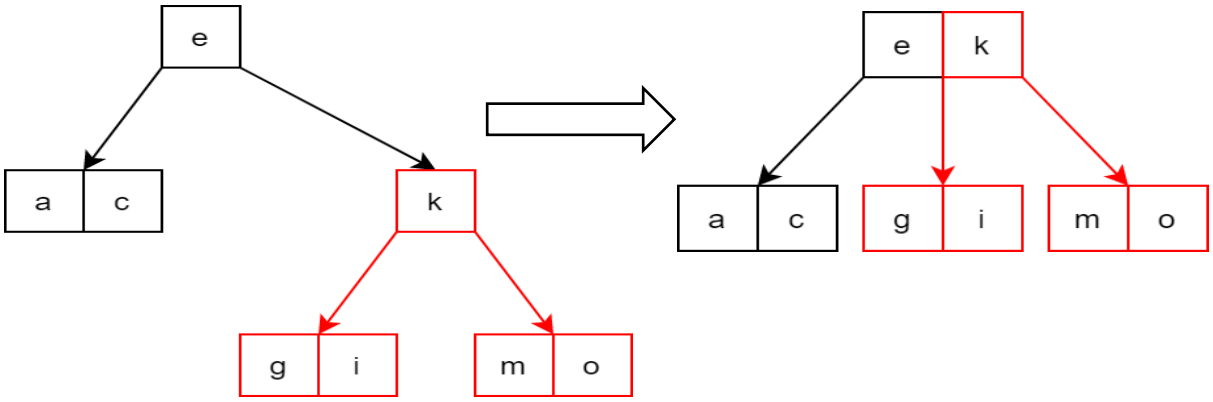
A node in 5-way B-tree other than root node must have at least [(5 – 1)/2] = 2 keys.



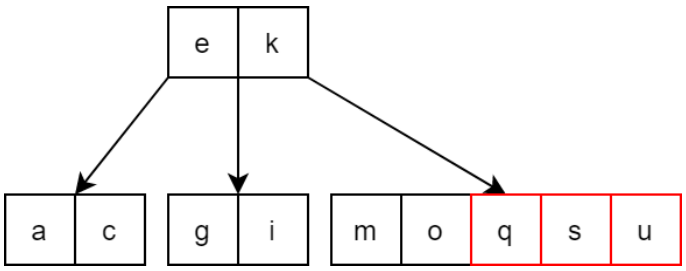
Continue on adding k and m, then o.



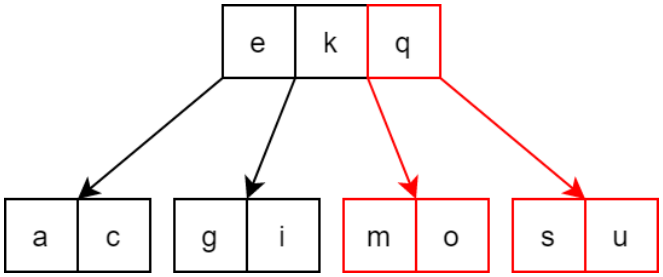
The leaf node is now full, we split it in half and k move to the parent node.



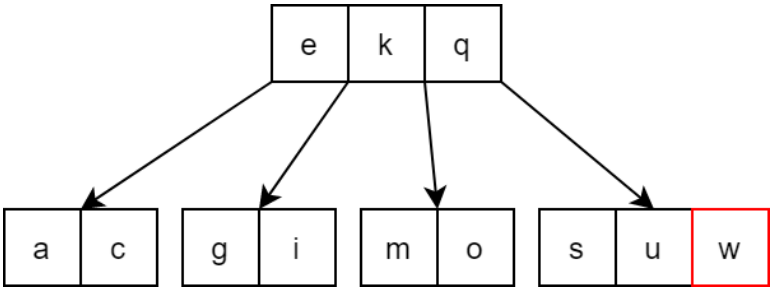
The same thing happen as q, s and u is inserted.



Split and move q to parent:



Now we finish by adding w.



If we add values to a normal m-way tree in sorted order, the tree keep growing in height and become totally unbalanced, while B-tree rebalance itself.

The inserting algorithms is implemented in C++ below. This is NOT the best way to do it, even bad, but it follows the idea of the algorithms step by step for easy understanding.

```
int findChildPos(node* child, node* parent)
{
   // find the position of child in parent's child array
   for (int i = 0; i <= parent->count; i++)
   {
      if (parent->child[i] == child) return i;
   }
   return -1;
}

void insert_at_leaf(node *curr,int key)
{
   // add the new key to the leaf node
   int i = 0;
   // find position for key
```

```
    while ((curr->value[i] < key)&&(i < curr->count)) i++;
    // shift value > key to the right
    for(int j = curr->count; j > i;j--)
    {
        curr->value[j] = curr->value[j - 1];
    }
    curr->value[i] = key; // add key
    curr->count++; // update count value
}

void split_node(node* parent, int childpos)
{
    // split child in half and move middle element to parent
    node* c1 = parent->child[childpos];
    // create new node for the second half
    node* c2 = new node;
    c2->isLeaf = c1->isLeaf;

    // move MIN elements to the second node
    c2->count = MIN;
    for (int i = 0; i < MIN; i++)
    {
        c2->value[i] = c1->value[WAY - MIN + i];
        c2->child[i] = c1->child[WAY - MIN + i];
        if (c2->child[i] != NULL) c2->child[i]->parent = c2;
    }
    c2->child[c2->count] = c1->child[c1->count];
    if (c2->child[c2->count] != NULL) c2->child[c2->count]->parent = c2;

    int midvalue = c1->value[WAY - MIN - 1];

    // if the child is not root node
    if (parent != NULL)
    {
        // move middle element to parent
        for (int i = parent->count; i > childpos; i--)
        {
            parent->value[i] = parent->value[i - 1];
            parent->child[i + 1] = parent->child[i];
        }
        parent->value[childpos] = midvalue;
        parent->child[childpos + 1] = c2;
        parent->count++;

        c2->parent = parent;
```

```
    }
    c1->count -= (MIN + 1);
}

void insert_key(node*& root, int key)
{
    // if tree is empty create a new root and add key to it
    if (root == NULL)
    {
        root = create_node();
        root->isLeaf = 1;
        root->count = 1;
        root->value[0] = key;
        return;
    }

    node* temp = root;
    // find a leaf node to insert key
    while (!temp->isLeaf)
    {
        int pos;
        if (search_in_node(key, temp, pos)) return;
        else temp = temp->child[pos];
    }

    // add key to found leaf node
    insert_at_leaf(temp, key);
    for (int i = 0; i <= WAY;i++) temp->child[i] = NULL;

    // check if the node is full and split if full
    // repeat which the parent node until the node is not full or reached root node
    while (temp->parent != NULL)
    {
        if (temp->count == WAY)
        {
            split_node(temp->parent, findChildPos(temp, temp->parent));
            temp = temp->parent;
        }
        else break;
    }
    // if root node is full
    if (temp->count == WAY)
    {
        // repeat the first half of splitChild function
        node* c1 = temp;
```

```
        node* c2 = new node;
        c2->isLeaf = c1->isLeaf;

        c2->count = MIN;
        for (int i = 0; i < MIN; i++)
        {
           c2->value[i] = c1->value[WAY - MIN + i];
           c2->child[i] = c1->child[WAY - MIN + i];
           if(c2->child[i] != NULL) c2->child[i]->parent = c2;
        }
        c2->child[c2->count] = c1->child[c1->count];
        if (c2->child[c2->count] != NULL)
              c2->child[c2->count]->parent = c2;

        int midvalue = c1->value[WAY - MIN - 1];

        // create new root and move middle element to it
        node* newroot = create_node();
        newroot->count = 1;
        newroot->value[0] = midvalue;
        newroot->child[0] = c1;
        newroot->child[1] = c2;

        c1->parent = newroot;
        c2->parent = newroot;

        root = newroot;
        c1->count -= (MIN + 1);
    }
    return;
}
```

3. Deletion operation:

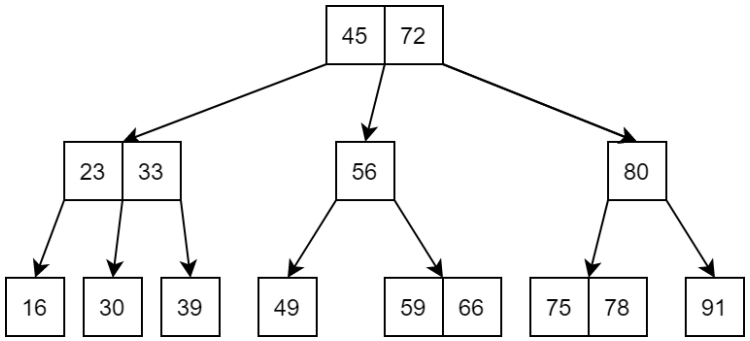
To delete a key from a B-tree, first we do the same as a normal m-way tree. The actual node to have a key remove from will always be the leaf node.

If a node has less than [(m – 1)/2] keys, we do one of the following:

- If an adjacent sibling node has enough key, borrow 1 key from it to replace a key in parent node and add that key to the current node.

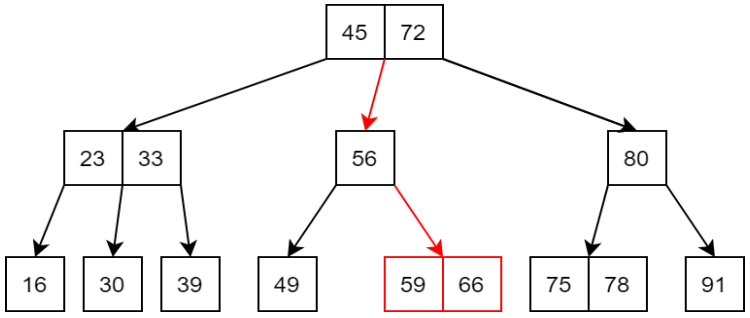
- Otherwise, merge the current node with an adjacent sibling node and a corresponding key from parent node.

*** Example:**

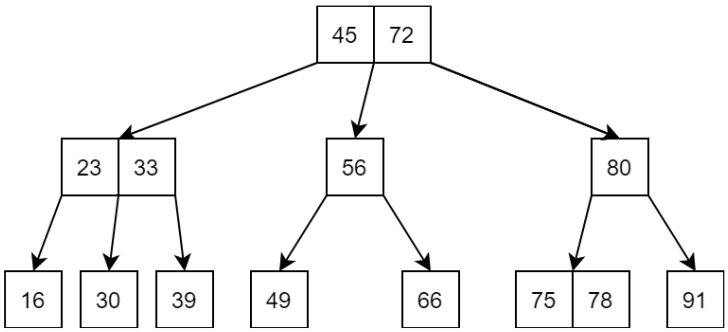


Remove 59 and 80 from the above 3-way B-tree.

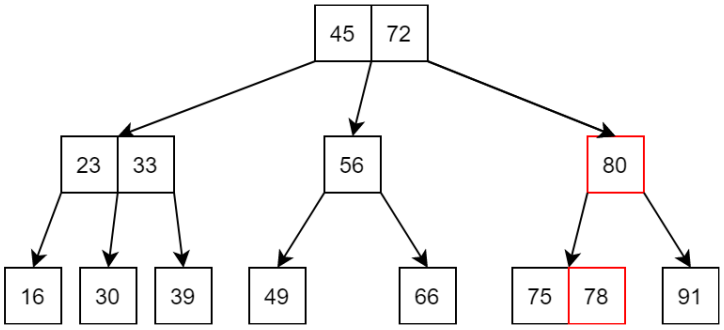
We traverse the tree to find the node contain 59. 59 doesn't have any child, so we just remove it.



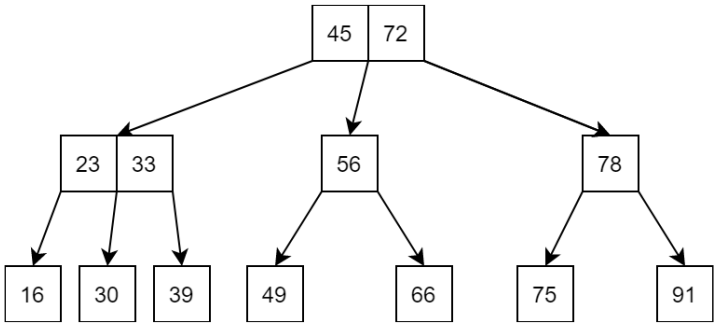
After removing 59, the node still has (3-1)/2 = 1 key, no adjustment needed.



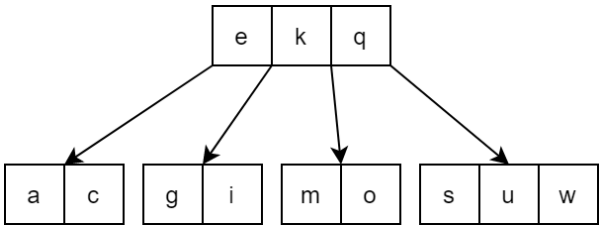
Now we find the node contain 80. 80 has both left and right child, we can choose to replace 80 with the largest value in the left sub tree, or the smallest value in the right sub-tree. We choose the left one this time. The largest value in the left sub tree is 78.



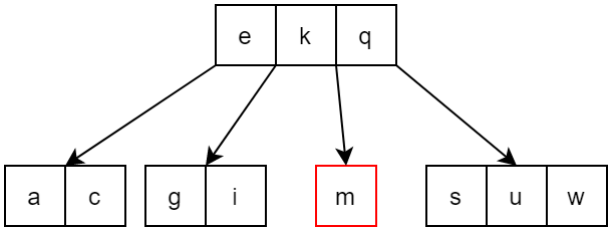
After removing 78 from its old node, the node still has enough key, no adjustment needed.



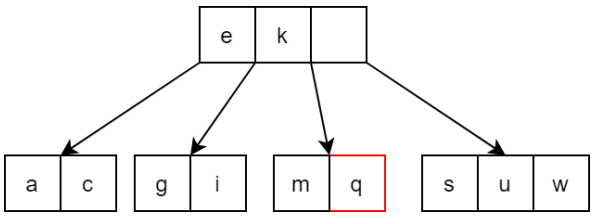
Now let's try removing o from this 5-way B-tree:



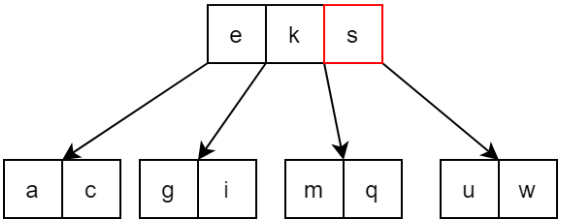
We find the node holding o, but after we remove o from it, it only has 1 key left, which is less than (5 – 1)/2 = 2 keys.



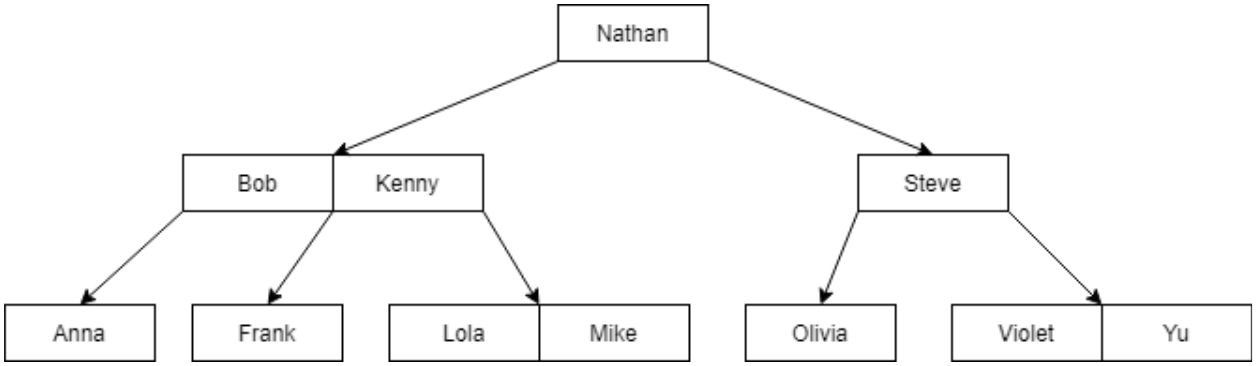
The left adjacent sibling node doesn't have enough key to share, but the right one does, so we borrow s from the right sibling node. We start by moving the key in parent node that is in between current node and right sibling node down to current node.



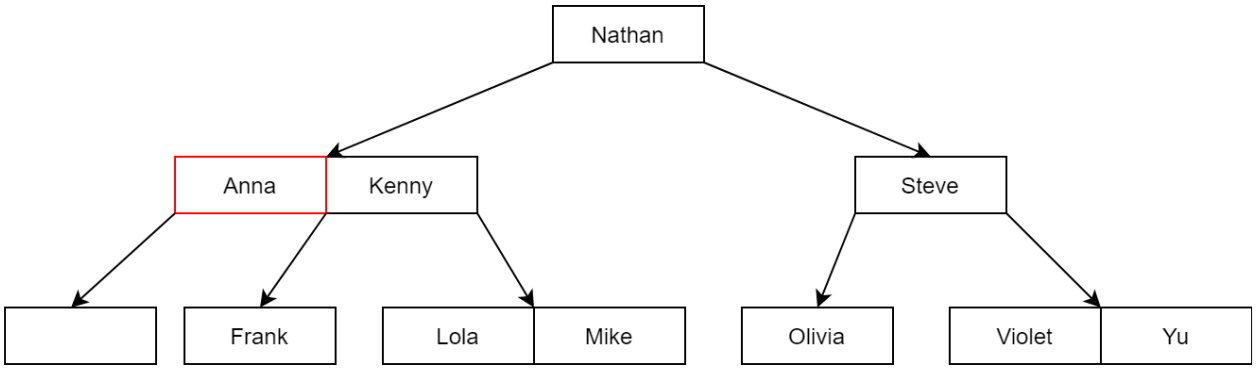
Now the borrowed value in the right sibling node moves up to free slot in the parent node.



Now let's remove Bob from this 4-way B-tree.

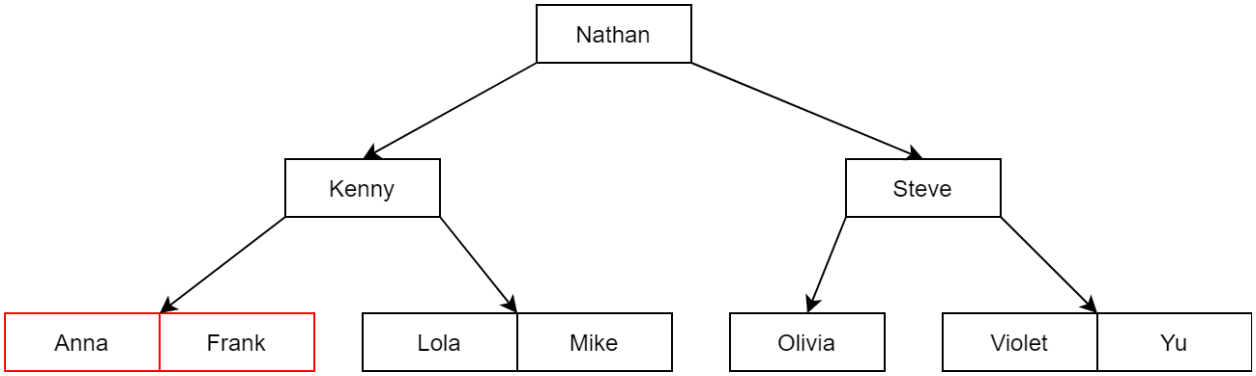


Bob has both children, we can replace him with the largest value in the left sub tree, Anna.



Now Anna's old node doesn't have enough key. Because the node is the first child of its parent node, it only has right adjacent sibling node. But the right sibling node doesn't have enough key to share, so we much do the merging operation.

Merge the current node, the right sibling, and the key in between them in parent node into one node and place it in the old node position.



For deeper understanding, we separate the fixing operation after deleting into 4 cases. In the following part:

- P represent keys of parent node.

- S represent keys of siblling node.

- N represent keys of current node.

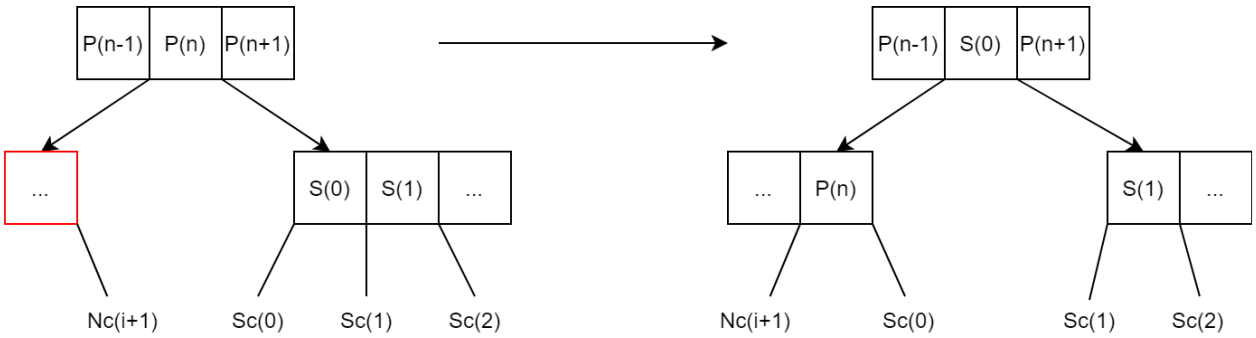
- Sc represent children of sibling node.

- Nc represent children of current node.

- (…) represent 0 or more keys and corresponding child.

- The red node is the current node that missing keys, it can be empty or not, but always have at least one child.

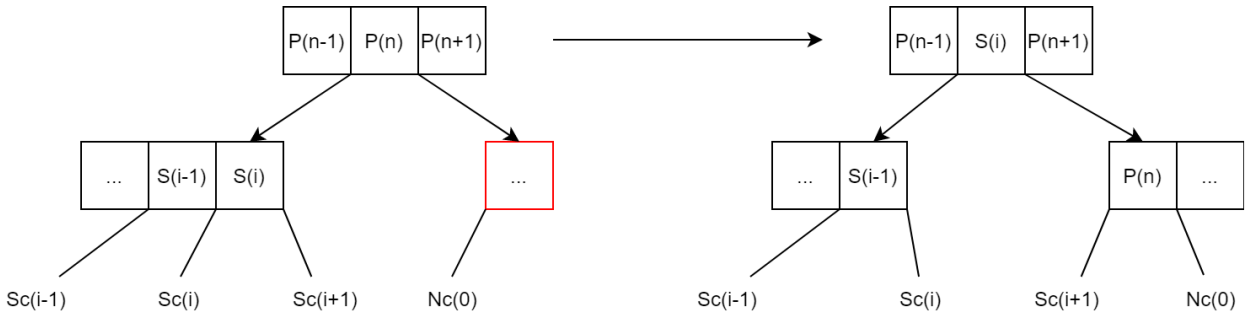
Case 1: Borrow from right sibling



Pseudo code:

```
void fixCase1(node *N, node *S, node *P,int n)
{       N[N->count] = P[n] ;
        P[n] = S[0];
        N->child[N->count + 1] = S->child[0];
        for i = 0 to S->count – 2 : S[i] = S[i+1]; S->child[i] = S->child[i+1];
        S->child[S->count - 1] = S->child[S->count];
        S->count = S->count – 1;
        N-> count = N-> count + 1;
}
```

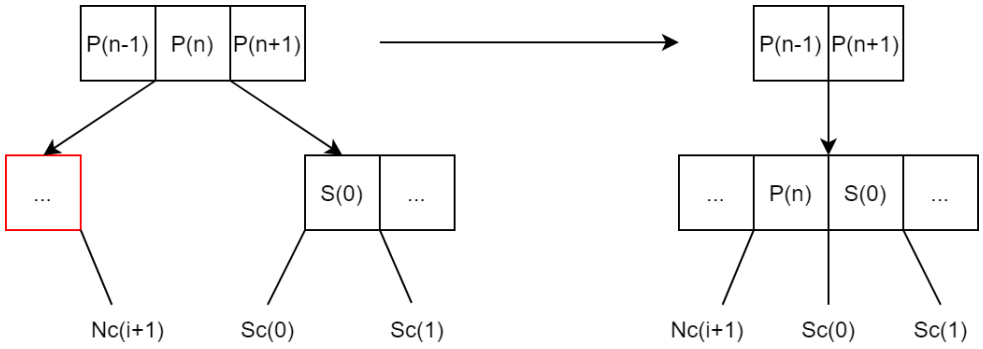
Case 2: Borrow from left sibling



Pseudo code:

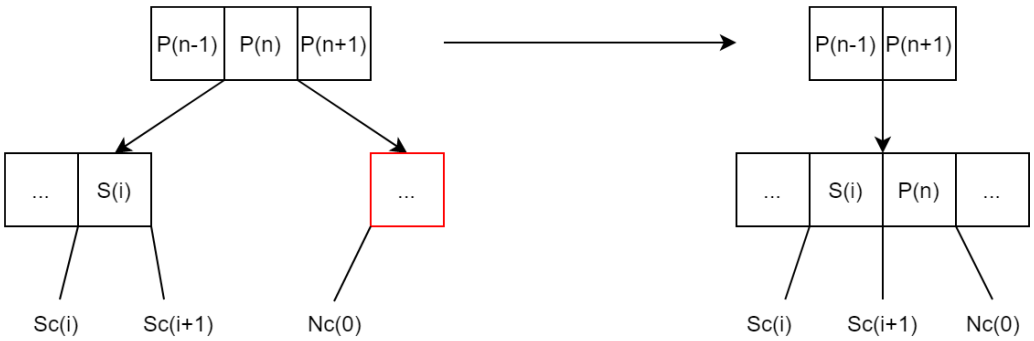
```
void fixCase2(node *N, node *S, node *P,int n)
{       for i = N->count downto 1 : N[i] = N[i-1]; N->child[i+1] = N->child[i];
        N->child[1] = N->child[0];
        N[0] = P[n];
        P[n] = S[S->count – 1];
        N->child[0] = S->child[S->count];
        S->count = S->count – 1;
        N-> count = N-> count + 1;
}
```

Case 3: Merge with right sibling



Pseudo code:

```
void fixCase3(node *N, node *S, node *P, int n)
{       N[N->count] = P[n];
        N->count = N->count + 1;
        for i = 0 to S->count – 1 :
        {       N[N->count] = S[i];
                N->child[N->count] = S->child[i];
                N->count = N->count + 1;       }
        N->child[N->count] = S->child[S->count];
        for i = n to P->count – 2 : P[i] = P[i+1]; P->child[i+1] = P->child[i+2];
        P->count = P->count – 1;
        delete S;
}
```



Case 4: Megre with left sibling

Pseudo code:

```
void fixCase4(node *N, node *S, node *P, int n)
{       S[S->count] = P[n];
        S->count = S->count + 1;
        for i = 0 to N->count – 1 :
        {       S[S->count] = N[i];
                S->child[S->count] = N->child[i];
                S->count = S->count + 1;        }
        S->child[S->count] = N->child[N->count];
        for i = n to P->count – 2 : P[i] = P[i+1]; P->child[i+1] = P->child[i+2];
        P->count = P->count – 1;
        delete N;
        }
```

Topic 6: Graph

Contents:

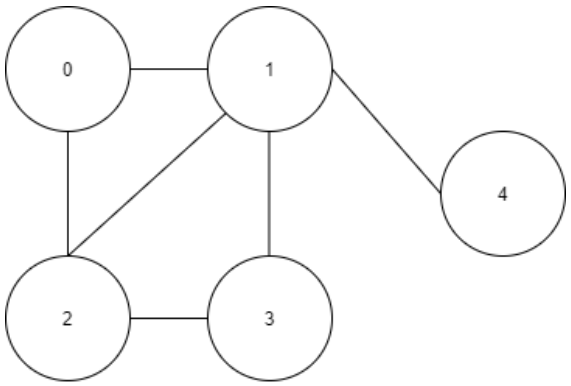
I. Definition:

A Graph G = (V, E) is a data structure that consists of two sets: a finite set of vertices (V) (also called vertexs/nodes/points) and a set of edges (E) (also called links/lines) that connect the vertices.

- V(G) represents the set of vertices of the Graph.
- E(G) represents the set of edges of the Graph.

Some examples of graphs:

a)

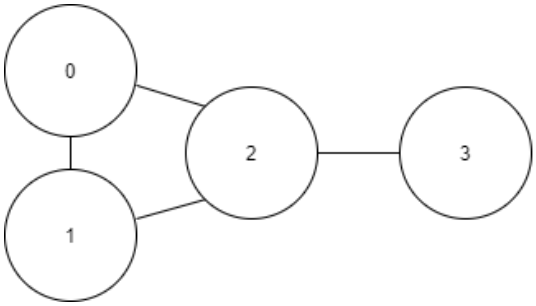


V(G) = {0,1,2,3,4}

E(G) = {(0,1), (0,2), (1,3), (1,2), (1,4), (2,3)}

G = (V, E)

b)

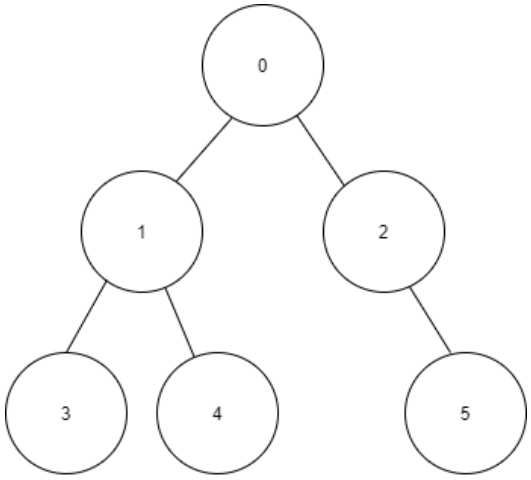


V(G) = {0,1,2,3}

E(G) = {(0,1), (0,2), (1,2), (2,3)}

G = (V, E)

c)



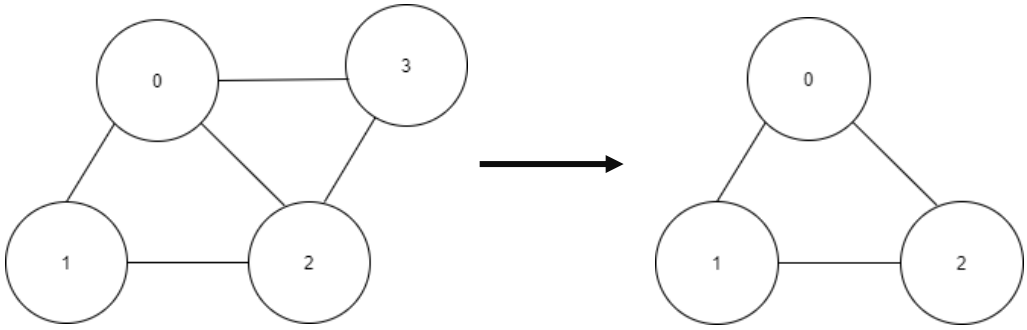
V(G) = {0,1,2,3,4,5}

E(G) = {(0,1), (0,2), (1,3), (1,4), (2,5)}

G = (V, E)

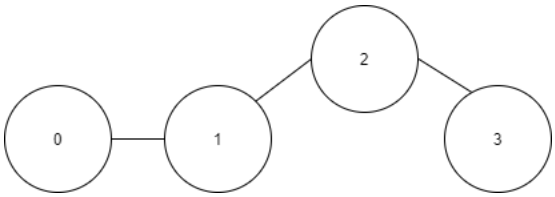
II. Graph terminology:

- Subgraph: A subgraph consists of a subset of graph's vertices and edges.

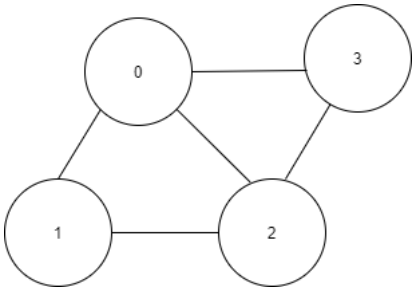


Graph A One subgraph of graph A

- Adjacency: Two vertices are adjacent if they are connected by an edge.

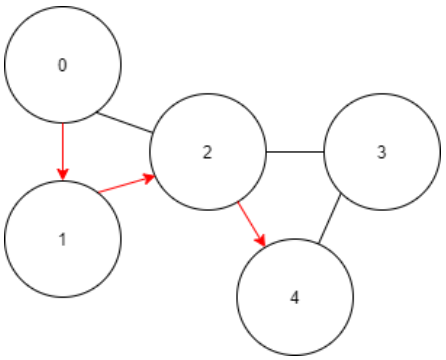


These pairs of vertices are adjacent: (1) & (0), (1) & (2), (2) & (3).



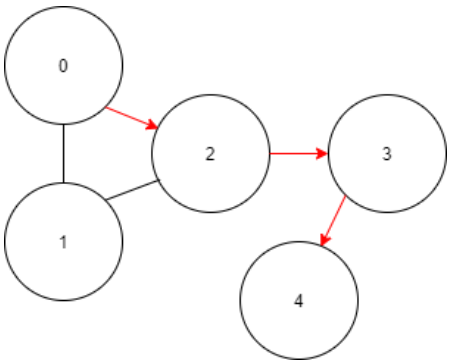
These pairs of vertices are adjacent: (0) & (1), (1) & (2), (0) & (2), (2) & (3)

- Path: A sequence of edges that go from one vertex to another vertex is called path. A *simple path* is the path that passes through the same vertices of the graph just **one** time.



There is a path that begins at vertex (0), then go to vertex (1), then (2), and ends at vertex (4).

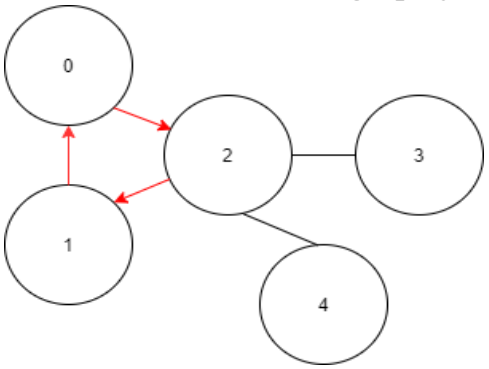
$$(0 \rightarrow 1 \rightarrow 2 \rightarrow 4)$$



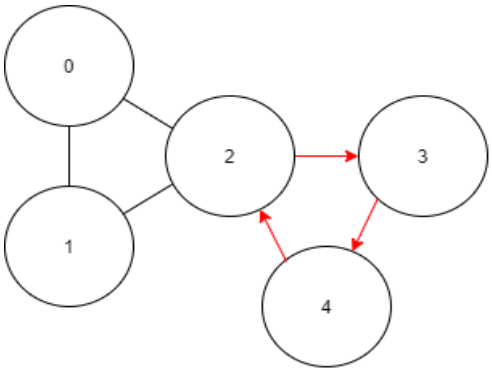
There is also another path that begins at vertex (0) and ends at vertex (4):

$$(0 \rightarrow 2 \rightarrow 3 \rightarrow 4)$$

- Cycle: A cycle is a path that goes from a beginning vertex and ends at that vertex again. A *simple cycle* is a cycle that passes through the same vertices of the graph just **one** time (except the start-end vertex).

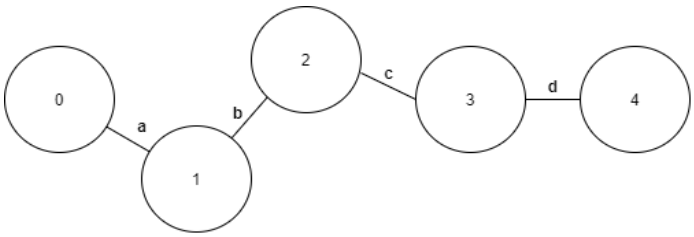


The path $(0 \rightarrow 2 \rightarrow 1 \rightarrow 0)$ is a cycle in the graph. (Start at 0, end at 0)



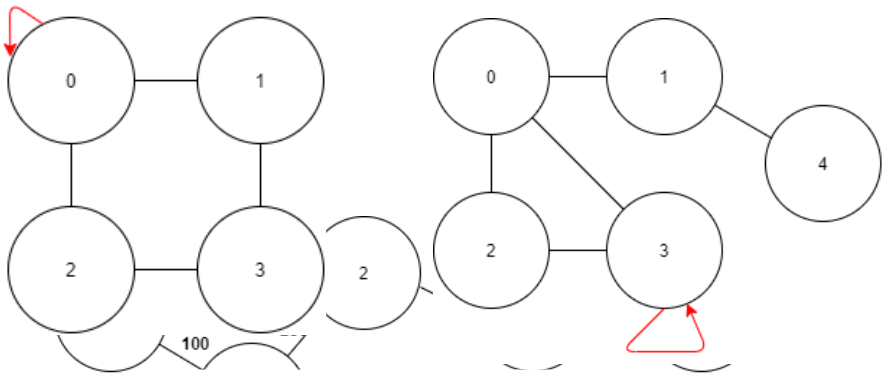
The path $(2 \rightarrow 3 \rightarrow 4 \rightarrow 2)$ is a cycle in the graph. (Start at 2, end at 2)

- Label: Each edge of a graph can have a name/value called its label.



Each edge of graph has a name: a, b, c, d.

- Self edge (loop): A edge begin and end at the same vertex is called a self edge (or loop).



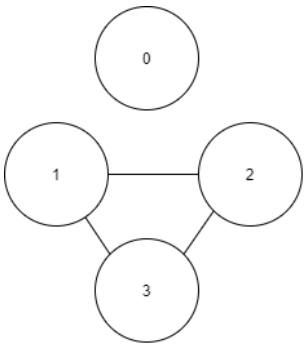
Some self edge (in red) of graphs

Each edge of graph has a value: 100,250,50,10.

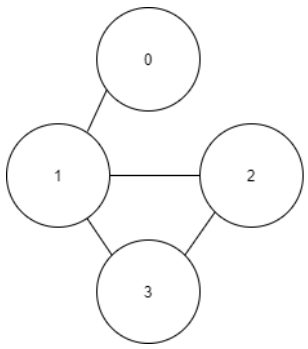
III. Types of Graphs:

- Connected and Disconnected Graph: A connected graph is a graph whose each pair of vertices is connected by a path. Oppositely, a disconnected graph is a graph that has at least one pair of vertices not connnected by any path.

Example:

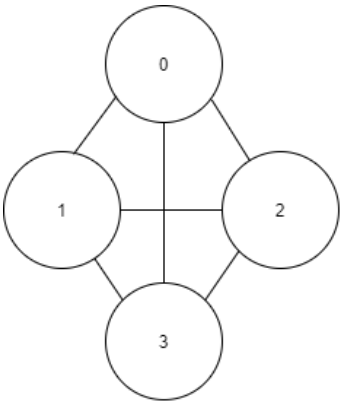


Disconnected Graph

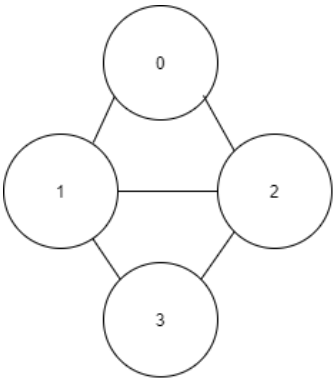


Connected Graph

- Complete Graph: A complete graph is a graph whose each pair of distinct vertices is connected by a edge. A complete graph is also a connected graph, but the converse is not true

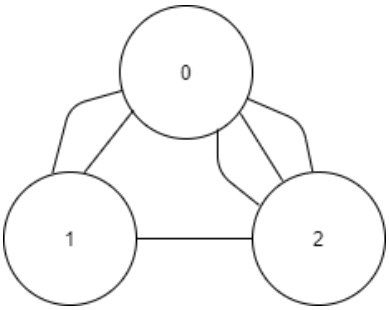


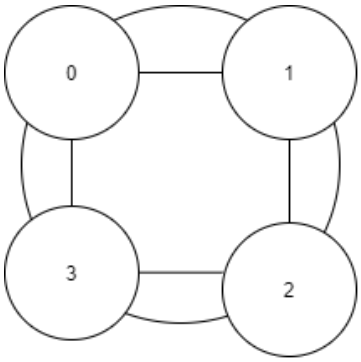
Complete Graph



Not a Complete Graph

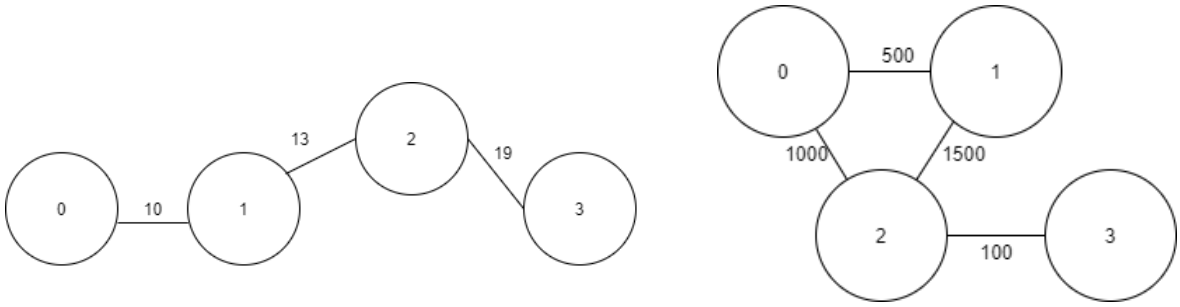
- Multigraph: A graph can't have more than one edge between a pair of vertices. A multigraph allows multiple edges between a pair of vertices. So, a multigraph is **not** a graph.





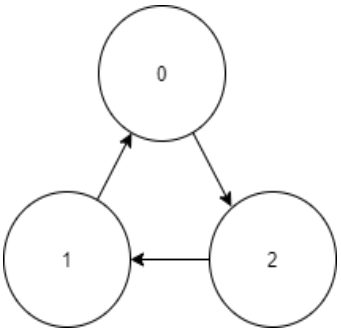
Multigraphs

- Weighted Graph: If the labels of graph's edges represent numeric values, the graph is called weighted graph. (The value can be represented for some kind of common units such as: km/m, time,)



Weighted Graphs

- Undirected Graph and Directed Graph: In undirected graph, the graph's edges are bi-directional, meaning that they do not point to any specific direction. Oppositely, in directed graph (digraph), each edge of graph points to one specific direction, and it's called directed edges i.e an edge **(a, b)** doesn't mean that there is also an edge **(b, a)**, **a** is adjacent to **b** but **b** is not adjacent to **a**. In digraph, the edge is also called the arc or the arrow.

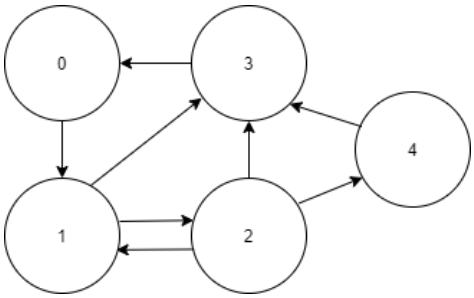


V(G) = {0,1,2}

E(G) = {<1,0>, <0,2>, <2,1>}

In this graph, each edge points to a single direction, edges: $(1 -> 0), (0 -> 2), (2 -> 1)$.

The vertex (0) is adjacent to the vertex (1), but the vertex (1) isn't adjacent to the vertex (0). Similar to vertex (2) and (0), (1) and (2).



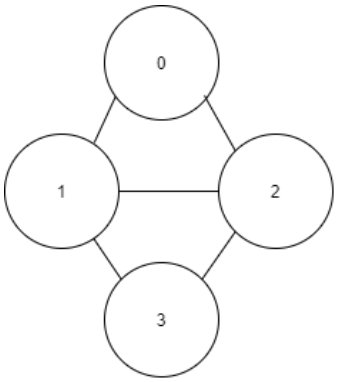
V(G) = {0,1,2,3,4}

E(G) = {<0,1>,<1,2>,<2,1>,<2,3>,<3,0>,<2,4>,<4,3>}

In this graph, each edge points to a single direction, edges: $(0 \rightarrow 1), (1 \rightarrow 2), (2 \rightarrow 1), (2 \rightarrow 3)$

$$(2 -> 4), (3 \rightarrow 0), (2 \rightarrow 4), (4 \rightarrow 3)$$

The vertex (0) is adjacent to the vertex (1), but the vertex (1) isn't adjacent to the vertex (0). The vertex (1) is adjacent to the vertex (2) and the vertex (2) is also adjacent to the vertex (1).



This is an undirected graph. Each edge doesn't point to any specific direction.

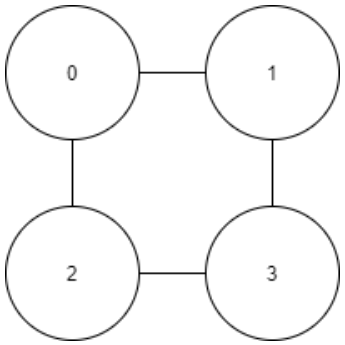
IV. Graph representations:

There are two most commonly used ways to represent a graph:

❖ *Adjacency matrix*

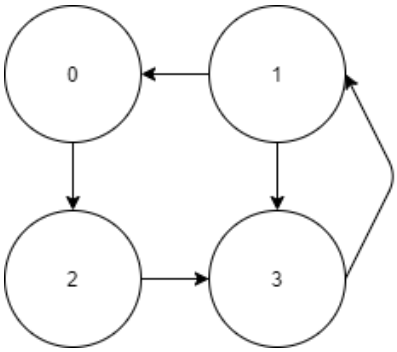
- An adjacency matrix is a 2D – array $V * V$ (A[][]), V is the number of vertices in the graph. In the matrix, A[i][j] = 1 means that there is an edge between vertex i and vertex j, A[i][j] = 0 means that there isn't any edges between verter i and vertex j.
- Adjacency matrix is also used to reprensent a weighted graph. A[i][j] = W means that there is an edge between vertex i and vertex j and the value of that edge is equal to W. (so that if there isn't a vertex from i to j, don't use W = 0 because in some real experience, we need to use W = 0 to store a value of edge equal to 0. Use symbol: A[i][j] = ∞.

- In undirected graph, if (A[i][j] = 1), (A[j][i] = 1) also. But in directed graph, (A[i][j] = 1) doesn't mean that (A[j][i] = 1).
- Examples:



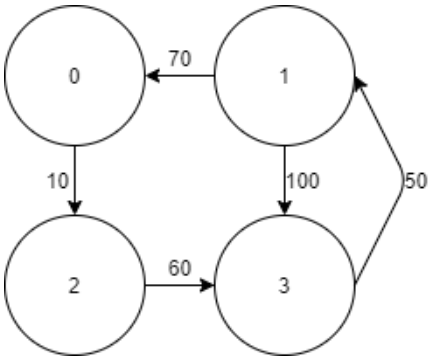
Adjacency matrix of this undirected graph

	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	1	0	0	1
3	0	1	1	0



Adjacency matrix of this directed graph

	0	1	2	3
0	0	0	1	0
1	1	0	0	1
2	0	0	0	1
3	0	1	0	0



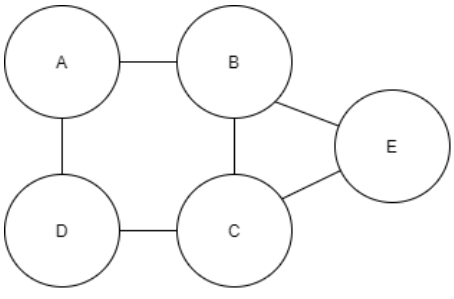
Adjacency matrix of this directed graph with edges's values

	0	1	2	3
0	∞	∞	10	∞
1	70	∞	∞	100
2	∞	∞	∞	60
3	∞	50	∞	∞

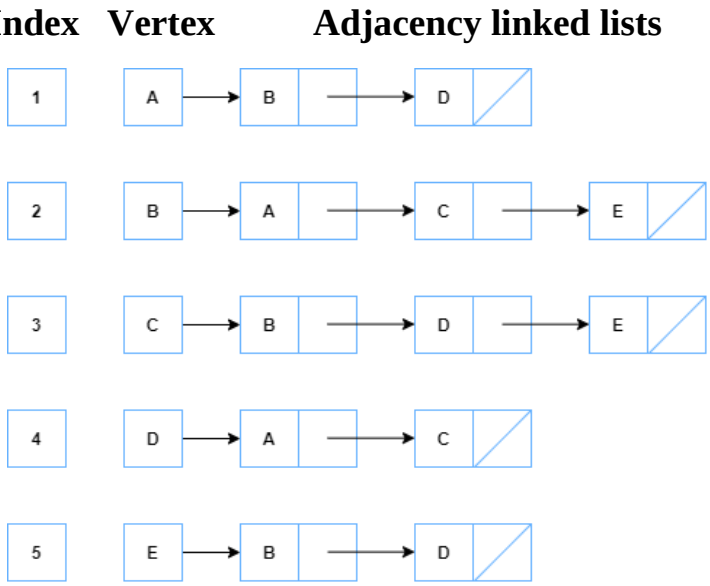
- Advantages of using adjacency matrix:
    - Demonstrating a 2D – array is easy to follow and understand.
    - Determine if two vertices are adjacent to each other, add an edge, delete an edge in a constant of time and these actions take effectively if you want to add/delete edges very frequently.
- Disadvantages of using adjacency matrix:
    - It's take the same memory even when the graph is spraser (always O(n^2)).
    - It's slow to add/delele a vertex or iterate over all edges because it's a 2D – array.

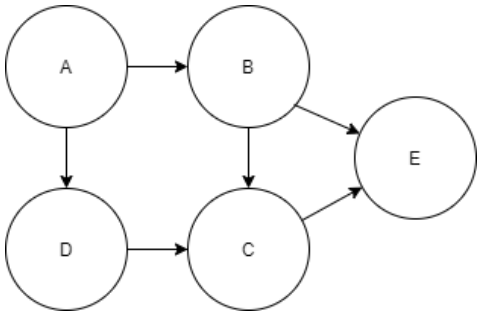
❖ *Adjacency list:*

- An adjacency list represents a graph as an array of linked lists. The array size is equal to the number of graph's vertices. The i^{th} linked list in the array represents the i^{th} vertex in the graph and each element in that linked listed represents the other vertices that form an edge with that i^{th} vertex and i^{th} vertex is adjacent to each element (especially in directed graph). Each element in each linked list can also carry a edge's value.
- Examples: represent adjacency lists for some kinds of graph:

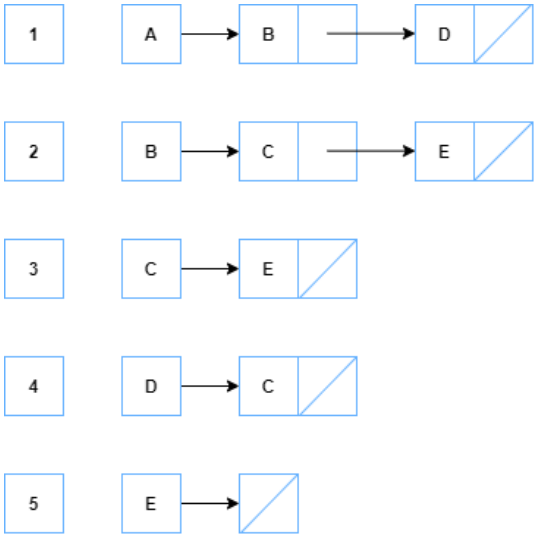


Undirected graph



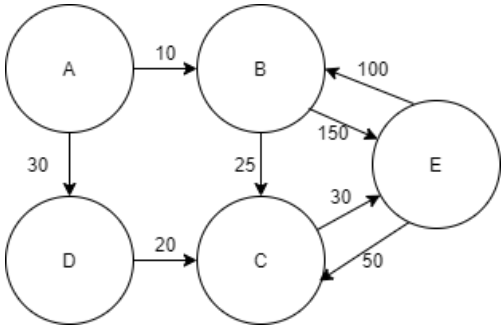


Directed graph

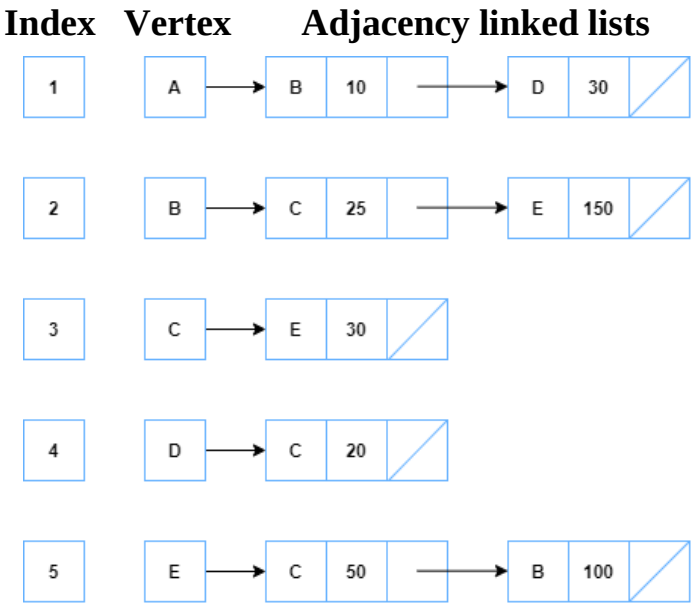


Explain the difference between undirected graph and directed graph in adjacency list representation of two example: in undirected graph: vertex B is adjacent to the vertex A and C, but in directed graph, because there is just one arc from A to B so A is adjacent to B but B isn't adjacent to A. Vertex E, C and D is also follow this rule too.

Here is one more example of directed graph with edges's value:



Directed graph with edges's value



- Advantages of using adjacency list:
    - Memory used depends on number of edges and verticles.
    - It's fast to add/delete a vertex, a edge.
    - It's fast to iterate over all edges/nodes.
- Disadvantages of using adjacency list:
    - It's time consuming finding relationship between two nodes is less effective than adjacency-matrix's.

5) Graph operations:

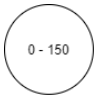
These are some basic operations of graph as data structure:

(graph G, node x, node y, value val)

- ***checkEmptyGraph(G):*** test whether the graph is empty.
- ***countVertices(G):*** get the number of vertices in a graph.
- ***countEdges(G):*** get the number of edges in a graph.
- ***adjacent(G,x,y):*** if the graph is undirected, test whether there is a edge between vertex ***x*** and vertex ***y***. If the graph is directed, test whether there is a edge start from vertex ***x*** and end at vertex ***y***.
- ***addVertex(G,x,val):*** add vertex ***x*** in the graph, if ***val*** isn't in the graph.
- ***addEdge(G,x,y,val):*** if the graph is undirected, add a edge between vertex ***x*** and ***y*** with edge's value is ***val***, if it doesn't exist in the graph. If the graph is directed, add a edges start from vertex ***x*** and end at vertex ***y*** with edge's value is ***val***, if it is not there before. With non – weighted graph, the variable val doesn't been used.
- ***removeVertex(G,x):*** delete vertex ***x*** and every edges that connected with ***x*** in the graph, if ***x*** exists in the graph.
- ***removeEdge(G,x,y):*** delete the edges between vertex ***x*** and vertex ***y***, if it exists in the graph.
- ***setVertexValue(G,x,val):*** if vertex ***x*** exists, update its value to ***val***.

- ***setEdgeValue(G,x,y,val):*** if there is an edge between ***x*** and ***y***, update its value to ***val***. With non – weighted graph, this function is meanless.

Example for graph's operations: Directed – Weighted graph

Note: 

this vertex means: vertex **0** has the value **150**.

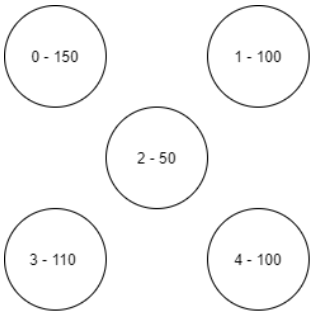
Action 1: start with empty graph G.

checkEmptyGraph(G) = True.

Action 2: Insert some vertices

addVertex(G,0,150)
addVertex(G,1,100)
addVertex(G,2,50)
addVertex(G,3,110)
addVertex(G,4,100)

checkEmptyGraph(G) = False



Action 3: Insert some edges

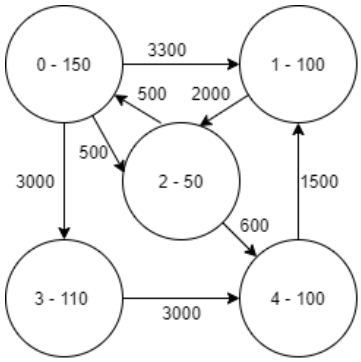
addEdge(G,0,1,3300)
addEdge(G,0,2,500)
addEdge(G,0,3,3000)
addEdge(G,1,2,2000)
addEdge(G,4,1,1500)
addEdge(G,3,4,3000)
addEdge(G,2,0,500)
addEdge(G,2,4,600)

adjacent(G,0,2) = True

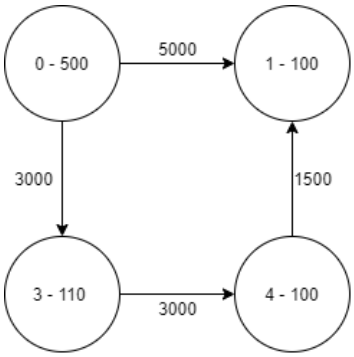
adjacent(G,4,2) = False

countVertices(G) = 5

countEdges(G) = 8



Action 4: Some other operations:
removeVertex(G,2) = True
removeEdge(G,0,4) = False
(because there is not a edge
from 0→4 in the graph)
setVertexValue(G,0,500) = True
setEdgeValue(G,0,1,5000) = True



6) Graph traversal:

- Graph traversal refers to the process of visiting each vertex in the graph. If the graph is not a connected graph, graph-tranversing will just visit a subgraph (the unconnected vertex will not be visited). Graph traversal may require that some vertices be visit more than one, so it's important to remember that which vertex has already been visited in case that the graph has a loop, the traversing may be loop indefinitely.
- There are two most common ways to tranverse the graph: Depth-First Search (DFS) and Breath-First Seach (BFS). DFS and BFS is a basic procedure for many graph-related algorithms.

❖ **Depth-First Search (DFS):**

- DFS algorithm is a algorithm that tranverses the depth of a whole particular path before it tranverses its breadth's path in a graph. In other words, DFS visits all child - vertices before visiting the sibling - vertices.
- DFS algorithm starts from a selected vertex (root) and then iteratively goes to its adjacent-unvisited vertex until at its position, it can't find any more unvisited vertex (it has been going to the deepest vertex of that path), then, the vertex moves back to the previous vertex until it finds another adjacent-unvisited vertex of its current position's vertex. And the above process is continuelly used for the new path which DFS just

found. Until the position backs to the root vertex and the root vertex doesn't have any unvisited – adjacent vertex.

- Remember to mark which vertex is visited because it's very easy to get in an endless loop if DFS visits a vertex many times.
- With directed graph and unconnected graph, the DFS may be visit just a subgraph of the graph instead of the whole graph.
- In Adjacency List, the complexity of DFS is $O(V + E)$. In Adjacency Matrix, the complexity of DFS is $O(V^2)$. With V is the number of vertices, E is the number of edges in the graph.
- Pseudo-code of DFS using recursive function:

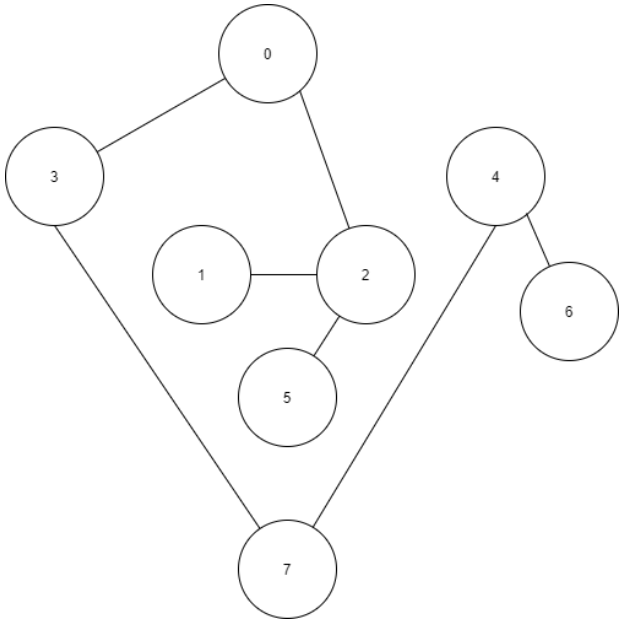
```
DFS(G,v)   // G stands for graph, v stand for vertex where DFS starts
        If visited[] = true: return // if all vertices are visited, return.
        visited[u] = true;
        // visited[] is an array that check whether a vertex visited, false is unvisited, true is visited.
        for each vertex u which adjacent to vertex v:
            DFS(G,u);
END DFS()
```

- Pseudo-code of DFS using stack structure:

```
DFS(G,v)
        visited[] = false;
        S = new stack;
        Push(s, v); // push start vertex into stack
        Visited[v] = true;
        While(S != empty)
                If (adjacent(S.top()) = 0) // there isn't any adjacent vertex of stack top vertex
                        Pop(s);
                Else
                        U = unvisited_adjacent of S.top();
                        Push(u);
                        Visited[u] = true;
END DFS()
```

- Examples for DFS algorithm:

Example 1:



Connected – Undirected graph

Step 1: Start DFS algorithm at vertex 0.

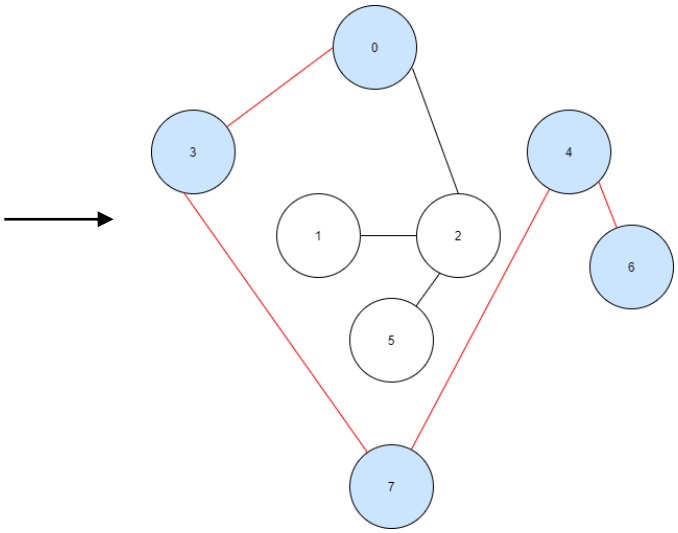
Go through the path:

$(0 \rightarrow 3)(3 \rightarrow 7)(7 \rightarrow 4)(4 \rightarrow 6)$

Then stop because vertex 6 hasn't any unvisited adjacency.

Visited vertex: 0,3,7,4,6

Unvisited vertex: 1,2,5



Step 2: From vertex 6, back to vertex 4,7,3,0. Stop at vertex 0 because vertex 0 has another unvisited adjacent vertex: vertex 2

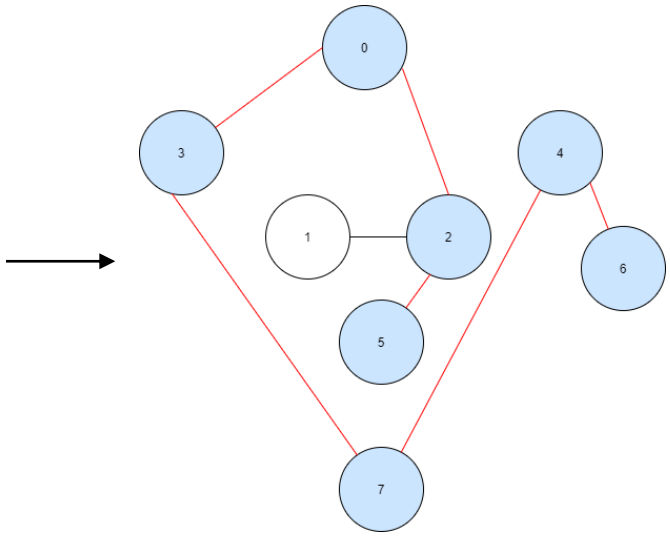
Go through the path:

$(0 \rightarrow 2)(2 \rightarrow 5)$

Then stop because vertex 5 hasn't any unvisited adjacency.

Visited vertex: 0,3,7,4,6,2,5

Unvisited vertex: 1



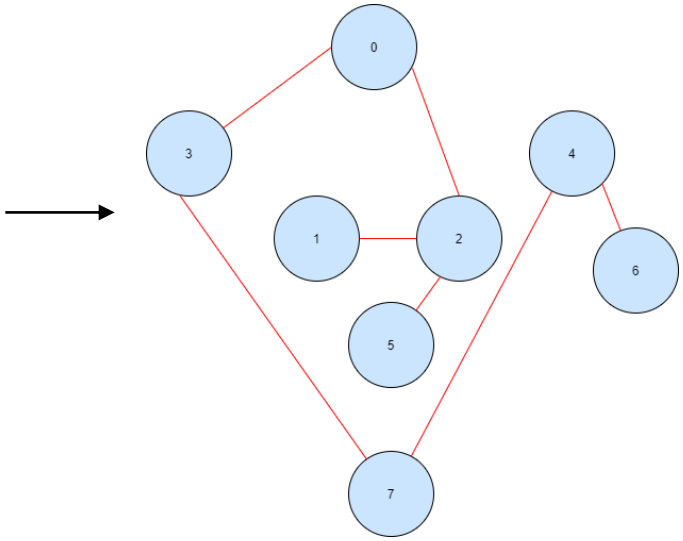
Step 3: From vertex 5, back to vertex 2. Stop at vertex 2 because vertex 2 has another unvisited adjacent vertex: vertex 1.

Go through the path: $(2 \rightarrow 1)$

Then stop because vertex 1 hasn't any unvisited adjacency.

Visited vertex: 0,3,7,4,6,2,5,1

Unvisited vertex: None



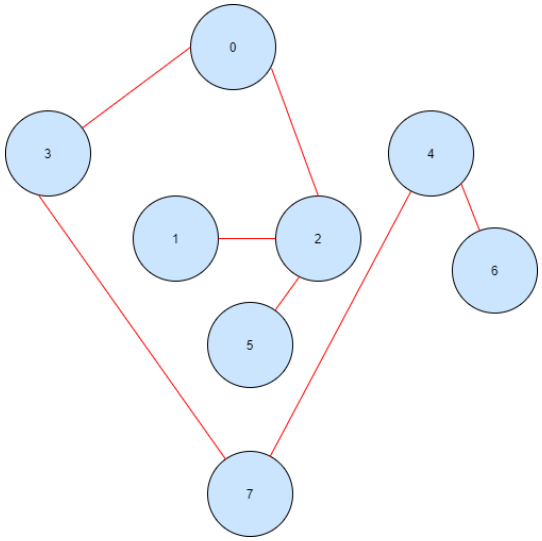
Step 4: From vertex 1, back to vertex 2,0. Stop at vertex 0 because this is the root and vertex 0 hasn't no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

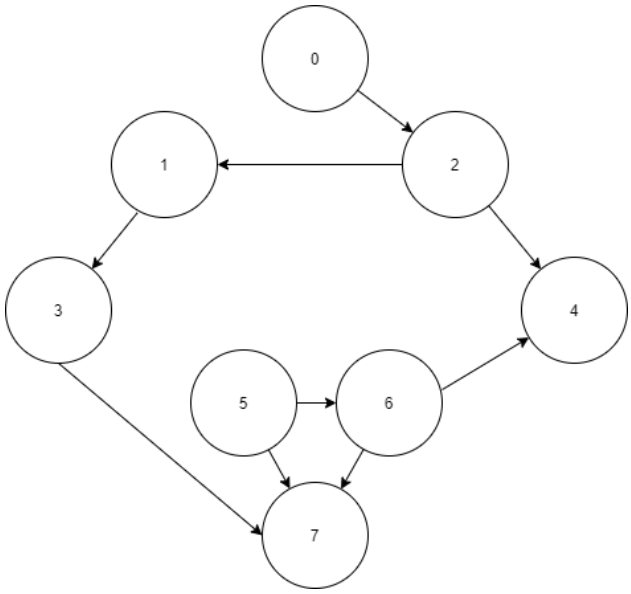
All vertices of the graph have been visited because this is the connected – undirected graph.

Visited vertex: 0,3,7,4,6,2,5,1

Unvisited vertex: None



Example 2:



Connected – Directed graph

Step 1: Start DFS algorithm at vertex 0.

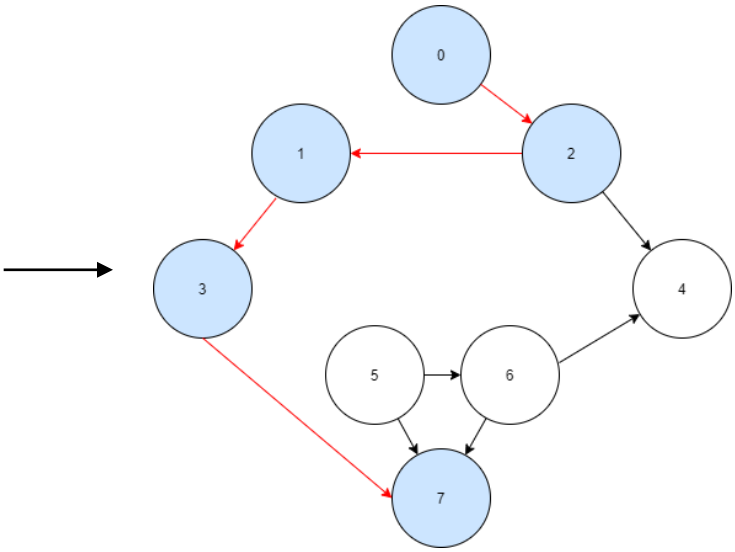
Go through the path:

$(0 \rightarrow 2)(2 \rightarrow 1)(1 \rightarrow 3)(3 \rightarrow 7)$

Then stop because vertex 7 hasn't any unvisited adjacency.

Visited vertex: 0,2,1,3,7

Unvisited vertex: 5,6,4



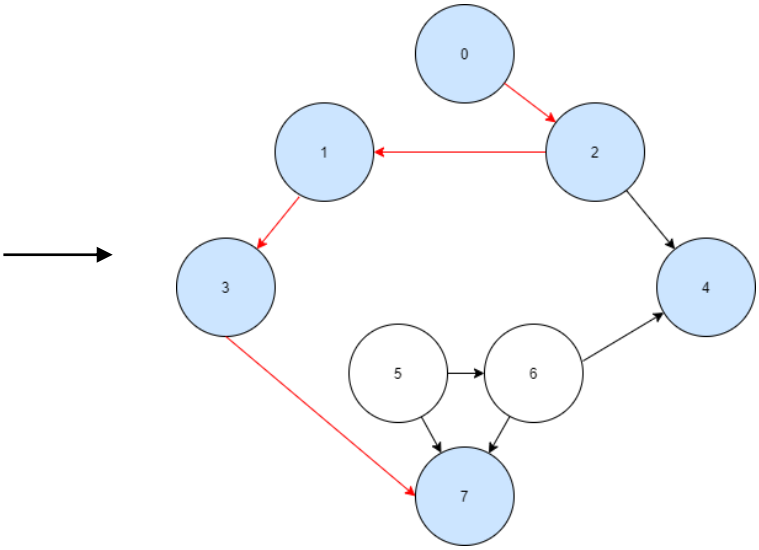
Step 2: From vertex 7, back to vertex 3,1,2,0. Stop at vertex 2 because vertex 2 has another unvisited adjacent vertex: vertex 4.

Go through the path: $(2 \rightarrow 4)$

Then stop because vertex 4 hasn't any unvisited adjacency.

Visited vertex: 0,2,1,3,7,4

Unvisited vertex: 5,6



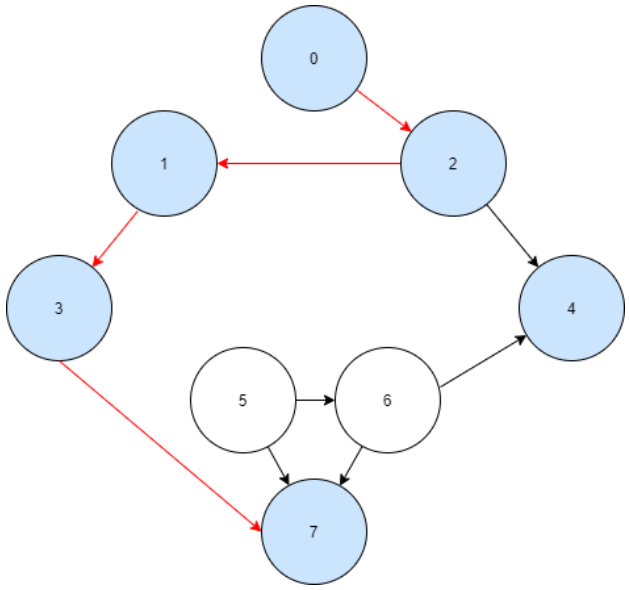
Step 3: From vertex 4, back to vertex 2,0. Stop at vertex 0 because this is the root and vertex 0 hasn't no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

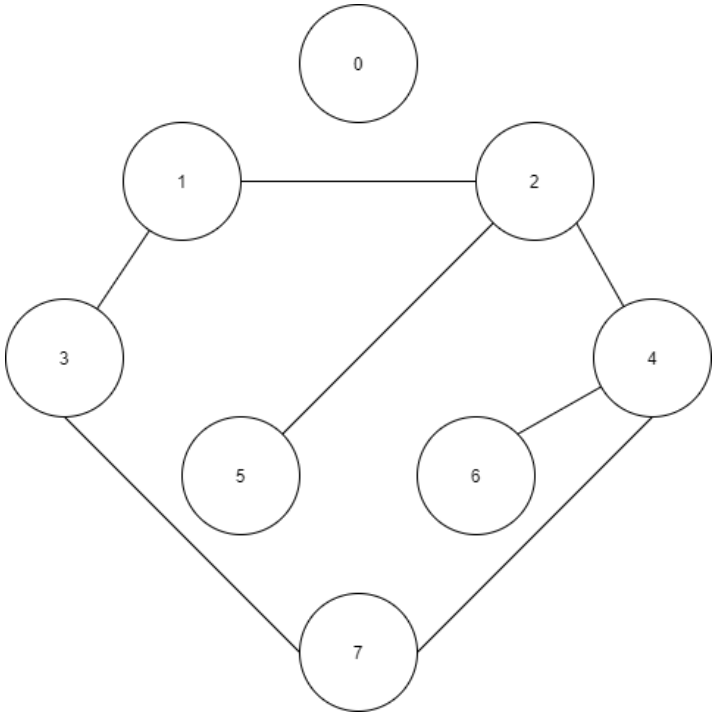
Some vertices (5,6) hadn't been visited because this is a directed graph.

Visited vertex: 0,2,1,3,7,4

Unvisited vertex: 5,6



Example 3:



Unconnected – Undirected graph

Step 1: Start DFS algorithm at vertex 1.

Go through the path:

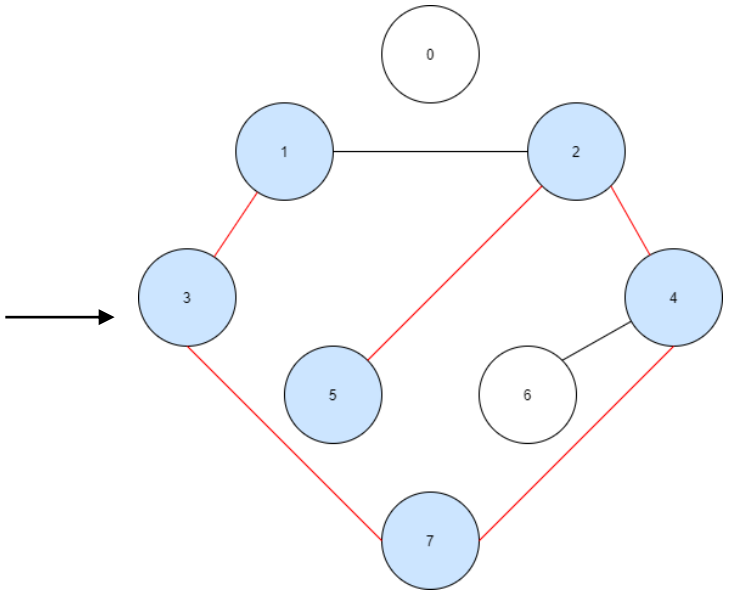
$$(1 \rightarrow 3)(3 \rightarrow 7)(7 \rightarrow 4)(4 \rightarrow 2)$$

$$(2 \rightarrow 5)$$

Then stop because vertex 5 hasn't any unvisited adjacency.

Visited vertex: 1,3,7,4,2,5

Unvisited vertex: 6,0



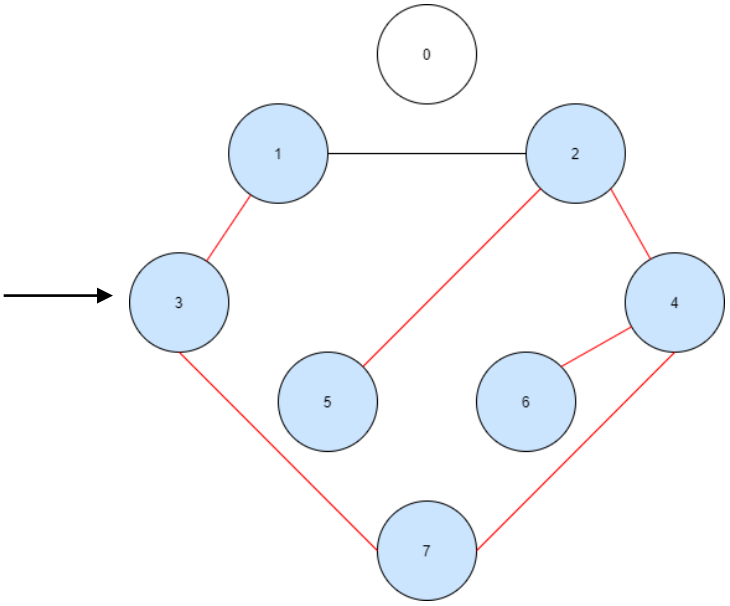
Step 2: From vertex 5, back to vertex 2,4. Stop at vertex 4 because vertex 4 has another unvisited adjacent vertex: vertex 6.

Go through the path: $(4 \rightarrow 6)$

Then stop because vertex 6 hasn't any unvisited adjacency.

Visited vertex: 1,3,7,4,2,5,6

Unvisited vertex: 0



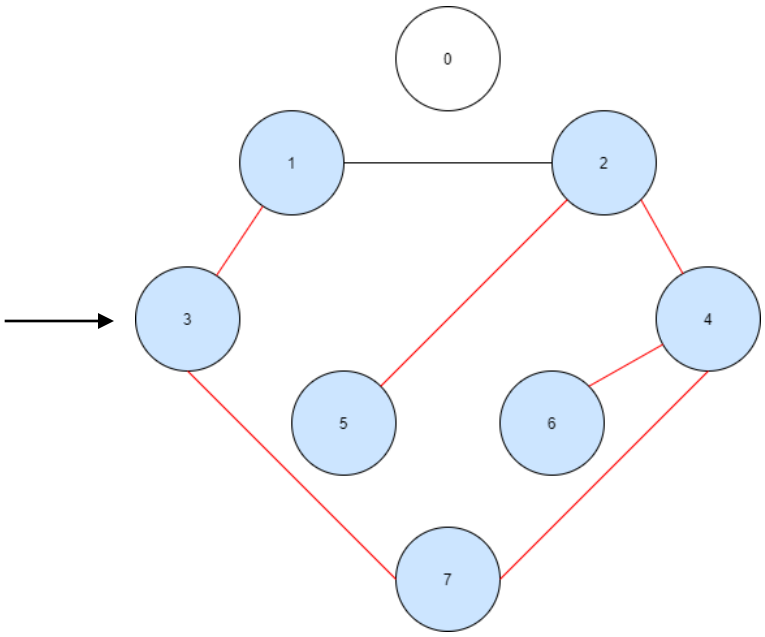
Step 3: From vertex 6, back to vertex 4,7,3,1. Stop at vertex 1 because this is the root and vertex 1 hasn't no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

Some vertices (0) hadn't been visited because this is an unconnected graph.

Visited vertex: 1,3,7,4,2,5,6

Unvisited vertex: 0



❖ **Breath-First Search (BFS):**

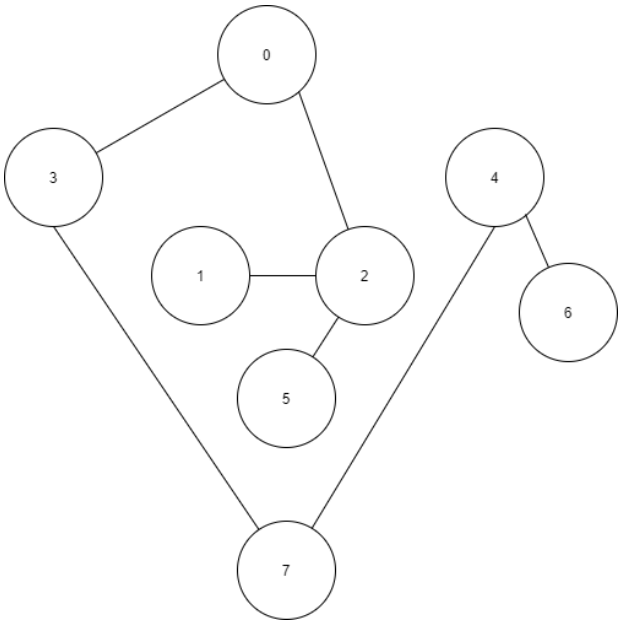
- Breath-First Search is a graph traversing algorithm that starts from a selected vertex (root), then visits all its unvisited adjacent vertices. After visiting all unvisited adjacent vertices, BFS moves towards the next-level unvisited adjacent vertex (it means that visiting the current vertex's grandchildrens). This process repeats until there are no more unvisited adjacent vertices which can be visited.
- In other words, BFS algorithms visits all sibling – vertices before it goes to deeper layers. It is the opposite of DFS algorithm.
- Remember to mark which vertex is visited because it's very easy to get in a endless loop if BFS visits a vertex many times or if the graph has a loop.
- With directed graph and unconnected graph, the BFS may be visit just a subgraph of the graph instead of the whole graph.
- In Adjacency List, the complexity of BFS is $O(V + E)$. In Adjacency Matrix, the complexity of BFS is $O(V^2)$. With V is the number of vertices, E is the number of edges in the graph.

- Pseudo-code of BFS using queue:

```
BFS(G,v)
        visited[] = false;
        Q = new queue;
        Enqueue(Q, v);
        Visited[v] = true;
        While(Q != empty)
                R = Dequeue(Q);
                For all vertices adjacent to R (adj[R][i]):
                        Visited[Adj[R][i]] = true;
                        Enqueue(Q, Adj[R][i]);
END BFS()
```

- Examples for BFS algorithm:

Example 1:



Connected – Undirected graph

Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices (3,2).

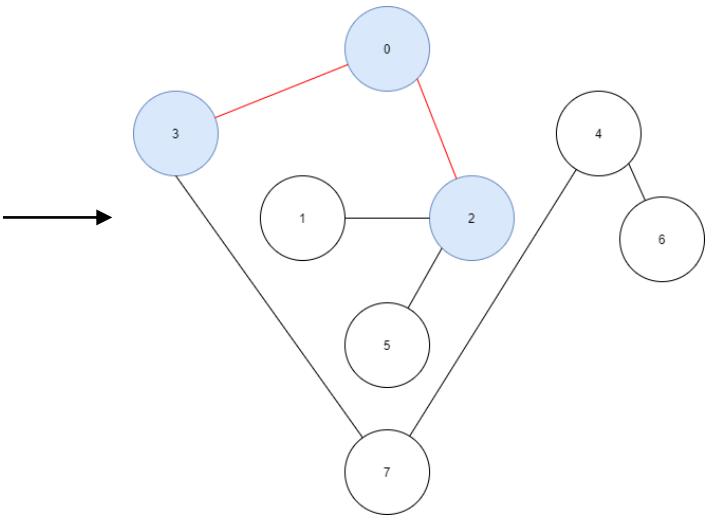
Go these next paths:

$(0 \rightarrow 2)(0 \rightarrow 3)$

Then stop because vertex 0 hasn't any more unvisited adjacency.

Visited vertex: 0,2,3

Unvisited vertex: 1,5,7,4,6



Step 2: From vertex 2, go to its unvisited adjacent vertices. (1,5)

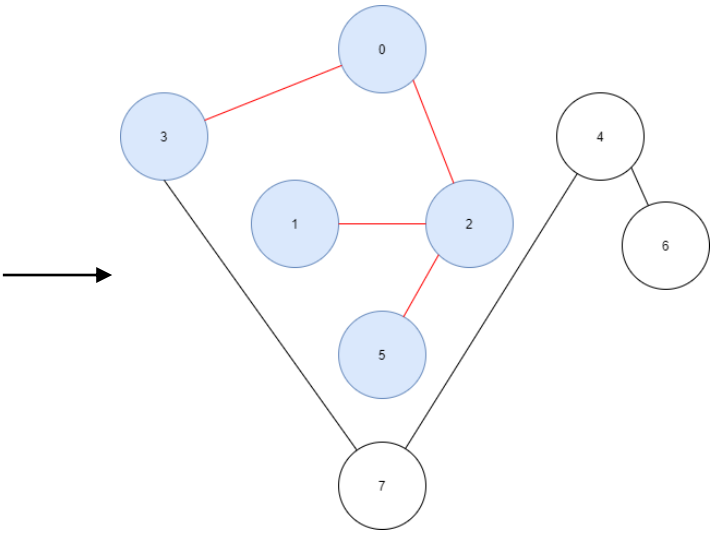
Go these next paths:

$(2 \rightarrow 1)(2 \rightarrow 5)$

Then stop because vertex 2 hasn't any more unvisited adjacency.

Visited vertex: 0,2,3,1,5

Unvisited vertex: 7,4,6



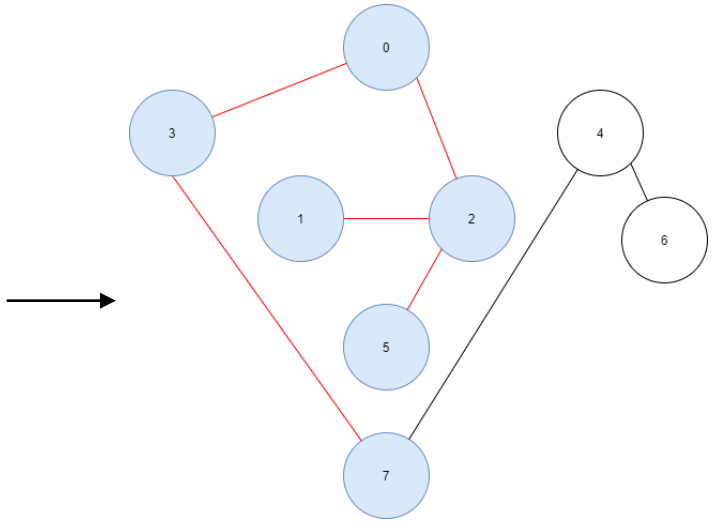
Step 3: Back to select vertex 3 (sibling of vertex 2), go to its unvisited adjacent vertices. (7)

Go these next paths:(3 → 7)

Then stop because vertex 3 hasn't any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7

Unvisited vertex: 4,6



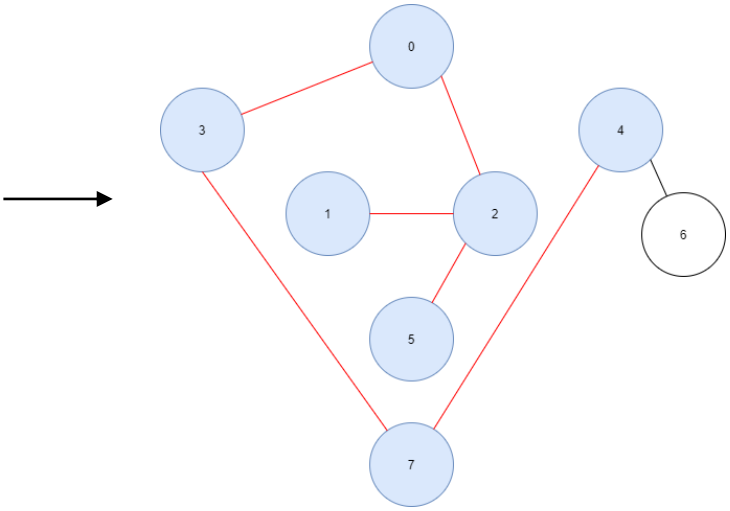
Step 4: Because vertex 2 children don't have any more unvisited adjacent vertex, continue with vertex 3 (sibling of vertex 2) child (vertex 7).

Go these next paths:(7 → 4)

Then stop because vertex 7 hasn't any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7,4

Unvisited vertex: 6



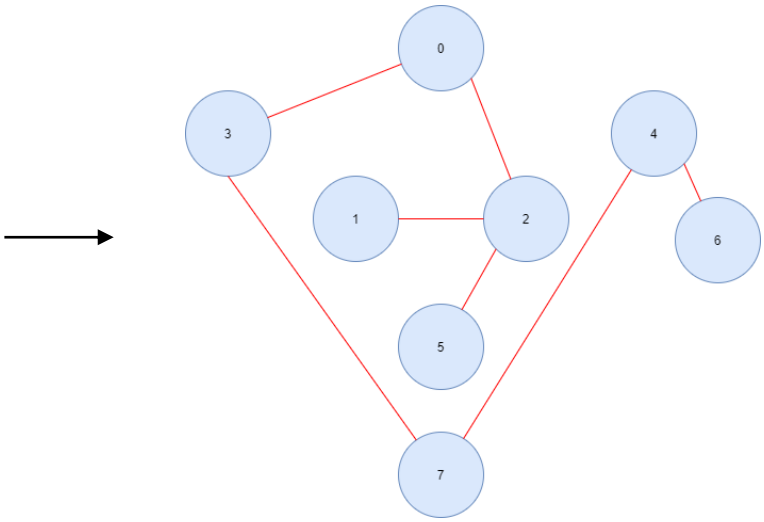
Step 5: Because vertex 7 doesn't have any sibling whose parent is vertex 3, continue with vertex 7 child (vertex 4)

Go these next paths:(4 → 6)

Then stop because vertex 4 hasn't any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7,4,6

Unvisited vertex: None

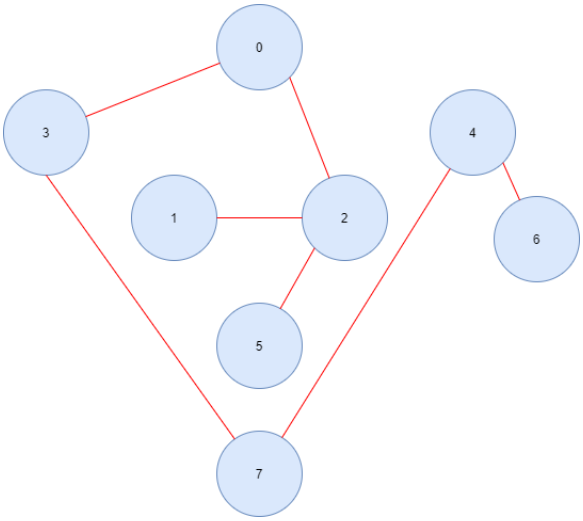


Step 6: Back to vertex 4, stop because vertex 4 doesn't have any other unvisited adjacency vertex.

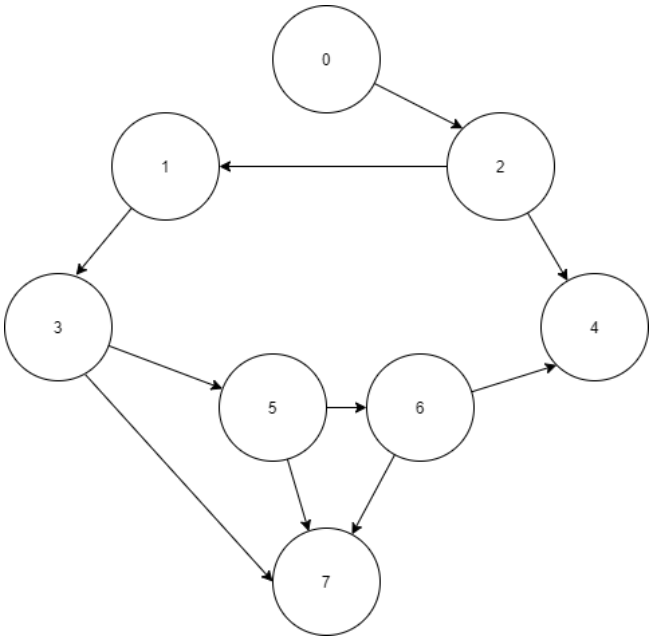
The algorithm now stops because there is no more unvisited vertex to visit.

Visited vertex: 0,2,3,1,5,7,4,6

Unvisited vertex: None



Example 2:



Connected – Directed graph

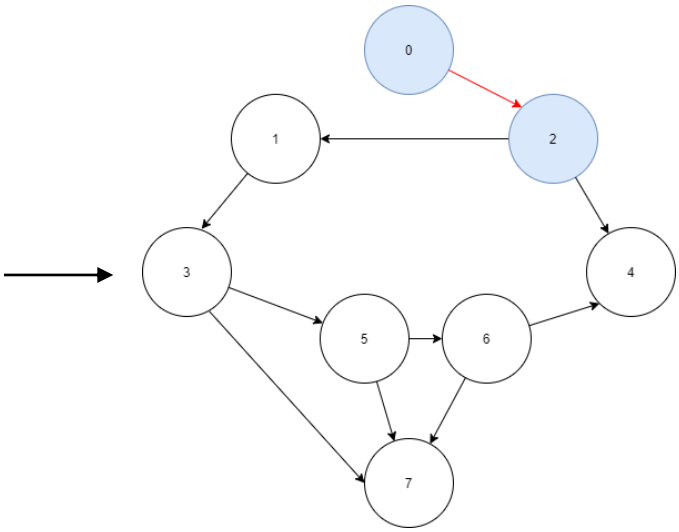
Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices. (2)

Go these next paths:(0 → 2)

Then stop because vertex 0 hasn't any more unvisited adjacency.

Visited vertex: 0,2

Unvisited vertex: 1,3,5,7,4,6



Step 2: Because vertex 2 is the only child of vertex 0. So the next step continue to deal with children of vertex 2. (1,4)

From vertex 2, go to its adjacent vertices (1,4).

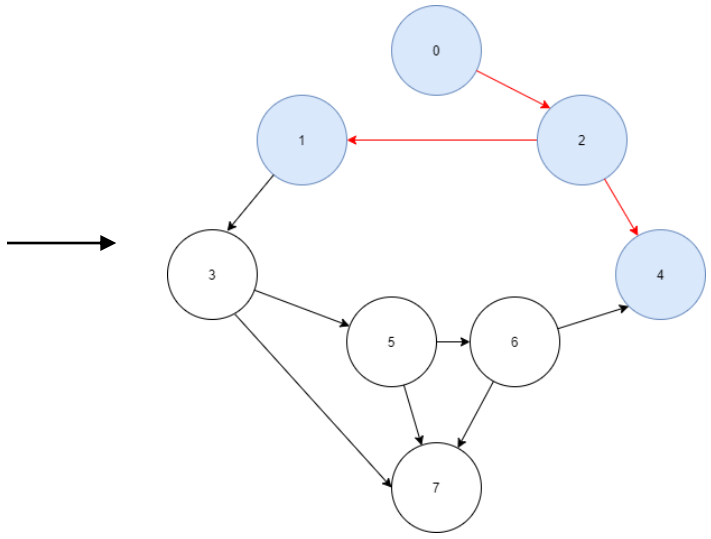
Go these next paths:

$$(2 \rightarrow 1)(2 \rightarrow 4)$$

Then stop because vertex 2 hasn't any more unvisited adjacency.

Visited vertex: 0,2,1,4

Unvisited vertex: 3,5,7,6



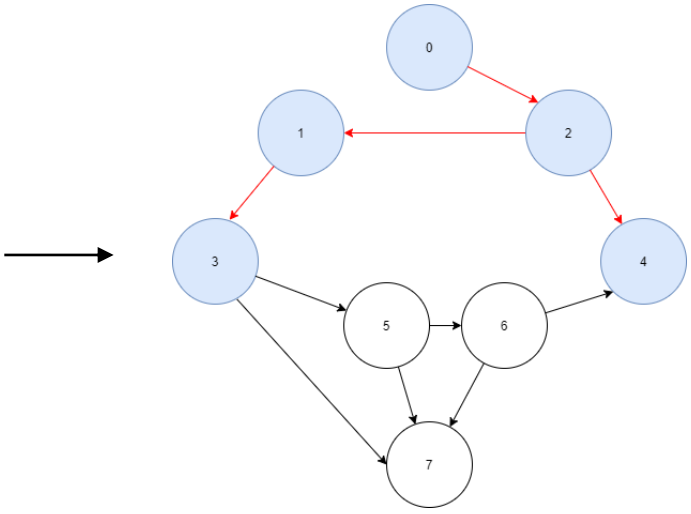
Step 3: From vertex 1, go to its adjacent vertices (3).

Go these next paths: $(1 \rightarrow 3)$

Then stop because vertex 1 hasn't any more unvisited adjacency.

Visited vertex: 0,2,1,4,3

Unvisited vertex: 5,7,6



Step 4: Next go to vertex 4 (sibling of vertex 1). But because vertex 4 doesn't have any unvisited adjacent vertex, so then continue deal with children of vertex 1 (3).

Because vertex 3 is the only child of vertex 1, next step will deal with children of vertex 3. (5,7)

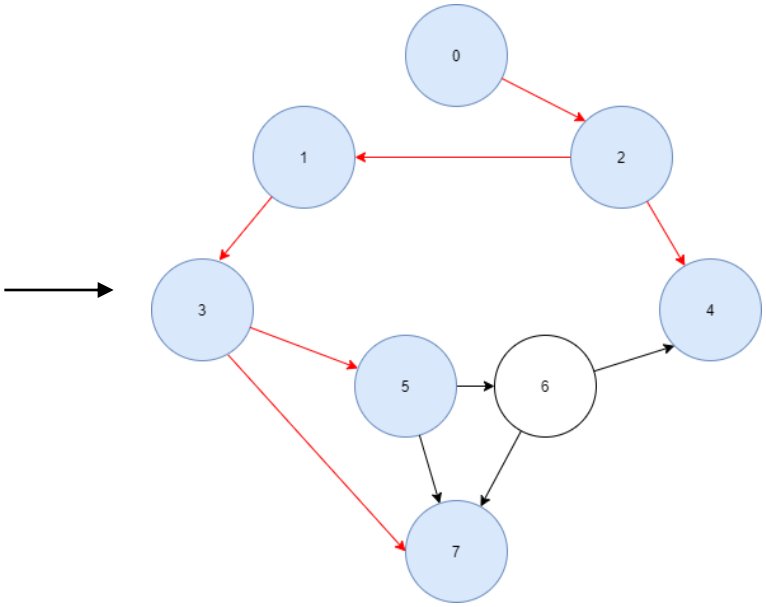
Go these next paths:

$$(3 \rightarrow 5)(3 \rightarrow 7)$$

Then stop because vertex 3 hasn't any more unvisited adjacency.

Visited vertex: 0,2,1,4,3,5,7

Unvisited vertex: 6



Step 5: From vertex 5, go to its unvisited adjacent vertices. (6)

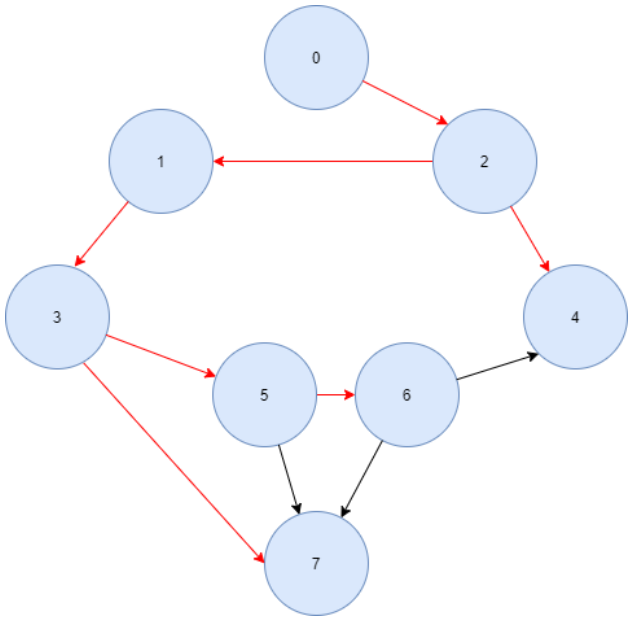
Go these next paths:

$$(5 \rightarrow 6)$$

Then stop because vertex 6 hasn't any more unvisited adjacency.

Visited vertex: 0,2,1,4,3,5,7,6

Unvisited vertex: None

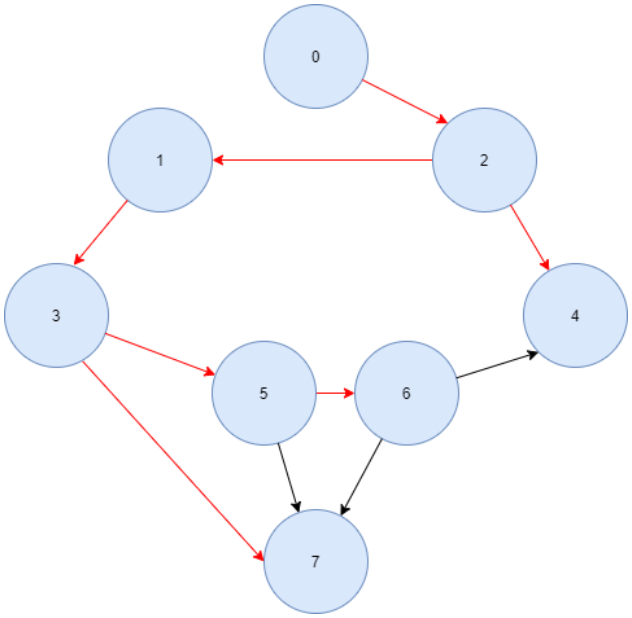


Step 6: Next go to vertex 7 (sibling of vertex 5). But because vertex 7 doesn't have any unvisited adjacent vertex, so then continue to deal with children of vertex 5 (6). But because vertex 6 doesn't have any unvisited adjacent vertex too, stop at vertex 6.

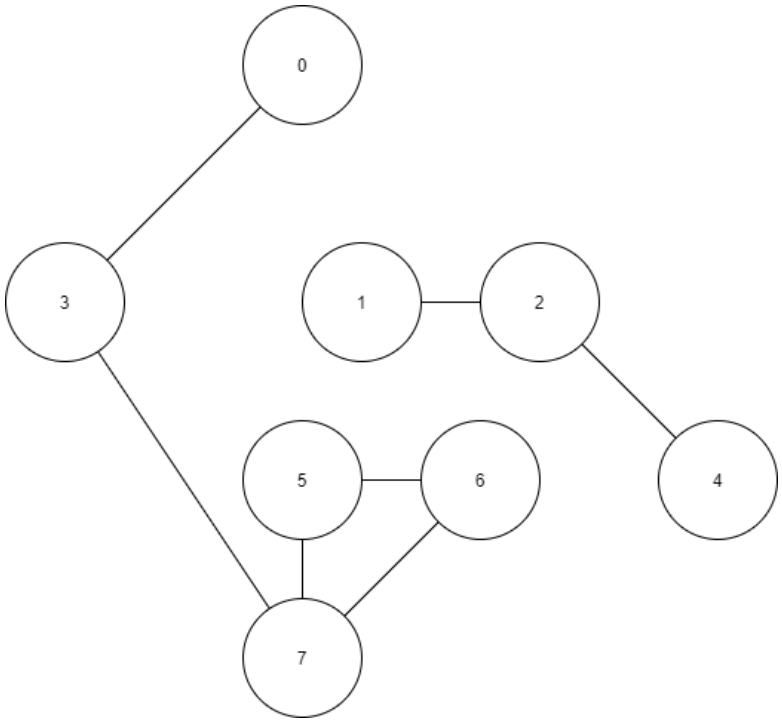
The algorithm now stops because there are no more unvisited vertex which can be visited.

Visited vertex: 0,2,1,4,3,5,7,6

Unvisited vertex: None



Example 3:



Unconnected – Undirected graph

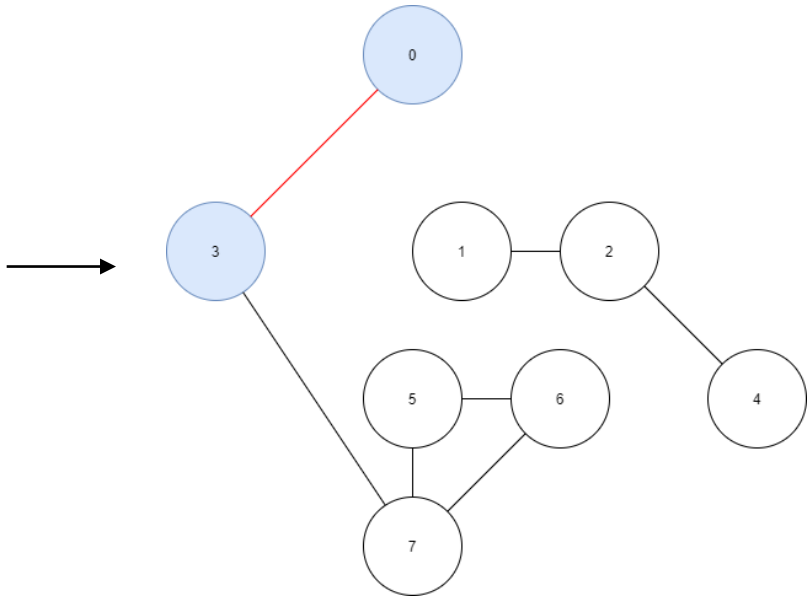
Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices. (3)

Go these next paths:(0 → 3)

Then stop because vertex 0 hasn't any more unvisited adjacency.

Visited vertex: 0,3

Unvisited vertex: 1,2,5,7,4,6



Step 2: Because vertex 3 is the only child of vertex 0. So, the next step continue to deal with children of vertex 3. (7)

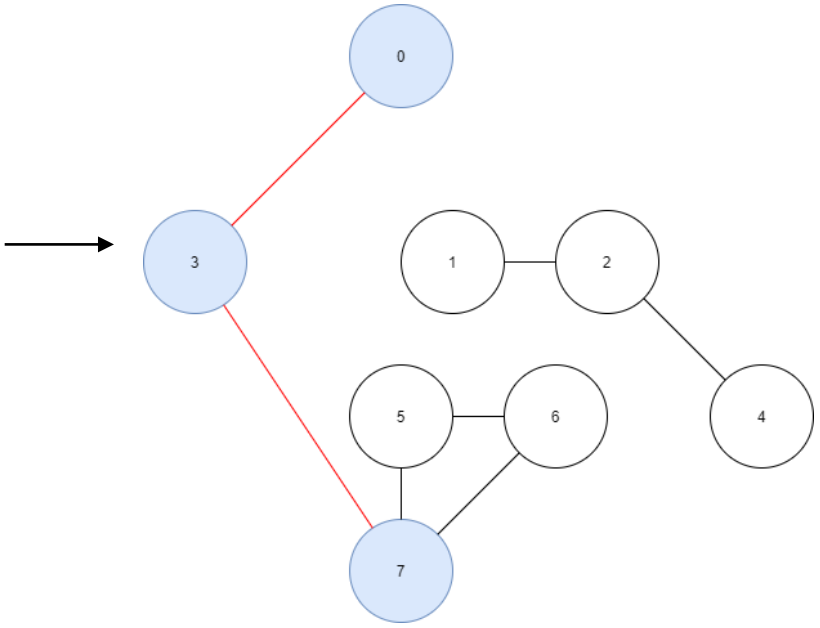
From vertex 3, go to its adjacent vertices (7).

Go these next paths: (3 → 7)

Then stop because vertex 3 hasn't any more unvisited adjacency.

Visited vertex: 0,3,7

Unvisited vertex: 1,2,5,4,6



Step 3: Because vertex 7 is the only child of vertex 3. So, the next step continues to deal with children of vertex 7. (5,6)

From vertex 7, go to its adjacent vertices (5,6).

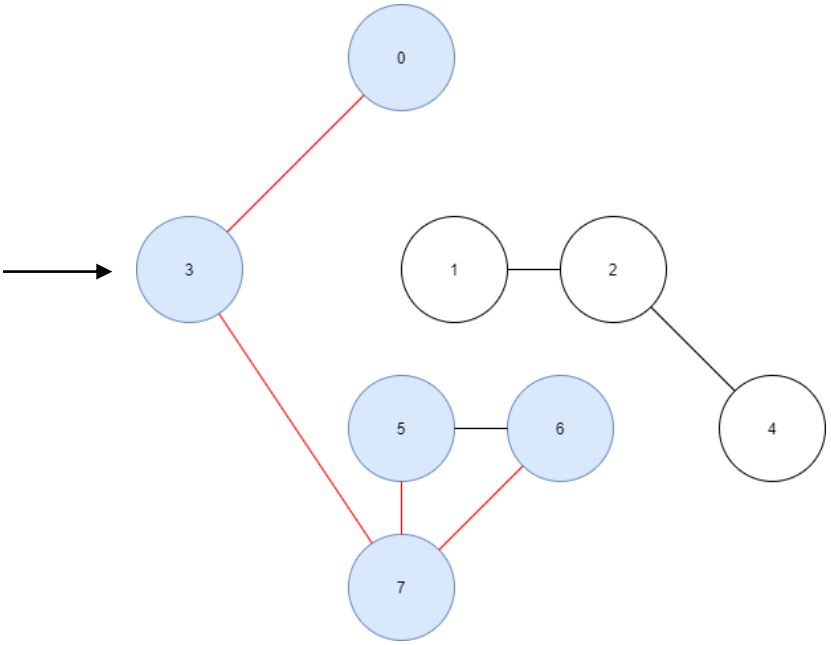
Go these next paths:

$$(7 \rightarrow 5)(7 \rightarrow 6)$$

Then stop because vertex 3 hasn't any more unvisited adjacency.

Visited vertex: 0,3,7

Unvisited vertex: 1,2,5,4,6



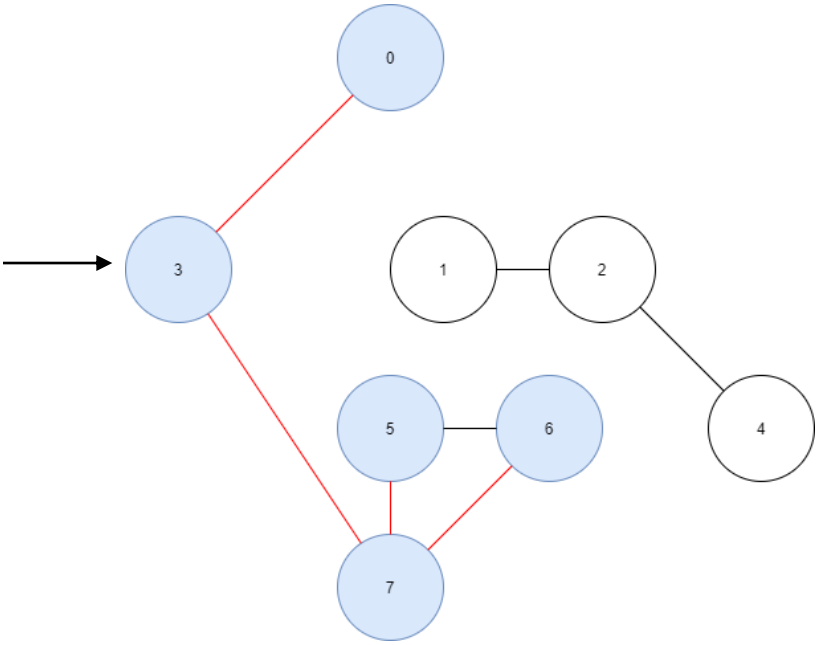
Step 4: Because vertex 5 and vertex 6 (children of vertex 7) don't have any unvisited adjacent vertex, stop at vertex 5 and 6.

The algorithm now stops because there is no more unvisited vertex which can be visited.

Visited vertex: 0,3,7,5,6

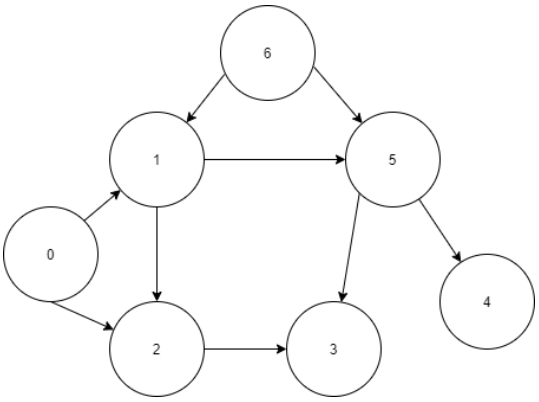
Unvisited vertex: 1,2,4

Because this is an unconnected graph, there are some vertices that can't be reached by BFS algorithm.

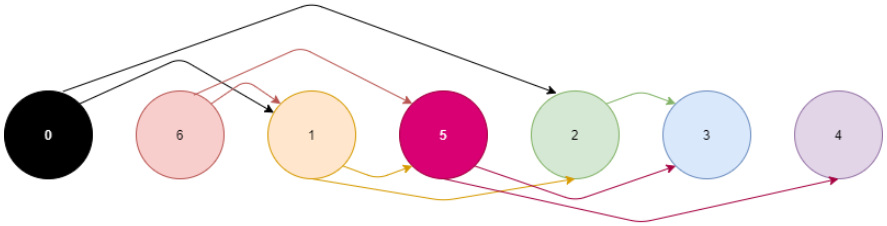


7) Topological Sorting:

- Topological Sorting for Directed Acyclic Graph (a directed without cycle graph) is a linear ordering of vertices in the graph. The edges after sorting will point in one direction left or right.



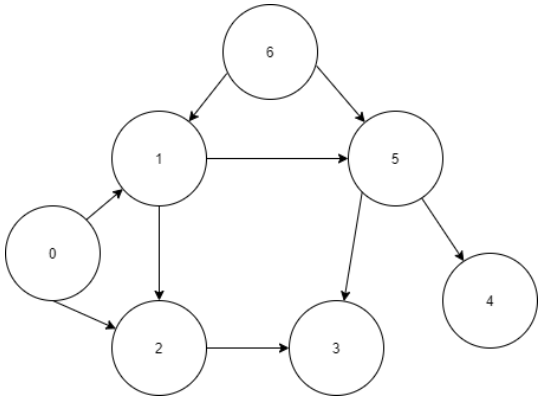
Unsorted Graph



Topological Sorted List

- Topological Sorting steps:
  - Step 1: Find a vertex that doesn't have any successor – which means the vertex that doesn't adjacent to any vertex in directed graph.
  - Step 2: Remove from the graph this vertex and all edges link with this vertex. Add this vertex to the beginning of the sorted list.
  - Step 3: Repeat step 2 until the graph empty. The sorted list now has all vertices of the old graph.
    (In examples, we can draw the edges of vertices at each step in the sorted list for easier to understand implementation).

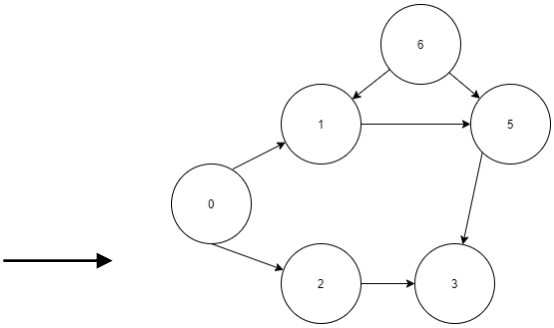
Examples 1:



Unsorted Graph

Step 1: Start Topological algorithm at vertex 4 because vertex 4 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.

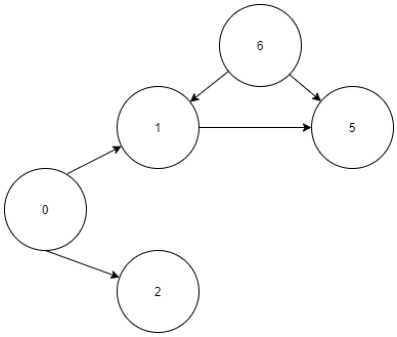


After delete 4

Ordered List

Step 2: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.

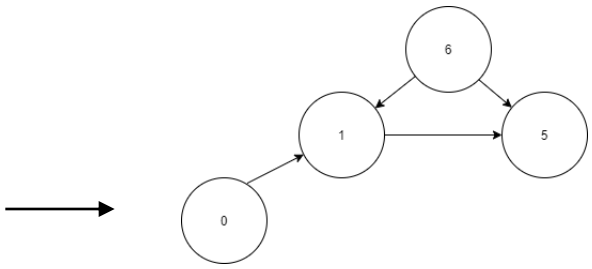


After delete 3

Ordered List

Step 3: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.

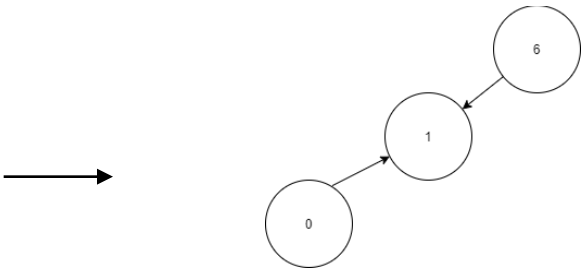


After delete 2

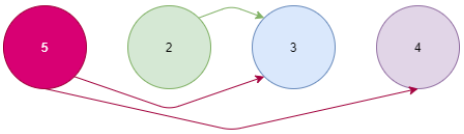
Ordered List

Step 4: Continually, Topological algorithm takes vertex 5 because vertex 5 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.



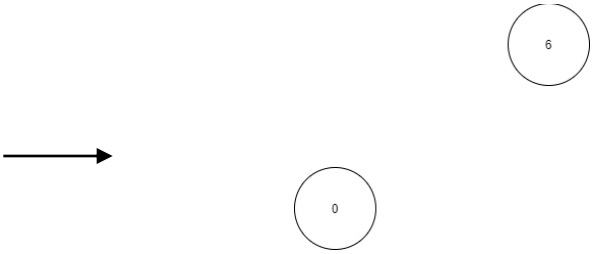
After delete 5



Ordered List

Step 5: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.



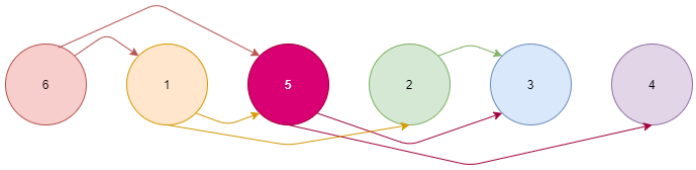
After delete 1

Ordered List

Step 6: Continually, Topological algorithm takes vertex 6 because vertex 6 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 6 in the beginning of the sorted list.

After delete 6



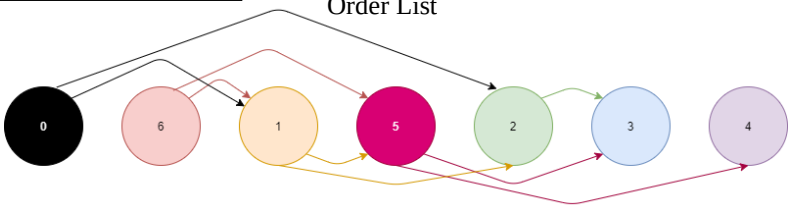
Ordered List

Step 7: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

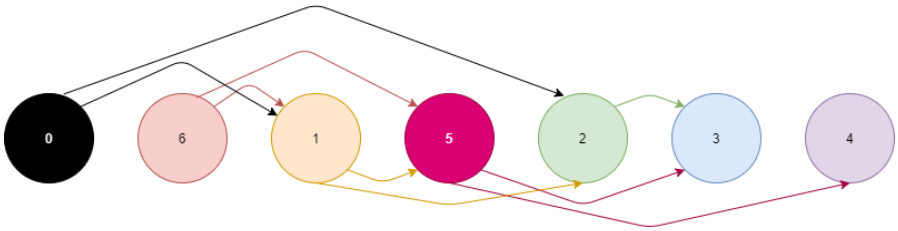
After delete 0: Empty graph

Order List

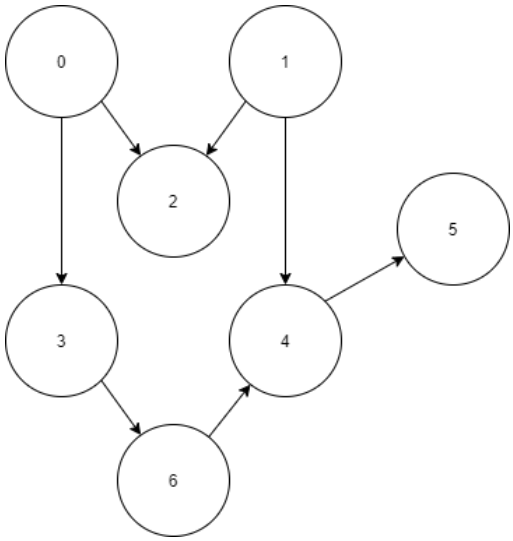


Step 8: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.

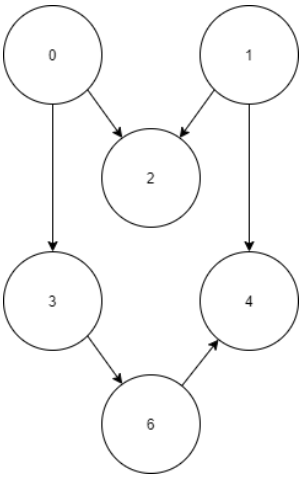


Examples 2:



Step 1: Start Topological algorithm at vertex 5 because vertex 5 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.

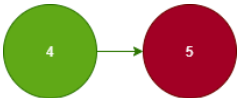


Ordered List

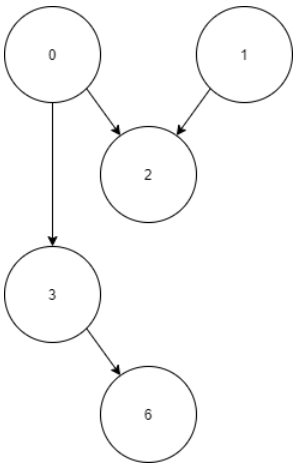
After delete 5

Step 2: Continually, Topological algorithm takes vertex 4 because vertex 4 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.



Ordered List

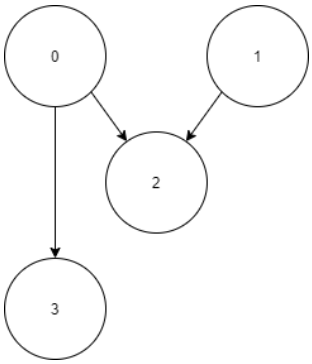


After delete 4

Step 3: Continually, Topological algorithm takes vertex 6 because vertex 6 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 6 in the beginning of the sorted list.

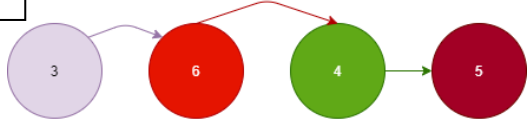
Ordered List



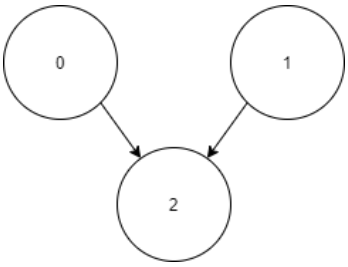
After delete 6

Step 4: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.



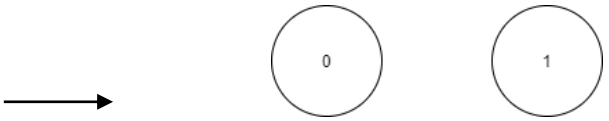
Order List



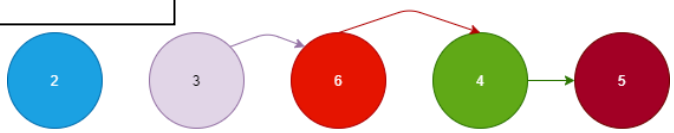
After delete 3

Step 5: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.



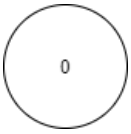
After delete 2



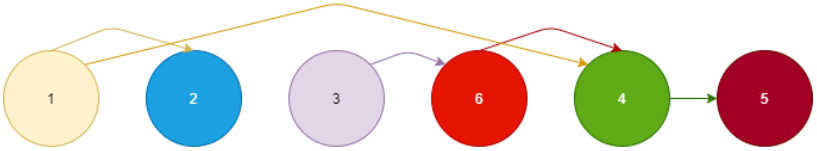
Ordered List

Step 6: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.



After delete 1

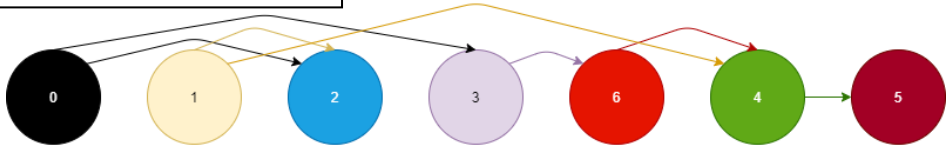


Ordered List

Step 7: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn't adjacent to any vertex.

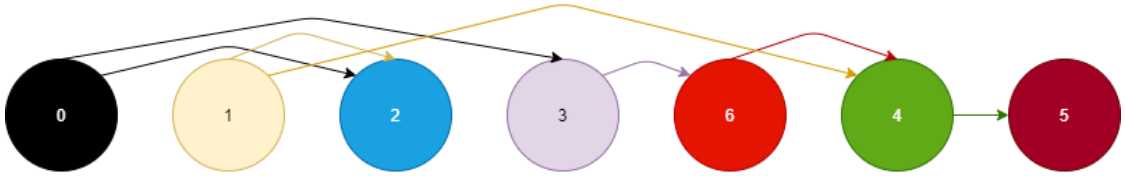
Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

After delete 0: Empty graph



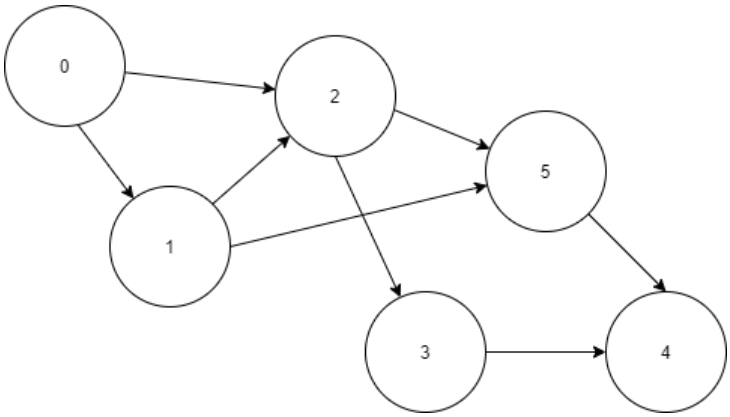
Step 8: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.



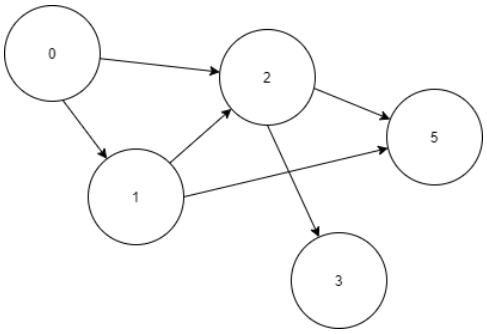
Ordered List

Example 3:



Step 1: Start Topological algorithm at vertex 4 because vertex 4 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.

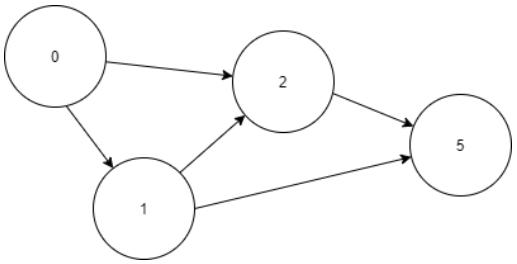


After delete 4

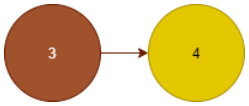
Ordered List

Step 2: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.



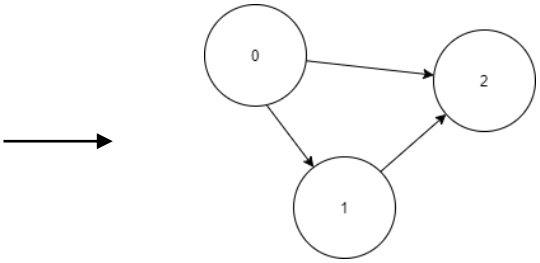
After delete 3



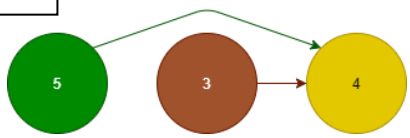
Ordered List

Step 3: Continually, Topological algorithm takes vertex 5 because vertex 5 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.



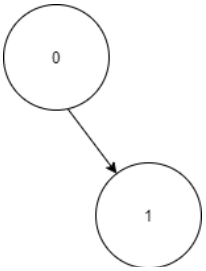
After delete 5



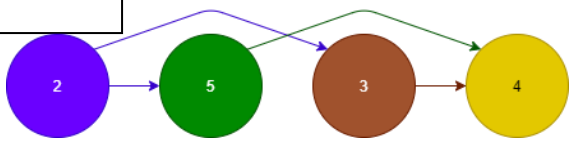
Ordered List

Step 4: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.



After delete 2

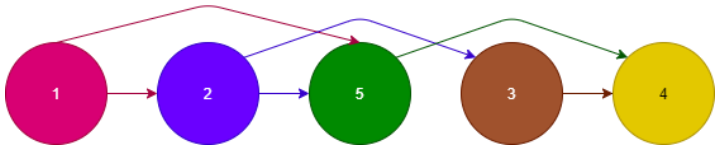


Ordered List

Step 5: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.

After delete 1

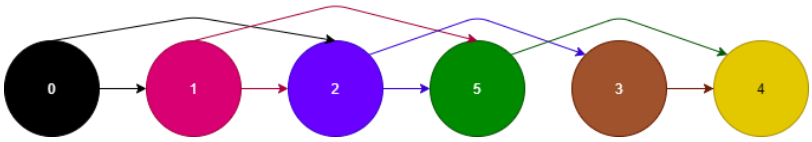


Ordered List

Step 6: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn't adjacent to any vertex.

Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

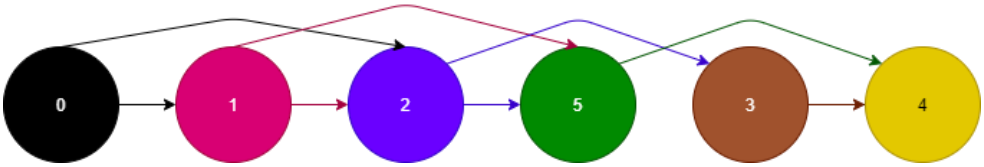
After delete 0: Empty graph



Ordered List

Step 7: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.

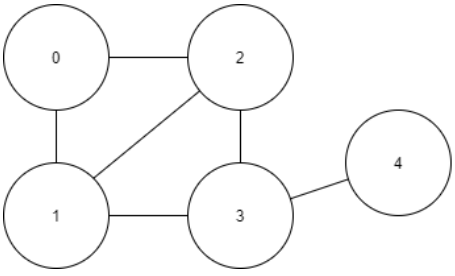


- The Topological algorithm sorting can be implemented with stack data structure. Pseudo-code:

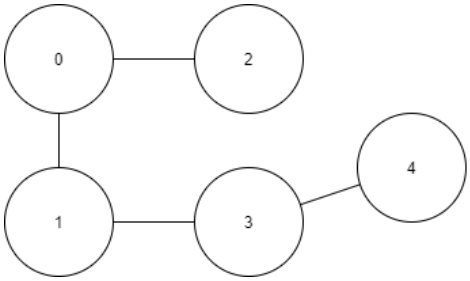
```
topoSorting(G, A) // G is the graph, A is the array of vertices after sorted
        visited[] = false;
        S = new Stack;
        For all unvisited vertice in the graph (v[])
                If (adjacent(v[i]) = 0) // v[i] doesn't have any successor
                Push(S, v)
                Visited[v] = true;
        Int k = 0;
        While (!S.empty())
                X = Pop(S);
                A[k++] = x;
                Visited[x] = true;
END topoSorting()
//After then, array A will be the list of ordered vertices.
```

8) Spanning tree:

- Spanning tree is a subgraph of Graph which includes all vertices covered with minimum possible number of edges to form a tree. A spanning tree has **n** vertices and **n-1** edges.
- Every connected undirected graph has at least one spanning tree. A disconnected graph can't draw any spanning tree.



Graph G

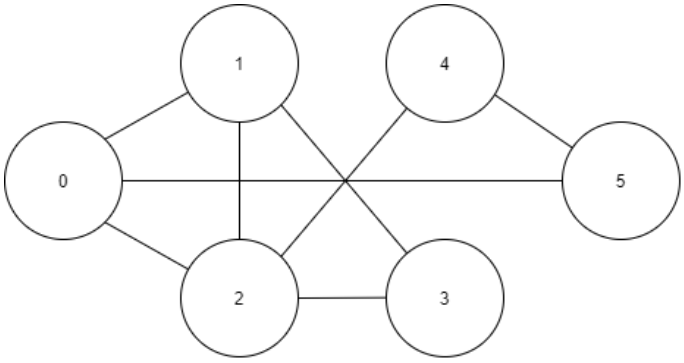


One spanning tree of G root 0

- With DFS algorithms, we can form a spanning tree by marked each edge which visited by DFS. Pseudo-code recursive DFS:

```
BFSspanningtree(G, root, tree) // root is the root vertex of the tree, where we start BFS
        If visited[] = false;
        Q = new Queue;
        Enqueue(Q,root)
        Visited[root] = true;
        While (Q!=empty)
                R = dequeue(Q);
                For all vertices adjacent to R (adj[R][i])
                        Visited[adj[r][i]] = true;
                        Insert in the tree the edge adj[R][i] between R
                        Enqueue(Q, adj[R][i]);
END BFSspanningtree()
```

- Example implementations for Spanning tree with DFS:



Step 1: Start DFS algorithm at vertex 0.

Go to vertex 2 (vertex 0's adjacent), 3,1:

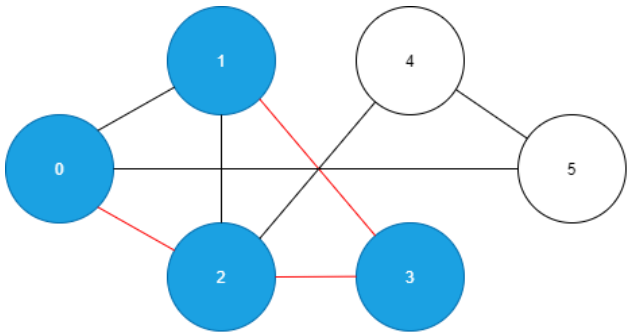
$$(0 \rightarrow 2)(2 \rightarrow 3)(3 \rightarrow 1)$$

Marked these edges (red lines).

Stop at vertex 1 because it doesn't have any unvisited vertices.

Visited vertices: 0,2,3,1

Unvisited vertices: 4,5



Step 2: From vertex 1, back to vertex 3,2. Continue DFS at vertex 2 because it has adjacent vertex. (4)

Go to vertex 2,4,5:

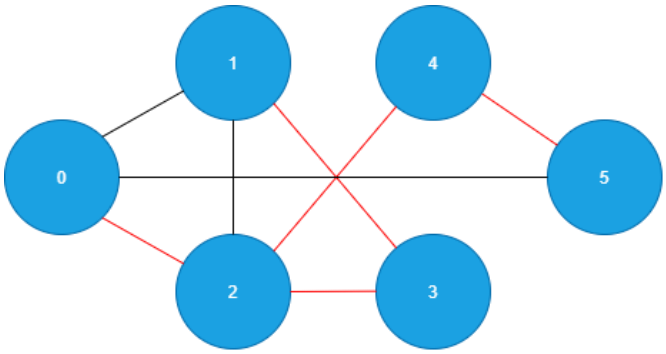
$$(2 \rightarrow 4)(4 \rightarrow 5)$$

Marked these edges (red lines).

Stop at vertex 5 because it doesn't have any unvisited vertices.

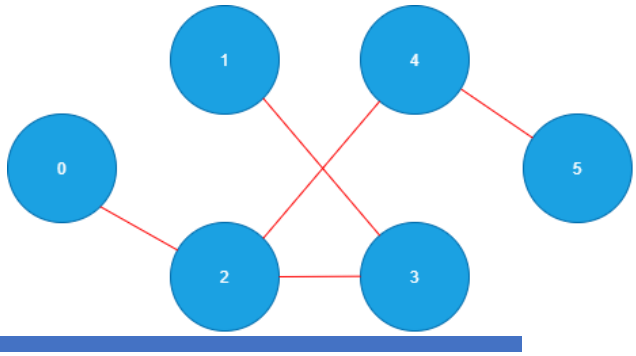
Visited vertices: 0,2,3,1,4,5

Unvisited vertices: None

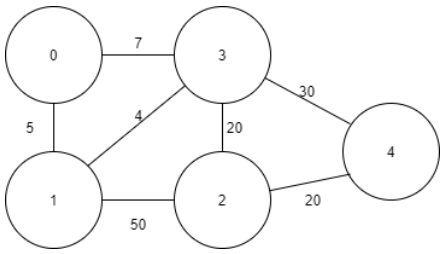


Step 3: Now DFS stops because all vertices are visited.

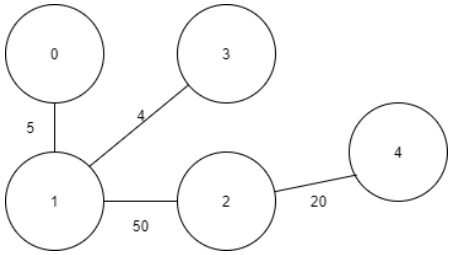
These marked edges with visited vertices form a tree. (Root is 0)



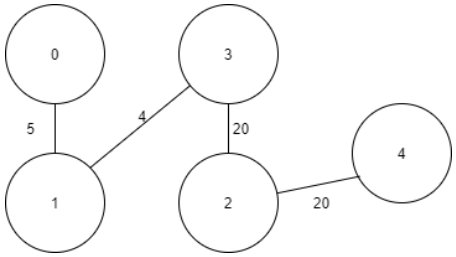
- With weighted graph, minimun spanning tree is a spanning tree that has less minimum total weight than all other spanning trees which can form in a same graph. In real life, this kind of spanning tree of weighted graph solves many problems about distance, traffic, data...



Graph G



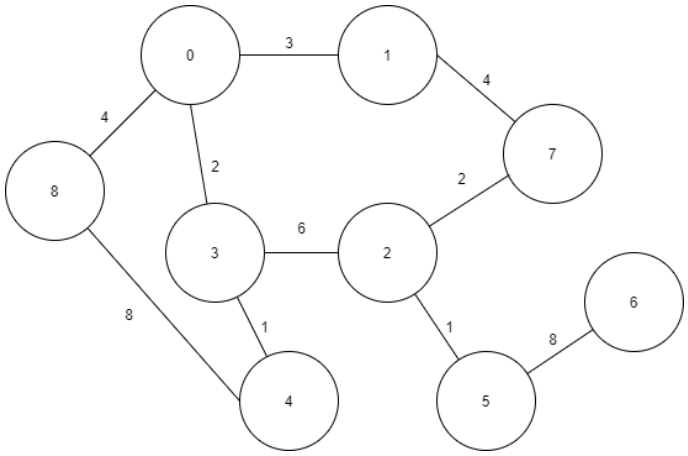
Spanning tree cost: 79



Minimum spanning tree cost: 49

- We can form a minimun spanning by using Prim's algorithms.
  - Step 1: Start Prim's algorithm at the selected root vertex. Add it in the tree as the root.
  - Step 2: Select the least cost edge which begin with a vertex in the current tree and end at the vertex not in the tree.
  - Step 3: Add the end vertex and the edge chosen in the tree.
  - Step 4: Repeat step 2 until all vertices of the graph are visited.
- Examples of Prim's algorithm:

Example 1:

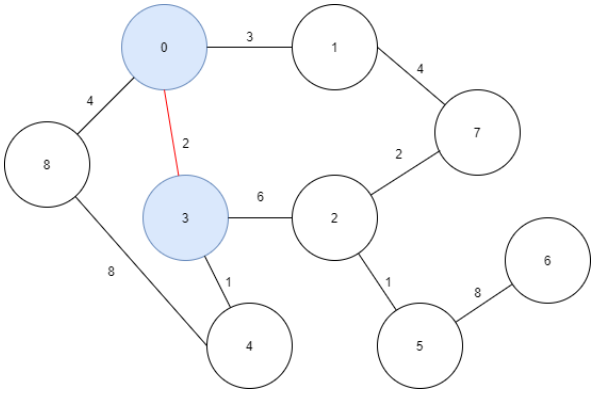


Step 1: Start DFS algorithm at vertex 0.

Now choose the edge that has least cost: $(0 \rightarrow 3)$ cost = 2.

Visited vertices: 0,3

Unvisited vertices: 1,2,4,5,6,7,8

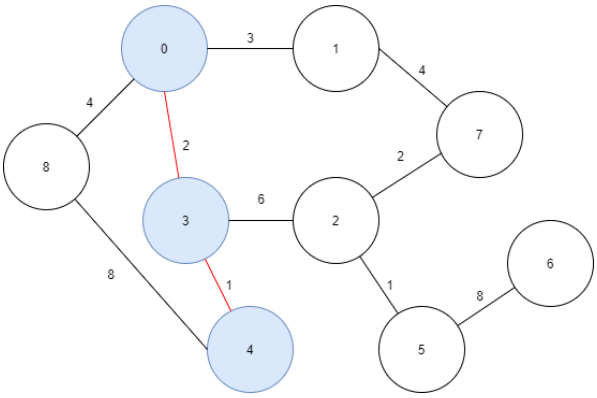


Step 2: Continually, choose the edge that has least cost among the vertices in the tree (0,3):

$(3 \rightarrow 4)$ cost = 1.

Visited vertices: 0,3,4

Unvisited vertices: 1,2,5,6,7,8

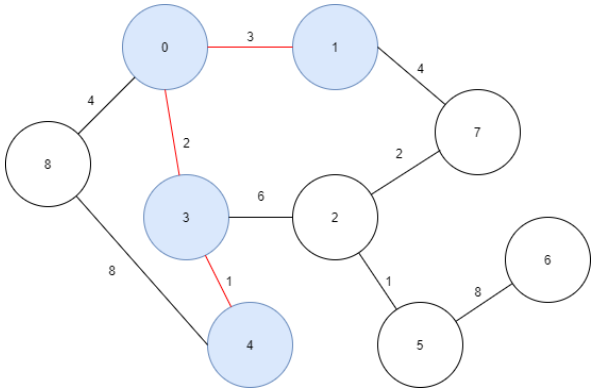


Step 3: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4):

(0 → 1) cost = 3.

Visited vertices: 0,3,4,1

Unvisited vertices: 2,5,6,7,8

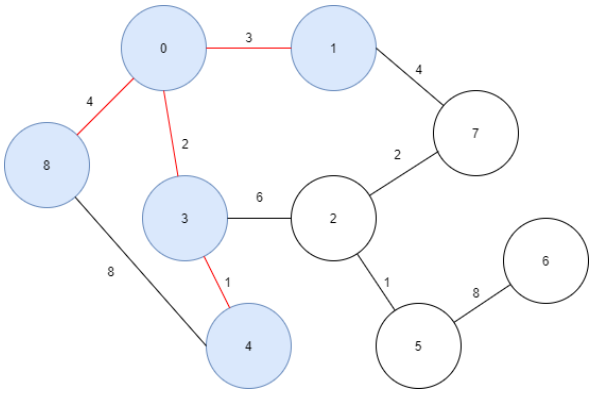


Step 4: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1):

(0 → 8) cost = 4.

Visited vertices: 0,3,4,1,8

Unvisited vertices: 2,5,6,7

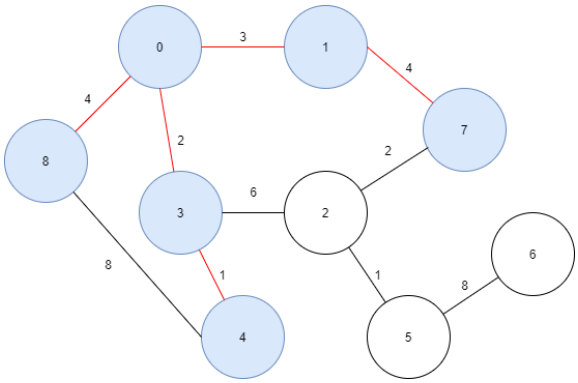


Step 5: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8):

(1 → 7) cost = 4.

Visited vertices: 0,3,4,1,8,7

Unvisited vertices: 2,5,6

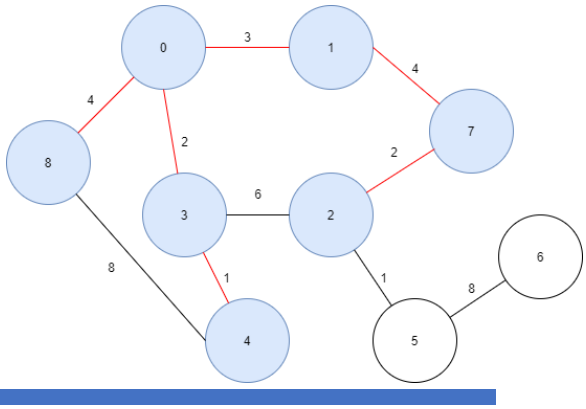


Step 6: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8):

(7 → 2) cost = 2.

Visited vertices: 0,3,4,1,8,7,2

Unvisited vertices: 5,6

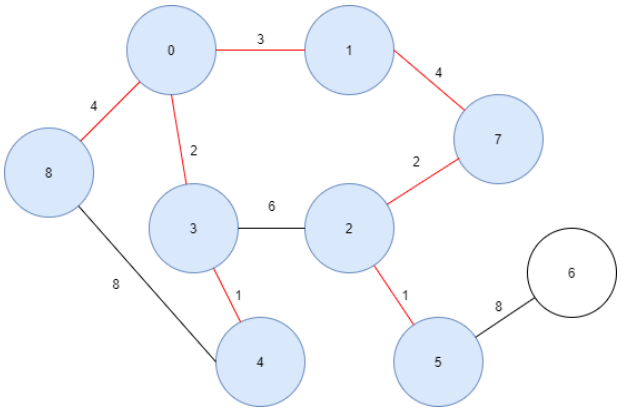


Step 7: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8,2):

(2 → 5) cost = 1.

Visited vertices: 0,3,4,1,8,7,2,5

Unvisited vertices: 6

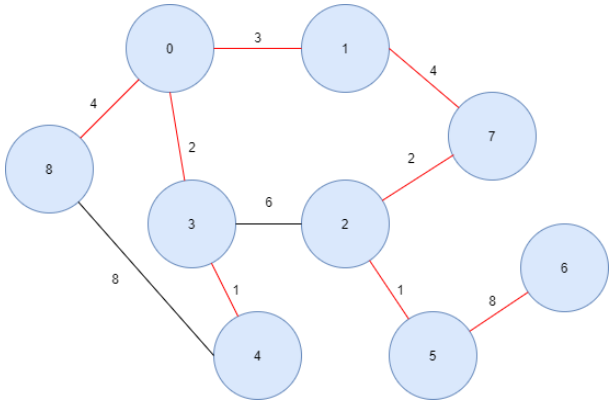


Step 8: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8,5):

(5 → 6) cost = 8.

Visited vertices: 0,3,4,1,8,7,2,5,6

Unvisited vertices: None

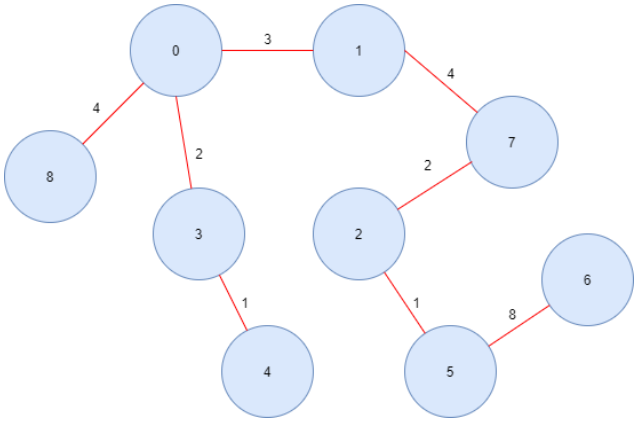


Step 9: Now the Prim's algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

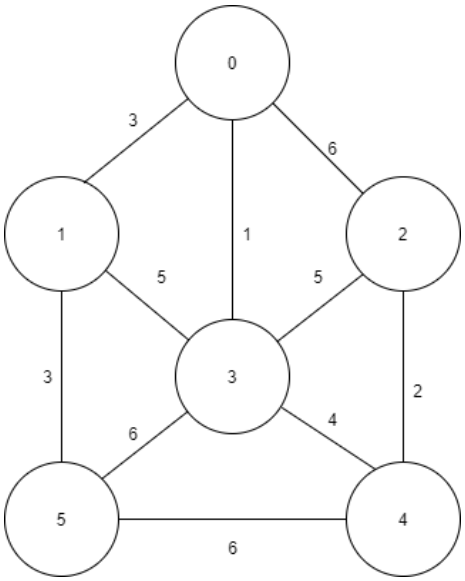
Visited vertices: 0,3,4,1,8,7,2,5,6

Unvisited vertices: None



Minimum spanning tree

Example 2:

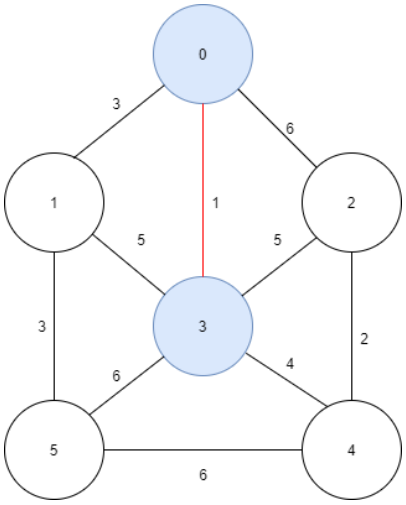


Step 1: Start DFS algorithm at vertex 0.

Now choose the edge that has least cost: (0 → 3) cost = 1.

Visited vertices: 0,3

Unvisited vertices: 1,2,4,5

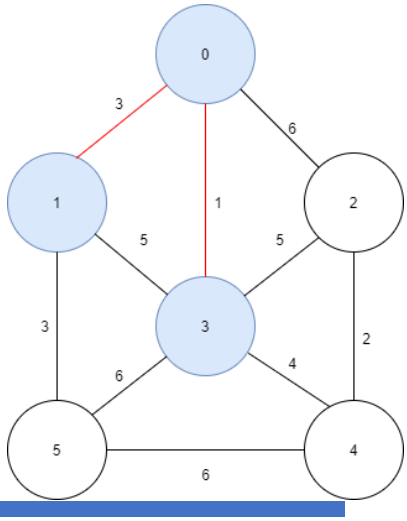


Step 2: Continually, choose the edge that has least cost among the vertices in the tree (0,3):

(0 → 1) cost = 3.

Visited vertices: 0,3,1

Unvisited vertices: 2,4,5

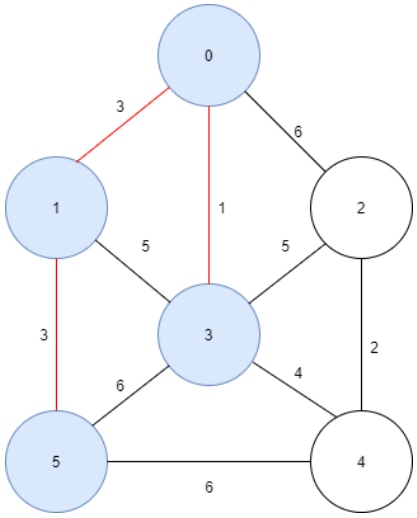


Step 3: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1):

(1 → 5) cost = 3.

Visited vertices: 0,3,1,5

Unvisited vertices: 2,4

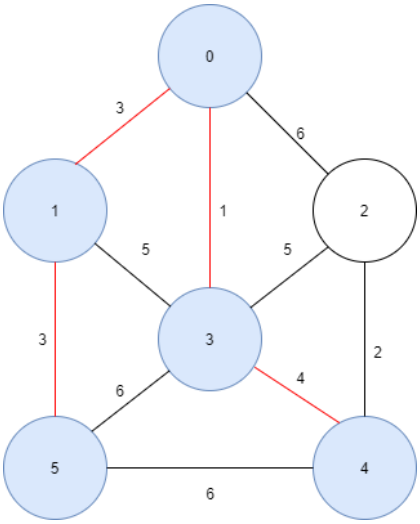


Step 4: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1,5,4):

(3 → 4) cost = 4.

Visited vertices: 0,3,1,5,4

Unvisited vertices: 2

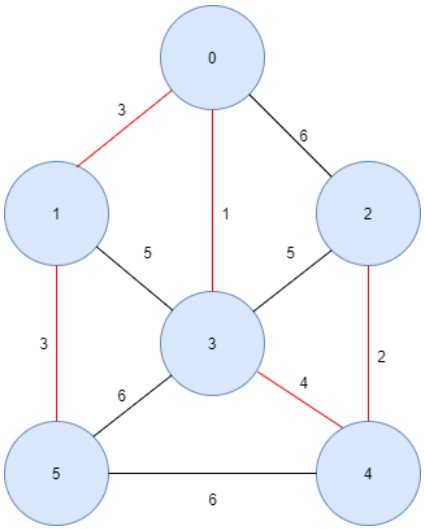


Step 5: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1,5,4,2):

(4 → 2) cost = 2.

Visited vertices: 0,3,1,5,4,2

Unvisited vertices: None

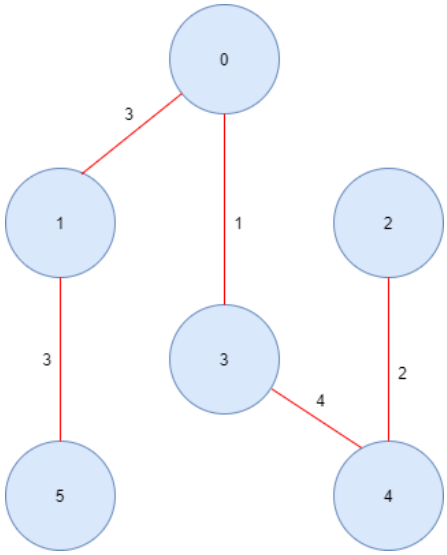


Step 6: Now the Prim's algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

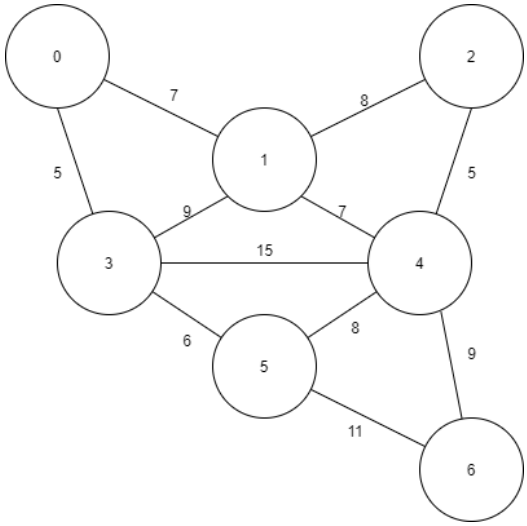
Visited vertices: 0,3,1,5,4,2

Unvisited vertices: None



Minimum spanning tree

Example 3:

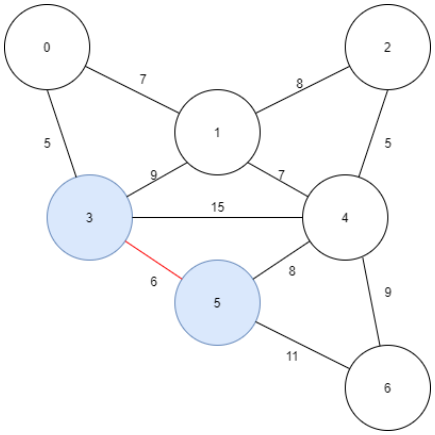


Step 1: Start DFS algorithm at vertex 5.

Now choose the edge that has least cost: (5 → 3) cost = 6.

Visited vertices: 5

Unvisited vertices: 0,1,2,4,6

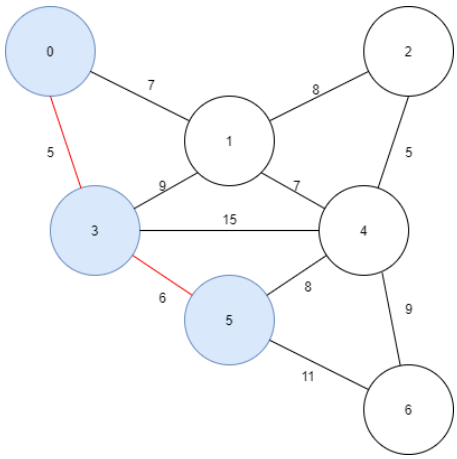


Step 2: Continually, choose the edge that has least cost among the vertices in the tree (5,3):

(3 → 0) cost = 5.

Visited vertices: 5,3,0

Unvisited vertices: 1,2,4,6

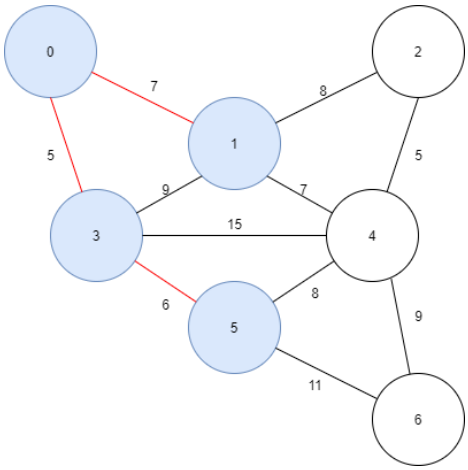


Step 3: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0):

(0 → 1) cost = 7.

Visited vertices: 5,3,0,1

Unvisited vertices: 2,4,6

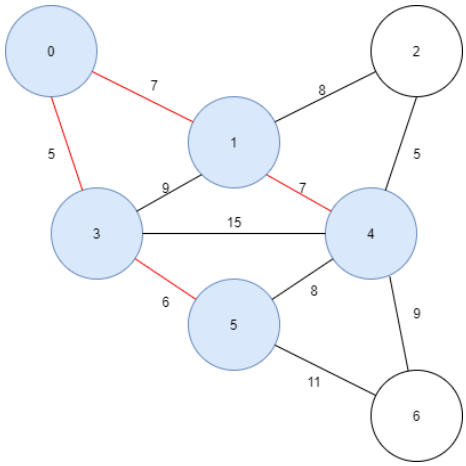


Step 4: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1):

(1 → 4) cost = 7.

Visited vertices: 5,3,0,1,4

Unvisited vertices: 2,6

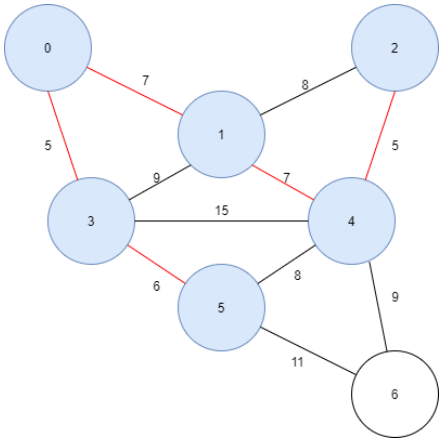


<u>Step 5:</u> Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1,4):

(4 → 2) cost = 5.

Visited vertices: 5,3,0,1,4,2

Unvisited vertices: 6

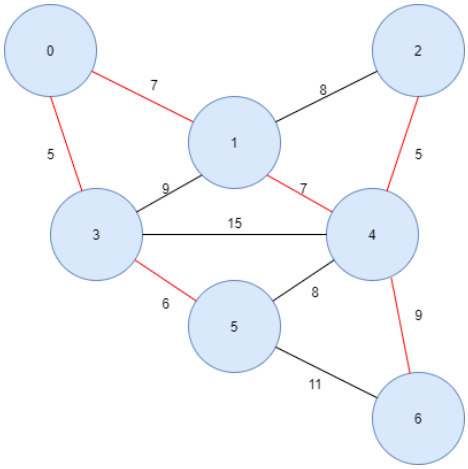


<u>Step 6:</u> Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1,4,2):

(4 → 6) cost = 9.

Visited vertices: 5,3,0,1,4,2,6

Unvisited vertices: None

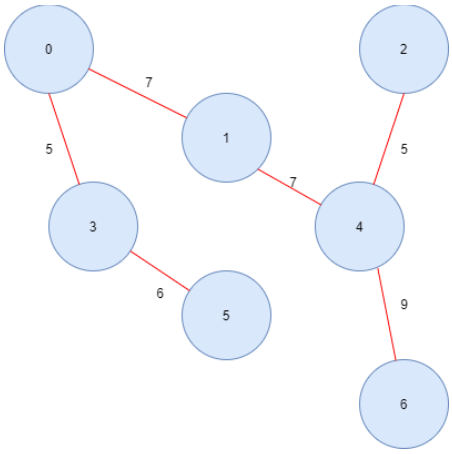


<u>Step 7:</u> Now the Prim's algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

Visited vertices: 5,3,0,1,4,2,6

Unvisited vertices: None



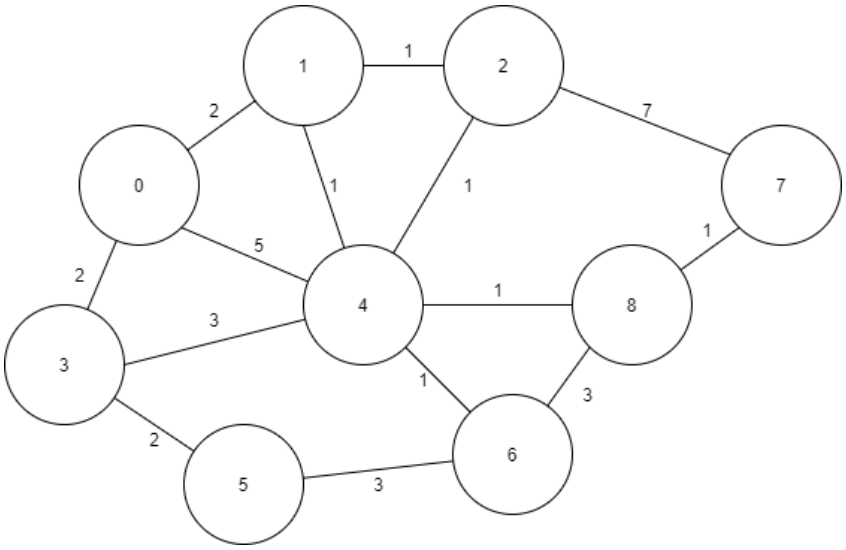
Minimum spanning tree

9) Shortest path:

- In unweighted graph, a shortesh path between two vertices is the path that has least edges. By this we can use BFS algorithms.
- In weighted graph, a shortest path between two vertices is the path that costs smallest weight.
- We can use E. Dijkstra's algorithm to find shortest path from one vertex to all other vertex in the graph.
- E. Dijkstra's algorithm steps:
  - Step 1: E. Dijkstra's algorithm includes one visited vertices set. Firstly, choose a start vertex. Mark it vertex as visited in the set. Mark its edge's weight to all other vertices. (if there isn't a edge, mark it as a simpol such as: ∞)
  - Step 2: Find a other vertex that is unvisited. Mark it as visited. Now update the path from the start vertex to other vertice in the graph by using edges of visited vertices set. (in examples, I will use a table to marked every update for easy understand).
  - Step 3: Repeat step 2 until all vertices of the graph is written in the visited vertices set.
- Pseudo-code of E. Dijkstra's algorithm:

```
DijkstraAlgorithm(G, root, list[])
//root is the start vertex, list[] is a array that keep path's weight from root to all other vertices in Graph
        Visited[] = false;
        Visited[root] = true;
        For all unvisited vertices (i):
                List[i] = edges from root to unvisited vertex;
        while (true)
                minVal = Min(list[]); // find min value in list[]
                Visited[Min(list[])] = true;
                for all unvisited vertices (i):
                        List[i] = Min weight path from root to i (use vertices in visited[] as intermediary;
                If Visited[] = true: break; // all vertices are visited, break
END DijkstraAlgorithm()
```

Example 1:



Start at root is vertex 0. Find shortest path from 0 to all other vertices.

Step 1: Mark all edges's weight from vertex 0 to all other vertices.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞

The visited vertex will be marked as red character.

Step 2: Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 2. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	**3**	2	**3**	∞	∞	∞	∞

Now the path from vertex 0 to vertex 4 is smaller than the old one weight $(0 \to 4) > (0 \to 1)(1 \to 4)$, so update it. The path from vertex 0 to vertex 2 appears $(0 \to 1)(1 \to 2)$ so update it also.

Step 3: Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 2. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	**4**	∞	∞	∞

Now, the path from vertex 0 to vertex 5 appears $(0 \rightarrow 3)(3 \rightarrow 5)$ so update it.

Step 4: Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 3. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	**10**	∞

Now, the path from vertex 0 to vertex 7 appears $(0 \rightarrow 1)(1 \rightarrow 2)(2 \rightarrow 7)$ so update it.

Step 5: Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 3. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	10	∞
0,1,3,2,4	0	2	3	2	3	4	**6**	10	**6**

Now, the path from vertex 0 to vertex 6 $(0 \rightarrow 4)(4 \rightarrow 6)$ and 8 $(0 \rightarrow 4)(4 \rightarrow 8)$ appears so update it.

Step 6: Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 4. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	10	∞
0,1,3,2,4	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5	0	2	3	2	3	4	6	10	6

Nothing update by using vertex 5.

Step 7: Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 6. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	10	∞
0,1,3,2,4	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5,8	0	2	3	2	3	4	6	**7**	6

Now the path from vertex 0 to vertex 7 is smaller than the old one
$(0 \to 4)(4 \to 8)(8 \to 7) < (0 \to 1)(1 \to 2)(2 \to 7)$, so update it.

Step 8: Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 6. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	10	∞
0,1,3,2,4	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5,8	0	2	3	2	3	4	6	7	6
0,1,3,2,4,5,8,6	0	2	3	2	3	4	6	7	6

Nothing updates by using vertex 6.

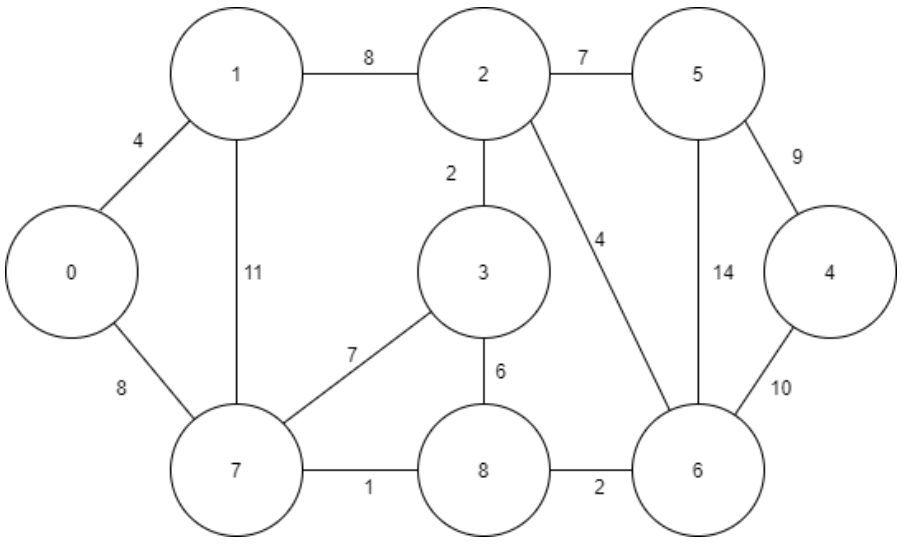
Step 9: Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 7. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	2	∞	2	5	∞	∞	∞	∞
0,1	0	2	3	2	3	∞	∞	∞	∞
0,1,3	0	2	3	2	3	4	∞	∞	∞
0,1,3,2	0	2	3	2	3	4	∞	10	∞
0,1,3,2,4	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5	0	2	3	2	3	4	6	10	6
0,1,3,2,4,5,8	0	2	3	2	3	4	6	7	6
0,1,3,2,4,5,8,6	0	2	3	2	3	4	6	7	6
0,1,3,2,4,5,8,6,7	0	2	3	2	3	4	6	7	6

Nothing updates by using vertex 7.

Step 10: Now Dijkstra's algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.

Example 2:



Start at root is vertex 0. Find shortest path from 0 to all other vertices.

Step 1: Mark all edges's weight from vertex 0 to all other vertices.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞

The visited vertex will be marked as red character.

Step 2: Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 4. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	**12**	∞	∞	∞	∞	8	∞

Now the path from vertex 0 to vertex 2 appears $(0 \rightarrow 1)(1 \rightarrow 2)$ so update it.

Step 3: Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 8. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	**15**	∞	∞	∞	8	**9**

Now, the path from vertex 0 to vertex 3 $(0 \rightarrow 7)(7 \rightarrow 3)$ and vertex 8 $(0 \rightarrow 7)(7 \rightarrow 8)$ appears so update it.

Step 4: Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 9. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	**11**	8	9

Now, the path from vertex 0 to vertex 6 appears $(0 \rightarrow 7)(7 \rightarrow 8)(8 \rightarrow 6)$ so update it.

Step 5: Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 11. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	11	8	9
0,1,7,8,6	0	4	12	15	**21**	**25**	11	8	9

Now, the path from vertex 0 to vertex 4 $(0 \to 7)(7 \to 8)(8 \to 6)(6 \to 4)$ and 5 $(0 \to 7)(7 \to 8)(8 \to 6)(6 \to 5)$ appears so update it.

Step 6: Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 12. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	11	8	9
0,1,7,8,6	0	4	12	15	21	25	11	8	9
0,1,7,8,6,2	0	4	12	**14**	21	**19**	11	8	9

Now the paths from vertex 0 to vertex 3 and vertex 5 are smaller than the old one $(0 \to 1)(1 \to 2)(2 \to 3) < (0 \to 7)(7 \to 3)$, $(0 \to 1)(1 \to 2)(2 \to 5) < (0 \to 7)(7 \to 8)(8 \to 6)(6 \to 5)$ so update it.

Step 7: Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 14. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	11	8	9
0,1,7,8,6	0	4	12	15	21	25	11	8	9
0,1,7,8,6,2	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3	0	4	12	14	21	19	11	8	9

Nothing updates by using vertex 3.

Step 8: Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 19. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	11	8	9
0,1,7,8,6	0	4	12	15	21	25	11	8	9
0,1,7,8,6,2	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3,5	0	4	12	14	21	19	11	8	9

Nothing updates by using vertex 5.

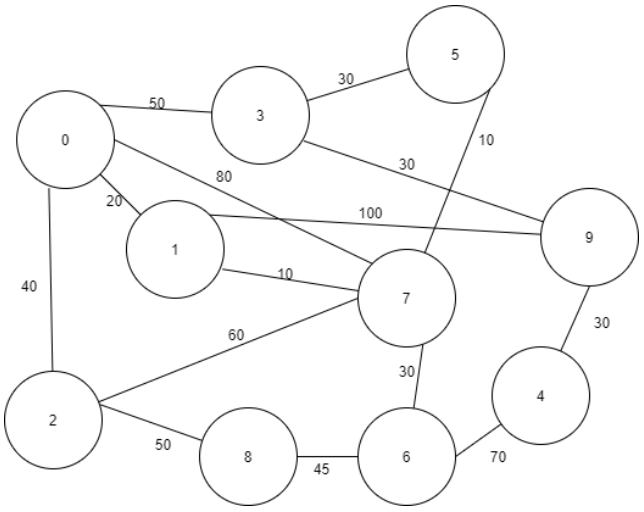
Step 9: Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 21. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8
0	0	4	∞	∞	∞	∞	∞	8	∞
0,1	0	4	12	∞	∞	∞	∞	8	∞
0,1,7	0	4	12	15	∞	∞	∞	8	9
0,1,7,8	0	4	12	15	∞	∞	11	8	9
0,1,7,8,6	0	4	12	15	21	25	11	8	9
0,1,7,8,6,2	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3,5	0	4	12	14	21	19	11	8	9
0,1,7,8,6,2,3,5,4	0	4	12	14	21	19	11	8	9

Nothing updates by using vertex 4.

Step 10: Now Dijkstra's algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.

Example 3:



Start at root is vertex 0. Find shortest path from 0 to all other vertices.

Step 1: Mark all edges's weight from vertex 0 to all other vertices.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞

The visited vertex will be marked as red character.

Step 2: Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 20. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	**30**	∞	**120**

Now the path from vertex 0 to vertex 9 appears $(0 \rightarrow 1)(1 \rightarrow 9)$ so update it. The path from vertex 0 to vertex 7 is smaller than the old one $(0 \rightarrow 1)(1 \rightarrow 7) < (0 \rightarrow 7)$, update it also.

Step 3: Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 30. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	**40**	**60**	30	∞	120

Now the paths from vertex 0 to vertex 5 and vertex 6 appear $(0 \rightarrow 1)(1 \rightarrow 7)(7 \rightarrow 5)$, $(0 \rightarrow 1)(1 \rightarrow 7)(7 \rightarrow 6)$, update them.

Step 4: Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 40. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	**90**	120

Now, the path from vertex 0 to vertex 8 appears $(0 \rightarrow 2)(2 \rightarrow 8)$ so update it.

Step 5: Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 40. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120

Nothing changes by using vertex 5.

Step 6: Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 50. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5,3	0	20	40	50	∞	40	60	30	90	**80**

Now the path from vertex 0 to vertex 9 is smaller than the old one so updates it. $(0 \to 3)(3 \to 9) < (0 \to 1)(1 \to 9)$

Step 7: Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 60. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5,3	0	20	40	50	∞	40	60	30	90	80
0,1,7,2,5,3,6	0	20	40	50	**130**	40	60	30	90	80

Now the path from vertex 0 to vertex 4 appears so update it.

$(0 \to 1)(1 \to 7)(7 \to 6)(6 \to 4)$

Step 8: Continually, choose smallest path weight of a vertex which unvisited: vertex 9 with cost 80. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5,3	0	20	40	50	∞	40	60	30	90	80
0,1,7,2,5,3,6	0	20	40	50	130	40	60	30	90	80
0,1,7,2,5,3,6,9	0	20	40	50	**110**	40	60	30	90	80

Now the path from vertex 0 to vertex 4 is smaller than the old one $(0 \to 3)(3 \to 9)(9 \to 4) < (0 \to 1)(1 \to 7)(7 \to 6)(6 \to 4)$ so update it.

Step 9: Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 90. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5,3	0	20	40	50	∞	40	60	30	90	80
0,1,7,2,5,3,6	0	20	40	50	130	40	60	30	90	80
0,1,7,2,5,3,6,9	0	20	40	50	110	40	60	30	90	80
0,1,7,2,5,3,6,9,8	0	20	40	50	110	40	60	30	90	80

Nothing updates by using vertex 8.

Step 10: Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 110. Add it in visited set and update shortest path.

VisitedSet/Vertex	0	1	2	3	4	5	6	7	8	9
0	0	20	40	50	∞	∞	∞	80	∞	∞
0,1	0	20	40	50	∞	∞	∞	30	∞	120
0,1,7	0	20	40	50	∞	40	60	30	∞	120
0,1,7,2	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5	0	20	40	50	∞	40	60	30	90	120
0,1,7,2,5,3	0	20	40	50	∞	40	60	30	90	80
0,1,7,2,5,3,6	0	20	40	50	130	40	60	30	90	80
0,1,7,2,5,3,6,9	0	20	40	50	110	40	60	30	90	80
0,1,7,2,5,3,6,9,8	0	20	40	50	110	40	60	30	90	80
0,1,7,2,5,3,6,9,8,4	0	20	40	50	110	40	60	30	90	80

Nothing updates by using vertex 4.

Step 11: Now Dijkstra's algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.

References:

- Geeksforgeeks.org
- kc97ble.
- Lesson Slides.