

Data Structures

Compiled by - Sagar Udasi

(Version 1.1.0)

*"Give someone a code, and you frustrate them for a day;
Teach someone to code, and you frustrate them for a lifetime."*

- Anonymous

Chapter 1 – Array Rotations

1.1 Reversal algorithm for array rotations

For Left Rotations:

Let AB be the two parts of the input array where A = arr[0 ... d-1] and B = arr[d ... n-1]. The idea of the algorithm is:

- Reverse A to get A'B, where A' is reverse of A.
- Reverse B to get A'B', where B' is reverse of B.
- Reverse all to get (A'B')' = BA.

For example, Let the array be arr = [1, 2, 3, 4, 5, 6, 7], d = 2 and n = 7.

A = [1, 2] and B = [3, 4, 5, 6, 7]

- Reversing A, we get A'B = [2, 1, 3, 4, 5, 6, 7]
- Reversing B, we get A'B' = [2, 1, 7, 6, 5, 4, 3]
- Reversing all, we get (A'B')' = [3, 4, 5, 6, 7, 1, 2]

```
void reverseArray (int arr[], int start, int end) {  
    while (start < end) {  
        int temp = arr[start];  
        arr[start] = arr[end];  
        arr[end] = temp;  
        start++; end--;  
    }  
}  
  
void leftRotate(int arr[], int n, int d) {  
    reverseArray(arr, 0, d-1);  
    reverseArray(arr, d, n-1);  
    reverseArray(arr, 0, n-1);  
}
```

For Right Rotations:

Similarly, the algorithm of right rotations is:

- Reversing all, we get $(AB)' = B'A' = \{7, 6, 5, 4, 3, 2, 1\}$.
Now $B' = \{7, 6\}$ and $A' = \{5, 4, 3, 2, 1\}$
- Reversing B , we get $BA' = \{6, 7, 5, 4, 3, 2, 1\}$
- Reversing A , we get $BA = \{6, 7, 1, 2, 3, 4, 5\}$

```
void rightRotate(int arr[], int n, int d) {  
    reverseArray(arr, 0, n-1);  
    reverseArray(arr, 0, d-1);  
    reverseArray(arr, d, n-1);  
}
```

The time complexity of reversal algorithm is $O(n)$.

Note that: If we have to rotate our array by 1, there is no need to write the reversal algorithm for that. We could use the basic approach which shifts each element by 1.

```
void leftRotateByOne (int arr[], int n) {  
    int temp = arr[0], i;  
    for (i = 0; i < n - 1; i++)  
        arr[i] = arr[i + 1];  
    arr[i] = temp;  
}  
  
void rightRotateByOne (int arr[], int n) {  
    int temp = arr[n - 1], i;  
    for (i = n - 1; i > 0; i--)  
        arr[i] = arr[i - 1];  
    arr[0] = temp;  
}
```

Problem

Q. 1 Given an array, split it from a specified position and move the first part of the array to the end. So, if the input is – arr = $\{12, 10, 5, 6, 52, 36\}$ and $k = 2$; then the output should be – arr = $\{5, 6, 52, 36, 12, 10\}$.

Solution: This problem can be solved in $O(n)$ time complexity using Reversal algorithm.

```
void splitArray(int arr[], int n, int k) {  
    reverseArray(arr, 0, n - 1);  
    reverseArray(arr, 0, n - k - 1);  
    reverseArray(arr, n - k, n - 1);  
}
```

1.2 Search in sorted and rotated array

An element in a sorted array can be found in $O(\log n)$ time via binary search. But suppose we rotate a sorted array and number of rotations is unknown to us beforehand. For instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in $O(\log n)$ time.

The idea which we use here is – when we divide a sorted and rotated array into two halves, at least one of the two halves will always be sorted. We can easily know which half is sorted by comparing start and end elements of each half. Once we found which half is sorted, we can see if ‘key’ is present in that half. If yes, the problem reduces to simple binary search. If no, we divide the other half and repeat above procedure.

Time complexity: $O(\log n)$

```
int searchRotated (int arr[], int low, int high, int key) {  
    int mid = low + (high-low)/2;  
    // Key is not present  
    if (low > high)  
        return -1;  
    // Key found  
    if (arr[mid] == key)  
        return mid;  
    // If left half is sorted  
    if (arr[low] <= arr[mid]) {  
        // If key is present in left half  
        if (arr[low] <= key && arr[mid] >= key)  
            return searchRotated(arr, low, mid-1, key);  
        // Else search right half  
        else  
            return searchRotated(arr, mid+1, high, key);  
    }  
    // If right half is sorted  
    else {  
        // If key is present in right half  
        if (arr[mid] <= key && arr[high] >= key)  
            return searchRotated(arr, mid+1, high, key);  
        // Else search left half  
        else  
            return searchRotated(arr, low, mid-1, key);  
    }  
}
```

Another similar problem which could be solved using binary search in $O(\log n)$ is – **finding the pivot element**. Pivot is the maximum element in sorted and rotated array. Once we find the index of the pivot element, we can easily find the number of rotations performed on an array.

If the index of pivot element is i and size of array is N , then –

$$\text{No. of rotations performed} = \begin{cases} N - i - 1, & \text{if left rotations are performed} \\ i + 1, & \text{if right rotations are performed} \end{cases}$$

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

```
int findPivot (int arr[], int low, int high) {
    while (low <= high) {
        int mid = low+(high-low)/2;
        // If pivot is at 'mid'
        if (mid < high && arr[mid] > arr[mid+1])
            return mid;
        // If pivot is at 'mid-1'
        if (mid > low && arr[mid] < arr[mid-1])
            return mid-1;
        if (arr[low] >= arr[mid])
            high = mid-1;
        else
            low = mid+1;
    }
    // If reached here, it means array is not at all rotated
    return -1;
}
```

Finding pivot element becomes necessary when we need to use ‘two pointer method’ in problems like – Given a sorted and rotated array, find if there is a pair with given sum, assuming all elements to be distinct.

We follow the standard two pointer method to solve this problem. The only thing we should be careful of is – now these two pointers may exceed the length of the array and had to be brought back to the start using modulo length. The standard two pointer method (when there are no rotations) is –

- If sum is equal to x, then increment count.
- If sum is less than x, then move left pointer by incrementing it.
- If sum is greater than x, then move right pointer by decrementing it.

```

int pairsInRotated(int arr[], int n, int x) {
    int pivot = findPivot(arr, 0, n-1);
    int r = pivot; // r is index of maximum element.
    int l = (pivot + 1)%n; // l is index of minimum element.
    int count = 0; // Variable to store count of number of pairs

    while (l != r) {
        if (arr[l] + arr[r] == x) {
            count++;
            cout << arr[l] << " " << arr[r] << endl;

            // Condition to check that left and right pointers have met
            // At this point, loop should terminate
            if(l == (r - 1 + n) % n)
                return count;

            // Move left and right pointers for next finding pair
            l = (l + 1) % n;
            r = (r - 1 + n) % n;
        }

        // If current pair sum is less, move to the higher sum side.
        else if (arr[l] + arr[r] < x)
            l = (l + 1) % n;

        // If current pair sum is greater, move to the lower sum side.
        else
            r = (n + r - 1)%n;
    }
    return count;
}

```

1.3 Quickly print the element at a particular index after K rotations

If the size of an array is 5 and we make 6 rotations, the result would be same as after making 1 rotation. Here 1 is the effective number of rotations, which is less than the size of the array. Let k = effective number of rotations = $K \% \text{size}$.

Consider array = {1, 2, 3, 4, 5, 6, 7, 8} and k = 3. Then the rotated array is – {4, 5, 6, 7, 8, 1, 2, 3}.

We can see that–

This element	became this element after k rotations
0 th	$5^{\text{th}} = (8 - 3 + 0) \% 8$
1 st	$6^{\text{th}} = (8 - 3 + 1) \% 8$
3 rd	$0^{\text{th}} = (8 - 3 + 3) \% 8$
i^{th}	$(n - k + i) \% n$

Using above result, we can directly find the element at a particular index after K rotations. We can also print the rotated array, without actually rotating the original array.

```
void findElement (int arr[], int n, int N, int i) {
    int k = N%n;      // Effective number of rotations
    cout << arr[(n-k+i)%n] << " ";
}

void printRotatedArray (int arr[], int n, int N){
    for (int i = 0; i < n; i++)
        findElement(arr, n, N, i);
    cout << endl;
}
```

Chapter 2 – Sorting Arrays

2.1 $O(n^2)$ algorithms

These are short and simple sorting algorithms containing two nested loops. There are 3 famous $O(n^2)$ algorithms – Selection sort, Bubble sort and Insertion sort. Let's look at them one by one.

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (if sorting has to be done in ascending order) from unsorted part and putting it at the beginning.

```
void selectionSort (int arr[], int n) {  
    int i, j, min_index;  
    for (i = 0; i < n-1; i++) {  
        min_index = i;  
        for (j = i+1; j < n; j++)  
            if (arr[j] < arr[min_index])  
                min_index = j;  
        swap (arr[min_index], arr[i]);  
    }  
}
```

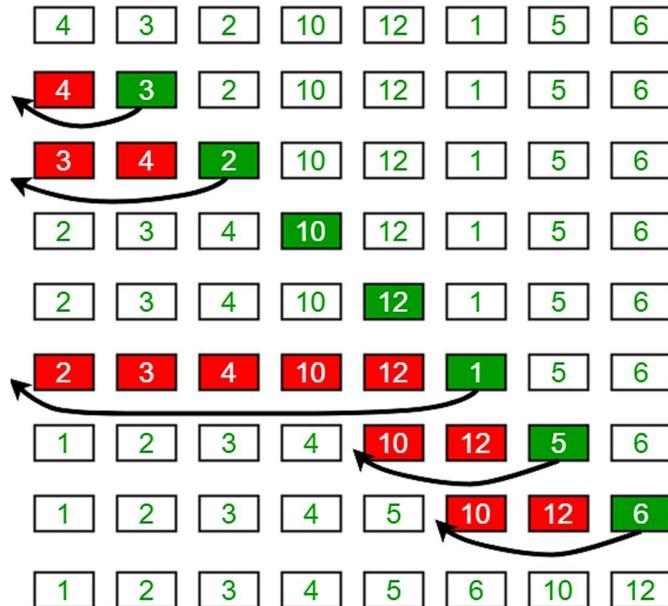
Bubble sort:

Bubble sort works by repeatedly swapping the adjacent elements if they are in wrong order. In this algorithm, we end up finding the biggest element in each iteration and swap it with the last element of unsorted part. So, after 1st iteration, we move the largest element to the end. After 2nd iteration, we move second largest element to the second last position and so on. We can implement bubble sort iteratively and recursively as well. Recursive bubble sort has no performance advantage, but can be a good question to test one's understanding of bubble sort and recursion.

```
void bubbleSortItr (int arr[], int n) {  
    for (int i = 0; i < n-1; i++)  
        for (int j = 0; j < n-i-1; j++)  
            if (arr[j] > arr[j+1])  
                swap(&arr[j], &arr[j+1]);  
}  
  
void bubbleSortRec (int arr[], int n) {  
    if (n==1) return; //Base Case  
    for (int i = 0; i < n-1; i++)  
        if (arr[i] > arr[i+1])  
            //Passing the largest element to end  
            swap(arr[i], arr[i+1]);  
    bubbleSort2 (arr, n-1);  
}
```

Insertion sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.



```

void insertionSortItr (int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i-1;
        // Move elements of arr[0...i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        // Can't replace key by arr[i] or arr[j+1] = arr[i]
        arr[j+1] = key;
    }
}

void insertionSortRec (int arr[], int n) {
    if (n <= 1) return;
    insertionSort2 (arr, n-1);
    int last = arr[n-1];
    int j = n-2;
    while (j >= 0 && arr[j] > last) {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}

```

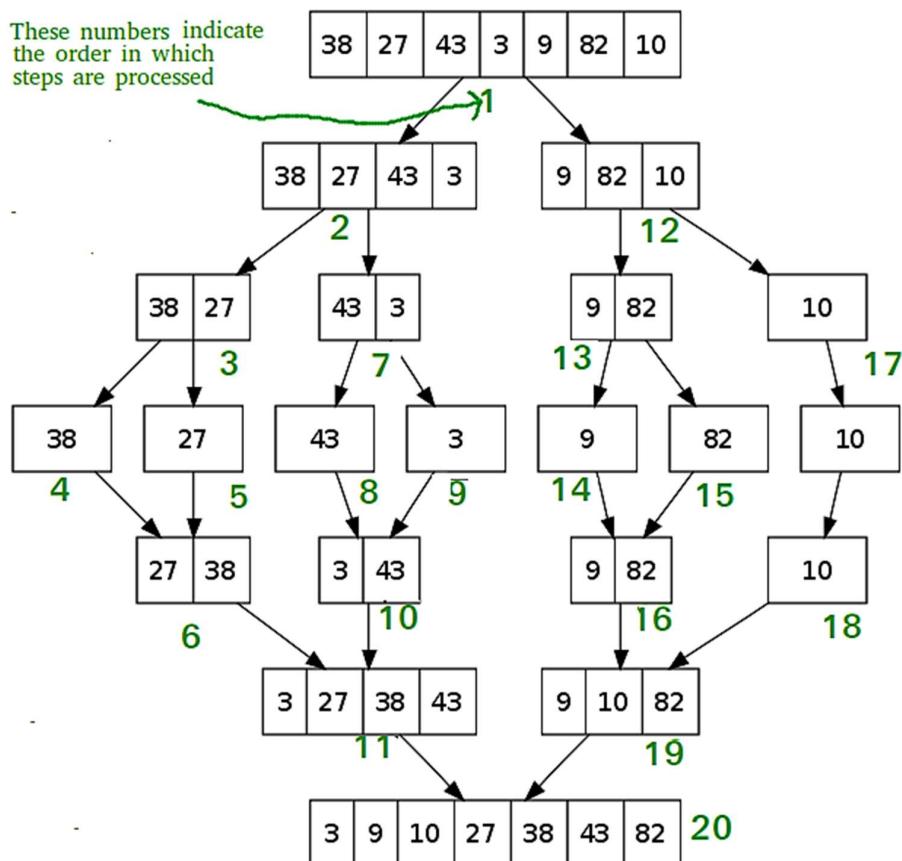
2.2 $O(n * \log n)$ algorithms

It is possible to sort an array efficiently in $O(n * \log n)$ time using sophisticated algorithms that are not limited to swapping elements. There are 3 famous algorithms which belong to this category – Mergesort, Quicksort and Heapsort. Out of these three, we will look into Mergesort and Quicksort now. We will do Heapsort after we have studied – Heap data structure.

Mergesort:

Mergesort is the famous algorithm which implements ‘Divide and Conquer’ technique. In Merge Sort, the given unsorted array with n elements is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

Below is the pictorial representation of Mergesort.



```

// Merge function to merge two sorted arrays into one
void merge (int arr[], int low, int mid, int high) {
    int n1 = mid - low + 1, n2 = high - mid;
    //Temporary arrays
    int* left = new int[n1]; int* right = new int[n2];
    int i, j;
    //Copying data to Temp array
    for (i = 0; i < n1; i++)
        left[i] = arr[low + i];
    for (j = 0; j < n2; j++)
        right[j] = arr[mid + 1+ j];
    //Now merge the temporary arrays back
    i = 0;           //Initial index of first subarray
    j = 0;           //Initial index of second subarray
    int k = low;     //Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (left[i] <= right[j])
            arr[k++] = left[i++];
        else
            arr[k++] = right[j++];
    }
    //Copying remaining elements of left[] if any
    while (i < n1)
        arr[k++] = left[i++];
    //Copying remaining elements of right[] if any
    while (j < n2)
        arr[k++] = right[j++];
}

void mergeSortRec (int arr[], int low, int high) {
    if (low < high) {
        int mid = low +(high-low)/2;
        mergeSort1 (arr, low, mid);
        mergeSort1 (arr, mid+1, high);
        merge (arr, low, mid, high);
    }
}

void mergeSortItr (int arr[], int n) {
    int current_size, left_start;
    for (current_size = 1; current_size < n-1; current_size = 2*current_size) {
        for (left_start = 0; left_start < n-1; left_start += 2*current_size) {
            int mid = left_start + current_size - 1;
            int right_end = min(left_start + 2*current_size - 1, n-1);
            merge (arr, left_start, mid, right_end);
        }
    }
}

```

One thing you may notice that – Iterative mergesort requires only two arguments – array and size of array, whereas recursive mergesort takes three arguments – array, lower index and higher index. If we wish to write recursive mergesort with only two arguments, here is how we do it:

```

void merge (int *arr, int *L, int leftCount, int *R, int rightCount) {
    int i = 0, j = 0, k = 0;
    // i - to mark the index of left subarray (L)
    // j - to mark the index of right subarray (R)
    // k - to mark the index of merged subarray (arr)
    while(i < leftCount && j < rightCount) {
        if(L[i] < R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while(i < leftCount) arr[k++] = L[i++];
    while(j < rightCount) arr[k++] = R[j++];
}

void mergeSort(int input[], int size) {
    int mid, i;
    if (size < 2) return;
    mid = size/2;
    int* L = new int[mid];
    int* R = new int[size-mid];
    for (i = 0; i < mid; i++) L[i] = input[i];
    for (i = mid; i < size; i++) R[i-mid] = input[i];
    mergeSort(L, mid);
    mergeSort(R, size-mid);
    merge(input, L, mid, R, size-mid);
    delete L, R;
}

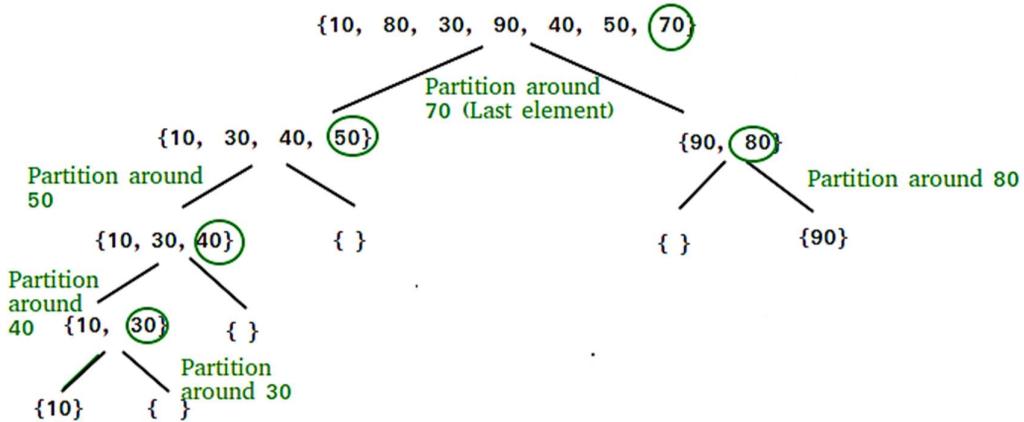
```

Quicksort Algorithm:

Like Mergesort, Quicksort is a ‘Divide and Conquer’ algorithm. It picks an element as pivot and partitions the given array around the pivot. There are many different versions of quickSort that pick the pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Partition process:

We start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```

// Following partition() function takes last element as pivot,
// and places the pivot element at its correct position.
int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high-1; j++) {
        if (arr[j] <= pivot)
            swap (&arr[i++], &arr[j]);

        swap (arr[i+1], arr[high]);
        return (i+1);
    }

    // Following function implements quickSort recursively
    void quickSortRec (int arr[], int low, int high) {
        if (low < high) {
            int partition_index = partition (arr, low, high);
            quickSortRec (arr, low, partition_index - 1);
            quickSortRec (arr, partition_index + 1, high);
        }
    }
}
  
```

```

// Following function implements quickSort iteratively
void quickSortItr (int arr[], int low, int high) {
    // Create an auxiliary stack
    int stack [high - low + 1];
    //Initialize top of the stack
    int top = -1;
    //Push initial values of low and high to stack
    stack[++top] = low;
    stack[++top] = high;
    //Keep popping from stack while it is not empty
    while (top >= 0) {
        high = stack[top--];
        low = stack[top--];
        //Sets pivot at correct position
        int partition_index = partition (arr, low, high);
        //If elements are on the left side of pivot,
        //then push left side to stack
        if (partition_index - 1 > low ) {
            stack[++top] = low;
            stack[++top] = partition_index - 1;
        }
        //If elements are on the right side of pivot,
        //then push right side to stack
        if (partition_index + 1 < high ) {
            stack[++top] = partition_index + 1;
            stack[++top] = high;
        }
    }
}

```

Here are few differences between mergesort and quicksort:

- The worst-case time complexity of quicksort is $O(n^2)$ whereas that of mergesort is $O(n * \log(n))$.
- Mergesort requires additional memory whereas quicksort is in-place sorting algorithm.
- Quicksort is preferred when the elements are stored in main/internal memory and mergesort is preferred when the elements are stored in external memory. Due to this reason, mergesort is preferred in sorting when elements are in order of billions.
- Mergesort is stable whereas quicksort is unstable. i.e. quicksort doesn't guarantee the order of appearance of elements in the sorted array.
- Quicksort is preferred for arrays whereas mergesort is preferred for linked lists.

2.3 $O(n)$ algorithms

It is not possible to sort elements in time less than $O(n * \log n)$ if the algorithm is based on comparisons of array elements. However, there are certain algorithms which use memory or any other information of the elements, which make them faster than comparison-based algorithms. Two famous algorithms of this section are – Counting sort and Bucket sort.

Counting sort:

This sorting technique works when elements are between a specific range. Let's say the range is from 0 to 9, and input is $A = \{1, 4, 1, 2, 7, 5, 2\}$.

- Store the count of occurrences of particular number in the array from 0 to range.

Index	0	1	2	3	4	5	6	7	8	9
Count	0	2	2	0	1	1	0	1	0	0

- Find the cumulative sum array of count[] by modifying it.

Index	0	1	2	3	4	5	6	7	8	9
Count	0	2	4	4	5	6	6	7	7	7

- Now we have two arrays – input array and count array. The first element of the input array is 1. We will now go to index 1 of the count array. The value is 2. This means, in sorted array, the position of element 1 is 2. Now we reduce the count by one, so count[1] = 1. Now we see 4. This means the position of 4 in sorted array is 5. In this way, after traversing whole array, we will get our sorted array.

Sorted	1	1	2	2	4	5	7
Index	1	2	3	4	5	6	7

```
#define RANGE 100

void countSort (int arr[], int n) {
    int count[RANGE] = {0};
    int i, output[n];
    for(i=0; i<n; i++)
        ++count[arr[i]];
    for(i=1; i<RANGE; i++)
        count[i] += count[i-1];
    for(i=n-1; i>=0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        --count[arr[i]];
    }
    for(i=0; i<n; i++)
        arr[i] = output[i];
}
```

The problem with the above counting sort implementation is that we could not sort the array if there are negative elements present in it, because there are no negative indices. So, what we do is – we find the minimum element and we will store count of that minimum element at zeroth index. So, the whole index array is now shifted by value of minimum element.

```
void countSort(vector <int>& arr) {
    int max = *max_element(arr.begin(), arr.end());
    int min = *min_element(arr.begin(), arr.end());
    int range = max - min + 1;
    vector<int> count(range), output(arr.size());

    for(int i = 0; i < arr.size(); i++)
        count[arr[i]-min]++;
    for(int i = 1; i < count.size(); i++)
        count[i] += count[i-1];
    for(int i = arr.size()-1; i >= 0; i--) {
        output[ count[arr[i]-min] - 1 ] = arr[i];
        count[arr[i]-min]--;
    }
    for(int i=0; i < arr.size(); i++)
        arr[i] = output[i];
}
```

Bucket sort:

Bucket sort is mainly useful when the input is uniformly distributed over a range. For example, sort {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434}. Here the input is uniformly distributed in the range 0.0 to 1.0 and also, we cannot use counting sort because there are no floating-point indices. So here is the bucketsort algorithm:

```
bucketSort(arr[ ], n)
1) Create n empty buckets (or lists).
2) Do following for every array element arr[i].
   • Insert arr[i] into bucket[n*arr[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```

```
void bucketSort(float arr[], int n) {
    // 1) Create n empty buckets
    vector<float> b[n];
    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++) {
        int bi = n*arr[i]; // Index in bucket
        b[bi].push_back(arr[i]);
    }
```

```

    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}

```

Now again, we need to modify our bucketsort logic, so that it could sort negative elements as well. The idea is – to store negative numbers separately. Make them positive by multiplying them by -1. Now bucketsort these elements and then convert them back to negative and finally concatenate.

```

void bucketSort(vector<float> &arr, int n) {
    vector<float> b[n];
    for (int i=0; i<n; i++) {
        int bi = n*arr[i];
        b[bi].push_back(arr[i]);
    }
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    int index = 0; arr.clear();
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr.push_back(b[i][j]);
}

void sortMixed(float arr[], int n) {
    vector<float> Neg, Pos;
    // segregate elements in respective arrays
    for (int i=0; i<n; i++) {
        if (arr[i] < 0)
            Neg.push_back (-1 * arr[i]) ;
        else
            Pos.push_back (arr[i]) ;
    }
    bucketSort(Neg, (int)Neg.size());
    bucketSort(Pos, (int)Pos.size());
    // First store elements of Neg[] array by converting into -ve
    for (int i = 0; i < Neg.size(); i++)
        arr[i] = -1 * Neg[Neg.size() - 1 - i];
    // store +ve elements
    for (int j = Neg.size(); j < n; j++)
        arr[j] = Pos[j - Neg.size()];
}

```

Problem

Q. 1 Assume that you are given ‘n’ pairs of items as input, where the first item is a number and second item is one of the colors – red, blue or yellow. Further, assume that the items are sorted by number. Give an $O(n)$ algorithm to sort the items by color (all reds before all blues before all yellows) such that the numbers of identical color stay sorted. For example,

{(1, blue), (3, red), (4, blue), (6, yellow), (9, red)} should become

{(3, red), (9, red), (1, blue), (4, blue), (6, yellow)}.

Solution: Create 3 buckets, one for each color. For every pair P, append number to the bucket of that color. Output the data from red, blue and then yellow buckets.

```
struct Item {
    int number;
    string color;
};

void bucketSort (Item* arr, int n) {
    vector <Item> b[3];
    for (int i = 0; i < n; i++) {
        if (arr[i].color == "red")
            b[0].push_back(arr[i]);
        else if (arr[i].color == "blue")
            b[1].push_back(arr[i]);
        else
            b[2].push_back(arr[i]);
    }
    int index = 0;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}

int main() {
    Item arr[] = {{1, "blue"}, {3, "red"}, {4, "blue"}, {6, "yellow"}, {9, "red"}};
    int size = sizeof(arr)/sizeof(arr[0]);
    bucketSort(arr, size);
    for (int i = 0; i < size; i++) {
        cout << arr[i].number << " " << arr[i].color << endl;
    }
    return 0;
}
```

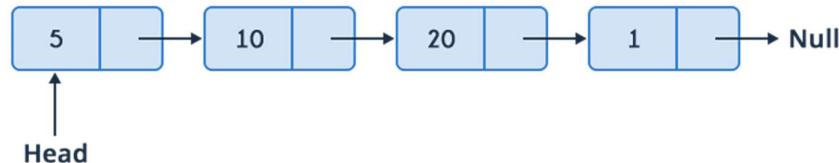
Chapter 3 – Singly Linked List

3.1 Introduction

A **linked list** is a way to store a collection of elements. Like an array these can be characters or integers. Each element in a linked list is stored in the form of a **node**. A node is a collection of two sub-elements or parts - a **data** part that stores the element and a **next** part that stores the link to the next node.



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.



In C++, we represent Node as:

```
class Node {  
public:  
    int data;  
    Node* next;  
};
```

OR

```
struct Node {  
    int data;  
    Node* next;  
};
```

We can use any of the above representation of Node. Let's see how can we create a linked list using above Node.

```
int main() {  
    Node* head = NULL;  
    Node* second = NULL;  
    Node* third = NULL;
```

```

// Allocate 3 nodes in the memory heap
head = new Node();
second = new Node();
third = new Node();
head->data = 1;           // assign data to first node
head->next = second;     // Link first node with second node
second->data = 2;         // assign data to second node
second->next = third;    // Link second node with third node
third->data = 3;          // assign data to third node
third->next = NULL;       // We ground/null the last node
return 0;
}

```

Now let's traverse the list which we created and print it.

```

void printList (Node* n) {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
    cout << endl;
}

```

To use this function, call it inside main() as:

```
printList(head);
```

Note that, head is of type `Node*` and the `printList()` receives an argument of type `Node*`. So, this is pass by value. If any function receives an argument of type `Node**`, then it is pass by reference. We use pass by reference when we write a function which is intended to modify the original linked list. Here `printList` just prints the list. Hence, we have used pass by value here.

3.2 Inserting an element in linked list

A node in a linked list can be added in three ways:

- a) At the front of the list (this added node will become the new head)
- b) After a given node (somewhere in between)
- c) At the end of the list (the pointer of this node will point to `NULL` then)

We will write three functions which will insert nodes according to above three cases. Since here function intends to modify the original linked list, we use pass by reference.

```

void insertBeginning (Node** head_ref, int new_data) {
    // 1. Create a new node to be added in the beginning of the list
    Node* new_node = new Node();
    new_node->data = new_data;
    // 2. Make next of this new_node as head
    new_node->next = *head_ref;
    // 3. Move the head to point to this new_node
    *head_ref = new_node;
}

void insertAfter (Node** prev_node, int new_data) {
    // 1. Check if previous node is not Null
    if (*prev_node == NULL) {
        cout << "The given previous node cannot be Null.\n";
        return;
    }
    // 2. Create a new node to be added
    Node* new_node = new Node();
    new_node->data = new_data;
    // 3. Make next of this new node as the next of previous node
    new_node->next = (*prev_node)->next;
    // 4. Move the next of prev node to this new node
    (*prev_node)->next = new_node;
}

void insertEnd (Node** head_ref, int new_data) {
    // 1. Create new node to be added
    Node* new_node = new Node();
    new_node->data = new_data;
    Node* last = *head_ref;      // This is used in step 4
    // 2. This new node is now last node so make it's next as Null
    new_node->next = NULL;
    // 3. If the linked list is empty, make this new node as head.
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    // 4. Else travel to the last node and change the next of the
    // last node
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
    return;
}

```

```

int main() {
    // Start with empty list
    Node* head = NULL;
    // Now the list becomes 6 -> NULL
    insertEnd(&head, 6);
    // Now the list becomes 7 -> 6 -> NULL
    insertBeginning(&head, 7);
    // Now the list becomes 7 -> 6 -> 8 -> NULL
    insertAfter(&(head->next), 8);
    //Now the list becomes 7 -> 6 -> 8 -> 9 -> NULL
    insertAfter(&(head->next->next), 9);
    //Now the list becomes 7 -> 6 -> 8 -> 9 -> 5 -> NULL
    insertEnd (&head, 5);
    printList(head);
    return 0;
}

```

Instead of writing such functions which take the pointer to the element in the list where the node is to be inserted, we may write a function which will take the position and insert a node at that position. This position cannot exceed the length of the linked list at any time. In following code, getNode() function creates and returns a node which has the data passed as argument and it's next points to NULL.

```

void insertAtPos (Node** current, int pos, int data) {
    // Following condition checks if the position is valid
    if (pos < 1 || pos > (sizes + 1))
        cout << "Invalid Position\n";
    else {
        //Keep on looping till position is zero
        while (pos--) {
            if (pos == 0) {
                // Create a node with given data
                Node* temp = getNode(data);
                // Make new node to point to old node at the
                // same position
                temp->next = *current;
                // Change the pointer of the node previous to
                // old node to point to new Node
                *current = temp;
            }
            else
                // Just move the current's next forward
                current = &(*current)->next;
        }
        sizes++;
    }
}

```

This is the implementation of getNode() function. Soon we will replace this method by defining the constructor in the Node class, which will initialize the node at the time of its creation. Then we will simply replace all the getNode(data) calls, by constructor calls - `new Node(data);`

```
Node* getNode (int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

One special case comes, when you want to add a node at the middle of the linked list. Now you can do this in two ways. First one is – find the number of nodes by traversing the entire linked list once. This step gives you the length of the linked list. This is almost similar to printing the list. But instead of printing the list, you are just incrementing some counter.

```
int iterativeCount (Node* head) {
    int count = 0;
    Node* current = head;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

int recursiveCount (Node* head) {
    if (head == NULL)
        return 0;
    else
        return 1 + recursiveCount(head->next);
}
```

Once you got the length, you calculate the position as

$$position = \begin{cases} \frac{\text{length}}{2}, & \text{in case length is even} \\ \frac{\text{length} + 1}{2}, & \text{in case length is odd} \end{cases}$$

Traverse the linked list again, reach at position and add a node there using above insertAtPos() method. This method requires traversing the linked list twice.

However, you can add node in the middle of the linked list by single traversal of the linked list as well, by maintaining a ‘slow’ and a ‘fast’ pointer. This is called ‘Hare and Tortoise’ algorithm. Slow pointer jumps by 1 and fast pointer jumps by 2 nodes. When fast reaches the end of the linked list, slow must be at the middle of the list. Insert your node after the element pointed by the slow pointer.

```
void insertAtMid (Node** head_ref, int x) {
    // If list is empty, make the new node as head
    if (*head_ref == NULL)
        *head_ref = getNode(x);
    else {
        Node* newNode = getNode(x);
        Node* slow = *head_ref;
        Node* fast = (*head_ref)->next;
        while (fast && fast->next) {
            //move slow pointer to next node
            slow = slow->next;
            //move fast pointer two nodes at a time
            fast = fast->next->next;
        }
        newNode->next = slow->next;
        slow->next = newNode;
    }
}
```

3.3 Deleting a node

To delete any node in the linked list, we need to be at the element which is previous to the node to be deleted. We cannot delete a node if we are standing on that node. The reason is – the next of the previous node was pointing to the current node. If we delete the current node, we won’t be able to traverse backwards in order to link the next of the previous node with the next of the current node. You can do so by maintaining a pointer at the previous node at every instant, but it is better to follow the convention – If you want to delete 5th node, you will traverse upto 4th node and then perform operations to delete the next node.

```
void deleteNode (Node** head_ref, int pos) {
    // If the linked list is empty
    if (*head_ref == NULL) {
        cout << "Empty list\n";
        return;
    }
```

```

Node* temp1 = *head_ref;
// If the node to be deleted is head node
if (pos == 1) {
// Head is now pointing to second node
*head_ref = temp1->next;
delete temp1;

// We cannot free *head_ref because it is the
// identity of linked list.
// Once we free head, we have lost the way to access
// other elements of linked list.
}
else {
// After below statement, temp1 now points to (pos-1)th node
for (int i = 0; (temp1 != NULL) && (i < pos-2); i++)
// temp1 will be NULL if pos is greater than size of linked list
temp1 = temp1->next;

if (temp1 == NULL) {
cout << "Invalid Position\n";
}
else {
// temp2 points to (pos)th node
Node* temp2 = temp1->next;

// Now temp1's next (next of pos-1 node)
// should point to temp2's next node (pos+1 node)
temp1->next = temp2->next;

// Memory is to be freed which is occupied by
// temp2 (pos)th element
delete temp2;
}
}
}
}

```

One follow-up question could be – Instead of position, if we are given the value of the node which is to be deleted, what will you do now? The idea is very simple, just keep on checking if the next node's data matches the value.

```

void deleteNode (Node** head_ref, int key) {
if (*head_ref == NULL) {
cout << "Empty List.\n";
return;
}
Node* temp1 = *head_ref;

```

```

    if (temp1->data == key) {
        *head_ref = temp1->next;
        delete temp1;
    }
    else {
        while (temp1->next != NULL && temp1->next->data != key)
            temp1 = temp1->next;

        if (temp1 == NULL) {
            cout << "Key not found.\n";
        }
        else {
            Node* temp2 = temp1->next;
            temp1->next = temp2->next;
            delete temp2;
        }
    }
}

```

One final question of this section – How will you delete the entire linked list?
Following is the simple implementation for this case.

```

void deleteList (Node** head_ref) {
    Node* current = *head_ref;
    Node* next;
    while (current != NULL) {
        next = current->next;
        delete current;
        current = next;
    }
    *head_ref = NULL;
}

```

3.4 Traversing the linked list

Suppose you want to find/search an element in the linked list, how would you do that? Simply traverse the list and check if the value of the node is equal to search key. This takes $O(n)$ time. Can we do better? Something like binary search? No! Because in linked list, you can't jump to middle directly. Even for reaching at the middle position, you have to traverse it. So linear search is the best search possible in linked lists.

Following is the iterative and recursive implementation of linear search for linked lists:

```

int iterativeSearch (Node* head, int x) {
    Node* current = head; int count = 1;
    while (current != NULL)
        if (current->data == x)
            return count;
        else {
            count++;
            current = current->next;
        }
    return -1;
}

int recursiveSearch (Node* head, int x) {
    if (head == NULL)
        return -1;
    if (head->data == x)
        return 1;
    int indexFromChildList = recursiveSearch(head->next, x);
    if (indexFromChildList != -1)
        indexFromChildList++;
    return indexFromChildList;
}

```

Let's move on to the next question. Write a function `getNthFromLast()`, that returns the value of the node which is 'n' nodes away from the end.

The simple logic would be – first calculate the length of the array and then traverse $(length-n+1)^{th}$ nodes. You are now at n^{th} node from last. This method requires two traversals of the linked list. You can do this in single traversal by maintaining two pointers – reference pointer and main pointer. Both the pointers start from head. Reference pointer moves n nodes in the list. After that, both pointers move ahead till reference pointer reaches end of the list. At this moment, the main pointer will be pointing at the n^{th} node from the end.

```

void getNthFromLast (Node* head, int n) {
    Node *main_ptr = head, *ref_ptr = head;
    int count = 0;
    if (head != NULL) {
        while (count < n) {
            if (ref_ptr == NULL) {
                cout << n << " is greater than number of nodes in list.";
                return;
            }
            ref_ptr = ref_ptr->next;
            count++;
        }
    }
}

```

```

        while (ref_ptr != NULL) {
            main_ptr = main_ptr->next;
            ref_ptr = ref_ptr->next;
        }
        cout << "Node number " << n << " from last is "
            << main_ptr->data << endl;
    }
}

```

You can solve this problem recursively as well. Create a global/static counter i and initialize this to zero. Keep calling your function recursively for the childlist. Once head becomes NULL, that means you have reached the end. Now the recursive calls which are stacked will now return in reverse order. During this return step, increment i and check if it is equal to n at any moment. The moment they are equal, print head->data.

```

void getNthFromLast(struct Node* head, int n) {
    static int i = 0;
    if (head == NULL)
        return;
    getNthFromLast(head->next, n);
    if (++i == n)
        printf("%d", head->data);
}

```

One last thing which you should try is – finding the frequency of the element in the given linked list. Following is the simple implementation:

```

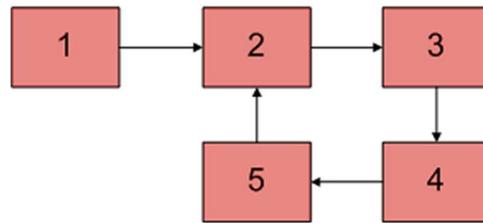
int countIterative (Node* head, int key) {
    Node* current = head; int count = 0;
    while (current != NULL) {
        if (current->data == key)
            count++;
        current = current->next;
    }
    return count;
}

int countRecursive (struct Node* head, int key) {
    if (head == NULL)    return 0;
    int count = countRecursive(head->next, key);
    if (head->data == key)
        count++;
    return count;
}

```

3.5 Loops in linked lists

A loop in a linked list looks like this:



To detect a loop in linked list, we will use **Floyd's Cycle Detection Algorithm**. This is the fastest method of loop detection and uses 'slow' and 'fast' pointers. The idea is - if there is a loop, these pointers will definitely meet at same node at some instance.

Time complexity of this method is $O(n)$.

```
bool detectLoop (Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (slow && fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            return true;
    }
    return false;
}
```

You can use hashing as well. It is also an $O(n)$ algorithm in time complexity, but it also uses $O(n)$ extra space.

```
bool detectLoop (Node* h) {
    unordered_set <Node*> s;
    while (h != NULL) {
        if (s.find(h) != s.end())
            return true;
        s.insert(h);
        h=h->next;
    }
    return false;
}
```

Once we have detected the loop, our next task would be to find the length of the loop. The idea is – when there is a loop, slow and fast pointers meet at a common node. We store the address of this node in third pointer ‘ptr’ and initialize count as 1. We then start moving ptr and keep increasing the count, till we see ptr again as the next node. We then return the count. Note that the following countNodes() utility function is different from the previous iterativeCount and recursiveCount methods, as it is intended to be called only when there is a loop.

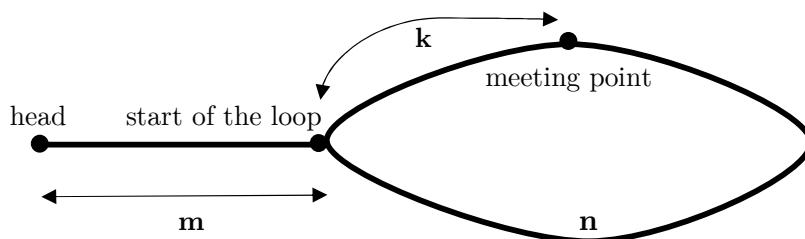
```

int countNodes (Node* n) {
    int count = 1; Node* temp = n;
    while (temp->next != n) {
        count++;
        temp = temp->next;
    }
    return count;
}

int countNodesInLoop (Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (slow && fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            return countNodes(slow);
    }
    return 0;
}

```

Now that you know how to find length of the loop, you may wish to remove it as well. Now removing a loop from a linked list is a bit tricky task because for that you will need the starting point of the loop. Below diagram can be considered as a linked list with the loop.



Here, n = length of the loop, m = distance of first node of cycle from head, and k = distance of point where slow and fast meet from the start. Here is our analysis:

*distance travelled by fast = 2 * distance travelled by slow*

$$\therefore (m + n * x + k) = 2 * (m + n * y + k)$$

where x = number of complete cyclic rounds made by fast pointer before they meet first time; and y = number of complete cyclic rounds made by slow pointer before they meet first time.

$$\therefore m + k = (x - 2y) * n$$

which means $(m+k)$ is a multiple of n .

So, if we start moving both the pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of loop (has made m steps), fast pointer would have made also moved m steps as they are now moving same pace. Since $m+k$ is a multiple of n and fast starts from k , they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

```
void detectAndRemoveLoop (Node* head) {
    // If the list is empty or has one node, it has no loop
    if (head == NULL || head->next == NULL)
        return;
    Node* slow = head;
    Node* fast = head;
    slow = slow->next;
    fast = fast->next->next;
    while (fast && fast->next) {
        if (slow==fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }
    // If loop exists
    if (slow == fast) {
        slow = head;
        while (slow->next != fast->next) {
            slow = slow->next;
            fast = fast->next;
        }
        // Now slow and fast are pointing towards the beginning of loop.
        // This means fast must be the ending node of the loop.
        fast->next = NULL;
    }
}
```

You can use hashing as well. We can hash the addresses of the linked list nodes in an unordered set and just check if the element already exists in the map. If it exists, we have reached a node which already exists by a cycle, hence we need to make the last node's next pointer NULL.

```
void hashAndRemove(Node *head) {
    unordered_set<Node*> s;
    // pointer to last node
    Node* last = NULL;
    while(head!=NULL) {
        // if node not present in the set, insert it in the set
        if(s.find(head) == s.end()) {
            s.insert(head);
            last = head;
            head = head->next;
        }
        // if present, it is a cycle, make the last node's
        // next pointer NULL
        else {
            last->next = NULL;
            break;
        }
    }
}
```

3.6 Sorting linked lists

Here we will study insertion sort, $O(n^2)$ algorithm, and mergesort, $O(n * \log n)$ algorithm.

Insertion Sort:

Before studying insertion sort, we must know how to insert a node in sorted linked list, such that the node is inserted at right place and list remains sorted.

```
void sortedInsert (Node** head_ref, Node* new_node) {
    Node* current;
    // When the node to be inserted occupies the head position
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data) {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else {
        // Locating node before the point of insertion
        current = *head_ref;
```

```

        while (current->next != NULL &&
               current->next->data < new_node->data)
            current = current->next;
        // Inserting the node
        new_node->next = current->next;
        current->next = new_node;
    }
}

```

Now for insertion sort, create an empty sorted list. Traverse the input list and add each node of it in empty result list in sortedInsert way.

```

void insertionSort (Node** head_ref) {
    // Initialize sorted linked list
    Node* sorted = NULL;
    // Traverse the given linked list & insert every node in sorted list
    Node* current = *head_ref;
    while (current != NULL) {
        Node* next = current->next;
        sortedInsert(&sorted, current);
        current = next;
    }
    *head_ref = sorted;
}

```

Merge Sort:

Again, before directly implementing mergesort, we must know how to merge two sorted linked lists into one sorted linked list. Here is the algorithm:

- Create two pointers, `result_ptr` which will act as head of result list and `last_ptr` which will be always pointing to the end of result list.
- If first element of `list1` is smaller than first element of `list2`, move this first element from `list1` to result list and similar for reverse case.
- If any list turns `NULL`, append the remaining list to result list.

We will use a `moveNode()` helper function in following implementation, which takes the node from the front of the source list and move it to the front of the destination list. For example,

Before calling `moveNode()`, `source = 1-2-3, dest = 1-2-3`

After calling `moveNode()`, `source = 2-3, dest = 1-1-2-3`.

Since recursive implementation doesn't need this utility function, we prefer using recursive implementation of `mergeSorted()` routine in `mergeSort()` algorithm.

```

void moveNode (Node** source_ref, Node** dest_ref) {
    // New Node is Front node of source list
    Node* new_node = *source_ref;
    // Throw error if the source list is empty.
    assert(new_node != NULL);
    // Advance the source pointer
    *source_ref = new_node->next;
    // Linking the first node of source to first node of destination
    new_node->next = *dest_ref;
    // Making *dest_ref point to new node
    *dest_ref = new_node;
}

Node* mergeSortedItr (Node* a, Node* b) {
    Node* result = NULL;
    // Reference to pointer pointing to last element of the result list.
    Node** last_ref = &result;
    while (1) {
        if (a == NULL) {
            *last_ref = b; break;
        }
        else if (b == NULL) {
            *last_ref = a; break;
        }
        else if (a->data <= b->data) moveNode (&a, last_ref);
        else moveNode (&b, last_ref);
        // Tricky way to advance a pointer to the next field
        last_ref = &((*last_ref)->next);
        printList(A); printList(B); printList(result); cout << endl;
    }
    return result;
}

Node* mergeSortedRec (Node* a, Node* b) {
    Node* result = NULL;
    if (a == NULL) return (b);
    else if (b == NULL) return (a);
    if (a->data < b->data) {
        result = a;
        result->next = mergeSortedRec (a->next, b);
    }
    else {
        result = b;
        result->next = mergeSortedRec (a, b->next);
    }
    return result;
}

```

mergeSort() routine needs a helper/utility function which splits the list into almost equal halves. Let's call this helper function as – frontBackSplit(). If the length of the list is odd, extra node should go to the front list. This helper function uses ‘slow’ and ‘fast’ pointers.

```

void frontBackSplit(Node* source, Node** front_ref, Node** back_ref) {
    Node* slow = source;
    Node* fast = source->next;
    while (fast!=NULL){
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    *front_ref = source;
    *back_ref = slow->next;
    slow->next = NULL;
}

void mergeSort(Node** headRef) {
    Node* head = *headRef;
    Node* a;
    Node* b;
    // Base case for length 0 or 1
    if ((head == NULL) || (head->next == NULL)) {
        return;
    }
    // Split head into 'a' and 'b' sublists
    frontBackSplit(head, &a, &b);
    // Recursively sort the sublists
    mergeSort(&a);
    mergeSort(&b);
    // answer = merge the two sorted lists together
    *headRef = mergeSorted(a, b);
}

```

One another famous merge problem is - Given two linked lists, insert nodes of second list into first list at alternate positions of first list. The nodes of second list should only be inserted when there are positions available.

For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, then first list should become 5->12->7->10->17->2->13->4->11->6 and second be empty. Also, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Here is the solution of above problem. Since head of first list will never change and the head of the second list may change, we need to pass first list by value and other by reference.

```
void mergeAlternate (Node* p, Node** q) {  
    Node *p_curr = p, *q_curr = *q;  
    Node *p_next, *q_next;  
    //While there are available positions in p  
    while (p_curr != NULL && q_curr != NULL) {  
        // Save next pointers  
        p_next = p_curr->next;  
        q_next = q_curr->next;  
        // Make q_curr as next of p_curr  
        q_curr->next = p_next; // Change next pointer of q_curr  
        p_curr->next = q_curr; // Change next pointer of p_curr  
        // Update current pointers for next iteration  
        p_curr = p_next;  
        q_curr = q_next;  
    }  
    *q = q_curr; // Update head pointer of second list  
}
```

Problems

Q. 1 Given pointer to the head node of a linked list, the task is to reverse the linked list.

For example,

Input : 1->2->3->4->NULL

Output : 4->3->2->1->NULL

Solution:

Approach - 1 (Iterative Method - 1)

1. Initialize three pointers, prev as NULL, curr as head and next as NULL.
2. Traverse the linked list and do the following:

```
// Before changing next of current, store next node  
a) next = curr->next  
    // Now change next of current to previous node.  
    // This is the step where reversing happens.  
b) curr->next = prev  
    // Move prev and curr one step forward  
c) prev = curr  
curr = next
```

Approach - 2 (Recursive Method)

1. Divide the list in two parts - first node and the rest of the linked list.
2. Call reverse for the rest of the linked list.
3. Link rest to first.
4. Fix head pointer

Approach - 3 (Iterative Method - 2)

In iterative method we had used 3 pointers prev, cur and next. This approach only uses two pointers.

```
void reverse1 (Node** head_ref) {  
    Node* current = *head_ref;  
    Node* prev = NULL, *next = NULL;  
    while (current != NULL) {  
        next = current->next; // store next  
        current->next = prev; // reverse current node's pointer  
        prev = current; // Move pointers one position ahead  
        current = next;  
    }  
    *head_ref = prev;  
}  
  
void reverse2 (Node** head_ref) {  
    if (*head_ref == NULL) return;  
    Node* first = *head_ref, *rest = first->next;  
    // If list has only one node  
    if (rest == NULL) return;  
    // Recurse on the rest of the string  
    reverse2(&rest);  
    // Link rest to first  
    first->next->next = first;  
    first->next = NULL;  
    // fix the head pointer  
    *head_ref = rest;  
}  
  
void reverse3 (Node** head_ref) {  
    Node* current = *head_ref, *subsequent;  
    while(current->next != NULL) {  
        subsequent = current->next;  
        current->next = subsequent->next;  
        subsequent->next = *head_ref;  
        *head_ref = subsequent;  
    }  
}
```

Q. 2 Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Solution:

```
Node* reverse (Node* head, int k) {  
    Node* current = head;  
    Node* subsequent = NULL;  
    Node* previous = NULL;  
    int count = 0;  
    // Reverse first K nodes of the linked list  
    while (current != NULL && count < k) {  
        subsequent = current->next;  
        current->next = previous;  
        previous = current;  
        current = subsequent;  
        count++;  
    }  
    // 'subsequent' is now pointing to the (k+1)th node  
    // Recursively call the function for the remaining list  
    if (subsequent != NULL)  
        head->next = reverse(subsequent, k);  
    // Previous is the new head of the input list  
    return previous;  
}
```

Q. 3 Given a singly linked list, rotate the linked list counter-clockwise by k nodes.

Solution:

1. To rotate the linked list, we need to

- change next of kth node to NULL
- change next of the last node to the previous head node
- change head to (k+1)th node.

So we need to get hold of three nodes: kth node, (k+1)th node and last node.

2. Traverse the list from the beginning and stop at kth node.

Store pointer to kth node. We can get (k+1)th node using kthNode->next.

Keep traversing till the end and store pointer to last node also.

Finally, change pointers as stated above.

```
void rotate (Node** head_ref, int k) {  
    if (k==0)    return;  
    Node* current = *head_ref;  
    // Advance current by 'k' nodes  
    int count = 1;
```

```

while (count < k && current != NULL) {
    current = current->next;
    count++;
}
// If current is NULL that means 'k' is
// greater than the size of the list
if (current == NULL)    return;
// Current points to kth node now. Store it.
Node* kthNode = current;
// Make current point to the last node now
while (current->next!=NULL)
    current = current->next;
// Change next of the last node to the head
current->next = *head_ref;
// Change head of (k+1)th node
*head_ref = kthNode->next;
// Change next of kth node to NULL
kthNode->next = NULL;
}

```

Q. 4 Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.

Solution:

Approach 1 - Use a stack

1. Push every node from head to tail into stack.
2. Traverse the list again and compare it with every popped element.
3. If all matched, return true, else return false.

Time Complexity: $O(n)$ and Space Complexity: $O(n)$

Approach 2 - Reversing the list

1. Reverse the second half of the list by getting at the middle.
2. Check if first and second half of the lists are identical.
3. Construct the original linked list by reversing the second half again and attach it back to the first half.

Time Complexity: $O(n)$ and Space Complexity: $O(1)$

Approach 3 - Using Recursion

1. Use two pointers - left and right. Left is at start in the beginning and right is at the end.
2. Increment left and decrement right pointers, and call the function recursively on the sublist.

Time Complexity: $O(n)$ and Space Complexity: $O(n)$

Below we have implemented approach 2 and 3.

```

// Utility Functions for Iterative method
void reverse (Node** head_ref) {
    Node* previous = NULL;
    Node* current = *head_ref;
    Node* subsequent = NULL;
    while (current != NULL) {
        subsequent = current->next;
        current->next = previous;
        previous = current;
        current = subsequent;
    }
    *head_ref = previous;
}

bool compareLists (Node* head1, Node* head2) {
    Node* temp1 = head1;
    Node* temp2 = head2;
    while (temp1 && temp2) {
        if (temp1->data == temp2->data) {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else return false;
    }
    // If both temp1 and temp2 point to NULL now,
    // it means the lists are equal.
    if (temp1 == NULL && temp2 == NULL)
        return true;
    // If reached here, means one list is NULL while other is not.
    return false;
}

bool isPalindrome1 (struct Node* head) {
    Node* slow_ptr = head, *fast_ptr = head;
    Node* prev_of_slow_ptr = head;
    Node* second_half = NULL;
    Node* middle_node = NULL; //Will be used to handle odd sized lists
    bool result = true;
    if (head != NULL && head->next != NULL) {
        // Get the middle of node by slow and fast ptr method
        while (fast_ptr != NULL && fast_ptr->next != NULL) {
            fast_ptr = fast_ptr->next->next;
            // We need previous of the slow_ptr for the
            // linked lists with odd elements
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }
    }
}

```

```

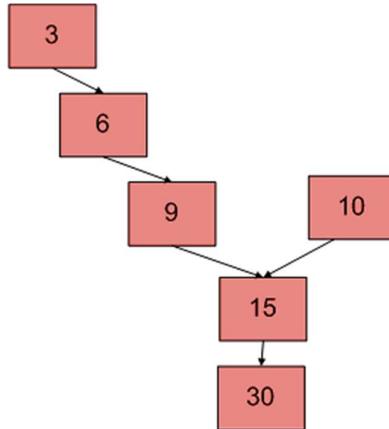
        // fast pointer will become Null if there are even
        // elements in the list. If not Null, we need to
        // skip the middle node and store it somewhere so
        // that we can restore the original list
        if (fast_ptr != NULL) {
            middle_node = slow_ptr;
            slow_ptr = slow_ptr->next;
        }
        // Now reverse the second half of the list
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL;
        reverse(&second_half);
        result = compareLists(head, second_half);
        // Now reverse the second half again
        reverse(&second_half);
        // Connect the second half back to first half
        // If there was a middle node (odd sized)
        if (middle_node!=NULL) {
            prev_of_slow_ptr->next = middle_node;
            middle_node->next = second_half;
        }
        else
            prev_of_slow_ptr->next = second_half;
    }
    return result;
}

bool isPalindromeUtil (Node** left, Node* right) {
    // Base Case
    if (right == NULL)
        return true;
    // Check if sublist is palindrome or not
    bool temp = isPalindromeUtil(left, right->next);
    if (temp == false)
        return false;
    // Check values at current left and right
    bool result = (right->data == (*left)->data);
    // Move left to next node
    *left = (*left)->next;
    return result;
}

bool isPalindrome2(Node* head) {
    return isPalindromeUtil(&head, head);
}

```

Q. 5 There are two singly linked lists in a system. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y shaped list. Write a program to get the point where two linked lists merge.



Solution:

Approach 1 - Using Two Loops

Use 2 nested for loops. The outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of the 2nd list is same as the current node of the first linked list.

Time complexity: $O(M * N)$ where M and N are the numbers of nodes in two lists.

Space complexity: $O(1)$

Approach 2 - Use difference of node counts

1. Get the count of nodes in the first list. Let it be c1.
2. Get the count of nodes in the second list. Let it be c2.
3. Get difference $d = \text{abs}(c1 - c2)$
4. Traverse d nodes in the bigger list.
5. Then we can traverse both the lists in parallel till we come across a common node.
(Note that getting a common node is done by comparing the address of the nodes.)

Time complexity: $O(M+N)$ and Space Complexity: $O(1)$

```

int getCount (Node* head) {
    Node* current = head;
    int count = 0;
    while (current != NULL) {
        current = current->next; count++;
    }
    return count;
}
  
```

```

int getIntersectionNode (Node* head1, Node* head2) {
    // Count the number of nodes in both the linked list
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d = c1>c2 ? (c1-c2) : (c2-c1);
    Node* current1 = head1;
    Node* current2 = head2;
    int count = 0;
    if (c1 > c2) {
        while (count < d) {
            current1 = current1->next; count++;
        }
    } else {
        while (count < d) {
            current2 = current2->next; count++;
        }
    }
    while (current1 != NULL && current2 != NULL) {
        if (current1 == current2)
            return current1->data;
        current1 = current1->next;
        current2 = current2->next;
    }
    return -1;
}

```

Q. 6 Given a linked list of integers, write a function to modify the LL such that all even numbers appear before all the odd numbers in the modified LL. Also, keep the order of even and odd numbers same.

Solution:

The idea is to split the linked list into two: one containing all even nodes and other containing all odd nodes. And finally attach the odd node linked list after the even node linked list. To split the linked list, traverse the original linked list and move all odd nodes to a separate linked list of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of last pointer in the odd node list.

```

void segregateEvenOdd(Node** head_ref) {
    Node *evenStart = NULL, *evenEnd = NULL, *oddStart = NULL;
    Node *oddEnd = NULL; Node* current = *head_ref;
    while (current!=NULL) {
        int value = current->data;
        // If value is even, add node to even list
        // or else, add node to odd list

```

```

        if (value%2 == 0)
            if (evenStart == NULL) {
                evenStart = current; evenEnd = evenStart;
            }
            else {
                evenEnd->next = current; evenEnd = evenEnd->next;
            }
        else
            if (oddStart == NULL) {
                oddStart = current; oddEnd = oddStart;
            }
            else {
                oddEnd->next = current; oddEnd = oddEnd->next;
            }
        // Move head pointer one step forward
        current = current->next;
    }
    // If either list is empty, no change is required
    // as all elements are either even or odd
    if (oddStart == NULL || evenStart == NULL)
        return;
    // Add odd list after even list
    evenEnd->next = oddStart;
    oddEnd->next = NULL;
    *head_ref = evenStart;
}

```

Q. 7 Add two numbers which are represented in the form of linked lists.

Solution:

Possibility 1:

We may represent numbers in reverse order. For example,

5-3-6-NUL^l // This represents number 635

8-4-2-NUL^l // This represents number 248

Output: 3-8-8-NUL^l // This represents sum 883

Solving in above way is bit easier. Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0

Possibility 2:

When we are not allowed to represent numbers in reverse order. For example,

5-6-3-NUL^l // represents number 563

8-4-2-NUL^l // represents number 842

Output: 1-4-0-5-NUL^l // This represents sum 1405

- 1) Calculate sizes of given two linked lists.
- 2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of the rightmost nodes and forward carry to left side.
- 3) If size is not same, then follow below steps:
 - a) Calculate difference of sizes of two linked lists. Let the difference be diff
 - b) Move diff nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
 - c) Calculate sum of the carry (calculated in previous step) with the remaining left sub-list of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

```

Node* addTwoListsReversed (Node* first, Node* second) {
    Node* result = NULL, *temp, *prev = NULL;
    int carry = 0, sum;
    while (first || second) {
        sum = carry + (first? first->data : 0) +
              (second? second->data : 0);
        carry = (sum >= 10)? 1: 0; sum = sum%10;
        temp = newNode(sum);
        if (result == NULL) result = temp;
        else prev->next = temp;
        prev = temp;
        if (first) first = first->next;
        if (second) second = second->next;
    }
    if (carry > 0)
        temp->next = newNode(carry);
    return result;
}

void swapPointer(Node** a, Node** b) {
    Node* t = *a; *a = *b; *b = t;
}

int getSize(Node* node) {
    int size = 0;
    while (node != NULL)
        node = node->next; size++;
    return size;
}

Node* addSameSize(Node* head1, Node* head2, int* carry) {
    // Checking only head1 as both lists are of same size
    if (head1 == NULL)
        return NULL;
    Node* result = newNode;

```

```

        result->next = addSameSize(head1->next, head2->next, carry);
        // Add digits of current nodes and propagated carry
        int sum = head1->data + head2->data + *carry;
        *carry = sum/10;
        sum = sum % 10;
        result->data = sum;
        return result;
    }

/* This function is called when smaller list is added to bigger list's
   sublist of same size. Once the right sublist is added, the carry
   must be added to left side of larger list to get final result.*/
void addCarryToRemaining (Node* head1, Node* current, int* carry,
                         Node** result) {
    int sum;
    // If 'diff' number of nodes are not traversed, add carry
    if (head1 != current) {
        addCarryToRemaining(head1->next, current, carry, result);
        sum = head1->data + *carry;
        *carry = sum/10; sum %= 10;
        // Add this node to the front of the result list
        push(result, sum);
    }
}

Node* addTwoLists(Node* first, Node* second) {
    Node* result = new Node;
    Node* current = new Node;
    // If any one of the lists is empty
    if (first == NULL)
        return second;
    if (second == NULL)
        return first;
    int size1 = getSize(first);
    int size2 = getSize(second);
    int carry = 0;
    // Add lists directly if they are of same size
    if (size1 == size2)
        result = addSameSize(first, second, &carry);
    else {
        int diff = abs(size1-size2);
        // First list should be greater than second list
        // If not, then swap pointers
        if (size1 < size2)
            swapPointer (&first, &second);
        // Move 'diff' number of nodes in first list
        for (current = first; diff--; current = current->next);

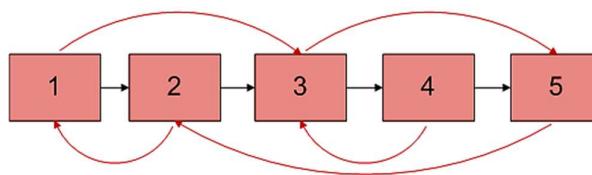
```

```

        // Now add the same sized lists
        result = addSameSize (current, second, &carry);
        // Addition of remaining first list and carry
        addCarryToRemaining (first, current, &carry, &result);
    }
    // If some carry is left, eg. 997 + 87, push 1 in front of
    // result list
    if (carry)
        push(&result, carry);
    return result;
}

```

Q. 8 You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however can point to any node in the list and not just the previous node. Now write a program in **O(n)** time to duplicate this list. That is, write a program which will create a copy of this list.



Solution: For explanation, refer this video:

<https://www.youtube.com/watch?v=xbpUHSKoALg>

```

Node* cloneHashed (Node* head) {
    Node* original_current = head;
    Node* cloned_current = NULL;
    //Map contains node to node mapping of original and cloned linked list
    unordered_map <Node*, Node*> m;
    // First cloning the linked list using next pointer and filling Map
    while (original_current != NULL) {
        cloned_current = new Node (original_current->data);
        m[original_current] = cloned_current;
        original_current = original_current->next;
    }
    // Traverse linked list again to adjust next and random references
    original_current = head;
    while (original_current != NULL) {
        cloned_current = m[original_current];
        cloned_current->next = m[original_current->next];
        cloned_current->random = m[original_current->random];
        original_current = original_current->next;
    }
    return m[head];
}

```

```

// Implementing optimized solution without Hash Map
Node* cloneOptimum (Node* head) {
    Node* current = head, *temp;
    // Inserting additional node after every node of original list
    while (current) {
        temp = current->next;
        current->next = new Node(current->data);
        current->next->next = temp;
        current = temp;
    }
    // Adjust the random pointers of newly added nodes
    current = head;
    while (current) {
        if (current->next)
            current->next->random = current->random ?
                current->random->next : current->random;
        // move to next newly added node by skipping original node
        current = current->next ? current->next->next : current->next;
    }

    // Now we need to separate the original and cloned list
    Node* original = head;
    Node* copy = head->next;
    temp = copy;
    while (original && copy) {
        original->next = original->next ?
            original->next->next : original->next;
        copy->next = copy->next ? copy->next->next : copy->next;
        original = original->next;
        copy = copy->next;
    }
    return temp;
}

```

Q. 9a Given a singly linked list and a number k, write a function to find the (n/k) th element, where n is the number of elements in the list. We need to consider ceil value in case of decimals.

Examples:

Input: list = 1->2->3->4->5->6 and k = 2

Output: 3

Since n = 6 and k = 2, we print $(6/2)$ -th node which is 3.

Input: list = 2->7->9->3->5 and k = 3

Output: 7

Since n is 5 and k is 3, we print $\text{ceil}(5/3)$ th node which is 2nd node, i.e., 7.

Solution:

1. Take two pointers temp and fractionalNode and initialize them with null and head respectively.
2. For every k jumps of the temp pointer, make one jump of the fractionalNode pointer.

```
Node* fractionalNodes(Node* head, int k) {  
    if (k <= 0 || head == NULL) return NULL;  
    Node* fractionalNode = NULL;  
    // Traverse the given list  
    int i = 0;  
    for (Node* temp = head; temp != NULL; temp = temp->next) {  
        // For every k nodes, we move fractionalNode one step ahead.  
        if (i % k == 0) {  
            // First time we see a multiple of k  
            if (fractionalNode == NULL) fractionalNode = head;  
            else fractionalNode = fractionalNode->next;  
        }  
        i++;  
    }  
    return fractionalNode;  
}
```

Q. 9b Given a singly linked list and a number k, find the last node whose $n \% k == 0$, where n is the number of nodes in the list.

Examples:

Input: list = 1->2->3->4->5->6->7 and k = 3

Output: 6

Input: list = 3->7->1->9->8 and k = 2

Output: 9

Solution:

1. Take a pointer modularNode and initialize it with NULL. Traverse the linked list.
2. For every $i \% k = 0$, update modularNode.

```
Node* modularNode(Node* head, int k) {  
    if (k <= 0 || head == NULL)  
        return NULL;  
    int i = 1;  
    Node* modularNode = NULL;  
    for (Node* temp = head; temp != NULL; temp = temp->next) {  
        if (i % k == 0)  
            modularNode = temp;  
        i++;  
    }  
    return modularNode;  
}
```

Q. 9c Given a Linked List, write a function that accepts the head node of the linked list as a parameter and returns the value of node present at $(\text{floor}(\sqrt{n}))^{\text{th}}$ position in the Linked List, where n is the length of the linked list or the total number of nodes in the list.

Input: 1->2->3->4->5->NULL

Output: 2

Input: 10->20->30->40->NULL

Output: 20

Input: 10->20->30->40->50->60->70->80->90->NULL

Output: 30

Solution:

1. Simple method:

The simple method is to first find the total number of nodes present in the linked list, then find the value of $\text{floor}(\sqrt{n})$ where n is the total number of nodes. Then traverse from the first node in the list to this position and return the node at this position. This method traverses the linked list 2 times.

2. Optimized approach: In this method, we can get the required node by traversing the linked list once only.

- Initialize two counters i and j both to 1 and a pointer sqrttn to NULL to traverse till the required position is reached.
- Start traversing the list using head node until the last node is reached.
- While traversing check if the value of j is equal to \sqrt{i} . If the value is equal increment both i and j and sqrttn to point sqrttn->next otherwise increment only i.
- Now, when we will reach the last node of list i will contain value of n, j will contain value of \sqrt{i} and sqrttn will point to node at jth position.

```
int printsqrtn(Node* head) {  
    Node* sqrttn = NULL;  
    int i = 1, j = 1;  
    while (head!=NULL) {  
        if (i == j*j) { // check if j = sqrt(i)  
            if (sqrttn == NULL) // for first node  
                sqrttn = head;  
            else  
                sqrttn = sqrttn->next;  
            j++;  
        }  
        i++;  
        head=head->next;  
    }  
    return sqrttn->data;  
}
```

Q. 10 Given a linked list and a key in it, the task is to move all occurrences of given key to end of linked list, keeping order of all other elements same.

Input : 1 -> 2 -> 2 -> 4 -> 3 and key = 2

Output : 1 -> 4 -> 3 -> 2 -> 2

Input : 6 -> 6 -> 7 -> 6 -> 3 -> 10 and key = 6

Output : 7 -> 3 -> 10 -> 6 -> 6 -> 6

Solution:

pCrawl => Pointer to traverse the whole list one by one.

pKey => Pointer to an occurrence of key if a key is found, else same as pCrawl.

We start both of the above pointers from head of linked list. We move pKey only when pKey is not pointing to a key. We always move pCrawl. So, when pCrawl and pKey are not same, we must have found a key which lies before pCrawl, so we swap data of pCrawl and pKey, and move pKey to next location. The loop invariant is, after swapping of data, all elements from pKey to pCrawl are keys.

```
void moveToEnd(Node* head, int key) {
    // Keeps track of locations where key is present.
    struct Node* pKey = head;
    struct Node* pCrawl = head;
    while (pCrawl != NULL) {
        // If current pointer is not same as pointer to a key location,
        // then we must have found a key in linked list. We swap data of
        // pCrawl and pKey and move pKey to next position.
        if (pCrawl != pKey && pCrawl->data != key) {
            pKey->data = pCrawl->data;
            pCrawl->data = key;
            pKey = pKey->next;
        }
        // Find next position where key is present
        if (pKey->data != key)
            pKey = pKey->next;
        // Moving to next Node
        pCrawl = pCrawl->next;
    }
}
```

Chapter 4 – Other Linked Lists

4.1 Circularly linked lists

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

Insertion in CLL:

In a circular linked list, a node can be added in four ways:

- a) Insertion in empty list
- b) Insertion at the beginning of the list
- c) Insertion at the end of the list
- d) Insertion in between the nodes

In circular linked lists, we don't use 'head' pointer to insert nodes. We use 'last' pointer.

For case a) - Initially the 'last' pointer will be NULL. After insertion, 'last' pointer points to added node. Since added node is first and last node, it points to itself.

For case b) - To insert a node at the beginning of list, do:

1. Create a node, say T.
2. Make T->next = last->next.
3. last->next = T.

For case c) - To insert a node at the end of list, do:

1. Create a node, say T.
2. Make T->next = last->next.
3. last->next = T.
4. last = T.

For case d) - To insert a node in between the nodes, do:

1. Create a node, say T.
2. Search the node after which T is to be inserted, say that node be P.
3. Make T->next = P->next.
4. P->next = T.

```
Node* addToEmpty (Node* last, int data) {  
    if (last != NULL)    return last;  
    Node* temp = new Node(data);  
    last = temp;  
    last->next = last;  
    return last;  
}
```

```

Node* addBegin (Node* last, int data) {
    if (last == NULL)
        return addToEmpty(last, data);
    Node* temp = new Node(data);
    temp->next = last->next;
    last->next = temp;
    return last;
}

Node* addEnd (Node* last, int data) {
    if (last == NULL)
        return addToEmpty(last, data);
    Node* temp = new Node(data);
    temp->next = last->next;
    last->next = temp;
    last = temp;
    return last;
}

Node* addAfter (Node* last, int data, int item) {
    if (last == NULL)
        return NULL;
    Node *temp, *p;
    p = last->next;
    do {
        if (p->data == item) {
            temp = new Node(data);
            temp->next = p->next;
            p->next = temp;
            if (p == last)
                last = temp;
            return last;
        }
        p = p->next;
    } while (p != last->next);
    cout << item << " not present in the list.\n";
    return last;
}

```

Here is the function to print a CLL:

```

void printList (Node* last) {
    Node* temp;
    if (last == NULL) {
        cout << "List is empty.\n";
        return;
    }

```

```

temp = last->next;
do {
    cout << temp->data << " ";
    temp = temp->next;
} while (temp != last->next);
cout << endl;
}

```

Deletion in CLL:

If the list is empty, we will simply return.

If the list is not empty, then we define two pointers 'curr' and 'prev'. Initialize the 'curr' pointer with the head node. Traverse the list using 'curr' pointer and find the node to be deleted. Maintain a 'prev' node, which will be previous to 'curr' pointer at any time.

- If the node is found, check if it's the only node in the list.
If yes, set head = NULL and delete curr.
- If the list has more than one node, check if it is the first node of the list.
If yes, then move the prev pointer to last node. Then,
head = head->next and prev->next = head. Delete curr.
- If curr is not first node, we check if it is the last node in the list.
If yes, set prev->next = head and delete curr.
- If node to be deleted is neither the first node and nor the last node,
set prev->next = temp->next and delete curr.

```

void deleteNode (Node** head, int key) {
    if (*head == NULL) return;
    if ((*head)->data == key && (*head)->next == *head) {
        delete *head;
        *head = NULL;
    }
    Node* last = *head, *d;
    // If head is to be deleted
    if ((*head)->data == key) {
        // Find the last node of the list
        while (last->next != *head)
            last = last->next;
        // Point last node to the next of head i.e. second last node
        last->next = (*head)->next;
        delete *head;
        *head = last->next;
    }
    // Either the node to be deleted is not found
    // Or the end of the list is not reached
    while (last->next != *head && last->next->data != key) {
        last = last->next;
    }
}

```

```

// If node to be deleted was found
if (last->next->data == key) {
    d = last->next;
    last->next = d->next;
    delete d;
}
else
    cout << "No such key found!\n";
}

```

Traversing CLL:

Finding the length of CLL is a very simple task and here is the code:

```

int countNodes(struct Node* head) {
    struct Node* temp = head;
    int result = 0;
    if (head != NULL) {
        do {
            temp = temp->next;
            result++;
        } while (temp != head);
    }
    return result;
}

```

How to check whether the linked list given is circular or not? Simple – if we reached NULL, then No. If we reached head, then Yes.

```

bool isCircular(struct Node *head) {
    if (head == NULL)
        return true; // An empty linked list is circular
    struct Node *node = head->next;
    // Loop will stop in both cases (1) If Circular (2) Not circular
    while (node != NULL && node != head)
        node = node->next;
    // If loop stopped because of circular condition
    return (node == head);
}

```

Sorted insert in CLL:

- 1) Linked List is empty:
 - a) since new_node is the only node in CLL, make a self-loop.
`new_node->next = new_node;`
 - b) change the head pointer to point to new node.
`*head_ref = new_node;`

2) New node is to be inserted just before the head node:

a) Find out the last node using a loop.

```
while(current->next != *head_ref)
    current = current->next;
```

b) Change the next of last node.

```
current->next = new_node;
```

c) Change next of new node to point to head.

```
new_node->next = *head_ref;
```

d) Change the head pointer to point to new node.

```
*head_ref = new_node;
```

3) New node is to be inserted somewhere after the head:

a) Locate the node after which new node is to be inserted.

```
while (current->next != *head_ref && current->next->data < data)
    current = current->next;
```

b) Make next of new_node as next of the located pointer

```
new_node->next = current->next;
```

c) Change the next of the located pointer

```
current->next = new_node;
```

```
void sortedInsert (Node** head_ref, Node* new_node) {
    Node* current = *head_ref;
    if (current == NULL) {
        new_node->next = new_node; *head_ref = new_node;
    }
    else if (current->data >= new_node->data) {
        // If the value is smaller than head's value
        // then we need to change next of last node
        while (current->next != *head_ref)
            current = current->next;
        current->next = new_node; new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else {
        // Locate the node before the point of insertion
        while (current->next != *head_ref &&
               current->next->data < new_node->data)
            current = current->next;
        new_node->next = current->next; current->next = new_node;
    }
}
```

Split CLL in two halves:

1. Store the mid and last pointers of the CLL using tortoise and hare algorithm.
2. Make the second half circular
3. Make the first half circular
4. Set head (or start) pointers of the two linked lists.

```

void splitList (Node* head, Node** head1_ref, Node** head2_ref) {
    Node* slow_ptr = head, *fast_ptr = head;
    if (head == NULL) return;
    /* If there are odd nodes in the circular list
       then fast_ptr->next becomes head and for
       even nodes fast_ptr->next->next becomes head. */
    while (fast_ptr->next != head && fast_ptr->next->next != head) {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }
    // If there are even number of nodes, then move fast_ptr
    if (fast_ptr->next->next == head)
        fast_ptr = fast_ptr->next;
    // Set the head pointer of first half
    *head1_ref = head;
    // Set the head pointer of second half
    if (head->next != head)
        *head2_ref = slow_ptr->next;
    // Make second half circular
    fast_ptr->next = slow_ptr->next;
    // Make first half circular
    slow_ptr->next = head;
}

```

Exchange first and last nodes:

The idea is to first find the pointer to the previous of last node. Let this node be 'p'. Now we change next links so that the last and first nodes are swapped.

```

Node* exchangeNodes (Node* head) {
    Node* p = head;
    // Find previous of last node
    while (p->next->next != head)
        p = p->next;
    // Now exchange the nodes
    p->next->next = head->next;
    head->next = p->next;
    p->next = head;
    head = head->next;
    return head;
}

```

4.2 Doubly Linked Lists

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

Advantages over singly linked list:

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- 3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list:

- 1) Every node of DLL requires extra space for a previous pointer. However, it is possible to implement with single pointer only. (This is discussed later).
- 2) All operations require an extra pointer previous to be maintained. Due to this, task of fixing links becomes little cumbersome.

Insertion in DLL:

A node can be added in a DLL in three ways:

- 1) At the front of the DLL
- 2) After a given node
- 3) At the end of the DLL

```
/* The following function inserts the node in front of DLL */
void push (Node** head_ref, int new_data) {
    // 1. Allocate new node
    Node* new_node = new Node(new_data);
    // 2. Make next of new_node as head and previous as NULL
    new_node->next = *head_ref;
    new_node->prev = NULL;
    // 3. Change prev of head_node to new_node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;
    // 4. Move the head to point to the new_node
    (*head_ref) = new_node;
}
```

```

/* The following function inserts the node after the given node */
void insertAfter (Node* prev_node, int new_data) {
    // 1. Check if the previous node given is NULL
    if (prev_node == NULL) {
        cout << "The given previous node cannot be NULL.\n";
        return;
    }
    // 2. Allocate a new node
    Node* new_node = new Node(new_data);
    // 3. Make next of new_node as next of previous node
    new_node->next = prev_node->next;
    // 4. Make the next of prev_node as new node
    prev_node->next = new_node;
    // 5. Make prev_node as previous of new node
    new_node->prev = prev_node;
    // 6. Change previous of new_node's next node
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}

/* Following function adds node in the last of list */
void append (Node** head_ref, int new_data) {
    // 1. Allocate Node
    Node* new_node = new Node(new_data);
    Node* last = *head_ref; // Will be used in step 4
    // 2. This new node will be last. So make next of it as NULL.
    new_node->next = NULL;
    // 3. If the linked list is empty, make new node as head.
    if (*head_ref == NULL) {
        new_node->prev = NULL; *head_ref = new_node;
        return;
    }
    // 4. Else traverse till last node
    while (last->next != NULL)
        last = last->next;
    // 5. Change the next of last node
    last->next = new_node;
    // 6. Make last node as previous of new node
    new_node->prev = last;
}

```

Deletion in DLL:

Let the node to be deleted be 'P'.

1. If node to be deleted is head node, then change the head pointer to next current head.
2. Set next of previous to 'P', if previous to 'P' exists.
3. Set prev of next to 'P', if next to 'P' exists.

```

void deleteNode (Node** head_ref, Node* P) {
    if (*head_ref == NULL || P == NULL)
        return;
    // If node to be deleted is head node
    if (*head_ref == P)
        *head_ref = P->next;
    // Change next only if node to be deleted is NOT the last node
    if (P->next != NULL)
        P->next->prev = P->prev;
    // Change prev only if node to be deleted is NOT the first node
    if (P->prev != NULL)
        P->prev->next = P->next;
    delete P;
}

```

Remove Duplicates:

```

removeDuplicates(head_ref, x):
    if head_ref == NULL
        return
    Initialize current = head_ref
    while current->next != NULL
        if current->data == current->next->data
            deleteNode(head_ref, current->next)
        else
            current = current->next

void removeDuplicates(Node** head_ref) {
    if ((*head_ref) == NULL)      return;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current->next != NULL) {
        // Compare current node's data with next node's data
        if (current->data == current->next->data)
            // delete the node pointed to by 'current->next'
            deleteNode(head_ref, current->next);
        // else simply move to the next node
        else
            current = current->next;
    }
}

```

Reverse DLL:

All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointe in the end.

```
void reverse (Node **head_ref) {
    Node *temp = NULL;
    Node *current = *head_ref;
    // swap next and prev for all nodes of DLL
    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }
    // Before changing head, check for the cases like
    // empty list and list with only one node
    if(temp != NULL)
        *head_ref = temp->prev;
}
```

Mergesort DLL:

Merge sort for DLL is same as SLL. The important change here is to modify the previous pointers also while merging two lists.

```
Node* merge(Node *first, Node *second) {
    // If first linked list is empty
    if (!first) return second;
    // If second linked list is empty
    if (!second) return first;
    // Pick the smaller value
    if (first->data < second->data) {
        first->next = merge(first->next,second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else {
        second->next = merge(first,second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}
```

```

Node *split(Node *head)  {
    Node *fast = head,*slow = head;
    while (fast->next && fast->next->next)  {
        fast = fast->next->next;
        slow = slow->next;
    }
    Node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

Node *mergeSort(Node *head)  {
    if (!head || !head->next)
        return head;
    Node *second = split(head);
    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);
    // Merge the two sorted halves
    return merge(head,second);
}

```

Memory Efficient DLL:

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

Ordinary Representation:

Node A: prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B: prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C: prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D: prev = add(C), next = NULL // previous is address of C and next is NULL

XOR List Representation:

Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A: npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B: npx = add(A) XOR add(C) // XOR of address of A and address of C

Node C: npx = add(B) XOR add(D) // XOR of address of B and address of D

Node D: npx = add(C) XOR 0 // bitwise XOR of address of C and 0

Traversal of XOR Linked List:

We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example, when we are at node C, we must have address of B. XOR of add(B) and *npx* of C gives us the add(D). The reason is simple: *npx*(C) is “add(B) XOR add(D)”. If we do xor of *npx*(C) with add(B), we get the result as “add(B) XOR add(D) XOR add(B)” which is “add(D) XOR 0” which is “add(D)”. So we have the address of next node. Similarly, we can traverse the list in backward direction. This is the complete code for XOR LL:

```
#include <bits/stdc++.h>
#include <iinttypes.h>
using namespace std;

class Node {
public:
    int data; Node* npx; // XOR of next and previous node
    Node(int data) {
        this->data = data; this->npx = NULL;
    }
};

Node* XOR (Node *a, Node *b) {
    return (Node*) ((uintptr_t) (a) ^ (uintptr_t) (b));
}

// Insert a node at the beginning of the XORED linked list
// and makes the newly inserted node as head
void insert(Node **head_ref, int data) {
    Node *new_node = new Node(data);
    /* Since new node is being inserted at the beginning,
       npx of new node will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);
    /* If linked list is not empty, then npx of current head
       node will be XOR of new node and node next to current head
    */
    if (*head_ref != NULL) {
        // *(head_ref)->npx is XOR of NULL and next.
        // So if we do XOR of it with NULL, we get next
        Node* next = XOR((*head_ref)->npx, NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }
    // Change head
    *head_ref = new_node;
}
```

```

// Function to traverse XOR implemented DLL
void printList (Node *head) {
    Node *curr = head, *prev = NULL, *next;

    cout << "Created DLL: \n";
    while (curr != NULL) {
        // print current node
        cout << curr->data << " ";
        // get address of next node: curr->npx is next^prev,
        // so curr->npx^prev will be next^prev^prev which is next
        next = XOR (prev, curr->npx);
        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
    cout << endl;
}

int main () {
    Node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);
    printList (head);
    return (0);
}

```

Chapter 5 – Stacks

5.1 Implementation:

In C Lang:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <limits.h>

struct Stack {
    int* array;
    int top;
    unsigned capacity;
};

struct Stack* createStack (unsigned capacity) {
    struct Stack* stack = (struct Stack*) malloc(sizeof (struct Stack));
    stack -> capacity = capacity;
    stack -> top = -1;
    stack -> array = (int*) malloc((stack -> capacity)*sizeof(int));
    return stack;
}

int isFull (struct Stack* stack) {
    return (stack -> top == stack -> capacity-1);
}

int isEmpty (struct Stack* stack) {
    return (stack->top == -1);
}

void push (struct Stack* stack, int item) {
    if (isFull(stack)) {
        printf ("Overflow.\n");
        return;
    }
    else {
        stack -> array[+(stack->top)] = item;
        printf("%d pushed to stack.\n", item);
    }
}
```

```

int pop (struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Underflow.\n");
        return INT_MIN;
    }
    return (stack->array[(stack->top)--]);
}

int top (struct Stack* stack) {
    if (isEmpty(stack)) {
        return INT_MIN;
    }
    else {
        return (stack->array[stack->top]);
    }
}

int main() {
    struct Stack* stack = createStack(100);
    int option, val;
    do {
        printf ("\n1.PUSH\n2.POP\n3.TOP\n4.EXIT\n");
        scanf("%d", &option);
        switch (option) {
            case 1:
                printf("Enter the number to be pushed: ");
                scanf("%d", &val);
                push(stack, val);
                break;
            case 2:
                printf("%d is popped out.\n", pop(stack));
                break;
            case 3:
                printf ("%d is the top element.\n", top(stack));
                break;
            case 4:
                exit(0);
            default:
                printf ("Invalid option.\n");
        }
    }
    while (option != 4);

    return 0;
}

```

In C++ Lang:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class Stack {
private:
    int top;
public:
    int max, *array;
Stack() {
    top = -1;
}
void size(int x);
bool isEmpty();
bool isFull();
void push(int item);
int pop();
int peek();
};

void Stack :: size (int x) {
    max = x;
    array = new int[x];
}

bool Stack :: isEmpty () {
    return (top == -1);
}

bool Stack :: isFull () {
    return (top == max-1);
}

void Stack :: push (int item) {
    array[++(this->top)] = item;
}

int Stack :: pop() {
    return array[(this->top)--];
}

int Stack :: peek () {
    return array[this->top];
}
```

```

int main() {
    Stack stack;
    int max, option, val;

    cout << "Enter the size of the stack: ";
    cin >> max;

    stack.size(max);

    do {
        cout << "\n1.PUSH\n2.POP\n3.TOP\n4.EXIT\n";
        cin >> option;

        switch(option) {
            case 1:
                if (stack.isFull())
                    cout << "Overflow.\n";
                else {
                    cout << "Enter the value to be pushed: ";
                    cin >> val;
                    stack.push(val);
                }
                break;

            case 2:
                if (stack.isEmpty())
                    cout << "Underflow.\n";
                else
                    cout << stack.pop() << " popped out of stack.\n";
                break;

            case 3:
                cout << stack.peek() << " is top element.\n";
                break;

            case 4:
                exit (0);

            default:
                cout << "Invalid Option.\n";
        }
    } while (option != 4);

    return 0;
}

```

Dynamic Stacks:

Limitation of array-based implementation:

The maximum size of the stack must be defined first and it cannot be changed.

Try this:

If the stack is full and push is called, we may create a new stack with size one more and then copy all elements of previous stack into this new stack and finally push the element.

But this way of incrementing array size is too expensive. For e.g. at $n = 1$, to push an element, we create a new array of size 2 and copy previous elements and add the new element. Then at $n = 2$, we create a new array of size 3 and copy previous elements and add the new element. Going in same fashion, at $n = n-1$, we create an array of size n and copy all old elements and add new element in end.

Thus, time taken for n push operations is proportional to,

$$T(n) = 1 + 2 + \dots + n - 1 \sim O(n^2)$$

Proper approach:

The technique used is called ‘Array Doubling Technique.’ If the array gets full, create a new array of twice the size and copy the items.

Let us assume we started at $n = 1$ and moved till $n = 32$. This means we performed doubling at 1, 2, 4, 8, 16.

At $n = 1$, we do 1 copy operation.

At $n = 2$, we do 2 copy operations.

At $n = 4$, we do 4 copy operations.

$$\begin{aligned} \text{Hence, total copy operations} &= 1 + 2 + 4 + 8 + 16 \\ &= 31 \sim 2n \text{ i.e. } 32 \text{ (where } n = 16) \end{aligned}$$

Thus, total time $T(n)$ of n push operation is proportional to,

$$\begin{aligned} T(n) &= 1 + 2 + 4 + 8 + \dots + n/4 + n/2 + n \\ &= n(1 + 1/2 + 1/4 + \dots + 8/n + 4/n + 2/n + 1/n) \\ &\sim n(2) \\ &\sim 2n \\ T(n) &= O(n) \end{aligned}$$

Hence amortized time complexity for single push operation is $O(1)$.

```

#include <bits/stdc++.h>
using namespace std;

class dynamicStack {
private:
    int top; void doubleStack();
public:
    int capacity, *array;

dynamicStack() {
    top = -1; capacity = 1;
    array = new int[capacity];
}
bool isFull();
bool isEmpty();
void push(int item);
int pop();
int peek();
};

void dynamicStack :: doubleStack() {
// Doubling the capacity
capacity *= 2;
// Copying available data into temporary array
// and then deleting the original array
int* temp = new int[capacity];
for (int i = 0; i < capacity/2; i++)
    temp[i] = array[i];
delete array;
// Allocating new memory to original again
// which is double the size as of previous one
// and deleting the temporary array after copying
// the elements into newly allocated array
array = new int[capacity];
for (int i = 0; i < capacity/2; i++)
    array[i] = temp[i];
delete temp;
}

bool dynamicStack :: isFull() {
    return (top == capacity-1);
}

bool dynamicStack :: isEmpty() {
    return (top == -1);
}

```

```

void dynamicStack :: push(int item) {
    if (this->isFull())
        this->doubleStack();
    array[++(this->top)] = item;
}

int dynamicStack :: pop() {
    return array[(this->top)--];
}

int dynamicStack :: peek () {
    return array[this->top];
}

int main() {
    dynamicStack stack;
    int option, val;
    do {
        cout << "\n1.PUSH\n2.POP\n3.TOP\n4.EXIT\n";
        cin >> option;
        switch(option) {
            case 1:
                cout << "Enter the value to be pushed: ";
                cin >> val;
                stack.push(val);
                break;
            case 2:
                if (stack.isEmpty())
                    cout << "Underflow\n";
                else
                    cout << stack.pop() << " popped out of stack.\n";
                break;
            case 3:
                cout << stack.peek() << " is top element.\n";
                break;
            case 4:
                exit (0);
            default:
                cout << "Invalid Option.\n";
        }
    }
    while (option != 4);

    return 0;
}

```

Stack from C++ STL:

```
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

int main() {
    stack <int> s;
    int option, val, topmost, popped, sized;
    do {
        cout << "\n1.PUSH\n2.POP\n3.TOP\n4.SIZE\n5.EXIT\n";
        cin >> option;
        switch (option) {
            case 1:
                cout << "Enter the value to be pushed: ";
                cin >> val;
                s.push(val);
                break;
            case 2:
                if (s.empty())
                    cout << "Underflow" << endl;
                else {
                    popped = s.top();
                    s.pop();
                    cout << popped << " popped out of stack.\n";
                }
                break;
            case 3:
                topmost = s.top();
                cout << topmost << " is top element." << endl;
                break;
            case 4:
                sized = s.size();
                cout << "Size of stack is " << sized << endl;
                break;
            case 5:
                exit (0);
            default:
                cout << "Invalid option.\n";
        }
    } while (option != 5);

    return 0;
}
```

5.2 Design Problems

Create a data structure twoStacks that represent two stacks and both the stacks should use same array for storing elements.

```
push1 (int x) -> pushes x to first stack  
push2 (int x) -> pushes x to second stack  
pop1()         -> pops an element from stack 1  
pop2()         -> pops an element from stack 2
```

Method 1:

A simple way to implement two stacks is to divide the array in two halves and use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2. The problem with this method is inefficient use of array space. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2:

This method efficiently utilizes the available space. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This is implemented in following code.

```
#include <iostream>  
#include <stdlib.h>  
using namespace std;  
  
class twoStacks {  
private:  
    int *arr, size, top1, top2;  
public:  
    twoStacks(int n) {  
        size = n; arr = new int[n];  
        top1 = -1; top2 = size;  
    }  
    void push1 (int x);  
    void push2 (int x);  
    int pop1 ();  
    int pop2 ();  
};
```

```

void twoStacks :: push1 (int x) {
    // There is atleast one empty space for new element
    if (top1 < top2 - 1)
        arr[++top1] = x;
    else {
        cout << "Stack Overflow\n"; exit(1);
    }
}

void twoStacks :: push2 (int x) {
    // There is atleast one empty space for new element
    if (top1 < top2 - 1)
        arr[--top2] = x;
    else {
        cout << "Stack Overflow\n"; exit (1);
    }
}

int twoStacks :: pop1 () {
    if (top1 >= 0) {
        int x = arr[top1]; top1--;
        return x;
    }
    else {
        cout << "Stack Underflow\n"; exit (1);
    }
}

int twoStacks :: pop2 () {
    if (top2 < size) {
        int x = arr[top2]; top2++;
        return x;
    }
    else {
        cout << "Stack Underflow\n"; exit (1);
    }
}

int main() {
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}

```

Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be $O(1)$. To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, etc.

Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum.

Let us see how push() and pop() operations work.

Push(int x)

- 1) Push x to the first stack (the stack with actual elements)
- 2) Compare x with the top element of the second stack (the auxiliary stack).
Let the top element be y.
 - a) If x is smaller than y then push x to the auxiliary stack.
 - b) If x is greater than y then push y to the auxiliary stack.

int Pop()

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin()

- 1) Return the top element of auxiliary stack.

```
#include <bits/stdc++.h>
using namespace std;

// This is simple stack with basic functionalities
class Stack {
private:
    static const int max = 100; int arr[max]; int top;
public:
    Stack() { top = -1; }
    bool isEmpty();
    bool isFull();
    int pop();
    void push(int x);
};
```

```

    bool Stack::isEmpty() {
        return (top == -1);
    }

    bool Stack::isFull() {
        return (top == max-1);
    }

    int Stack::pop() {
        if(isEmpty()) {
            cout<<"Stack Underflow.\n"; abort();
        }
        int x = arr[top--];
        return x;
    }

    void Stack::push(int x) {
        if(isFull()) {
            cout<<"Stack Overflow.\n";
            abort();
        }
        arr[++top] = x;
    }

// Our Special Stack class that inherits all basic properties of stack
class SpecialStack: public Stack {
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

// SpecialStack's push() method makes sure that the min stack
// is also updated with appropriate minimum values
void SpecialStack::push(int x) {
    if(isEmpty()==true) {
        Stack::push(x); min.push(x);
    }
    else {
        Stack::push(x);
        int y = min.pop(); min.push(y);
        if( x < y ) min.push(x);
        else min.push(y);
    }
}

```

```

// SpecialStack's pop() method removes top element from min stack also.
int SpecialStack::pop() {
    int x = Stack::pop(); min.pop();
    return x;
}

// SpecialStack's method to find minimum element
int SpecialStack::getMin() {
    int x = min.pop(); min.push(x);
    return x;
}

int main() {
    SpecialStack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.getMin() << endl;
    s.push(5);
    cout << s.getMin() << endl;
    return 0;
}

```

How to implement a stack which will support following operations in $O(1)$ time complexity?

- 1) push() which adds an element to the top of stack.
- 2) pop() which removes an element from top of stack.
- 3) findMiddle() which will return middle element of the stack.
- 4) deleteMiddle() which will delete the middle element.

The important question is, whether to use linked list or array implementation of stack? We need to find and delete middle element. Deleting an element from middle is not $O(1)$ for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

The idea is to use Doubly Linked List (DLL). We can delete middle element in $O(1)$ time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

```
#include <bits/stdc++.h>
using namespace std;
```

```

class DLLNode {
public:
    DLLNode *prev, *next; int data;
    DLLNode(int new_data) {
        data = new_data;
        prev = next = NULL;
    }
};

// Stack is implemented using DLL. It maintains pointer to head
// and middle nodes and count of nodes.
class myStack {
public:
    DLLNode *head, *mid;
    int count;
    myStack() {
        head = NULL; mid = NULL; count = 0;
    }
    void push (int item);
    int pop ();
    int findMiddle ();
    void deleteMiddle ();
};

void myStack :: push(int item) {
    // Allocate DLL node to be pushed in stack
    DLLNode* new_node = new DLLNode(item);
    // Since we are adding node in the beginning previous will always
    // be NULL. We just need to link the old list to this new node
    new_node->next = head;
    count++;
    // We need to change the middle pointer only when -
    // a) Linked List becomes empty
    // b) Number of linked list is odd
    if (count == 1) {
        mid = new_node;
    }
    else {
        head->prev = new_node;
        // Update if count is even
        if (count%2 == 0)
            mid = mid->prev;
    }
    head = new_node;
}

```

```

int myStack :: pop() {
    if (count == 0) {
        cout << "Stack Underflow.\n";
        return -1;
    }
    DLLNode* start = head;
    int item = head->data;
    head = head->next;
    // If linked list is not yet empty,
    // update prev of new head as NULL
    if (head != NULL)
        head->prev = NULL;
    count--;
    // Update the mid pointer when we have
    // odd number of items in the stack
    if (count%2 == 1)
        mid = mid->next;
    delete start;
    return item;
}

int myStack :: findMiddle() {
    if (count == 0) {
        cout << "Stack Underflow.\n";
        return -1;
    }
    return mid->data;
}

void myStack :: deleteMiddle() {
    // We have pointer to the middle element. We know how
    // to delete a node in LL with just a pointer of that node.
    // Copy the data of next node and delete the next node.
    DLLNode* temp = mid->next;
    mid->data = temp->data;
    mid->next = temp->next;
    temp = temp->next;
    temp->prev = mid;
    delete temp;
}

int main() {
    myStack s;
    for (int i = 1; i < 5; i++)
        s.push(2*i);
}

```

```

cout << s.findMiddle() << endl;
s.pop();
s.pop();
cout << s.findMiddle() << endl;
for (int i = 5; i < 8; i++)
    s.push(2*i+1);
cout << s.findMiddle() << endl;
s.deleteMiddle();
cout << s.findMiddle() << endl;

return 0;
}

```

5.3 Operations on Stacks

Reverse a stack:

The idea of the solution is to hold all values in function Call-Stack of recursion until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.

```

1 <-- top
2
3
4

```

First 4 is inserted at the bottom.

```
4 <-- top
```

Then 3 is inserted at the bottom

```
4 <-- top
3
```

Then 2 is inserted at the bottom

```
4 <-- top
3
2
```

Then 1 is inserted at the bottom

```
4 <-- top
3
2
1
```

So, we need a function that inserts at the bottom of a stack using the above given basic stack function.

void insertAtBottom(): First pops all stack items and stores the popped item in function call stack using recursion. And when stack becomes empty, pushes new item and all items stored in call stack.

void reverse(): This function mainly uses insertAtBottom() to pop all items one by one and insert the popped items at the bottom.

```
void insertAtBottom (stack<int> &s, int element) {
    if (s.empty()) {
        s.push(element);
        return;
    }
    int x = s.top();
    s.pop();
    insertAtBottom(s, element);
    s.push(x);
}

void reverse (stack<int> &s) {
    if (s.empty()) return;
    int temp = s.top();
    s.pop();
    reverse(s);
    insertAtBottom(s, temp);
}
```

Here is the utility function to print a stack:

```
void printStack (stack<int> s) {
    if (s.empty())
        return;
    int x = s.top();
    s.pop();
    printStack(s);
    cout << x << " ";
    s.push(x);
}
```

If we are asked to reverse the stack in $O(1)$ space, we can't use recursion. In this case - we will use loops. We can reverse a string in $O(n)$ time if we internally represent the stack as a linked list. Reverse a stack would require a reversing a linked list which can be done with $O(n)$ time and $O(1)$ extra space.

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node (int data) {
        this->data = data;
        this->next = NULL;
    }
};

class Stack {
Node* top;
public:
    void push(int data);
    Node* pop();
    void display();
    void reverse();
};

void Stack::push(int data) {
if (top == NULL) {
    top = new Node (data);
    return;
}
Node* t = new Node(data);
t->next = top;
top = t;
}

Node* Stack :: pop() {
Node* s = top;
top = top->next;
return s;
}

void Stack :: display() {
Node* curr = top;
while (curr != NULL) {
    cout << curr->data << " ";
    curr = curr->next;
}
cout << endl;
}

```

```

void Stack:: reverse() {
    Node *prev, *cur, *succ;
    cur = prev = top;
    cur = cur->next;
    prev->next = NULL;
    while (cur != NULL) {
        succ = cur->next;
        cur->next = prev;
        prev = cur;
        cur = succ;
    }
    top = prev;
}

int main() {
    Stack *s = new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    s->push(4);
    cout << "Original Stack:\n";
    s->display();
    s->reverse();
    cout << "Reversed Stack:\n";
    s->display();
    return 0;
}

```

Sort a stack:

This problem is mainly a variant of reverse stack using recursion. The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order.

```

void sortedInsert(stack<int> &s, int x) {
    // Base case is - either the stack is empty or newly added element
    // is greater than top i.e. it is more than all existing elements
    if (s.empty() || x>s.top()) {
        s.push(x); return;
    }
    // If top is greater, remove the top item and recur
    int temp = s.top(); s.pop(); sortedInsert(s, x);
    // Put back the top item back to stack
    s.push(temp);
}

```

```

void sortRecursive (stack<int> &s) {
    if (!s.empty()) {
        // Remove the top element
        int x = s.top(); s.pop();
        // Sort the remaining stack
        sortRecursive(s);
        // Push the top item back in sorted order
        sortedInsert (s, x);
    }
}

void sortIterative (stack<int> &s) {
    // Using a temporary stack
    stack <int> tempStack;
    while (!s.empty()) {
        // Pop the first element
        int temp = s.top(); s.pop();
        // While the temporary stack is not empty
        // and top of stack is greater than temp
        while (!tempStack.empty() && tempStack.top() > temp) {
            // Pop from temporary stack and push it into stack
            s.push(tempStack.top());
            tempStack.pop();
        }
        // Push temp in tempStack back
        tempStack.push(temp);
    }
    // Now, tempStack contains all elements of s in sorted order.
    // Copy the elements from tempStack to s.
    stack<int> helper;
    while (!tempStack.empty()) {
        helper.push(tempStack.top());
        tempStack.pop();
    }
    while (!helper.empty()) {
        s.push(helper.top());
        helper.pop();
    }
}

```

Sort using Stack:

Given an array of elements, task is to sort these elements using stack.

This is how we do this:

1. Create a temporary stack say tmpStack.
2. While input stack is NOT empty do this:
 - Pop an element from input stack call it temp
 - while temporary stack is NOT empty and top of temporary stack is greater than temp, pop from temporary stack and push it to the input stack
 - push temp in temporary stack
3. The sorted numbers are in tmpStack. Put them back to array.

Here is the dry run for above steps:

Element taken out	Input	tmpStack
23	[34, 3, 31, 98, 92]	[23]
92	[34, 3, 31, 98]	[23, 92]
98	[34, 3, 31]	[23, 92, 98]
31	[34, 3, 98, 92]	[23, 31]
92	[34, 3, 98]	[23, 31, 92]
98	[34, 3]	[23, 31, 92, 98]
3	[34, 98, 92, 31, 23]	[3]
23	[34, 98, 92, 31]	[3, 23]
31	[34, 98, 92]	[3, 23, 31]
92	[34, 98]	[3, 23, 31, 92]
98	[34]	[3, 23, 31, 92, 98]
34	[98, 92]	[3, 23, 31, 34]
92	[98]	[3, 23, 31, 34, 92]
98	[]	[3, 23, 31, 34, 92, 98]

final sorted list: [3, 23, 31, 34, 92, 98]

```
stack<int> sortStack(stack<int> input) {
    stack<int> tmpStack;
    while (!input.empty()) {
        int tmp = input.top(); input.pop();
        // while temporary stack is not empty
        // and top of stack is smaller than temp
        while (!tmpStack.empty() && tmpStack.top() < tmp) {
            // pop from temporary stack and push it to the input stack
            input.push(tmpStack.top());
            tmpStack.pop();
        }
        // push temp in temporary stack
        tmpStack.push(tmp);
    }
    return tmpStack;
}
```

```

void sortArrayUsingStacks(int arr[], int n) {
    stack<int> input;
    for (int i=0; i<n; i++)
        input.push(arr[i]);
    // Sort the temporary stack
    stack<int> tmpStack = sortStack(input);
    // Put stack elements in arr[]
    for (int i=0; i<n; i++) {
        arr[i] = tmpStack.top();
        tmpStack.pop();
    }
}

```

5.4 Application of Stacks

Balanced Parentheses:

Given an expression string exp, write a program to examine whether the pairs and the orders of { }, (), [] are correct in exp.

Input: exp = “[(){}{[()])()”

Output: Balanced

Input: exp = “[()”

Output: Not Balanced

Algorithm:

1. Declare a character stack S.
2. Now traverse the expression string exp.
 - a) If the current character is a starting bracket (‘(’ or ‘{‘ or ‘[‘) then push it to stack.
 - b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3. After complete traversal, if there is some starting bracket left in stack then “not balanced”.

```

bool areParenthesesBalanced (string expr) {
    stack<char> s;
    char x;
    // Traversing the expression
    for (int i = 0; i < expr.length(); i++) {
        if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{') {
            s.push(expr[i]);
            continue;
        }

```

```

        // If current character is not opening bracket, then it must
        // be closing. So stack cannot be empty at this point.
        if (s.empty())    return false;
        switch(expr[i]) {
            case ')':
                x = s.top();
                s.pop();
                if (x=='{' || x=='[')
                    return false;
                break;
            case ']':
                x = s.top();
                s.pop();
                if (x=='(' || x=='{')
                    return false;
                break;
            case '}':
                x = s.top();
                s.pop();
                if (x=='(' || x=='[')
                    return false;
                break;
        }
    }
    // Finally check if stack is empty
    return s.empty();
}

```

Suppose we are being asked a similar question like above - Find the maximum depth of nested and balanced parentheses in a given expression (consider here only circular brackets).

Maximum depth is the maximum number of opening parentheses at any point of the string which are not yet closed by their matching closing parentheses.

A simple solution is to use a stack that keeps track of current open parentheses.

- 1) Create a stack.
- 2) Traverse the string, do following for every character
 - a) If current character is '(' push it to the stack.
 - b) If character is ')', pop an element.
 - c) Maintain maximum count during the traversal.

This can also be done without using stack.

- 1) Take two variables max and current_max, initialize both of them as 0.
- 2) Traverse the string, do following for every character
 - a) If current character is '(', increment current_max and update max value if required.
 - b) If character is ')'. Check if current_max is positive or not (this condition ensure that parentheses are balanced). If positive that means we previously had a '(' character. So decrement current_max without worry. If not positive then the parentheses are not balanced. Thus return -1.
- 3) If current_max is not 0, then return -1 to ensure that the parentheses are balanced. Else return max.

Here is the code for non-stack implementation:

```

int maxDepth(string S) {
    int current_max = 0;
    int max = 0;
    int n = S.length();
    for (int i = 0; i < n; i++) {
        if (S[i] == '(') {
            current_max++;
            if (current_max > max)
                max = current_max;
        }
        else if (S[i] == ')') {
            if (current_max > 0)
                current_max--;
            else
                return -1;
        }
    }
    if (current_max != 0)
        return -1;
    return max;
}

```

Prefix, Infix and Postfix:

Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets () to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication. Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:
((A (B C +) *) D)
Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to / * A + B C D

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:
(/ (* A (+ B C)) D)

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication). Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A). We will now see the interconversion between these notations.

Infix to Postfix conversion:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 - a) If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '(', push it.)
 - b) Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

```
int precedence (char c) {  
    if(c == '^')  return 3;  
    else if(c == '*' || c == '/')  return 2;  
    else if(c == '+' || c == '-')  return 1;  
    else  return -1;  
}  
void infixToPostfix(string s) {  
    stack<char> st; st.push('N');  
    int l = s.length(); string ns;  
    for(int i = 0; i < l; i++) {  
        // If the scanned character is an operand, add it to output string.  
        if((s[i] >= 'a' && s[i] <= 'z')||(s[i] >= 'A' && s[i] <= 'Z'))  
            ns+=s[i];  
        // If the scanned character is an '(', push it to the stack.  
        else if(s[i] == '(')  
            st.push('(');  
        // If the scanned character is an ')', pop and to output string  
        // from the stack until an '(' is encountered.  
        else if(s[i] == ')') {  
            while(st.top() != 'N' && st.top() != '(') {  
                char c = st.top(); st.pop(); ns += c;  
            }  
            if(st.top() == '(') {  
                char c = st.top(); st.pop();  
            }  
        }  
    }  
}
```

```

//If an operator is scanned
else {
    while(st.top() != 'N' &&
          precedence(s[i]) <= precedence(st.top())) {
        char c = st.top(); st.pop(); ns += c;
    }
    st.push(s[i]);
}
}

//Pop all the remaining elements from the stack
while(st.top() != 'N') {
    char c = st.top(); st.pop(); ns += c;
}
cout << ns << endl;
}

int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}

```

In above code, we have converted Infix to Postfix. Now, to convert Infix to Prefix, do the following:

1. Reverse the infix expression i.e. A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.
2. Obtain the postfix expression of the modified expression i.e. CB*A+.
3. Reverse the postfix expression. Hence in our example prefix is +A*BC.

Postfix to Infix conversion:

1. While there are input symbols left, read the next symbol from the input.
2. If the symbol is an operand push it onto the stack.
3. Else the symbol is an operator. Pop the top 2 values from the stack. Put the operator, within the values as arguments and form a string. Push the resulted string back to stack.
4. If there is only one value in the stack that value in the stack is the desired infix string.

```

bool isOperand(char x) {
    return (x >= 'a' && x <= 'z') || (x >= 'A' && x <= 'Z');
}

```

```

string getInfix(string exp) {
    stack<string> s;
    for (int i=0; exp[i]!='\0'; i++) {
        if (isOperand(exp[i])) {
            string op(1, exp[i]);
            s.push(op);
        }
        else {
            string op1 = s.top(); s.pop();
            string op2 = s.top(); s.pop();
            s.push("(" + op2 + exp[i] + op1 + ")");
        }
    }
    return s.top();
}

int main() {
    string exp = "abcd^e-fgh*+^*+i-";
    cout << getInfix(exp);
    return 0;
}

```

Like in previous code, if we are asked to convert - Prefix expression to Infix expression, we will reverse the Prefix string and convert it into Postfix string. Then using above logic, we will convert Postfix to Infix.

Evaluating Postfix, Infix, Prefix:

Postfix Evaluation:

- 1) Create a stack to store operands (or values).
- 2) Scan the given expression and do following for every scanned element.
 - a) If the element is a number, push it into the stack
 - b) If the element is a operator, pop operands for the operator from stack.
Evaluate the operator and push the result back to the stack
- 3) When the expression is ended, the number in the stack is the final answer

```

int evaluatePostfix (string expr) {
    stack<int> s;
    // Scan all characters one by one
    for (int i = 0; i < expr.length(); i++) {
        // If the scanned character is digit, push it to stack
        // NOTE: Here we assume all numbers to be less than 10

        if (isdigit(expr[i]))
            s.push(expr[i]-'0');
    }
}

```

```

    // Scanned character is operator
    else {
        int val1 = s.top(); s.pop();
        int val2 = s.top(); s.pop();
        switch (expr[i]) {
            case '+': s.push(val2 + val1); break;
            case '-': s.push(val2 - val1); break;
            case '*': s.push(val2 * val1); break;
            case '/': s.push(val2 / val1); break;
        }
    }
    return s.top();
}

```

Infix evaluation:

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well-known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra. This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to a postfix notation. Same algorithm can be modified so that it outputs result of evaluation of expression instead of a queue. Trick is using two stacks instead of one, one for operands and one for operators.

Refer this video for explanation: <https://www.youtube.com/watch?v=Wz85Hiwi5MY>

```

int precedence(char op){
    if(op == '+' || op == '-') return 1;
    if(op == '*' || op == '/') return 2;
    return 0;
}
// Function to perform arithmetic operations.
int applyOperation(int a, int b, char op){
    switch(op){
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}
int evaluate (string expression) {
    stack<int> values;
    stack<char> operators;

```

```

for (int i = 0; i < expression.length(); i++) {
    // If token is a whitespace, skip it
    if (expression[i] == ' ') continue;
    else if (expression[i] == '(')
        operators.push(expression[i]);
    else if (isdigit(expression[i])) {
        int val = 0;
        // There may be more than 1 digit in number
        while (i < expression.length() && isdigit(expression[i]))
            val = val*10 + (expression[i++]- '0');
        values.push(val);
    }
    // If closing brace is encountered, then solve entire brace
    else if (expression[i] == ')') {
        while (!operators.empty() && operators.top() != '('){
            int val2 = values.top(); values.pop();
            int val1 = values.top(); values.pop();
            char op = operators.top(); operators.pop();
            values.push(applyOperation(val1, val2, op));
        }
        operators.pop();
    }
    // Else current token is an operator
    else {
        // While top of operators has same or greater precedence to
        // current token (viz. an operator), apply operator on top
        // of operators to top two elements in values stack
        while (!operators.empty() && precedence(operators.top())
               >= precedence(expression[i])) {
            int val2 = values.top(); values.pop();
            int val1 = values.top(); values.pop();
            char op = operators.top(); operators.pop();
            values.push(applyOperation(val1, val2, op));
        }
        operators.push(expression[i]); // Push 'operators'
    }
}
// Entire expression has been parsed at this point,
// apply remaining operations to remaining values.
while(!operators.empty()){
    int val2 = values.top(); values.pop();
    int val1 = values.top(); values.pop();
    char op = operators.top(); operators.pop();
    values.push(applyOperation(val1, val2, op));
}
return values.top();
}

```

```

int main() {
    cout << evaluate("10 + 2 * 6") << "\n";
    cout << evaluate("100 * 2 + 12") << "\n";
    cout << evaluate("100 * ( 2 + 12 )") << "\n";
    cout << evaluate("100 * ( 2 + 12 ) / 14");
    return 0;
}

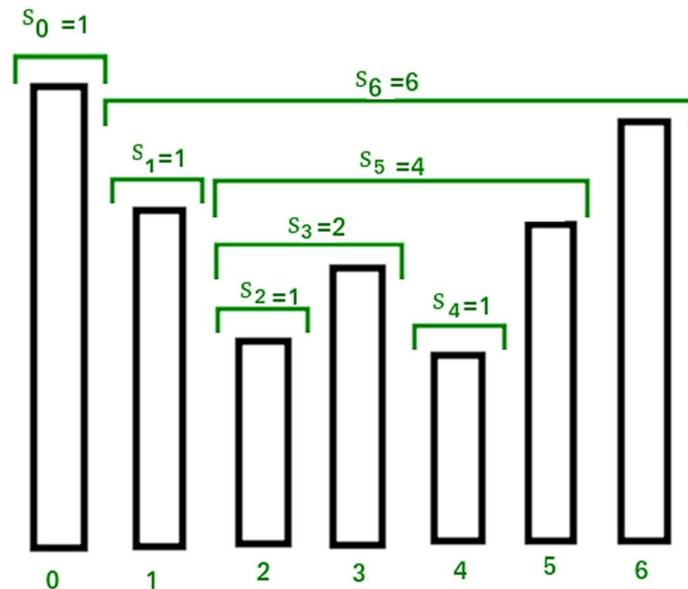
```

Stock Span Problem:

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}.



We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$. To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $price[i]$ and then push the value of day i back into the stack.

```

void calculateSpan(int price[], int n, int S[]) {
    // Create a stack and push index of first element to it
    stack<int> st;
    st.push(0);

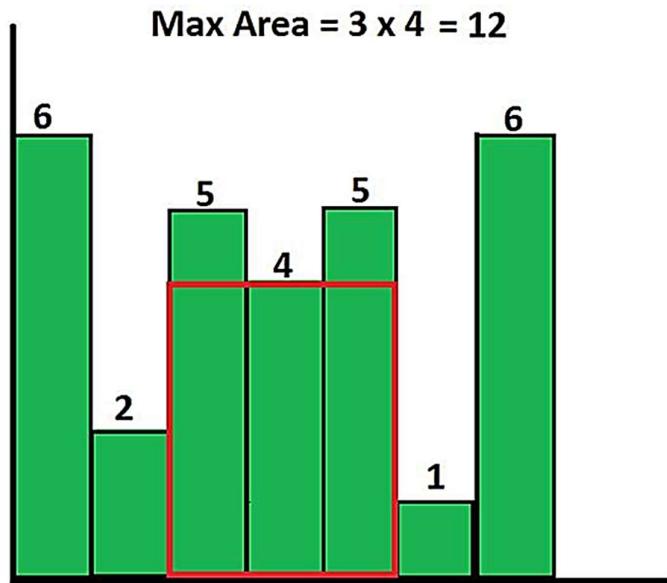
    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++) {
        // Pop elements from stack while stack is not
        // empty and top of stack is smaller than price[i]
        while (!st.empty() && price[st.top()] <= price[i])
            st.pop();
        // If stack becomes empty, then price[i] is greater
        // than all elements on left of it, i.e., price[0],
        // price[1], ..price[i-1]. Else price[i] is greater
        // than elements after top of stack
        S[i] = (st.empty()) ? (i + 1) : (i - st.top());
        // Push this element to stack
        st.push(i);
    }
}

```

Max Area under Histogram:

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars.



1. Create an empty stack.
2. Start from first bar, and do following for every bar 'hist[i]' where i varies from 0 to n-1.
 - a) If stack is empty or hist[i] is higher than the bar at top of stack, then push i to stack.
 - b) If this bar is smaller than the top of stack, then keep removing the top of stack while top of the stack is greater. Let the removed bar be hist[tp]. Calculate area of rectangle with hist[tp] as the smallest bar. For hist[tp], the 'left index' is previous (previous to tp) item in stack and 'right index' is 'i' (current index).
3. If the stack is not empty, then one by one remove all bars from stack and do step 2.b for every removed bar.

Since every bar is pushed and popped only once, the time complexity is $O(n)$.

```

int getMaxArea (int hist[], int n) {
    // Stack holds indexes of hist[] array. The bars stored in
    // stack are always in increasing order of their heights
    stack<int> s;
    int max_area = 0;    // Initialize max area
    int top;              // To store top of the stack
    int area_with_top = 0; // To store area with top bar as smallest bar
    int i = 0;
    while (i < n) {
        // If this bar is higher than the bar on top, push it in stack
        if (s.empty() || hist[s.top()] <= hist[i])
            s.push(i++);
        // If the bar is lower than top of stack, then calculate area
        // of rectangle with stack top as bar with minimum height
        else {
            top = s.top();
            s.pop();
            area_with_top = hist[top]*(s.empty() ? i : (i-s.top()-1));
            if (max_area < area_with_top)
                max_area = area_with_top;
        }
    }
    // Now pop the remaining bars from stack and calculate
    // area with every popped bar as the smallest bar
    while(!s.empty()) {
        top = s.top(); s.pop();
        area_with_top = hist[top]*(s.empty() ? i : (i-s.top()-1));
        if (max_area < area_with_top)
            max_area = area_with_top;
    }
    return max_area;
}

```

Nearest Greater Element:

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exists, consider next greater element as -1.

1. Push the first element to stack.
2. Pick rest of the elements one by one and follow the following steps in loop.
 - a) Mark the current element as next.
 - b) If stack is not empty, compare top element of stack with next.
 - c) If next is greater than the top element, pop element from stack. 'next' is the next greater element for the popped element.
 - d) Keep popping from the stack while the popped element is smaller than 'next'. 'next' becomes the next greater element for all such popped elements
3. Finally, push the next in the stack.
4. After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

```
void printNGE(int arr[], int n) {  
    stack <int> s;  
    s.push(arr[0]);  
    for (int i = 1; i < n; i++) {  
        if (s.empty()) {  
            s.push(arr[i]);  
            continue;  
        }  
        while (s.empty() == false && s.top() < arr[i]) {  
            printf("%3d %3d\n", s.top(), arr[i]);  
            s.pop();  
        }  
        s.push(arr[i]);  
    }  
    while (!s.empty()) {  
        printf("%3d %3d\n", s.top(), -1);  
        s.pop();  
    }  
}
```

In the output of above code, you will see that the elements printed in output are not in same order as that of input.

So, solve the previous question - but the next greater elements should be printed in same order as input array.

Here we traverse array from the rightmost element.

1. In this approach we have started iterating from the last element(nth) to the first(1st) element. The benefit is that when we arrive at a certain index his next greater element will be already in stack and we can directly get this element while at the same index.
2. After reaching a certain index we will pop the stack till we get the greater element on top from the current element and that element will be the answer for current element.
3. If stack gets empty while doing the pop operation then the answer would be -1. Then we will store the answer in an array for the current index.

```
void printNGE(int arr[], int n) {  
    stack<int> s;  
    int result[n];  
    for (int i = n - 1; i >= 0; i--) {  
        while (!s.empty() && s.top() < arr[i])  
            s.pop();  
        if (s.empty())  
            result[i] = -1;  
        else  
            result[i] = s.top();  
        s.push(arr[i]);  
    }  
    cout << "Original Array: \n";  
    for (int i = 0; i < n; i++)  
        cout << arr[i] << " ";  
    cout << endl;  
  
    cout << "NGE: \n";  
    for (int i = 0; i < n; i++)  
        cout << result[i] << " ";  
    cout << endl;  
}
```

Next Greater Frequency Element:

Given an array, for each element find the value of the nearest element to the right which is having frequency greater than as that of current element. If there doesn't exist an answer for a position, make value -1.

Example:

Input: {1, 1, 2, 3, 4, 2, 1}
Output: {-1, -1, 1, 2, 2, 1, -1}

Explanation:

Given array is {1, 1, 2, 3, 4, 2, 1}.

Now frequency of each element in array is {3, 3, 2, 1, 1, 2, 3}.

For element $a[0] = 1$, frequency = 3. There is no element to the right of 1, whose frequency is greater than 3. So $\text{output}[0] = -1$.

For element $a[1] = 1$, $\text{output}[1] = -1$ by same logic.

For element $a[2] = 2$, frequency = 2. There is element '1' in array, at position 6, whose frequency is 3 and is more than 2. So $\text{output}[2] = 1$.

For element $a[3] = 3$, frequency = 1. The nearest next higher frequency element for 3 is '2' whose frequency is 2. So $\text{output}[3] = 2$.

We can go on for other elements now.

Approach:

1. Create an array/map to store frequency of each element.
2. Push the position of first element to stack.
3. Push rest of the positions of elements one by one and follow following steps in loop:
 - a) Mark the position of current element as ' i '
 - b) If
 - the frequency of the element which is pointed by the top of stack is GREATER than frequency of the current element,
push the current position i to stack
 - Else
 - If
 - the frequency of the element which is pointed by the top of stack is LESS than frequency of the current element,
and the stack is not empty,
while (above condition fails) {
 - output[stack's top position] = frequency of this i^{th} element
 - pop the element from stack
- Then push the current position i to stack.
4. After the loop in step 3 is over, the positions left in stack are those positions whose NGF element was not found.

Time Complexity: $O(n)$ and Space Complexity: $O(n)$

```

void nextGreaterFreq (int arr[], int n) {
    // Creating a hash Table which stores frequency of each element
    map <int, int> freq;
    for (int i = 0; i < n; i++)
        freq[arr[i]]++;

    // Stack to store position of array elements
    stack <int> s;
    s.push(0);

    // result array to store NGF element for each element
    int result[n] = {0};

    for (int i = 1; i < n; i++) {
        /* If the frequency of the element, which is at the position
         at the top of the stack, is greater than frequency of current
         element, then push the current position i in stack */
        if (freq[arr[s.top()]] > freq[arr[i]])
            s.push(i);
        else {
            /* If the frequency of the element, which is at the position
             at the top of the stack, is less than frequency of current
             element, then pop the stack and continue popping until
             the above condition is true while the stack is not empty */
            while (freq[arr[s.top()]] < freq[arr[i]] && !s.empty()) {
                result[s.top()] = arr[i];
                s.pop();
            }
            // Now push the current element
            s.push(i);
        }
    }

    while (!s.empty()) {
        result[s.top()] = -1;
        s.pop();
    }

    for (int i = 0; i < n; i++)
        cout << result[i] << " ";
    cout << endl;
}

```

Chapter 6 – Queues

6.1 Implementation

In C Lang:

```
#include <stdio.h>
#include <stdlib.h>
#include <climits>
#include <malloc.h>

struct Queue {
    int* array;
    int front, rear, size;
    unsigned capacity;
};

struct Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = -1;
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}

int isFull(struct Queue* queue) {
    return queue->size == queue->capacity;
}

int isEmpty(struct Queue* queue) {
    return queue->size == 0;
}

void enqueue(struct Queue* queue, int item) {
    if (isFull(queue)) {
        printf ("Overflow\n");
        return;
    }
    else {
        queue->rear = (queue->rear + 1)%queue->capacity;
        queue->array[queue->rear] = item;
        queue->size = queue->size+1;
        printf("%d enqueued to queue\n", item);
    }
}
```

```

int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf ("Underflow.\n"); return INT_MIN;
    }
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

int front (struct Queue* queue) {
    if (isEmpty(queue)) return INT_MIN;
    else return queue->array[queue->front];
}

int rear (struct Queue* queue) {
    if (isEmpty(queue)) return INT_MIN;
    else return queue->array[queue->rear];
}

int main() {
    struct Queue* queue = createQueue (100); int option, val;
    do {
        printf ("\n1.ENQUEUE\n2.DEQUEUE\n3.FRONT\n4.REAR\n5.EXIT\n");
        scanf ("%d", &option);
        switch (option) {
            case 1:
                printf ("Enter the number to be enqueued: ");
                scanf ("%d", &val); enqueue (queue, val);
                break;
            case 2:
                printf ("%d is dequeued.\n", dequeue(queue));
                break;
            case 3:
                printf ("%d is front element.\n", front(queue));
                break;
            case 4:
                printf ("%d is rear element.\n", rear(queue));
                break;
            case 5: exit (0);
            default: printf ("Invalid option.\n");
        }
    }
    while (option != 5);
    return 0;
}

```

In C++ Lang:

```
#include <iostream>
#include <stdlib.h>
#include <malloc.h>
#include <climits>
using namespace std;

class Queue {
private:
    int Front, Rear;
public:
    int capacity, queue_size, *array;

    Queue() {
        queue_size = 0; Front = 0; Rear = -1;
    }
    void size(int x);
    bool isEmpty();
    bool isFull();
    void enqueue (int item);
    int dequeue();
    int front();
    int rear();
};

void Queue :: size(int x) {
    capacity = x; array = (int*)malloc(x*sizeof(int));
}

bool Queue :: isEmpty() {
    return (queue_size == 0);
}

bool Queue :: isFull() {
    return (queue_size == capacity);
}

void Queue :: enqueue (int item) {
    array[++Rear] = item; ++queue_size;
}

int Queue :: dequeue() {
    int item = array[Front]; Front = (Front + 1)%capacity;
    --queue_size;
    return item;
}
```

```

int Queue :: front() {
    return array[Front];
}

int Queue :: rear() {
    return array[Rear];
}

int main() {
    Queue queue; int max, option, val;
    cout << "Enter the size of the queue: ";
    cin >> max;
    queue.size(max);
    do {
        printf ("\n1.Enqueue\n2.Dequeue\n3.Front\n4.Rear\n5.Exit\n");
        cin >> option;
        switch (option) {
            case 1:
                if (queue.isFull())
                    cout << "Overflow" << endl;
                else {
                    cout << "Enter the value to be enqueue: ";
                    cin >> val;
                    queue.enqueue(val);
                }
                break;
            case 2:
                if (queue.isEmpty())
                    cout << "Underflow" << endl;
                else
                    cout << queue.dequeue() << " dequeued from queue.";
                break;
            case 3:
                cout << queue.front() << " is the front element.";
                break;
            case 4:
                cout << queue.rear() << " is the rear element.";
                break;
            case 5: exit (0);
            default: printf ("Invalid option.\n");
        }
    }
    while (option != 5);
    return 0;
}

```

In C++ STL:

```
#include <iostream>
#include <queue>
#include <stdlib.h>
using namespace std;

int main() {
    queue<int> s;
    int option, val, front, rear, dequeued, queue_size;
    do {
        cout<<"\n1.ENQUEUE\n2.DEQUEUE\n3.FRONT\n4.REAR\n5.SIZE\n6.EXIT\n";
        cin >> option;
        switch (option) {
            case 1:
                cout << "Enter the value to be enqueued: ";
                cin >> val; s.push(val);
                break;
            case 2:
                if (s.empty())
                    cout << "Underflow" << endl;
                else {
                    dequeued = s.front(); s.pop();
                    cout << dequeued << " dequeued from queue.\n";
                }
                break;
            case 3:
                front = s.front();
                cout << front << " is the front element." << endl;
                break;
            case 4:
                rear = s.back();
                cout << rear << " is the last element in queue" << endl;
                break;
            case 5:
                queue_size = s.size();
                cout << "Size of queue is " << queue_size << endl;
                break;
            case 6:
                exit (0);
            default:
                cout << "Invalid option.\n";
        }
    } while (option != 6);
    return 0;
}
```

6.2 Design Problems

Design a stack using queues.

To do so – we require two queues, say q1 and q2.

1. push(x):
 - a) Enqueue x to q2
 - b) One by one dequeue everything from q1 and enqueue to q2
 - c) Swap the names q1 and q2
2. pop():
 - a) Dequeue an item from q1 and return it

Of course, time complexity of push and pop operations is no more $O(1)$. The question is just to test whether one can use the existing data structure as another in a tricky way.

```
#include<bits/stdc++.h>
using namespace std;

class Stack {
    queue<int> q1, q2; int curr_size;
public:
    Stack() {
        curr_size = 0;
    }
    void push(int x) {
        curr_size++; q2.push(x);
        while (!q1.empty()) {
            q2.push(q1.front()); q1.pop();
        }
        queue<int> q = q1; q1 = q2; q2 = q;
    }
    void pop(){
        if (q1.empty()) return;
        q1.pop(); curr_size--;
    }
    int top(){
        if (q1.empty()) return -1;
        return q1.front();
    }
    int size(){
        return curr_size;
    }
};
```

```

int main() {
    Stack s;
    s.push(1); s.push(2); s.push(3);
    cout << "Current size: " << s.size() << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << "Current size: " << s.size() << endl;
    return 0;
}

```

Design a queue using stacks.

Again, to do so, we require two stacks – say stack1 and stack2. We have to make sure that the oldest entered element is always at the top of stack 1, so that dequeue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enqueue(q, x):

- a) While stack1 is not empty, push everything from stack1 to stack2.
- b) Push x to stack1 (assuming size of stacks is unlimited).
- c) Push everything back to stack1.

Here time complexity will be O(n)

dequeue(q):

- a) If stack1 is empty then error
- b) Pop an item from stack1 and return it

Here time complexity will be O(1)

```

#include <bits/stdc++.h>
using namespace std;

class Queue {
    stack <int> s1, s2;
public:
    void enqueue (int x);
    void dequeue ();
    int front();
};

void Queue :: enqueue(int x) {
    while (!s1.empty()) {
        s2.push(s1.top()); s1.pop();
    }
}

```

```

    s1.push(x);
    while (!s2.empty()) {
        s1.push(s2.top()); s2.pop();
    }
}

void Queue::dequeue() {
    if (s1.empty()) {
        cout << "Queue is empty.\n"; return;
    }
    s1.pop();
}

int Queue::front() {
    return s1.top();
}

int main() {
    Queue q;
    q.enqueue(1); q.enqueue(2); q.enqueue(3);
    cout << q.front() << endl;
    q.dequeue(); q.dequeue();
    cout << q.front() << endl;
    return 0;
}

```

6.3 Operations on queues

Reverse a queue:

Reversing a queue is easy.

- Dequeue all elements from queue and push them onto stack.
- Now pop all elements from stack and enqueue them in a queue.

```

void reverseItr (queue<int> &q) {
    stack<int> s;
    while (!q.empty()) {
        s.push(q.front());
        q.pop();
    }
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }
}

```

```

void reverseRec (queue<int> &q) {
    if (q.empty()) return;
    int data = q.front();
    q.pop();
    reverseRec(q);
    q.push(data);
}

```

Sort a queue:

Following is the idea to sort a queue with constant space:

- On every pass on the queue, we seek for the next minimum index. To do this we dequeue and enqueue elements until we find the next minimum. In this operation the queue is not changed at all. After we have found the minimum index, we dequeue and enqueue elements from the queue except for the minimum index, after we finish the traversal in the queue, we insert the minimum to the rear of the queue. We keep on this until all minimums are pushed all the way long to the front and the queue becomes sorted.
- On every next seeking for the minimum, we exclude seeking on the minimums that have already sorted.
- We repeat this method n times.
- At first, we seek for the maximum, because on every pass we need find the next minimum, so we need to compare it with the largest element in the queue.

Time complexity: $O(n^2)$ and Space complexity: $O(1)$

```

/* Utility function: Queue elements after sortedIndex
are already sorted. This function returns index of
min element from front to sortedIndex. */
int minIndex (queue<int> &q, int sortedIndex) {
    int min_index = -1, min_val = INT_MAX, n = q.size();
    for (int i = 0; i < n; i++) {
        int curr = q.front();
        q.pop();
        if (curr <= min_val && i <= sortedIndex) {
            min_index = i;
            min_val = curr;
        }
        q.push(curr);
    }
    return min_index;
}

```

```

/* Utility Function: This function moves the given
   minimum element to rear of queue. */
void insertMinToRear (queue<int> &q, int min_index) {
    int min_val;
    int n = q.size();
    for (int i = 0; i < n; i++) {
        int curr = q.front(); q.pop();
        if (i != min_index) q.push(curr);
        else min_val = curr;
    }
    q.push(min_val);
}

void sortQueue(queue<int> &q) {
    for (int i = 1; i <= q.size(); i++) {
        int min_index = minIndex(q, q.size() - i);
        insertMinToRear(q, min_index);
    }
}

```

Interleaving the halves:

Given a queue of integers of even length, rearrange the elements by interleaving the first half of the queue with the second half of the queue. Only stack data structure is allowed to be used as auxiliary space.

Input: 11 12 13 14 15 16 17 18 19 20

Output: 11 16 12 17 13 18 14 19 15 20

1. Push the first half elements of queue to stack.
2. Enqueue back the stack elements.
3. Dequeue the first half elements of the queue and enqueue them back.
4. Again push the first half elements into the stack.
5. Interleave the elements of queue and stack.

Time and Space complexity: $O(n)$

```

void interleave (queue<int> &q) {
    stack <int> s;
    int halfsize = q.size()/2;

    // Push first half elements onto stack
    for (int i = 0; i < halfsize; i++){
        s.push(q.front()); q.pop();
    }

```

```

// Enqueue back the stack elements
while (!s.empty()) {
    q.push(s.top()); s.pop();
}
// Dequeue the first half elements of stack and enqueue them back
for (int i = 0; i < halfsize; i++) {
    q.push(q.front()); q.pop();
}
// Again push the first half elements into the stack
for (int i = 0; i < halfsize; i++) {
    s.push(q.front()); q.pop();
}
// Interleave the elements of queue and stack
while (!s.empty()) {
    q.push(s.top()); s.pop();
    q.push(q.front()); q.pop();
}
}

```

6.4 Application of Queues

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements. For example: If the input array is {8, 1, 7, 6, 0}, output should be “8 7 6 0”.

1. Sort the array in non-decreasing order.
2. Take three queues. One for storing elements which on dividing by 3 gives remainder as 0. The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2.
3. Find the sum of all the digits.
4. Three cases arise:
 - 4.1 The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.
 - 4.2 The sum of digits produces remainder 1 when divided by 3. Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.
 - 4.3 The sum of digits produces remainder 2 when divided by 3. Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.
5. Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.

```

// Comparison function to sort numbers in lexicographic order
bool myCompare (int a, int b) {
    string temp1 = to_string(a).append(to_string(b));
    string temp2 = to_string(b).append(to_string(a));
    return (temp1 > temp2);
}

// Utility function: This function puts all the
// elements of 3 queues in the auxiliary array
void addInAuxArray (int aux[], queue<int> q0, queue<int> q1, queue<int>
q2, int* top) {
    while (!q0.empty()) {
        aux[(*top)++] = q0.front(); q0.pop();
    }
    while(!q1.empty()) {
        aux[(*top)++] = q1.front(); q1.pop();
    }
    while(!q2.empty()) {
        aux[(*top)++] = q2.front(); q2.pop();
    }
}

void findMaxMultipleOf3 (int arr[], int size) {
    // flag will be turned false when there is no
    // number which could be formed that is multiple of 3
    bool flag = true;
    // 1. Sort the array in non-decreasing order
    sort (arr, arr+size);
    // 2. and 3. Get the sum of numbers and place
    // them into proper queues
    queue<int> queue0, queue1, queue2;
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += arr[i];
        if ((arr[i]%3) == 0) queue0.push(arr[i]);
        else if ((arr[i] % 3) == 1) queue1.push(arr[i]);
        else queue2.push(arr[i]);
    }
    // 4.2 The sum produces remainder 1
    if ((sum % 3) == 1) {
        // either remove one item from queue1
        if (!queue1.empty()) queue1.pop();
        // or remove two items from queue2
        else {
            if (!queue2.empty()) queue2.pop();
            else flag = false;
        }
    }
}

```

```

        if (!queue2.empty()) queue2.pop();
        else flag = false;
    }
}

// 4.3 The sum produces remainder 2
else if ((sum % 3) == 2) {
    // either remove one item from queue2
    if (!queue2.empty()) queue2.pop();
    // or remove two items from queue1
    else {
        if (!queue1.empty()) queue1.pop();
        else flag = false;
        if (!queue1.empty()) queue1.pop();
        else flag = false;
    }
}
int aux[size], top = 0;
addInAuxArray(aux, queue0, queue1, queue2, &top);
sort (aux, aux+top, myCompare);
if (!flag)
    cout << "Not possible to form any number.\n";
else {
    for (int j = 0; j < top; j++)
        cout << aux[j] << " ";
    cout << endl;
}
}

int main() {
    int arr[] = { 8, 81, 7, 61, 0 };
    int size = sizeof(arr) / sizeof(arr[0]);
    findMaxMultipleOf3 (arr, size);
    return 0;
}

```

Change to Everyone:

Given an array of N integers where A_i denotes the currency of note that the i -th person has. The possible currencies are 5, 10 and 20. All the N people are standing in a queue waiting to buy an ice-cream from X which costs Rs 5. Initially, X has an initial balance of 0. Check if X will be able to provide change to every people who are waiting to buy an ice-cream.

Input: $a[] = \{5, 5, 5, 10, 20\}$

Output: YES

- ⇒ When the fourth person chance comes to buy an ice-cream, X has three Rs 5 change, hence X gives him 1, and now when the fifth person comes to buy the ice-cream, X has two Rs 5 and one Rs 10 note, hence he gives him one Rs 10 and one Rs 5 note.

Input: $a[] = \{5, 10, 10, 20\}$

Output: NO

The approach is to keep a track of the number of Rs 5 and Rs 10 currencies. Rs 20 currencies will not be used since it is the highest currency that a person can give and thus it cannot be given as a change. Initialize two variables to count Rs 5(fiveCount) and Rs 10(tenCount). If the person has a Rs 10 currency and fiveCount > 0, then decrease fiveCount and increase tenCount. If X does not have Rs 5, then X cannot give the person the required change. If the person has 5\$ note, increase fiveCount by one. If the person has a Rs 20, then three conditions will be:

- If fiveCount > 0 and tencount > 0, decrease both.
- else if, fiveCount ≥ 3 , decrease fivecount by three.
- else, return false.

If all the person in the queue gets the change, then print “YES” else print “NO”.

```
int isChangeable(int notes[], int n) {
    int fiveCount = 0, tenCount = 0;
    for (int i = 0; i < n; i++) {
        if (notes[i] == 5)
            fiveCount++;
        else if (notes[i] == 10) {
            if (fiveCount > 0) {
                fiveCount--; tenCount++;
            }
            else
                return 0;
        }
        else {
            if (fiveCount > 0 && tenCount > 0) {
                fiveCount--; tenCount--;
            }
            else if (fiveCount >= 3) {
                fiveCount -= 3;
            }
            else
                return 0;
        }
    }
    return 1;
}
```

Generating binary numbers:

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

Input: n = 2

Output: 1, 10

Input: n = 5

Output: 1, 10, 11, 100, 101

A simple method is to run a loop from 1 to n, and convert decimal to binary inside the loop.

Following is an interesting method that uses queue data structure to print binary numbers.

- 1) Create an empty queue of strings
- 2) Enqueue the first binary number “1” to queue.
- 3) Now run a loop for generating and printing n binary numbers.
 - a) Dequeue and Print the front of queue.
 - b) Append “0” at the end of front item and enqueue it.
 - c) Append “1” at the end of front item and enqueue it.

```
void generatePrintBinary(int n) {  
    queue<string> q;  
    q.push("1");  
  
    // This loops is like BFS of a tree with 1 as root  
    // 0 as left child and 1 as right child and so on  
    while (n--) {  
        // print the front of queue  
        string s1 = q.front(); q.pop();  
        cout << s1 << "\n";  
        string s2 = s1; // Store s1 before changing it  
        // Append "0" to s1 and enqueue it  
        q.push(s1.append("0"));  
        // Append "1" to s2 and enqueue it. Note that s2 contains  
        // the previous front  
        q.push(s2.append("1"));  
    }  
}
```

Generate divisible number with 9s and 0s:

We are given an integer N. We need to write a program to find the least positive integer X made up of only digits 9's and 0's, such that, X is a multiple of N.

Note: It is assumed that the value of X will not exceed 10^5 .

Examples:

Input : N = 5

Output : X = 90

Explanation: 90 is the smallest number made up of 9's and 0's, divisible by 5.

Input : N = 7

Output : X = 9009

Explanation: 9009 is the smallest number made up of 9's and 0's, divisible by 7.

The idea to solve this problem is to generate and store all of the numbers which can be formed using digits 0 & 9. Then find the smallest number among these generated number which is divisible by N.

```
// Maximum number of numbers made of 0 and 9
#define MAX_COUNT 10000
// vector to store all numbers that can be formed
// using digits 0 and 9 and are less than  $10^5$ 
vector<string> vec;
// Preprocessing function to generate all possible numbers formed by
// 0 and 9
void generateNumbersUtil() {
    queue<string> q; q.push("9");
    // This loops is like BFS of a tree with 9 as root
    // 0 as left child and 9 as right child and so on
    for (int count = MAX_COUNT; count > 0; count--) {
        string s1 = q.front(); q.pop();
        // storing the front of queue in the vector
        vec.push_back(s1);
        string s2 = s1;
        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));
        // Append "9" to s2 and enqueue it. Note that
        // s2 contains the previous front
        q.push(s2.append("9"));
    }
}
```

```
string findSmallestMultiple(int n) {
    // traverse the vector to find the smallest multiple of n
    for (int i = 0; i < vec.size(); i++)
        // stoi() is used for string to int conversion
        if (stoi(vec[i])%n == 0)
            return vec[i];
}
```

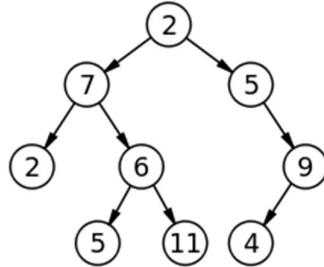
Chapter 7 – Binary Trees

7.1 Traversals in BT

We extend the concept of linked data structures to structure containing nodes with more than one self-referenced field. A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

A typical binary tree structure looks like this:



More tree terminology:

- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.

Here is the structure of node in a tree:

```
class Node {  
public:  
    int data;  
    Node* left;  
    Node* right;  
    Node(int data) {  
        this->data = data;  
        this->left = this->right = NULL;  
    }  
};
```

This is how we create a basic tree:

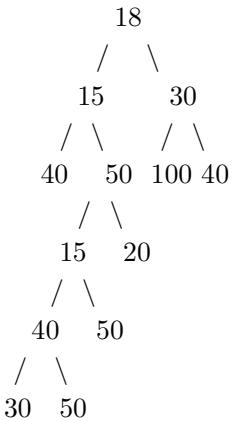
```
int main() {
    // Create Root Node
    Node* root = new Node(1);
    // Attach left and right children
    root->left = new Node(2);
    root->right = new Node(3);

    /*
        1
       /   \
      2     3
     / \   / \
    NULL NULL NULL NULL
    */
    return 0;
}
```

Types of BT:

1. FULL/STRICT Binary Tree

A Binary Tree is full if every node has 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children. Following is the example of a full binary tree.



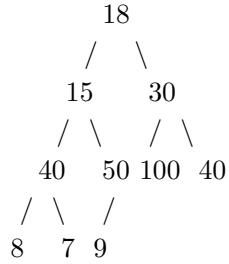
In a Full Binary Tree, number of leaf nodes is number of internal nodes plus 1.

$$L = N + 1$$

where L = Number of leaf nodes, N = Number of internal nodes.

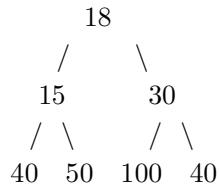
2. COMPLETE Binary Tree

A Binary Tree is a complete binary tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.



3. PERFECT Binary Tree

A Binary Tree is a perfect binary tree if all internal nodes have two children and all leaves are at the same level.



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ nodes.

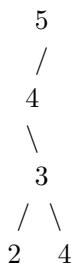
4. SKEWED Binary Tree

A skewed tree is a tree where each node has only one child node or none.

Examples:



This is a skewed tree.



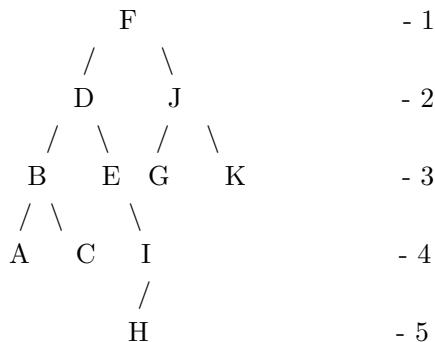
This is not a skewed tree.

Traversals:

Following is the Classification of various types of Tree Traversals:

1. BREADTH FIRST TRAVERSAL
 - Level Order Traversal
2. DEPTH FIRST TRAVERSAL
 - Pre Order Traversal
 - In Order Traversal
 - Post Order Traversal

Consider the following tree:



A) Level order traversal:

In level order traversal, we need to print -

F D J B E G K A C I H

Traversal:

The Level order traversal is bit tricky. Let's see why? Let's say, we print our first node - F.

Now, we may go to left child, and print D. The problem is now I have to print J, and I can't go backwards to F as D is not storing any previous/parent pointer.

We will overcome this problem and see its implementation in the upcoming codes. For now, we should just learn, how to get the expected outcome of certain order traversal of a tree. And predicting the outcome of level order traversal is the easiest as we need to cover nodes from left to right at each level.

B) Pre-order Traversal:

In this traversal, Remember the order - <root> <left> <right>

Traversal:

- 1) We will start with root F. Then we will move to left subtree. We will now complete travelling the entire left subtree whose root is D. Once left subtree is done, we will move to right subtree.
- 2) Once we are at D, we can say for the left subtree, D is the root. We will again move towards left. Hence, the next node which we will move to, after D, is B.
- 3) Again, when we are at B, it is the root of the subtree with nodes (B-A-C). So, we will move left towards A. Once we are at A, we see there is no left subtree.
- 4) So now we return to our node B. We know, if B is the root and we have travelled left subtree, now we need to go to the right subtree.
So, after A, we go to C.
- 5) We are done with left subtree of D, we go to E. Now we are done with left subtree of F. So, we may go to J.
- 6) When we are at J, there is left subtree which is unvisited. So, we go to G. We say there isn't any left node in case of G. So, we have returned to G and we go to I. For I, we visit left first viz. H. Then we return to I, then return to G, then finally return to J.
- 7) Finally, we go to the right of Node J i.e. K.

So, the order is: F D B A C E J G I H K

The implementation of this traversal will be a simple Recursion problem as we can see when we move to a subtree, we have just moved to a smaller problem of same kind.

C) In-order traversal:

In this traversal, the order is - <left> <root> <right>

Traversal:

- 1) In this traversal, we will first visit the left subtree and then visit the current node. We will then go right. So, we will start from F, we will print F later, first we will visit left subtree. So, we move to D. With same reasoning, we move to B and then to A.
- 2) For A, there is nothing in left. So, we can now read this as root. We should now move to the right of A. But there is nothing in the right of A, so we are done with A and we will return to B.
- 3) We will now print B, and then we can move to the right of B. For C, there isn't anything in left or right, so we are done with C.
- 4) Left of D is now done. So, we will read D. Then move to right, E. Now return from E to D. Now left of F is done. Now we print F and go to right to J. We will move to left to G. There is nothing in the left of G, so print G and move to I.
- 5) Continuing in this way, we will get the complete inorder traversal.

So, the order is: A B C D E F G H I J K

If you observe here, we got our alphabets in sorted order. This is because, the Binary Tree which we have chosen above is a special binary tree called "Binary Search Tree (BST)".

In BST, the value of the nodes is in the relation:

left node < root node < right node.

The inorder traversal of BST gives a sorted list of tree elements.

D) Post-order traversal:

In this traversal, the order is - <left> <right> <root>

Traversal:

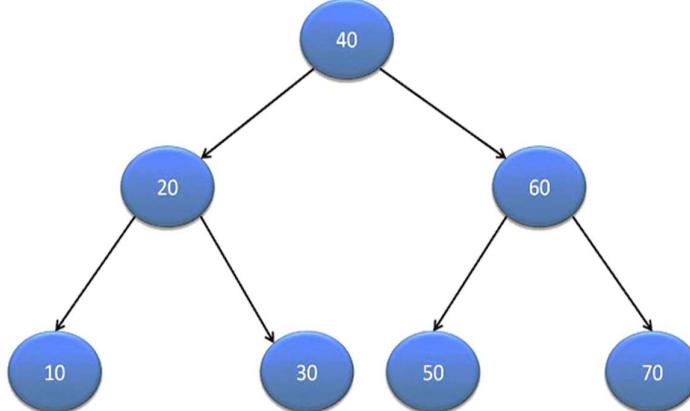
We will move to A and print A. Then return to B, move to right and print C. Then return to B and print B. By now, you should be able to guess the remaining post order traversal of the tree.

So, the order is: A C B E D H I G K J F

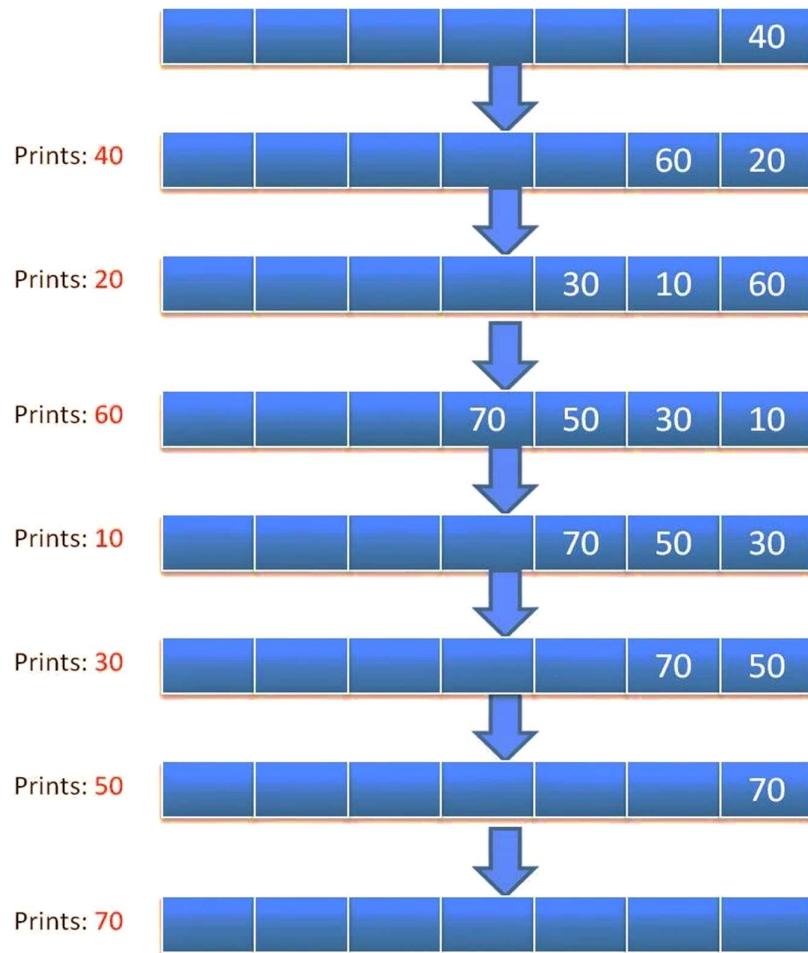
We shall now look at the implementations of these traversals.

Level order traversal:

For maintaining and accessing left as well as right children, we use a queue. Let's say your binary tree is:



The level order traversal would work as below:



```

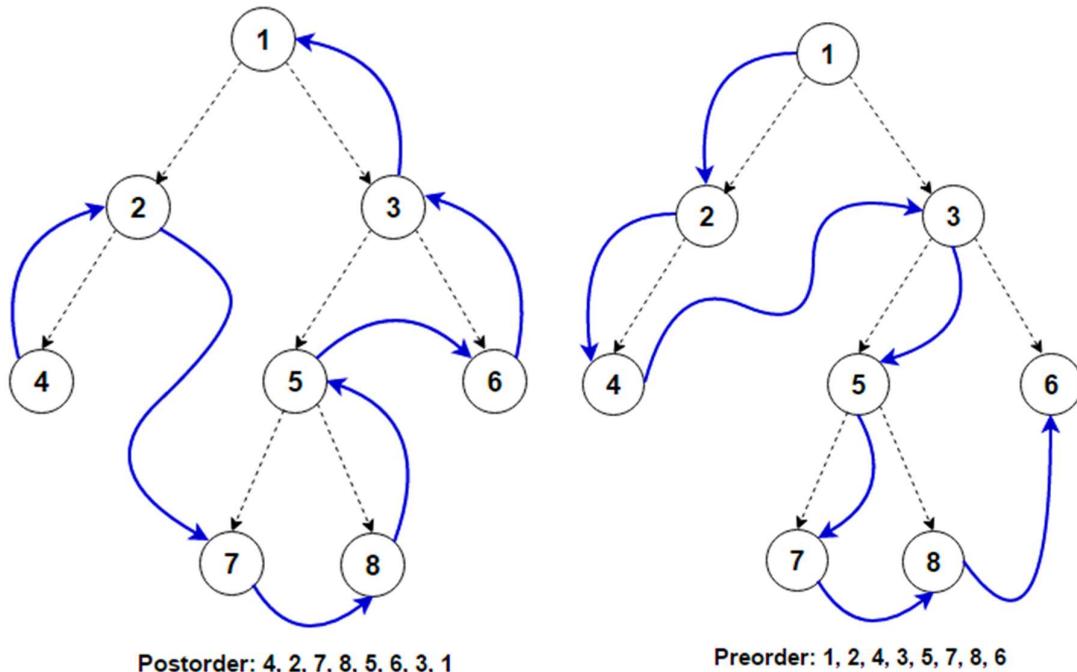
void levelOrder (Node* root) {
    // If the tree is empty, we can simply return
    if (root == NULL)    return;
    // If tree is not empty, we will need Queue
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* current = q.front();
        cout << current->data << " ";
        q.pop();
        if (current->left != NULL)
            q.push(current->left);
        if (current->right != NULL)
            q.push(current->right);
    }
}

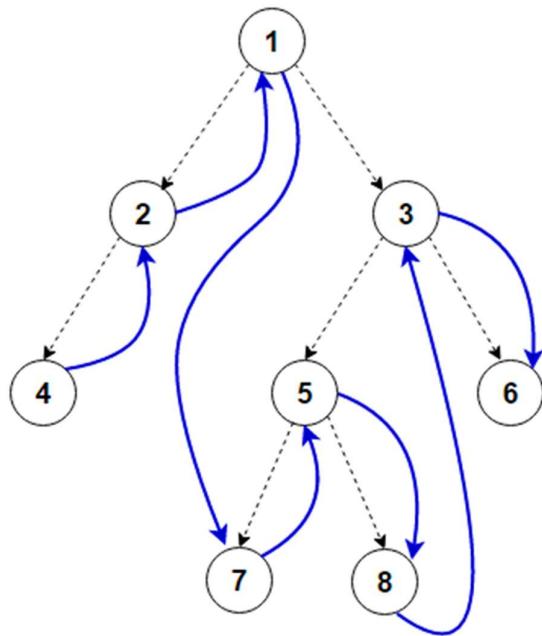
```

Recursive Pre-In-Post order traversals:

As tree is not a linear data structure i.e. from a given node, there can be multiple possible next nodes, so some nodes must be stored in some way for later visiting.

The traversals can be done in iterative manner by storing the nodes in stack or in recursive manner where the nodes are implicitly stored in the call stack. Following images show the order of the elements in which they are traversed.





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

```

void preOrder (Node* node) {
    if (node == NULL) return;
    cout << node->data << " ";
    preOrder (node->left);
    preOrder (node->right);
}

void inOrder (Node* node) {
    if (node == NULL) return;
    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}

void postOrder (Node* node) {
    if (node == NULL)    return;
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}

```

Iterative Preorder traversal:

To convert an inherently recursive procedures to iterative, we need an explicit stack. Following is a simple stack based iterative process to print Preorder traversal.

- 1) Create an empty stack s and push root node to stack.
- 2) Do following while s is not empty.
 - a) Pop an item from stack and print it.
 - b) Push right child of popped item to stack
 - c) Push left child of popped item to stack

Right child is pushed before left child to make sure that left subtree is processed first.

```
void preOrder (Node* root) {  
    if (root == NULL)    return;  
    stack <Node*> s; s.push(root);  
    while (!s.empty()) {  
        Node* current = s.top();  
        cout << current->data << " ";  
        s.pop();  
        if (current->right) s.push(current->right);  
        if (current->left) s.push(current->left);  
    }  
}
```

Iterative Inorder traversal:

- 1) Initialize current node as root
- 2) Create an empty stack S.
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
 - a) Pop the top item from stack.
 - b) Print the popped item, set current = popped_item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

```
void inOrder (Node* root) {  
    Node* current = root;  
    stack <Node*> s;  
    while (true) {  
        if (current != NULL) {  
            s.push(current); current = current->left;  
        }  
        else {  
            if (s.empty()) break;  
            cout << s.top()->data << " ";  
            current = s.top()->right; s.pop();  
        }  
    }  
}
```

Iterative Postorder traversal:

a) Method 1 - Using Two Stacks

1. Push root to auxiliary stack.
2. Loop while auxiliary stack is not empty
 - a) Pop a node from auxiliary stack and push it to result stack
 - b) Push left and right children of the popped node to auxiliary stack
3. Print contents of result stack

b) Method 2 - Using One Stack

1. Create an empty stack
2. Do following while root is not NULL
 - a) Push root's right child and then root to stack.
 - b) Set root as root's left child.
3. Pop an item from stack and set it as root.
 - a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
 - b) Else print root's data and set root as NULL.
4. Repeat steps 2 and 3 while stack is not empty.

Note:

One stack solution is no better than 2 stacks solution in terms of space complexity.

If you look at 2 stack solution, every time you pop from one stack you push onto another stack, so essentially you are not maintaining more than one copy of the same element across the stacks. It is just a different way to think the solution.

```
void postOrderTwo(Node* root) {  
    if (root == NULL)    return;  
    stack <Node*> st, result;  
    st.push(root);  
    while (!st.empty()) {  
        Node* current = st.top(); st.pop();  
        result.push(current);  
        if (current->left != NULL) st.push(current->left);  
        if (current->right != NULL) st.push(current->right);  
    }  
    while (!result.empty()) {  
        cout << result.top()->data << " ";  
        result.pop();  
    }  
}
```

```

void postOrderOne (Node* root) {
    Node *current = root;
    stack <Node*> st;
    while (true) {
        if (current != NULL) {
            st.push(current);
            current= current->left;
        }
        else {
            if (st.empty()) break;
            current = st.top()->right;
            if (current == NULL) {
                Node* last = NULL;
                while (!st.empty() && st.top()->right == last) {
                    last = st.top(); st.pop();
                    cout << last->data << " ";
                }
            }
        }
    }
}

```

How to decide which way of traversal is better? BFT or DFT?

1. There are many tree questions that can be solved using any of the above four traversals. Examples of such questions are size of tree, maximum element, minimum element, etc.
2. All four traversals require $O(n)$ time as they visit every node exactly once.

So where does the difference come?

1. There is difference in terms of extra space required.

Extra Space required for DFT is $O(w)$ where w is maximum width of BT. In BFT, queue one by one stores nodes of different level.

Extra Space required for DFT is $O(h)$ where h is maximum height of BT. In DFT, stack (or function call stack) stores all ancestors of a node.

Maximum Width of a BT at depth (or height) h can be 2^h where h starts from 0. So, the maximum number of nodes can be at the last level. And the worst case occurs when BT is a perfect BT with numbers of nodes like 1, 3, 7, 15, ..., etc. In the worst case, value of 2^h is $\text{Ceil}(n/2)$. Height for a Balanced BT is $O(\log n)$. The worst case occurs for skewed tree and worst-case height becomes $O(n)$.

So, in the worst case extra, space required is $O(n)$ for both. But worst cases occur for different types of trees.

It is evident from above points that extra space required for DFT is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.

2. Another point of difference is - BFS starts visiting nodes from root while DFS starts visiting nodes from leaves.

So, if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

Q. Which traversal should be used to print leaves of BT?

-> Depth First Traversal

Q. Which traversal should be used to print nodes at k'th level where k is much less than total number of levels?

-> Breadth First Traversal

Insertion of Node:

The idea is to do iterative level order traversal of the given tree using queue.

If we find a node whose left child is empty, we make new key as left child of the node. Else if we find a node whose right child is empty, we make new key as right child. We keep traversing the tree until we find a node whose either left or right is empty.

```
void insertNode (Node* root, int key) {  
    queue<Node*> q; q.push(root);  
    while (!q.empty()) {  
        Node* current = q.front(); q.pop();  
        if (!(current->left)) {  
            current->left = new Node(key);  
            break;  
        }  
        else  
            q.push(current->left);  
        if (!(current->right)) {  
            current->right = new Node(key);  
            break;  
        }  
        else  
            q.push(current->right);  
    }  
}
```

Deletion of Node:

1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.
2. Replace the deepest rightmost node's data with node to be deleted.
3. Then delete the deepest rightmost node.

```
void deleteDeepest (Node* root, Node* delete_node) {  
    queue <Node*> q; q.push(root);  
    while (!q.empty()) {  
        Node* current = q.front(); q.pop();  
        if (current == delete_node) {  
            current = NULL; delete (delete_node);  
            return;  
        }  
        if (current->right)  
            if (current->right == delete_node) {  
                current->right = NULL; delete (delete_node);  
                return;  
            }  
        else  
            q.push(current->right);  
        if (current->left)  
            if (current->left == delete_node) {  
                current->left = NULL; delete (delete_node);  
                return;  
            }  
        else  
            q.push(current->left);  
    }  
}  
  
void deletion (Node* root, int key) {  
    queue<struct Node*> q; q.push(root);  
    struct Node *current, *key_node = NULL;  
    // Do level order traversal to find deepest  
    // node(temp) and node to be deleted (key_node)  
    while (!q.empty()) {  
        current = q.front(); q.pop();  
        if (current->data == key) key_node = current;  
        if (current->left) q.push(current->left);  
        if (current->right) q.push(current->right);  
    }  
    int x = current->data;  
    deleteDeepest(root, current);  
    key_node->data = x;  
}
```

Sum of all nodes:

For a given BT, find sum of all nodes. This is simple BT traversal question.

```
int addRecursive (Node* root) {  
    if (root == NULL)  
        return 0;  
    return (root->data + addRecursive(root->left) +  
            addRecursive(root->right));  
}  
  
int addIterative (Node* root) {  
    int total = 0;  
    if (root == NULL)    return 0;  
    queue<Node*> q; q.push(root);  
    while (!q.empty()) {  
        Node* current = q.front();  
        total += current->data;  
        q.pop();  
        if (current->left) q.push(current->left);  
        if (current->right) q.push(current->right);  
    }  
    return total;  
}
```

Height of Binary Tree:

```
int heightRecursive(Node* root) {  
    if (root == NULL)    return 0;  
    return (max(heightRecursive(root->left),  
                heightRecursive(root->right)) + 1);  
}  
  
int heightIterative(Node* root) {  
    if (root == NULL)    return 0;  
    queue<Node*> q; q.push(root);  
    Node* front = NULL;  
    int height = 0;  
    while (!q.empty()) {  
        int size = q.size();  
        while(size--) {  
            front = q.front(); q.pop();  
            if (front->left)   q.push(front->left);  
            if (front->right)  q.push(front->right);  
        }  
        height++;  
    }  
    return height;  
}
```

Types of Binary Tree (code):

Full/Strict Binary Tree:

```
bool isFullRec (Node* root) {
    // If tree is empty
    if (root == NULL)    return true;
    // If we are at leaf node
    if (root->left == NULL && root->right == NULL)  return true;
    // If both - left and right subtrees exist
    if (root->left && root->right)
        return (isFullRec(root->left) && isFullRec(root->right));
    return false;
}

bool isFullItr (Node* root) {
    // If tree is empty
    if (root == NULL)    return true;
    queue<Node*> q; q.push(root);
    while (!q.empty()) {
        Node* node = q.front(); q.pop();
        // If we are at leaf node
        if (node->left == NULL && node->right == NULL)
            continue;
        // If either of the child is not null, tree is not full
        if (node->left == NULL || node->right == NULL)
            return false;
        q.push(node->left); q.push(node->right);
    }
    return true;
}
```

Complete Binary Tree:

```
int totalNodes(Node* root) {
    if (root == NULL)
        return 0;
    return (1 + totalNodes(root->left) + totalNodes(root->right));
}

bool isCompleteHelper(Node* root, int index, int nodes_count) {
    if (root == NULL)    return true;
    if (index >= nodes_count)  return false;
    return isCompleteHelper(root->left, 2*index+1, nodes_count) && \
           isCompleteHelper(root->right, 2*index+2, nodes_count);
}
```

```

bool isCompleteRec(Node* root) {
    return isCompleteHelper (root, 0, totalNodes(root));
}

bool isCompleteItr(Node* root) {
    if (root == NULL)    return true;
    // flag will be set to true when non-full node is seen
    bool flag = false;
    queue <Node*> q; q.push(root);
    while(!q.empty()) {
        Node* temp = q.front(); q.pop();
        // Check if left child is present
        if (temp->left) {
            if (flag)
                return false;
            q.push(temp->left);
        }
        else
            flag = true;
        if (temp->right) {
            if (flag)
                return false;
            q.push(temp->right);
        }
        else
            flag = true;
    }
    return true;
}

```

Perfect Binary Tree:

```

// Returns depth of leftmost leaf
int height(Node *root) {
    int d = 0;
    while(root != NULL) {
        d++;
        root = root->left;
    }
    return d;
}

```

```

bool isPerfectHelper(Node* root, int depth, int level) {
    if (root == NULL)    return true;
    // If we are at leaf node, it's depth must be same as
    // depth of all other leaves
    if (root->left == NULL && root->right == NULL)
        return (depth == level + 1);
    // If internal node and one child is empty
    if (root->left == NULL || root->right == NULL)
        return false;
    return isPerfectHelper(root->left, depth, level+1) && \
           isPerfectHelper(root->right, depth, level+1);
}

bool isPerfectRec (Node* root) {
    int d = height(root);
    return isPerfectHelper(root, d, 0);
}

bool isPerfectItr (Node* root) {
    if (root == NULL)    return true;
    // Once leaf node is found, flag will become true
    bool flag = false;
    queue <Node*> q; q.push(root);

    while(!q.empty()){
        Node* temp = q.front(); q.pop();
        // If current node has both children
        if (temp->left && temp->right) {
            // If leaf is already found, return false
            if (flag)
                return false;
            else {
                q.push(temp->left);
                q.push(temp->right);
            }
        }
        // If we are at leaf node, set flag to true
        if (!temp->left && !temp->right)
            flag = true;
        // If the current node has only one child, return false
        if (!temp->left || !temp->right)
            return false;
    }
    return true;
}

```

Skewed Binary Tree:

```
bool isSkewedRec (Node* root) {  
    // Check if node is NULL or leaf node  
    if (root == NULL || (root->left == NULL && root->right == NULL))  
        return true;  
    // Check if the node has two children  
    if (root->left && root->right)  
        return false;  
    if (root->left)  
        return isSkewedRec(root->left);  
    return isSkewedRec(root->right);  
}
```

Comparing two BTs:

Identical:

```
bool isIdentical (Node* a, Node* b) {  
    // Check if both trees are empty  
    if (a == NULL && b == NULL)  
        return true;  
    // If both are not empty and their data is equal,  
    // compare left and right  
    if (a!=NULL && b!=NULL && a->data == b->data)  
        return isIdentical(a->left, b->left) &&  
            isIdentical(a->right, b->right);  
    // If we reached here, means one child is empty  
    // in one tree and not in other  
    return false;  
}
```

Similar (having same structure irrespective of data):

```
bool isSimilar (Node* a, Node* b) {  
    // If both are empty  
    if (a == NULL && b == NULL)  
        return true;  
    // If both are not empty, compare left and right  
    if (a!=NULL && b!=NULL)  
        return isSimilar(a->left, b->left) &&  
            isSimilar(a->right, b->right);  
    // If we reached here, means one child is empty in  
    // one tree and not in other  
    return false;  
}
```

Foldable and Symmetric BT:

A tree can be folded if left and right subtrees of the tree are structure wise mirror-image of each other. An empty tree is considered as foldable.

```
// A utility function that checks if trees with roots as 'a' and 'b'  
// are structure-wise mirror of each other.  
bool isFoldableUtil(Node* a, Node* b) {  
    // If both, left and right subtrees are NULL, return true  
    if (a==NULL && b==NULL) return true;  
    // If one of the subtree is NULL and other is not, return false  
    if (a==NULL || b==NULL) return false;  
  
    // Otherwise check for left and right subtrees  
    return isFoldableUtil(a->left, b->right) &&  
           isFoldableUtil(a->right, b->left);  
}  
  
bool isFoldable(Node* root) {  
    if (root == NULL) return true;  
    return isFoldableUtil(root->left, root->right);  
}
```

A symmetric BT is a tree which is a mirror-image of itself.

```
bool isMirrorUtil(Node* a, Node* b){  
    if (a == NULL && b == NULL) return true;  
    if (a && b && a->data == b->data)  
        return isMirrorUtil(a->left, b->right) &&  
               isMirrorUtil(a->right, b->left);  
    return false;  
}  
  
bool isMirrorRec(Node* root) {  
    return isMirrorUtil(root, root);  
}  
  
bool isMirrorItr(Node* root) {  
    if (root == NULL) return true;  
    queue <Node*> q;  
    // Add root to queue two times, so that we can check if either  
    // one child alone is Null or not.  
    q.push(root); q.push(root);  
  
    Node *leftNode, *rightNode;
```

```

while (!q.empty()) {
    leftNode = q.front(); q.pop();
    rightNode = q.front(); q.pop();
    // If both - left and right children exists but have different
    // values, return false
    if (leftNode->data != rightNode->data)
        return false;
    // Push left child of left subtree node and right child of
    // right subtree node in queue
    if (leftNode->left && rightNode->right) {
        q.push(leftNode->left); q.push(rightNode->right);
    }
    // If one child is present and other is null, return false
    // as tree is not symmetric
    else if (leftNode->left || rightNode->right)
        return false;
    // Push right child of left subtree and left child
    // of right subtree
    if (leftNode->right && rightNode->left) {
        q.push(leftNode->right);
        q.push(rightNode->left);
    }
    // If only one child is present, tree is asymmetric
    else if (leftNode->right || rightNode->left)
        return false;
}
return true;
}

```

Problems

Q. 1 Given a binary tree, find the maximum element in it.

Solution:

```

int findMaxRec (Node* root) {
    // Base case
    if (root == NULL)    return INT_MIN;
    int res = root->data;
    int lres = findMaxRec(root->left);
    int rres = findMaxRec(root->right);
    return max(res, max(lres, rres));
}

```

```

int findMaxItr(Node* root) {
    if (root == NULL)    return INT_MIN;
    Node* temp = NULL;
    int max = INT_MIN;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        temp = q.front();
        q.pop();
        if (temp->data > max)
            max = temp->data;
        if (temp->left) q.push(temp->left);
        if (temp->right) q.push(temp->right);
    }
    return max;
}

```

Q. 2 Given a Binary Node and a node. The task is to search and check if the given node exists in the binary tree or not. If it exists, print YES otherwise print NO.

Solution: The idea is to use any of the tree traversals to traverse the tree and while traversing check if the current node matches with the given node. Print YES if any node matches with the given node and stop traversing further and if the tree is completely traversed and none of the node matches with the given node then print NO.

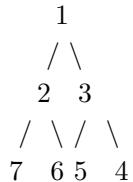
```

// Function to traverse in Preorder and check if node exists
bool ifNodeExists (Node* root, int key) {
    if (root == NULL)    return false;
    if (root->data == key)    return true;
    bool res1 = ifNodeExists(root->left, key);
    bool res2 = ifNodeExists(root->right, key);
    return (res1||res2);
}

```

Q. 3 Write a function to print **spiral order traversal** of a tree.

For example, for given tree:



Output: 1 2 3 4 5 6 7

NOTE: The above spiral traversal is in counter-clockwise direction.

Solution:

Recursive Approach:

We will use normal Level Order Traversal. An additional Boolean variable ltr is used to change printing order of levels. If ltr is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of ltr is flipped in each iteration to change the order.

Worst case time complexity of the above method is $O(n^2)$. Worst case occurs in case of skewed trees.

Iterative Approach:

We can print spiral order traversal in $O(n)$ time and $O(n)$ extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We print the nodes, and push nodes of next level in another stack.

```
// Utility Functions for Recursive Approach
int height (Node* root) {
    if (root == NULL)    return 0;
    return (max(height(root->left), height(root->right)) + 1);
}

void printGivenLevel (Node* root, int level, int ltr) {
    if (root == NULL)    return;
    if (level == 1)
        cout << root->data << " ";
    if (ltr) {
        printGivenLevel(root->left, level-1, ltr);
        printGivenLevel(root->right, level-1, ltr);
    } else {
        printGivenLevel(root->right, level-1, ltr);
        printGivenLevel(root->left, level-1, ltr);
    }
}

// Main Logic of Recursive Approach
void printSpiralRec (Node* root) {
    int h = height (root);
    bool ltr= false;
    for (int i = 1; i <= h; i++) {
        printGivenLevel(root, i, ltr);
        ltr = !ltr;
    }
    cout << endl;
}
```

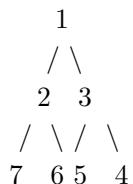
```

// Main Logic of Iterative Approach
void printSpiralItr (Node* root) {
    if (root == NULL)    return;
    // Stack 1: For levels to be printed from R to L
    stack <Node*> s1;
    // Stack 2: For levels to be printed from L to R
    stack <Node*> s2;
    s1.push(root);
    while(!s1.empty() || !s2.empty()) {
        while (!s1.empty()) {
            Node* temp = s1.top(); s1.pop();
            cout << temp->data << " ";
            // Note right node is pushed before left
            if (temp->right) s2.push(temp->right);
            if (temp->left) s2.push(temp->left);
        }
        while (!s2.empty()) {
            Node* temp = s2.top();
            s2.pop();
            cout << temp->data << " ";
            // Note left node is pushed before right
            if (temp->left) s1.push(temp->left);
            if (temp->right) s1.push(temp->right);
        }
    }
    cout << endl;
}

```

Q. 4 Write a function to print reverse level order traversal of a tree.

For example, for given tree:



Output: 7 6 5 4 2 3 1

Solution:

Approach 1 - Recursive Method

We have a method printGivenLevel() which prints a given level number. The only thing we need to change is, instead of calling printGivenLevel() from first level to last level, we call it from last level to first level.

Approach 2 - Iterative Method

The idea is to use a stack to get the reverse level order. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print contents of stack, we get “4 5 6 7 3 2 1” for above example tree, but output should be “7 6 5 4 2 3 1”. So, to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to stack, then left subtree.

```
// Auxiliary Functions
int height (Node* root) {
    if (root == NULL)    return 0;
    return (max(height(root->left), height(root->right)) + 1);
}

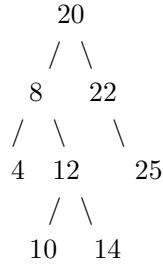
void printGivenLevel (Node* root, int level) {
    if (root == NULL)    return;
    if (level == 1) cout << root->data << " ";
    printGivenLevel(root->left, level-1);
    printGivenLevel(root->right, level-1);
}

// Recursive Function
void reverseLevelOrderRec(Node* root) {
    int h = height(root);
    //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER
    for (int i=h; i>=1; i--)
        printGivenLevel(root, i);
    cout << endl;
}

// Iterative Function
void reverseLevelOrderItr(Node* root) {
    if (root == NULL)    return;
    // Following are the differences with normal level order traversal:
    // 1) Instead of printing a node, we push the node to stack
    // 2) Right subtree is visited before left subtree
    stack <Node*> s; queue <Node*> q; q.push(root);
    while (!q.empty()) {
        root = q.front(); q.pop(); s.push(root);
        if (root->right) q.push(root->right);
        if (root->left) q.push(root->left);
    }
    while (!s.empty()) {
        root = s.top();
        cout << root->data << " "; s.pop();
    }
    cout << endl;
}
```

Q. 5 Given a binary tree, print boundary nodes of BT in anti-clockwise direction starting from the root.

For example, for the following tree:



Output: 20 8 4 10 14 25 22

Solution: We break the problem in 3 parts:

1. Print the left boundary in top-down manner.
2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
 - 2.1 Print all leaf nodes of left sub-tree from left to right.
 - 2.2 Print all leaf nodes of right subtree from left to right.
3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

```
/* A simple function to print leaf nodes of a binary tree */
void printLeaves(Node* root) {
    if (root == NULL)    return;
    printLeaves(root->left);
    if (!(root->left) && !(root->right))
        cout << root->data << " ";
    printLeaves(root->right);
}

/* Function to print all left boundary nodes in Top-Down manner,
except a leaf node. */
void printLeftBoundary (Node* root) {
    if (root == NULL)    return;
    // To ensure top-down, print the node before calling for left subtree
    if (root->left) {
        cout << root->data << " ";
        printLeftBoundary(root->left);
    }
    else if (root->right) {
        cout << root->data << " ";
        printLeftBoundary(root->right);
    }
}
```

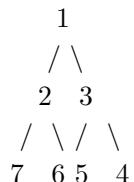
```

/* Function to print all right boundary nodes in Bottom-up manner,
except a leaf node. */
void printRightBoundary (Node* root) {
    if (root == NULL)    return;
    // for bottom-up, call for right subtree first and then print the node
    if (root->right) {
        printRightBoundary(root->right);
        cout << root->data << " ";
    }
    else if (root->left) {
        printLeftBoundary(root->left);
        cout << root->data << endl;
    }
}

void boundaryTraversal (Node* root) {
    if (root == NULL)    return;
    cout << root->data << " ";
    printLeftBoundary(root->left);
    printLeaves(root->left);
    printLeaves(root->right);
    printRightBoundary(root->right);
    cout << endl;
}

```

Q. 6 Given a BT, print the corner nodes at each level. The node at the leftmost and the node at the rightmost. For example, for given tree:



Output: 1 2 3 7 4

Solution:

Approach 1:

A Simple Solution is to do two traversals using the approaches discussed in the earlier question for printing left view and right view.

Approach 2:

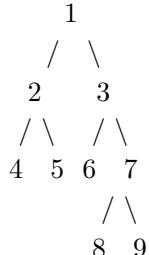
The idea is to use Level Order Traversal. Every time we store the size of the queue in a variable n, which is the number of nodes at that level. For every level we check three conditions, whether there is one node or more than one node, in case there is only one node we print it once and in case we have more than 1 nodes, we print the first (i.e node at index 0) and the node at last index (i.e node at index n-1).

```

void printCorner (Node* root) {
    if (root == NULL)    return;
    queue<Node*> q; q.push(root);
    vector<int> result; //This will store answer
    while (!q.empty()) {
        int n = q.size();
        for (int i = 0; i < n; i++) {
            Node* temp = q.front(); q.pop();
            // leftmost corner value
            if (i == 0)
                result.push_back(temp->data);
            // rightmost corner value
            else if (i == n-1)
                result.push_back(temp->data);
            if (temp->left) q.push(temp->left);
            if (temp->right) q.push(temp->right);
        }
    }
    for (auto i : result)
        cout << i << " ";
}

```

Q. 7 Given a binary tree, print it vertically. For example, for the given binary tree:



OUTPUT:

```

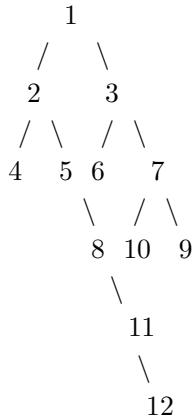
4
2
1 5 6
3 8
7
9

```

Solution:

We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do preorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the HD as 0 for root. For left subtree, we pass the HD as HD of root minus 1. For right subtree, we pass the HD as HD of root plus 1. For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in map. However, there is a problem with preorder usage in vertical traversal: Nodes in the vertical line may not be printed in same order as they appear. For example, preorder traversal prints 12 before 9 in below tree.



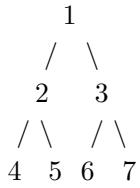
If we use level order traversal, we can make sure that if a node like 12 comes below in same vertical line, it is printed after a node like 9 which comes above in vertical line.

Time Complexity of hashing-based solution can be considered as $O(n)$ under the assumption that we have good hashing function that allows insertion and retrieval operations in $O(1)$ time. In the given C++ implementation, map of STL is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take $O(\log n)$ time. Therefore, time complexity of above implementation is $O(n * \log n)$

```

void verticalOrder(Node* root) {
    if (root == NULL)    return;
    map<int, vector<int>> m; int hd = 0;
    queue<pair<Node*, int>> q; q.push(make_pair(root, hd));
    while (!q.empty()) {
        pair <Node*, int> temp = q.front(); q.pop();
        hd = temp.second; Node* node = temp.first;
        m[hd].push_back(node->data);
        if (node->left) q.push(make_pair(node->left, hd-1));
        if (node->right) q.push(make_pair(node->right, hd+1));
    }
    map< int,vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++) {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}
    
```

One follow-up question could be – finding the vertical sums. Consider the following tree:



The tree has 5 vertical lines:

Vertical-Line-1 has only one node 4 => vertical sum is 4
Vertical-Line-2: has only one node 2 => vertical sum is 2
Vertical-Line-3: has three nodes: 1, 5, 6 => vertical sum is $1+5+6 = 12$
Vertical-Line-4: has only one node 3 => vertical sum is 3
Vertical-Line-5: has only one node 7 => vertical sum is 7

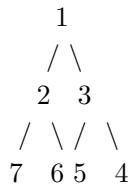
Output: 4 2 12 3 7

```
// Traverses the tree in in-order form and fills
// a hashMap that contains the vertical sum
void verticalSumUtil(Node *node, int hd, map<int, int> &Map) {
    if (node == NULL) return;
    // Recur for left subtree
    verticalSumUtil(node->left, hd-1, Map);
    // Add val of current node to map entry of corresponding HD
    Map[hd] += node->data;
    // Recur for right subtree
    verticalSumUtil(node->right, hd+1, Map);
}

// Main Logic
void verticalSum(Node *root) {
    // a map to store sum of nodes for each horizontal distance
    map <int, int> Map;
    map <int, int> :: iterator it;
    verticalSumUtil(root, 0, Map);
    for (it = Map.begin(); it != Map.end(); ++it)
        cout << it->second << " ";
    cout << endl;
}
```

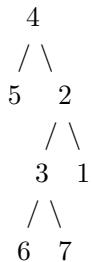
Q. 8 Write the functions for printing different views of trees. There are total 4 views of BT.

a) Left View: The left view of the following tree:



is: 1 2 7

It seems like printing the left boundary but it's different. For example, for the following tree:



Left boundary is: 4 5

But the Left View is: 4 5 3 6

We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level.

b) Right View: The right view of above tree is: 4 2 1

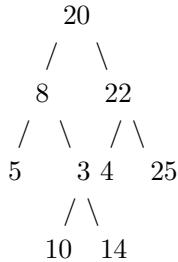
We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. And traverse the tree in a manner that right subtree is visited before left subtree. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the last node in its level. Note we traverse right subtree first in this case and then left subtree.

c) Top View: The top view of above tree is: 5 4 2 1

Like vertical Order Traversal, we need to put nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

d) Bottom View: The bottom view of above tree is: 5 6 3 7 1

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.



Output: 5 10 4 14 25

Create a map like, map where key is the horizontal distance and value is a pair(a, b) where a is the value of the node and b is the height of the node. Perform a pre-order traversal of the tree. If the current node at a horizontal distance of h is the first we've seen, insert it in the map. Otherwise, compare the node with the existing one in map and if the height of the new node is greater, update in the Map.

Following is the code to print different views of BT. We have not implemented right view function, as it is just the modification of left view function.

```

// LEFT VIEW
void leftViewUtil (Node* root, int level, int *max_level) {
    if (root == NULL)    return;
    // Check if the node is the first node of its level
    if (*max_level < level) {
        cout << root->data << " ";
        *max_level = level;
    }
    // Recur for left and right subtrees
    leftViewUtil(root->left, level+1, max_level);
    leftViewUtil(root->right, level+1, max_level);
    // Just swap the order of above recurring calls ie.
    // call root->right first and then root->left, and
    // the function will get converted to rightViewUtil.
}

void leftView(Node* root) {
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
    cout << endl;
}

```

```

// TOP VIEW
void topViewUtil (Node* root, int height, int hd,
                  map<int, pair<int, int>> &m) {
    if (root == NULL)    return;
    // If the node for particular horizontal distance
    // is not present in the map, add it.
    // For top view, we consider the first element at
    // horizontal distance in level order traversal
    if (m.find(hd) == m.end()) {
        m[hd] = make_pair(root->data, height);
    }
    else {
        pair<int, int> p = (m.find(hd))->second;
        if (p.second > height) {
            m.erase(hd);
            m[hd] = make_pair(root->data, height);
        }
    }
    // Recur for left and right subtree
    topViewUtil(root->left, height + 1, hd - 1, m);
    topViewUtil(root->right, height + 1, hd + 1, m);
}

void topView (Node* root) {
    // Map to store horizontal dist, height and node's data
    map <int, pair<int, int>> m;
    topViewUtil (root, 0, 0, m);
    map<int, pair<int, int> >::iterator it;
    for (it = m.begin(); it != m.end(); it++) {
        pair<int, int> p = it->second;
        cout << p.first << " ";
    }
    cout << endl;
}

// BOTTOM VIEW
void bottomViewUtil (Node* root, int height, int hd,
                     map<int, pair<int, int>> &m) {
    if (root == NULL)    return;

    // If node for particular HD is not present, add it.
    if (m.find(hd) == m.end())
        m[hd] = make_pair(root->data, height);
    // Compare height of already present node at similar HD
    else {
        pair <int,int> p = m[hd];

```

```

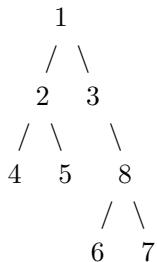
        if (p.second <= height) {
            m[hd].second = height;
            m[hd].first = root->data;
        }
    }
    // Recurr for left and right subtrees
    bottomViewUtil (root->left, height+1, hd-1, m);
    bottomViewUtil (root->right, height+1, hd+1, m);
}

void bottomView (Node* root) {
    map <int, pair<int,int>> m;
    bottomViewUtil(root, 0, 0, m);
    map <int, pair<int,int>> :: iterator it;
    for (it = m.begin(); it != m.end(); ++it) {
        pair < int, int > p = it -> second;
        cout << p.first << " ";
    }
    cout << endl;
}

```

Q. 9 Given a binary tree, write a function to get the maximum width of the given tree.

Let us consider the below example tree.



So, the maximum width of the tree is 3. And note it is not the bottom-most level of tree.

Solution:

We store all the child nodes at the current level in the queue and then count the total number of nodes after the level order traversal for a particular level is completed. Since the queue now contains all the nodes of the next level, we can easily find out the total number of nodes in the next level by finding the size of queue.

We then follow the same procedure for the successive levels. We store and update the maximum number of nodes found at each level.

Time Complexity: $O(n)$

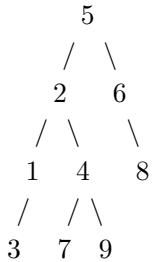
```

int maxWidth (Node* root) {
    if (root == NULL)    return 0;
    int result = 0;
    queue <Node*> q; q.push(root);
    while (!q.empty()) {
        // Get the size of queue when the level order
        // traversal of one level finishes
        int count = q.size();
        result = max (count, result);
        // Iterate for all the nodes in queue currently
        while (count--) {
            Node* temp = q.front(); q.pop();
            if (temp->left) q.push(temp->left);
            if (temp->right) q.push(temp->right);
        }
    }
    return result;
}

```

Q. 10 Given a Binary Tree, find the difference between the sum of nodes at odd level and the sum of nodes at even level. Consider root as level 1, left and right children of root as level 2 and so on.

For example, in the following tree, sum of nodes at odd level is $(5 + 1 + 4 + 8)$ which is 18. And sum of nodes at even level is $(2 + 6 + 3 + 7 + 9)$ which is 27. The output for following tree should be $18 - 27$ which is -9.



Solution:

Approach 1 - Iterative Method

A straightforward method is to use level order traversal. In the traversal, check level of current node, if it is odd, increment odd sum by data of current node, otherwise increment even sum. Finally return difference between odd sum and even sum.

Approach 2 - Recursive Method

We can recursively calculate the required difference as, value of root's data subtracted by the difference for subtree under left child and the difference for subtree under right child.

Code of recursive method is this small:

```
int getLevelDiff (Node* root) {  
    if (root == NULL)    return 0;  
    return root->data - getLevelDiff(root->left) - getLevelDiff(root->right);  
}
```

Below is the implementation of iterative approach.

```
int evenOddLevelDifference(Node* root) {  
    if (!root) return 0;  
    queue<Node*> q; q.push(root);  
    int level = 0, evenSum = 0, oddSum = 0;  
    while (!q.empty()) {  
        int size = q.size();  
        level += 1;  
        while (size > 0) {  
            Node* temp = q.front(); q.pop();  
            if (level % 2 == 0)  
                evenSum += temp->data;  
            else  
                oddSum += temp->data;  
            if (temp->left) q.push(temp->left);  
            if (temp->right) q.push(temp->right);  
            size -= 1;  
        }  
    }  
    return (oddSum - evenSum);  
}
```

Q. 11 Given Inorder and Preorder traversals of a binary tree, print Postorder traversal.

For example:

Input:

Inorder traversal in[] = {4, 2, 5, 1, 3, 6}

Preorder traversal pre[] = {1, 2, 4, 5, 3, 6}

Output:

Postorder traversal is {4, 5, 2, 6, 3, 1}

SOLUTION:

When given two traversals (one of which is Inorder), we can find the third traversal without actually creating the tree.

The idea is, root is always the first term in preorder and must be last in postorder traversal. Now to find boundaries of left and right subtrees in pre[] and in[], we search root in in[]. All elements before root belong to left subtree and all elements after root belong to right subtree. We will now use above properties recursively on subtrees.

```

void printPostOrder (int in[], int pre[], int n) {
    // First element in pre is root, search it in in[]
    int root = search(in, pre[0], n);
    // If left subtree is not empty, print left subtree
    if (root != 0)
        printPostOrder(in, pre+1, root);
    // If right subtree is not empty, print right subtree
    if (root != n-1)
        printPostOrder(in+root+1, pre+root+1, n-root-1);
    // Print root
    cout << pre[0] << " ";
}

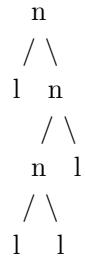
```

Q. 12 Given preorder of a full binary tree, calculate its depth [starting from depth 0]. The preorder is given as a string with two possible characters: 'l' denotes the leaf and 'n' denotes internal node. For example,

Input : nlnnlll

Output: 3

Tree:



```

int findDepthUtil (string preorder, int &index) {
    if (index >= preorder.length() || preorder[index] == 'l')
        return 0;
    // Calc height of left subtree
    // In preorder left subtree is processed before right
    index++;
    int left = findDepthUtil(preorder, index);
    // Calc height of right subtree
    index++;
    int right = findDepthUtil(preorder, index);

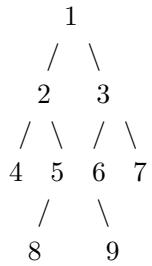
    return (max(left, right) + 1);
}

int findDepth (string preorder) {
    int index = 0;
    return findDepthUtil(preorder, index);
}

```

Q. 13 Given a binary tree containing n nodes. The problem is to get the sum of all the leaf nodes which are at minimum level in the binary tree.

Input:



Output: 11

Leaf nodes 4 and 7 are at minimum level.

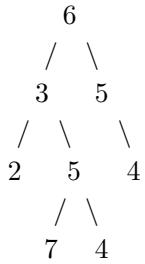
Their sum = $(4 + 7) = 11$.

Solution:

Perform iterative level order traversal using queue and find the first level containing a leaf node. Sum up all the leaf nodes at this level and then stop performing the traversal further.

```
int sumOfLeafNodesAtMinLevel(Node* root) {
    if (!root) return 0;
    if (!root->left && !root->right)
        return root->data;
    queue<Node*> q; q.push(root);
    int sum = 0; bool f = 0;
    while (f == 0) {
        int n = q.size();
        // traverse the current level nodes
        while (n--) {
            Node* top = q.front(); q.pop();
            if (!top->left && !top->right) {
                sum += top->data;
                f = 1;
            }
            else {
                if (top->left) q.push(top->left);
                if (top->right) q.push(top->right);
            }
        }
    }
    return sum;
}
```

Q. 14 Given a binary tree, print all its root to leaf paths.



There are 4 leaves, hence 4 root-to-leaf paths -

6->3->2
6->3->5->7
6->3->5->4
6->5->4

Solution:

Use a path array `path[]` to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array `path[]`. When we reach a leaf node, print the path array.

```
void printPathsUtil (Node* node, int path[], int pathLen) {  
    if (node == NULL)    return;  
    path[pathLen] = node->data;  
    pathLen++;  
    // If it's a leaf, then print the path  
    if (node->left == NULL && node->right == NULL) {  
        printArray(path, pathLen);  
    } else {  
        printPathsUtil(node->left, path, pathLen);  
        printPathsUtil(node->right, path, pathLen);  
    }  
}  
  
void printPaths(Node* node)  {  
    int path[1000];  
    printPathsUtil(node, path, 0);  
}
```

One follow-up question - Given a binary tree and a number, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given number and return false if no such path is found. Recursively check if left or right child has path sum equal to (number - value at current node).

```

bool hasPath (Node* root, int sum) {
    if (root == NULL)    return (sum==0);
    bool ans = false;
    int subSum = sum - root->data;
    if (subSum == 0 && root->left == NULL && root->right == NULL)
        return true;
    if (root->left)
        ans = ans || hasPath(root->left, subSum);
    if (root->right)
        ans = ans || hasPath(root->right, subSum);
    return ans;
}

```

Q. 15 The diameter of a tree is the number of nodes on the longest path between two end nodes. Write a code to find the diameter of a tree.

Solution:

Basic observation:

The diameter of a tree T is the largest of the following quantities:

- a) the diameter of T's left subtree
- b) the diameter of T's right subtree
- c) the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

The code of above implementation is:

```

int diameter (Node* root) {
    if (tree == NULL)    return 0;

    int l_height = height(tree->left);
    int r_height = height(tree->right);

    int l_diameter = diameter(tree->left);
    int r_diameter = diameter(tree->right);

    return max(l_height+r_height+1, max(l_diameter, r_diameter));
}

```

Time Complexity: $O(n^2)$

Here is another way to think about diameter of tree:

Diameter of a tree can be calculated by only using the height function, because the diameter of a tree is nothing but maximum value of ($\text{left_height} + \text{right_height} + 1$) for each node. So, we need to calculate this value ($\text{left_height} + \text{right_height} + 1$) for each node and update the result.

Time Complexity: $O(n)$

```

// Function to compute height
int height (Node* root, int &ans) {
    if (root == NULL)    return 0
    int left_height = height(root->left, ans);
    int right_height = height(root->right, ans);
    ans = max(ans, 1+left_height+right_height);
    return (1 + max(left_height, right_height));
}

// Computes the diameter of binary tree
int diameter (Node* root) {
    if (root == NULL)    return 0;
    int ans = INT_MIN;
    int height_of_tree = height(root, ans);
    return ans;
}

```

Q. 16 Given a binary tree in which each node element contains a number. Find the maximum possible sum from one leaf node to another.

Solution:

The idea is to maintain two values in recursive calls

- 1) Maximum root to leaf path sum for the subtree rooted under current node.
- 2) The maximum path sum between leaves (desired output).

For every visited node X, we find the maximum root to leaf sum in left and right subtrees of X. We add the two values with X->data, and compare the sum with maximum path sum found so far.

```

int maxPathSumUtil (Node* root, int &result) {
    if (root == NULL)    return 0;
    if (!root->left && !root->right) return root->data;
    // Find maximum root to leaf sums in left and right subtrees
    int lsum = maxPathSumUtil(root->left, result);
    int rsum = maxPathSumUtil(root->right, result);
    // If both left and right children exist
    if (root->left && root->right) {
        // Update result if needed
        result = max(result, lsum + rsum + root->data);
        // Return maximum possible value for root being on one side
        return max(lsum, rsum) + root->data;
    }
    // If any of the two children is empty,
    // return root sum for root being on one side
    return (!root->left)? (rsum + root->data):( lsum + root->data);
}

```

```

int maxPathSum(struct Node *root) {
    int res = INT_MIN;
    maxPathSumUtil(root, res);
    return res;
}

```

7.2 BT Construction

How many different BSTs are possible with n nodes?

Now if we are asked to write a function that takes 'n' and returns the number of possible BSTs, we basically need to write function of "*Nth Catalan Number*". Here is the code:

```

// Recursive Implementation
unsigned long int catalan(int n) {
    if (n <= 1) return 1;
    // catalan(n) is sum of catalan(i)*catalan(n-i-1)
    unsigned long int res = 0;
    for (int i=0; i<n; i++)
        res += catalan(i)*catalan(n-i-1);
    return res;
}

```

Let us see the time complexity of above function:

To evaluate the complexity, let us focus on the number of recursive calls performed, let $C(n)$. A call for n implies exactly $2(n-1)$ recursive calls, each of them adding their own costs, $2(C(1) + C(2) + \dots + C(n-1))$.

A call for $n+1$ implies exactly $2n$ recursive calls, each of them adding their own costs, $2(C(1) + C(2) + \dots + C(n-1) + C(n))$.

By difference, $C(n+1) - C(n) = 2 + 2C(n)$, which can be written
 $C(n) = 2 + 3C(n-1)$.

$$\begin{aligned}
C(1) &= 0 \\
C(2) &= 2 + 2C(1) = 2 + 3C(0) = 2 \\
C(3) &= 4 + 2(C(1) + C(2)) = 2 + 3C(2) = 8 \\
C(3) &= 6 + 2(C(1) + C(2) + C(3)) = 2 + 3C(3) = 26 \\
C(4) &= 8 + 2(C(1) + C(2) + C(3) + C(4)) = 2 + 3C(4) = 80 \\
&\dots \\
C(n) &= 2n - 2 + 2(C(1) + C(2) + \dots + C(n-1)) = 2 + 3C(n-1)
\end{aligned}$$

To solve this recurrence easily, notice that

$$C(n) + 1 = 3(C(n-1) + 1) = 9(C(n-2) + 1) = \dots = 3^{n-2}(C(2) + 1) = 3^{n-1}$$

Hence, for $n > 1$ the exact formula is $C(n) = 3^{n-1} - 1$

Therefore, time complexity is $O(3^n)$.

The time complexity could be greatly reduced by using ‘Dynamic Programming’ as we can observe that the above recursive implementation does a lot of repeated work.

Here is the dynamic programming approach for n^{th} Catalan number:

```
unsigned long int catalanDP (int n) {  
    unsigned long int catalan[n+1];  
    catalan[0] = catalan[1] = 1;  
    for (int i = 2; i <= n; i++) {  
        catalan[i] = 0;  
        for (int j = 0; j < i; j++)  
            catalan[i] += catalan[j]*catalan[i-j-1];  
    }  
    return catalan[n];  
}
```

Therefore,

Number of BTs = $catalan(n) * factorial(n)$;
with n nodes

and

Number of BSTs = $catalan(n)$;
with n nodes

If given two traversals of BT, can we construct BT uniquely?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed uniquely, otherwise not.

Therefore, the following combinations can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

The following combinations do not uniquely identify a tree.

- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

For example, Preorder, Level-order and Postorder traversals are the same for the below given trees:



Additional Point:

If someone gives you a Preorder Traversal of a binary tree, and asks - how many binary trees with this preorder traversal are possible?

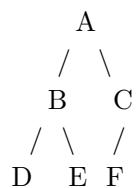
You should just count the number of nodes, say n, and return the 'n'th Catalan number. Thus, by giving preorder or postorder traversal of binary tree with 'n' nodes, the possibilities of binary trees reduced by $n!$.

Let's say, we want to construct a binary tree from given inorder and preorder traversals.

Inorder sequence: D B E A F C

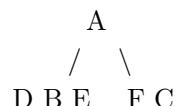
Preorder sequence: A B D E C F

Constructed tree should be:



Approach 1 - Recursion (without Hashmap)

In a Preorder sequence, the leftmost element is the root of the tree. So, we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So, we know below structure now.



We recursively follow above steps and get the following tree.

Here is the implementation of above method:

```
int search(char arr[], int strt, int end, char value);

Node* buildTree(char in[], char pre[], int inStart, int inEnd) {
    if (inStart > inEnd) return NULL;
    static int preIndex = 0;
    // Pick current node from Preorder traversal using preIndex
    // and increment preIndex
    Node* tNode = new Node(pre[preIndex++]);
    // If this node has no children then return
    if (inStart == inEnd) return tNode;
    // Else find the index of this node in Inorder traversal
    int inIndex = search(in, inStart, inEnd, tNode->data);
    //Using index in Inorder traversal, construct left and right subtress
    tNode->left = buildTree(in, pre, inStart, inIndex - 1);
    tNode->right = buildTree(in, pre, inIndex + 1, inEnd);
    return tNode;
}
```

Time Complexity: $O(n^2)$

Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for the worst case are {A, B, C, D} and {D, C, B, A}.

We can optimize the above solution using hashing (unordered_map). We store indexes of inorder traversal in a hash table. So that search can be done $O(1)$ time.

Time Complexity: $O(n)$

```
Node* buildTreeUtil(char in[], char pre[], int inStart, int inEnd,
                     unordered_map<char, int> &map) {
    if (inStart > inEnd) return NULL;
    static int preIndex = 0;
    // Pick current node from preorder traversal
    // using preIndex and increment preIndex
    char curr = pre[preIndex++];
    Node* tNode = new Node(curr);
    // If this node has no children, then return
    if (inStart == inEnd) return tNode;
    // Else find the index of this node in inorder traversal
    int inIndex = map[curr];
    // Using index in inorder traversal, construct
    // left and right subtrees
    tNode->left = buildTreeUtil(in, pre, inStart, inIndex - 1, map);
    tNode->right = buildTreeUtil(in, pre, inIndex + 1, inEnd, map);
    return tNode;
}
```

```

Node* buildTree (char in[], char pre[], int len) {
    unordered_map<char, int> map;
    for (int i = 0; i < len; i++)
        map[in[i]] = i;
    return buildTreeUtil(in, pre, 0, len-1, map);
}

```

We know that we cannot construct a unique BT from only its pre and post order traversals. We need some additional information. One such additional information could be - the type of BT. For example, consider the following problem:

PROBLEM:

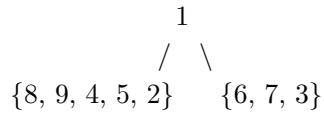
Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree. Remember a Full Binary Tree is a binary tree where every node has either 0 or 2 children.

SOLUTION:

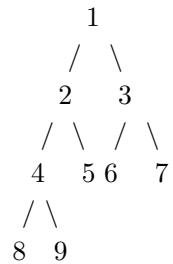
Let us consider the two given arrays as $\text{pre}[] = \{1, 2, 4, 8, 9, 5, 3, 6, 7\}$ and $\text{post}[] = \{8, 9, 4, 5, 2, 6, 7, 3, 1\}$;

In $\text{pre}[]$, the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in $\text{pre}[]$, must be left child of root. So, we know 1 is root and 2 is left child. How to find the all nodes in left subtree?

We know 2 is root of all nodes in left subtree. All nodes before 2 in $\text{post}[]$ must be in left subtree. Now we know 1 is root, elements $\{8, 9, 4, 5, 2\}$ are in left subtree, and the elements $\{6, 7, 3\}$ are in right subtree.



We recursively follow the above approach and get the following tree.



```

Node* constructTreeUtil (int pre[], int post[], int* preIndex, int l,
                        int h, int size) {
    if (*preIndex >= size || l > h) return NULL;
    // first node in preorder traversal is root. So take the node at
    // preIndex from preorder, make it root and increment preIndex
    Node* root = new Node(pre[*preIndex]);
    (*preIndex)++;
    // If the current subarray has only 1 element left, no need to recurr
    if (l == h) return root;
    // Search the next element of pre[] in post[]
    int i;
    for (i = l; i <= h; ++i)
        if (pre[*preIndex] == post[i])
            break;
    // Use the index of element found in postOrder to divide postOrder
    // array in two parts - left and right subtree
    if (i <= h) {
        root->left = constructTreeUtil(pre, post, preIndex, l, i, size);
        root->right = constructTreeUtil(pre, post, preIndex, i+1, h, size);
    }
    return root;
}

Node* constructTree (int pre[], int post[], int size) {
    int preIndex = 0;
    return constructTreeUtil(pre, post, &preIndex, 0, size-1, size);
}

```

If given a binary tree, you are asked to find number of minimum swaps needed to convert this BT into a BST. BT is given in the form of array i.e. if index i is the parent, index $2*i + 1$ is the left child and index $2*i + 2$ is the right child. How will you do that?

We know the inorder traversal of BST gives a sorted array. So, we need to find inorder traversal of given tree and store the result into an array. The question now reduces to - minimum number of swaps required to sort this array.

Time Complexity: $O(n * \log n)$

```

void inorder(int a[], std::vector<int> &v, int n, int index) {
    // if index is greater or equal to vector size
    if(index >= n) return;
    inorder(a, v, n, 2 * index + 1);
    v.push_back(a[index]);
    inorder(a, v, n, 2 * index + 2);
}

```

```

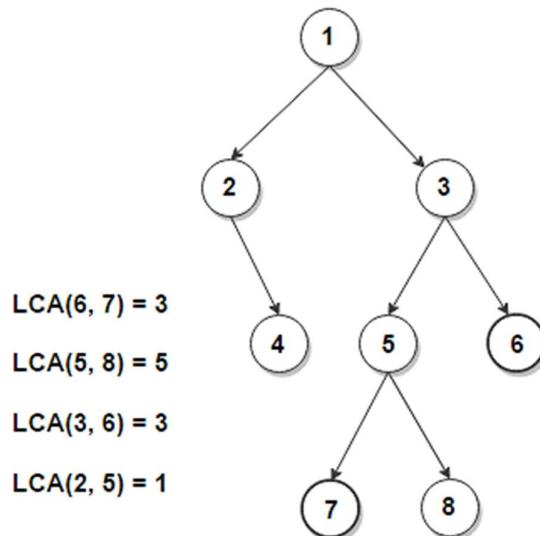
int minSwaps (vector <int> &v) {
    vector<pair<int, int>> t(v.size());
    int ans = 0;
    for (int i = 0; i < v.size(); i++)
        t[i].first = v[i], t[i].second = i;
    sort(t.begin(), t.end());
    for (int i = 0; i < t.size(); i++) {
        if (i==t[i].second)
            continue;
        else {
            swap(t[i].first, t[t[i].second].first);
            swap(t[i].second, t[t[i].second].second);
        }
        if (i != t[i].second)
            --i;
        ans++;
    }
    return ans;
}

```

7.3 Least Common Ancestor

Finding LCA:

The lowest common ancestor between two nodes n_1 and n_2 is defined as the lowest node in T that has both n_1 and n_2 as descendants (where we allow a node to be a descendant of itself).



Method 1 - By storing root to n1 and root to n2 paths

1. Find path from root to n1 and store it in an array
2. Find path from root to n2 and store it in another array
3. Traverse both the paths till the values in array are same.
Return the common element just before the mismatch

Time Complexity: $O(n)$

```
// Utility function which finds the path from root to given
// node and stores the path in vector path[] if it exists
bool findPath (Node* root, vector <int> &path, int k) {
    if (root == NULL)    return false;
    // Store this node in path. It will be removed
    // if it doesn't lie on the path from root to k
    path.push_back(root->data);
    // Now check if root's data is equal to key
    if (root->data == k)
        return true;
    // Check if k is found in left and right subtrees
    if ((root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)))
        return true;
    // If not present in subtree rooted with root, remove
    // root from path[] and return false
    path.pop_back();
    return false;
}

int findLCA (Node* root, int n1, int n2) {
    vector <int> path1, path2;
    if (!findPath(root, path1, n1) || !findPath(root, path2, n2))
        return -1;
    // Compare the paths to get first different value
    int i;
    for (i = 0; i < path1.size() && i < path2.size(); i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}
```

Method 2 - Using single traversal

Method 1 requires 3 tree traversals plus extra spaces for path arrays. In this method we will find LCA without extra space and in single traversal.

The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA. If root doesn't match with any of the nodes, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

Time complexity of the above solution is $O(n)$ as the method does a simple tree traversal in bottom up fashion.

```
// This function returns pointer to LCA of two given values
// Function assumes that n1 and n2 are present in BT
Node* findLCA (Node* root, int n1, int n2) {
    if (root == NULL)    return NULL;
    // If either n1 or n2 matches with root's data, report
    // the presence by returning root. Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA
    if (root->data == n1 || root->data == n2)
        return root;
    // Look for keys in left and right subtrees
    Node* left_lca = findLCA (root->left, n1, n2);
    Node* right_lca = findLCA (root->right, n1, n2);
    // If both of the above calls return Non-NULL, then one
    // key is present in one subtree and other is present
    // in another subtree. So this node is LCA.
    if (left_lca && right_lca)  return root;
    // Otherwise check if left subtree or right subtree contains LCA
    return ((left_lca != NULL) ? left_lca : right_lca);
}
```

Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA. (Ideally should have returned NULL). We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

```
Node *findLCAUtil(Node* root, int n1, int n2, bool &v1, bool &v2) {
    if (root == NULL)  return NULL;
    if (root->key == n1) {
        v1 = true;  return root;
    }
```

```

    if (root->key == n2) {
        v2 = true;
        return root;
    }
    Node *left_lca = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);
    if (left_lca && right_lca) return root;
    return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k) {
    if (root == NULL) return false;
    if (root->key == k || find(root->left, k) || find(root->right, k))
        return true;
    else
        return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2
// are present in tree, otherwise returns NULL;
Node *findLCA (Node *root, int n1, int n2) {
    bool v1 = false, v2 = false;
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;
    else
        return NULL;
}

```

Distance between two nodes:

Distance between two nodes is the minimum number of edges to be traversed to reach one node from other. We first find LCA of two nodes. Then we find distance from LCA to two nodes.

```

Node* findLCA (Node* root, int n1, int n2) {
    if (root == NULL) return root;
    if (root->data == n1 || root->data == n2)
        return root;
    Node* left_lca = findLCA (root->left, n1, n2);
    Node* right_lca = findLCA (root->right, n1, n2);
    if (left_lca != NULL && right_lca != NULL)
        return root;
    if (left_lca != NULL)
        return findLCA(root->left, n1, n2);
    return findLCA (root->right, n1, n2);
}

```

```

// Utility function that returns level of key k
// if it is present in the tree, or else returns -1
int findLevel(Node *root, int k, int level) {
    if(root == NULL) return -1;
    if(root->data == k) return level;
    int left = findLevel(root->left, k, level+1);
    if (left == -1)
        return findLevel(root->right, k, level+1);
    return left;
}
int findDistance(Node* root, int a, int b) {
    Node* lca = findLCA(root, a, b);
    int d1 = findLevel(lca, a, 0);
    int d2 = findLevel(lca, b, 0);
    return d1 + d2;
}

```

Kth Ancestor:

Given a binary tree in which nodes are numbered from 1 to n. Given a node and a positive integer K. We have to print the Kth ancestor of the given node in the binary tree. If there does not exist any such ancestor then print -1.

The idea to do this is to first traverse the binary tree and store the ancestor of each node in an array of size n. For example, suppose the array is ancestor[n]. Then at index i, ancestor[i] will store the ancestor of ith node. So, the 2nd ancestor of ith node will be ancestor[ancestor[i]] and so on. We will use this idea to calculate the kth ancestor of the given node. We can use level order traversal to fill this array of ancestors.

```

// Function to generate array of ancestors
void generateArray(Node *root, int ancestors[]) {
    // There will be no ancestor of root node
    ancestors[root->data] = -1;
    // Level order traversal to generate 1st ancestor
    queue<Node*> q; q.push(root);
    while(!q.empty()) {
        Node* temp = q.front(); q.pop();
        if (temp->left) {
            ancestors[temp->left->data] = temp->data;
            q.push(temp->left);
        }
        if (temp->right) {
            ancestors[temp->right->data] = temp->data;
            q.push(temp->right);
        }
    }
}

```

```

int kthAncestor(Node *root, int n, int k, int node) {
    // create array to store 1st ancestors
    int ancestors[n+1] = {0};
    // generate first ancestor array
    generateArray(root,ancestors);
    // variable to track record of number of ancestors visited
    int count = 0;
    while (node != -1) {
        node = ancestors[node]; count++;
        if(count==k)
            break;
    }
    // print Kth ancestor
    return node;
}

```

We have discussed a BFS based solution for this problem in our previous code. If you observe that solution carefully, you will see that the basic approach was to first find the node and then backtrack to the kth parent. The same thing can be done using recursive DFS without using an extra array.

The idea of using DFS is to first find the given node in the tree, and then backtrack k times to reach to kth ancestor, once we have reached to the kth parent, we will simply print the node and return NULL.

```

Node* kthAncestorDFS(Node *root, int node , int &k) {
    if (!root)  return NULL;
    static Node* temp = NULL;
    if (root->data == node || 
        (temp = kthAncestorDFS(root->left,node,k)) ||
        (temp = kthAncestorDFS(root->right,node,k))) {
        if (k > 0) k--;
        else if (k == 0) {
            // print the kth ancestor
            cout << "Kth ancestor is: " << root->data << endl;
            // return NULL to stop further backtracking
            return NULL;
        }
        // return current node to previous call
        return root;
    }
}

```

ADDITIONAL APPROACH (simple one):

Firstly, we find the path of given key data from the root and we will store it into a vector then we simply return the kth index of the vector from the last.

```
bool RootToNode(node* root, int key, vector<int>& v) {
    if (root == NULL)    return false;
    // Add current node to the path
    v.push_back(root->data);
    // If current node is the target node
    if (root->data == key)  return true;
    // If the target node exists in the left or the right sub-tree
    if (RootToNode(root->left, key, v) ||
        RootToNode(root->right, key, v))
        return true;
    // Remove the last inserted node as it is not a part of
    // the path from root to target
    v.pop_back();
    return false;
}
```

Cousin Nodes:

Given the binary Tree and the two nodes say ‘a’ and ‘b’, determine whether the two nodes are cousins of each other or not. Two nodes are cousins of each other if they are at same level and have different parents.

The idea is to find level of one of the nodes. Using the found level, check if ‘a’ and ‘b’ are at this level. If ‘a’ and ‘b’ are at given level, then finally check if they are not children of same parent.

```
// Recursive function to check if two Nodes are siblings
int isSibling(Node *root, Node *a, Node *b) {
    if (root==NULL)  return 0;
    return ((root->left==a && root->right==b) ||
            (root->left==b && root->right==a) || \
            isSibling(root->left, a, b) || isSibling(root->right, a, b));
}

// Recursive function to find level of Node 'ptr' in a binary tree
int level(Node *root, Node *ptr, int lev) {
    if (root == NULL) return 0;
    if (root == ptr)  return lev;
    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev+1);
    if (l != 0)  return l;
    // Else search in right subtree
    return level(root->right, ptr, lev+1);
}
```

```

// Returns 1 if a and b are cousins, otherwise 0
int isCousin( Node *root, Node *a, Node *b) {
    //1. The two Nodes should be on the same level in the binary tree.
    //2. The two Nodes should not be siblings (means that they should
    // not have the same parent Node).
    if ((level(root,a,1) == level(root,b,1)) && !(isSibling(root,a,b)))
        return 1;
    else return 0;
}

```

Previous solution, which finds whether given nodes are cousins or not, performs three traversals of binary tree. The problem can be solved by performing single level order traversal.

The idea is to use a queue to perform level order traversal, in which each queue element is a pair of node and parent of that node. For each node visited in level order traversal, check if that node is either first given node or second given node. If any node is found store parent of that node. While performing level order traversal, one level is traversed at a time. If both nodes are found in given level, then their parent values are compared to check if they are siblings or not. If one node is found in given level and another is not found, then given nodes are not cousins.

```

bool isCousin (Node* root, Node* a, Node* b) {
    if (root == NULL)    return false;
    // To store parent of node a.
    Node* parent_A = NULL;
    // To store parent of node b.
    Node* parent_B = NULL;
    // Each element of queue is a pair of node and its parent.
    queue<pair<Node*, Node*> > q;
    // Dummy node to act like parent of root node.
    Node* temp = new Node(-1);
    // To store front element of queue.
    pair<Node*, Node*> ele;
    q.push(make_pair(root, temp));
    int levelSize;
    while (!q.empty()) {
        // find number of elements in current level.
        levelSize = q.size();
        while (levelSize) {
            ele = q.front(); q.pop();
            // check if current node is node a or node b or not.
            if (ele.first->data == a->data)
                parent_A = ele.second;
            if (ele.first->data == b->data)
                parent_B = ele.second;
    }
}

```

```

        // push children of current node to queue.
        if (ele.first->left)
            q.push(make_pair(ele.first->left, ele.first));
        if (ele.first->right)
            q.push(make_pair(ele.first->right, ele.first));
        levelSize--;
        // If both nodes are found in current level then no need
        // to traverse current level further.
        if (parent_A && parent_B)
            break;
    }
    // Check if both nodes are siblings or not.
    if (parent_A && parent_B)
        return parent_A != parent_B;
    // If one node is found in current level and another is not
    // found, then both nodes are not cousins.
    if ((parent_A && !parent_B) || (parent_B && !parent_A))
        return false;
}
return false;
}

```

Problems

Q. 1 Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.



Solution:

We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be n1 and n2 respectively. There are following two conditions for subtrees, rooted with n1 and n2, to be isomorphic.

- 1) Data of n1 and n2 is same.
- 2) One of the following two is true for children of n1 and n2
 - a) Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.
 - b) Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.

```

bool isIsomorphic(Node* n1, Node *n2) {
    // Both roots are NULL, trees isomorphic by definition
    if (n1 == NULL && n2 == NULL)
        return true;
    // Exactly one of the n1 and n2 is NULL, trees not isomorphic
    if (n1 == NULL || n2 == NULL)
        return false;
    if (n1->data != n2->data)
        return false;
    // There are two possible cases for n1 and n2 to be isomorphic
    // Case 1: Subtrees rooted at these nodes have NOT been "Flipped".
    // Both of these subtrees have to be isomorphic, hence &&
    // Case 2: Subtrees rooted at these nodes have been "Flipped"
    return
        (isIsomorphic(n1->left,n2->left) &&
         isIsomorphic(n1->right,n2->right)) ||
        (isIsomorphic(n1->left,n2->right) &&
         isIsomorphic(n1->right,n2->left));
}

```

Q. 2 Factor Tree is an intuitive method to understand the factors of a number. It shows how all the factors are been derived from the number. It is a special diagram where you find the factors of a number, then the factors of those numbers, etc until you can't factor anymore. The ends are all the prime factors of the original number.

Input: v = 48

Output: Root of below tree

```

48
 / \
2 24
   / \
 2 12
    / \
   2 6
    / \
   2 3

```

Solution: The factor tree is created recursively as follows:

1. We start with a number and find the minimum divisor possible.
2. Then, we divide the parent number by the minimum divisor.
3. We store both the divisor and quotient as two children of the parent number.
4. Both the children are sent into function recursively.
5. If a divisor less than half the number is not found, two children are stored as NULL.

```
void createFactorTree(struct Node **node_ref, int v) {  
    (*node_ref) = new Node(v);  
    // the number is factorized  
    for (int i = 2 ; i < v/2 ; i++) {  
        if (v % i != 0)  
            continue;  
        // If we found a factor, we construct left and right subtrees  
        // and return. Since we traverse factors starting from smaller  
        // to greater, left child will always have smaller factor  
        createFactorTree(&(*node_ref)->left), i);  
        createFactorTree(&(*node_ref)->right), v/i);  
        return;  
    }  
}
```

Q. 3 Given a binary tree, find all duplicate subtrees. For each duplicate subtrees, we only need to return the root node of any one of them. Two trees are duplicate if they have the same structure with same node values.

Input:

```
1  
/\  
2 3  
/ /\  
4 2 4  
/  
4
```

Output:

```
2  
/ and 4  
4
```

Solution:

The idea is to use hashing. We store inorder traversals of subtrees in a hash. Since simple inorder traversal cannot uniquely identify a tree, we use symbols like '(' and ')' to represent NULL nodes. We pass an Unordered Map in C++ as an argument to the helper function

which recursively calculates inorder string and increases its count in map. If any string gets repeated, then it will imply duplication of the subtree rooted at that node so push that node in Final result and return the vector of these nodes.

```

string inorder(Node* node, unordered_map<string, int>& m) {
    if (!node)
        return "";
    string str = "(";
    str += inorder(node->left, m);
    str += to_string(node->data);
    str += inorder(node->right, m);
    str += ")";
    // Subtree already present (Note that we use unordered_map
    // instead of unordered_set because we want to print multiple
    // duplicates only once.
    if (m[str] == 1)
        cout << node->data << " ";
    m[str]++;
    return str;
}

// Wrapper over inorder()
void printAllDups(Node* root) {
    unordered_map<string, int> m;
    inorder(root, m);
}

```

Q. 4 Given a binary tree, return the tilt of the whole tree. The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of all right subtree node values. Null nodes are assigned tilt to be zero. Therefore, tilt of the whole tree is defined as the sum of all nodes' tilt.

Input:

```

1
/
2 \ 3

```

Output: 1

Explanation:

Tilt of node 2: 0

Tilt of node 3: 0

Tilt of node 1: $|2-3| = 1$

Tilt of binary tree: $0 + 0 + 1 = 1$

Input:

```
4
 / \
2   9
 / \   \
3   5   7
```

Output: 15

Explanation:

Tilt of node 3: 0

Tilt of node 5: 0

Tilt of node 7: 0

Tilt of node 2: $|3-5| = 2$

Tilt of node 9: $|0-7| = 7$

Tilt of node 4: $|(3+5+2)-(9+7)| = 6$

Tilt of binary tree: $0 + 0 + 0 + 2 + 7 + 6 = 15$

Solution:

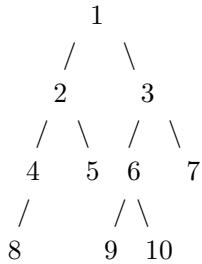
The idea is to recursively traverse tree. While traversing, we keep track of two things, sum of subtree rooted under current node, tilt of current node. Sum is needed to compute tilt of parent.

```
int traverse(Node* root, int* tilt) {
    if (!root)
        return 0;
    // Compute tilts of left and right subtrees
    // and find sums of left and right subtrees
    int left = traverse(root->left, tilt);
    int right = traverse(root->right, tilt);
    // Add current tilt to overall
    *tilt += abs(left - right);
    // Returns sum of nodes under current tree
    return left + right + root->data;
}

// Driver function to print Tilt of whole tree
int Tilt(Node* root) {
    int tilt = 0;
    traverse(root, &tilt);
    return tilt;
}
```

Q. 5 Given a binary tree and two nodes. The task is to count the number of turns needed to reach from one node to another node of the Binary tree.

Input: Below is the BT and two nodes are 5 & 6



Output: Number of Turns needed to reach from 5 to 6: 3

Input: For above tree if two nodes are 1 & 4

Output: Straight line. Hence, 0 turns.

Solution:

1. Find the LCA of given two nodes
2. Given node present either on the left side or right side or equal to LCA.

According to above condition, we fall in one of the following two cases.

Case 1:

If none of the nodes is equal to LCA, we get these nodes either on the left side or right side. We call two functions for each node.

- a) `if (CountTurn(LCA->right, first, false, &Count) || CountTurn(LCA->left, first, true, &Count)) ;`
- b) Same for second node.

Here Count is used to store number of turns need to reached the target node.

Case 2:

If one of the nodes is equal to LCA _ Node. Then we count only number of turns needed to reached the second node.

```
if LCA == (Either first or second)
    if (countTurn(LCA->right, second/first, false, &Count) ||
        countTurn(LCA->left, second/first, true, &Count)) ;
```

Working of CountTurn Function:

We pass turn true if we move left subtree and false if we move right subTree.

```

CountTurn(LCA, Target_node, count, Turn):
    // if found the data value in tree
    if (root->data == data)
        return true;

```

case 1:

If Turn is true that means we are in left_subtree
If we going left_subtree then there is no need to increment count
Else increment count and set turn as false

case 2:

If Turn is false that means we are in right_subtree
If we going right_subtree then there is no need to increment count
Else increment count and set turn as true.

// if data is not found.

return false;

```

bool CountTurn(Node* root, int data, bool turn, int* count) {
    if (root == NULL)
        return false;
    // if found the data value in tree
    if (root->data == data)
        return true;
    // Case 1:
    if (turn == true) {
        if (CountTurn(root->left, data, turn, count))
            return true;
        if (CountTurn(root->right, data, !turn, count)) {
            *count += 1;
            return true;
        }
    } // Case 2:
    else {
        if (CountTurn(root->right, data, turn, count))
            return true;
        if (CountTurn(root->left, data, !turn, count)) {
            *count += 1;
            return true;
        }
    }
    return false;
}

```

```

// Function to find nodes common to given two nodes
int NumberOFTurn(struct Node* root, int first, int second) {
    struct Node* LCA = findLCA(root, first, second);
    if (LCA == NULL)
        return -1;
    int Count = 0;
    // case 1:
    if (LCA->data != first && LCA->data != second) {
        // count number of turns needed to reach second node from LCA
        if (CountTurn(LCA->right, second, false, &Count) || \
            CountTurn(LCA->left, second, true, &Count)) ;
        // count number of turns needed to reach first node from LCA
        if (CountTurn(LCA->left, first, true, &Count) || \
            CountTurn(LCA->right, first, false, &Count)) ;
        return Count + 1;
    }
    // case 2:
    if (LCA->data == first) {
        // count number of turns needed to reach second node from LCA
        CountTurn(LCA->right, second, false, &Count);
        CountTurn(LCA->left, second, true, &Count);
        return Count;
    } else {
        // count number of turns needed to reach first node from LCA
        CountTurn(LCA->right, first, false, &Count);
        CountTurn(LCA->left, first, true, &Count);
        return Count;
    }
}

```

Q. 6 Given a binary tree, find the path length having maximum number of bends.

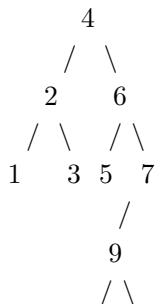
Note: Here, bend indicates switching from left to right or vice versa while traversing in the tree. For example, consider below paths:

LLRRRR – 1 Bend

RLLLRR – 2 Bends

LRLRLR – 5 Bends

Input:



```

12 10
 \
11
 / \
45 13
 \
14

```

Output: 6

In the above example, the path 4-> 6-> 7-> 9-> 10-> 11-> 45 is having the maximum number of bends, i.e., 3. The length of this path is 6.

Solution: The idea is to traverse the tree for left and right subtrees of the root. While traversing, keep track of the direction of motion (left or right). Whenever, direction of motion changes from left to right or vice versa increment the number of bends in the current path by 1. On reaching the leaf node, compare the number of bends in the current path with the maximum number of bends (i.e., maxBends) seen so far in a root-to-leaf path. If the number of bends in the current path is greater than the maxBends, then update the maxBends equal to the number of bends in the current path and update the maximum path length (i.e., len) also to the length of the current path.

```

// soFar => length of the current path so far traversed
// len => length of the path having maximum number of bends
void findMaxBendsUtil(struct Node* node, char direction, int bends,
                      int* maxBends, int soFar, int* len) {
    if (node == NULL) return;
    if (node->left == NULL && node->right == NULL) {
        if (bends > *maxBends) {
            *maxBends = bends; *len = soFar;
        }
    }
    else {
        if (direction == 'l') {
            findMaxBendsUtil(node->left, direction, bends,
                              maxBends, soFar + 1, len);
            findMaxBendsUtil(node->right, 'r', bends + 1,
                              maxBends, soFar + 1, len);
        }
        else {
            findMaxBendsUtil(node->right, direction, bends,
                              maxBends, soFar + 1, len);
            findMaxBendsUtil(node->left, 'l', bends + 1,
                              maxBends, soFar + 1, len);
        }
    }
}

```

```

int findMaxBends(struct Node* node) {
    if (node == NULL)
        return 0;
    int len = 0, bends = 0, maxBends = -1;
    // Call for left subtree of the root
    if (node->left)
        findMaxBendsUtil(node->left, 'l', bends, &maxBends, 1, &len);
    // Call for right subtree of the root
    if (node->right)
        findMaxBendsUtil(node->right, 'r', bends, &maxBends, 1, &len);
    // Include the root node as well in the path length
    len++;
    return len;
}

```

Q. 7 Given a skewed tree (Every node has at most one child) with N nodes and K colors. You have to assign a color from 1 to K to each node such that parent and child has different colors. Find the maximum number of ways of coloring the nodes.

Input: N = 2, K = 2.

Output: 2

Let A1 and A2 be the two nodes.

Let A1 is parent of A2.

Colors are Red and Blue.

Case 1: A1 is colored Red and A2 is colored Blue.

Case 2: A1 is colored Blue and A2 is colored Red.

No. of ways: 2

Input: N = 3, K = 3.

Output: 12

A1, A2, A3 are the nodes.

A1 is parent of A2 and A2 is parent of A3.

Let colors be R, B, G.

A1 can choose any three colors and A2 can choose any other two colors and A3 can choose any other two colors than its parents.

No. of ways: 12

Solution:

Note that only the root and children (children, grandchildren, grand-grand-children ... and all) should have different colors. The root of the tree can choose any of the K colors so K ways. Every other node can choose other K-1 colors other than its parent. So, every node has K-1 choices.

Here, we select the tree as every node as only one child. We can choose any of the K colors for the root of the tree so K ways. And we are left with K-1 colors for its child. So, for every child we can assign a color other than its parent. Thus, for each of the N-1 nodes we are left with K-1 colors. Thus, the answer is $K * (K - 1)^{N-1}$.

```

int fastPow(int N, int K) {
    if (K == 0)
        return 1;
    int temp = fastPow(N, K / 2);
    if (K % 2 == 0)
        return temp * temp;
    else
        return N * temp * temp;
}

int countWays(int N, int K) {
    return K * fastPow(K - 1, N - 1);
}

```

Q. 8 Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration).

Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child's parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Solution:

This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

- If a child node i takes c_i iterations to pass info below its subtree, then its parent will take $(c_i + 1)$ iterations to pass info to subtree rooted at that child i.
- If parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes $c_1, c_2, c_3, c_4, \dots, c_n$ iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in i^{th} iteration, then parent will take $(i + c_i)$ iterations to pass info to child i and all its subtree.
- In any iteration, when parent passes info to a child $i+1$, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.

- To pass info to whole tree in minimum iterations, it needs to be made sure that bandwidth is utilized as efficiently as possible (i.e. maximum passable no of nodes should pass info further down in any iteration).

The best possible scenario would be that in n^{th} iteration, n different nodes pass info to their child.

Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn't receive the info yet). One iteration needed for each child and so no of iterations would be no of children. So, node with height 1 with n children will take n iterations. Take a counter initialized with zero, loop through all children and keep incrementing counter.

Nodes with height > 1 : Let's assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c_1, c_2, \dots, c_n . To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no. of iterations so that all subsequent iterations can be utilized efficiently.

If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.

If there are two children i and j with minimum iterations c_i and c_j where $c_i > c_j$, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be: $\max(1 + c_j, 2 + c_i) = 2 + c_i$.

If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: $\max(1 + c_i, 2 + c_j) = 1 + c_i$ (So picking c_i gives better result than picking c_j).

This tells that parent should always choose child i with max c_i value in any iteration. So here greedy approach is:

- Sort all c_i values decreasing order,
- Let's say after sorting, values are $c_1 > c_2 > c_3 > \dots > c_n$
- Take a counter c , set $c = 1 + c_1$ (for child with maximum no of iterations)
- For all children i from 2 to n , $c = c + 1 + c_i$
- Then total no of iterations needed by parent is $\max(n, c)$

Let $\text{minItr}(A)$ be the minimum iteration needed to pass info from node A to its all the subtree. Let $\text{child}(A)$ be the count of all children for node A.
So recursive relation would be:

1. Get $\text{minItr}(B)$ of all children (B) of a node (A)
 2. Sort all $\text{minItr}(B)$ in descending order
 3. Get minItr of A based on all $\text{minItr}(B)$
- $$\text{minItr}(A) = \text{child}(A)$$
- For children B from $i = 0$ to $\text{child}(A)$
- $$\text{minItr}(A) = \max(\text{minItr}(A), \text{minItr}(B) + i + 1)$$

Base cases would be:

If node is leaf, $\text{minItr} = 0$

If node's height is 1, $\text{minItr} = \text{children count}$

```
// A class to represent n-ary tree
// (Note that the implementation is similar to graph)
class NAryTree {
    int N; // No. of nodes in Tree
    // Pointer to an array containing list of children
    list<int> *adj;
    // A function used by getMinIter(), it basically does postorder
    void getMinIterUtil(int v, int minItr[]);

public:
    NAryTree(int N); // Constructor
    // function to add a child w to v
    void addChild(int v, int w);
    // The main function to find minimum iterations
    int getMinIter();
    static int compare(const void * a, const void * b);
};

NAryTree::NAryTree(int N) {
    this->N = N;
    adj = new list<int>[N];
}

// To add a child w to v
void NAryTree::addChild(int v, int w) {
    adj[v].push_back(w); // Add w to v's list.
}
```

```

/* A recursive function to be used by getMinIter(). This function
mainly does postorder traversal and gets minimum iterations of all
children of node u. Sort them in decreasing order and then get the
minimum iteration of node u.

1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
    minItr(A) = child(A) --> child(A) is children count of node A
    For children B from i = 0 to child(A)
        minItr(A) = max ( minItr(A), minItr(B) + i + 1)

Base cases would be:
If node is leaf, minItr = 0
If node's height is 1, minItr = children count
*/
void NARYTree::getMinIterUtil(int u, int minItr[]) {
    minItr[u] = adj[u].size();
    int *minItrTemp = new int[minItr[u]];
    int k = 0, tmp = 0;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        getMinIterUtil(*i, minItr);
        minItrTemp[k++] = minItr[*i];
    }
    qsort(minItrTemp, minItr[u], sizeof (int), compare);
    for (k = 0; k < adj[u].size(); k++) {
        tmp = minItrTemp[k] + k + 1;
        minItr[u] = max(minItr[u], tmp);
    }
    delete[] minItrTemp;
}

int NARYTree::getMinIter() {
    // Set minimum iteration all the vertices as zero
    int *minItr = new int[N];
    int res = -1;
    for (int i = 0; i < N; i++)
        minItr[i] = 0;
    // Start Post Order Traversal from Root
    getMinIterUtil(0, minItr);
    res = minItr[0];
    delete[] minItr;
    return res;
}

```

```

int NArTree::compare(const void * a, const void * b) {
    return ( *(int*)b - *(int*)a );
}

int main() {
    // TestCase 1
    NArTree tree1(17);
    tree1.addChild(0, 1);
    tree1.addChild(0, 2);
    tree1.addChild(0, 3);
    tree1.addChild(0, 4);
    tree1.addChild(0, 5);
    tree1.addChild(0, 6);

    tree1.addChild(1, 7);
    tree1.addChild(1, 8);
    tree1.addChild(1, 9);

    tree1.addChild(4, 10);
    tree1.addChild(4, 11);

    tree1.addChild(6, 12);

    tree1.addChild(7, 13);
    tree1.addChild(7, 14);
    tree1.addChild(10, 15);
    tree1.addChild(11, 16);

    cout << "TestCase 1 - Minimum Iteration: " << tree1.getMinIter()
        << endl;
    return 0;
}

```

Chapter 8 – Binary Search Trees

8.1 Search and Insertion

Binary Search Tree (BST) is a BT which has the following properties:

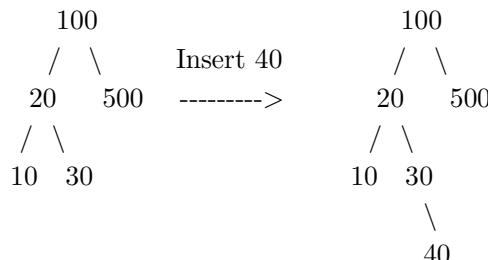
1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a BST.
4. There are no duplicate nodes.

Searching a key in BST:

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Inserting a key in BST:

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.



Time Complexity: The worst-case time complexity of search and insert operations is $O(h)$, where h is height of Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity may become $O(n)$.

```
Node* search (Node* root, int key) {  
    if (root == NULL || root->data == key)  
        return root;  
    if (root->data < key)  
        return search(root->right, key);  
    else  
        return search(root->left, key);  
}
```

```

Node* insert (Node* root, int key) {
    if (root == NULL) {
        Node* temp = new Node(key);
        return temp;
    }
    if (root->data < key)
        root->right = insert (root->right, key);
    else
        root->left = insert (root->left, key);
    return root;
}

```

8.2 Deletion

Before understanding how deletion works, we need to know – how to find the inorder successor and predecessor of a given key. In case the given key is not found, we then find two values within which the key will lie. Here is a recursive algorithm:

1. If root is NULL, return.
2. If key is found, then
 - a) If its left subtree is not NULL, then predecessor will be the rightmost child of left subtree or that left child itself.
 - b) If its right subtree is not NULL, then successor will be the leftmost child of right subtree or that right child itself.
3. If key is smaller than root,
 - a) set the successor as root and search recursively into left subtree
 - Else
 - b) set the predecessor as root and search recursively into right subtree

```

void findPreSuc (Node* root, Node* &pre, Node* &suc, int key) {
    if (root == NULL) return;
    if (root->data == key) {
        if (root->left != NULL) {
            Node* temp = root->left;
            while (temp->right)
                temp = temp->right;
            pre = temp;
        }
        if (root->right != NULL) {
            Node* temp = root->right;
            while (root->left)
                temp = temp->left;
            suc = temp;
        }
    }
    return;
}

```

```

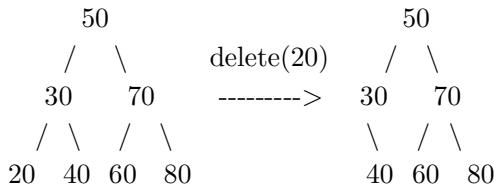
    if (root->data > key) {
        suc = root;
        findPreSuc(root->left, pre, suc, key);
    }
    else {
        pre = root;
        findPreSuc(root->right, pre, suc, key);
    }
}

```

When we delete a node, three possibilities arise:

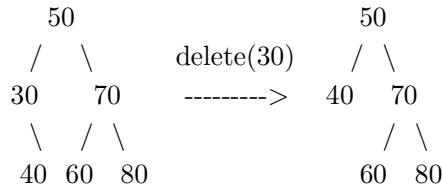
1. Node to be deleted is leaf:

In that case simply remove it from the tree.



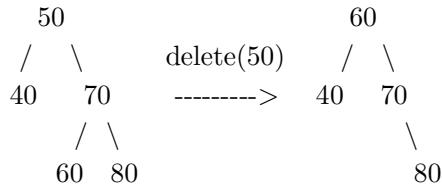
2. Node to be deleted has only one child:

Copy the child to the node and delete the child.



3. Node to be deleted has two children:

Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



```

// Utility function that finds minimum value in BST
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

```

```

Node* deleteNode (Node* root, int key) {
    if (root == NULL)    return root;
    if (root->data > key)
        root->left = deleteNode(root->left, key);
    else if (root->data < key)
        root->right = deleteNode(root->right, key);
    else {
        // node with only one child or no child
        if (root->left == NULL) {
            Node* temp = root->right; delete root;
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left; delete root;
            return temp;
        }
        // Node with two children
        // Get inorder successor
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

```

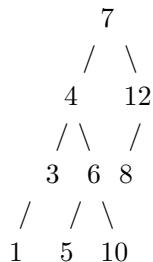
8.3 Traversals of BST

Given an array of size n. The problem is to check whether the given array can represent the level order traversal of a Binary Search Tree or not.

Input: arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}

Output: Yes

For the given arr[] the Binary Search Tree is:



Input: arr[] = {11, 6, 13, 5, 12, 10}

Output: No

The given arr[] do not represent the level order traversal of a BST.

The idea is to use a queue data structure. Every element of queue has a structure say NodeDetails which stores details of a tree node. The details are node's data, and two variables min and max where min stores the lower limit for the node values which can be a part of the left subtree and max stores the upper limit for the node values which can be a part of the right subtree for the specified node in NodeDetails structure variable. For the 1st array value arr[0], create a NodeDetails structure having arr[0] as node's data and min = INT_MIN and max = INT_MAX. Add this structure variable to the queue. This Node will be the root of the tree. Move to 2nd element in arr[] and then perform the following steps:

1. Pop NodeDetails from the queue in temp.
2. Check whether the current array element can be a left child of the node in temp with the help of min and temp.data values. If it can, then create a new NodeDetails structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, and move to next element in arr[].
3. Check whether the current array element can be a right child of the node in temp with the help of max and temp.data values. If it can, then create a new NodeDetails structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, and move to next element in arr[].
4. Repeat steps 1, 2 and 3 until there are no more elements in arr[] or there are no more elements in the queue.
5. Finally, if all the elements of the array have been traversed then the array represents the level order traversal of a BST, else NOT.

```
bool isLevelOrderOfBST(int arr[], int n) {  
    if (n == 0) return true;  
    queue<NodeDetails> q; NodeDetails newNode;  
    int i=0; newNode.data = arr[i++];  
    newNode.min = INT_MIN; newNode.max = INT_MAX;  
    q.push(newNode);  
    while (i != n && !q.empty()) {  
        NodeDetails temp = q.front(); q.pop();  
        if (i < n && (arr[i] < temp.data && arr[i] > temp.min)) {  
            newNode.data = arr[i++];  
            newNode.min = temp.min; newNode.max = temp.data;  
            q.push(newNode);  
        }  
        if (i < n && (arr[i] > temp.data && arr[i] < temp.max)) {  
            newNode.data = arr[i++];  
            newNode.min = temp.data; newNode.max = temp.max;  
            q.push(newNode);  
        }  
    }  
}
```

```

    if (i == n) return true;
    return false;
}

```

Given an array of numbers, return true if given array can represent preorder traversal of a BST, else return false.

Simple Solution:

Do following for every node $\text{pre}[i]$ starting from first one:

Find the first greater value on right side of current node. Let the index of this node be j .

- a) All values after the above found greater value are greater than current node.
- b) Recursive calls for the subarrays $\text{pre}[i+1\dots j-1]$ and $\text{pre}[j+1\dots n-1]$ also return true.

Return true if above conditions hold or else return false.

Time Complexity: $O(n^2)$

Efficient Solution:

The idea is to use a stack. This problem is similar to Next Greater Element problem. Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

1. Create an empty stack.
2. Initialize root as INT_MIN .
3. Do following for every element $\text{pre}[i]$
 - a) If $\text{pre}[i]$ is smaller than current root, return false.
 - b) Keep removing elements from stack while $\text{pre}[i]$ is greater than stack top. Make the last removed item as new root (to be compared next). At this point, $\text{pre}[i]$ is greater than the removed root (That is why if we see a smaller element in step a), we return false)
 - c) push $\text{pre}[i]$ to stack (All elements in stack are in decreasing order)

Time Complexity: $O(n)$

Below is the implementation of efficient solution.

```

bool isPreBST (int pre[], int n) {
    stack<int> s; int root = INT_MIN;
    for (int i = 0; i < n; i++) {
        if (pre[i] < root) return false;
        while (!s.empty() && s.top()<pre[i]) {
            root = s.top(); s.pop();
        }
        s.push(pre[i]);
    }
    return true;
}

```

8.4 LCA

The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., $n_1 < n < n_2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n_1 < n_2$).

So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST).

```
Node* lca (Node* root, int n1, int n2) {  
    if (root == NULL) return NULL;  
    if (root->data > n1 && root->data > n2)  
        return lca(root->left, n1, n2);  
    if (root->data < n1 && root->data < n2)  
        return lca(root->right, n1, n2);  
    return root;  
}
```

Given a Binary Search Tree and two keys in it. Find the distance between two nodes with given two keys. It may be assumed that both keys exist in BST.

For this, we start from the root and for every node, we do the following:

- If both keys are greater than the current node, we move to the right child of the current node.
- If both keys are smaller than current node, we move to left child of current node.
- If one keys is smaller and other key is greater, current node is the Lowest Common Ancestor (LCA) of two nodes.

We find distances of current node from two keys and return sum of the distances.

```
int distanceFromRoot(Node* root, int x) {  
    if (root->data == x)  
        return 0;  
    else if (root->data > x)  
        return (1 + distanceFromRoot(root->left, x));  
    return (1 + distanceFromRoot(root->right, x));  
}
```

```

int distanceBetween2(Node* root, int a, int b) {
    if (!root) return 0;
    // Both keys lie in left
    if (root->data > a && root->data > b)
        return distanceBetween2(root->left, a, b);
    // Both keys lie in right
    if (root->data < a && root->data < b)
        return distanceBetween2(root->right, a, b);
    // Lie in opposite directions (Root is LCA of two nodes)
    if (root->data >= a && root->data <= b)
        return distanceFromRoot(root, a) + distanceFromRoot(root, b);
}

int findDist(Node *root, int a, int b) {
    if (a > b)
        swap(a, b);
    return distanceBetween2(root, a, b);
}

```

8.5 Interval Trees

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently:

- 1) Add an interval
- 2) Remove an interval
- 3) Given an interval x, find if x overlaps with any of the existing intervals.

The idea is to use – ‘Interval Trees’. Every node of Interval Tree stores the following information.

- a) i: An interval which is represented as a pair [low, high]
- b) max: Maximum high value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST.

Below we implement Interval Trees to solve problem of conflicting appointments (intervals).

```

#include <bits/stdc++.h>
using namespace std;

struct Interval {
    int low, high;
};

```

```

class Node {
public:
    Interval *i;
    int max;
    Node *left, *right;
    Node (Interval x) {
        this->i = new Interval (x);
        this->max = i->high;
        this->left = this->right = NULL;
    }
};

Node* insert (Node* root, Interval i) {
    if (root == NULL) {
        Node* temp = new Node(i);
        return temp;
    }
    int l = root->i->low;
    if (i.low < l)
        root->left = insert(root->left, i);
    else
        root->right = insert(root->right, i);
    if (root->max < i.high)
        root->max = i.high;
    return root;
}

bool doOverlap (Interval i1, Interval i2) {
    if (i1.low < i2.high && i2.low < i1.high)
        return true;
    return false;
}

Interval* overlapSearch (Node* root, Interval i) {
    if (root == NULL)    return NULL;
    if (doOverlap(*root->i), i))
        return root->i;
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);
    return overlapSearch(root->right, i);
}

void printConflicting (Interval appt[], int n) {
    Node* root = NULL;
    root = insert (root, appt[0]);
    for (int i = 1; i < n; i++) {

```

```

        Interval* res = overlapSearch(root, appt[i]);
        if (res != NULL)
            cout << "[" << appt[i].low << "," << appt[i].high
                << "] Conflicts with [" << res->low << ","
                << res->high << "]\n";
            root = insert(root, appt[i]);
    }
}

int main() {
    Interval appt[]={{1, 5}, {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}};
    int n = sizeof(appt)/sizeof(appt[0]);
    cout << "Following are conflicting intervals:\n";
    printConflicting(appt, n);
    return 0;
}

```

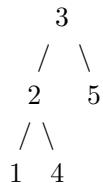
Problems

Q. 1 Given a BT, check if it's a BST or not.

Solution:

The first idea that comes to mind is - to write a recursive function which checks if the left node is smaller than the current node and right node is greater. If yes return true, else return false.

However, this approach is wrong as above approach will give true for the following BT which is not a BST because '4' lies in the left subtree of '3'.



The correct idea is - look each node only once.

```
isBST (Node* root, Node* left, Node* right);
```

that traverses down the tree keeping track of the narrowing min and max allowed values as it goes down. Finally compare max and min with the root.

Time Complexity: $O(n)$

```

bool isBST (Node* root, Node* l = NULL, Node* r = NULL) {
    if (root == NULL)    return true;
    if (l != NULL and root->data <= l->data)
        return false;
    if (r != NULL and root->data >= r->data)
        return false;
    return isBST(root->left, l, root) && isBST(root->right, root, r);
}

```

Q. 2 Given a BT, write a function that returns the size of the largest subtree which is also a BST. If the complete BT is BST, then return the size of the whole tree.

Solution:

The idea of solution is somewhat similar to previous question.

A Tree is BST if following is true for every node x:

- a) The largest value in left subtree (of x) is smaller than value of x.
- b) The smallest value in right subtree (of x) is greater than value of x.

We traverse tree in bottom up manner. For every traversed node, we return maximum and minimum values in subtree rooted with it. If any node follows above properties, return the size of subtree rooted at this node.

Time Complexity: $O(n)$

```

struct Info {
    int size;    // size of subtree
    int max;     // max value of subtree
    int min;     // min value of subtree
    int ans;     // size of the largest BST
    bool isBST; // If subtree is BST
};

// Returns Info about the size of largest subtree which is BST
Info largestBST (Node* root) {
    // if the tree is empty or has lone node
    if (root == NULL)
        return {0, INT_MIN, INT_MAX, 0, true};
    if (root->left == NULL && root->right == NULL)
        return {1, root->data, root->data, 1, true};
    // Recurr for left and right subtrees
    Info l = largestBST(root->left);
    Info r = largestBST(root->right);
    // Create a return variable
    Info ret;
    ret.size = 1 + l.size + r.size;
    if (l.max <= root->data && r.min >= root->data)
        ret.isBST = true;
    else
        ret.isBST = false;
    return ret;
}

```

```

// If whole tree rooted under current node is BST
if (l.isBST && r.isBST && l.max < root->data &&
    r.min > root->data) {
    ret.min = min (l.min, min(r.min, root->data));
    ret.max = max (r.max, max(l.max, root->data));
    // Update answer for tree rooted under current node 'root'
    ret.ans = ret.size;
    ret.isBST = true;
}
// If whole tree is not BST, return max of left and right subtrees
ret.ans = max(l.ans, r.ans);
ret.isBST = false;
return ret;
}

```

Q. 3 Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Solution:

Approach 1:

A simple way is to one by once search every node of first tree in second tree. Time complexity of this solution is $O(m * h)$ where m is number of nodes in first tree and h is height of second tree. In the worst case, second tree may be skewed and the time complexity will turn $O(m * n)$.

Approach 2:

We can find common elements in linear time, by following way:

1. Do inorder traversal of first tree and store the traversal in an auxiliary array arr1[].
2. Do inorder traversal of second tree and store the traversal in an auxiliary array arr2[].
3. Find the intersection of sorted arrays arr1[] and arr2[].

Time and space complexity: $O(m + n)$

Approach 3:

We can find common elements in $O(n)$ time and $O(h1 + h2)$ extra space where h1 and h2 are heights of first and second BSTs respectively. The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

```

void printCommon (Node* root1, Node* root2) {
    stack <Node*> stack1, s1, s2;
    while (1) {
        if (root1) {
            s1.push(root1);
            root1 = root1->left;
        }

```

```

        else if (root2) {
            s2.push(root2);
            root2 = root2->left;
        }
        else if (!s1.empty() && !s2.empty()){
            root1 = s1.top();
            root2 = s2.top();
            if (root1->data == root2->data) {
                cout << root1->data << " ";
                s1.pop();
                s2.pop();
                root1 = root1->right;
                root2 = root2->right;
            }
            else if (root1->data < root2->data) {
                s1.pop();
                root1 = root1->right;
                root2 = NULL;
            }
            else if (root1->data > root2->data) {
                s2.pop();
                root2 = root2->right;
                root1 = NULL;
            }
        }
        else
            break;
    }
}

```

Q. 4 Given root of BST and K as input, find K-th smallest element in BST.

Solution:

Method 1 - Using inorder traversal

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes. The idea is to keep track of popped elements which participate in order statistics. Time complexity: $O(n)$ and Space complexity: $O(n)$

```

int kthSmallest (Node* root, int k) {
    stack<Node*> s; int count = 0;
    Node* ptr = root; s.push(ptr);
    while (ptr->left) {
        s.push(ptr->left); ptr = ptr->left;
    }
    while (++count < k) {
        Node* temp = s.top()->right; s.pop();

```

```

        if(temp) {
            s.push(temp);
            while (temp->left) {
                s.push(temp->left); temp = temp->left;
            }
        }
    }
    return s.top()->data;
}

```

Method 2 – Ranking each node

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If $K = N + 1$, root is K-th node. If $K < N$, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If $K > N + 1$, we continue our search in the right subtree for the $(K - N - 1)$ th smallest element. Note that we need the count of elements in left subtree only. Time complexity: $O(h)$ where h is height of tree.

```

// Utility Function that returns number of nodes
int num (Node* root) {
    if (root == NULL)
        return 0;
    return (num(root->left) + num(root->right) + 1);
}

int kthSmallest (Node* root, int k) {
    int nodes_in_left = num(root->left);
    if (nodes_in_left == k-1)
        return root->data;
    else if (nodes_in_left > k-1)
        return kthSmallest(root->left, k);
    else
        return kthSmallest(root->right, k-nodes_in_left-1);
}

```

Q. 5 Find the median of the elements in BST in $O(n)$ time and $O(1)$ space complexity.

Solution:

1. Use any algo to count the number of nodes. Let it be n.
2. Use Morris Inorder. for each node visited increment the count.
3. if n is even, return when counter == n/2
else, n is odd, return when counter == (n+1)/2

```

int num (Node* root) {
    if (root == NULL)    return 0;
    int count = 0;
    Node *curr, *prev; curr = root;
    while (curr != NULL) {
        if (curr->left == NULL) {
            // Count node if its left is NULL
            count++;
            // Move to its right
            curr = curr->right;
        }
        else {
            // Find inorder predecessor of current
            prev = curr->left;
            while (prev->right != NULL && prev->right != curr)
                prev = prev->right;
            // Make current as right child of its inorder predecessor
            if (prev->right == NULL) {
                prev->right = curr;
                curr = curr->left;
            }
            // Revert the changes made in if part to restore the
            // original tree i.e., fix the right child of predecessor
            else {
                prev->right = NULL;
                // Increment count if the current node is to be visited
                count++;
                curr = curr->right;
            } // End of if condition: prev->right == NULL
        } // End of if condition: curr->left == NULL
    } // End of while
    return count;
}

// Find median with O(1) space using 'Morris Inorder Traversal'
double findMedian (Node* root) {
    if (root == NULL)    return 0.0;
    int count = num(root);
    int current_count = 0;
    Node* current = root, *pre, *prev;
    while (current != NULL){
        if (current->left == NULL) {
            current_count++;
            // Check if current node is median
            // Odd case:
            if (count%2 != 0 && current_count == (count+1)/2)
                return (double)prev->data;
        }
    }
}

```

```

        // Even case:
    else if (count%2 == 0 && current_count == (count/2)+1)
        return ((prev->data + current->data)/2.0);
    // Update prev for even number of nodes
    prev = current;
    current = current->right;
}
else {
    // Finding the inorder predecessor of current
    pre = current->left;
    while (pre->right != NULL && pre->right != current)
        pre = pre->right;
    // Make current as right child of its inorder predecessor
    if (pre->right == NULL) {
        pre->right = current;
        current = current->left;
    }
    // Revert the changes made in if part to restore the
    // original tree i.e., fix the right child of predecessor
    else {
        pre->right = NULL;
        prev = pre;
        current_count++;
        // Check if the current node is median
        if (count%2 != 0 && current_count == (count+1)/2)
            return (double)prev->data;
        else if (count%2 == 0 && current_count == (count/2)+1)
            return ((prev->data + current->data)/2.0);
        // Update prev node for the case of even number of nodes
        prev = current;
        current = current->right;
    }
}
}
}
}

```

Q. 6 Given a BST and a target node K. The task is to find the node with minimum absolute difference with given target value K.

Solution:

A simple solution for this problem is to do inorder traversal of BST and find the node with the minimum absolute difference.

Time complexity: $O(n)$

An efficient solution for this problem is to take advantage of characteristics of BST.

Here is the algorithm to solve this problem:

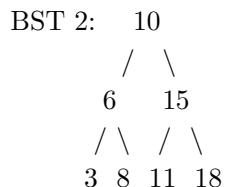
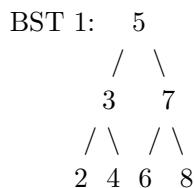
- a) If target value K is present in given BST, then it's the node having minimum absolute difference.
- b) If target value K is less than the value of current node then move to the left child.
- c) If target value K is greater than the value of current node then move to the right child.

Time Complexity: $O(h)$

```
Node* closestHelper (Node* root, int val, Node* closestNode) {  
    if (root == NULL)    return closestNode;  
  
    if (root->data == val)  
        return root;  
  
    if (closestNode == NULL || abs(root->data - val) <  
        abs(closestNode->data - val))  
        closestNode = root;  
  
    if (val < root->data)  
        return closestHelper(root->left, val, closestNode);  
    else  
        return closestHelper(root->right, val, closestNode);  
}  
  
Node* closestElement (Node* root, int val) {  
    return closestHelper(root, val, NULL);  
}
```

Q. 7 Given two BSTs containing n1 and n2 distinct nodes respectively and a value x. The problem is to count all pairs from both the BSTs whose sum is equal to x.

Input: x = 16 for following BSTs.



Output: 3

Explanation: The pairs are: (5, 11), (6, 10) and (8, 8)

Solution:

Method 1:

For each node value a in BST 1, search the value $(x - a)$ in BST 2. If value found then increment the count. Time complexity: $O(n1 * h2)$, here $n1$ is number of nodes in first BST and $h2$ is height of second BST. In the worst case of skewed trees, $h2 = n2$.

Method 2:

1. Traverse BST 1 from the smallest value to node to the largest. This can be achieved with the help of iterative inorder traversal.
2. Traverse BST 2 from the largest value node to the smallest. This can be achieved with the help of reverse inorder traversal.
3. Perform these two traversals simultaneously. Sum up the corresponding node's value from both the BSTs at a particular instance of traversals.
4. If $\text{sum} == x$, then increment count.
5. If $x > \text{sum}$, then move to the inorder successor of the current node of BST 1, else move to the inorder predecessor of the current node of BST 2.
6. Perform these operations until either of the two traversals gets completed.

Time and Space Complexity: $O(n1 + n2)$

```
int countPairs (Node* root1, Node* root2, int x) {  
    if (root1 == NULL || root2 == NULL)  
        return 0;  
    // stack1 is for inorder traversal of BST1  
    // stack2 is for reverse inorder traversal of BST2  
    stack <Node*> s1, s2;  
    Node *top1, *top2;  
    int count = 0;  
    while (1) {  
        // Find next node in inorder traversal of BST1  
        while (root1 != NULL) {  
            s1.push(root1); root1 = root1->left;  
        }  
        // Find next node in reverse inorder of BST2  
        while (root2!= NULL) {  
            s2.push(root2); root2 = root2->right;  
        }  
        // If either gets empty, traversal is complete  
        if (s1.empty() || s2.empty())  
            break;  
        top1 = s1.top(); top2 = s2.top();
```

```

        // if the sum of top nodes is equal to x, increment count
        if ((top1->data + top2->data) == x) {
            count++;

            s1.pop(); s2.pop();
            root1 = top1->right;
            root2 = top2->left;
        }
        // else if move to next possible node in BST1
        else if ((top1->data + top2->data) < x) {
            s1.pop();
            root1 = top1->right;
        }
        // else move to next possible node in reverse inorder
        // traversal of BST2
        else {
            s2.pop();
            root2 = top2->left;
        }
    }
    return count;
}

```

Q. 8 Given two values k_1 and k_2 (where $k_1 < k_2$) and a root pointer to a Binary Search Tree. Print all the keys of tree in range k_1 to k_2 . i.e. print all x such that $k_1 \leq x \leq k_2$ and x is a key of given BST.

Solution:

Approach 1: Using Inorder Traversal

- 1) If value of root's key is greater than k_1 , then recursively call in left subtree.
- 2) If value of root's key is in range, then print the root's key.
- 3) If value of root's key is smaller than k_2 , then recursively call in right subtree.

Time Complexity: $O(n)$ and Space Complexity: $O(n)$

Approach 2: Using Morris Inorder Traversal

The logic remains the same as above. Only the space complexity will improve to $O(1)$. We have implemented Morris Inorder solution below.

```

void rangeTraversal (Node* root, int n1, int n2) {
    if (!root)  return;
    Node* curr = root;
    while (curr != NULL) {
        if (curr->left == NULL) {
            if (curr->data <= n2 && curr->data >= n1)
                cout << curr->data << " ";
            curr = curr->right;
        }
    }
}

```

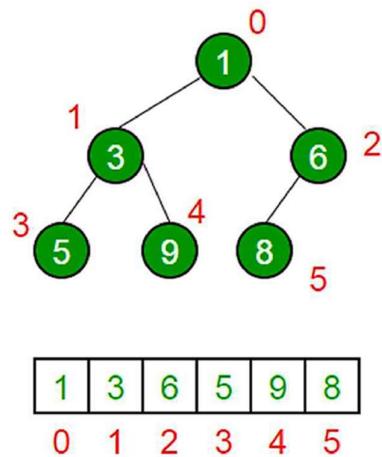
```
    else {
        Node* pre = curr->left;
        while (pre->right != NULL && pre->right != curr)
            pre = pre->right;
        if (pre->right == NULL) {
            pre->right = curr;
            curr = curr->left;
        }
        else {
            pre->right = NULL;
            if (curr->data <= n2 && curr->data >= n1)
                cout << curr->data << " ";
            curr = curr->right;
        }
    }
}
```

Chapter 9 - Heaps

9.1 Implementation

A Binary Heap is a Binary Tree with following properties:

1. It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.



A binary heap can be represented using an array.

The root element will be at $\text{Arr}[0]$.

For the i^{th} node, i.e., $\text{Arr}[i]$:

- a) $\text{Arr}[(i-1)/2]$ -> Returns the parent node
- b) $\text{Arr}[(2*i)+1]$ -> Returns the left child node
- c) $\text{Arr}[(2*i)+2]$ -> Returns the right child node

Operations on Min-Heap:

- 1) $\text{getMin}()$: It returns the root element of Min Heap.

Time Complexity of this operation is $O(1)$.

- 2) $\text{extractMin}()$: Removes the minimum element from MinHeap.

Time Complexity of this Operation is $O(\log n)$ as this operation needs to maintain the heap property (by calling $\text{heapify}()$) after removing root.

3) decreaseKey(): Decreases value of key.

Time complexity of this operation is $O(\log n)$. If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) insert(): Inserting a new key takes $O(\log n)$ time. We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) delete(): Deleting a key also takes $O(\log n)$ time. We replace the key to be deleted with negative infinity by calling decreaseKey(). After decreaseKey(), the negative infinite value must reach root, and then we can call extractMin() to remove the key.

```
#include <bits/stdc++.h>
using namespace std;

class MinHeap {
    int* arr;
    int capacity;
    int heap_size;
public:
    MinHeap(int capacity);
    void minHeapify (int);
    int parent(int i) { return (i-1)/2; }
    int left (int i) { return (2*i+1); }
    int right (int i) { return (2*i+2); }
    int extractMin();
    void decreaseKey (int i, int new_value);
    int getMin() { return arr[0]; }
    void deleteKey (int i);
    void insertKey (int k);
};

// Function Definitions
MinHeap :: MinHeap (int cap) {
    heap_size = 0;
    capacity = cap;
    arr = new int[cap];
}

void MinHeap::insertKey(int k) {
    if (heap_size == capacity) {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }
```

```

// Insert node key in the end
heap_size++;
int i = heap_size - 1;
arr[i] = k;
// Heapify if heap property is violated
while (i != 0 && arr[parent(i)] > arr[i]) {
    swap(arr[i], arr[parent(i)]);
    i = parent(i);
}
}

void MinHeap :: decreaseKey (int i, int new_value) {
    arr[i] = new_value;
    while (i != 0 && arr[parent(i)]>arr[i]) {
        swap (arr[i], arr[parent(i)]);
        i = parent(i);
    }
}

int MinHeap :: extractMin () {
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1) {
        heap_size--;
        return arr[0];
    }
    int root = arr[0]; arr[0] = arr[heap_size-1];
    heap_size--;
    minHeapify(0);
    return root;
}

void MinHeap :: deleteKey (int i) {
    decreaseKey(i, INT_MIN); extractMin();
}

void MinHeap :: minHeapify (int i) {
    int l = left(i), r = right(i), smallest = i;
    if (l < heap_size && arr[l] < arr[i])
        smallest = l;
    if (r < heap_size && arr[r] < arr[smallest])
        smallest = r;
    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        minHeapify(smallest);
    }
}

```

```

int main() {
    MinHeap h(11);
    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    cout << h.extractMin() << " ";
    cout << h.getMin() << " ";
    h.decreaseKey(2, 1);
    cout << h.getMin() << endl;
    return 0;
}

```

9.2 Kth smallest and Heap Sort

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that all array elements are distinct.

```

int kthSmallest(int arr[], int n, int k) {
    MinHeap heap(arr, n);
    for (int i=0; i<k-1; i++)
        heap.extractMin();
    return heap.getMin();
}

```

Given an array, how to check if the given array represents a Binary Max-Heap.

The idea is to compare root only with its children (not all descendants), if root is greater than its children and the same is true for all nodes, then tree is max-heap (This conclusion is based on transitive property of $>$ operator, i.e., if $x > y$ and $y > z$, then $x > z$).

```

bool isHeap (int arr[], int i, int n) {
    if (i > (n-2)/2) return true; // If node is a leaf node
    // If node is an internal node and is greater than its children
    // and same is recursively true for children as well
    if (arr[i] >= arr[2*i+1] && arr[i] >= arr[2*i+2] &&
        isHeap(arr, 2*i+1, n) && isHeap(arr, 2*i+2, n))
        return true;
    return false;
}

```

Consider an array 'arr' which is to be sorted using Heap Sort.

- a) Initially build a max heap of elements in 'arr'.
- b) The root element, that is arr[0], will contain maximum element of arr. After that, swap this element with the last element of arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- c) Repeat the step 2, until all the elements are in their correct position.

Time complexity: $O(n + n * \log n) = O(n * \log n)$

```
#include<bits/stdc++.h>
using namespace std;

// Auxiliary Functions
void printArray(int arr[], int n) {
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

void heapify (int arr[], int n, int i) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;
    if (l<n && arr[l]>arr[largest])
        largest = l;
    if (r<n && arr[r]>arr[largest])
        largest = r;
    if (largest != i) {
        swap (arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort (int arr[], int n) {
    for (int i = n/2-1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n-1; i>=0; i--) {
        swap (arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

```

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    heapSort(arr, n);
    cout << "Sorted array is: \n";
    printArray(arr, n);
}

```

9.3 Using priority queues

Heaps in STL are implemented as ‘priority queues.’

Let’s say, we have to – ‘Sort an almost sorted array’. That means, given an array of ‘ n ’ elements, where each element is at most k away from its target position, devise an algorithm that sorts in $O(n * \log k)$.

For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Input : arr[] = {6, 5, 3, 2, 8, 10, 9} and $k = 3$

Output : arr[] = {2, 3, 5, 6, 8, 9, 10}

Input : arr[] = {10, 9, 8, 7, 4, 70, 60, 50} and $k = 4$

Output : arr[] = {4, 7, 8, 9, 10, 50, 60, 70}

We can do so, with the use of Heap data structure.

- 1) Create a Min Heap of size $k+1$ with first $k+1$ elements. This will take $O(k)$ time.
- 2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take $\log k$ time. So, the overall complexity will be $O(k) + O((n - k) * \log k)$.

```

int sortK (int arr[], int n, int k) {
    priority_queue<int, vector<int>, greater<int>> pq (arr, arr+k+1);
    int index = 0;
    for (int i = k+1; i < n; i++) {
        arr[index++] = pq.top(); pq.pop();
        pq.push(arr[i]);
    }
    while (!pq.empty()) {
        arr[index++] = pq.top(); pq.pop();
    }
}

```

Q. 1 Given an array of n numbers and a positive integer k. The problem is to find k numbers with most occurrences, i.e., the top k numbers having the maximum frequency. If two numbers have same frequency then the larger number should be given preference. The numbers should be displayed in decreasing order of their frequencies. It is assumed that the array consists of k numbers with most occurrences.

Method 1: Use Hashing

Using hash table, we create a frequency table which stores the frequency of occurrence of each number in the given array. In the hash table we define (x, y) tuple, where x is the key (number) and y is its frequency in the array. Now we traverse this hash table and create an array freq_arr[] which stores these (number, frequency) tuples. Sort this freq_arr[] on the basis of the conditions defined in the problem statement. Now, print the first k numbers of this freq_arr[].

Time Complexity: $O(d * \log d)$ and Space Complexity: $O(d)$ where d is the number of distinct elements.

Method 2: Use Heap

We create a frequency table which stores the frequency of occurrence of each number in the given array. In the hash table we define (x, y) tuple, where x is the key (number) and y is its frequency in the array. Now we traverse this hash table and create an array freq_arr[] which stores these (number, frequency) tuples. Sort this freq_arr[] on the basis of the conditions defined in the problem statement. Now, print the first k numbers of this freq_arr[].

Time Complexity: $O(k * \log d)$ and Space Complexity: $O(d)$ where d is the number of distinct elements.

Both the methods have been implemented below.

```
// Comparison function to sort the freq_arr[]
bool compare1 (pair<int, int> p1, pair<int, int> p2) {
    // If frequencies are same, higher number gets precedence
    if (p1.second == p2.second)
        return p1.first > p2.first;
    // Sort on the basis of decreasing order of frequencies
    return (p1.second > p2.second);
}

struct compare2 {
    bool operator() (pair<int, int> p1, pair<int, int> p2) {
        if (p1.second == p2.second)
            return p1.first < p2.first;
        return p1.second < p2.second;
    }
};
```

```

// Function to print the k numbers with most occurrences
void kMostFrequent1 (int arr[], int n, int k) {
    unordered_map<int, int> m;
    for (int i = 0; i < n; i++)
        m[arr[i]]++;
    // Store the elements of m in the vector 'freq_arr'
    vector <pair<int, int>> freq_arr (m.begin(), m.end());
    sort (freq_arr.begin(), freq_arr.end(), compare1);
    cout << k << " numbers with most occurrences are:\n";
    for (int i = 0; i < k; i++)
        cout << freq_arr[i].first << " ";
    cout << endl;
}

void kMostFrequent2 (int arr[], int n, int k) {
    unordered_map <int, int> m;
    for (int i = 0; i < n; i++)
        m[arr[i]]++;
    vector <pair<int, int>> freq_arr (m.begin(), m.end());
    priority_queue <pair<int, int>, vector<pair<int, int>>, compare2>
    pq(m.begin(), m.end());

    cout << k << " numbers with most occurrences are:\n";
    for (int i = 1; i <= k; i++) {
        cout << pq.top().first << " ";
        pq.pop();
    }
    cout << endl;
}

int main() {
    int arr[] = {3, 1, 4, 4, 5, 2, 6, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;
    kMostFrequent1(arr, n, k);
    kMostFrequent2(arr, n, k);
    return 0;
}

```

Q. 2 Given k sorted arrays of possibly different sizes, merge them and print the sorted output.

Input: k = 3

```

arr[ ][ ] = { {1, 3},
              {2, 4, 6},
              {0, 9, 10, 11}} ;

```

Output: 0 1 2 3 4 6 9 10 11

A simple solution is to create an output array and one by one copy all arrays to it. Finally, sort the output array using. This approach takes $O(N \log N)$ time where N is count of all elements.

An efficient solution uses Heap data structure.

1. Create an output array.
2. Create a min heap of size k and insert 1st element in all the arrays into the heap
3. Repeat following steps while priority queue is not empty:
 - a) Remove minimum element from heap (minimum is always at root) and store it in output array.
 - b) Insert next element from the array from which the element is extracted. If the array doesn't have any more elements, then do nothing.

Time Complexity: $O(N * \log k)$

```
// A pair of int and pair. First element is
// element and second pair stores index of 2D array
typedef pair<int, pair<int, int>> ppi;

vector<int> mergeKArrays (vector<vector<int>> arr) {
    vector <int> output;
    // Create a min heap with k heap nodes. Every
    // heap node has first element of array
    priority_queue <ppi, vector<ppi>, greater<ppi>> pq;
    for (int i = 0; i < arr.size(); i++)
        pq.push({arr[i][0], {i, 0}});
    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    while (!pq.empty()) {
        ppi curr = pq.top();
        pq.pop();
        int i = curr.second.first;    // array number
        int j = curr.second.second;   // index in array number
        output.push_back (curr.first);
        // next element belongs to same element as current
        if (j+1 < arr[i].size())
            pq.push({arr[i][j+1], {i, j+1}});
    }
    return output;
}
```

```

int main() {
    vector<vector<int>> arr{ { 2, 6, 12 },
                           { 1, 9 },
                           { 23, 34, 90, 200 } };
    vector<int> output = mergeKArrays(arr);
    cout << "Merged array is " << endl;
    for (auto x : output)
        cout << x << " ";
    cout << endl;

    return 0;
}

```

Q. 3 Given that integers are being read from a data stream. Find median of all the elements read so far starting from the first integer till the last integer. This is also called Median of Running Integers. The data stream can be any source of data, example: a file, an array of integers, input stream etc.

Input: 5 10 15

Output: 5 7.5 10

Explanation: Given the input stream as an array of integers [5,10,15]. We will now read integers one by one and print the median correspondingly. So, after reading first element 5, median is 5. After reading 10, median is 7.5 After reading 15, median is 10.

The idea is to use max heap and min heap to store the elements of higher half and lower half. Max heap and min heap can be implemented using priority_queue in C++ STL. Algorithm:

1. Create two heaps - a max heap to maintain elements of lower half and a min heap to maintain elements of higher half at any point.
2. Take initial median as 0.
3. For every newly read element, insert it into either max heap or min heap and calculate the median based on the following conditions:
 - a) If the size of max heap is greater than size of min heap and the element is less than previous median then pop the top element from max heap and insert into min heap and insert the new element to max heap else insert the new element to min heap. Calculate the new median as average of top of elements of both max and min heap.
 - b) If the size of max heap is less than size of min heap and the element is greater than previous median then pop the top element from min heap and insert into max heap and insert the new element to min heap else insert the new element to max heap. Calculate the new median as average of top of elements of both max and min heap.
 - c) If the size of both heaps is same. Then check if current is less than previous median or not. If the current element is less than previous median then insert it to max heap and new median will be equal to top element of max heap. If the current element is

greater than previous median then insert it to min heap and new median will be equal to top element of min heap.

Time Complexity: $O(n * \log k)$ and Space Complexity: $O(n)$

```
void medianOfRunningIntegers (double arr[], int n) {
    // max heap to store the smaller half elements
    priority_queue <double> s;
    // min heap to store the greater half elements
    priority_queue<double, vector<double>, greater<double> > g;
    double med = arr[0];
    s.push(arr[0]);
    cout << med << endl;
    for (int i=1; i < n; i++) {
        double x = arr[i];
        // case1 (left side heap has more elements)
        if (s.size() > g.size()) {
            if (x < med) {
                g.push(s.top()); s.pop();
                s.push(x);
            }
            else
                g.push(x); med = (s.top() + g.top())/2.0;
        }
        // case2 (both heaps are balanced)
        else if (s.size()==g.size()) {
            if (x < med) {
                s.push(x); med = (double)s.top();
            }
            else {
                g.push(x); med = (double)g.top();
            }
        }
        // case 3 (right side heap has more elements)
        else {
            if (x > med) {
                s.push(g.top()); g.pop();
                g.push(x);
            }
            else
                s.push(x); med = (s.top() + g.top())/2.0;
        }
        cout << med << endl;
    }
}
```

Chapter 10 – Graphs

10.1 Graph Representation

Graphs are mathematical structures that represent pairwise relationships between objects. It can be visualized by using the following two basic components:

Nodes: These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes, A and B, and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.

Edges: Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

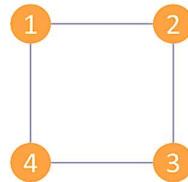
Types of nodes:

Root node: The root node is the ancestor of all other nodes in a graph. It does not have any ancestor. Each graph consists of exactly one root node. Generally, you start traversing a graph from the root node.

Leaf nodes: In a graph, leaf nodes represent the nodes that do not have any successors. These nodes only have ancestor nodes. They can have any number of incoming edges but they will not have any outgoing edges.

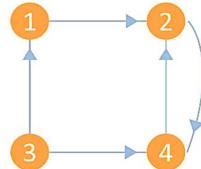
Types of graphs:

Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



Undirected Graph

Directed: A directed graph is a graph in which all the edges are unidirectional i.e. the edges point in a single direction.



Directed Graph

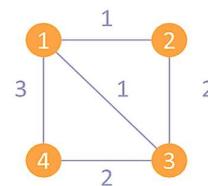
Weighted: In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

1 -> 2 -> 3

1 -> 3

1 -> 4 -> 3

Therefore, the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units. The total cost of 1 -> 3 will be 1 unit. The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



Weighted Graph

Cyclic: A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex, then that path is called a cycle. An acyclic graph is a graph that has no cycle.

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has $N - 1$ edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

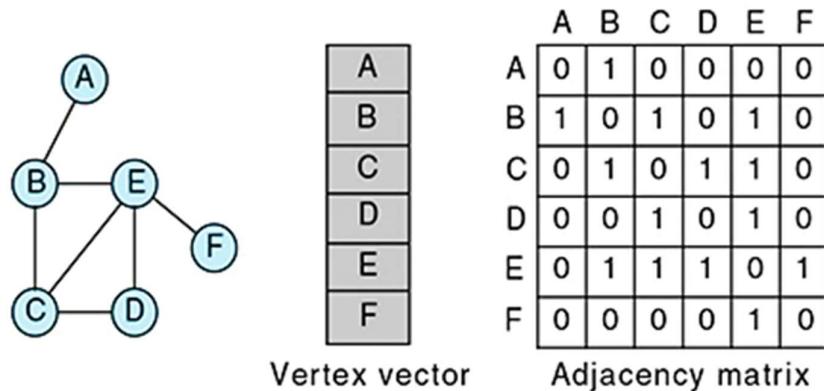
Note: A root node has no parent.

A tree cannot contain any cycles or self-loops, however, the same does not apply to graphs.

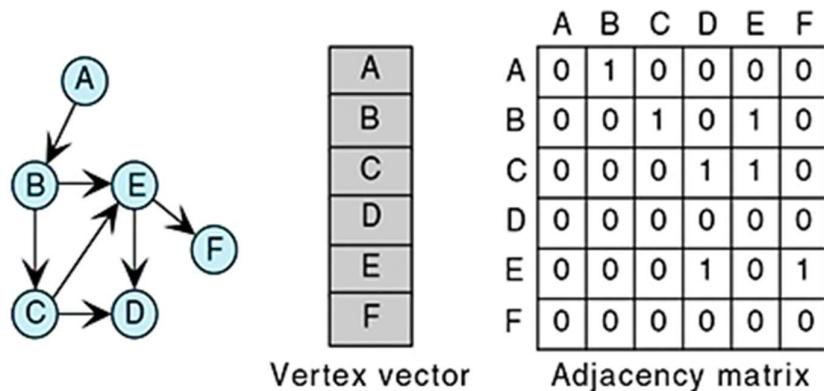
Adjacency Matrix Representation:

Having mapped the vertices to integers, one simple representation for the graph uses an **adjacency matrix**. Using a $|V| \times |V|$ matrix of booleans, we set $a_{ij} = \text{true}$ if an edge connects i and j . Edges can be *undirected*, in which case if $a_{ji} = \text{true}$, then $a_{ji} = \text{true}$ also, or **directed**, in which $a_{ij} \neq a_{ji}$, unless there are *two* edges, one in either direction, between i and j . The diagonal elements, a_{ii} , may be either ignored or, in cases such as state machines, where the presence or absence of a connection from a node to itself is relevant, set to **true** or **false** as required.

If the graph is *dense*, i.e. most of the nodes are connected by edges, then the $\mathbf{O}(|V|^2)$ cost of initialising an adjacency matrix is matched by the cost of inputting and setting the edges. However, if the graph is *sparse*, ie $|E|$ is closer to $|V|$, then an adjacency list representation may be more efficient.



(a) Adjacency matrix for non-directed graph

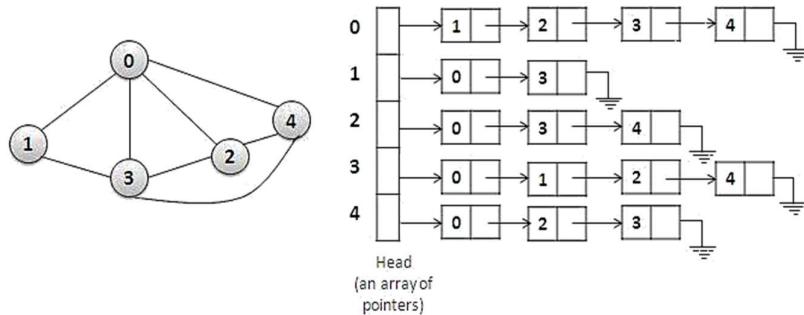


(a) Adjacency matrix for directed graph

Adjacency List Representation:

Adjacency lists are lists of nodes that are connected to a given node. For each node, a linked list of nodes connected to it can be set up. Adding an edge to a graph will generate two entries in adjacency lists - one in the lists for each of its extremities.

The space complexity of adjacency list is $O(V + E)$ because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.



10.2 Graph Traversals

Graph traversal means visiting every vertex and edge in a well-defined order. While using certain graph traversal algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited is important and may depend upon the algorithm or question that you are solving.

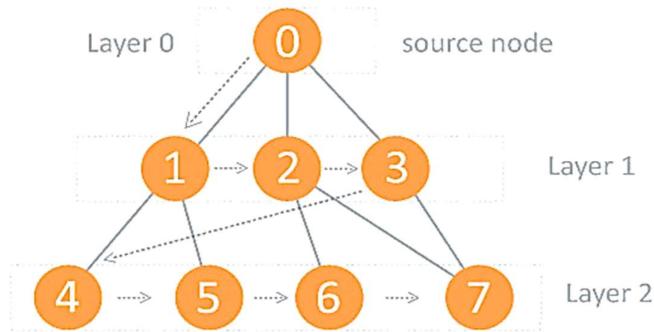
During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them by using boolean array.

Breadth First Search (BFS):

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the graph by discovering the neighbouring nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbouring nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes:

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7).

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First-In First-Out (FIFO) queuing method, and therefore, the neighbours of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Algorithm:

```
BFS(G, s):
    parent[s] = null
    state[s] = "discovered"
    i = 1;
    level[s] = i-1;
    Queue Q = {s}
    while Q is not empty do
        u = deQueue(Q)
        // process vertex 'u' as desired
        for each vertex belonging to Adj[u] do
            // process edge (u, v) as desired
            if state[vertex] = "undiscovered" then
                level[vertex] = i
                state[vertex] = "discovered"
                parent[vertex] = u
                enQueue (Q, vertex)
        i++
```

We are given a graph G in form of adjacency list and a starting vertex s. The code starts with some following initializations:

Parent of vertex s is null as vertex s is the starting vertex. Maintaining this parent list is optional in BFS, but this will give some cool structure later on. Once we have touched a vertex, we should set its state as ‘This vertex has been visited or discovered’, so that we shouldn’t get stuck in a loop of processing the processed nodes again and again.

Next is the level list. This list will store the info that - in how many steps from the starting vertex, we will reach to the desired node. So, we reach vertex ‘s’ in zero steps because we have started from there.

Like tree’s level order traversal, we will use queue for BFS. We will now start with vertex ‘u’. For every vertex u, we will visit its adjacent neighbours. If the adjacent vertex has not been visited, we will enqueue this vertex, set its state, level and parent. At the end of this loop, we will increment ‘i’ so as to increment the level for the next nodes.

The parent list actually forms a tree, where if we continue to find the parent of nodes, we will end up at root ‘s’ in the shortest path.

Below is the implementation of BFS for both – Adjacency list as well as adjacency matrix.

For Adjacency List:

```
#include<bits/stdc++.h>
using namespace std;

class Graph {
    int vertices;
    list<int>*> adj; // can use vector<int>* as well
public:
    Graph(int V);
    void addEdge(int u, int v);
    void BFS(int s);
};

Graph :: Graph (int V) {
    this->vertices = V; adj = new list<int>[V];
}

void Graph :: addEdge (int u, int v) {
    adj[u].push_back(v); adj[v].push_back(u);
}

void Graph :: BFS (int s) {
    int* parent = new int[vertices];
    int* level = new int[vertices];
    bool* visited = new bool[vertices];
    for (int i = 0; i < vertices; i++)
        visited[i] = false;

    visited[s] = true;
    parent[s] = -1;
    int i = 1; level[s] = i-1;
    queue<int> q; q.push(s);
    cout << "BFS Traversal is:\n";
    while (!q.empty()) {
        int u = q.front();
        cout << u << " ";
        q.pop();
        for (auto itr = adj[u].begin(); itr!=adj[u].end(); itr++)
            if (!visited[*itr]) {
                visited[*itr] = true;
                level[*itr] = i;
                parent[*itr] = u;
                q.push(*itr);
            }
        i++;
    }
    cout << endl;
}
```

```

cout << "Level of each node is:\n";
for (int i = 0; i < vertices; i++)
    cout << level[i] << " ";
cout << endl;

cout << "Parent of each node is:\n";
for (int i = 0; i < vertices; i++)
    cout << parent[i] << " ";
cout << endl;
}

int main() {
    int vertices, edges;
    cin >> vertices >> edges;
    Graph g(vertices);
    int a, b;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        g.addEdge(a, b);
    }
    g.BFS(0);
    return 0;
}

```

For Adjacency Matrix:

```

#include<bits/stdc++.h>
using namespace std;

void printBFS (int** edges, int n, int start_vertex, bool* visited) {
    queue<int> pending_vertices;
    pending_vertices.push(0);
    visited[start_vertex] = true;
    while (!pending_vertices.empty()) {
        int current_vertex = pending_vertices.front();
        pending_vertices.pop();
        cout << current_vertex << " ";
        for (int i = 0; i < n; i++) {
            if (i == current_vertex)
                continue;
            if (edges[current_vertex][i] == 1 && !visited[i]) {
                pending_vertices.push(i); visited[i] = true;
            }
        }
    }
    cout << endl;
}

```

```

void BFS(int** edges, int n) {
    bool* visited = new bool[n]();
    for (int i = 0; i < n; i++)
        if (!visited[i])
            printBFS(edges, n, i, visited);
    delete[] visited;
}

int main() {
    int vertices, edges;
    cin >> vertices >> edges;
    int** graph = new int*[vertices];
    for (int i = 0; i < vertices; i++)
        graph[i] = new int[vertices];
    int a, b;
    for (int i = 0; i < edges; i++) {
        cin >> a >> b;
        graph[a][b] = 1;
        graph[b][a] = 1;
    }
    BFS(graph, vertices);
    for (int i = 0; i < vertices; i++)
        delete[] graph[i];
    delete[] graph;
    return 0;
}

```

Assume an adjacency list representation, V is the number of vertices, E the number of edges. Each vertex is enqueued and dequeued at most once. Scanning for all adjacent vertices takes $O(|E|)$ time, since sum of lengths of adjacency lists is $|E|$. Hence, the time complexity of BFS is $O(|V| + |E|)$.

If it is an adjacency matrix, then it will be $O(V^2)$.

Depth First Search (DFS):

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

The basic idea is as follows:

- a) Pick a starting node and push all its adjacent nodes into a stack.
- b) Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- c) Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

DFSvisit() is the basic depth-first-search which will visit all the vertices reachable from a given source vertex s. If the graph is undirected, DFSvisit() alone can reach all the nodes. But in case of directed graphs, we may not visit all the vertices.

```
DFS-visit(G, s): // Recursive
    visited[s] = true
    for vertex v in Adj[s]:
        if (!visited[v]):
            DFS-visit (G, v);
```

```
DFS-visit (G, s): // Iterative
    stack S
    S.push(s);
    visited[s] = true;
    while stack is not empty:
        vertex v = S.top()
        S.pop()
        for all vertices w in Adj[v]:
            if (!visited[w]):
                S.push(w)
                visited[w] = true
```

The next part is the DFS which will visit the entire graph and not just the vertices reachable from S. This is helpful in directed graphs. This DFS function uses DFSvisit() as utility function.

```
DFS(G):
    no vertex has been visited yet
    for each vertex s in Vertices:
        if (!visited[s]):
            visited[s] = true
            DFS-visit(G, s)
```

For adjacency list:

```
void Graph :: DFS_visit(int s, bool* visited) {
    stack <int> S;
    S.push(s); visited[s] = true;
    while (!S.empty()) {
        int v = S.top(); S.pop();
        cout << v << " ";
        for (auto itr = adj[v].begin(); itr != adj[v].end(); itr++)
            if (!visited[*itr]) {
                S.push(*itr);
                visited[*itr] = true;
            }
    }
    cout << endl;
}

void Graph :: DFS() {
    bool* visited = new bool[vertices];
    for (int i = 0; i < vertices; i++)
        visited[i] = false;
    for (int i = 0; i < vertices; i++)
        if (!visited[i])
            DFS_visit(i, visited);
}
```

For adjacency matrix:

```
void printDFS (int** edges, int n, int start_vertex, bool* visited) {
    cout << start_vertex << " ";
    visited[start_vertex] = true;
    for (int i = 0; i < n; i++) {
        if (i == start_vertex)
            continue;
        if (edges[start_vertex][i] == 1 && !visited[i])
            printDFS(edges, n, i, visited);
    }
}

void DFS (int** edges, int n) {
    bool* visited = new bool[n]();
    for (int i = 0; i < n; i++)
        if (!visited[i])
            printDFS(edges, n, i, visited);
    cout << endl;
    delete[] visited;
}
```

If your graph is implemented as an adjacency matrix, then for each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges. So, the complexity of DFS is $O(V \times V) = O(V^2)$.

If your graph is implemented using adjacency lists, wherein each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbours by traversing its adjacency list just once in linear time. For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E (total number of edges). So, the complexity of DFS is $O(V) + O(E) = O(V + E)$. For an undirected graph, each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be $O(V) + O(2E) \sim O(V + E)$.

10.3 Path between two nodes

Let's say we want to find – if there is a path between two vertices in a directed graph or not. The idea is very simple – use BFS!

For adjacency list:

```
// s represents source and d represents destination.
bool Graph :: isReachable (int s, int d) {
    if (s == d)
        return true;
    bool *visited = new bool[vertices];
    for (int i = 0; i < vertices; i++)
        visited[i] = false;
    queue<int> q;
    q.push(s);
    visited[s] = true;
    while (!q.empty()) {
        s = q.front();
        q.pop();
        for (auto itr = adj[s].begin(); itr != adj[s].end(); itr++) {
            if (*itr == d)
                return true;
            if (!visited[*itr]) {
                visited[*itr] = true;
                q.push(*itr);
            }
        }
    }
    return false;
}
```

For adjacency matrix:

```
bool isReachable(int** edges, int n, int s, int d) {
    if (s == d)
        return true;
    bool hasRoute = false;
    bool visited[n+1] = {false};
    queue<int> vertices;
    visited[s] = true;
    vertices.push(s);
    while (!vertices.empty() && !hasRoute) {
        int current = vertices.front();
        vertices.pop();
        for (int i = 0; i < n; i++) {
            if (edges[current][i] != 0 && !visited[i]) {
                if (i == d) {
                    hasRoute = true;
                    break;
                }
                visited[i] = true;
                vertices.push(i);
            }
        }
    }
    return hasRoute;
}
```

Problems

Q. 1 Above code just tells whether a path between two nodes exists or not. Your task is to actually print that path. Try to solve above problem using both – BFS and DFS. (Assume you are working on adjacency matrix).

Using DFS:

```
bool getPath (int** adj, int n, int v1, int v2, bool* visited,
              vector<int> &v) {
    v.push_back(v1); visited[v1] = true;
    if (v1 == v2)    return true;
    for (int i = 0; i < n; i++)
        if (!visited[i] && adj[v1][i] == 1)
            if (getPath(adj, n, i , v2, visited, v))
                return true;
    v.pop_back();
    return false;
}
```

```

int main() {
    int V, E;
    cin >> V >> E;
    int** adj = new int*[V];
    for (int i = 0; i < V; i++) {
        adj[i] = new int[V];
        for (int j = 0; j < V; j++)
            adj[i][j] = 0;
    }
    int a, b;
    for (int i = 0; i < E; i++) {
        cin >> a >> b;
        adj[a][b] = 1;
        adj[b][a] = 1;
    }
    int v1, v2;
    cin >> v1 >> v2;
    vector<int> v;
    bool* visited = new bool[V]();
    if (path(adj, V, v1, v2, visited, v)){
        for (int i = 0; i < v.size(); i++)
            cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Using BFS:

```

bool getPath (int** adj, int n, int s, int d, bool* vis, int* parent) {
    queue<int> q;
    q.push(s);
    vis[s] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == d)
            return true;
        for (int i = 0; i < n; i++)
            if (adj[u][i] == 1 && !vis[i]) {
                q.push(i);
                parent[i] = u;
                vis[i] = true;
            }
    }
    return false;
}

```

```

int main() {
    int N, E;
    cin >> N >> E;
    int** adj = new int*[N];
    for (int i = 0; i < N; i++) {
        adj[i] = new int[N];
        for (int j = 0; j < N; j++)
            adj[i][j] = 0;
    }
    int a, b;
    while (E--) {
        cin >> a >> b;
        adj[a][b] = 1;
        adj[b][a] = 1;
    }
    int v1, v2;
    cin >> v1 >> v2;
    bool* visited = new bool[N]();
    int* parent = new int[N];
    for (int i = 0; i < N; i++)
        parent[i] = -1;
    if (getPath(adj, N, v1, v2, visited, parent)) {
        /* The parent array points from destination to source
           i.e. in reverse direction. So, we will store reverse
           path in vector and print it in reverse way, to get
           proper s to d path. */
        vector<int> path;
        int crawl = v2;
        path.push_back(crawl);
        while (parent[crawl] != -1) {
            path.push_back(parent[crawl]);
            crawl = parent[crawl];
        }
        for (int i = path.size()-1; i >= 0; i--)
            cout << path[i] << " ";
        cout << endl;
    }
    return 0;
}

```

One follow-up question here:

Above code just prints **a path** from one vertex to another. Write a code which prints all the paths possible from one vertex to another. Try using both – DFS and BFS. (Assume this time you are working on adjacency lists).

Using DFS:

```
void Graph :: printAllPathsUtil (int s, int d, bool visited[],  
                                int path[], int& path_index) {  
    visited[s] = true;  
    path[path_index] = s; path_index++;  
    if (s == d) {  
        for (int i = 0; i < path_index; i++)  
            cout << path[i] << " ";  
        cout << endl;  
    }  
    else {  
        for (auto itr = adj[s].begin(); itr != adj[s].end(); itr++)  
            if (!visited[*itr])  
                printAllPathsUtil(*itr, d, visited, path, path_index);  
    }  
    path_index--;  
    visited[s] = false;  
}  
  
void Graph :: printAllPaths (int s, int d) {  
    bool* visited = new bool[vertices];  
    for (int i = 0; i < vertices; i++)  
        visited[i] = false;  
    // An array to store paths  
    int* path = new int[vertices];  
    // Initializing path[] as empty  
    int path_index = 0;  
    // Call the recursive utility function  
    printAllPathsUtil(s, d, visited, path, path_index);  
}
```

Using BFS:

1. Create a queue of vectors which will store path(s).
2. Initialise the queue with first path starting from 'src'.
3. Now run a loop till queue is not empty
 - Get the frontmost path from queue
 - Check if the lastnode of this path is destination
 - a) If true then print the path
 - Run a loop for all the vertices connected to the current vertex
 - i.e. lastnode extracted from path
 - If the vertex is not visited in current path
 - a) create a new path from earlier path and append this vertex
 - b) insert this new path to queue

```

bool isNotVisited(int x, vector<int>& path) {
    int size = path.size();
    for (int i = 0; i < size; i++)
        if (path[i] == x)
            return false;
    return true;
}

void findPaths (vector<vector<int>> graph, int n, int src, int dest) {
    // Create a queue which stores paths
    queue<vector<int>> q;
    // Vector to store current path
    vector<int> path;
    path.push_back(src);
    q.push(path);
    while (!q.empty()) {
        path = q.front(); q.pop();
        int last = path[path.size()-1];
        // If the last vertex is the desired destination
        if (last == dest)
            printPath(path);
        // Traverse all the nodes connected to the
        // current vertex and push new path to queue
        for (int i = 0; i < graph[last].size(); i++) {
            if (isNotVisited(graph[last][i], path)) {
                vector<int> newpath(path);
                newpath.push_back(graph[last][i]);
                q.push(newpath);
            }
        }
    }
}

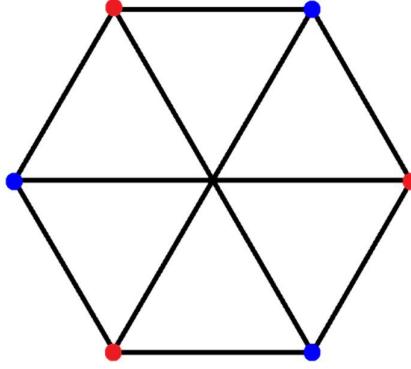
```

10.4 Bipartite Graph

A Bipartite graph is a graph whose vertices can be divided into two independent sets, U and V, such that every edge connects a vertex from U to V or a vertex from V to U. In other words, no edge connects the vertices belonging to same set.

A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.

Note: A graph represented by hexagon is bipartite but a graph represented by pentagon is not bipartite.



Algorithm:

We will consider two colors: 0(RED) and 1(BLUE). The visited array will be an integer array, with every element initialized as -1. While we visit every node, we will assign that node either RED or BLUE.

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbours with BLUE color (putting into set V).
3. Color all neighbour's neighbour with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbour which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite.

Checking Bipartiteness of graph using BFS:

```
bool isBipartiteUtil (int** graph, int n, int src, int* color) {
    color[src] = 1;
    queue<int> q; q.push(src);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        // Return false if there is a self loop
        if (graph[u][u] == 1)
            return false;
        // Find all the non-colored adjacent vertices
        for (int v = 0; v < n; v++)
            if (graph[u][v] == 1 && color[v] == -1) {
                // Assign alternate color to this adjacent v of node u
                color[v] = 1 - color[u];
                q.push(v);
            }
    }
}
```

```

        // An edge from u to v exists and destination v
        // is colored with the same color as u
        else if (graph[u][v]==1 && color[v]==color[u])
            return false;
    }
    // All vertices are colored with alternate colors
    return true;
}

bool isBipartite (int** graph, int n) {
    int color[n] = {-1};
    for (int i = 0; i < n; i++)
        if (color[i] == -1)
            if (!isBipartiteUtil(graph, n, i, color))
                return false;
    return true;
}

```

Checking Bipartiteness of graph using DFS:

```

bool isBipartite (vector<int> adj[], int v, bool visited[], int color[]){
    for (int u : adj[v]) {
        if (!visited[u]) {
            visited[u] = false;
            color[u] = !color[v];
            if (!isBipartite(adj, u, visited, color))
                return false;
        }
        else if (color[u] == color[v])
            return false;
    }
    return true;
}

```

10.5 Flood-Fill Algorithm

Flood fill algorithm helps in visiting each and every point in a given grid. It determines the area connected to a given cell in a multi-dimensional array. Following are some famous implementations of flood fill algorithm:

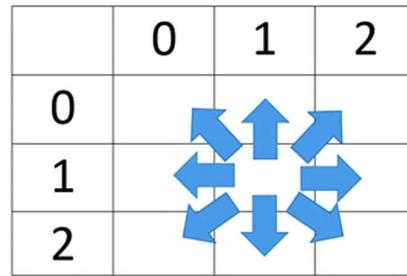
Solving a Maze:

Given a matrix with some starting point and some destination, with some obstacles in between, this algorithm helps in finding out the path from source to destination.

Minesweeper:

When a blank cell is discovered, this algorithm helps in revealing neighbouring cells. This step is done recursively till cells having numbers are discovered.

Flood fill algorithm can be simply modelled as graph traversal problem, representing the given area as a matrix and considering every cell of that matrix as a vertex that is connected to points above it, below it, to right of it, and to left of it and in case of 8-connections, to the points at both diagonals also. For example, consider the image given below.



It clearly shows how the cell in the middle is connected to cells around it. For instance, there are 8-connections like there are in Minesweeper (clicking on any cell that turns out to be blank reveals 8 cells around it which contains a number or are blank). In general, any cell (x, y) is connected to $(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1, y-1), (x+1, y), (x+1, y+1)$. Of course, boundary conditions are to be kept in mind. This area now can be modelled as a graph. For (x, y) cell, all possible 8 neighbours are its adjacent cells. So DFS or BFS can be used to traverse the graph.

The pseudocode for DFS is:

```
function DFS(x, y, visited, n, m):
    if (x ≥ n OR y ≥ m)
        return
    if (x < 0 OR y < 0)
        return
    if (visited[x][y] == True)
        return
    visited[x][y] = True
    DFS(x-1, y-1, visited, n, m)
    DFS(x-1, y, visited, n, m)
    DFS(x-1, y+1, visited, n, m)
    DFS(x, y-1, visited, n, m)
    DFS(x, y+1, visited, n, m)
    DFS(x+1, y-1, visited, n, m)
    DFS(x+1, y, visited, n, m)
    DFS(x+1, y+1, visited, n, m)
```

The above code visits each and every cell of matrix of size n x m.

Time Complexity: $O(nm)$

One important use of Flood Fill Algorithm is to solve a matrix maze. Let's implement the algorithm below by solving the following question:

A maze is represented by a matrix of size N x M, where rows are numbered from 1 to N and columns are numbered from 1 to M. Each cell of a matrix can be either 0 or 1. If a cell is 0, that cell cannot be visited and if it is 1, it can be visited. Allowed moves are U(up), D(down), L(left), R(right). Find a path from cell (1, 1) to (N, M) given that the source and destination cells always have value 1. If there is no path, print -1.

```
bool solveMazeUtil(int** maze, bool** visited, int n, int m, int currX,
int currY, stack<char> &route) {
    if (currX == n-1 && currY == m-1)
        return true;
    if (currX < 0 || currX >= n || currY < 0 || currY >= m) {
        if (!route.empty())
            route.pop();
        return false;
    }
    if (visited[currX][currY] || (maze[currX][currY] == 0)) {
        if (!route.empty())
            route.pop();
        return false;
    }
    visited[currX][currY] = true;
    if (solveMazeUtil(maze, visited, n, m, currX+1, currY, route)) {
        route.push('D');
        return true;
    }
    if (solveMazeUtil(maze, visited, n, m, currX-1, currY, route)) {
        route.push('U');
        return true;
    }
    if (solveMazeUtil(maze, visited, n, m, currX, currY+1, route)) {
        route.push('R');
        return true;
    }
    if (solveMazeUtil(maze, visited, n, m, currX, currY-1, route)) {
        route.push('L');
        return true;
    }
    return false;
}
```

```

string solveMaze (int** maze, int n, int m) {
    stack<char> route;
    bool** visited = new bool*[n];
    for (int i = 0; i < n; i++)
        visited[i] = new bool[m]();
    solveMazeUtil (maze, visited, n, m, 0, 0, route);
    string ans = "";
    while (!route.empty()) {
        ans += route.top();
        route.pop();
    }
    return ans;
}

int main() {
    int N, M;
    cin >> N >> M;
    int** maze = new int*[N];
    for (int i = 0; i < N; i++)
        maze[i] = new int[M];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            cin >> maze[i][j];

    string path = solveMaze (maze, N, M);
    if (path.empty())
        cout << -1 << endl;
    else
        cout << path << endl;
    return 0;
}

```

Problems

Q. 1 Connecting dots problem. You have a board of size $N \times M$. Each cell in the board has a colored dot. There are only 26 colors denoted by A, B, ..., Z. Now you need to find a cycle that contains dots of same color. Formally, we call a sequence of dots d_1, d_2, \dots, d_k a cycle if and only if it meets the following condition:

1. All the k dots are different and k is at least 4.
2. All dots belong to same color.
3. For all i , $1 \leq i \leq k-1$, d_i and d_{i+1} are adjacent. Cells x and y are adjacent if they share an edge.

Input:

Line 1: Two integers N and M

Next N lines: A string consisting of M characters

Output:

Return 1 if there is a cycle else return 0

Constraints:

$2 \leq N, M \leq 50$

```
#define MAX 51
bool visited[MAX][MAX];
int cycleFound = 0;
int dx[] = {1, -1, 0, 0};
int dy[] = {0, 0, 1, -1};

void dfs(char board[][][MAX], int n, int m, int x, int y, int fromX,
         int fromY, char colorNeeded) {
    if (x < 0 || x >= n || y < 0 || y >= m) return;
    if (board[x][y] != colorNeeded) return;
    if (visited[x][y]) {
        cycleFound = 1; return;
    }
    visited[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int nextX = x + dx[i];
        int nextY = y + dy[i];
        if (nextX == fromX && nextY == fromY)
            continue;
        dfs(board, n, m, nextX, nextY, x, y, colorNeeded);
    }
}

int connectDots(char board[][][MAX], int n, int m) {
    memset(visited, false, sizeof(visited));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (!visited[i][j])
                dfs(board, n, m, i, j, 0, 0, board[i][j]);
    return cycleFound;
}

int main() {
    int N, M, i;
    char board[MAX][MAX];
    cin >> N >> M;
    for(i = 0; i < N; i++)
        cin >> board[i];
    cout << connectDots(board,N,M) << endl;
    return 0;
}
```

Q. 2 Largest Island Problem. Given a 2D square matrix of size N that contains 0s and 1s. 1 represents land and 0 represents water. Two lands with 1s belong to same island, if they share an edge with each other. Find the size of the largest island.

Input:

Line 1: An integer N denoting the size of graph

Next N lines: N characters denoting the graph

Output:

Size of the biggest piece of '1's and no '0's

Constraints:

$1 \leq N \leq 50$

```
#define MAX 51

int dfs (char graph[MAX] [MAX], int n, bool** visited, int i, int j, int
count) {
    visited[i] [j] = true;
    if (i > 0 && graph[i-1] [j] == '1' && !visited[i-1] [j])
        count = dfs (graph, n, visited, i-1, j, count) + 1;
    if (j < n-1 && graph[i] [j+1] == '1' && !visited[i] [j+1])
        count = dfs (graph, n, visited, i, j+1, count) + 1;
    if (i < n-1 && graph[i+1] [j] == '1' && !visited[i+1] [j])
        count = dfs (graph, n, visited, i+1, j, count) + 1;
    if (j > 0 && graph[i] [j-1] == '1' && !visited[i] [j-1])
        count = dfs (graph, n, visited, i, j-1, count) + 1;
    return count;
}

int largestIsland(char graph[MAX] [MAX], int n) {
    bool** visited = new bool*[n];
    for (int i = 0; i < n; i++)
        visited[i] = new bool[n]();
    int max = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            if (!visited[i] [j] && graph[i] [j] == '1') {
                int count = 0;
                int n = dfs(graph, n, visited, i, j, count) + 1;
                if (n > max)
                    max = n;
            }
            for (int k = 0; k < n; k++)
                for (int m = 0; m < n; m++)
                    visited[k] [m] = false;
        }
    return max;
}
```

```

int main() {
    char graph[MAX][MAX];
    int n;
    cin >> n;
    for(int i=0; i<n; i++)
        scanf("%s", graph[i]);
    cout << largestIsland(graph, n) << endl;
    return 0;
}

```

Q. 3 Stepping Numbers problem. Given two integers 'n' and 'm', find all the stepping numbers in range [n, m]. A number is called stepping number if all adjacent digits have an absolute difference of 1. For e.g. 321 is a stepping number while 421 is not. Note: All single digit numbers are stepping numbers.

Solution:

Approach 1: (Brute Force)

A brute force approach is used to iterate through all the integers from n to m and check if it's a stepping number. Call the following function for each number in range.

```

bool isStepNum(int n) {
    int prevDigit = -1;
    while (n) {
        int currDigit = n%10;
        if (prevDigit == -1)
            prevDigit = currDigit;
        else
            if (abs(prevDigit - currDigit) != 1)
                return false;
        prevDigit = currDigit; n /= 10;
    }
    return true;
}

```

Approach 2: (Using BFS)

At first it looks strange that this problem could be solved using BFS. So, let us see - how to build the graph first?

Every node in the graph represents a stepping number; there will be a directed edge from a node U to V if V can be transformed from U.

'lastDigit' refers to the last digit of U (i.e. $U \% 10$)

An adjacent number V can be:

- $U * 10 + \text{lastDigit} + 1$ (Neighbour A)
- $U * 10 + \text{lastDigit} - 1$ (Neighbour B)

Therefore, every Node will have at most 2 neighbouring nodes.

Edge Cases: When the last digit of U is 0 or 9

Case 1: lastDigit is 0: In this case only digit 1 can be appended.

Case 2: lastDigit is 9: In this case only digit 8 can be appended.

What will be the source/startng Node?

Every single digit number is considered as a stepping number, so BFS traversal for every digit will give all the stepping numbers starting from that digit.

Note: For node 0, no need to explore neighbours during BFS traversal since it will lead to 01, 012, 010 and these will be covered by the BFS traversal starting from node 1.

Below is the implementation of approach 2.

```
void bfs(int n, int m, int num) {  
    // Queue will contain all the stepping numbers  
    queue<int> q; q.push(num);  
    while (!q.empty()) {  
        // Get the front element and pop from the queue  
        int stepNum = q.front(); q.pop();  
        // If the stepping number is in the range [n, m] then display  
        if (stepNum <= m && stepNum >= n)  
            cout << stepNum << " ";  
        // If stepping number is 0 or greater than m, need to  
        // explore the neighbors  
        if (num == 0 || stepNum > m) continue;  
        // Get the last digit of the currently visited stepping number  
        int lastDigit = stepNum % 10;  
        // There can be 2 cases either digit to be appended is  
        // lastDigit+1 or lastDigit-1  
        int stepNumA = stepNum * 10 + (lastDigit - 1);  
        int stepNumB = stepNum * 10 + (lastDigit + 1);  
        // If lastDigit is 0 then only possible digit after 0 can be  
        // 1 for a stepping number  
        if (lastDigit == 0) q.push(stepNumB);  
        // If lastDigit is 9 then only possible digit after 9 can be  
        // 8 for a stepping number  
        else if (lastDigit == 9) q.push(stepNumA);  
        else {  
            q.push(stepNumA);  
            q.push(stepNumB);  
        }  
    }  
}
```

```

void displaySteppingNumbers(int n, int m) {
    for (int i = 0 ; i <= 9 ; i++)
        bfs(n, m, i);
    cout << endl;
}

```

We can also solve above problem using DFS.

```

void dfs(int n, int m, int stepNum) {
    // If Stepping Number is in the range [n,m] then display
    if (stepNum <= m && stepNum >= n)
        cout << stepNum << " ";
    // If Stepping Number is 0 or greater than m, then return
    if (stepNum == 0 || stepNum > m)
        return;
    // Get the last digit of the currently visited Stepping Number
    int lastDigit = stepNum % 10;
    // 2 cases either digit to be appended is lastDigit+1 or lastDigit-1
    int stepNumA = stepNum*10 + (lastDigit-1);
    int stepNumB = stepNum*10 + (lastDigit+1);
    // If lastDigit is 0 then only possible digit after 0 can be 1
    if (lastDigit == 0)
        dfs(n, m, stepNumB);
    // If lastDigit is 9 then only possible digit after 9 can be 8
    else if (lastDigit == 9)
        dfs(n, m, stepNumA);
    else {
        dfs(n, m, stepNumA);
        dfs(n, m, stepNumB);
    }
}

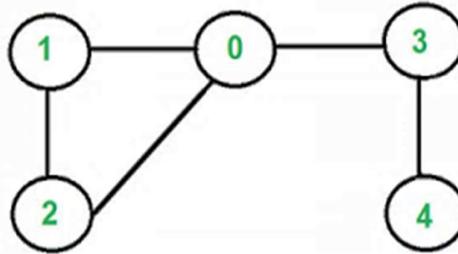
void displaySteppingNumbers(int n, int m) {
    for (int i = 0 ; i <= 9 ; i++)
        dfs(n, m, i);
    cout << endl;
}

```

10.6 Cycles in Graph

We will first look at the undirected graphs.

The following graph has a cycle 1-0-2-1.



We will use a DFS-alike function 'hasCycle()' that will call a utility function 'hasCycleUtil()'. hasCycleUtil() uses visited[] and parent to detect cycle in a subgraph reachable from vertex v. We recur the hasCycleUtil() function for all the vertices adjacent to vertex v. The main idea then is:

If an adjacent vertex is visited and that adjacent vertex is not the parent of the current vertex, then there is cycle.

Time Complexity: $O(V + E)$

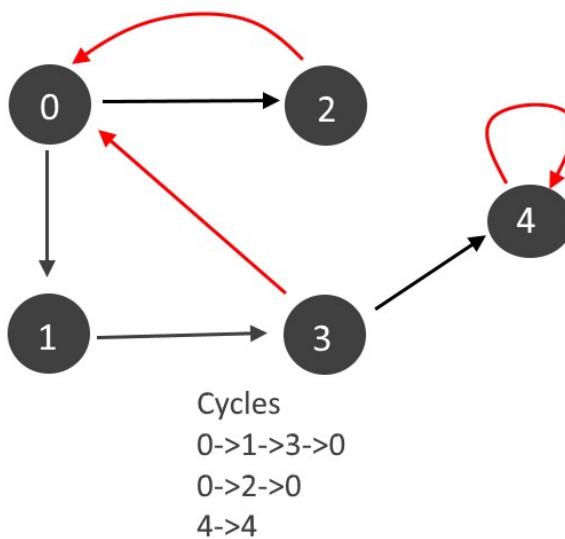
```
bool hasCycleUtil (vector<int> *adj, bool* visited, int v, int parent) {
    visited[v] = true;
    for (auto i = adj[v].begin(); i != adj[v].end(); i++)
        if (!visited[*i]){
            if (hasCycleUtil(adj, visited, *i, v))
                return true;
        }
        else if (*i != parent)
            return true;
    return false;
}

bool hasCycle (vector<int>* adj, int n) {
    bool* visited = new bool[n]();
    for (int i = 0; i < n; i++)
        if (!visited[i])
            if (hasCycleUtil(adj, visited, i, -1))
                return true;
    return false;
}
```

In case of directed graph, there is a cycle in a graph, only if there is a 'back edge' present in the graph. A back edge is an edge that is from a node to itself (self-loop) or through one of its ancestors in the tree produced by DFS.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We have used `recStack[]` array to keep track of vertices in the recursion stack.

In the following image, the edges in red color represent backedges.



Time Complexity: $O(V + E)$

```

bool hasCycleUtil (vector<int> *adj, bool* visited, bool* recStack,
                   int n, int v) {
    if (!visited[v]) {
        visited[v] = true;
        recStack[v] = true;
        for (auto i = adj[v].begin(); i != adj[v].end(); i++)
            if (!visited[*i] && hasCycleUtil(adj, visited, recStack, n,
                                              *i))
                return true;
            else if (recStack[*i])
                return true;
    }
    recStack[v] = false;
    return false;
}

```

```

bool hasCycle(vector<int>* adj, int n) {
    bool* visited = new bool[n]();
    bool* recStack = new bool[n]();
    for (int i = 0; i < n; i++)
        if (hasCycleUtil(adj, visited, recStack, n, i))
            return true;
    return false;
}

```

Problems

Q. 1 Given an undirected and connected graph and a number n, count total number of cycles of length n in the graph.

Solution:

Using DFS, we can find every possible path of length $(n - 1)$ for a particular source (or starting point). Then we check if this path ends with the vertex it started with, if yes then we count this as the cycle of length n. Notice that we looked for path of length $(n - 1)$ because the n^{th} edge will be the closing edge of cycle.

Every possible path of length $(n - 1)$ can be searched using only $V - (n - 1)$ vertices (where V is the total number of vertices).

For example, consider the following graph:

$V = 5$ and $E = 6$

Edges are: 0 - 1, 1 - 2, 2 - 3, 3 - 4, 3 - 0 and 4 - 1.

Total cycles of length 4 = 3. The cycles are:

```

0 -> 1 -> 2 -> 3 -> 0
0 -> 1 -> 4 -> 3 -> 0
1 -> 2 -> 3 -> 4 -> 1

```

For above example, all the cycles of length 4 can be searched using only $5 - (4 - 1) = 2$ vertices. The reason behind this is quite simple, because we search for all possible path of length $(n - 1) = 3$ using these 2 vertices which include the remaining 3 vertices. So, these 2 vertices cover the cycles of remaining 3 vertices as well, and using only 3 vertices we can't form a cycle of length 4 anyways.

One more thing to notice is that, every vertex finds 2 duplicate cycles for every cycle that it forms. For above example 0th vertex finds two duplicate cycle namely $0 -> 3 -> 2 -> 1 -> 0$ and $0 -> 1 -> 2 -> 3 -> 0$. Hence the total count must be divided by 2 because every cycle is counted twice.

```

void dfs (vector<int>* adj, bool* visited, int n, int length, int vert,
int start, int &count) {
    visited[vert] = true;
    // If a path of length 'length' is found
    if (length == 0){
        // Mark it false so that it becomes usable again
        visited[vert] = false;
        // Check if vertex 'vert' can end with vertex 'start'
        bool contains_start = false;
        for (int i = 0; i < adj[vert].size(); i++)
            if (adj[vert][i] == start)
                contains_start = true;
        if (contains_start) {
            count++;
            return;
        }
        else
            return;
    }
    // If the path of length 'length' is not found,
    // search for every other possible path of length 'length-1'
    for (int i = 0; i < adj[vert].size(); i++)
        if (!visited[adj[vert][i]])
            dfs(adj, visited, n, length-1, adj[vert][i], start, count);
    // mark vert as unvisited to make it usable again
    visited[vert] = false;
}

int countCycles(vector<int>* adj, int n, int length) {
    bool* visited = new bool[n]();
    int count = 0;
    for (int i = 0; i < n-(length-1); i++) {
        dfs(adj, visited, n, length-1, i, i, count);
        visited[i] = true;
    }
    return count/2;
}

```

Q. 2 Given a graph, the task is to find if it has a cycle of odd length or not.

Solution: The idea is based on an important fact that a graph does not contain a cycle of odd length if and only if it is Bipartite, i.e., it can be colored with two colors. It is obvious that if a graph has an odd length cycle then it cannot be Bipartite. In Bipartite graph there are two sets of vertices such that no vertex in a set is connected with any other vertex of the same set). For a cycle of odd length, two vertices must of the same set be connected which contradicts Bipartite definition.

```

bool containsOdd (vector<int>* graph, int n) {
    int color[n] = {-1};
    // Assign first color to source, here 0 is source
    color[0] = 1;
    // Create a queue for BFS traversal
    queue<int> q;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        // Return true if there is a self loop
        for (int i = 0; i < graph[u].size(); i++)
            if (graph[u][i] == u)
                return true;
        // For all non colored adjacent vertices
        for (int i = 0; i < n; i++) {
            for (auto itr = graph[i].begin(); itr != graph[i].end();
                 itr++)
            if (color[*itr] == -1) {
                color[*itr] = 1 - color[u];
                q.push(*itr);
            }
            else if (color[*itr] == color[u]) {
                return true;
            }
        }
    }
    return false;
}

```

Q. 3 Given a weighted and undirected graph, we need to find if a cycle exists in this graph such that the sum of weights of all the edges in that cycle comes out to be odd.

Examples:

Input: Number of vertices, n = 4,

Number of edges, m = 4

Weighted Edges =

1 2 2

2 3 1

4 3 1

4 1 2

Output: No! There is no odd weight cycle in the given graph

Input: Number of vertices, $n = 5$,

Number of edges, $m = 3$

Weighted Edges =

1 2 1

3 2 1

3 1 1

Output: Yes! There is an odd weight cycle in the given graph

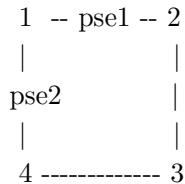
Solution:

The solution is based on the fact that – ‘If a graph has no odd length cycle then it must be Bipartite, i.e., it can be colored with two colors.’

The idea is to convert given problem to a simpler problem where we have to just check if there is cycle of odd length or not. To convert, we do the following:

- Convert all even weight edges into two edges of unit weight.
- Convert all odd weight edges to a single edge of unit weight.

For example, graph in Example 1) can be visualized as:



Here, edges [1 — 2] have been broken into two parts such that [1-pseudo1-2] a pseudo node has been introduced. We are doing this so that each of our even weighted edge is taken into consideration twice while the edge with odd weight is counted only once. Doing this would help us further when we color our cycle.

We assign weight 1 to all the edges and then traverse the whole graph by using 2 color method. Now we start coloring our modified graph using two colors only. In a cycle with even number of nodes, when we color it using two colors only, none of the two adjacent edges have the same color. While if we try coloring a cycle having odd number of edges, surely a situation will arise where two adjacent edges will have the same color. This is our pick! Thus, if we are able to color the modified graph completely using 2 colors only in a way no two adjacent edges get the same color assigned to them then there must be either no cycle in the graph or a cycle with even number of nodes. If any conflict arises while coloring a cycle with 2 colors only, then we have an odd cycle in our graph.

```

// This function returns true if the current subpart of the forest is
// two colorable, else false.
bool twoColorUtil(vector<int>G[], int src, int N, int colorArr[]) {
    // Assign first color to source
    colorArr[src] = 1;
    // Create a queue (FIFO) of vertex numbers and enqueue source vertex
    // for BFS traversal
    queue <int> q;
    q.push(src);
    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        // Find all non-colored adjacent vertices
        for (int v = 0; v < G[u].size(); ++v){
            // An edge from u to v exists and destination v is not
            // colored
            if (colorArr[G[u][v]] == -1) {
                // Assign alternate color to this adjacent v of u
                colorArr[G[u][v]] = 1 - colorArr[u];
                q.push(G[u][v]);
            }
            // An edge from u to v exists and destination v is colored
            // with same color as u
            else if (colorArr[G[u][v]] == colorArr[u])
                return false;
        }
    }
    return true;
}

// This function returns true if graph G[V][V] is two colorable,
// else false
bool twoColor(vector<int>G[], int N) {
    // Create a color array to store colors assigned to all veritces.
    // Vertex number is used as index in this array. The value '-1' of
    // colorArr[i] is used to indicate that no color is assigned to
    // vertex 'i'. The value 1 is used to indicate first color is
    // assigned and value 0 indicates second color is assigned.
    int colorArr[N];
    for (int i = 1; i <= N; ++i)
        colorArr[i] = -1;

    // As we are dealing with graph, the input might come as a forest,
    // thus, start coloring from a node and if true is returned we'll
    // know that we successfully colored the subpart of our forest and
    // we start coloring again from a new uncolored node.
}

```

```

    // This way we cover the entire forest.
    for (int i = 1; i <= N; i++)
        if (colorArr[i] == -1)
            if (twoColorUtil(G, i, N, colorArr) == false)
                return false;
    return true;
}

// Returns false if an odd cycle is present else true
// int info[][] is the information about our graph
// int n is the number of nodes
// int m is the number of informations given to us
bool isOddSum(int info[][], int n, int m) {
    // Declaring adjacency list of a graph
    // Here at max, we can encounter all the edges with
    // even weight thus there will be 1 pseudo node for each edge
    vector<int> G[2*n];
    int pseudo = n+1;
    int pseudo_count = 0;
    for (int i=0; i<m; i++) {
        // For odd weight edges, we directly add it in our graph
        if (info[i][2]%2 == 1) {
            int u = info[i][0];
            int v = info[i][1];
            G[u].push_back(v);
            G[v].push_back(u);
        }
        // For even weight edges, we break it
        else {
            int u = info[i][0];
            int v = info[i][1];
            // Entering a pseudo node between u---v
            G[u].push_back(pseudo);
            G[pseudo].push_back(u);
            G[v].push_back(pseudo);
            G[pseudo].push_back(v);
            // Keeping a record of number of pseudo nodes inserted
            pseudo_count++;
            // Making a new pseudo node for next time
            pseudo++;
        }
    }
    // We pass number graph G[][] and total number of node =
    // actual number of nodes + number of pseudo nodes added.
    return twoColor(G,n+pseudo_count);
}

```

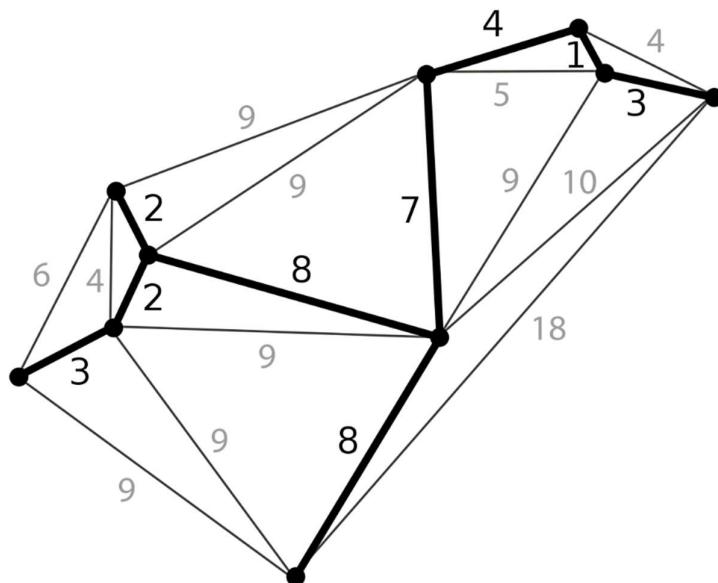
```

int main() {
    // 'n' correspond to number of nodes in our graph while
    // 'm' correspond to the number of information about this graph.
    int n = 4, m = 3;
    int info[4][3] = {{1, 2, 12},
                      {2, 3, 1},
                      {4, 3, 1},
                      {4, 1, 20}};
    // This function breaks the even weighted edges in two parts.
    // Makes the adjacency representation of the graph and sends it
    // for two coloring.
    if (isOddSum(info, n, m) == true)
        cout << "No\n";
    else
        cout << "Yes\n";
    return 0;
}

```

10.7 Minimum Spanning Trees

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.



There are quite a few use cases for minimum spanning trees. One example would be a telecommunications company trying to lay cable in a new neighbourhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

We will be covering 3 algorithms, which can find minimum spanning trees.

- a) Kruskal's Algorithm
- b) Prim's Algorithm
- c) Reverse Delete Algorithm

Kruskal's Algorithm:

Let's forget about MST for a moment and look at another problem – connectivity. Let's say, you have a set of N elements which are partitioned into further subsets, and you have to keep track of connectivity of each element in a particular subset or connectivity of subsets with each other. To do this operation efficiently, you can use Union-Find Data Structure.

Let's say there are 5 people A, B, C, D E. A is a friend of B, B is a friend of C and D is a friend of E. As we can see:

- 1) A, B and C are connected to each other.
- 2) D and E are connected to each other.

So, we can use Union-Find Data Structure to check whether one friend is connected to another in a direct or indirect way or not. We can also determine the two different disconnected subsets. Here 2 different subsets are {A, B, C} and {D, E}.

You have to perform two operations here:

- a) Union (A, B) - Connect two elements A and B.
- b) Find (A, B) - Check if Arr[A] is equal to Arr[B] or not.

Example: You have a set of elements S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Here you have 10 elements (N = 10). Initially there are 10 subsets (0-9) and each subset has single element in it.

When each subset contains only single element, the array Arr is:

Arr: 0 1 2 3 4 5 6 7 8 9

Index: 0 1 2 3 4 5 6 7 8 9

Whenever we come to know, objects A and B are connected, we do $\text{Arr}[A] = \text{Arr}[B]$.

i) Union(2, 1)

Arr: 0 1 1 3 4 5 6 7 8 9

Index: 0 1 2 3 4 5 6 7 8 9

ii) Union(4, 3), Union(8, 4), Union(9, 3).

Arr: 0 1 1 3 3 5 6 7 3 3

Index: 0 1 2 3 4 5 6 7 8 9

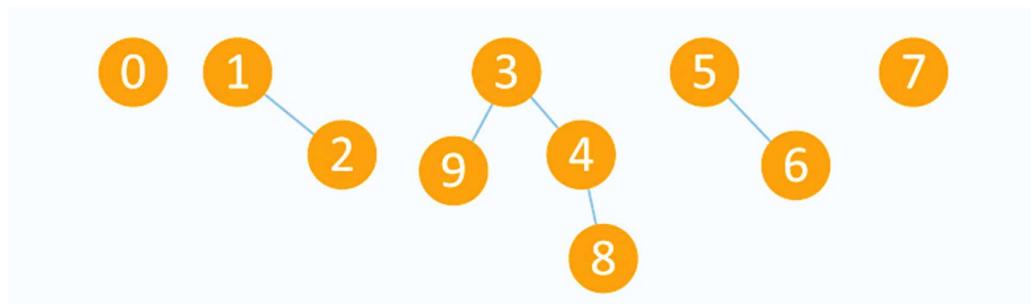
iii) 5) Union(6, 5)

Arr: 0 1 1 3 3 5 5 7 3 3

Index: 0 1 2 3 4 5 6 7 8 9

After performing some operations of $\text{Union}(A, B)$, you can see that now there are 5 subsets. First has elements {3, 4, 8, 9}, second has {1, 2}, third has {5, 6}, fourth has {0} and fifth has {7}. All these subsets are said to be 'Connected Components'.

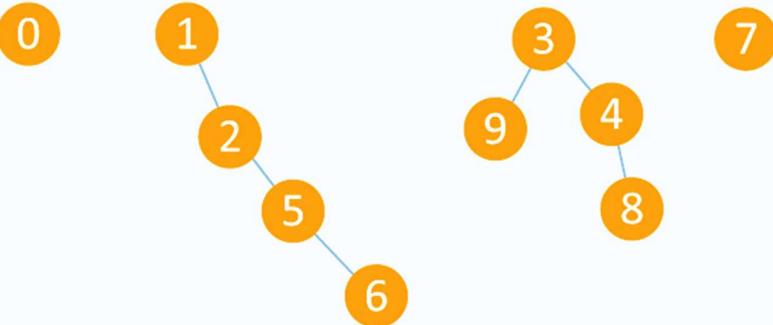
One can also relate these elements with nodes of a graph. The elements in one subset can be considered as the nodes of the graph which are connected to each other directly or indirectly, therefore each subset can be considered as connected component.



Let's perform some Find(A, B) operations.

- i) Find (0, 7) - As 0 and 7 are disconnected, this will give false result.
- ii) Find (8, 9) - Though 8 and 9 are not connected directly, but there exists a path connecting 8 and 9, so it will give us true result.

Note: If we perform operation Union(5, 2) on above components, then it will be :



Arr: 0 1 1 3 3 1 1 7 3 3

Index: 0 1 2 3 4 5 6 7 8 9

IMPLEMENTATION 1: (Naive Union and Find implementation)

Initially there are N subsets containing single element in each subset. So to initialize array we will use initialize() function.

```
void initialize(int Arr[], int N) {  
    for(int i = 0; i < N; i++)  
        Arr[i] = i;  
}  
  
// Following function returns true if A and B are connected  
bool find(int Arr[], int A, int B) {  
    if(Arr[A] == Arr[B]) return true;  
    else return false;  
}  
  
// Change all entries from Arr[A] to Arr[B].  
void union(int Arr[], int N, int A, int B) {  
    int temp = Arr[A];  
    for(int i = 0; i < N; i++) {  
        if(Arr[i] == temp)  
            Arr[i] = Arr[B];  
    }  
}
```

As loop in Union function iterates through all the N elements for connecting two elements. So, performing this operation on N objects will take $O(N^2)$ time, which is quite inefficient.

Here is another approach:

We can consider a root element of each subset, which is the only special element in that subset having itself as the parent. Let's say if R is a root element, then

$$\text{Arr}[R] = R.$$

To make it clearer, let's take a subset $S = \{0, 1, 2, 3, 4, 5\}$.

Initially each element is the root of itself in all subsets, as $\text{Arr}[i] = i$, where i is element in the set, therefore $\text{root}(i) = i$.

Arr: 0 1 2 3 4 5

Index: 0 1 2 3 4 5

Performing $\text{Union}(1, 0)$ will connect 1 to 0 and will set $\text{root}(0)$ as the parent of $\text{root}(1)$. As $\text{root}(1) = 1$, and $\text{root}(0) = 0$, therefore value of $\text{Arr}[1]$ will be changed from 1 to 0. It will make 0 as a root of subset containing elements $\{0, 1\}$.

Arr: 0 0 2 3 4 5

Index: 0 1 2 3 4 5

Now performing $\text{Union}(0, 2)$, will indirectly connect 0 to 2, by setting $\text{root}(2)$ as the parent of $\text{root}(0)$. As $\text{root}(0)$ is 0 and $\text{root}(2)$ is 2, therefore it will change value $\text{Arr}[0]$ from 0 to 2. Now 2 will be the root of subset containing elements $\{2, 0, 1\}$.

Arr: 2 0 2 3 4 5

Index: 0 1 2 3 4 5

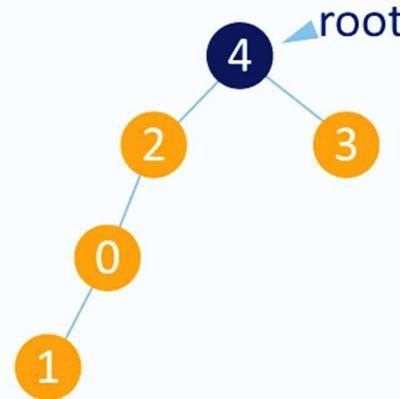
Similarly $\text{Union}(3, 4)$ will indirectly connect 3 to 4, by setting $\text{root}(4)$ as the parent of $\text{root}(3)$. As $\text{root}(3)$ is 3 and $\text{root}(4)$ is 4, therefore it will change value of $\text{Arr}[3]$ from 3 to 4. It will make 4 as a root of subset containing elements $\{3, 4\}$.

Arr: 2 0 2 4 4 5

Index: 0 1 2 3 4 5

Performing $\text{Union}(1, 4)$ will indirectly connect 1 to 4, by setting $\text{root}(4)$ as the parent of $\text{root}(1)$.

As $\text{root}(4)$ is 4 and $\text{root}(1)$ is 2, therefore it will change value of $\text{Arr}[2]$ from 2 to 4. It makes 4 as root of set containing elements $\{0, 1, 2, 3, 4\}$.



Arr	2	0	4	4	4	5
	0	1	2	3	4	5

Arr: 2 0 4 4 4 5

Index: 0 1 2 3 4 5

After performing required Union(A, B) operations, we can easily perform the Find(A, B) operation to check whether A and B are connected or not. It can be checked by calculating roots of both A and B. If roots of A and B are same, that means both A and B are in same subset and are connected.

How to calculate the root of an element?

As we know that $\text{Arr}[i]$ is the parent of i (where i is the element of set), then the root of i is $\text{Arr}[\text{Arr}[\text{Arr}[\dots \text{Arr}[i] \dots]]]$ until $\text{Arr}[i]$ is not equal to i .

Simply we can run a loop until we get an element which is a parent of itself.

Note: This can be only done when there is no cycle in the elements of subset, otherwise loop will run infinitely.

Find(1, 4) - 1 and 4 have same root as 4, therefore it means they are connected and this operation will give true as a result.

Find(3, 5) - 3 and 5 do not have same root, as $\text{root}(3)$ is 4 and $\text{root}(5)$ is 5. It means they are not connected and it will give false as a result.

IMPLEMENTATION 2: (Union-Find by finding parent)

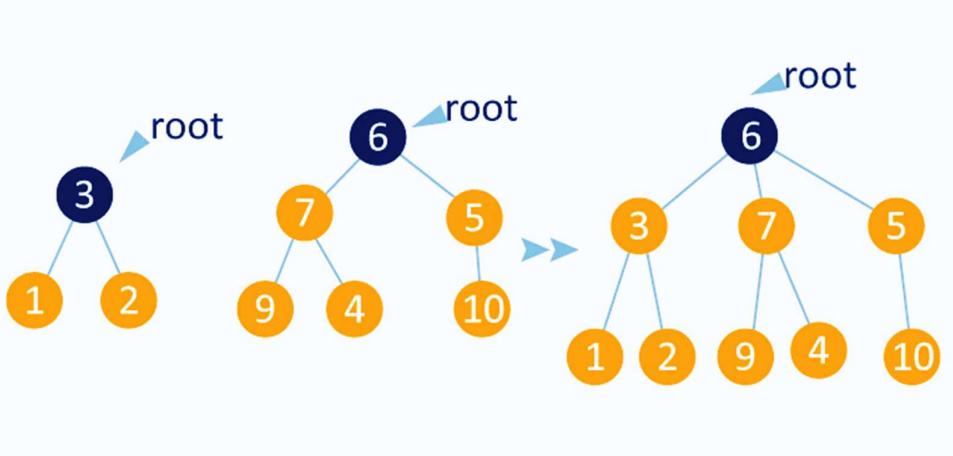
```
// finding root of an element.  
int root(int Arr[], int i) {  
    //chase parent of current element until it reaches root.  
    while (Arr[i] != i) {  
        i = Arr[i];  
    }  
    return i;  
}  
  
// modified union function where we connect the elements by  
// changing the root  
void union(int Arr[], int A, int B) {  
    int root_A = root(Arr, A);  
    int root_B = root(Arr, B);  
    //setting parent of root(A) as root(B).  
    Arr[root_A] = root_B;  
}  
  
bool find(int A, int B) {  
    // if A and B have same root, means they are connected.  
    if(root(A) == root(B))  
        return true;  
    else  
        return false;  
}
```

Now as you can see, in the worst case, this idea will also take linear time in connecting 2 elements and even in finding that if two elements are connected or not, it will take linear time.

Another disadvantage is that while connecting two elements, we do not check which subset has more element than other and sometimes it creates a big problem as in the worst case we have to perform approximately linear time operations.

We can avoid this, by keeping the track of size of each subset and then while connecting two elements, we can connect the root of subset having smaller number of elements to the root of subset having larger number of elements.

Consider the following subsets:



Here if we want to connect 1 and 5, then we will connect the root of Subset A (subset which contains 1) will be connected to root of Subset B (contains 5), this is because Subset A contains a smaller number of elements than of Subset B. It will balance the tree formed by the above operations. We call this operation as weighted_union operation.

IMPLEMENTATION 3: (Union-Find by Rank)

Initially the size of each subset will be one as each subset will have only one element and we can initialize it in the initialize function discussed above: size[] array will keep track of size of each subset.

```
// modified initialize function
void initialize(int Arr[], int N) {
    for(int i = 0; i < N; i++) {
        Arr[i] = i; size[i] = 1;
    }
}

// root() and find() function will be same as above.

// modified union function
void weighted-union(int Arr[], int size[], int A, int B) {
    int root_A = root(A), root_B = root(B);
    if(size[root_A] < size[root_B]) {
        Arr[root_A] = Arr[root_B];
        size[root_B] += size[root_A];
    }
    else {
        Arr[root_B] = Arr[root_A];
        size[root_A] += size[root_B];
    }
}
```

Take an example:

You have a set $S = \{0, 1, 2, 3, 4, 5\}$ Initially all the subsets have a single element and each element is a root of itself.

Size	1	1	1	1	1	1
Arr	0	1	2	3	4	5
Index	0	1	2	3	4	5

Perform Union(0, 1). Here we can connect any root of any element with root of other one as both the element's subsets have same size and then we will update the respective size. If we connect 1 to 0 and make 0 as a root and then size of 0 will change from 1 to 2.

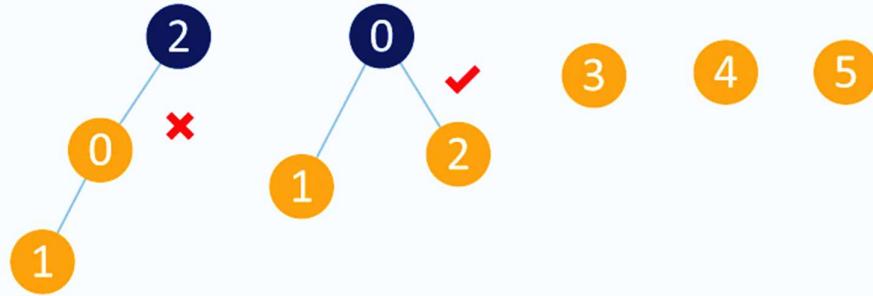
Size	2	1	1	1	1	1
Arr	0	0	2	3	4	5
Index	0	1	2	3	4	5



size	2	1	1	1	1	1
	0	1	2	3	4	5
Arr	0	0	2	3	4	5
	0	1	2	3	4	5

While performing Union(1, 2), we will connect root(2) with root(1) as subset of 2 has less number of elements than number of elements in subset of 1.

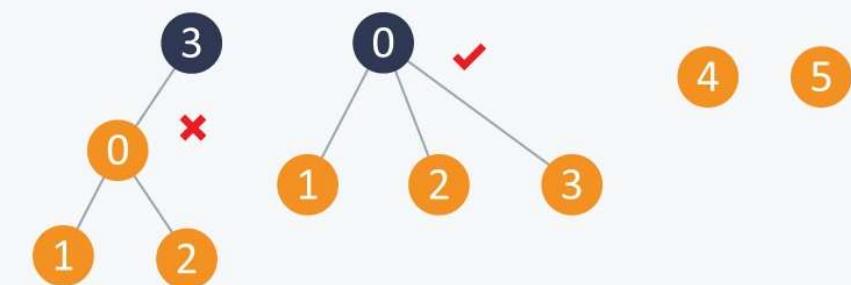
Size	3	1	1	1	1	1
Arr	0	0	0	3	4	5
Index	0	1	2	3	4	5



size	3	1	1	1	1	1
	0	1	2	3	4	5
Arr	0	0	0	3	4	5
	0	1	2	3	4	5

Similarly in Union(3, 2), it will connect root(3) to root(2) as subset of 3 has less number of element than number of elements in subset of 2.

Size 4 1 1 1 1 1
 Arr 0 0 0 0 4 5
 Index 0 1 2 3 4 5



size	4	1	1	1	1	1
	0	1	2	3	4	5
Arr	0	0	0	0	4	5
	0	1	2	3	4	5

Maintaining a balance tree will reduce complexity of union and find function from N to $\log_2 N$.

Can we improve more?

While computing the root of A, set each i to point to its grandparent (thereby halving the path length), where i is the node which comes in between path, while computing root of A.

IMPLEMENTATION 4: (Union Find Algo with Path Compression)

```
// modified root function.  
int root (int Arr[], int i) {  
    while(Arr[i] != i) {  
        Arr[i] = Arr[Arr[i]];  
        i = Arr[i];  
    }  
    return i;  
}
```

When we use Weighted-union with path compression it takes $\log^* N$ for each union find operation, where N is the number of elements in the set.

$$\log^* N \Rightarrow \log_2(\log_2 N)$$

$\log^* N$ is much better than $\log(N)$, as its value reaches at most up to 5 in the real world.

Union-Find is used to determine the connected components in a graph. We can determine whether 2 nodes are in the same connected component or not in the graph. We can also determine that by adding an edge between 2 nodes whether it leads to cycle in the graph or not. Let's look at an example – how to find cycles using Union-Find data structure.

```
struct Edge {  
    int src, dest;  
};  
  
// Wrapper Class for Graph  
class Graph {  
public:  
    vector<vector<int>> adjList;  
    Graph(vector<Edge> const &edges, int N) {  
        adjList.resize(N+1);  
        // Add edges to the graph  
        for (auto &edge: edges)  
            adjList[edge.src].push_back(edge.dest);  
    }  
};
```

```

// Wrapper Class to represent Disjoint Set
class DisjointSet {
unordered_map <int, int> parent;
public:
    // Create N disjoint sets, one for each vertex
    void initialize(int N) {
        for (int i = 1; i <= N; i++)
            parent[i] = i;
    }

    // Function that returns root of element in the set
    int root (int i) {
        while(parent[i] != i) {
            parent[i] = parent[parent[i]];
            i = parent[i];
        }
        return i;
    }

    // Find method
    bool find(int A, int B) {
        if(root(A) == root(B))
            return true;
        else
            return false;
    }

    // Union method
    void union_func(int A, int B) {
        int root_A = root(A);
        int root_B = root(B);
        parent[root_A] = root_B;
    }
};

// function that returns true if cycle is found
bool findCycle(Graph const &graph, int N) {
    DisjointSet ds;
    ds.initialize(N);
    // Consider every edge (u, v)
    for (int u = 1; u <= N; u++) {
        // For all adjacent verticies,
        for (int v: graph.adjList[u]) {
            // Find root of the sets to which elements u and v belong
            int x = ds.root(u);
            int y = ds.root(v);

```

```

        // If both u and v have same parent, cycle is found
        if (x == y)
            return true;
        else
            ds.union_func(x, y);
    }
}

return false;
}

int main() {
    // vector of graph edges as per above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 7}, {1, 8}, {2, 3}, {2, 6}, {3, 4},
        {3, 5}, {8, 9}, {8, 12}, {9, 10}, {9, 11}, {11, 12}
        // edge (11, 12) introduces a cycle in the graph
    };
    // Number of nodes in the graph
    int N = 12;
    // create a graph from edges
    Graph graph(edges, N);
    if (findCycle(graph, N))
        cout << "Cycle Found\n";
    else
        cout << "No Cycle Found\n";
    return 0;
}

```

Now, let's come back to our discussion on Minimum Spanning Trees.

How many edges does a minimum spanning tree has?

An MST has $(V - 1)$ edges where V is the number of vertices in the given graph.

Below are the steps for finding Minimum Spanning Tree using Kruskal's Algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are $(V-1)$ edges in the spanning tree.

```

#include <bits/stdc++.h>
using namespace std;

class Edge {
public:
    int src, dest, weight;
};

```

```

bool compare (Edge e1, Edge e2) {
    return e1.weight < e2.weight;
}

// This is kindof similar to root() method in disjoint set union algo,
// but it's implemented recursively.
int getParent (int v, int* parent) {
    if (parent[v] == v)
        return v;
    else
        return getParent(parent[v], parent);
}

Edge* kruskal (Edge* edges, int n, int E) {
    sort (edges, edges + E, compare);
    Edge* output = new Edge[n-1];
    int* parent = new int[n];
    for (int i = 0; i < n; i++)
        parent[i] = i;
    int count = 0, i = 0;
    while (count < n-1) {
        Edge currentEdge = edges[i];
        int srcParent = getParent (currentEdge.src, parent);
        int destParent = getParent (currentEdge.dest, parent);
        if (srcParent != destParent) {
            output[count] = currentEdge;
            parent[srcParent] = destParent; count++;
        }
        i++;
    }
    delete[] parent;
    return output;
}

int main() {
    int n, E; cin >> n >> E;
    Edge* edges = new Edge[E];
    for (int i = 0; i < E; i++)
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    cout << "Edges in MST: " << endl;
    Edge* output = kruskal (edges, n, E);
    for (int i = 0; i < n-1; i++) {
        cout << output[i].src << " " << output[i].dest << " "
            << output[i].weight << endl;
    }
    return 0;
}

```

Following is the implementation of Kruskal's Algorithm on Adjacency matrix:

```
int getParent(int i, int* parent) {
    while(parent[i] != i) { // Get parent method (Iterative)
        parent[i] = parent[parent[i]]; i = parent[i];
    }
    return i;
}

void union_func(int i, int j, int* parent) {
    int a = getParent(i, parent), b = getParent(j, parent);
    parent[a] = b;
}

void kruskal (vector<vector<int>> cost, int V) {
    int* parent = new int[V];
    for (int i = 0; i < V; i++)
        parent[i] = i;
    int count = 0;
    while (count < V-1) {
        int min = INT_MAX, a = -1, b = -1;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (getParent(i, parent) != getParent(j, parent)
                    && cost[i][j] < min) {
                    min = cost[i][j]; a = i; b = j;
                }
            }
        }
        union_func(a, b, parent);
        printf("Edge %d:(%d, %d) cost:%d \n", count++, a, b, min);
    }
    delete[] parent;
}

int main() {
    int V = 5;
    vector<vector<int>> cost = {
        { INT_MAX, 2, INT_MAX, 6, INT_MAX },
        { 2, INT_MAX, 3, 8, 5 },
        { INT_MAX, 3, INT_MAX, INT_MAX, 7 },
        { 6, 8, INT_MAX, INT_MAX, 9 },
        { INT_MAX, 5, 7, 9, INT_MAX },
    };
    kruskal(cost, V);
    return 0;
}
```

In Kruskal's algorithm, the most time-consuming operation is sorting because the total complexity of the Disjoint-Set operations will be $O(E \log V)$, which is the overall Time Complexity of the algorithm.

Prim's Algorithm:

Prim's Algorithm also uses Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm:

- Maintain two disjoint sets of vertices. One containing the vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

Here is the implementation of Prim's Algorithm in case of adjacency matrix. First try to understand this implementation, and then look at the implementation of adjacency list.

```

int getMinVertex (bool* visited, int* weight, int n) {
    int minVertex = -1;
    for (int i = 0; i < n; i++)
        if (!visited[i] && ((minVertex == -1) ||
                               (weight[minVertex] > weight[i])))
            minVertex = i;
    return minVertex;
}

void prims (int** edges, int n) {
    bool* visited = new bool[n];
    int *parent = new int[n], *weight = new int[n];
    for (int i = 0; i < n; i++) {
        visited[i] = false;
        weight[i] = INT_MAX;
    }
}

```

```

parent[0] = -1; weight[0] = 0;
for (int i = 0; i < n-1; i++) {
    int minVertex = getMinVertex (visited, weight, n);
    visited[minVertex] = true;
    for (int j = 0; j < n; j++) {
        if (edges[minVertex][j] != 0 && !visited[j]) {
            if (weight[j] > edges[minVertex][j]) {
                weight[j] = edges[minVertex][j];
                parent[j] = minVertex;
            }
        }
    }
}
for (int i = 1; i < n; i++)
    cout << parent[i] << " " << i << " " << weight[i] << endl;
delete[] visited; delete[] parent; delete[] weight;
}

int main() {
    int n, e;
    cin >> n >> e;
    int** edges = new int*[n];
    for (int i = 0; i < n; i++)
        edges[i] = new int[n]();
    int f, s, weight;
    for (int i = 0; i < e; i++) {
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }
    prims (edges, n);
    for (int i = 0; i < n; i++)
        delete[] edges[i];
    delete[] edges;
    return 0;
}

```

Here is the implementation for adjacency list:

```

class Graph {
    int V;
    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    vector<pair<int, int>> *adj;

```

```

public:
    Graph(int V) {
        this->V = V; adj = new vector<pair<int, int>> [V];
    }
    void addEdge(int u, int v, int w) {
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w));
    }
    void prims();
};

void Graph::prims() {
    // Create a priority queue to store vertices that are
    // yet to be added in MST.
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>> pq;
    int src = 0; // Taking vertex 0 as source
    // Create a vector for keys and initialize all keys as
    // infinite (INT_MAX)
    vector<int> key(V, INT_MAX);
    // To store parent array which in turn store MST
    vector<int> parent(V, -1);
    // To keep track of vertices included in MST
    vector<bool> inMST(V, false);
    // Insert source itself in priority queue and initialize
    // its key as 0.
    pq.push(make_pair(0, src)); key[src] = 0;
    // Loop till priority queue becomes empty
    while (!pq.empty()) {
        // The first vertex in pair is the minimum key vertex,
        // get it from pq. Vertex label is stored in second of pair.
        // It has to be done this way to keep the vertices sorted key
        // (key must be first item in pair).
        int u = pq.top().second; pq.pop();
        inMST[u] = true; // Include vertex in MST
        // 'i' is used to get all adjacent vertices of a vertex
        vector<pair<int, int>>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i) {
            // Get vertex label and weight of current adjacent of u.
            int v = (*i).first, weight = (*i).second;
            // If v is not in MST and weight of (u,v) is
            // smaller than current key of v
            if (inMST[v] == false && key[v] > weight) {
                // Updating key of v
                key[v] = weight; pq.push(make_pair(key[v], v));
                parent[v] = u;
            }
        }
    }
}

```

```

// Print edges of MST using parent array
for (int i = 1; i < V; ++i)
    printf("%d - %d\n", parent[i], i);
}

int main() {
    int V = 9;
    Graph g(V);
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.prims();
    return 0;
}

```

The time complexity of the Prim's Algorithm is $O((V + E)\log V)$ because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

Additional Point:

Let's say - we want to find Minimum Product MST. A minimum product spanning tree for a weighted, connected and undirected graph is a spanning tree with weight product less than or equal to the weight product of every other spanning tree. This problem can be solved using standard minimum spanning tree algorithms like Kruskal's and prim's algorithm, but we need to modify our graph to use these algorithms. Minimum spanning tree algorithms tries to minimize total sum of weights, here we need to minimize total product of weights.

We can use property of logarithms to overcome this problem. As we know,

$$\log(w_1 * w_2 * w_3 * \dots * w_N) = \log(w_1) + \log(w_2) + \log(w_3) + \dots + \log(w_N)$$

We can replace each edge-weight of graph by its log value, then we apply any minimum spanning tree algorithm which will try to minimize sum of $\log(w_i)$ which in-turn minimizes weight product.

Reverse Delete Algorithm:

Reverse Delete algorithm is closely related to Kruskal's algorithm. In Kruskal's algorithm what we do is: Sort edges by increasing order of their weights. After sorting, we one by one pick edges in increasing order. We include current picked edge if by including this in spanning tree not form any cycle until there are $V-1$ edges in spanning tree.

In Reverse Delete algorithm, we sort all edges in decreasing order of their weights. After sorting, we one by one pick edges in decreasing order. We include current picked edge if excluding current edge causes disconnection in current graph. The main idea is to delete an edge if its deletion does not lead to disconnection of graph.

Algorithm:

- 1) Sort all edges of graph in non-increasing order of edge weights.
- 2) Initialize MST as original graph and remove extra edges using step 3.
- 3) Pick the highest weight edge from remaining edges and check if deleting the edge disconnects the graph or not.

If disconnects, then we don't delete the edge.

Else we delete the edge and continue.

```
class Edge {  
public:  
    int src, dest, weight;  
    Edge(int src, int dest, int weight) {  
        this->src = src;  
        this->dest = dest;  
        this->weight = weight;  
    }  
};  
  
bool compare(Edge e1, Edge e2) {  
    return e1.weight < e2.weight;  
}  
  
class Graph {  
    int V;  
    // Used list instead of vector, because we will  
    // need remove() method of list  
    list<int>* adjList;    // Will be used for checking connectivity  
    vector<Edge> edges;    // Will be used for sorting edges  
public:  
    Graph(int V) {  
        this->V = V;  
        adjList = new list<int>[V];  
    }  
}
```

```

void addEdge(int u, int v, int w) {
    adjList[u].push_back(v);
    adjList[v].push_back(u);
    edges.push_back(Edge(u, v, w));
}

void DFS (int v, bool* visited) {
    visited[v] = true;
    list<int> :: iterator i;
    for (i = adjList[v].begin(); i != adjList[v].end(); i++)
        if(!visited[*i])
            DFS(*i, visited);
}

bool isConnected() {
    bool* visited = new bool[V]();
    DFS(0, visited);
    for (int i = 1; i < V; i++)
        if (visited[i] == false)
            return false;
    return true;
}

void reverseDelete(); // Implemented below
};

void Graph::reverseDelete() {
    sort(edges.begin(), edges.end(), compare);
    // Traverse in descending order
    for (int i = edges.size() - 1; i >= 0; i--) {
        int u = edges[i].src;
        int v = edges[i].dest;
        int w = edges[i].weight;
        adjList[u].remove(v);
        adjList[v].remove(u);
        if (isConnected() == false) {
            adjList[u].push_back(v);
            adjList[v].push_back(u);
            // This edge is a part of MST
            printf("Edge (u = %d, v = %d, w = %d)\n", u, v, w);
        }
    }
}

```

```

int main() {
    int V = 9;
    Graph g(V);
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.reverseDelete();
    return 0;
}

```

10.8 Shortest Path Algorithms

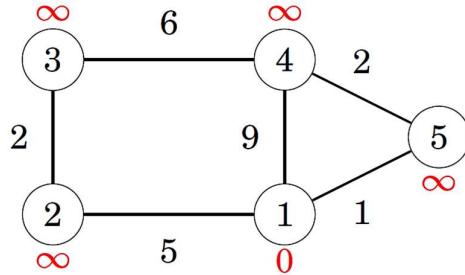
Finding the shortest path between two nodes of a graph is an important problem that has many practical applications. For example, a natural problem related to a road network is to calculate the shortest possible length of a route between two cities, given the lengths of the roads.

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find the shortest path. However, in this chapter we focus on weighted graphs where more sophisticated algorithms are needed for finding the shortest paths.

Dijkstra's Algorithm:

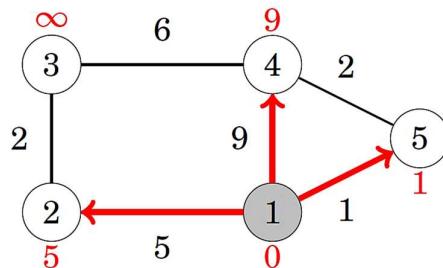
Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in the shortest path tree, other set includes vertices not yet included in the shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Let us consider how Dijkstra's algorithm works in the following graph when the starting node is node 1:



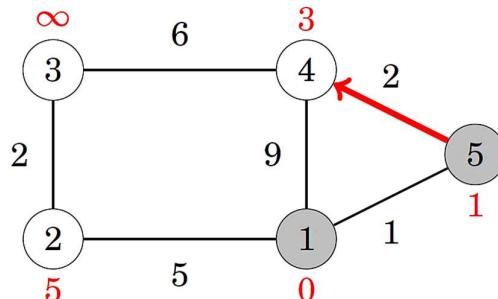
At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. The first such node is node 1 with distance 0.

When a node is selected, the algorithm goes through all edges that start at the node and reduces the distances using them:

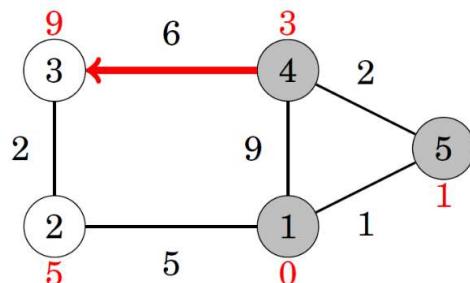


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

The next node to be processed is node 5 with distance 1. This reduces the distance to node 4 from 9 to 3:

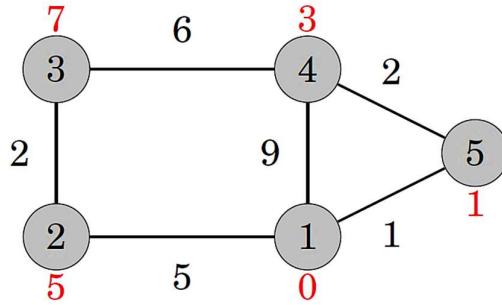


After this, the next node is node 4, which reduces the distance to node 3 to 9:



A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

After this, the algorithm processes the two remaining nodes, and the final distances are as follows:



Dijkstra's Implementation for Adjacency Matrix:

Algorithm:

- 1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as infinite. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While sptSet doesn't include all vertices
 - a) Pick a vertex u which is not there in sptSet and has minimum distance value.
 - b) Include u to sptSet.
 - c) Update distance value of all adjacent vertices of u . To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v , if sum of distance value of u (from source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v .

Time Complexity of Dijkstra's Algorithm is $O(V^2)$, but after using Fibonacci heaps (advanced topic, not covered here) it drops down to $O(E + V \log V)$. However, we will use this $O(V^2)$ algorithm.

```

int V, E;

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}

```

```

void printSolution(int dist[], int n) {
    for (int i = 0; i < V; i++)
        printf("%d %d\n", i, dist[i]);
}

void dijkstra(int** graph, int V) {
    int* dist = new int[V];
    bool* sptSet = new bool[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[0] = 0;
    for (int count = 0; count < V-1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
                dist[u]+graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist, V);
    delete[] dist, sptSet;
}

int main() {
    cin >> V >> E;
    int** edges = new int*[V];
    for (int i = 0; i < V; i++)
        edges[i] = new int[V]();
    int f, s, weight;
    for (int i = 0; i < E; i++) {
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }
    dijkstra (edges, V);
    for (int i = 0; i < V; i++)
        delete[] edges[i];
    delete[] edges;
    return 0;
}

```

Dijkstra's Implementation for Adjacency List:

Algorithm:

- 1) Initialize distances of all vertices as infinite.
- 2) Create an empty set. Every item of set is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs.
- 3) Insert source vertex into the set and make its distance as 0.
- 4) While Set doesn't become empty, do following
 - a) Extract minimum distance vertex from Set. Let the extracted vertex be u.
 - b) Loop through all adjacent of u and do following for every vertex v.
 - // If there is a shorter path to v through u.
 - If $\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)$
 - (i) Update distance of v, i.e., do $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$
 - (ii) If v is in set, update its distance in set by removing it first, then inserting with new distance
 - (iii) If v is not in set, then insert it in set with new distance.
- 5) Print distance array dist[] to print all the shortest paths.

```
class Graph {  
    int V;  
    vector<pair<int, int>> *adj; // because weighted graph, hence pair.  
public:  
    Graph(int V) {  
        this->V = V;  
        adj = new vector<pair<int, int>> [V];  
    }  
  
    void addEdge(int u, int v, int w) {  
        adj[u].push_back(make_pair(v, w));  
        adj[v].push_back(make_pair(u, w));  
    }  
  
    void dijkstra(int src); // Implemented below  
};  
  
void Graph::dijkstra(int src) {  
    // Create a set to store the vertices that are being processed.  
    set<pair<int, int>> vertices;  
    // Create a vector for distances and initialize them with INFINITY  
    vector<int> distances(V, INT_MAX);  
    // Insert source itself in set and initialize it's distance to 0  
    vertices.insert(make_pair(0, src));  
    distances[src] = 0;  
    // Loop till vertices are empty.
```

```

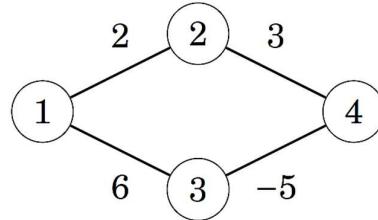
while (!vertices.empty()) {
    // The first vertex in Set is the minimum distance vertex.
    pair<int, int> tmp = *(vertices.begin());
    vertices.erase(vertices.begin());
    // Vertex label is stored in second of pair.
    // It has to be done this way to keep the vertices sorted as
    // per the distance. 'distance' must be first item in pair.
    int u = tmp.second;
    // 'i' is used to get all adjacent vertices of a vertex
    vector<pair<int, int>> :: iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        // Get vertex label and weight of current adjacent of u.
        int v = (*i).first;
        int weight = (*i).second;
        // If there is shorter path to v through u.
        if (distances[v] > distances[u] + weight) {
            // If distance of v is not INF then it must be in our set,
            // so removing it and inserting again with updated less
            // distance. Note: We extract only those vertices from
            // Set for which distance is finalized. So, for them,
            // we would never reach here.
            if (distances[v] != INT_MAX)
                vertices.erase(vertices.find(make_pair(distances[v], v)));
            // Updating distance of v
            distances[v] = distances[u] + weight;
            vertices.insert(make_pair(distances[v], v));
        }
    }
}
// Print shortest distances stored in dist[]
printf("Vertex  Distance from Source\n");
for (int i = 0; i < V; ++i)
    printf("%d \t\t %d\n", i, distances[i]);
}

int main() {
    int V = 9; Graph g(V);
    g.addEdge(0, 1, 4); g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8); g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7); g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4); g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14); g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2); g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6); g.addEdge(7, 8, 7);
    g.dijkstra(0);
    return 0;
}

```

Problem with Dijkstra's Algorithm:

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:



The shortest path from node 1 to node 4 is 1->3->4 and its length is 1. However, Dijkstra's algorithm finds the path 1->2->4 by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

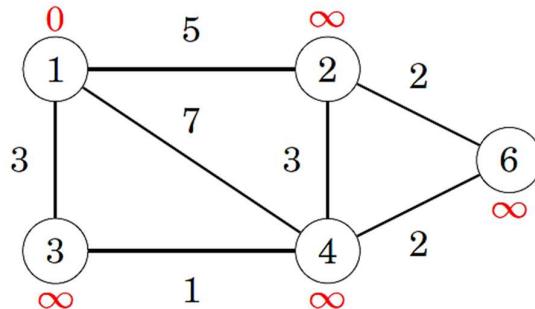
An algorithm which handles negative weights as well is Bellman-Ford Algorithm.

Bellman-Ford Algorithm:

The Bellman–Ford algorithm finds the shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

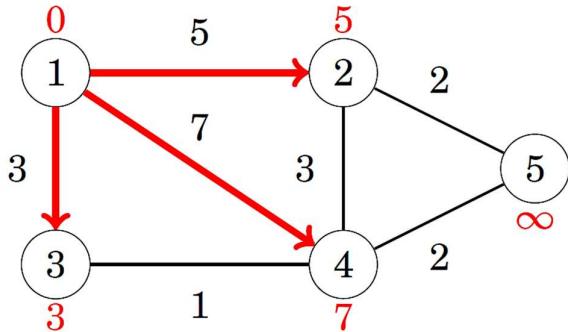
The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes is infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Let us consider how the Bellman–Ford algorithm works in the following graph:

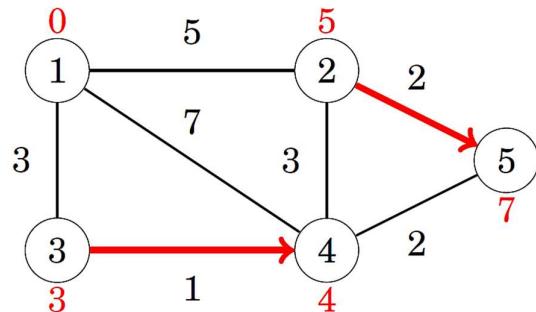


Each node of the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

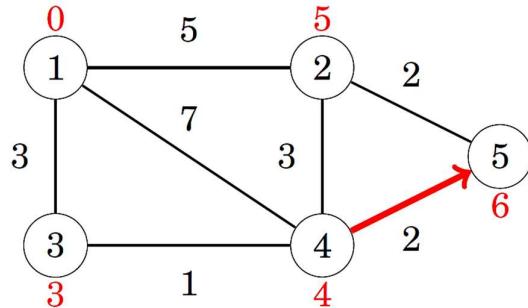
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



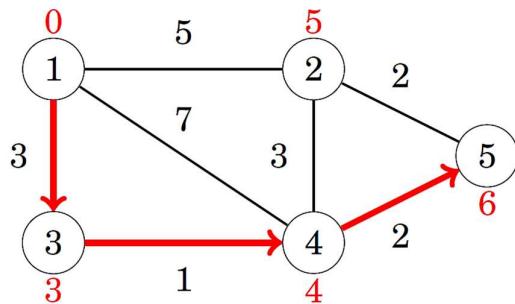
After this, edges 2->5 and 3->4 reduce distances:



Finally, there is one more change:

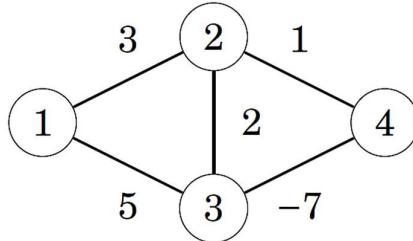


After this, no edge can reduce any distance. This means that the distances are final, and we have successfully calculated the shortest distances from the starting node to all nodes of the graph. For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



Negative Cycles:

The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ with length -4.

If the graph contains a negative cycle, we can shorten infinitely many times any path that contains the cycle by repeating the cycle again and again. Thus, the concept of the shortest path is not meaningful in this situation.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

Simple Implementation of Bellman-Ford:

Algorithm:

- 1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array $\text{dist}[]$ of size $|V|$ with all values as infinite except $\text{dist}[\text{src}]$ where src is the source vertex.
- 2) This step calculates shortest distances. Do following $|V|-1$ times where $|V|$ is the number of vertices in given graph.

- a) Do following for each edge $u-v$
If $\text{dist}[v] > \text{dist}[u] + \text{weight of edge } uv$, then update $\text{dist}[v]$
 $\text{dist}[v] = \text{dist}[u] + \text{weight of edge } uv$

- 3) This step reports if there is a negative weight cycle in graph.

- Do following for each edge $u-v$
If $\text{dist}[v] > \text{dist}[u] + \text{weight of edge } uv$, then
'Graph contains negative weight cycle'

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

Time complexity: $O(VE)$

```

void BellmanFord (int graph[][] [3], int V, int E, int src) {
    int* distance = new int[V];
    for (int i = 0; i < V; i++)
        distance[i] = INT_MAX;
    distance[src] = 0;
    for (int i = 0; i < V-1; i++) {
        for (int j = 0; j < E; j++) {
            if (distance[graph[j][0]] + graph[j][2] <
                distance[graph[j][1]])
                distance[graph[j][1]] = distance[graph[j][0]] +
                    graph[j][2];
        }
    }
    // Following step checks for negative cycles
    for (int i = 0; i < E; i++) {
        int x = graph[i][0];
        int y = graph[i][1];
        int weight = graph[i][2];
        if (distance[x] != INT_MAX && distance[x]+weight < distance[y])
            cout << "Graph contains negative weight cycle." << endl;
    }

    cout << "Vertex      Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << "\t\t" << distance[i] << endl;
}

int main() {
    int V = 5;
    int E = 8;
    int graph[][] [3] = { { 0, 1, -1 }, { 0, 2, 4 },
                        { 1, 2, 3 }, { 1, 3, 2 },
                        { 1, 4, 2 }, { 3, 2, 5 },
                        { 3, 1, 1 }, { 4, 3, -3 } };

    BellmanFord(graph, V, E, 0);
    return 0;
}

```

Implementation of Bellman-Ford for Adjacency List:

```
class Edge{
public:
    int src, dest, weight;
    Edge(int src, int dest, int weight) {
        this->src = src;
        this->dest = dest;
        this->weight = weight;
    }
};

class Graph {
    int V, E;
    vector<Edge> edges;
public:
    Graph(int V, int E) {
        this->V = V; this->E = E;
    }
    void addEdge(int u, int v, int w) {
        edges.push_back(Edge(u, v, w));
    }
    void BellmanFord(int src);
};

void Graph::BellmanFord(int src) {
    int* distance = new int[V];
    for (int i = 0; i < V; i++)
        distance[i] = INT_MAX;
    distance[src] = 0;
    for (int i = 1; i < V-1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j].src, v = edges[j].dest;
            int weight = edges[j].weight;
            if (distance[u] != INT_MAX && distance[u] + weight <
                distance[v])
                distance[v] = distance[u] + weight;
        }
    }
    for (int i = 0; i < E; i++) {
        int u = edges[i].src, v = edges[i].dest;
        int weight = edges[i].weight;
        if (distance[u] != INT_MAX && distance[u]+weight < distance[v]) {
            printf("Graph contains negative weight cycle.\n");
            return;
        }
    }
}
```

```

cout << "Vertex      Distance from Source" << endl;
for (int i = 0; i < V; i++)
    cout << i << "\t\t" << distance[i] << endl;
return;
}

int main() {
    int V = 5, E = 8;
    Graph g(V, E);
    g.addEdge(0, 1, -1); g.addEdge(0, 2, 4);
    g.addEdge(1, 2, 3);  g.addEdge(1, 3, 2);
    g.addEdge(1, 4, 2);  g.addEdge(3, 2, 5);
    g.addEdge(3, 1, 1);  g.addEdge(4, 3, -3);

    g.BellmanFord(0);
    return 0;
}

```

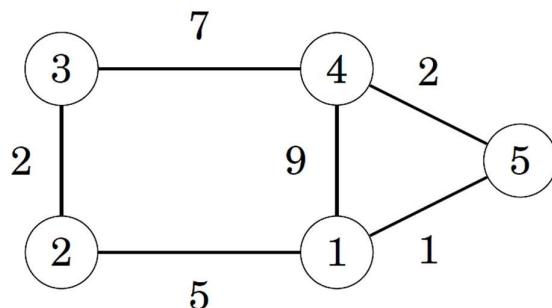
By now, we have studied two shortest path algorithms which find the shortest path from source to all the vertices in a graph. However, if we wish to find the shortest path between all the pairs of nodes in a graph, using above methods will be very much inefficient.

Floyd-Warshall Algorithm:

The Floyd–Warshall algorithm provides an alternative way to approach the problem of finding the shortest paths. Unlike the other algorithms of this chapter, it finds all the shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Let us consider how the Floyd–Warshall algorithm works in the following graph:



Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x. All other distances are infinite.

In this graph, the initial array is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

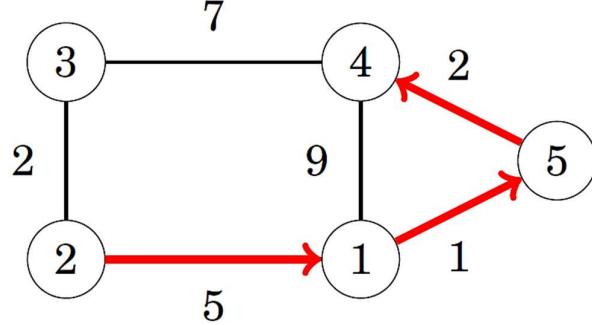
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

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

The algorithm continues like this, until all nodes have been appointed intermediate nodes. After the algorithm has finished, the array contains the minimum distances between any two nodes:

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

For example, the array tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:



The time complexity of the algorithm is $O(n^3)$, because it contains three nested loops that go through the nodes of the graph.

Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Implementation of Floyd Warshall Algorithm:

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, ... , k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- 1) k is not an intermediate vertex in shortest path from i to j.

We keep the value of $\text{dist}[i][j]$ as it is.

- 2) k is an intermediate vertex in shortest path from i to j.

We update the value of -

$$\text{dist}[i][j] = \text{dist}[i][k] + \text{dist}[k][j] \text{ if } \text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$$

```
// print path of given vertex u from source vertex v recursively
void printPath(int** path, int v, int u) {
    if (path[v][u] == v)
        return;
    printPath(path, v, path[v][u]);
    cout << path[v][u] << " ";
}

void FloydWarshall(vector<vector<int>> adjMatrix, int V) {

    int** cost = new int*[V];
    int** path = new int*[V];

    for (int i = 0; i < V; i++) {
        cost[i] = new int[V];
        path[i] = new int[V];

        // Initialize cost and path arrays now.
        // Initial cost would be same as weight.
        for (int j = 0; j < V; j++) {
            cost[i][j] = adjMatrix[i][j];

            if (i == j)
                path[i][j] = 0;
            else if (cost[i][j] != INT_MAX)
                path[i][j] = i;
            else
                path[i][j] = -1;
        }
    }
}
```

```

// Run Floyd Warshall Algo
for (int k = 0; k < V; k++) {
    for (int v = 0; v < V; v++) {
        for (int u = 0; u < V; u++) {
            // If vertex k is on the shortest path from v to u,
            // then update the value of cost[v][u], path[v][u]
            if (cost[v][k] != INT_MAX && cost[k][u] != INT_MAX
                && cost[v][k] + cost[k][u] < cost[v][u]) {
                cost[v][u] = cost[v][k] + cost[k][u];
                path[v][u] = path[k][u];
            }
        }
    }
    // if diagonal elements become negative, the
    // graph contains a negative weight cycle
    if (cost[v][v] < 0) {
        cout << "Negative Weight Cycle Found!!";
        return;
    }
}
}

// Printing solution
for (int v = 0; v < V; v++) {
    for (int u = 0; u < V; u++) {
        if (cost[v][u] == INT_MAX)
            cout << setw(5) << "inf";
        else
            cout << setw(5) << cost[v][u];
    }
    cout << endl;
}
cout << endl;
for (int v = 0; v < V; v++) {
    for (int u = 0; u < V; u++) {
        if (u != v && path[v][u] != -1) {
            cout << "Shortest Path from vertex " << v <<
                " to vertex " << u << " is (" << v << " ";
            printPath(path, v, u);
            cout << u << ")" << endl;
        }
    }
}
}

```

```

int main() {
    int V = 4;
    vector<vector<int>> adjMatrix {
        {0, INT_MAX, -2, INT_MAX},
        {4, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 2},
        {INT_MAX, -1, INT_MAX, 0},
    };
    FloydWarshall(adjMatrix, V);
    return 0;
}

```

Speeding up the Dijkstra (Advanced Topic):

Can we optimize Dijkstra's shortest path algorithm to work better than $O(E * \log V)$ if maximum weight is small (or range of edge weights is small)?

For example, in the above diagram, maximum weight is 14. Many times, the range of weights on edges in is in small range (i.e. all edge weight can be mapped to 0, 1, 2 ... w where w is a small number). In that case, Dijkstra's algorithm can be modified by using different data structure, buckets, which is called *Dial implementation of Dijkstra's algorithm*.

Time complexity is $O(E + WV)$ where W is maximum weight on any edge of graph, so we can see that, if W is small then this implementation runs much faster than traditional algorithm. Following are important observations.

- Maximum distance between any two node can be at max $w(V - 1)$ (w is maximum edge weight and we can have at max $V-1$ edges between two vertices).
- In Dijkstra algorithm, distances are finalized in non-decreasing, i.e., distance of the closer (to given source) vertices is finalized before the distant vertices.

Algorithm:

1. Maintains some buckets, numbered 0, 1, 2, ..., wV .
2. Bucket k contains all temporarily labelled nodes with distance equal to k.
3. Nodes in each bucket are represented by list of vertices.
4. Buckets 0, 1, 2, ..., wV are checked sequentially until the first non-empty bucket is found. Each node contained in the first non-empty bucket has the minimum distance label by definition.
5. One by one, these nodes with minimum distance label are permanently labelled and deleted from the bucket during the scanning process.
6. Thus operations involving vertex include:
 - Checking if a bucket is empty
 - Adding a vertex to a bucket

- Deleting a vertex from a bucket.
- The position of a temporarily labelled vertex in the buckets is updated accordingly when the distance label of a vertex changes.
 - Process repeated until all vertices are permanently labelled (or distances of all vertices are finalized).

Since the maximum distance can be $w(V - 1)$, we create wV buckets (more for simplicity of code) for implementation of algorithm which can be large if w is big.

```

class Graph {
    int V;
    list<pair<int, int>> *adj;
public:
    Graph(int V) {
        this->V = V;
        adj = new list<pair<int, int>>[V];
    }

    void addEdge(int u, int v, int w) {
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w));
    }

    void shortestPath(int src, int W);
};

void Graph::shortestPath(int src, int W) {
    // With each distance, iterator to that vertex in its bucket
    // is stored so that vertex can be deleted in O(1) at time of
    // updation. So, dist[i].first = distance of ith vertex from src
    // vertex dist[i].second = iterator to vertex i in bucket number.
    vector<pair<int, list<int>::iterator> > dist(V)
    // Initialize all distances as infinite (INT_MAX)
    for (int i = 0; i < V; i++)
        dist[i].first = INT_MAX;
    // Create buckets B[]. B[i] keep vertex of distance label i.
    list<int> B[W*V + 1];
    B[0].push_back(src);
    dist[src].first = 0;
    int index = 0;
    while (1) {
        // Go sequentially through buckets till one non-empty
        // bucket is found
        while (B[index].size() == 0 && index < W*V)
            index++;
        ...
    }
}

```

```

        // If all buckets are empty, we are done.
        if (index == W * V)
            break;
        // Take top vertex from bucket and pop it
        int u = B[index].front();
        B[index].pop_front();
        // Process all adjacents of extracted vertex 'u' and
        // update their distanced if required.
        for (auto i = adj[u].begin(); i != adj[u].end(); ++i) {
            int v = (*i).first;
            int weight = (*i).second;
            int du = dist[u].first;
            int dv = dist[v].first;
            // If there is shorted path to v through u.
            if (dv > du + weight) {
                // If dv is not INF then it must be in B[dv] bucket,
                // so erase its entry using iterator in O(1)
                if (dv != INT_MAX)
                    B[dv].erase(dist[v].second);
                // updating the distance
                dist[v].first = du + weight;
                dv = dist[v].first;
                // pushing vertex v into updated distance's bucket
                B[dv].push_front(v);
                // storing updated iterator in dist[v].second
                dist[v].second = B[dv].begin();
            }
        }
    }
    // Print shortest distances stored in dist[]
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d\t%d\n", i, dist[i].first);
}

int main() {
    int V = 9; Graph g(V);
    g.addEdge(0, 1, 4); g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8); g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7); g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4); g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14); g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2); g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6); g.addEdge(7, 8, 7);
    g.shortestPath(0, 14);
    return 0;
}

```

Problems

Q. 1 0-1 BFS. Given a graph where every edge has weight as either 0 or 1. A source vertex is also given in the graph. Find the shortest path from source vertex to every other vertex.

Solution:

In normal BFS of a graph all edges have equal weight but in 0-1 BFS some edges may have 0 weight and some may have 1 weight. In this we will not use bool array to mark visited nodes but at each step we will check for the optimal distance condition. We use double ended queue to store the node. While performing BFS if an edge having weight = 0 is found node is pushed at front of double ended queue and if an edge having weight = 1 is found, it is pushed at back of double ended queue. The approach is similar to Dijkstra that the if the shortest distance to node is relaxed by the previous node then only it will be pushed in the queue.

The above idea works in all cases, when pop a vertex (like Dijkstra), it is the minimum weight vertex among remaining vertices. If there is a 0-weight vertex adjacent to it, then this adjacent has same distance. If there is a 1 weight adjacent, then this adjacent has maximum distance among all vertices in dequeue (because all other vertices are either adjacent of currently popped vertex or adjacent of previously popped vertices).

```
void Graph::ZeroOneBFS(int src) {
    int* dist = new int[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    // Doubly ended queue keeps track of edges during BFS
    deque<int> dq; dq.push_back(src);
    dist[src] = 0;
    while (!dq.empty()) {
        int v = dq.front(); dq.pop_front();
        for (int i = 0; i < adj[v].size(); i++) {
            // checks for the shortest distance to the nodes
            if (dist[adj[v][i].first] > dist[v] + adj[v][i].second) {
                dist[adj[v][i].first] = dist[v] + adj[v][i].second;
                // Put destination vertex connected by 0 edge to front and
                // vertex connected by 1 edge to back so that vertices are
                // processed in ascending order of weights.
                if (adj[v][i].second == 0)
                    dq.push_front(adj[v][i].first);
                else
                    dq.push_back(adj[v][i].first);
            }
        }
    }
}
```

```

    // printing the dist array that stores the shortest distances
    for (int i = 0; i < V; i++)
        cout << i << " : " << dist[i] << "\n";
}

```

Q. 2 Reverse edges to make a path. Given a directed graph and a source node and destination node, we need to find how many edges we need to reverse in order to make at least 1 path from source node to destination node.

Solution:

This problem can be solved assuming a different version of the given graph. In this version we make a reverse edge corresponding to every edge and we assign that a weight 1 and assign a weight 0 to original edge.

Now we can see that we have modified the graph in such a way that, if we move towards original edge, no cost is incurred, but if we move toward reverse edge 1 cost is added. So, if we apply Dijkstra's shortest path on this modified graph from given source, then that will give us minimum cost to reach from source to destination i.e. minimum edge reversal from source to destination.

```

class Graph {
    int V;
    vector<pair<int, int>> *adj;
public:
    Graph(int V) {
        this->V = V;
        adj = new vector<pair<int, int>>[V];
    }

    void addEdge(int u, int v, int w) {
        adj[u].push_back(make_pair(v, w));
        //adj[v].push_back(make_pair(u, w)); This is directed graph
    }

    vector<int> shortestPath(int src);
};

vector<int> Graph::shortestPath(int src) {
    // set to store vertices that are being processed.
    set<pair<int, int>> vertices;
    // Create a vector for distances and initialize all
    // distances as infinite
    vector<int> dist(V, INT_MAX);
    // Insert source itself in Set and initialize its distance as 0.
    vertices.insert(make_pair(0, src));
    dist[src] = 0;
}

```

```

// Loop while vertices set is empty
while(!vertices.empty()) {
    // The first vertex in Set is the minimum distance vertex.
    pair<int, int> tmp = *(vertices.begin());
    vertices.erase(vertices.begin());
    // vertex label is stored in second of pair.
    int u = tmp.second;
    vector< pair<int, int> >::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        // Get vertex label and weight of current adjacent of u.
        int v = (*i).first;
        int weight = (*i).second;
        // If there is shorter path to v through u.
        if (dist[v] > dist[u] + weight) {
            // If distance of v is not INF then it must be in our set,
            // so removing it and inserting again with updated less
            // distance.
            if (dist[v] != INT_MAX)
                vertices.erase(vertices.find(make_pair(dist[v], v)));
            // Updating distance of v
            dist[v] = dist[u] + weight;
            vertices.insert(make_pair(dist[v], v));
        }
    }
}
return dist;
}

// Following method adds reverse edge of each original edge in
// the graph. It gives reverse edge a weight = 1 and all original
// edges a weight of 0. Now, the length of the shortest path will
// give us the answer. If shortest path is p:
// it means we used p reverse edges in the shortest path.
Graph modelGraphWithEdgeWeight(int edge[][][2], int E, int V) {
    Graph g(V);
    for (int i = 0; i < E; i++) {
        // original edge : weight 0
        g.addEdge(edge[i][0], edge[i][1], 0);
        // reverse edge : weight 1
        g.addEdge(edge[i][1], edge[i][0], 1);
    }
    return g;
}

```

```

// Following method returns minimum number of edges to be reversed
int getMinEdgeReversal(int edge[][] [2], int E, int V, int src, int dest) {
    // get modified graph with edge weight
    Graph g = modelGraphWithEdgeWeight(edge, E, V);
    // get shortest path vector
    vector<int> dist = g.shortestPath(src);
    // If distance of destination is still infinite, not possible
    if (dist[dest] == INT_MAX)
        return -1;
    else
        return dist[dest];
}

int main() {
    int V = 7;
    int edge[] [2] = {{0, 1}, {2, 1}, {2, 3}, {5, 1}, {4, 5}, {6, 4},
{6, 3}};
    int E = sizeof(edge) / sizeof(edge[0]);
    int minEdgeToReverse = getMinEdgeReversal(edge, E, V, 0, 6);
    if (minEdgeToReverse != -1)
        cout << minEdgeToReverse << endl;
    else
        cout << "Not possible" << endl;
    return 0;
}

```

Q. 3 Minimum Cost Path. Given a two-dimensional grid, each cell of which contains integer cost which represents a cost to traverse through that cell, we need to find a path from top left cell to bottom right cell by which total cost incurred is minimum.

A cost grid is given in below diagram, minimum cost to reach bottom right from top left is 327 (= 31 + 10 + 13 + 47 + 65 + 12 + 18 + 6 + 33 + 11 + 20 + 41 + 20). The chosen path is shown in green.

31	100	65	12	18
10	13	47	157	6
100	113	174	11	33
88	124	41	20	140
99	32	111	41	20

Solution:

It is not possible to solve this problem using dynamic programming (mentioned this because many 2-D grid problems are solved using DP) because here current state depends not only on right and bottom cells but also on left and upper cells. We solve this problem using Dijkstra's algorithm. Each cell of grid represents a vertex and neighbour cells adjacent vertices. We do not make an explicit graph from these cells instead we will use matrix as it is in our Dijkstra's algorithm.

```
// Class for information of each cell
class cell {
public:
    int x, y, distance;
    // newer syntax for constructor
    cell (int x, int y, int distance): x(x), y(y), distance(distance) {}
};

// Utility method for comparing two cells (operator '<' overloaded)
bool operator < (const cell &a, const cell &b) {
    if (a.distance == b.distance) {
        if (a.x != b.x)
            return (a.x < b.x);
        else
            return (a.y < b.y);
    }
    return (a.distance < b.distance);
}

// Utility method to check whether the point is inside grid or not
bool isInsideGrid(int i, int j, int row, int col) {
    return (i >= 0 && i < col && j >= 0 && j < row);
}

int dijkstra(vector<vector<int>> grid, int row, int col) {
    int dis[row][col];
    // initializing distance array by INT_MAX
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            dis[i][j] = INT_MAX;
    // direction arrays for simplification of getting neighbour
    int dx[] = {-1, 0, 1, 0};
    int dy[] = {0, 1, 0, -1};
    set<cell> st;
    // insert (0, 0) cell with 0 distance
    st.insert(cell(0, 0, 0));
    // initialize distance of (0, 0) with its grid value
    dis[0][0] = grid[0][0];
```

```

while (!st.empty()) {
    cell k = *(st.begin());
    st.erase(st.begin());
    for (int i = 0; i < 4; i++) {
        int x = k.x + dx[i];
        int y = k.y + dy[i];
        // if not inside boundary, ignore them
        if (!isInsideGrid(x, y, row, col))
            continue;
        // If distance from current cell is smaller, then update
        // distance of neighbour cell
        if (dis[x][y] > dis[k.x][k.y] + grid[x][y]) {
            // If cell is already there in set, then
            // remove its previous entry
            if (dis[x][y] != INT_MAX)
                st.erase(st.find(cell(x, y, dis[x][y])));
            // update the distance and insert new updated
            // cell in set
            dis[x][y] = dis[k.x][k.y] + grid[x][y];
            st.insert(cell(x, y, dis[x][y]));
        }
    }
}
return dis[row - 1][col - 1];
}

int main() {
    int row = 5, col = 5;
    vector<vector<int>> grid {
        {31, 100, 65, 12, 18},
        {10, 13, 47, 157, 6},
        {100, 113, 174, 11, 33},
        {88, 124, 41, 20, 140},
        {99, 32, 111, 41, 20},
    };
    cout << dijkstra(grid, row, col) << endl;
    return 0;
}

```

Q. 4 Shortest distance from guard in a bank. Given a matrix that is filled with 'O', 'G', and 'W' where 'O' represents open space, 'G' represents guards and 'W' represents walls in a Bank. Replace all of the O's in the matrix with their shortest distance from a guard, without being able to go through any walls. Also, replace the guards with 0 and walls with -1 in output matrix.

O ==> Open Space

G ==> Guard

W ==> Wall

Input:

```
O O O O G  
O W W O O  
O O O W O  
G W W W O  
O O O O G
```

Output:

```
3 3 2 1 0  
2 -1 -1 2 1  
1 2 3 -1 2  
0 -1 -1 -1 1  
1 2 2 1 0
```

Solution:

The idea is to do BFS. We first enqueue all cells containing the guards and loop till queue is not empty. For each iteration of the loop, we dequeue the front cell from the queue and for each of its four adjacent cells, if cell is an open area and its distance from guard is not calculated yet, we update its distance and enqueue it. Finally, after BFS procedure is over, we print the distance matrix.

```
#define N 5

struct cell {
    // i, j and distance stores x and y-coordinates of a matrix
    // cell and its distance from guard respectively
    int i, j, distance;
};

// These arrays are used to get row and column numbers of
// 4 neighbors of a given cell
int row[] = { -1, 0, 1, 0};
int col[] = { 0, 1, 0, -1 };

// return true if row number and column number is in range
bool isValid(int i, int j, int m, int n) {
    if ((i < 0 || i > m - 1) || (j < 0 || j > n - 1))
        return false;
    return true;
}
```

```

// return true if current cell is an open area and its distance
// from guard is not calculated yet
bool isSafe(int i, int j, char matrix[][][N], int** output) {
    if (matrix[i][j] != '0' || output[i][j] != -1)
        return false;
    return true;
}

// Function to replace all of the 0's in the matrix with their shortest
// distance from a guard
void findDistance(char matrix[][][N], int m, int n) {
    int** output = new int*[m];
    for (int i = 0; i < m; i++)
        output[i] = new int[n];
    queue<cell> q;
    // finding Guards location and adding into queue
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            // initialize each cell as -1
            output[i][j] = -1;
            if (matrix[i][j] == 'G') {
                cell pos = {i, j, 0};
                q.push(pos);
                // guard has 0 distance
                output[i][j] = 0;
            }
        }
    }
    // do till queue is empty
    while (!q.empty()) {
        // get the front cell in the queue and update its adjacent cells
        cell curr = q.front();
        int x = curr.i, y = curr.j, dist = curr.distance;
        for (int i = 0; i < 4; i++) {
            // if adjacent cell is valid, has path and not visited yet,
            // en-queue it.
            if (isSafe(x + row[i], y + col[i], matrix, output) && \
                isValid(x + row[i], y + col[i], m, n)) {
                output[x + row[i]][y + col[i]] = dist + 1;
                cell pos = {x + row[i], y + col[i], dist + 1};
                q.push(pos);
            }
        }
        // dequeue the front cell as its distance is found
        q.pop();
    }
}

```

```

// print output matrix
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
        cout << std::setw(3) << output[i][j];
    cout << endl;
}
}

int main() {
    int m = 5, n = 5;
    char matrix[][][N] = {
        {'0', '0', '0', '0', 'G'},
        {'0', 'W', 'W', '0', '0'},
        {'0', '0', '0', 'W', '0'},
        {'G', 'W', 'W', 'W', '0'},
        {'0', '0', '0', '0', 'G'}
    };
    findDistance(matrix, m, n);
    return 0;
}

```

10.9 Connected Components

Articulation Points:

In a graph, a vertex is called an articulation point if removing it and all the edges associated with it results in the increase of the number of connected components in the graph. For example, consider the graph given in following figure.

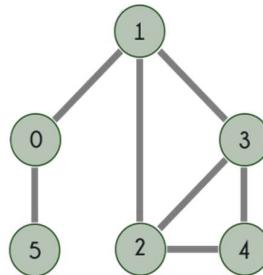


Fig. 1

If in the above graph, vertex 1 and all the edges associated with it, i.e. the edges 1-0, 1-2 and 1-3 are removed, there will be no path to reach any of the vertices 2, 3 or 4 from the vertices 0 and 5, that means the graph will split into two separate components - one consisting of the vertices 0 and 5 and another one consisting of the vertices 2, 3 and 4 as shown in the following figure.

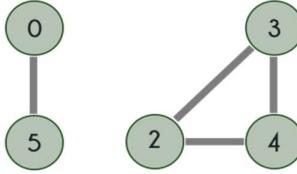


Fig. 2

Likewise removing the vertex 0 will disconnect the vertex 5 from all other vertices. Hence the given graph has two articulation points: 0 and 1.

Articulation Points represent vulnerabilities in the network. In order to find all the articulation points in a given graph, the brute force approach is to check for every vertex if it is an articulation point or not, by removing it and then counting the number of connected components in the graph.

For every vertex v , do following

- Remove v from graph
- See if the graph remains connected (We can either use BFS or DFS)
- Add v back to the graph

Time complexity of above method is $O(V * (V + E))$ for a graph represented using adjacency list. Can we do better? There is an algorithm that can help find all the articulation points in a given graph by a single Depth First Search, that means with complexity $O(V + E)$, but it involves a new term called 'Back-Edge' which is explained below:

Given a DFS tree of a graph, a Back Edge is an edge that connects a vertex to a vertex that is discovered before its parent. For example, consider the graph given above. The figure given below depicts a DFS tree of that graph.

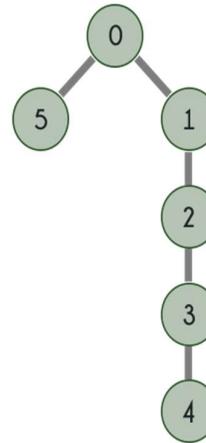


Fig. 3

In the above case, the edge 4 - 2 connects 4 to an ancestor of its parent i.e. 3, so it is a Back Edge. And similarly, 3 - 1 is also a Back edge. But why do we bother about

Back Edge? Presence of a back edge means presence of an alternative path in case the parent of the vertex is removed. Suppose a vertex u is having a child v such that none of the vertices in the subtree rooted at v have a back edge to any vertex discovered before u . Then if vertex u is removed, there will be no path left for vertex v or any of the vertices present in the subtree rooted at vertex v to reach any vertex discovered before u . That implies, the subtree rooted at vertex v will get disconnected from the entire graph, and thus the number of components will increase and u will be counted as an articulation point. On the other hand, if the subtree rooted at vertex v has a vertex x that has back edge that connects it to a vertex discovered before u , say y , then there will be a path for any vertex in subtree rooted at v to reach y even after removal of u , and if that is the case with all the children of u , then u will not be counted as an articulation point.

So ultimately it all converges down to finding a back edge for every vertex. So, for that we apply a DFS and record the discovery time of every vertex and maintain for every vertex v the earliest discovered vertex that can be reached from any of the vertices in the subtree rooted at v . If a vertex u is having a child v such that the earliest discovered vertex that can be reached from the vertices in the subtree rooted at v has a discovery time greater than or equal to u , then v does not have a back edge, and thus u will be an articulation point.

So, till now the algorithm says that if all children of a vertex u are having a back edge, then u is not an articulation point. But what will happen when u is root of the tree, as root does not have any ancestors. Well, it is very easy to check if the root is an articulation point or not. If root has more than one child than it is an articulation point otherwise it is not. Now how does that help?? Suppose root has two children, v_1 and v_2 . If there had been an edge between vertices in the subtree rooted at v_1 and those of the subtree rooted at v_2 , then they would have been a part of the same subtree.

Here's the pseudo code of the above algorithm:

```

time = 0
function DFS(adj[][], disc[], low[], visited[], parent[], AP[], vertex, V)
    visited[vertex] = true
    disc[vertex] = low[vertex] = time+1
    child = 0
    for i = 0 to V
        if adj[vertex][i] == true
            if visited[i] == false
                child = child + 1
                parent[i] = vertex
                DFS(adj, disc, low, visited, parent, AP, i, n, time+1)
                low[vertex] = minimum(low[vertex], low[i])
            if parent[vertex] == nil and child > 1
                AP.append(vertex)

```

```

AP[vertex] = true
if parent[vertex] != nil and low[i] >= disc[vertex]
    AP[vertex] = true
else if parent[vertex] != i
    low[vertex] = minimum(low[vertex], disc[i])

```

Here's what everything means:

`adj[]`: It is an $N \times N$ matrix denoting the adjacency matrix of the given graph.

`disc[]`: It is an array of N elements which stores the discovery time of every vertex. It is initialized by 0.

`low[]`: It is an array of N elements which stores, for every vertex v , the discovery time of the earliest discovered vertex to which v or any of the vertices in the subtree rooted at v is having a back edge. It is initialized by infinity.

`AP[]`: It is an array of size N . $AP[i] = \text{true}$, if i^{th} vertex is an articulation point.

`time`: Current value of discovery time.

Here is the dry run of above algorithm:

The above algorithm starts with an initial vertex say u , marks it visited, record its discovery time, $\text{disc}[u]$, and since it is just discovered, the earliest vertex it is connected to is itself, so $\text{low}[u]$ is also set equal to vertex's discovery time.

It keeps a counter called 'child' to count the number of children of a vertex. Then the algorithm iterates over every vertex in the graph and see if it is connected to u . If it finds a vertex v that is connected to u but has already been visited, then it updates the value $\text{low}[u]$ to minimum of $\text{low}[u]$ and discovery time of v i.e., $\text{disc}[v]$. But if the vertex v is not yet visited, then it sets the $\text{parent}[v]$ to u and calls the DFS again with $\text{vertex} = v$. So, the same things that just happened with u will happen for v also. When that DFS call will return, $\text{low}[v]$ will have the discovery time of the earliest discovered vertex that can be reached from any vertex in the subtree rooted at v . So set $\text{low}[u]$ to minimum of $\text{low}[v]$ and itself. And finally, if u is not the root, it checks whether $\text{low}[v]$ is greater than or equal to $\text{disc}[u]$, and if so, it marks $AP[u]$ as true. And if u is root it checks whether it has more than one child or not, and if so, it marks $AP[u]$ as true.

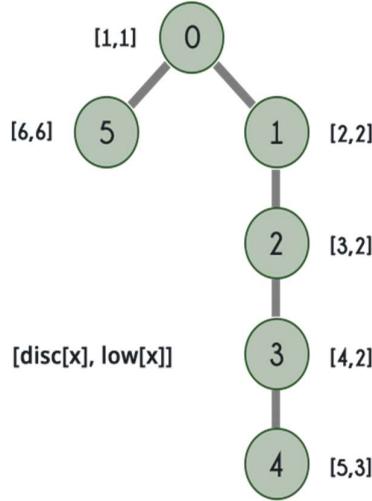


Fig. 4

Clearly only for vertices 0 and 1, $\text{low}[5] \geq \text{disc}[0]$ and $\text{low}[2] \geq \text{disc}[1]$, so these are the only two articulation points in the given graph.

Here is the implementation of above concept:

```

#define NIL -1

class Graph {
    int V;
    list<int> *adj;
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    void addEdge(int v, int w) {
        adj[v].push_back(w);
        adj[w].push_back(v); // The graph is undirected
    }

    void AP();
};


```

```

void Graph::APUtil(int u, bool visited[], int disc[], int low[],
                   int parent[], bool ap[]) {
    // A static variable is used for simplicity; we can avoid use of
    // static variable by passing a pointer.
    static int time = 0;
    // Count of children in DFS Tree
    int children = 0;
    // Mark the current node as visited
    visited[u] = true;
    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        int v = *i; // v is current adjacent of u
        // If v is not visited yet, then make it a child of
        // u in DFS tree and recur for it
        if (!visited[v]) {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);
            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);
            // u is an articulation point in following cases
            // (1) u is root of DFS tree and has two or more children.
            if (parent[u] == NULL && children > 1)
                ap[u] = true;
            // (2) If u is not root and low value of one of its child is
            // more than discovery value of u.
            if (parent[v] != NULL && low[v] >= disc[u])
                ap[u] = true;
        }
        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP() {
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points
}

```

```

// Initialize parent and visited, and ap(articulation point) arrays
for (int i = 0; i < V; i++) {
    parent[i] = NIL;
    visited[i] = false;
    ap[i] = false;
}
// Call the recursive helper function to find articulation points
// in DFS tree rooted with vertex 'i'
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        APUtil(i, visited, disc, low, parent, ap);
// Now ap[] contains articulation points, print them
for (int i = 0; i < V; i++)
    if (ap[i] == true)
        cout << i << " ";
delete[] visited, disc, low, parent, ap;
}

int main() {
    cout << "\nArticulation points in first graph: \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();
    cout << "\nArticulation points in second graph: \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();
    cout << "\nArticulation points in third graph: \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.AP();
    return 0;
}

```

Bridges:

An edge in a graph between vertices say u and v is called a Bridge, if after removing it, there will be no path left between u and v. Its definition is very similar to that of Articulation Points. Just like them, they also represent vulnerabilities in the given network. For the graph given in Fig.1 (the one which we considered for AP), if the edge 0-1 is removed, there will be no path left to reach from 0 to 1, similarly if edge 0-5 is removed, there will be no path left that connects 0 and 5. So in this case the edges 0-1 and 0-5 are the Bridges in the given graph.

The Brute force approach to find all the bridges in a given graph is to check for every edge if it is a bridge or not, by first removing it and then checking if the vertices that it was connecting are still connected or not. Following is pseudo code of this approach:

```
function find_bridges(adj[][], V, Edge[], E, isBridge[])
    for i = 0 to E
        adj[Edge[i].u][Edge[i].v]=adj[Edge[i].v][Edge[i].u]=0
        for j = 0 to V
            visited[j] = false
            Queue.Insert(Edge[i].u)
            visited[Edge[i].u] = true
            check = false
            while Queue.isEmpty() == false
                x = Queue.top()
                if x == Edge[i].v
                    check = true
                    BREAK
                Queue.Delete()
                for j = 0 to V
                    if adj[x][j] == true AND visited[j] == false
                        Queue.insert(j)
                        visited[j] = true
        adj[Edge[i].u][Edge[i].v]=adj[Edge[i].v][Edge[i].u]=1
        if check == false
            isBridge[i] = true
```

The above code uses BFS to check if the vertices that were connected by the removed edge are still connected or not. It does so for every edge and thus its complexity is $O(E(V + E))$. Clearly it will fail for big values of V and E.

To check if an edge is a bridge or not the above algorithm checks if the vertices that the edge is connecting are connected even after removal of the edge or not. If they are still connected, this implies existence of an alternate path. So just like in the case of Articulation Points, the concept of Back Edge can be used to check the existence of the alternate path.

For any edge, $u-v$, (u having discovery time less than v), if the earliest discovered vertex that can be visited from any vertex in the subtree rooted at vertex v has discovery time **strictly greater** than that of u , then $u-v$ is a Bridge otherwise not. Unlike articulation point, here root is not a special case. Following is the pseudo code for the algorithm:

```

time = 0
function DFS(adj[][], disc[], low[], visited[], parent[], vertex, n)
    visited[vertex] = true
    disc[vertex] = low[vertex] = time+1
    child = 0
    for i = 0 to n
        if adj[vertex][i] == true
            if visited[i] == false
                child = child + 1
                parent[i] = vertex
                DFS(adj, disc, low, visited, parent, i, n, time+1)
                low[vertex] = minimum(low[vertex], low[i])
                if low[i] > disc[vertex]
                    print vertex, i
            else if parent[vertex] != i
                low[vertex] = minimum(low[vertex], disc[i])

```

The values of array `low[]` and `disc[]` are shown above in the figure 4. Clearly for only two edges i.e 0-1 and 0-5, $\text{low}[1] > \text{disc}[0]$ and $\text{low}[5] > \text{disc}[0]$, hence those are the only two bridges in the given graph.

```

// Recursive function that finds and prints bridges using DFS traversal.
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[], int low[],
                      int parent[]) {
    // A static variable is used for simplicity; we can avoid use of
    // static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    list<int>::iterator i;

```

```

        for (i = adj[u].begin(); i != adj[u].end(); ++i) {
            int v = *i; // v is current adjacent of u

            // If v is not visited yet, then recur for it
            if (!visited[v]) {
                parent[v] = u;
                bridgeUtil(v, visited, disc, low, parent);

                // Check if the subtree rooted with v has a
                // connection to one of the ancestors of u
                low[u] = min(low[u], low[v]);

                // If the lowest vertex reachable from subtree under v is
                // below u in DFS tree, then u-v is a bridge
                if (low[v] > disc[u])
                    cout << u << " " << v << endl;
            }
            // Update low value of u for parent function calls.
            else if (v != parent[u])
                low[u] = min(low[u], disc[v]);
        }
    }

    // DFS based function to find all bridges. It uses recursive func
    // bridgeUtil()
    void Graph::bridge() {
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        int *disc = new int[V];
        int *low = new int[V];
        int *parent = new int[V];
        // Initialize parent and visited arrays
        for (int i = 0; i < V; i++) {
            parent[i] = NILL;
            visited[i] = false;
        }

        // Call the recursive helper function to find Bridges in DFS tree
        // rooted with vertex 'i'
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                bridgeUtil(i, visited, disc, low, parent);

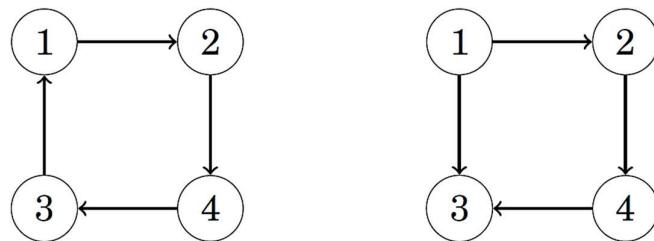
        delete[] visited, disc, low, parent;
    }
}

```

Strongly Connected Components:

In a directed graph, the edges can be traversed in one direction only. So even if the graph is connected, this does not guarantee that there would be a path from a node to another node. For this reason, it is meaningful to define a new concept that requires more than connectivity.

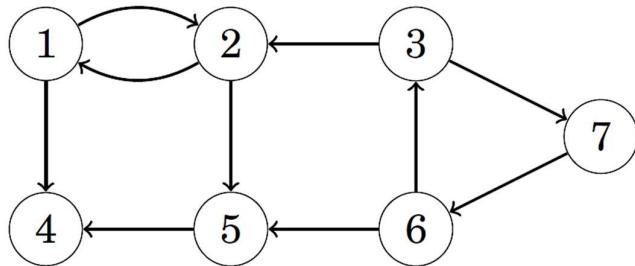
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.



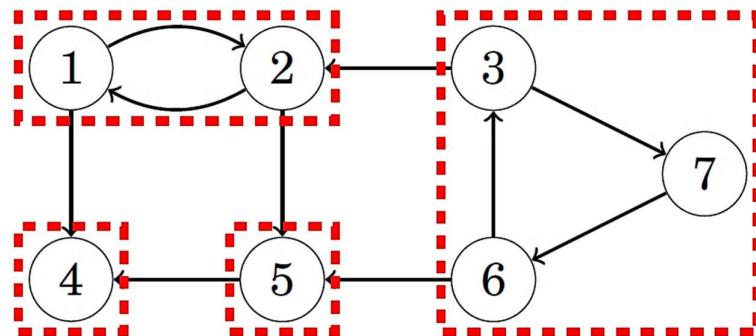
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The strongly connected components of a graph divide the graph into strongly connected parts that are as large as possible. The strongly connected components form an acyclic component graph that represents the deep structure of the original graph.

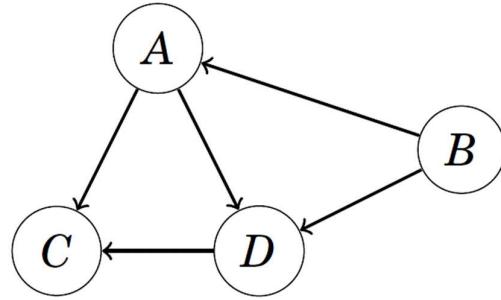
For example, for the graph



the strongly connected components are as follows:



The corresponding component graph is as follows:



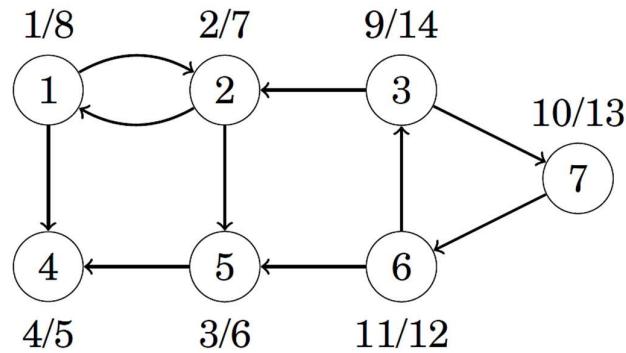
The components are $A = \{1,2\}$, $B = \{3,6,7\}$, $C = \{4\}$ and $D = \{5\}$.

A component graph is an acyclic, directed graph, so it is easier to process than the original graph.

Kosaraju's algorithm is an efficient method for finding the strongly connected components of a directed graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

Search 1

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which a depth-first search processes them. The algorithm goes through the nodes, and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed. In the example graph, the nodes are processed in the following order:

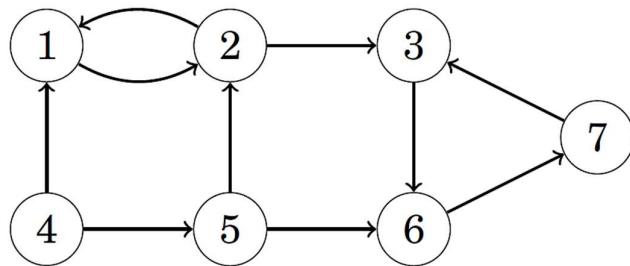


The notation x/y means that processing the node started at time x and finished at time y . Thus, the corresponding list is as follows:

<i>Node</i>	<i>Processing Time</i>
4	5
5	6
2	7
3	8
6	12
7	13
4	14

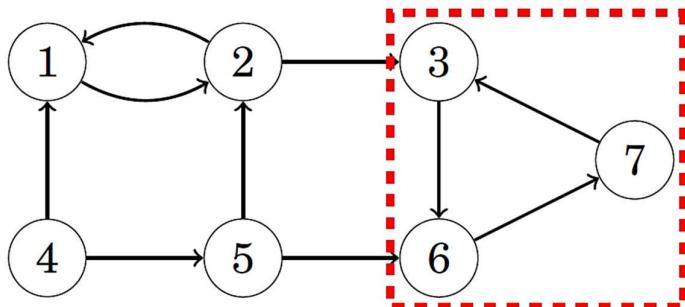
Search 2

The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes. After reversing the edges, the example graph is as follows:



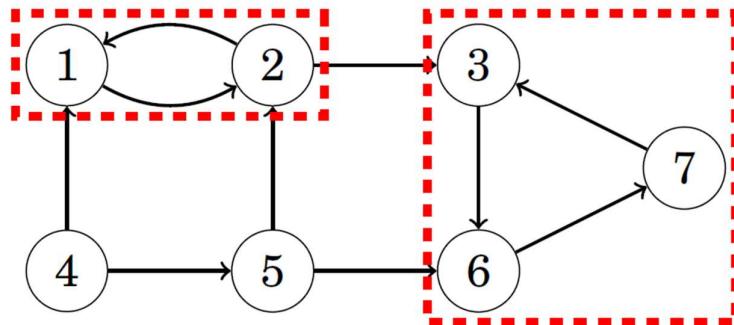
After this, the algorithm goes through the list of nodes created by the first search, in reverse order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

In the example graph, the first component begins at node 3:

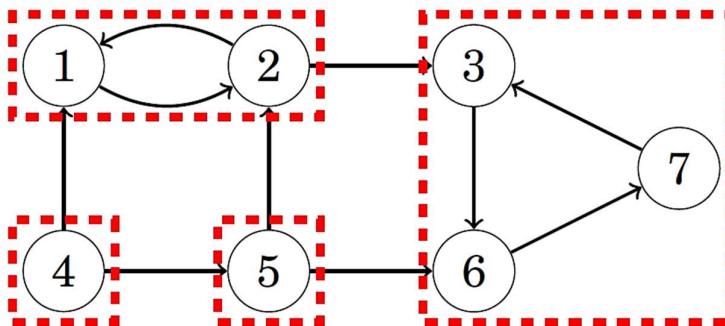


Note that since all edges are reversed, the component does not ‘leak’ to other parts in the graph.

The next nodes in the list are nodes 7 and 6, but they already belong to a component, so the next new component begins at node 1:



Finally, the algorithm processes nodes 5 and 4 that create the remaining strongly connected components:



The time complexity of the algorithm is $O(V + E)$, because the algorithm performs two depth-first searches.

```
class Graph {
    int V;
    list<int> *adj;
    // Fills Stack with vertices (in increasing order of finish times).
    // The top element of stack has the maximum finishing time.
    void fillOrder(int v, bool visited[], stack<int> &Stack);
    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
}
```

```

// The main function that finds and prints SCCs
void printSCCs();

// Function that returns reverse (or transpose) of this graph
Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[]) {
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

Graph Graph::getTranspose() {
    Graph g(V);
    for (int v = 0; v < V; v++) {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i) {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack) {
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);
    // All vertices reachable from v are processed by now, push v
    Stack.push(v);
}

```

```

// The main function that finds and prints all SCCs
void Graph::printSCCs() {
    stack<int> Stack;
    // Mark all the vertices as not visited (For first DFS)
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++) visited[i] = false;
    // Fill vertices in stack according to their finishing times
    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, Stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Now process all vertices in order defined by Stack
    while (Stack.empty() == false) {
        int v = Stack.top();
        Stack.pop();

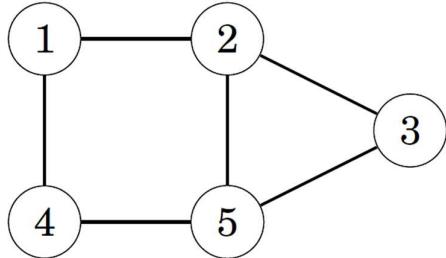
        // Print Strongly connected component of the popped vertex
        if (visited[v] == false) {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
    delete[] visited;
}

int main() {
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    cout << "Following are SCCs in given graph: \n";
    g.printSCCs();
    return 0;
}

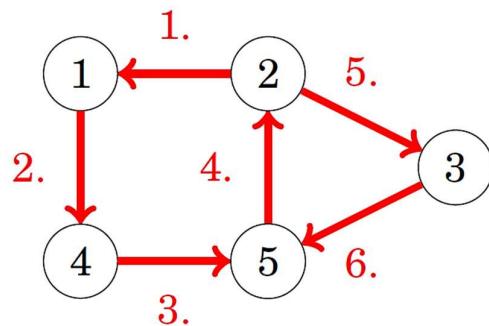
```

Eulerian Path:

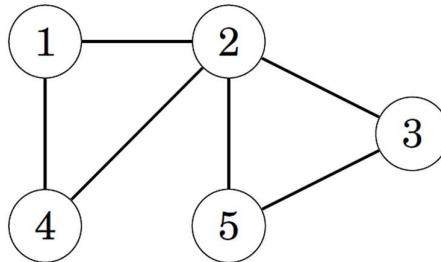
An **Eulerian path** is a path that goes exactly once through each edge of the graph. For example, the graph



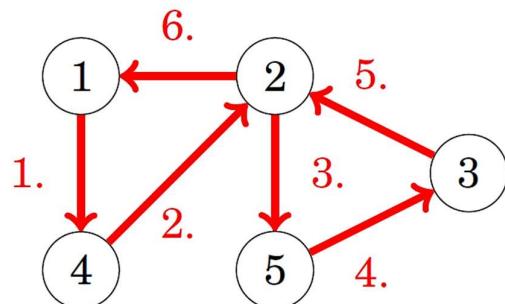
has an Eulerian path from node 2 to node 5:



An **Eulerian circuit** is an Eulerian path that starts and ends at the same node. For example, the graph



has an Eulerian circuit that starts and ends at node 1:



Condition of Existence:

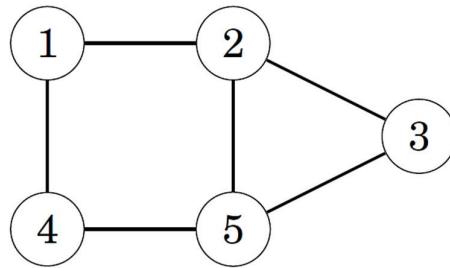
The existence of Eulerian paths and circuits depends on the degrees of the nodes.

First, an undirected graph has an Eulerian path exactly when all the edges belong to the same connected component and

- the degree of each node is even or
- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path which is not an Eulerian circuit.

For example, in the graph



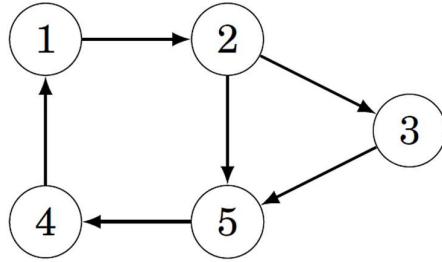
nodes 1, 3 and 4 have a degree of 2, and nodes 2 and 5 have a degree of 3. Exactly two nodes have an odd degree, so there is an Eulerian path between nodes 2 and 5, but the graph does not contain an Eulerian circuit.

In a directed graph, we focus on indegrees and outdegrees of the nodes. A directed graph contains an Eulerian path exactly when all the edges belong to the same connected component and

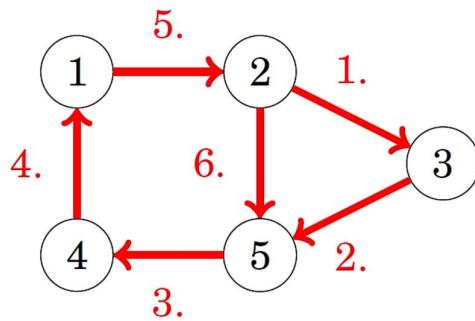
- in each node, the indegree equals the outdegree, or
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph contains an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger.

For example, in the graph



nodes 1, 3 and 4 have both indegree 1 and outdegree 1, node 2 has indegree 1 and outdegree 2, and node 5 has indegree 2 and outdegree 1. Hence, the graph contains an Eulerian path from node 2 to node 5:



We can print Euler's path for a given graph using – **Fleury's Algorithm**.

Fleury's Algorithm:

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, always choose the non-bridge.
4. Stop when you run out of edges.

```

class Graph {
    int V;
    list<int> *adj;
public:
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }
    // functions to add and remove edge
    void addEdge(int u, int v) {
        adj[u].push_back(v); adj[v].push_back(u);
    }
    void rmvEdge(int u, int v);
    // Methods to print Eulerian tour
    void printEulerTour();
    void printEulerUtil(int s);
}
  
```

```

// This function returns count of vertices reachable from v.
int DFSCount(int v, bool visited[]);

// Utility function to check if edge u-v is a valid next edge in
// Eulerian trail or circuit
bool isValidNextEdge(int u, int v);
};

/* The main function that print Eulerian Trail. It first finds an odd
degree vertex (if there is any) and then calls printEulerUtil()
to print the path. */
void Graph::printEulerTour() {
    // Find a vertex with odd degree
    int u = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1) {
            u = i;
            break;
        }
    // Print tour starting from oddv
    printEulerUtil(u);
    cout << endl;
}

// Print Euler tour starting from vertex u
void Graph::printEulerUtil(int u) {
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        int v = *i;
        // If edge u-v is not removed and it's a a valid next edge
        if (v != -1 && isValidNextEdge(u, v)) {
            cout << u << "-" << v << " ";
            rmvEdge(u, v);
            printEulerUtil(v);
        }
    }
}

// The function to check if edge u-
// v can be considered as next edge in Euler Tour
bool Graph::isValidNextEdge(int u, int v) {
    // The edge u-v is valid in one of the following two cases:

    // 1) If v is the only adjacent vertex of u
    int count = 0;
    list<int>::iterator i;

```

```

        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (*i != -1)
                count++;
        if (count == 1)
            return true;

        // 2) If there are multiple adjacents, then u-v is not a bridge

        // 2.a) count of vertices reachable from u
        bool visited[V];
        memset(visited, false, V);
        int count1 = DFSCount(u, visited);
        // 2.b) Remove edge (u, v) and after removing the edge,
        // count vertices reachable from u
        rmvEdge(u, v);
        memset(visited, false, V);
        int count2 = DFSCount(u, visited);
        // 2.c) Add the edge back to the graph
        addEdge(u, v);
        // 2.d) If count1 is greater, then edge (u, v) is a bridge
        return (count1 > count2)? false: true;
    }

    // This function removes edge u-v from graph. It removes the edge by
    // replacing adjacent vertex value with -1.
    void Graph::rmvEdge(int u, int v) {
        // Find v in adjacency list of u and replace it with -1
        list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);
        *iv = -1;
        // Find u in adjacency list of v and replace it with -1
        list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);
        *iu = -1;
    }

    // A DFS based function to count reachable vertices from v
    int Graph::DFSCount(int v, bool visited[]) {
        // Mark the current node as visited
        visited[v] = true;
        int count = 1;
        // Recur for all vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (*i != -1 && !visited[*i])
                count += DFSCount(*i, visited);
        return count;
    }
}

```

```

int main() {
    Graph g1(4);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.printEulerTour();

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 0);
    g2.printEulerTour();

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(3, 2);
    g3.addEdge(3, 1);
    g3.addEdge(2, 4);
    g3.printEulerTour();

    return 0;
}

```

Problem

Q. 1 Chain the strings. Given an array of strings, find if the given strings can be chained to form a circle. A string X can be put before another string Y in circle if the last character of X is same as first character of Y. Examples:

Input: arr[] = {"geek", "king"}

Output: Yes, the given strings can be chained.

Note that the last character of first string is same as first character of second string and vice versa is also true.

Input: arr[] = {"for", "geek", "rig", "kaf"}

Output: Yes, the given strings can be chained.

The strings can be chained as "for", "rig", "geek" and "kaf"

Input: arr[] = {"aab", "bac", "aaa", "cda"}

Output: Yes, the given strings can be chained.

The strings can be chained as "aaa", "aab", "bac" and "cda"

Input: arr[] = {"aaa", "bbb", "baa", "aab"};
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bbb" and "baa"

Input: arr[] = {"aaa"};
Output: Yes

Input: arr[] = {"aaa", "bbb"};
Output: No

Input: arr[] = ["abc", "efg", "cde", "ghi", "ija"]
Output: Yes
These strings can be reordered as, "abc", "cde", "efg", "ghi", "ija"

Input: arr[] = ["ijk", "kji", "abc", "cba"]
Output: No

Solution:

The idea is to create a directed graph of all characters and then find if there is an eulerian circuit in the graph or not.

```
#define CHARS 26

class Graph {
    int V;
    list<int> *adj;
    int *in;
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
        in = new int[V];
        for (int i = 0; i < V; i++)
            in[i] = 0;
    };
    ~Graph() { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); (in[w])++; }

    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();
```

```

// Function to do DFS starting from v. Used in isConnected();
void DFSUtil(int v, bool visited[]);

Graph getTranspose();
};

// Function returns true if the directed graph has an eulerian cycle.
bool Graph::isEulerianCycle() {
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;
    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;
    return true;
}

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[]) {
    // Mark the current node as visited and print it
    visited[v] = true;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph.
// This function is needed in isSC().
Graph Graph::getTranspose() {
    Graph g(V);
    for (int v = 0; v < V; v++) {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i) {
            g.adj[*i].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

```

```

// This function returns true if all non-zero degree vertices of
// graph are strongly connected.
bool Graph::isSC() {
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    // Find the first vertex with non-zero degree
    int n;
    for (n = 0; n < V; n++)
        if (adj[n].size() > 0)
            break;
    // Do DFS traversal starting from first non zero degree vertex.
    DFSUtil(n, visited);
    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (adj[i].size() > 0 && visited[i] == false)
            return false;
    // Create a reversed graph
    Graph gr = getTranspose();
    // Mark all the vertices as not visited (For second DFS)
    for (int i = 0; i < V; i++) visited[i] = false;
    // Do DFS for reversed graph starting from first vertex.
    // Starting Vertex must be same starting point of first DFS
    gr.DFSUtil(n, visited);
    // If all vertices are not visited in second DFS, then return false
    for (int i = 0; i < V; i++)
        if (adj[i].size() > 0 && visited[i] == false)
            return false;
    return true;
}

// This function takes an array of strings and returns true if the given
// array of strings can be chained to form cycle.
bool canBeChained(string arr[], int n) {
    // Create a graph with 'alpha' edges
    Graph g(CHARS);
    // Create an edge from first character to last character
    // of every string
    for (int i = 0; i < n; i++) {
        string s = arr[i];
        g.addEdge(s[0]-'a', s[s.length()-1]-'a');
    }
    // The given array of strings can be chained if there
    // is an eulerian cycle in the created graph
    return g.isEulerianCycle();
}

```

```

int main() {
    string arr1[] = {"for", "geek", "rig", "kaf"};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    canBeChained(arr1, n1)? cout << "Can be chained.\n" :
        cout << "Can't be chained.\n";
    string arr2[] = {"aab", "abb"};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    canBeChained(arr2, n2)? cout << "Can be chained.\n" :
        cout << "Can't be chained.\n";
    return 0;
}

```

Q. 2 Mother Vertex. Find mother vertex for a given graph. A mother vertex in a graph $G = (V, E)$ is a vertex v such that all other vertices in G can be reached by a path from v .

Solution:

Case 1: - Undirected Connected Graph: In this case, all the vertices are mother vertices as we can reach to all the other nodes in the graph.

Case 2: - Undirected/Directed Disconnected Graph: In this case, there is no mother vertices as we cannot reach to all the other nodes in the graph.

Case 3: - Directed Connected Graph: In this case, we have to find a vertex $-v$ in the graph such that we can reach to all the other nodes in the graph through a directed path.

A Naive approach:

A trivial approach will be to perform a DFS/BFS on all the vertices and find whether we can reach all the vertices from that vertex. This approach takes $O(V(E + V))$ time, which is very inefficient for large graphs.

Can we do better?

We can find a mother vertex in $O(V + E)$ time. The idea is based on Kosaraju's Strongly Connected Component Algorithm. In a graph of strongly connected components, mother vertices are always vertices of source component in component graph. The idea is based on below fact.

If there exist mother vertex (or vertices), then one of the mother vertices is the last finished vertex in DFS. (Or a mother vertex has the maximum finish time in DFS traversal).

A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

How does the above idea work?

Let the last finished vertex be v . Basically, we need to prove that there cannot be an edge from another vertex u to v if u is not another mother vertex (Or there cannot exist a non-mother vertex u such that $u \rightarrow v$ is an edge). There can be two possibilities.

Recursive DFS call is made for u before v . If an edge $u \rightarrow v$ exists, then v must have finished before u because v is reachable through u and a vertex finishes after all its descendants.

Recursive DFS call is made for v before u . In this case also, if an edge $u \rightarrow v$ exists, then either v must finish before u (which contradicts our assumption that v is finished at the end) OR u should be reachable from v (which means u is another mother vertex).

Algorithm:

- Do DFS traversal of the given graph. While doing traversal keep track of last finished vertex ' v '. This step takes $O(V + E)$ time.
- If there exists mother vertex (or vertices), then v must be one (or one of them). Check if v is a mother vertex by doing DFS/BFS from v . This step also takes $O(V + E)$ time.

```
class Graph {
    int V;
    list<int> *adj;
    void DFSUtil(int v, vector<bool> &visited);
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
    int findMother();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, vector<bool> &visited) {
    // Mark the current node as visited and print it
    visited[v] = true;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}
```

```

// Returns a mother vertex if exists. Otherwise returns -1
int Graph::findMother() {
    // visited[] is used for DFS. Initially all are initialized as
    // not visited
    vector <bool> visited(V, false);
    // To store last finished vertex (or mother vertex)
    int v = 0;
    // Do a DFS traversal and find the last finished vertex
    for (int i = 0; i < V; i++) {
        if (visited[i] == false) {
            DFSUtil(i, visited);
            v = i;
        }
    }
    // If there exist mother vertex (or vertices) in given graph,
    // then v must be one (or one of them). Now check if v is actually
    // a mother vertex (or graph has a mother vertex). We basically
    // check if every vertex is reachable from v or not. Reset all
    // values in visited[] as false and do DFS beginning from v to
    // check if all vertices are reachable from it or not.
    fill(visited.begin(), visited.end(), false);
    DFSUtil(v, visited);
    for (int i=0; i<V; i++)
        if (visited[i] == false)
            return -1;
    return v;
}

int main() {
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(4, 1);
    g.addEdge(6, 4);
    g.addEdge(5, 6);
    g.addEdge(5, 2);
    g.addEdge(6, 0);

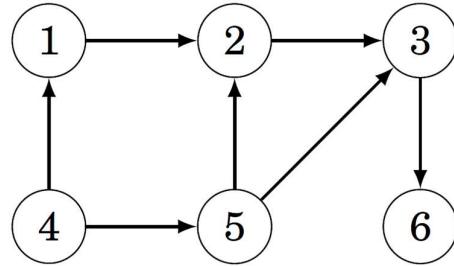
    cout << "A mother vertex is " << g.findMother() << ".\n";

    return 0;
}

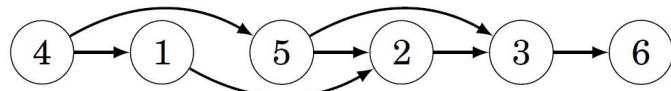
```

10.10 Topological Sorting

A topological sort is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. For example, for the graph



one topological sort is [4, 1, 5, 2, 3, 6]:



An acyclic graph always has a topological sort (directed acyclic graphs are sometimes called DAGs.). However, if the graph contains a cycle, it is not possible to form a topological sort, because no node of the cycle can appear before the other nodes of the cycle in the ordering. It turns out that depth-first search can be used to solve both problems - check if a directed graph contains a cycle and, if it does not contain a cycle, to construct a topological sort.

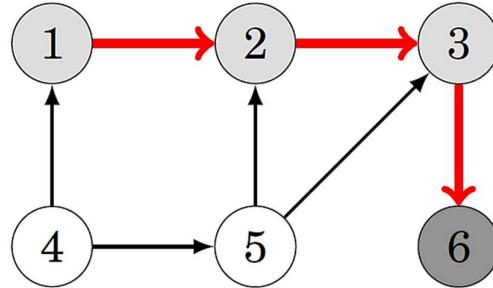
The idea is to go through the nodes of the graph and always begin a depth-first search at the current node if it has not been processed yet. During the searches, the nodes have three possible states:

- state 0: the node has not been processed (white)
- state 1: the node is under processing (light gray)
- state 2: the node has been processed (dark gray)

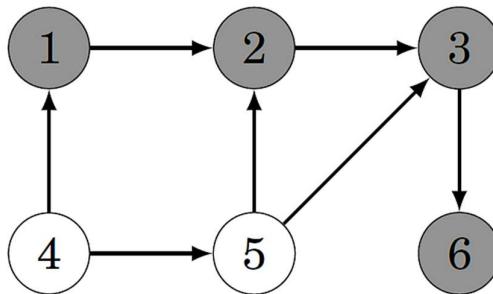
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2. If the graph contains a cycle, we will find this out during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort.

If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

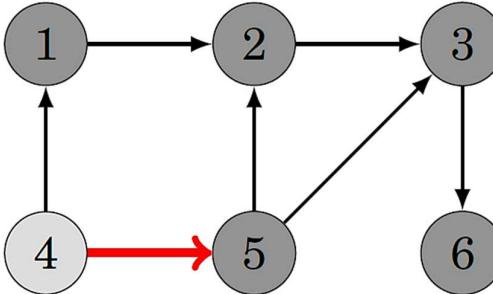
In the example graph, the search first proceeds from node 1 to node 6:



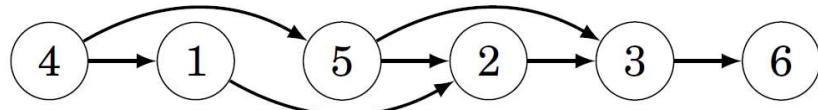
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:



At this point, the list is [6, 3, 2, 1]. The next search begins at node 4:

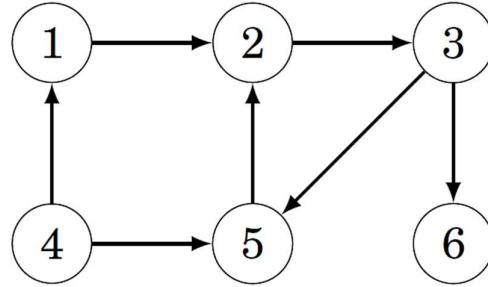


Thus, the final list is [6, 3, 2, 1, 5, 4]. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list [4, 5, 1, 2, 3, 6]:

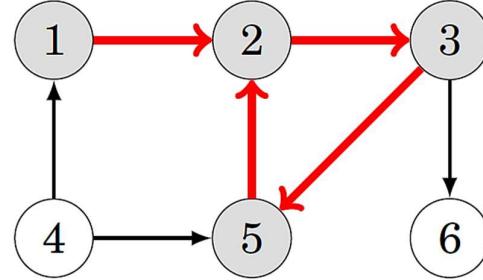


Note that a topological sort is not unique, and there can be several topological sorts for a graph.

Let us now consider a graph for which we cannot construct a topological sort, because the graph contains a cycle:



The search proceeds as follows:



The search reaches node 2 whose state is 1, which means that the graph contains a cycle. In this example, there is a cycle $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

Topological sort for adjacency list using DFS:

```

void topologicalSortUtil(vector<int>* adj, int n, int v, bool visited[],
                         stack<int> &s) {
    visited[v] = true;
    for (auto i = adj[v].begin(); i != adj[v].end(); i++)
        if (!visited[*i])
            topologicalSortUtil(adj, n, *i, visited, s);
    s.push(v);
}

void topologicalSort (vector<int>* adj, int n) {
    stack<int> s;
    bool visited[n] = {false};
    for (int i = 0; i < n; i++)
        if (!visited[i])
            topologicalSortUtil(adj, n, i, visited, s);

    // Printing the stack
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;
}

```

Topological sort for adjacency matrix using DFS:

```
void topologicalSortUtil (int** adj, int n, int curr_vertex,
bool* visited, stack<int> &s) {
    visited[curr_vertex] = true;
    for (int i = 0; i < n; i++) {
        if (adj[curr_vertex][i] != 0 && !visited[i])
            topologicalSortUtil (adj, n, i, visited, s);
    s.push(curr_vertex);
}

void topologicalSort (int** adj, int n) {
    // Everything is same
}
```

Topological sort for adjacency list using BFS:

We can also find Topological Sort using BFS and the algorithm is known as Kahn's algorithm.

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation).

Step-3: Remove a vertex from the queue (Dequeue operation)
and then -

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If in-degree of a neighbouring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

How to find indegree of each node?

Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

Time Complexity: $O(V + E)$

Overall time complexity: $O(V + E)$

```

void topologicalSort (vector<int>* adj, int n) {
    // Find indegree of each node in adj list
    int in_degree[n] = {0};

    for (int i = 0; i < n; i++)
        for (auto itr = adj[i].begin(); itr != adj[i].end(); itr++)
            in_degree[*itr]++;
    // Create a queue and enqueue all vertices with indegree 0
    queue<int> q;
    for (int i = 0; i < n; i++)
        if (in_degree[i] == 0)
            q.push(i);
    int count = 0;
    vector<int> topo_sort;
    // One by one dequeue vertices and enqueue adjacents if
    // indegree of adjacent becomes zero
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topo_sort.push_back(u);

        for (auto itr = adj[u].begin(); itr != adj[u].end(); itr++)
            if (--in_degree[*itr] == 0)
                q.push(*itr);
        count++;
    }
    // Print the topological sort
    if (count != n) {
        cout << "There exists a cycle.\n";
    } else {
        for (int i = 0; i < topo_sort.size(); i++)
            cout << topo_sort[i] << " ";
        cout << endl;
    }
}

```

One follow-up question: Find **all** topological sorts.

1. Initialize all vertices as unvisited.
2. Now choose vertex which is unvisited and has zero indegree and decrease indegree of all those vertices by 1 (corresponding to removing edges). Now add this vertex to result and call the recursive function again and backtrack.
3. After returning from function, reset values of visited, result and indegree for enumeration of other possibilities.

```

void allTopoSortUtil (vector<int>* adj, int n, int indegree[],
bool visited[], vector<int> &res) {
    // flag to indicate whether all topo sorts have been found or not
    bool flag = false;
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0 && !visited[i]) {
            // Reduce the indegree of adjacent vertices

            for (auto j = adj[i].begin(); j != adj[i].end(); j++)
                indegree[*j]--;
            // including in result
            res.push_back(i);
            visited[i] = true;
            allTopoSortUtil(adj, n, indegree, visited,res);
            // Resetting visited, res and indegree for backtracking
            visited[i] = false;
            res.erase(res.end()-1);

            for (auto j = adj[i].begin(); j != adj[i].end(); j++)
                indegree[*j]++;
            flag = true;
        }
    }
    if (!flag) {
        for (int i = 0; i < res.size(); i++)
            cout << res[i] << " ";
        cout << endl;
    }
}

void allTopoSort (vector<int>* adj, int n) {
    bool* visited = new bool[n]();
    vector<int> res;
    int* indegree = new int[n]();

    for (int i = 0; i < n; i++)
        for (auto itr = adj[i].begin(); itr != adj[i].end(); itr++)
            indegree[*itr]++;
    allTopoSortUtil(adj, n, indegree, visited, res);

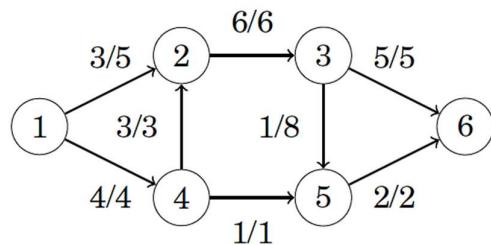
    delete[] visited, indegree;
}

```

10.11 Maximum Flow Problem

In the maximum flow problem, our task is to send as much flow as possible from the source to the sink. The weight of each edge is a capacity that restricts the flow that can go through the edge. In each intermediate node, the incoming and outgoing flow has to be equal.

For example, the maximum size of a flow in the example graph is 7. The following picture shows how we can route the flow:

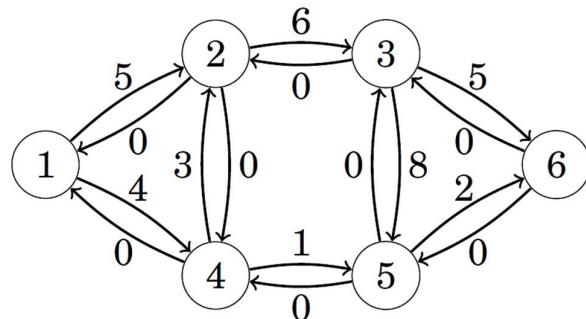


The notation v/k means that a flow of v units is routed through an edge whose capacity is k units. The size of the flow is 7, because the source sends 3+4 units of flow and the sink receives 5+2 units of flow. It is easy to see that this flow is maximum, because the total capacity of the edges leading to the sink is 7.

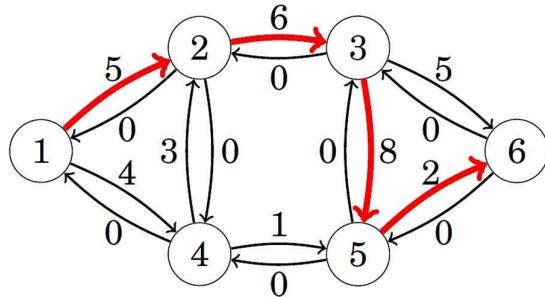
The **Ford–Fulkerson algorithm** finds the maximum flow in a graph. The algorithm begins with an empty flow, and at each step finds a path from the source to the sink that generates more flow. Finally, when the algorithm cannot increase the flow anymore, the maximum flow has been found.

The algorithm uses a special representation of the graph where each original edge has a reverse edge in another direction. The weight of each edge indicates how much more flow we could route through it. At the beginning of the algorithm, the weight of each original edge equals the capacity of the edge and the weight of each reverse edge is zero.

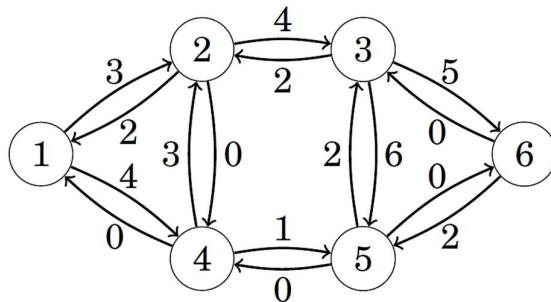
The new representation for the example graph is as follows:



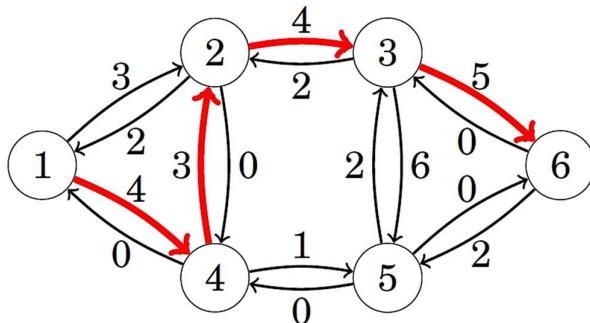
The Ford–Fulkerson algorithm consists of several rounds. On each round, the algorithm finds a path from the source to the sink such that each edge on the path has a positive weight. If there is more than one possible path available, we can choose any of them. For example, suppose we choose the following path:



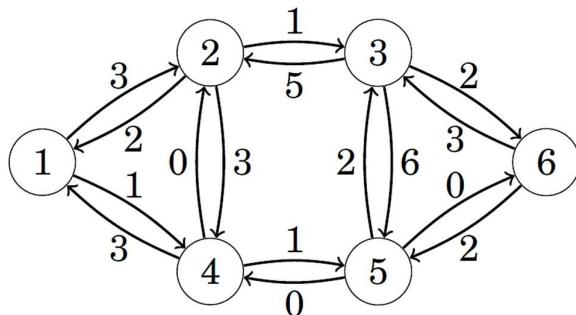
After choosing the path, the flow increases by x units, where x is the smallest edge weight on the path. In addition, the weight of each edge on the path decreases by x and the weight of each reverse edge increases by x . In the above path, the weights of the edges are 5, 6, 8 and 2. The smallest weight is 2, so the flow increases by 2 and the new graph is as follows:



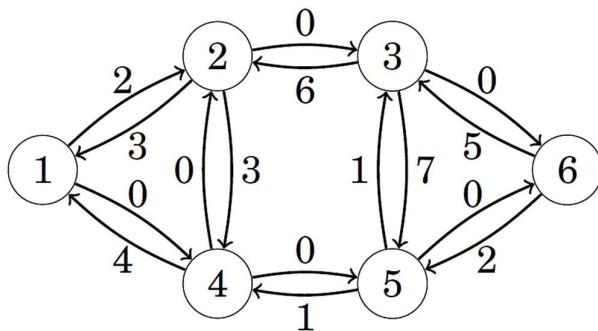
The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to cancel flow later using the reverse edges of the graph if it turns out that it would be beneficial to route the flow in another way. The algorithm increases the flow as long as there is a path from the source to the sink through positive-weight edges. In the present example, our next path can be as follows:



The minimum edge weight on this path is 3, so the path increases the flow by 3, and the total flow after processing the path is 5. The new graph will be as follows:



We still need two more rounds before reaching the maximum flow. For example, we can choose the paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ and $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$. Both paths increase the flow by 1, and the final graph is as follows:



It is not possible to increase the flow anymore, because there is no path from the source to the sink with positive edge weights. Hence, the algorithm terminates and the maximum flow is 7.

```
#define V 6
// Following function returns true if there is a path from source 's'
// to sink 't' in residual graph. Also fills parent[] to store the path.
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
    // Create a visited array and mark all vertices as not visited
    bool* visited = new bool[V]();
    // Create a queue, enqueue source vertex and mark source vertex as
    // visited
    queue<int> q; q.push(s);
    visited[s] = true; parent[s] = -1;
    // Standard BFS Loop
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v=0; v<V; v++) {
            if (visited[v]==false && rGraph[u][v] > 0) {
                q.push(v); parent[v] = u;
                visited[v] = true;
            }
        }
    }
}
```

```

    // If we reached sink in BFS starting from source, then return true.
    return (visited[t] == true);
}

int fordFulkerson(int graph[V][V], int s, int t) {
    int u, v;
    // Create a residual graph and fill the residual graph with given
    // capacities in the original graph as residual capacities in
    // residual graph
    int rGraph[V][V];

    // Residual graph where rGraph[i][j] indicates residual capacity
    // of edge from i to j (if there is an edge. If rGraph[i][j] is 0,
    // then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path
    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent)) {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
        // update residual capacities and reverse edges along the path
        for (v=t; v != s; v=parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

```

```
int main() {
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0} };
    cout << "Maximum possible flow is " << fordFulkerson(graph, 0, 5);
    return 0;
}
```
