Contents

Chapter 1

An Interesting Method to Generate Binary Numbers from 1 to n

An Interesting Method to Generate Binary Numbers from 1 to n - GeeksforGeeks

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, call decimal to binary inside the loop.

Following is an interesting method that uses queue data structureto print binary numbers. Thanks to Vivekfor suggesting this approach.
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.

Following is implementation of above algorithm.

C++

```
 // C++ program to generate binary numbers from 1 to n
#include <iostream>
#include <queue>
using namespace std;

// This function uses queue data structure to print binary numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
    queue<string> q;

    // Enqueue the first binary number
    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"));
    }
}

// Driver program to test above function
int main()
{
    int n = 10;
    generatePrintBinary(n);
    return 0;
}
```

Java

```
 //Java program to generate binary numbers from 1 to n

import java.util.LinkedList;
import java.util.Queue;
```

```
public class GenerateBNo
{
    // This function uses queue data structure to print binary numbers
    static void generatePrintBinary(int n)
    {
        // Create an empty queue of strings
        Queue<String> q = new LinkedList<String>();

        // Enqueue the first binary number
        q.add("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-- > 0)
        {
            // print the front of queue
            String s1 = q.peek();
            q.remove();
            System.out.println(s1);

            // Store s1 before changing it
            String s2 = s1;

            // Append "0" to s1 and enqueue it
            q.add(s1 + "0");

            // Append "1" to s2 and enqueue it. Note that s2 contains
            // the previous front
            q.add(s2 + "1");
        }
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int n=10;
        generatePrintBinary(n);
    }
}
//This code is contributed by Sumit Ghosh
```

Python

```
 # Python program to generate binary numbers from
# 1 to n

# This function uses queu data structure to print binary numbers
```

```
def generatePrintBinary(n):

    # Create an empty queue
    from Queue import Queue
    q = Queue()

    # Enqueu the first binary number
    q.put("1")

    # This loop is like BFS of a tree with 1 as root
    # 0 as left child and 1 as right child and so on
    while(n>0):
        n-= 1
        # Print the front of queue
        s1 = q.get()
        print s1

        s2 = s1 # Store s1 before changing it

        # Append "0" to s1 and enqueue it
        q.put(s1+"0")

        # Append "1" to s2 and enqueue it. Note that s2
        # contains the previous front
        q.put(s2+"1")


# Driver program to test above function
n = 10
generatePrintBinary(n)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
1
10
11
100
101
110
111
1000
1001
1010
```

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/interesting-method-generate-binary-numbers-1-n/

Chapter 2

Applications of Priority Queue

Applications of Priority Queue - GeeksforGeeks

A Priority Queue is different from a normal queue, because instead of being a "first-in-first-out", values come out in order by priority. It is an abstract data type that captures the idea of a container whose elements have "priorities" attached to them. An element of highest priority always appears at the front of the queue. If that element is removed, the next highest priority element advances to the front.

A priority queue is typically implemented using Heap data structure.

Applications:

Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

Data compression: It is used in Huffman codes which is used to compresses data.

Artificial Intelligence : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Operating systems: It is also use in Operating System for load balancing (load balancing on server), interrupt handling.

Source

https://www.geeksforgeeks.org/applications-priority-queue/

Chapter 3

Applications of Queue Data Structure

Applications of Queue Data Structure - GeeksforGeeks

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See thisfor more detailed applications of Queue and Stack.

References:
http://introcs.cs.princeton.edu/43stack/

Source

https://www.geeksforgeeks.org/applications-of-queue-data-structure/

Chapter 4

Averages of Levels in Binary Tree

Averages of Levels in Binary Tree - GeeksforGeeks

Given a non-empty binary tree, print the average value of the nodes on each level.

Examples:

```
Input :
    4
   / \
  2   9
 / \   \
3   5   7

Output : [4 5.5 5]
The average value of nodes on level 0 is 4,
on level 1 is 5.5, and on level 2 is 5.
Hence, print [4 5.5 5].
```

The idea is based on Level order traversal line by line | Set 2 (Using Two Queues)

1. Start by pushing the root node into the queue. Then, remove a node from the front of the queue.
2. For every node removed from the queue, push all its children into a new temporary queue.
3. Keep on popping nodes from the queue and adding these node' s children to the temporary queue till queue becomes empty.
4. Every time queue becomes empty, it indicates that one level of the tree has been considered.

5. While pushing the nodes into temporary queue, keep a track of the sum of the nodes along with the number of nodes pushed and find out the average of the nodes on each level by making use of these sum and count values.
6. After each level has been considered, again initialize the queue with temporary queue and continue the process till both queues become empty.

```
 // C++ program to find averages of all levels
// in a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int val;
    struct Node* left, *right;
};

/* Function to print the average value of the
   nodes on each level */
void averageOfLevels(Node* root)
{
    vector<float> res;

    // Traversing level by level
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {

        // Compute sum of nodes and
        // count of nodes in current
        // level.
        int sum = 0, count = 0;
        queue<Node*> temp;
        while (!q.empty()) {
            Node* n = q.front();
            q.pop();
            sum += n->val;
            count++;
            if (n->left != NULL)
                temp.push(n->left);
            if (n->right != NULL)
                temp.push(n->right);
        }
        q = temp;
        cout << (sum * 1.0 / count) << " ";
    }
```

```
}

/* Helper function that allocates a
   new node with the given data and
   NULL left and right pointers. */
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver code
int main()
{
    /* Let us construct a Binary Tree
        4
       / \
      2   9
     / \   \
    3   5   7 */

    Node* root = NULL;
    root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(9);
    root->left->left = newNode(3);
    root->left->right = newNode(8);
    root->right->right = newNode(7);
    averageOfLevels(root);
    return 0;
}
```

Output:

```
Average of levels:
[4 5.5 5]
```

Complexity Analysis:

- Time complexity : O(n).
  The whole tree is traversed atmost once. Here, n refers to the number of nodes in the given binary tree.
- Auxiliary Space : O(n).
  The size of queues can grow upto atmost the maximum number of nodes at any level in the given binary tree. Here, n refers to the maximum number of nodes at any level in the input tree.

Source

https://www.geeksforgeeks.org/averages-levels-binary-tree/

Chapter 5

Bank Of America (BA Continuum India Pvt. Ltd.) Campus Recruitment

Bank Of America (BA Continuum India Pvt. Ltd.) Campus Recruitment - GeeksforGeeks

Approved Offer.

Bank Of America has visited our college for on campus recruitment . The recruitment consisted of 4 Rounds in total.

The recruitment was for BA Continuum India Pvt Ltd. the technical field of BOA

Round 1:

This round was a general aptitude test, English proficiency, quantitative analysis and the logical questions. This was time specific f0r each and every section. Try practicing from GFG, indiabix and careerride youtube videos.

Round 2: TECHNICAL 1

This was a face to face technical interview. I was asked to submit my resume and then was asked to wait until my name was called out. The interview went around 30-40 minutes.

The interviewer started with a very basic aptitude problem(dices probability) and then he asked me a puzzle of water jug problem. Then he asked me to explain my projects in the resume and about the experience I had in the previous organization.

After that he went forward with the technical questions consisting of CODING problems .

1. What is recursion? Find the length of a linked list using recursion
   This was to test the basics of the datastructure
2. Balanced Parentheses problem
3. Find the position of the first unbalanced parentheses?

Round 3: TECHNICAL 2

It was another technical round. This round was based on advanced coding.

1. Base conversion. Given a number in 10 format convert it into base of 6?
2. Matrix generation.
3. Backtracking question

The interviewers above solely focused on the approach or the pseudo code and helped if we got stuck somewhere.

Round 4: HR ROUND

Here the HR was very friendly and asked about whether you are able to relocate if asked?

At last I was given a feedback by the interviewers as strong logical and coding skills . I received the offer letter on that day itself at night.

Position: Senior Tech Associate, Bank Of America

Source

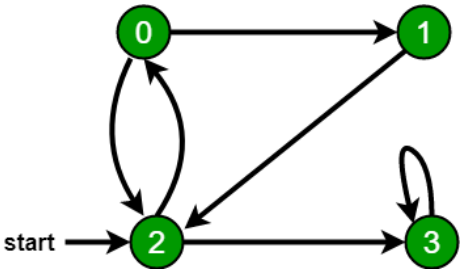
https://www.geeksforgeeks.org/bank-of-america-ba-continuum-india-pvt-ltd-campus-recruitment/

Chapter 6

Breadth First Search or BFS for a Graph

Breadth First Search or BFS for a Graph - GeeksforGeeks

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are the implementations of simple Breadth First Traversal from a given source.

The implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
```

```
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);
```

```
    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Java

```
 // Java program to print BFS traversal from a given source vertex.
```

```
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
```

```
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }

    // Driver method to
    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "+
                           "(starting from vertex 2)");

        g.BFS(2);
    }
}
// This code is contributed by Aakash Hasija
```

Python3

```
 # Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):
```

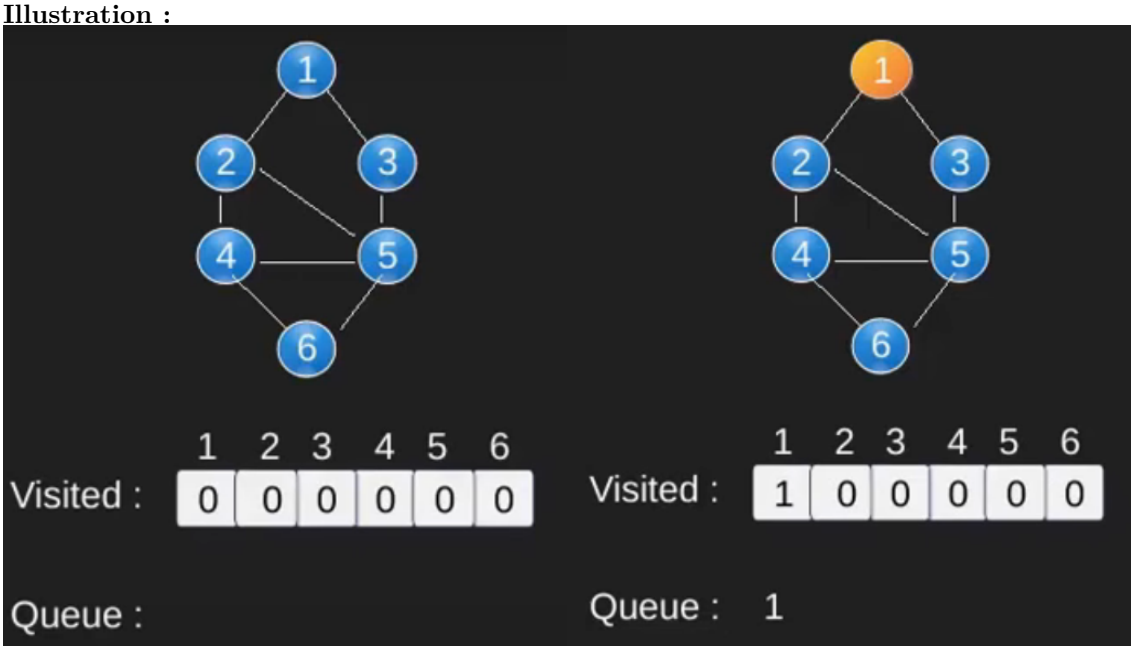
```
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
```

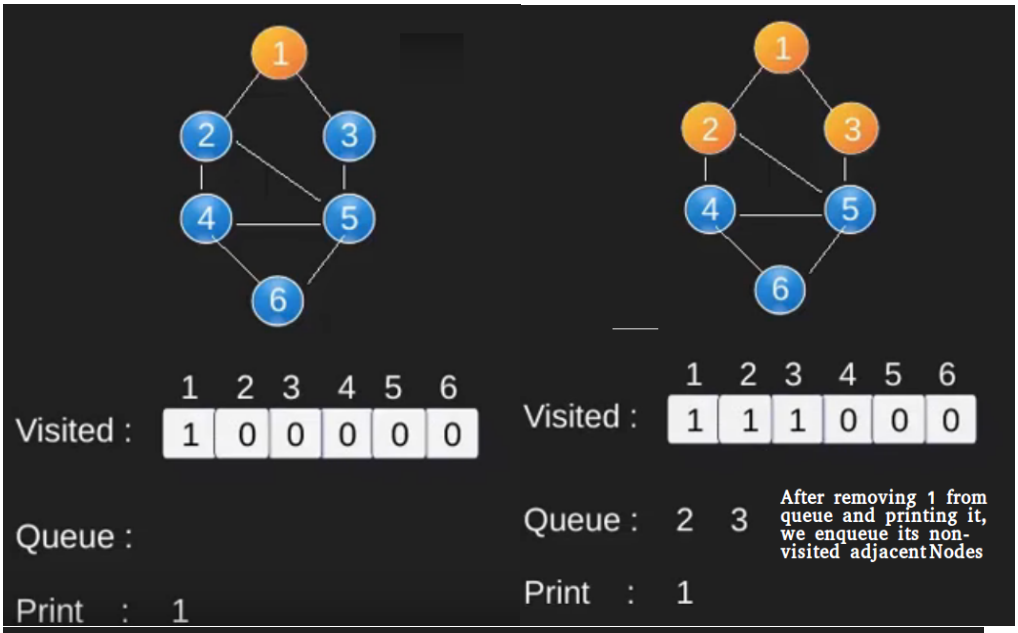
```
print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)

# This code is contributed by Neelam Yadav
```

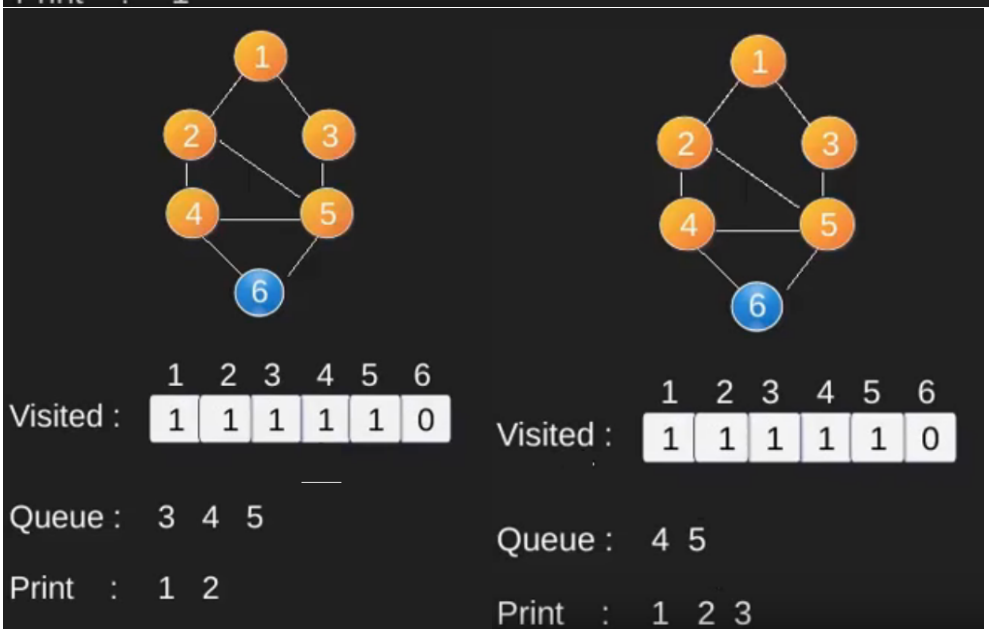
Output:

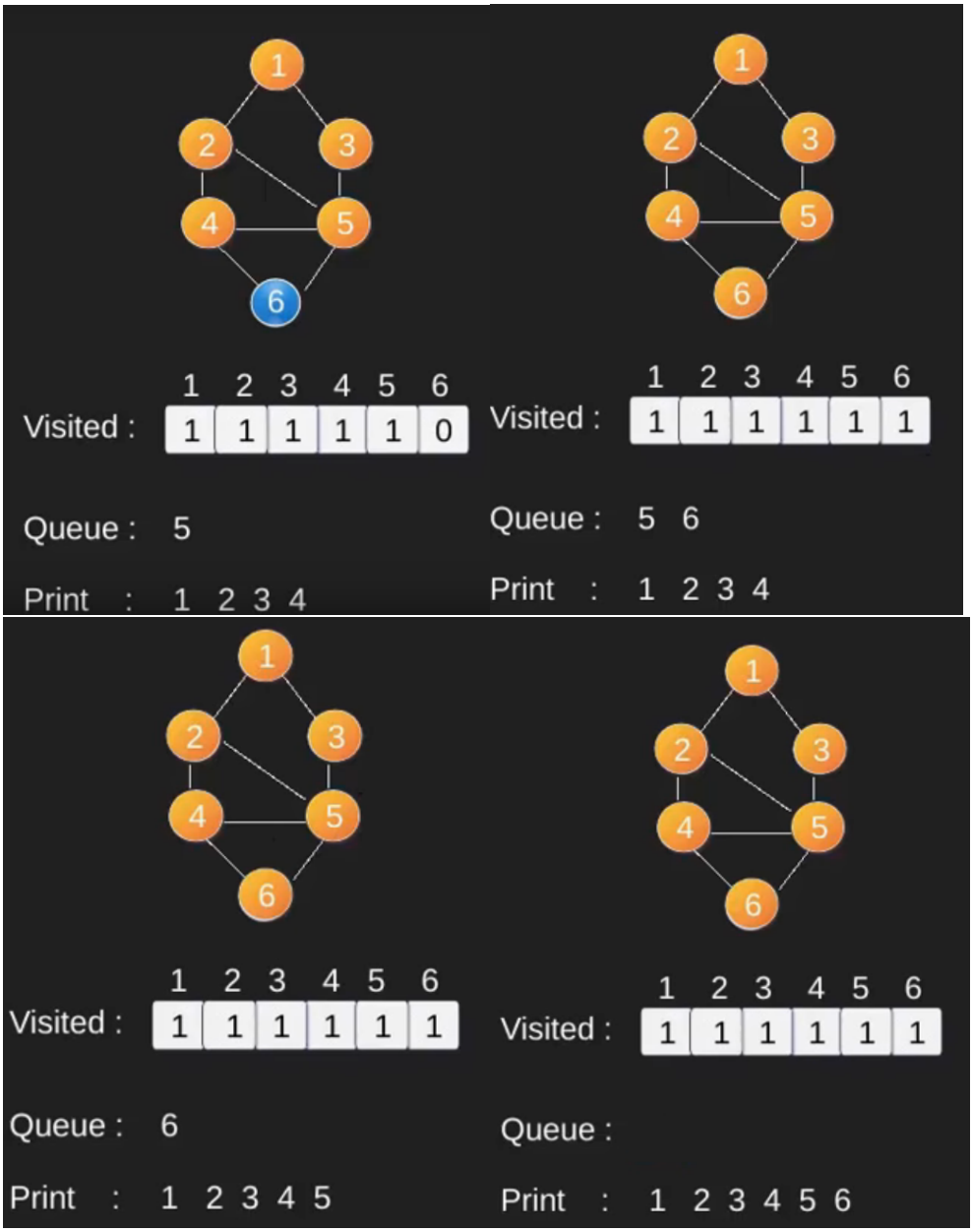
```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Illustration :




1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue :
Print : 1
1 2 3 4 5 6
Visited : 1 1 1 0 0 0
Queue : 2 3
After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes
Print : 1


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 3 4 5
Print : 1 2
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 4 5
Print : 1 2 3



Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the DFS modified version) .

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

You may like to see below also :

- Recent Articles on BFS
- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search

Source

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

Chapter 7

Check if a queue can be sorted into another queue using a stack

Check if a queue can be sorted into another queue using a stack - GeeksforGeeks

Given a Queue consisting of first **n** natural numbers (in random order). The task is to check whether the given Queue elements can be arranged in increasing order in another Queue using a stack. The operation allowed are:
1. Push and pop elements from the stack
2. Pop (Or enqueue) from the given Queue.
3. Push (Or Dequeue) in the another Queue.

Examples :

Input : Queue[] = { 5, 1, 2, 3, 4 }
Output : Yes
Pop the first element of the given Queue i.e 5.
Push 5 into the stack.
Now, pop all the elements of the given Queue and push them to
second Queue.
Now, pop element 5 in the stack and push it to the second Queue.

Input : Queue[] = { 5, 1, 2, 6, 3, 4 }
Output : No
Push 5 to stack.
Pop 1, 2 from given Queue and push it to another Queue.
Pop 6 from given Queue and push to stack.
Pop 3, 4 from given Queue and push to second Queue.
Now, from using any of above operation, we cannot push 5
into the second Queue because it is below the 6 in the stack.

Observe, second Queue (which will contain the sorted element) takes inputs (or enqueue elements) either from given Queue or Stack. So, next expected (which will initially be 1)

element must be present as a front element of given Queue or top element of the Stack. So, simply simulate the process for the second Queue by initializing the expected element as 1. And check if we can get expected element from the front of the given Queue or from the top of the Stack. If we cannot take it from the either of them then pop the front element of given Queue and push it in the Stack.
Also, observe, that the stack must also be sorted at each instance i.e the element at the top of the stack must be smallest in the stack. For eg. let x > y, then x will always be expected before y. So, x cannot be pushed before y in the stack. Therefore, we cannot push element with the higher value on the top of the element having lesser value.

Algorithm:
1. Initialize the expected_element = 1
2. Check if either front element of given Queue or top element of the stack have expected_element
....a) If yes, increment expected_element by 1, repeat step 2.
....b) Else, pop front of Queue and push it to the stack. If the popped element is greater than top of the Stack, return "No".

Below is the implementation of this approach:

C++

```
 // CPP Program to check if a queue of first
// n natural number can be sorted using a stack
#include <bits/stdc++.h>
using namespace std;

// Function to check if given queue element
// can be sorted into another queue using a
// stack.
bool checkSorted(int n, queue<int>& q)
{
    stack<int> st;
    int expected = 1;
    int fnt;

    // while given Queue is not empty.
    while (!q.empty()) {
        fnt = q.front();
        q.pop();

        // if front element is the expected element
        if (fnt == expected)
            expected++;

        else {
            // if stack is empty, push the element
            if (st.empty()) {
```

```
                st.push(fnt);
            }

            // if top element is less than element which
            // need to be pushed, then return fasle.
            else if (!st.empty() && st.top() < fnt) {
                return false;
            }

            // else push into the stack.
            else
                st.push(fnt);
        }

        // while expected element are coming from
        // stack, pop them out.
        while (!st.empty() && st.top() == expected) {
            st.pop();
            expected++;
        }
    }

    // if the final expected element value is equal
    // to initial Queue size and the stack is empty.
    if (expected - 1 == n && st.empty())
        return true;

    return false;
}

// Driven Program
int main()
{
    queue<int> q;
    q.push(5);
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);

    int n = q.size();

    (checkSorted(n, q) ? (cout << "Yes") :
                         (cout << "No"));

    return 0;
}
```

Java

```
 // Java Program to check if a queue
// of first n natural number can
// be sorted using a stack
import java.io.*;
import java.util.*;

class GFG
{
    static Queue<Integer> q =
                    new LinkedList<Integer>();

    // Function to check if given
    // queue element can be sorted
    // into another queue using a stack.
    static boolean checkSorted(int n)
    {
        Stack<Integer> st =
                    new Stack<Integer>();
        int expected = 1;
        int fnt;

        // while given Queue
        // is not empty.
        while (q.size() != 0)
        {
            fnt = q.peek();
            q.poll();

            // if front element is
            // the expected element
            if (fnt == expected)
                expected++;

            else
            {
                // if stack is empty,
                // push the element
                if (st.size() == 0)
                {
                    st.push(fnt);
                }

                // if top element is less than
                // element which need to be
                // pushed, then return fasle.
                else if (st.size() != 0 &&
```

```
                        st.peek() < fnt)
                {
                    return false;
                }

                // else push into the stack.
                else
                    st.push(fnt);
            }

            // while expected element are
            // coming from stack, pop them out.
            while (st.size() != 0 &&
                   st.peek() == expected)
            {
                st.pop();
                expected++;
            }
        }

        // if the final expected element
        // value is equal to initial Queue
        // size and the stack is empty.
        if (expected - 1 == n &&
                st.size() == 0)
            return true;

        return false;
    }

    // Driver Code
    public static void main(String args[])
    {
        q.add(5);
        q.add(1);
        q.add(2);
        q.add(3);
        q.add(4);

        int n = q.size();

        if (checkSorted(n))
            System.out.print("Yes");
        else
            System.out.print("No");
    }
}
```

```
// This code is contributed by
// Manish Shaw(manishshaw1)
```

C#

```
 // C# Program to check if a queue
// of first n natural number can
// be sorted using a stack
using System;
using System.Linq;
using System.Collections.Generic;

class GFG
{
    // Function to check if given
    // queue element can be sorted
    // into another queue using a stack.
    static bool checkSorted(int n,
                            ref Queue<int> q)
    {
        Stack<int> st = new Stack<int>();
        int expected = 1;
        int fnt;

        // while given Queue
        // is not empty.
        while (q.Count != 0)
        {
            fnt = q.Peek();
            q.Dequeue();

            // if front element is
            // the expected element
            if (fnt == expected)
                expected++;

            else
            {
                // if stack is empty,
                // push the element
                if (st.Count != 0)
                {
                    st.Push(fnt);
                }

                // if top element is less than
                // element which need to be
                // pushed, then return fasle.
```

```
                else if (st.Count != 0 &&
                         st.Peek() < fnt)
                {
                    return false;
                }

                // else push into the stack.
                else
                    st.Push(fnt);
            }

            // while expected element are
            // coming from stack, pop them out.
            while (st.Count != 0 &&
                   st.Peek() == expected)
            {
                st.Pop();
                expected++;
            }
        }
        // if the final expected element
        // value is equal to initial Queue
        // size and the stack is empty.
        if (expected - 1 == n &&
                st.Count == 0)
            return true;

        return false;
    }

    // Driver Code
    static void Main()
    {
        Queue<int> q = new Queue<int>();
        q.Enqueue(5);
        q.Enqueue(1);
        q.Enqueue(2);
        q.Enqueue(3);
        q.Enqueue(4);

        int n = q.Count;

        if (checkSorted(n, ref q))
            Console.Write("Yes");
        else
            Console.Write("No");
    }
}
```

```
// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output :

```
Yes
```

Video Contributed by Parul Shandilya

Improved By : manishshaw1, ParulShandilya, Abdul Mohsin

Source

https://www.geeksforgeeks.org/check-queue-can-sorted-another-queue-using-stack/

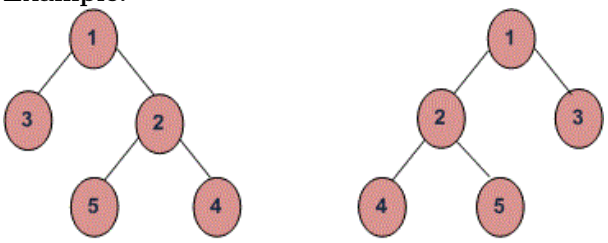
Chapter 8

Check if all levels of two trees are anagrams or not

Check if all levels of two trees are anagrams or not - GeeksforGeeks

Given two binary trees, we have to check if each of their levels are anagrams of each other or not.

Example:




```
Tree 1:
Level 0 : 1
Level 1 : 3, 2
Level 2 : 5, 4

Tree 2:
Level 0 : 1
Level 1 : 2, 3
Level 2 : 4, 5
```

As we can clearly see all the levels of above two binary trees are anagrams of each other, hence return true.

Naive Approach: Below is the step by step explanation of the naive approach to do this:

1. Write a recursive program for level order traversal of a tree.
2. Traverse each level of both the trees one by one and store the result of traversals in 2 different vectors, one for each tree.
3. Sort both the vectors and compare them iteratively for each level, if they are same for each level then return true else return false.

Time Complexity: O(n^2), where n is the number of nodes.

Efficient Approach:
The idea is based on below article.
Print level order traversal line by line | Set 1
We traverse both trees simultaneously level by level. We store each level both trees in vectors (or array). To check if two vectors are anagram or not, we sort both and then compare.

Time Complexity: O(n), where n is the number of nodes.

C++

```
 /* Iterative program to check if two trees are level
   by level anagram. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int data;
};

// Returns true if trees with root1 and root2
// are level by level anagram, else returns false.
bool areAnagrams(Node *root1, Node *root2)
{
    // Base Cases
    if (root1 == NULL && root2 == NULL)
        return true;
    if (root1 == NULL || root2 == NULL)
        return false;

    // start level order traversal of two trees
    // using two queues.
    queue<Node *> q1, q2;
    q1.push(root1);
    q2.push(root2);

    while (1)
    {
```

```
        // n1 (queue size) indicates number of Nodes
        // at current level in first tree and n2 indicates
        // number of nodes in current level of second tree.
        int n1 = q1.size(), n2 = q2.size();

        // If n1 and n2 are different
        if (n1 != n2)
            return false;

        // If level order traversal is over
        if (n1 == 0)
            break;

        // Dequeue all Nodes of current level and
        // Enqueue all Nodes of next level
        vector<int> curr_level1, curr_level2;
        while (n1 > 0)
        {
            Node *node1 = q1.front();
            q1.pop();
            if (node1->left != NULL)
                q1.push(node1->left);
            if (node1->right != NULL)
                q1.push(node1->right);
            n1--;

            Node *node2 = q2.front();
            q2.pop();
            if (node2->left != NULL)
                q2.push(node2->left);
            if (node2->right != NULL)
                q2.push(node2->right);

            curr_level1.push_back(node1->data);
            curr_level2.push_back(node2->data);
        }

        // Check if nodes of current levels are
        // anagrams or not.
        sort(curr_level1.begin(), curr_level1.end());
        sort(curr_level2.begin(), curr_level2.end());
        if (curr_level1 != curr_level2)
            return false;
    }

    return true;
}
```

```
// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Constructing both the trees.
    struct Node* root1 = newNode(1);
    root1->left = newNode(3);
    root1->right = newNode(2);
    root1->right->left = newNode(5);
    root1->right->right = newNode(4);

    struct Node* root2 = newNode(1);
    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    areAnagrams(root1, root2)? cout << "Yes" : cout << "No";
    return 0;
}
```

Java

```
 /* Iterative program to check if two trees
are level by level anagram. */
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.Queue;


public class GFG
{
    // A Binary Tree Node
    static class Node
    {
        Node left, right;
        int data;
        Node(int data){
            this.data = data;
```

```
        left = null;
        right = null;
    }
}

// Returns true if trees with root1 and root2
// are level by level anagram, else returns false.
static boolean areAnagrams(Node root1, Node root2)
{
    // Base Cases
    if (root1 == null && root2 == null)
        return true;
    if (root1 == null || root2 == null)
        return false;

    // start level order traversal of two trees
    // using two queues.
    Queue<Node> q1 = new LinkedList<Node>();
    Queue<Node> q2 = new LinkedList<Node>();
    q1.add(root1);
    q2.add(root2);

    while (true)
    {
        // n1 (queue size) indicates number of
        // Nodes at current level in first tree
        // and n2 indicates number of nodes in
        // current level of second tree.
        int n1 = q1.size(), n2 = q2.size();

        // If n1 and n2 are different
        if (n1 != n2)
            return false;

        // If level order traversal is over
        if (n1 == 0)
            break;

        // Dequeue all Nodes of current level and
        // Enqueue all Nodes of next level
        ArrayList<Integer> curr_level1 = new
                                    ArrayList<>();
        ArrayList<Integer> curr_level2 = new
                                    ArrayList<>();
        while (n1 > 0)
        {
            Node node1 = q1.peek();
            q1.remove();
```

```
            if (node1.left != null)
                q1.add(node1.left);
            if (node1.right != null)
                q1.add(node1.right);
            n1--;

            Node node2 = q2.peek();
            q2.remove();
            if (node2.left != null)
                q2.add(node2.left);
            if (node2.right != null)
                q2.add(node2.right);

            curr_level1.add(node1.data);
            curr_level2.add(node2.data);
        }

        // Check if nodes of current levels are
        // anagrams or not.
        Collections.sort(curr_level1);
        Collections.sort(curr_level2);

        if (!curr_level1.equals(curr_level2))
            return false;
    }

    return true;
}

// Driver program to test above functions
public static void main(String args[])
{
    // Constructing both the trees.
    Node root1 = new Node(1);
    root1.left = new Node(3);
    root1.right = new Node(2);
    root1.right.left = new Node(5);
    root1.right.right = new Node(4);

    Node root2 = new Node(1);
    root2.left = new Node(2);
    root2.right = new Node(3);
    root2.left.left = new Node(4);
    root2.left.right = new Node(5);


    System.out.println(areAnagrams(root1, root2)?
                        "Yes" : "No");
```

```
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
Yes
```

Note: In the above program we are comparing the vectors storing each level of a tree directly using not equal to function ' != ' which compares the vectors first on the basis of their size and then on the basis of their content, hence saving our work of iteratively comparing the vectors.

Source

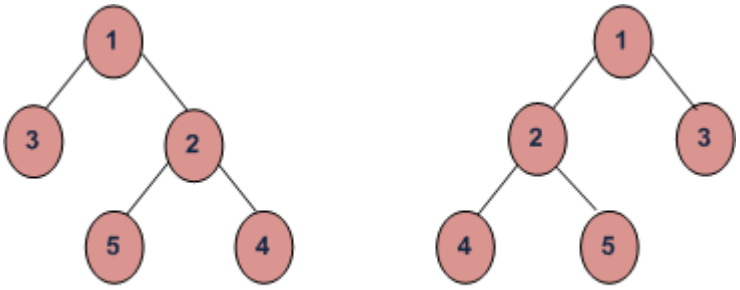
https://www.geeksforgeeks.org/check-if-all-levels-of-two-trees-are-anagrams-or-not/

Chapter 9

Check if two trees are Mirror

Check if two trees are Mirror - GeeksforGeeks

Given two Binary Trees, write a function that returns true if two trees are mirror of each other, else false. For example, the function should return true for following input trees.



Mirror Trees

This problem is different from the problem discussed here.

For two trees 'a' and 'b' to be mirror images, the following three conditions must be true:

1. Their root node's key must be same
2. Left subtree of root of 'a' and right subtree root of 'b' are mirror.
3. Right subtree of 'a' and left subtree of 'b' are mirror.

Below is implementation of above idea.

C++

```
 // C++ program to check if two trees are mirror
// of each other
#include<bits/stdc++.h>
using namespace std;
```

```
/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node
{
    int data;
    Node* left, *right;
};

/* Given two trees, return true if they are
   mirror of each other */
int areMirror(Node* a, Node* b)
{
    /* Base case : Both empty */
    if (a==NULL && b==NULL)
        return true;

    // If only one is empty
    if (a==NULL || b == NULL)
        return false;

    /* Both non-empty, compare them recursively
     Note that in recursive calls, we pass left
     of one tree and right of other tree */
    return  a->data == b->data &&
            areMirror(a->left, b->right) &&
            areMirror(a->right, b->left);
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data  = data;
    node->left  =  node->right  = NULL;
    return(node);
}

/* Driver program to test areMirror() */
int main()
{
    Node *a = newNode(1);
    Node *b = newNode(1);
    a->left = newNode(2);
    a->right = newNode(3);
    a->left->left  = newNode(4);
    a->left->right = newNode(5);
```

```
    b->left = newNode(3);
    b->right = newNode(2);
    b->right->left = newNode(5);
    b->right->right = newNode(4);

    areMirror(a, b)? cout << "Yes" : cout << "No";

    return 0;
}
```

Java

```
 // Java program to see if two trees
// are mirror of each other

// A binary tree node
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node a, b;

    /* Given two trees, return true if they are
       mirror of each other */
    boolean areMirror(Node a, Node b)
    {
        /* Base case : Both empty */
        if (a == null && b == null)
            return true;

        // If only one is empty
        if (a == null || b == null)
            return false;

        /* Both non-empty, compare them recursively
           Note that in recursive calls, we pass left
           of one tree and right of other tree */
        return a.data == b.data
```

```
                && areMirror(a.left, b.right)
                && areMirror(a.right, b.left);
    }

    // Driver code to test above methods
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        Node a = new Node(1);
        Node b = new Node(1);
        a.left = new Node(2);
        a.right = new Node(3);
        a.left.left = new Node(4);
        a.left.right = new Node(5);

        b.left = new Node(3);
        b.right = new Node(2);
        b.right.left = new Node(5);
        b.right.right = new Node(4);

        if (tree.areMirror(a, b) == true)
            System.out.println("Yes");
        else
            System.out.println("No");

    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Python3

```
 # Python3 program to check if two
# trees are mirror of each other

# A binary tree node
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Given two trees, return true
# if they are mirror of each other
def areMirror(a, b):

    # Base case : Both empty
    if a is None and b is None:
```

```
        return True

    # If only one is empty
    if a is None or b is None:
        return False

    # Both non-empty, compare them
    # recursively. Note that in
    # recursive calls, we pass left
    # of one tree and right of other tree
    return (a.data == b.data and
            areMirror(a.left, b.right) and
            areMirror(a.right , b.left))

# Driver code
root1 = Node(1)
root2 = Node(1)

root1.left = Node(2)
root1.right = Node(3)
root1.left.left = Node(4)
root1.left.right = Node(5)

root2.left = Node(3)
root2.right = Node(2)
root2.right.left = Node(5)
root2.right.right = Node(4)

if areMirror(root1, root2):
    print ("Yes")
else:
    print ("No")

# This code is contributed by AshishR
```

Output :

```
Yes
```

Time Complexity : O(n)

Iterative method to check if two trees are mirror of each other

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : AshishR

Source

https://www.geeksforgeeks.org/check-if-two-trees-are-mirror/

Chapter 10

Check mirror in n-ary tree

Check mirror in n-ary tree - GeeksforGeeks

Given two n-ary trees, the task is to check if they are mirror of each other or not. Print "Yes" if they are mirror of each other else "No".

Examples:

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : Yes
```

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : No
```

The idea is to use Queue and Stack to check if given N-ary tree are mirror of each other or not.
Let first n-ary tree be t1 and second n-ary tree is t2. For each node in t1, make stack and push its connected node in it. Now, for each node in t2, make queue and push its connected node in it.
Now, for each corresponding node do following:

```
  While stack and Queue is not empty.
  a = top element of stack;
```

```
  b = front of stack;
  if (a != b)
    return false;
  pop element from stack and queue.
```

```
 // C++ program to check if two n-ary trees are
// mirror.
#include <bits/stdc++.h>
using namespace std;

// First vector stores all nodes and adjacent of every
// node in a stack.
// Second vector stores all nodes and adjacent of every
// node in a queue.
bool mirrorUtil(vector<stack<int> >& tree1,
                      vector<queue<int> >& tree2)
{
    // Traversing each node in tree.
    for (int i = 1; i < tree1.size(); ++i) {
        stack<int>& s = tree1[i];
        queue<int>& q = tree2[i];

        // While stack is not empty && Queue is not empty
        while (!s.empty() && !q.empty()) {

            // checking top element of stack and front
            // of queue.
            if (s.top() != q.front())
                return false;

            s.pop();
            q.pop();
        }

        // If queue or stack is not empty, return false.
        if (!s.empty() || !q.empty())
            return false;
    }

    return true;
}

// Returns true if given two trees are mirrors.
// A tree is represented as two arrays to store
// all tree edges.
void areMirrors(int m, int n, int u1[], int v1[],
                            int u2[], int v2[])
{
```

```
    vector<stack<int> > tree1(m + 1);
    vector<queue<int> > tree2(m + 1);

    // Pushing node in the stack of first tree.
    for (int i = 0; i < n; i++)
        tree1[u1[i]].push(v1[i]);

    // Pushing node in the queue of second tree.
    for (int i = 0; i < n; i++)
        tree2[u2[i]].push(v2[i]);


    mirrorUtil(tree1, tree2) ? (cout << "Yes" << endl) :
                               (cout << "No" << endl);
}

// Driver code
int main()
{
    int M = 3, N = 2;

    int u1[] = { 1, 1 };
    int v1[] = { 2, 3 };

    int u2[] = { 1, 1 };
    int v2[] = { 3, 2 };

    areMirrors(M, N, u1, v1, u2, v2);

    return 0;
}
```

Output:

```
Yes
```

Reference: https://practice.geeksforgeeks.org/problems/check-mirror-in-n-ary-tree/0

Source

https://www.geeksforgeeks.org/check-mirror-n-ary-tree/

Chapter 11

Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution)

Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution) - GeeksforGeeks

Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. See following examples.

```
The following trees are examples of Complete Binary Trees
    1
  /   \
 2     3

       1
    /    \
   2       3
  /
 4

       1
    /    \
   2      3
  /  \    /
 4    5  6
```

```
The following trees are examples of Non-Complete Binary Trees
    1
      \
       3

       1
     /    \
   2       3
    \     /  \
     4   5    6

       1
     /    \
   2      3
         /  \
        4    5
```

The method 2 of level order traversal post can be easily modified to check whether a tree is Complete or not. To understand the approach, let us first define a term 'Full Node'. A node is 'Full Node' if both left and right children are not empty (or not NULL).
The approach is to do a level order traversal starting from root. In the traversal, once a node is found which is NOT a Full Node, all the following nodes must be leaf nodes.
Also, one more thing needs to be checked to handle the below case: If a node has empty left child, then the right child must be empty.

```
    1
  /   \
 2     3
  \
   4
```

Thanks to Guddu Sharma for suggesting this simple and efficient approach.

C/C++

```
 // A program to check if a given binary tree is complete or not
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
```

```
    struct node* right;
};

/* function prototypes for functions needed for Queue data
   structure. A queue is needed for level order tarversal */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isQueueEmpty(int *front, int *rear);

/* Given a binary tree, return true if the tree is complete
   else false */
bool isCompleteBT(struct node* root)
{
  // Base Case: An empty tree is complete Binary Tree
  if (root == NULL)
    return true;

  // Create an empty queue
  int rear, front;
  struct node **queue = createQueue(&front, &rear);

  // Create a flag variable which will be set true
  // when a non full node is seen
  bool flag = false;

  // Do level order traversal using queue.
  enQueue(queue, &rear, root);
  while(!isQueueEmpty(&front, &rear))
  {
    struct node *temp_node = deQueue(queue, &front);

    /* Check if left child is present*/
    if(temp_node->left)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if (flag == true)
         return false;

       enQueue(queue, &rear, temp_node->left);  // Enqueue Left Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;

    /* Check if right child is present*/
    if(temp_node->right)
```

```
    {
       // If we have seen a non full node, and we see a node
       // with non-empty right child, then the given tree is not
       // a complete Binary Tree
       if(flag == true)
         return false;

       enQueue(queue, &rear, temp_node->right);  // Enqueue Right Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;
  }

  // If we reach here, then the tree is complete Bianry Tree
  return true;
}


/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
  struct node **queue =
   (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

  *front = *rear = 0;
  return queue;
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
  queue[*rear] = new_node;
  (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
  (*front)++;
  return queue[*front - 1];
}

bool isQueueEmpty(int *front, int *rear)
{
   return (*rear == *front);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
```

```
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
   /* Let us construct the following Binary Tree which
      is not a complete Binary Tree
            1
          /   \
         2     3
        / \     \
       4   5     6
    */

  struct node *root  = newNode(1);
  root->left         = newNode(2);
  root->right        = newNode(3);
  root->left->left   = newNode(4);
  root->left->right  = newNode(5);
  root->right->right = newNode(6);

  if ( isCompleteBT(root) == true )
      printf ("Complete Binary Tree");
  else
      printf ("NOT Complete Binary Tree");

  return 0;
}
```

Java

```
 //A Java program to check if a given binary tree is complete or not

import java.util.LinkedList;
import java.util.Queue;

public class CompleteBTree
{
    /* A binary tree node has data, pointer to left child
       and a pointer to right child */
```

```
static class Node
{
    int data;
    Node left;
    Node right;

    // Constructor
    Node(int d)
    {
        data = d;
        left = null;
        right = null;
    }
}

/* Given a binary tree, return true if the tree is complete
   else false */
static boolean isCompleteBT(Node root)
{
    // Base Case: An empty tree is complete Binary Tree
    if(root == null)
        return true;

    // Create an empty queue
    Queue<Node> queue =new LinkedList<>();

    // Create a flag variable which will be set true
    // when a non full node is seen
    boolean flag = false;

    // Do level order traversal using queue.
    queue.add(root);
    while(!queue.isEmpty())
    {
        Node temp_node = queue.remove();

        /* Check if left child is present*/
        if(temp_node.left != null)
        {
             // If we have seen a non full node, and we see a node
             // with non-empty left child, then the given tree is not
             // a complete Binary Tree
            if(flag == true)
                return false;

             // Enqueue Left Child
            queue.add(temp_node.left);
        }
```

```
        // If this a non-full node, set the flag as true
        else
            flag = true;

        /* Check if right child is present*/
        if(temp_node.right != null)
        {
            // If we have seen a non full node, and we see a node
            // with non-empty right child, then the given tree is not
            // a complete Binary Tree
            if(flag == true)
                return false;

            // Enqueue Right Child
            queue.add(temp_node.right);

        }
        // If this a non-full node, set the flag as true
        else
            flag = true;
    }
     // If we reach here, then the tree is complete Bianry Tree
    return true;
}

/* Driver program to test above functions*/
public static void main(String[] args)
{

    /* Let us construct the following Binary Tree which
      is not a complete Binary Tree
            1
          /   \
         2     3
        / \     \
       4   5     6
    */

    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.right = new Node(6);

    if(isCompleteBT(root) == true)
        System.out.println("Complete Binary Tree");
    else
```

```
            System.out.println("NOT Complete Binary Tree");
     }

}
//This code is contributed by Sumit Ghosh
```

Python

```
 # Check whether binary tree is complete or not

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Given a binary tree, return true if the tree is complete
# else return false
def isCompleteBT(root):

    # Base Case: An empty tree is complete Binary tree
    if root is None:
        return True

    # Create an empty queue
    queue = []

    # Create a flag variable which will be set Trye
    # when a non ful node is seen
    flag = False

    # Do level order traversal using queue
    queue.append(root)
    while(len(queue) > 0):
        tempNode = queue.pop(0) # Dequeue

        # Check if left child is present
        if (tempNode.left):

            # If we have seen a non full node, and we see
            # a node with non-empty left child, then the
            # given tree is not a complete binary tree
            if flag == True :
                return False
```

```
            # Enqueue left child
            queue.append(tempNode.left)

            # If this a non-full node, set the flag as true
        else:
            flag = True

        # Check if right cild is present
        if(tempNode.right):

            # If we have seen a non full node, and we
            # see a node with non-empty right child, then
            # the given tree is not a compelete BT
            if flag == True:
                return False

            # Enqueue right child
            queue.append(tempNode.right)

        # If this is non-full node, set the flag as True
        else:
            flag = True

    # If we reach here, then the tree is compelete BT
    return True


# Driver program to test above function

""" Let us construct the following Binary Tree which
      is not a complete Binary Tree
            1
          /   \
         2     3
        / \     \
       4   5     6
    """
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(6)

if (isCompleteBT(root)):
    print "Complete Binary Tree"
else:
    print "NOT Complete Binary Tree"
```

```
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
NOT Complete Binary Tree
```

Time Complexity: O(n) where n is the number of nodes in given Binary Tree
Auxiliary Space: O(n) for queue.

Source

https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/

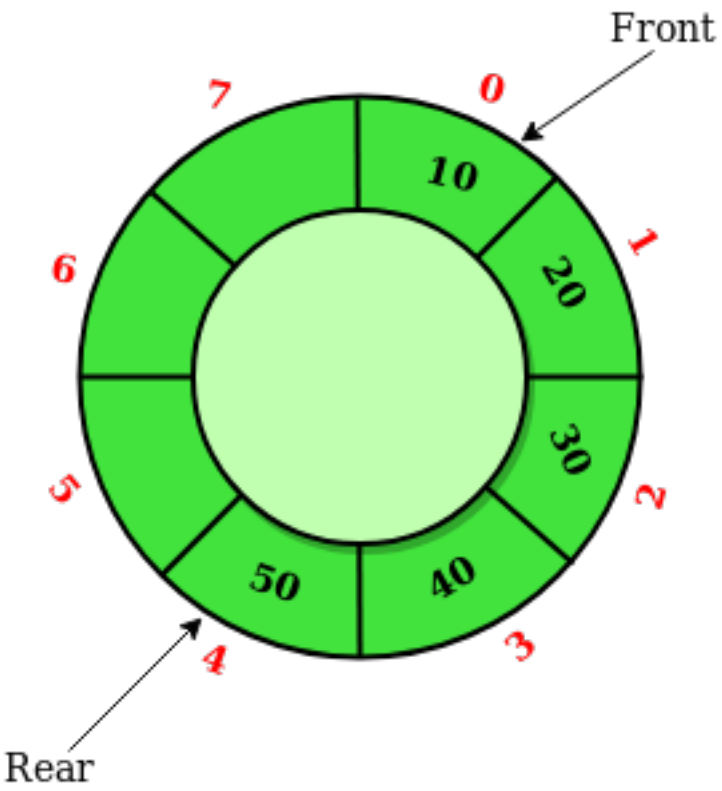
Chapter 12

Circular Queue | Set 1 (Introduction and Array Implementation)

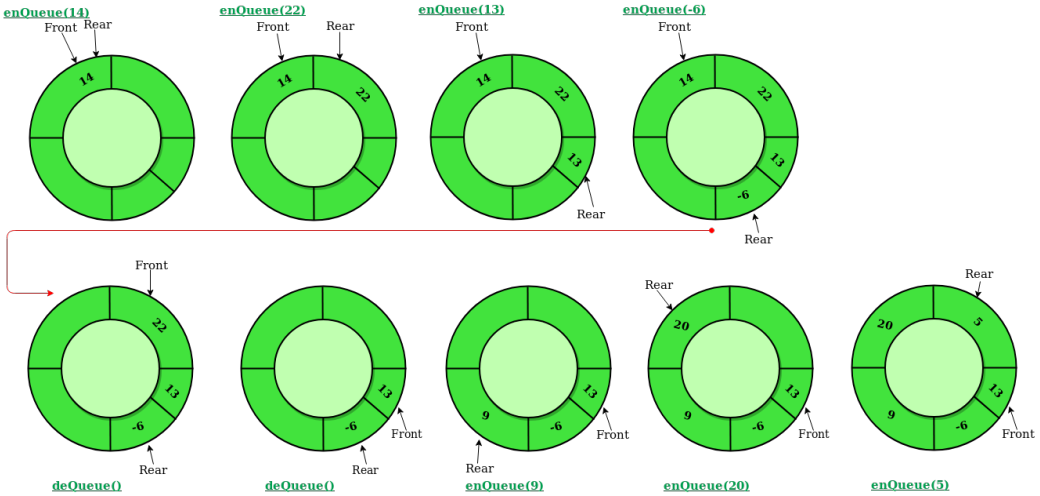
Circular Queue | Set 1 (Introduction and Array Implementation) - GeeksforGeeks

Prerequisite – Queues

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

- **Front:** Get the front item from queue.

- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
    1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
    2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
    1. Check whether queue is Empty means check (front==-1).
    2. If it is empty then display Queue is empty. If queue is not empty then step 3
    3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

```
 // C or C++ program for insertion and
// deletion in Circular Queue
#include<bits/stdc++.h>
using namespace std;

struct Queue
{
    // Initialize front and rear
    int rear, front;

    // Circular Queue
    int size;
    int *arr;

    Queue(int s)
    {
       front = rear = -1;
       size = s;
       arr = new int[s];
    }

    void enQueue(int value);
    int deQueue();
    void displayQueue();
};


/* Function to create Circular queue */
void Queue::enQueue(int value)
{
    if ((front == 0 && rear == size-1) ||
            (rear == (front-1)%(size-1)))
```

```
    {
        printf("\nQueue is Full");
        return;
    }

    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
        arr[rear] = value;
    }

    else if (rear == size-1 && front != 0)
    {
        rear = 0;
        arr[rear] = value;
    }

    else
    {
        rear++;
        arr[rear] = value;
    }
}

// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return INT_MIN;
    }

    int data = arr[front];
    arr[front] = -1;
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
    else
        front++;

    return data;
}
```

```
// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i <= rear; i++)
            printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i < size; i++)
            printf("%d ", arr[i]);

        for (int i = 0; i <= rear; i++)
            printf("%d ", arr[i]);
    }
}

/* Driver of the program */
int main()
{
    Queue q(5);

    // Inserting elements in Circular Queue
    q.enQueue(14);
    q.enQueue(22);
    q.enQueue(13);
    q.enQueue(-6);

    // Display elements present in Circular Queue
    q.displayQueue();

    // Deleting elements from Circular Queue
    printf("\nDeleted value = %d", q.deQueue());
    printf("\nDeleted value = %d", q.deQueue());

    q.displayQueue();

    q.enQueue(9);
    q.enQueue(20);
    q.enQueue(5);
```

```
    q.displayQueue();

    q.enQueue(20);
    return 0;
}
```

Output:

```
Elements in Circular Queue are: 14 22 13 -6
Deleted value = 14
Deleted value = 22
Elements in Circular Queue are: 13 -6
Elements in Circular Queue are: 13 -6 9 20 5
Queue is Full
```

Time Complexity: Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

Applications:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Improved By : Arpit Gaurav

Source

https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/

Chapter 13

Circular Queue | Set 2 (Circular Linked List Implementation)

Circular Queue | Set 2 (Circular Linked List Implementation) - GeeksforGeeks

Prerequisite – Circular Singly Linked List

We have discussed basics and how to implement circular queue using array in set 1.
Circular Queue | Set 1 (Introduction and Array Implementation)

In this post another method of circular queue implementation is discussed, using Circular Singly Linked List.

Operations on Circular Queue:

- **Front:**Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
    1. Create a new node dynamically and insert value into it.
    2. Check if front==NULL, if it is true then front = rear = (newly created node)
    3. If it is false then rare=(newly created node) and rear node always contains the address of the front node.
- **deQueue()** This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.
    1. Check whether queue is empty or not means front == NULL.
    2. If it is empty then display Queue is empty. If queue is not empty then step 3
    3. Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.

```
Elements in Circular Queue are: 14 22 6
Deleted value = 14
Deleted value = 22
Elements in Circular Queue are: 6
Elements in Circular Queue are: 6 9 20
```

Time Complexity: Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

Note : In case of linked list implementation, a queue can be easily implemented without being circular. However in case of array implementation, we need a circular queue to save space.

Source

https://www.geeksforgeeks.org/circular-queue-set-2-circular-linked-list-implementation/

Chapter 14

Connect n ropes with minimum cost

Connect n ropes with minimum cost - GeeksforGeeks

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.
1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5.
2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.

If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This approach is similar to Huffman Coding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes.
Let there be n ropes of lengths stored in an array len[0..n-1]
1) Create a min heap and insert all lengths into the min heap.
2) Do following while number of elements in min heap is not one.
……a) Extract the minimum and second minimum from min heap
……b) Add the above two extracted values and insert the added value to the min-heap.
……c) Maintain a variable for total cost and keep incrementing it by the sum of extracted

values.
3) Return the value of this total cost.

Following is C++ implementation of above algorithm.

```
 // C++ program for connecting n ropes with minimum cost
#include <iostream>

using namespace std;

// A Min Heap:  Collection of min heap nodes
struct MinHeap
{
    unsigned size;    // Current size of min heap
    unsigned capacity;   // capacity of min heap
    int *harr;  // Attay of minheap nodes
};

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap = new MinHeap;
    minHeap->size = 0;  // current size is 0
    minHeap->capacity = capacity;
    minHeap->harr = new int[capacity];
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->harr[left] < minHeap->harr[smallest])
      smallest = left;

    if (right < minHeap->size &&
        minHeap->harr[right] < minHeap->harr[smallest])
```

```
      smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->harr[smallest], &minHeap->harr[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
int extractMin(struct MinHeap* minHeap)
{
    int temp = minHeap->harr[0];
    minHeap->harr[0] = minHeap->harr[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, int val)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && (val < minHeap->harr[(i - 1)/2]))
    {
        minHeap->harr[i] = minHeap->harr[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->harr[i] = val;
}

// A standard funvtion to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// Creates a min heap of capacity equal to size and inserts all values
```

```
// from len[] in it. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(int len[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->harr[i] = len[i];
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that returns the minimum cost to connect n ropes of
// lengths stored in len[0..n-1]
int minCost(int len[], int n)
{
    int cost = 0;  // Initialize result

    // Create a min heap of capacity equal to n and put all ropes in it
    struct MinHeap* minHeap = createAndBuildMinHeap(len, n);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Extract two minimum length ropes from min heap
        int min     = extractMin(minHeap);
        int sec_min = extractMin(minHeap);

        cost += (min + sec_min);  // Update total cost

        // Insert a new rope in min heap with length equal to sum
        // of two extracted minimum lengths
        insertMinHeap(minHeap, min+sec_min);
    }

    // Finally return total minimum cost for connecting all ropes
    return cost;
}

// Driver program to test above functions
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}
```

Output:

```
Total cost for connecting ropes is 29
```

Time Complexity: Time complexity of the algorithm is O(nLogn) assuming that we use a O(nLogn) sorting algorithm. Note that heap operations like insert and extract take O(Logn) time.

Algorithmic Paradigm: Greedy Algorithm

A simple implementation with STL in C++
Following is a simple implementation that uses priority_queue available in STL. Thanks to Pango89 for providing below code.

C++

```
 #include<iostream>
#include<queue>

using namespace std;

int minCost(int arr[], int n)
{
    // Create a priority queue ( http://www.cplusplus.com/reference/queue/priority_queue/ )
    // By default 'less' is used which is for decreasing order
    // and 'greater' is used for increasing order
    priority_queue< int, vector<int>, greater<int> > pq(arr, arr+n);

    // Initialize result
    int res = 0;

    // While size of priority queue is more than 1
    while (pq.size() > 1)
    {
        // Extract shortest two ropes from pq
        int first = pq.top();
        pq.pop();
        int second = pq.top();
        pq.pop();

        // Connect the ropes: update result and
        // insert the new rope to pq
        res += first + second;
        pq.push(first + second);
    }

    return res;
}

// Driver program to test above function
```

```
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}
```

Java

```
 // Java program to connect n
// ropes with minimum cost
import java.util.*;

class ConnectRopes
{
static int minCost(int arr[], int n)
{
    // Create a priority queue
    PriorityQueue<Integer> pq =
                        new PriorityQueue<Integer>();

    // Adding items to the pQueue
    for(int i=0;i<n;i++)
    {
        pq.add(arr[i]);
    }

    // Initialize result
    int res = 0;

    // While size of priority queue
    // is more than 1
    while (pq.size() > 1)
    {
        // Extract shortest two ropes from pq
        int first = pq.poll();
        int second = pq.poll();

        // Connect the ropes: update result
        // and insert the new rope to pq
        res += first + second;
        pq.add(first + second);
    }

    return res;
}
```

```
// Driver program to test above function
public static void main(String args[])
{
    int len[] = {4, 3, 2, 6};
    int size = len.length;
    System.out.println("Total cost for connecting"+
                " ropes is " + minCost(len, size));

}
}
// This code is contributed by yash_pec
```

Output:

```
Total cost for connecting ropes is 29
```

This article is compiled by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : JAGRITIBANSAL, AbhijeetSrivastava

Source

https://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/

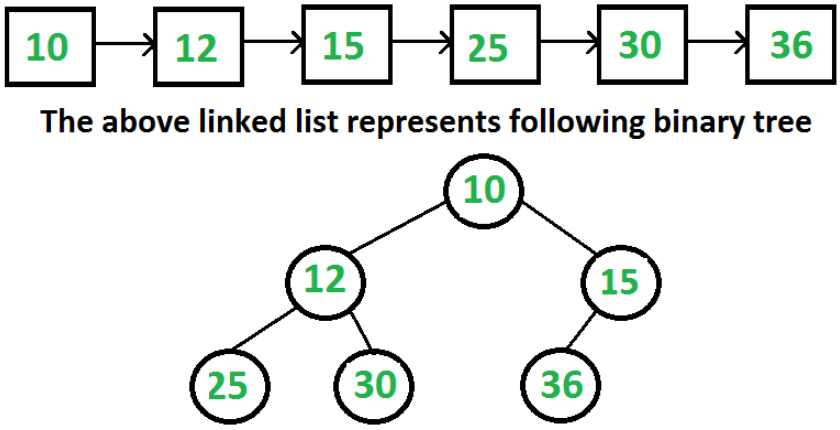
Chapter 15

Construct Complete Binary Tree from its Linked List Representation

Construct Complete Binary Tree from its Linked List Representation - GeeksforGeeks

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i, its left, and right children are stored at indices 2*i+1, 2*i+2 respectively.
Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.



We are mainly given level order traversal in sequential access form. We know head of linked list is always is root of the tree. We take the first node as root and we also know that the

next two nodes are left and right children of root. So we know partial Binary Tree. The idea is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

1. Create an empty queue.
2. Make the first node of the list as root, and enqueue it to the queue.
3. Until we reach the end of the list, do the following.
……**a.** Dequeue one node from the queue. This is the current parent.
……**b.** Traverse two nodes in the list, add them as children of the current parent.
……**c.** Enqueue the two nodes into the queue.

Below is the code which implements the same in C++.

C++

```
 // C++ program to create a Complete Binary tree from its Linked List
// Representation
#include <iostream>
#include <string>
#include <queue>
using namespace std;

// Linked list node
struct ListNode
{
    int data;
    ListNode* next;
};

// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
};

// Function to insert a node at the beginning of the Linked List
void push(struct ListNode** head_ref, int new_data)
{
    // allocate node and assign data
    struct ListNode* new_node = new ListNode;
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
```

```
    (*head_ref)    = new_node;
}

// method to create a new binary tree node from the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
    BinaryTreeNode *temp = new BinaryTreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// converts a given linked list representing a complete binary tree into the
// linked representation of binary tree.
void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
    // queue to store the parent nodes
    queue<BinaryTreeNode *> q;

    // Base Case
    if (head == NULL)
    {
        root = NULL; // Note that root is passed by reference
        return;
    }

    // 1.) The first node is always the root node, and add it to the queue
    root = newBinaryTreeNode(head->data);
    q.push(root);

    // advance the pointer to the next node
    head = head->next;

    // until the end of linked list is reached, do the following steps
    while (head)
    {
        // 2.a) take the parent node from the q and remove it from q
        BinaryTreeNode* parent = q.front();
        q.pop();

        // 2.c) take next two nodes from the linked list. We will add
        // them as children of the current parent node in step 2.b. Push them
        // into the queue so that they will be parents to the future nodes
        BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
        leftChild = newBinaryTreeNode(head->data);
        q.push(leftChild);
        head = head->next;
        if (head)
```

```
        {
            rightChild = newBinaryTreeNode(head->data);
            q.push(rightChild);
            head = head->next;
        }

        // 2.b) assign the left and right children of parent
        parent->left = leftChild;
        parent->right = rightChild;
    }
}

// Utility function to traverse the binary tree after conversion
void inorderTraversal(BinaryTreeNode* root)
{
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->data << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    // create a linked list shown in above diagram
    struct ListNode* head = NULL;
    push(&head, 36);  /* Last node of Linked List */
    push(&head, 30);
    push(&head, 25);
    push(&head, 15);
    push(&head, 12);
    push(&head, 10); /* First node of Linked List */

    BinaryTreeNode *root;
    convertList2Binary(head, root);

    cout << "Inorder Traversal of the constructed Binary Tree is: \n";
    inorderTraversal(root);
    return 0;
}
```

Java

```
 // Java program to create complete Binary Tree from its Linked List
// representation
```

```
// importing necessary classes
import java.util.*;

// A linked list node
class ListNode
{
    int data;
    ListNode next;
    ListNode(int d)
    {
        data = d;
        next = null;
    }
}

// A binary tree node
class BinaryTreeNode
{
    int data;
    BinaryTreeNode left, right = null;
    BinaryTreeNode(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    ListNode head;
    BinaryTreeNode root;

    // Function to insert a node at the beginning of
    // the Linked List
    void push(int new_data)
    {
        // allocate node and assign data
        ListNode new_node = new ListNode(new_data);

        // link the old list off the new node
        new_node.next = head;

        // move the head to point to the new node
        head = new_node;
    }

    // converts a given linked list representing a
    // complete binary tree into the linked
```

```
// representation of binary tree.
BinaryTreeNode convertList2Binary(BinaryTreeNode node)
{
    // queue to store the parent nodes
    Queue<BinaryTreeNode> q =
                    new LinkedList<BinaryTreeNode>();

    // Base Case
    if (head == null)
    {
        node = null;
        return null;
    }

    // 1.) The first node is always the root node, and
    //     add it to the queue
    node = new BinaryTreeNode(head.data);
    q.add(node);

    // advance the pointer to the next node
    head = head.next;

    // until the end of linked list is reached, do the
    // following steps
    while (head != null)
    {
        // 2.a) take the parent node from the q and
        //      remove it from q
        BinaryTreeNode parent = q.peek();
        BinaryTreeNode pp = q.poll();

        // 2.c) take next two nodes from the linked list.
        // We will add them as children of the current
        // parent node in step 2.b. Push them into the
        // queue so that they will be parents to the
        // future nodes
        BinaryTreeNode leftChild = null, rightChild = null;
        leftChild = new BinaryTreeNode(head.data);
        q.add(leftChild);
        head = head.next;
        if (head != null)
        {
            rightChild = new BinaryTreeNode(head.data);
            q.add(rightChild);
            head = head.next;
        }

        // 2.b) assign the left and right children of
```

```
            //      parent
            parent.left = leftChild;
            parent.right = rightChild;
        }

        return node;
    }

    // Utility function to traverse the binary tree
    // after conversion
    void inorderTraversal(BinaryTreeNode node)
    {
        if (node != null)
        {
            inorderTraversal(node.left);
            System.out.print(node.data + " ");
            inorderTraversal(node.right);
        }
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.push(36); /* Last node of Linked List */
        tree.push(30);
        tree.push(25);
        tree.push(15);
        tree.push(12);
        tree.push(10); /* First node of Linked List */
        BinaryTreeNode node = tree.convertList2Binary(tree.root);

        System.out.println("Inorder Traversal of the"+
                           " constructed Binary Tree is:");
        tree.inorderTraversal(node);
    }
}
// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to create a Complete Binary Tree from
# its linked list representation

# Linked List node
class ListNode:

        # Constructor to create a new node
```

```
        def __init__(self, data):
            self.data = data
            self.next = None

# Binary Tree Node structure
class BinaryTreeNode:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Class to convert the linked list to Binary Tree
class Conversion:

    # Constructor for storing head of linked list
    # and root for the Binary Tree
    def __init__(self, data = None):
        self.head = None
        self.root = None

    def push(self, new_data):

        # Creating a new linked list node and storing data
        new_node = ListNode(new_data)

        # Make next of new node as head
        new_node.next = self.head

        # Move the head to point to new node
        self.head = new_node

    def convertList2Binary(self):

        # Queue to store the parent nodes
        q = []

        # Base Case
        if self.head is None:
            self.root = None
            return

        # 1.) The first node is always the root node,
        # and add it to the queue
        self.root = BinaryTreeNode(self.head.data)
        q.append(self.root)
```

```
        # Advance the pointer to the next node
        self.head = self.head.next

        # Until th end of linked list is reached, do:
        while(self.head):

            # 2.a) Take the parent node from the q and
            # and remove it from q
            parent = q.pop(0) # Front of queue

            # 2.c) Take next two nodes from the linked list.
            # We will add them as children of the current
            # parent node in step 2.b.
            # Push them into the queue so that they will be
            # parent to the future node
            leftChild= None
            rightChild = None

            leftChild = BinaryTreeNode(self.head.data)
            q.append(leftChild)
            self.head = self.head.next
            if(self.head):
                rightChild = BinaryTreeNode(self.head.data)
                q.append(rightChild)
                self.head = self.head.next

            #2.b) Assign the left and right children of parent
            parent.left = leftChild
            parent.right = rightChild

    def inorderTraversal(self, root):
        if(root):
            self.inorderTraversal(root.left)
            print root.data,
            self.inorderTraversal(root.right)

# Driver Program to test above function

# Object of conversion class
conv = Conversion()
conv.push(36)
conv.push(30)
conv.push(25)
conv.push(15)
conv.push(12)
conv.push(10)

conv.convertList2Binary()
```

```
print "Inorder Traversal of the contructed Binary Tree is:"
conv.inorderTraversal(conv.root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Inorder Traversal of the constructed Binary Tree is:
25 12 30 10 36 15
```

Time Complexity: Time complexity of the above solution is O(n) where n is the number of nodes.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation

Chapter 16

Deque | Set 1 (Introduction and Applications)

Deque | Set 1 (Introduction and Applications) - GeeksforGeeks

Deque or Double Ended Queue is a generalized version of Queue data structurethat allows insert and delete at both ends.

Operations on Deque:
Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported
getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.

Applications of Deque:
Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.
Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque. For example see Maximum of all subarrays of size k problem., 0-1 BFS and Find the first circular tour that visits all petrol pumps.

See wiki pagefor another example of A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.

Language Support:
C++ STL provides implementation of Deque as std::deque and Java provides Deque interface. See thisfor more details.

Deque in Java
Deque in Python

Implementation:
A Deque can be implemented either using a doubly linked list or circular array. In both implementation, we can implement all operations in O(1) time. We will soon be discussing C/C++ implementation of Deque Data structure.

Implementation of Deque using circular array

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Improved By : TanmayS

Source

https://www.geeksforgeeks.org/deque-set-1-introduction-applications/

Chapter 17

Distance of nearest cell having 1 in a binary matrix

Distance of nearest cell having 1 in a binary matrix - GeeksforGeeks

Given a binary matrix of **N x M**, containing at least a value 1. The task is to find the distance of nearest 1 in the matrix for each cell. The distance is calculated as $|\mathbf{i_1} - \mathbf{i_2}| + |\mathbf{j_1} - \mathbf{j_2}|$, where i_1, j_1 are the row number and column number of the current cell and i_2, j_2 are the row number and column number of the nearest cell having value 1.

Examples:

```
Input : N = 3, M = 4
        mat[][] = {
                    0, 0, 0, 1,
                    0, 0, 1, 1,
                    0, 1, 1, 0
                  }
Output : 3 2 1 0
         2 1 0 0
         1 0 0 1

For cell at (0, 0), nearest 1 is at (0, 3),
so distance = (0 - 0) + (3 - 0) = 3.
Similarly all the distance can be calculated.
```

Method 1 (Brute Force):
The idea is to traverse the matrix for each cell and find the minimum distance.

Below is the implementation of this approach:

C++

```
 // C++ program to find distance of nearest
// cell having 1 in a binary matrix.
#include<bits/stdc++.h>
#define N 3
#define M 4
using namespace std;

// Print the distance of nearest cell
// having 1 for each cell.
void printDistance(int mat[N][M])
{
    int ans[N][M];

    // Initalize the answer matrix with INT_MAX.
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            ans[i][j] = INT_MAX;

    // For each cell
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
        {
            // Traversing the whole matrix
            // to find the minimum distance.
            for (int k = 0; k < N; k++)
                for (int l = 0; l < M; l++)
                {
                    // If cell contain 1, check
                    // for minimum distance.
                    if (mat[k][l] == 1)
                        ans[i][j] = min(ans[i][j],
                             abs(i-k) + abs(j-l));
                }
        }

    // Printing the answer.
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
            cout << ans[i][j] << " ";

        cout << endl;
    }
}

// Driven Program
int main()
{
```

```
    int mat[N][M] =
    {
        0, 0, 0, 1,
        0, 0, 1, 1,
        0, 1, 1, 0
    };

    printDistance(mat);

    return 0;
}
```

Java

```
 // Java program to find distance of nearest
// cell having 1 in a binary matrix.

import java.io.*;

class GFG {

    static int N = 3;
    static int M = 4;

    // Print the distance of nearest cell
    // having 1 for each cell.
    static void printDistance(int mat[][])
    {
        int ans[][] = new int[N][M];

        // Initalize the answer matrix with INT_MAX.
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                ans[i][j] = Integer.MAX_VALUE;

        // For each cell
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
            {
                // Traversing the whole matrix
                // to find the minimum distance.
                for (int k = 0; k < N; k++)
                    for (int l = 0; l < M; l++)
                    {
                        // If cell contain 1, check
                        // for minimum distance.
                        if (mat[k][l] == 1)
                            ans[i][j] =
```

```
                            Math.min(ans[i][j],
                                 Math.abs(i-k)
                                 + Math.abs(j-l));
                    }
            }

        // Printing the answer.
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < M; j++)
                System.out.print( ans[i][j] + " ");

            System.out.println();
        }
    }

    // Driven Program
    public static void main (String[] args)
    {
        int mat[][] = { {0, 0, 0, 1},
                        {0, 0, 1, 1},
                        {0, 1, 1, 0} };

        printDistance(mat);
    }
}

// This code is contributed by anuj_67.
```

C#

```
 // C# program to find distance of nearest
// cell having 1 in a binary matrix.

using System;

class GFG {

    static int N = 3;
    static int M = 4;

    // Print the distance of nearest cell
    // having 1 for each cell.
    static void printDistance(int [,]mat)
    {
        int [,]ans = new int[N,M];

        // Initalise the answer matrix with int.MaxValue.
```

```
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                ans[i,j] = int.MaxValue;

        // For each cell
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
            {
                // Traversing thewhole matrix
                // to find the minimum distance.
                for (int k = 0; k < N; k++)
                    for (int l = 0; l < M; l++)
                    {
                        // If cell contain 1, check
                        // for minimum distance.
                        if (mat[k,l] == 1)
                            ans[i,j] =
                            Math.Min(ans[i,j],
                                Math.Abs(i-k)
                                + Math.Abs(j-l));
                    }
            }

        // Printing the answer.
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < M; j++)
                Console.Write( ans[i,j] + " ");

            Console.WriteLine();
        }
    }

    // Driven Program
    public static void Main ()
    {
        int [,]mat = { {0, 0, 0, 1},
                       {0, 0, 1, 1},
                       {0, 1, 1, 0} };

        printDistance(mat);
    }
}

// This code is contributed by anuj_67.
```

PHP

```
 <?php
// PHP program to find distance of nearest
// cell having 1 in a binary matrix.
$N = 3;
$M = 4;

// Print the distance of nearest cell
// having 1 for each cell.
function printDistance( $mat)
{
    global $N,$M;
    $ans = array(array());

    // Initalize the answer
    // matrix with INT_MAX.
    for($i = 0; $i < $N; $i++)
        for ( $j = 0; $j < $M; $j++)
            $ans[$i][$j] = PHP_INT_MAX;

    // For each cell
    for ( $i = 0; $i < $N; $i++)
        for ( $j = 0; $j < $M; $j++)
        {

            // Traversing the whole matrix
            // to find the minimum distance.
            for ($k = 0; $k < $N; $k++)
                for ( $l = 0; $l < $M; $l++)
                {

                    // If cell contain 1, check
                    // for minimum distance.
                    if ($mat[$k][$l] == 1)
                        $ans[$i][$j] = min($ans[$i][$j],
                            abs($i-$k) + abs($j - $l));
                }
        }

    // Printing the answer.
    for ( $i = 0; $i < $N; $i++)
    {
        for ( $j = 0; $j < $M; $j++)
            echo $ans[$i][$j] , " ";

    echo "\n";
    }
}
```

```
    // Driver Code
    $mat = array(array(0, 0, 0, 1),
                 array(0, 0, 1, 1),
                 array(0, 1, 1, 0));

    printDistance($mat);

// This code is contributed by anuj_67.
?>
```

Output:

```
3 2 1 0
2 1 0 0
1 0 0 1
```

Time Complexity: O(N^2*M^2).

Method 2 (using BFS):
The idea is to use multisource Breadth First Search. Consider each cell as a node and each boundary between any two adjacent cells be an edge. Number each cell from 1 to N*M. Now, push all the node whose corresponding cell value is 1 in the matrix in the queue. Apply BFS using this queue to find the minimum distance of the adjacent node.

1. Create a graph with values assigned from 1 to M*N to all vertices. The purpose is to store position and adjacent information.
2. Create an empty queue.
3. Traverse all matrix elements and insert positions of all 1s in queue.
4. Now do a BFS traversal of graph using above created queue. In BFS, we first explore immediate adjacent of all 1's, then adjacent of adjacent, and so on. Therefore we find minimum distance.

Below is C++ implementation of this approach:

```
 // C++ program to find distance of nearest
// cell having 1 in a binary matrix.
#include<bits/stdc++.h>
#define MAX 500
#define N 3
#define M 4
using namespace std;

// Making a class of graph with bfs function.
class graph
```

```
{
private:
    vector<int> g[MAX];
    int n,m;

public:
    graph(int a, int b)
    {
        n = a;
        m = b;
    }

    // Function to create graph with N*M nodes
    // considering each cell as a node and each
    // boundry as an edge.
    void createGraph()
    {
        int k = 1;  // A number to be assigned to a cell

        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j <= m; j++)
            {
                // If last row, then add edge on right side.
                if (i == n)
                {
                    // If not bottom right cell.
                    if (j != m)
                    {
                        g[k].push_back(k+1);
                        g[k+1].push_back(k);
                    }
                }

                // If last column, then add edge toward down.
                else if (j == m)
                {
                    g[k].push_back(k+m);
                    g[k+m].push_back(k);
                }

                // Else make edge in all four direction.
                else
                {
                    g[k].push_back(k+1);
                    g[k+1].push_back(k);
                    g[k].push_back(k+m);
                    g[k+m].push_back(k);
```

```
                }

                k++;
            }
        }
    }

    // BFS function to find minimum distance
    void bfs(bool visit[], int dist[], queue<int> q)
    {
        while (!q.empty())
        {
            int temp = q.front();
            q.pop();

            for (int i = 0; i < g[temp].size(); i++)
            {
                if (visit[g[temp][i]] != 1)
                {
                    dist[g[temp][i]] =
                    min(dist[g[temp][i]], dist[temp]+1);

                    q.push(g[temp][i]);
                    visit[g[temp][i]] = 1;
                }
            }
        }
    }

    // Printing the solution.
    void print(int dist[])
    {
        for (int i = 1, c = 1; i <= n*m; i++, c++)
        {
            cout << dist[i] << " ";

            if (c%m == 0)
                cout << endl;
        }
    }
};

// Find minimum distance
void findMinDistance(bool mat[N][M])
{
    // Creating a graph with nodes values assigned
    // from 1 to N x M and matrix adjacent.
    graph g1(N, M);
```

```
    g1.createGraph();

    // To store minimum distance
    int dist[MAX];

    // To mark each node as visited or not in BFS
    bool visit[MAX] = { 0 };

    // Initalising the value of distance and visit.
    for (int i = 1; i <= M*N; i++)
    {
        dist[i] = INT_MAX;
        visit[i] = 0;
    }

    // Inserting nodes whose value in matrix
    // is 1 in the queue.
    int k = 1;
    queue<int> q;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            if (mat[i][j] == 1)
            {
                dist[k] = 0;
                visit[k] = 1;
                q.push(k);
            }
            k++;
        }
    }

    // Calling for Bfs with given Queue.
    g1.bfs(visit, dist, q);

    // Printing the solution.
    g1.print(dist);
}

// Driven Progarm
int main()
{
    bool mat[N][M] =
    {
        0, 0, 0, 1,
        0, 0, 1, 1,
        0, 1, 1, 0
```

```
    };

    findMinDistance(mat);

    return 0;
}
```

Output :

```
3 2 1 0
2 1 0 0
1 0 0 1
```

Time Complexity: O(N * M).

Improved By : vt_m

Source

https://www.geeksforgeeks.org/distance-nearest-cell-1-binary-matrix/

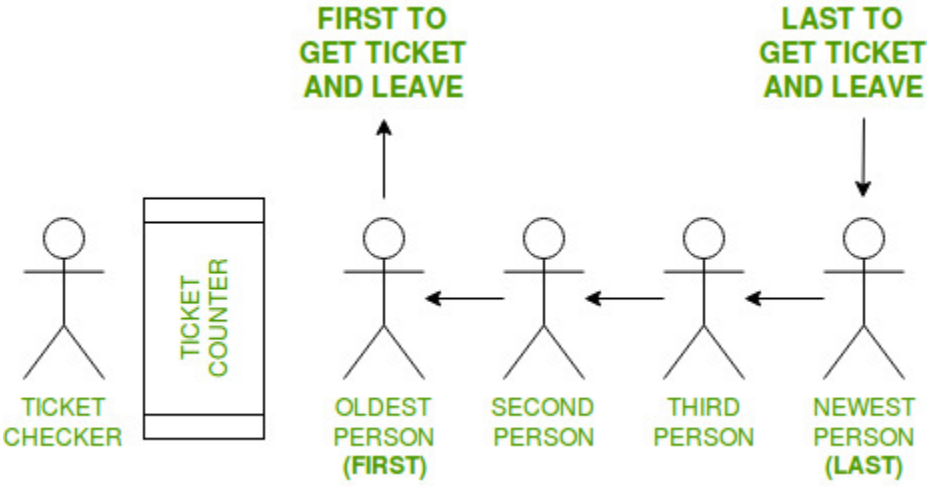
Chapter 18

FIFO (First-In-First-Out) approach in Programming

FIFO (First-In-First-Out) approach in Programming - GeeksforGeeks

FIFO is an abbreviation for **first in, first out**. It is a method for handling data structures where the **first element** is processed first and the **newest element** is processed last.

Real life example:



In this example, following things are to be considered:

- There is a ticket counter where people come, take tickets and go.
- People enter a line (queue) to get to the Ticket Counter in an organized manner.
- The person to enter the queue first, will get the ticket first and leave the queue.
- The person entering the queue next will get the ticket after the person in front of him
- In this way, the person entering the queue last will the tickets last

- Therefore, the First person to enter the queue gets the ticket first and the Last person to enter the queue gets the ticket last.

This is known as First-In-First-Out approach or FIFO.

Where is FIFO used:

1. **Data Structures**

   Certain data structures like Queue and other variants of Queue uses FIFO approach for processing data.

2. **Disk scheduling**

   Disk controllers can use the FIFO as a disk scheduling algorithm to determine the order in which to service disk I/O requests.

3. **Communications and networking**

   Communication network bridges, switches and routers used in computer networks use FIFOs to hold data packets en route to their next destination.

Program Examples for FIFO

Program 1: Queue

```
 // Java program to demonstrate
// working of FIFO
// using Queue interface in Java

import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args)
    {
        Queue<Integer> q = new LinkedList<>();

        // Adds elements {0, 1, 2, 3, 4} to queue
        for (int i = 0; i < 5; i++)
            q.add(i);

        // Display contents of the queue.
        System.out.println("Elements of queue-" + q);

        // To remove the head of queue.
        // In this the oldest element '0' will be removed
        int removedele = q.remove();
        System.out.println("removed element-" + removedele);

        System.out.println(q);
```

```
        // To view the head of queue
        int head = q.peek();
        System.out.println("head of queue-" + head);

        // Rest all methods of collection interface,
        // Like size and contains can be used with this
        // implementation.
        int size = q.size();
        System.out.println("Size of queue-" + size);
    }
}
```

Output:

```
Elements of queue-[0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

Source

https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/

Chapter 19

Find maximum level sum in Binary Tree

Find maximum level sum in Binary Tree - GeeksforGeeks

Given a Binary Tree having positive and negative nodes, the task is to find maximum sum level in it.

Examples:

```
Input :               4
                    /   \
                   2    -5
                  / \    /\
                -1   3 -2  6
Output: 6
Explanation :
Sum of all nodes of 0'th level is 4
Sum of all nodes of 1'th level is -3
Sum of all nodes of 0'th level is 6
Hence maximum sum is 6

Input :           1
                /   \
              2      3
            /  \      \
          4    5      8
                     /   \
                    6     7
Output :  17
```

This problem is a variation of maximum width problem. The idea is to do level order traversal of tree. While doing traversal, process nodes of different level separately. For

every level being processed, compute sum of nodes in the level and keep track of maximum sum.

```
 // A queue based C++ program to find maximum sum
// of a level in Binary Tree
#include<bits/stdc++.h>
using namespace std ;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data ;
    struct Node * left, * right ;
};

// Function to find the maximum sum of a level in tree
// using level order traversal
int maxLevelSum(struct Node * root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = root->data;

    // Do Level order traversal keeping track of number
    // of nodes at every level.
    queue<Node*> q;
    q.push(root);
    while (!q.empty())
    {
        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size() ;

        // Iterate for all the nodes in the queue currently
        int sum = 0;
        while (count--)
        {
            // Dequeue an node from queue
            Node *temp = q.front();
            q.pop();

            // Add this node's value to current sum.
            sum = sum + temp->data;
```

```
            // Enqueue left and right children of
            // dequeued node
            if (temp->left != NULL)
                q.push(temp->left);
            if (temp->right != NULL)
                q.push(temp->right);
        }

        // Update the maximum node count value
        result = max(sum, result);
    }

    return result;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

int main()
{
    struct Node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left  = newNode(6);
    root->right->right->right  = newNode(7);

    /*   Constructed Binary tree is:
                 1
               /   \
             2      3
           /  \      \
          4    5      8
                    /   \
                   6     7    */
    cout << "Maximum level sum is "
         << maxLevelSum(root) << endl;
    return 0;
}
```

Output :

```
Maximum level sum is 17
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Source

https://www.geeksforgeeks.org/find-level-maximum-sum-binary-tree/

Chapter 20

Find maximum vertical sum in binary tree

Find maximum vertical sum in binary tree - GeeksforGeeks

Given a binary tree, find the maximum vertical level sum in binary tree.

Examples:

```
Input :
                3
              /  \
             4    6
           /  \  /  \
         -1   -2 5   10
                  \
                   8

Output : 14
Vertical level having nodes 6 and 8 has maximum
vertical sum 14.

Input :
                 1
               /  \
              5    8
            /  \    \
           2   -6    3
            \       /
            -1     -4
              \
               9
```

```
Output : 4
```

A **simple** solution is to first find vertical level sum of each level starting from minimum vertical level to maximum vertical level. Finding sum of one vertical level takes O(n) time. In worst case time complexity of this solution is O(n^2).

An **efficient** solution is to do level order traversal of given binary tree and update vertical level sum of each level while doing the traversal. After finding vertical sum of each level find maximum vertical sum from these values.

Below is the implementation of above approach:

```
 // CPP program to find maximum vertical
// sum in binary tree.
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// A utility function to create a new
// Binary Tree Node
struct Node* newNode(int item)
{
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to find maximum vertical sum
// in binary tree.
int maxVerticalSum(Node* root)
{
    if (root == NULL) {
        return 0;
    }

    // To store sum of each vertical level.
    unordered_map<int, int> verSum;

    // To store maximum vertical level sum.
    int maxSum = INT_MIN;

    // To store vertical level of current node.
```

```
    int currLev;

    // Queue to perform level order traversal.
    // Each element of queue is a pair of node
    // and its vertical level.
    queue<pair<Node*, int> > q;
    q.push({ root, 0 });

    while (!q.empty()) {

        // Extract node at front of queue
        // and its vertical level.
        root = q.front().first;
        currLev = q.front().second;
        q.pop();

        // Update vertical level sum of
        // vertical level to which
        // current node belongs to.
        verSum[currLev] += root->data;

        if (root->left)
            q.push({ root->left, currLev - 1 });

        if (root->right)
            q.push({ root->right, currLev + 1 });
    }

    // Find maximum vertical level sum.
    for (auto it : verSum)
        maxSum = max(maxSum, it.second);

    return maxSum;
}

// Driver Program to test above functions
int main()
{
    /*
                3
              /  \
             4    6
           /  \  /  \
         -1   -2 5   10
                  \
                   8
    */
```

```
    struct Node* root = newNode(3);
    root->left = newNode(4);
    root->right = newNode(6);
    root->left->left = newNode(-1);
    root->left->right = newNode(-2);
    root->right->left = newNode(5);
    root->right->right = newNode(10);
    root->right->left->right = newNode(8);

    cout << maxVerticalSum(root);
    return 0;
}
```

Output:

```
14
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Source

https://www.geeksforgeeks.org/find-maximum-vertical-sum-in-binary-tree/

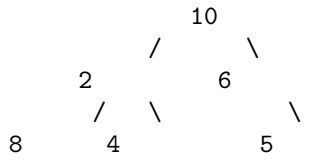
Chapter 21

Find next right node of a given key

Find next right node of a given key - GeeksforGeeks

Given a Binary tree and a key in the binary tree, find the node right to the given key. If there is no node on right side, then return NULL. Expected time complexity is O(n) where n is the number of nodes in the given binary tree.

For example, consider the following Binary Tree. Output for 2 is 6, output for 4 is 5. Output for 10, 6 and 5 is NULL.



Solution: The idea is to do level order traversal of given Binary Tree. When we find the given key, we just check if the next node in level order traversal is of same level, if yes, we return the next node, otherwise return NULL.

C++

```
 /* Program to find next right of a given key */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
```

```
    struct node *left, *right;
    int key;
};

// Method to find next right of given key k, it returns NULL if k is
// not present in tree or k is the rightmost node of its level
node* nextRight(node *root, int k)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order tarversal
    queue<node *> qn; // A queue to store node addresses
    queue<int> ql;   // Another queue to store node levels

    int level = 0;  // Initialize level as 0

    // Enqueue Root and its level
    qn.push(root);
    ql.push(level);

    // A standard BFS loop
    while (qn.size())
    {
        // dequeue an node from qn and its level from ql
        node *node = qn.front();
        level = ql.front();
        qn.pop();
        ql.pop();

        // If the dequeued node has the given key k
        if (node->key == k)
        {
            // If there are no more items in queue or given node is
            // the rightmost node of its level, then return NULL
            if (ql.size() == 0 || ql.front() != level)
               return NULL;

            // Otherwise return next node from queue of nodes
            return qn.front();
        }

        // Standard BFS steps: enqueue children of this node
        if (node->left != NULL)
        {
            qn.push(node->left);
            ql.push(level+1);
```

```
        }
        if (node->right != NULL)
        {
            qn.push(node->right);
            ql.push(level+1);
        }
    }

    // We reach here if given key x doesn't exist in tree
    return NULL;
}

// Utility function to create a new tree node
node* newNode(int key)
{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to test above functions
void test(node *root, int k)
{
    node *nr = nextRight(root, k);
    if (nr != NULL)
      cout << "Next Right of " << k << " is " << nr->key << endl;
    else
      cout << "No next right node found for " << k << endl;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(6);
    root->right->right = newNode(5);
    root->left->left = newNode(8);
    root->left->right = newNode(4);

    test(root, 10);
    test(root, 2);
    test(root, 6);
    test(root, 5);
    test(root, 8);
    test(root, 4);
```

```
    return 0;
}
```

Java

```
 // Java program to find next right of a given key

import java.util.LinkedList;
import java.util.Queue;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Method to find next right of given key k, it returns NULL if k is
    // not present in tree or k is the rightmost node of its level
    Node nextRight(Node first, int k)
    {
        // Base Case
        if (first == null)
            return null;

        // Create an empty queue for level order tarversal
        // A queue to store node addresses
        Queue<Node> qn = new LinkedList<Node>();

        // Another queue to store node levels
        Queue<Integer> ql = new LinkedList<Integer>();

        int level = 0;  // Initialize level as 0

        // Enqueue Root and its level
        qn.add(first);
        ql.add(level);
```

```
    // A standard BFS loop
    while (qn.size() != 0)
    {
        // dequeue an node from qn and its level from ql
        Node node = qn.peek();
        level = ql.peek();
        qn.remove();
        ql.remove();

        // If the dequeued node has the given key k
        if (node.data == k)
        {
            // If there are no more items in queue or given node is
            // the rightmost node of its level, then return NULL
            if (ql.size() == 0 || ql.peek() != level)
                return null;

            // Otherwise return next node from queue of nodes
            return qn.peek();
        }

        // Standard BFS steps: enqueue children of this node
        if (node.left != null)
        {
            qn.add(node.left);
            ql.add(level + 1);
        }
        if (node.right != null)
        {
            qn.add(node.right);
            ql.add(level + 1);
        }
    }

    // We reach here if given key x doesn't exist in tree
    return null;
}

// A utility function to test above functions
void test(Node node, int k)
{
    Node nr = nextRight(root, k);
    if (nr != null)
        System.out.println("Next Right of " + k + " is " + nr.data);
    else
        System.out.println("No next right node found for " + k);
}
```

```
    // Driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(10);
        tree.root.left = new Node(2);
        tree.root.right = new Node(6);
        tree.root.right.right = new Node(5);
        tree.root.left.left = new Node(8);
        tree.root.left.right = new Node(4);

        tree.test(tree.root, 10);
        tree.test(tree.root, 2);
        tree.test(tree.root, 6);
        tree.test(tree.root, 5);
        tree.test(tree.root, 8);
        tree.test(tree.root, 4);

    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Python program to find next right node of given key

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Method to find next right of a given key k, it returns
# None if k is not present in tree or k is the rightmost
# node of its level
def nextRight(root, k):

    # Base Case
    if root is None:
        return 0

    # Create an empty queue for level order traversal
    qn =  [] # A queue to store node addresses
    q1 = [] # Another queue to store node levels
```

```
    level = 0

    # Enqueue root and its level
    qn.append(root)
    q1.append(level)

    # Standard BFS loop
    while(len(qn) > 0):

        # Dequeu an node from qn and its level from q1
        node = qn.pop(0)
        level = q1.pop(0)

        # If the dequeued node has the given key k
        if node.key == k :

            # If there are no more items in queue or given
            # node is the rightmost node of its level,
            # then return None
            if (len(q1) == 0 or q1[0] != level):
                return None

            # Otherwise return next node from queue of nodes
            return qn[0]

        # Standard BFS steps: enqueue children of this node
        if node.left is not None:
            qn.append(node.left)
            q1.append(level+1)

        if node.right is not None:
            qn.append(node.right)
            q1.append(level+1)

    # We reach here if given key x doesn't exist in tree
    return None

def test(root, k):
    nr = nextRight(root, k)
    if nr is not None:
        print "Next Right of " + str(k) + " is " + str(nr.key)
    else:
        print "No next right node found for " + str(k)

# Driver program to test above function
root = Node(10)
root.left = Node(2)
```

```
root.right = Node(6)
root.right.right = Node(5)
root.left.left = Node(8)
root.left.right = Node(4)

test(root, 10)
test(root, 2)
test(root, 6)
test(root, 5)
test(root, 8)
test(root, 4)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
No next right node found for 10
Next Right of 2 is 6
No next right node found for 6
No next right node found for 5
Next Right of 8 is 4
Next Right of 4 is 5
```

Time Complexity: The above code is a simple BFS traversal code which visits every enqueue and dequeues a node at most once. Therefore, the time complexity is O(n) where n is the number of nodes in the given binary tree.

Exercise: Write a function to find left node of a given node. If there is no node on the left side, then return NULL.

Source

https://www.geeksforgeeks.org/find-next-right-node-of-a-given-key/

Chapter 22

Find the first circular tour that visits all petrol pumps

Find the first circular tour that visits all petrol pumps - GeeksforGeeks

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be "start = 1" (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

C/C++

```
 // C program to find circular tour for a truck
#include <stdio.h>
```

```
// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
  int petrol;
  int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end =  1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
      And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If curremt amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1)%n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
               return -1;
        }

        // Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1)%n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
```

```
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

    int n = sizeof(arr)/sizeof(arr[0]);
    int start = printTour(arr, n);

    (start == -1)? printf("No solution"): printf("Start = %d", start);

    return 0;
}
```

Java

```
 //Java program to find circular tour for a truck

public class Petrol
{
    // A petrol pump has petrol and distance to next petrol pump
    static class petrolPump
    {
        int petrol;
        int distance;

        // constructor
        public petrolPump(int petrol, int distance)
        {
            this.petrol = petrol;
            this.distance = distance;
        }
    }

    // The function returns starting point if there is a possible solution,
    // otherwise returns -1
    static int printTour(petrolPump arr[], int n)
    {
        int start = 0;
        int end = 1;
        int curr_petrol = arr[start].petrol - arr[start].distance;

        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while(end != start || curr_petrol < 0)
        {

            // If current amount of petrol in truck becomes less than 0, then
            // remove the starting petrol pump from tour
            while(curr_petrol < 0 && start != end)
            {
```

```
                // Remove starting petrol pump. Change start
                curr_petrol -= arr[start].petrol - arr[start].distance;
                start = (start + 1) % n;

                // If 0 is being considered as start again, then there is no
                // possible solution
                if(start == 0)
                    return -1;
            }
            // Add a petrol pump to current tour
            curr_petrol += arr[end].petrol - arr[end].distance;

            end = (end + 1)%n;
        }

        // Return starting point
        return start;
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {

        petrolPump[] arr = {new petrolPump(6, 4),
                            new petrolPump(3, 6),
                            new petrolPump(7, 3)};

        int start = printTour(arr, arr.length);

        System.out.println(start == -1 ? "No Solution" : "Start = " + start);

    }

}
//This code is contributed by Sumit Ghosh
```

Python

```
 # Python program to find circular tour for a track

# A petrol pump has petrol and distance to next petrol pimp
class PetrolPump:

    # Constructor to create a new node
    def __init__(self,petrol, distance):
        self.petrol = petrol
        self.distance = distance
```

```
# The funtion return starting point if there is a possible
# solution otherwise returns -1
def printTour(arr):

    n = len(arr)
    # Consider first petrol pump as starting point
    start = 0
    end = 1

    curr_petrol = arr[start].petrol - arr[start].distance

    # Run a loop whie all petrol pumps are not visited
    # And we have reached first petrol pump again with 0
    # or more petrol
    while(end != start or curr_petrol < 0 ):

        # If current amount of petrol pumps are not visited
        # And we have reached first petrol pump again with
        # 0 or more petrol
        while(curr_petrol < 0 and start != end):

            # Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance
            start = (start +1)%n

            # If 0 is being considered as start again, then
            # there is no possible solution
            if start == 0:
                return -1

        # Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance

        end = (end +1) % n

    return start

# Driver program to test above function
arr = [PetrolPump(6,4), PetrolPump(3,6), PetrolPump(7,3)]
start = printTour(arr)

print "No solution" if start == -1 else "start =", start

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is O(n).

Auxiliary Space: O(1)

Source

https://www.geeksforgeeks.org/find-a-tour-that-visits-all-stations/

Chapter 23

Find the first non-repeating character from a stream of characters

Find the first non-repeating character from a stream of characters - GeeksforGeeks

Given a stream of characters, find the first non-repeating character from stream. You need to tell the first non-repeating character in O(1) time at any moment.

If we follow the first approach discussed here, then we need to store the stream so that we can traverse it one more time to find the first non-repeating character at any moment. If we use extended approach discussed in the same post, we need to go through the count array every time first non-repeating element is queried. We can find the first non-repeating character from stream at any moment without traversing any array.

The idea is to use a DLL (**D**oubly **L**inked **L**ist) to efficiently get the first non-repeating character from a stream. The DLL contains all non-repeating characters in order, i.e., the head of DLL contains first non-repeating character, the second node contains the second non-repeating and so on.
We also maintain two arrays: one array is to maintain characters that are already visited two or more times, we call it repeated[], the other array is array of pointers to linked list nodes, we call it inDLL[]. The size of both arrays is equal to alphabet size which is typically 256.

1. Create an empty DLL. Also create two arrays inDLL[] and repeated[] of size 256. inDLL is an array of pointers to DLL nodes. repeated[] is a boolean array, repeated[x] is true if x is repeated two or more times, otherwise false. inDLL[x] contains pointer to a DLL node if character x is present in DLL, otherwise NULL.
2. Initialize all entries of inDLL[] as NULL and repeated[] as false.
3. To get the first non-repeating character, return character at head of DLL.
4. Following are steps to process a new character 'x' in stream.

- If repeated[x] is true, ignore this character (x is already repeated two or more times in the stream)
- If repeated[x] is false and inDLL[x] is NULL (x is seen first time). Append x to DLL and store address of new DLL node in inDLL[x].
- If repeated[x] is false and inDLL[x] is not NULL (x is seen second time). Get DLL node of x using inDLL[x] and remove the node. Also, mark inDLL[x] as NULL and repeated[x] as true.

Note that appending a new node to DLL is O(1) operation if we maintain tail pointer. Removing a node from DLL is also O(1). So both operations, addition of new character and finding first non-repeating character take O(1) time.

C/C++

```
 // A C++ program to find first non-repeating character
// from a stream of characters
#include <iostream>
#define MAX_CHAR 256
using namespace std;

// A linked list node
struct node
{
    char a;
    struct node *next, *prev;
};

// A utility function to append a character x at the end
// of DLL. Note that the function may change head and tail
// pointers, that is why pointers to these pointers are passed.
void appendNode(struct node **head_ref, struct node **tail_ref,
                char x)
{
    struct node *temp = new node;
    temp->a = x;
    temp->prev = temp->next = NULL;

    if (*head_ref == NULL)
    {
        *head_ref = *tail_ref = temp;
        return;
    }
    (*tail_ref)->next = temp;
    temp->prev = *tail_ref;
    *tail_ref = temp;
}

// A utility function to remove a node 'temp' fromt DLL.
```

```
// Note that the function may change head and tail pointers,
// that is why pointers to these pointers are passed.
void removeNode(struct node **head_ref, struct node **tail_ref,
                struct node *temp)
{
    if (*head_ref == NULL)
        return;

    if (*head_ref == temp)
        *head_ref = (*head_ref)->next;
    if (*tail_ref == temp)
        *tail_ref = (*tail_ref)->prev;
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    delete(temp);
}

void findFirstNonRepeating()
{
    // inDLL[x] contains pointer to a DLL node if x is present
    // in DLL. If x is not present, then inDLL[x] is NULL
    struct node *inDLL[MAX_CHAR];

    // repeated[x] is true if x is repeated two or more times.
    // If x is not seen so far or x is seen only once. then
    // repeated[x] is false
    bool repeated[MAX_CHAR];

    // Initialize the above two arrays
    struct node *head = NULL, *tail = NULL;
    for (int i = 0; i < MAX_CHAR; i++)
    {
        inDLL[i] = NULL;
        repeated[i] = false;
    }

    // Let us consider following stream and see the process
    char stream[] = "geeksforgeeksandgeeksquizfor";
    for (int i = 0; stream[i]; i++)
    {
        char x = stream[i];
        cout << "Reading " << x << " from stream n";

        // We process this character only if it has not occurred
        // or occurred only once. repeated[x] is true if x is
```

```
        // repeated twice or more.s
        if (!repeated[x])
        {
            // If the character is not in DLL, then add this at
            // the end of DLL.
            if (inDLL[x] == NULL)
            {
                appendNode(&head, &tail, stream[i]);
                inDLL[x] = tail;
            }
            else // Otherwise remove this character from DLL
            {
                removeNode(&head, &tail, inDLL[x]);
                inDLL[x] = NULL;
                repeated[x] = true; // Also mark it as repeated
            }
        }

        // Print the current first non-repeating character from
        // stream
        if (head != NULL)
            cout << "First non-repeating character so far is "
                 << head->a << endl;
    }
}

/* Driver program to test above function */
int main()
{
    findFirstNonRepeating();
    return 0;
}
```

Java

```
 //A Java program to find first non-repeating character
//from a stream of characters

import java.util.ArrayList;
import java.util.List;

public class NonReapeatingC
{
    final static int MAX_CHAR = 256;

    static void findFirstNonRepeating()
    {
        // inDLL[x] contains pointer to a DLL node if x is present
```

```
    // in DLL. If x is not present, then inDLL[x] is NULL
    List<Character> inDLL =new ArrayList<Character>();

    // repeated[x] is true if x is repeated two or more times.
    // If x is not seen so far or x is seen only once. then
    // repeated[x] is false
    boolean[] repeated =new boolean[MAX_CHAR];

    // Let us consider following stream and see the process
    String stream = "geeksforgeeksandgeeksquizfor";
    for (int i=0;i < stream.length();i++)
    {
        char x = stream.charAt(i);
        System.out.println("Reading "+ x +" from stream n");

        // We process this character only if it has not occurred
        // or occurred only once. repeated[x] is true if x is
        // repeated twice or more.s
        if(!repeated[x])
        {
            // If the character is not in DLL, then add this at
            // the end of DLL.
            if(!(inDLL.contains(x)))
            {
                inDLL.add(x);
            }
            else    // Otherwise remove this character from DLL
            {
                inDLL.remove((Character)x);
                repeated[x] = true; // Also mark it as repeated
            }
        }

        // Print the current first non-repeating character from
        // stream
        if(inDLL.size() != 0)
        {
            System.out.print("First non-repeating character so far is ");
            System.out.println(inDLL.get(0));
        }
    }
}

/* Driver program to test above function */
public static void main(String[] args)
{
    findFirstNonRepeating();
}
```

```
}
//This code is contributed by Sumit Ghosh
```

Python

```
 # A Python program to find first non-repeating character from
# a stream of characters
MAX_CHAR = 256

def findFirstNonRepeating():

    # inDLL[x] contains pointer to a DLL node if x is present
    # in DLL. If x is not present, then inDLL[x] is NULL
    inDLL = [] * MAX_CHAR

    # repeated[x] is true if x is repeated two or more times.
    #  If x is not seen so far or x is seen only once. then
    # repeated[x] is false
    repeated = [False] * MAX_CHAR

    # Let us consider following stream and see the process
    stream = "geeksforgeeksandgeeksquizfor"
    for i in xrange(len(stream)):
        x = stream[i]
        print "Reading " + x + " from stream"

        # We process this character only if it has not occurred
        # or occurred only once. repeated[x] is true if x is
        # repeated twice or more.s
        if not repeated[ord(x)]:

            # If the character is not in DLL, then add this
            # at the end of DLL
            if not x in inDLL:
                inDLL.append(x)
            else:
                inDLL.remove(x)

        if len(inDLL) != 0:
            print "First non-repeating character so far is ",
            print str(inDLL[0])

# Driver program
findFirstNonRepeating()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Reading g from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading k from stream
First non-repeating character so far is g
Reading s from stream
First non-repeating character so far is g
Reading f from stream
First non-repeating character so far is g
Reading o from stream
First non-repeating character so far is g
Reading r from stream
First non-repeating character so far is g
Reading g from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading k from stream
First non-repeating character so far is s
Reading s from stream
First non-repeating character so far is f
Reading a from stream
First non-repeating character so far is f
Reading n from stream
First non-repeating character so far is f
Reading d from stream
First non-repeating character so far is f
Reading g from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading k from stream
First non-repeating character so far is f
Reading s from stream
First non-repeating character so far is f
Reading q from stream
First non-repeating character so far is f
Reading u from stream
First non-repeating character so far is f
Reading i from stream
First non-repeating character so far is f
```

```
Reading z from stream
First non-repeating character so far is f
Reading f from stream
First non-repeating character so far is o
Reading o from stream
First non-repeating character so far is r
Reading r from stream
First non-repeating character so far is a
```

This article is contributed by **Amit Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/find-first-non-repeating-character-stream-characters/

Chapter 24

Find the largest multiple of 3 | Set 1 (Using Queue)

Find the largest multiple of 3 | Set 1 (Using Queue) - GeeksforGeeks

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, 9}, the output should be "9 8 1", and if the input array is {8, 1, 7, 6, 0}, output should be "8 7 6 0".

Method 1 (Brute Force)
The simple & straight forward approach is to generate all the combinations of the elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: O(n x 2^n). There will be 2^n combinations of array elements. To compare each combination with the largest number so far may take O(n) time.
Auxiliary Space: O(n) // to avoid integer overflow, the largest number is assumed to be stored in the form of array.

Method 2 (Tricky)
This problem can be solved efficiently with the help of O(n) extra space. This method is based on the following facts about numbers which are multiple of 3.

1) A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is 8 + 7+ 6+ 0 = 21, which is a multiple of 3.

2) If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, ….. are also multiples of 3.

3) We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same remainder 1.

What is the idea behind above facts?
The value of 10%3 and 100%3 is 1. The same is true for all the higher powers of 10, because 3 divides 9, 99, 999, … etc.
Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be 'a', 'b' and 'c' respectively. n can be written as

```
n = 100.a + 10.b + c
```

Since (10^x)%3 is 1 for any x, the above expression gives the same remainder as following expression

```
 1.a + 1.b + c
```

So the remainder obtained by sum of digits and 'n' is same.

Following is a solution based on the above observation.

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.

Based on the above, below is C implementation:

The below code works only if the input arrays has numbers from 0 to 9. It can be easily extended for any positive integer array. We just have to modify the part where we sort the array in decreasing order, at the end of code.

```
 /* A program to find the largest multiple of 3 from an array of elements */
#include <stdio.h>
#include <stdlib.h>
```

```
// A queue node
typedef struct Queue
{
    int front;
    int rear;
    int capacity;
    int* array;
} Queue;

// A utility function to create a queue with given capacity
Queue* createQueue( int capacity )
{
    Queue* queue = (Queue *) malloc (sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (int *) malloc (queue->capacity * sizeof(int));
    return queue;
}

// A utility function to check if queue is empty
int isEmpty (Queue* queue)
{
    return queue->front == -1;
}

// A function to add an item to queue
void Enqueue (Queue* queue, int item)
{
    queue->array[ ++queue->rear ] = item;
    if ( isEmpty(queue) )
        ++queue->front;
}

// A function to remove an item from queue
int Dequeue (Queue* queue)
{
    int item = queue->array[ queue->front ];
    if( queue->front == queue->rear )
        queue->front = queue->rear = -1;
    else
        queue->front++;

    return item;
}

// A utility function to print array contents
void printArr (int* arr, int size)
```

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

/* Following two functions are needed for library function qsort().
   Refer following link for help of qsort()
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compareAsc( const void* a, const void* b )
{
    return *(int*)a > *(int*)b;
}
int compareDesc( const void* a, const void* b )
{
    return *(int*)a < *(int*)b;
}

// This function puts all elements of 3 queues in the auxiliary array
void populateAux (int* aux, Queue* queue0, Queue* queue1,
                            Queue* queue2, int* top )
{
    // Put all items of first queue in aux[]
    while ( !isEmpty(queue0) )
        aux[ (*top)++ ] = Dequeue( queue0 );

    // Put all items of second queue in aux[]
    while ( !isEmpty(queue1) )
        aux[ (*top)++ ] = Dequeue( queue1 );

    // Put all items of third queue in aux[]
    while ( !isEmpty(queue2) )
        aux[ (*top)++ ] = Dequeue( queue2 );
}

// The main function that finds the largest possible multiple of
// 3 that can be formed by arr[] elements
int findMaxMultupleOf3( int* arr, int size )
{
    // Step 1: sort the array in non-decreasing order
    qsort( arr, size, sizeof( int ), compareAsc );

    // Create 3 queues to store numbers with remainder 0, 1
    // and 2 respectively
    Queue* queue0 = createQueue( size );
    Queue* queue1 = createQueue( size );
    Queue* queue2 = createQueue( size );
```

```
// Step 2 and 3 get the sum of numbers and place them in
// corresponding queues
int i, sum;
for ( i = 0, sum = 0; i < size; ++i )
{
    sum += arr[i];
    if ( (arr[i] % 3) == 0 )
        Enqueue( queue0, arr[i] );
    else if ( (arr[i] % 3) == 1 )
        Enqueue( queue1, arr[i] );
    else
        Enqueue( queue2, arr[i] );
}

// Step 4.2: The sum produces remainder 1
if ( (sum % 3) == 1 )
{
    // either remove one item from queue1
    if ( !isEmpty( queue1 ) )
        Dequeue( queue1 );

    // or remove two items from queue2
    else
    {
        if ( !isEmpty( queue2 ) )
            Dequeue( queue2 );
        else
            return 0;

        if ( !isEmpty( queue2 ) )
            Dequeue( queue2 );
        else
            return 0;
    }
}

// Step 4.3: The sum produces remainder 2
else if ((sum % 3) == 2)
{
    // either remove one item from queue2
    if ( !isEmpty( queue2 ) )
        Dequeue( queue2 );

    // or remove two items from queue1
    else
    {
        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );
```

```
            else
                return 0;

            if ( !isEmpty( queue1 ) )
                Dequeue( queue1 );
            else
                return 0;
        }
    }

    int aux[size], top = 0;

    // Empty all the queues into an auxiliary array.
    populateAux (aux, queue0, queue1, queue2, &top);

    // sort the array in non-increasing order
    qsort (aux, top, sizeof( int ), compareDesc);

    // print the result
    printArr (aux, top);

    return top;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 7, 6, 0};
    int size = sizeof(arr)/sizeof(arr[0]);

    if (findMaxMultupleOf3( arr, size ) == 0)
        printf( "Not Possible" );

    return 0;
}
```

The above method can be optimized in following ways.
1) We can use Heap Sort or Merge Sort to make the time complexity O(nLogn).

2) We can avoid extra space for queues. We know at most two items will be removed from the input array. So we can keep track of two items in two variables.

3) At the end, instead of sorting the array again in descending order, we can print the ascending sorted array in reverse order. While printing in reverse order, we can skip the two elements to be removed.

Time Complexity: O(nLogn), assuming a O(nLogn) algorithm is used for sorting.

Find the largest multiple of 3 | Set 2 (In O(n) time and O(1) space)

Source

https://www.geeksforgeeks.org/find-the-largest-number-multiple-of-3/

Chapter 25

First negative integer in every window of size k

First negative integer in every window of size k - GeeksforGeeks

Given an array and a positive integer k, find the first negative integer for each and every window(contiguous subarray) of size k. If a window does not contain a negative interger, then print 0 for that window.

Examples:

```
Input : arr[] = {-8, 2, 3, -6, 10}, k = 2
Output : -8 0 -6 -6
First negative integer for each window of size k
{-8, 2} = -8
{2, 3} = 0 (does not contain a negative integer)
{3, -6} = -6
{-6, 10} = -6

Input : arr[] = {12, -1, -7, 8, -15, 30, 16, 28} , k = 3
Output : -1 -1 -7 -15 -15 0
```

Naive Approach: Run two loops. In the outer loop, take all subarrays(windows) of size k. In the inner loop, get the first negative integer of the current subarray(window).

C++

```
 // C++ implementation to find the first negative
// integer in every window of size k
#include <bits/stdc++.h>
using namespace std;
```

```
// function to find the first negative
// integer in every window of size k
void printFirstNegativeInteger(int arr[], int n, int k)
{
    // flag to check whether window contains
    // a negative integer or not
    bool flag;

    // Loop for each subarray(window) of size k
    for (int i = 0; i<(n-k+1); i++)
    {
        flag = false;

        // traverse through the current window
        for (int j = 0; j<k; j++)
        {
            // if a negative integer is found, then
            // it is the first negative integer for
            // current window. Print it, set the flag
            // and break
            if (arr[i+j] < 0)
            {
                cout << arr[i+j] << " ";
                flag = true;
                break;
            }
        }

        // if the current window does not
        // contain a negative integer
        if (!flag)
            cout << "0" << " ";
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, -1, -7, 8, -15, 30, 16, 28};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printFirstNegativeInteger(arr, n, k);
    return 0;
}
```

Python

```
 # Python3 implementation to find the first negative
```

```
# integer in every window of size k

# Function to find the first negative
# integer in every window of size k
def printFirstNegativeInteger(arr, n, k):

    # Loop for each subarray(window) of size k
    for i in range(0, (n - k + 1)):
        flag = False

        # Traverse through the current window
        for j in range(0, k):

            # If a negative integer is found, then
            # it is the first negative integer for
            # current window. Print it, set the flag
            # and break
            if (arr[i + j] < 0):

                print(arr[i + j], end = " ")
                flag = True
                break

        # If the current window does not
        # contain a negative integer
        if (not(flag)):
            print("0", end = " ")

# Driver Code
arr = [12, -1, -7, 8, -15, 30, 16, 28]
n = len(arr)
k = 3
printFirstNegativeInteger(arr, n, k)

# This code is contributed by 'Smitha dinesh semwal'
```

Output :

```
-1 -1 -7 -15 -15 0
```

Time Complexity : The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(nk) when k is comparitively much smaller than n, otherwise when k tends to reach n, complexity becomes O(k).

Efficient Approach: It is a variation of the problem of Sliding Window Maximum.
We create a Dequeue, Di of capacity k, that stores only useful elements of current window

of k elements. An element is useful if it is in the current window and it is a negative integer. We process all array elements one by one and maintain Di to contain useful elements of current window and these useful elements are all negative integers. For a particular window, if Di is not empty then the element at front of the Di is the first negative integer for that window, else that window does not contain a negative integer.

```
 // C++ implementation to find the first negative
// integer in every window of size k
#include <bits/stdc++.h>

using namespace std;

// function to find the first negative
// integer in every window of size k
void printFirstNegativeInteger(int arr[], int n, int k)
{
    // A Double Ended Queue, Di that will store indexes of
    // useful array elements for the current window of size k.
    // The useful elements are all negative integers.
    deque<int>  Di;

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; i++)
        // Add current element at the rear of Di
        // if it is a negative integer
        if (arr[i] < 0)
            Di.push_back(i);

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; i++)
    {
        // if Di is not empty then the element at the
        // front of the queue is the first negative integer
        // of the previous window
        if (!Di.empty())
            cout << arr[Di.front()] << " ";

        // else the window does not have a
        // negative integer
        else
            cout << "0" << " ";

        // Remove the elements which are out of this window
        while ( (!Di.empty()) && Di.front() < (i - k + 1))
            Di.pop_front();  // Remove from front of queue

        // Add current element at the rear of Di
```

```
        // if it is a negative integer
        if (arr[i] < 0)
            Di.push_back(i);
    }

    // Print the first negative
    // integer of last window
    if (!Di.empty())
           cout << arr[Di.front()] << " ";
    else
        cout << "0" << " ";

}

// Driver program to test above functions
int main()
{
    int arr[] = {12, -1, -7, 8, -15, 30, 16, 28};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printFirstNegativeInteger(arr, n, k);
    return 0;
}
```

Output:

```
-1 -1 -7 -15 -15 0
```

Time Complexity: O(n)
Auxiliary Space: O(k)

Source

https://www.geeksforgeeks.org/first-negative-integer-every-window-size-k/

Chapter 26

How to efficiently implement k Queues in a single array?

How to efficiently implement k Queues in a single array? - GeeksforGeeks

We have discussed efficient implementation of k stack in an array. In this post, same for queue is discussed. Following is the detailed problem statement.

Create a data structure kQueues that represents k queues. Implementation of kQueues should use only one array, i.e., k queues should use the same array for storing elements. Following functions must be supported by kQueues.

enqueue(int x, int qn) –> adds x to queue number 'qn' where qn is from 0 to k-1
dequeue(int qn) –> deletes an element from queue number 'qn' where qn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)
A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.

The problem with this method is inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to second second queue. When we enqueue 4th element to first queue, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)
The idea is similar to the stack post, here we need to use three extra arrays. In stack post, we needed two extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:
1) **front**[]: This is of size k and stores indexes of front elements in all queues.

2) **rear[]**: This is of size k and stores indexes of rear elements in all queues.
2) **next[]**: This is of size n and stores indexes of next item for all items in array arr[].

Here arr[] is actual array that stores k stacks.

Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```
 // A C++ program to demonstrate implementation of k queues in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k queues in a single array of size n
class kQueues
{
    int *arr;   // Array of size n to store actual content to be stored in queue
    int *front;   // Array of size k to store indexes of front elements of queue
    int *rear;   // Array of size k to store indexes of rear elements of queue
    int *next;  // Array of size n to store next entry in all queues
                // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k queue in an array of size n
    kQueues(int k, int n);

    // A utility function to check if there is space available
    bool isFull()   {  return (free == -1);  }

    // To enqueue an item in queue number 'qn' where qn is from 0 to k-1
    void enqueue(int item, int qn);

    // To dequeue an from queue number 'qn' where qn is from 0 to k-1
    int dequeue(int qn);

    // To check whether queue number 'qn' is empty or not
    bool isEmpty(int qn)  {  return (front[qn] == -1); }
};

// Constructor to create k queues in an array of size n
kQueues::kQueues(int k1, int n1)
{
```

```
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    front = new int[k];
    rear = new int[k];
    next = new int[n];

    // Initialize all queues as empty
    for (int i = 0; i < k; i++)
        front[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1;  // -1 is used to indicate end of free list
}

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1
void kQueues::enqueue(int item, int qn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nQueue Overflow\n";
        return;
    }

    int i = free;      // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    if (isEmpty(qn))
        front[qn] = i;
    else
        next[rear[qn]] = i;

    next[i] = -1;

    // Update next of rear and then rear for queue number 'qn'
    rear[qn] = i;

    // Put the item in array
    arr[i] = item;
}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1
```

```
int kQueues::dequeue(int qn)
{
    // Underflow checkSAS
    if (isEmpty(qn))
    {
         cout << "\nQueue Underflow\n";
         return INT_MAX;
    }

    // Find index of front item in queue number 'qn'
    int i = front[qn];

    front[qn] = next[i];  // Change top to store next of previous top

    // Attach the previous front to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous front item
    return arr[i];
}

/* Driver program to test kStacks class */
int main()
{
    // Let us create 3 queue in an array of size 10
    int k = 3, n = 10;
    kQueues ks(k, n);

    // Let us put some items in queue number 2
    ks.enqueue(15, 2);
    ks.enqueue(45, 2);

    // Let us put some items in queue number 1
    ks.enqueue(17, 1);
    ks.enqueue(49, 1);
    ks.enqueue(39, 1);

    // Let us put some items in queue number 0
    ks.enqueue(11, 0);
    ks.enqueue(9, 0);
    ks.enqueue(7, 0);

    cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;
    cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;
    cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

    return 0;
```

```
}
```

Output:

```
Dequeued element from queue 2 is 15
Dequeued element from queue 1 is 17
Dequeued element from queue 0 is 11
```

Time complexities of enqueue() and dequeue() is O(1).

The best part of above implementation is, if there is a slot available in queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : siddhartha33, bejuzb

Source

https://www.geeksforgeeks.org/efficiently-implement-k-queues-single-array/

Chapter 27

Implement PriorityQueue through Comparator in Java

Implement PriorityQueue through Comparator in Java - GeeksforGeeks

Prerequisite : Priority Queue, Comparator
Priority Queue is like a regular queue, but each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. For this, it uses a comparison function which imposes a total ordering of the elements.

The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used

Constructors :

1. **public PriorityQueue() :** This creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
2. **public PriorityQueue(Collection c) :** This creates a PriorityQueue containing the elements in the specified collection(c). If the specified collection is an instance of a SortedSet, this priority queue will be ordered according to the same ordering, else this priority queue will be ordered according to the natural ordering of its elements.
3. **public PriorityQueue(int capacity, Comparator comparator) :** This creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

```
Parameters:
capacity - the initial capacity for this priority queue
comparator - the comparator that will be used to order this priority queue.
If null, the natural ordering of the elements will be used.
```

4. **public PriorityQueue(SortedSet ss) :** Creates a PriorityQueue containing the elements in the specified sorted set. This priority queue will be ordered according to the same ordering as the given sorted set.

Sample code provided illustrates students with high priority(based on cgpa) are served before the students having low cgpa.

```
 // Java program to demonstrate working of
// comparator based priority queue constructor
import java.util.*;

public class Example {
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
        // Creating Priority queue constructor having
        // initial capacity=3 and a StudentComparator instance
        // as its parameters
        PriorityQueue<Student> pq = new
             PriorityQueue<Student>(5, new StudentComparator());

                // Invoking a parameterized Student constructor with
                // name and cgpa as the elements of queue
                Student student1 = new Student("Nandini", 3.2);

                // Adding a student object containing fields
                // name and cgpa to priority queue
                pq.add(student1);
                Student student2 = new Student("Anmol", 3.6);
                        pq.add(student2);
                Student student3 = new Student("Palak", 4.0);
                        pq.add(student3);

                // Printing names of students in priority order,poll()
                // method is used to access the head element of queue
                System.out.println("Students served in their priority order");

                while (!pq.isEmpty()) {
                System.out.println(pq.poll().getName());
        }
    }
}

class StudentComparator implements Comparator<Student>{

            // Overriding compare()method of Comparator
                        // for descending order of cgpa
            public int compare(Student s1, Student s2) {
                if (s1.cgpa < s2.cgpa)
```

```
                return 1;
            else if (s1.cgpa > s2.cgpa)
                return -1;
                            return 0;
            }
        }

class Student {
    public String name;
    public double cgpa;

    // A parameterized student constructor
    public Student(String name, double cgpa) {

        this.name = name;
        this.cgpa = cgpa;
    }

    public String getName() {
        return name;
    }
}
```

Output:

```
Students served in their priority order
Palak
Anmol
Nandini
```

Note : This type of Priority queue is preferred in scenarios where customized ordering is required, i.e when one wants a different sorting order, then one can define its own way of comparing instances. Comparator can be implemented if there is a more complex comparing algorithm, e.g. multiple fields and so on.

Source

https://www.geeksforgeeks.org/implement-priorityqueue-comparator-java/

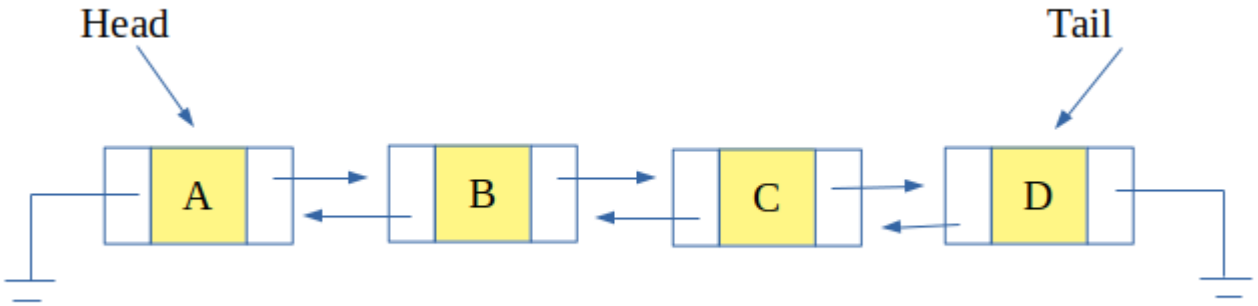
Chapter 28

Implement Stack and Queue using Deque

Implement Stack and Queue using Deque - GeeksforGeeks

Deque also known as **double eneded queue**, as name suggests is a special kind of queue in which insertions and deletions can be done at the last as well as at the beginning.

A link-list representation of deque is such that each node points to the next node as well as the previous node. So that insertion and deletions take constant time at both the beginning and the last.



Now, deque can be used to implement a stack and queue. One simply needs to understand how deque can made to work as a stack or a queue.

The functions of deque to tweak them to work as stack and queue are list below.

DEQUE	STACK	QUEUE
size()	size()	size()
isEmpty()	isEmpty()	isEmpty()
Insert_First()	-	-
Insert_Last()	Push()	Enqueue()
Remove_First()	-	Dequeue()
Remove_Last()	Pop()	-

Examples: Stack

```
Input : Stack : 1 2 3
        Push(4)
Output : Stack : 1 2 3 4

Input : Stack : 1 2 3
        Pop()
Output : Stack : 1 2
```

Examples: Queue

```
Input: Queue : 1 2 3
       Enqueue(4)
Output: Queue : 1 2 3 4

Input: Queue : 1 2 3
       Dequeue()
Output: Queue : 2 3
```

```
 // CPP Program to implement stack and queue using Deque
#include <iostream>
using namespace std;

// structure for a node of deque
struct DQueNode {
    int value;
    DQueNode* next;
    DQueNode* prev;
};
```

```
// Implementation of deque class
class deque {
private:

    // pointers to head and tail of deque
    DQueNode* head;
    DQueNode* tail;

public:
    // constructor
    deque()
    {
        head = tail = NULL;
    }

    // if list is empty
    bool isEmpty()
    {
        if (head == NULL)
            return true;
        return false;
    }

    // count the number of nodes in list
    int size()
    {
        // if list is not empty
        if (!isEmpty()) {
            DQueNode* temp = head;
            int len = 0;
            while (temp != NULL) {
                len++;
                temp = temp->next;
            }
            return len;
        }
        return 0;
    }

    // insert at the first position
    void insert_first(int element)
    {
        // allocating node of DQueNode type
        DQueNode* temp = new DQueNode[sizeof(DQueNode)];
        temp->value = element;

        // if the element is first element
```

```
    if (head == NULL) {
        head = tail = temp;
        temp->next = temp->prev = NULL;
    }
    else {
        head->prev = temp;
        temp->next = head;
        temp->prev = NULL;
        head = temp;
    }
}

// insert at last position of deque
void insert_last(int element)
{
    // allocating node of DQueNode type
    DQueNode* temp = new DQueNode[sizeof(DQueNode)];
    temp->value = element;

    // if element is the first element
    if (head == NULL) {
        head = tail = temp;
        temp->next = temp->prev = NULL;
    }
    else {
        tail->next = temp;
        temp->next = NULL;
        temp->prev = tail;
        tail = temp;
    }
}

// remove element at the first position
void remove_first()
{
    // if list is not empty
    if (!isEmpty()) {
        DQueNode* temp = head;
        head = head->next;
        head->prev = NULL;
        free(temp);
        return;
    }
    cout << "List is Empty" << endl;
}

// remove element at the last position
void remove_last()
```

```
    {
        // if list is not empty
        if (!isEmpty()) {
            DQueNode* temp = tail;
            tail = tail->prev;
            tail->next = NULL;
            free(temp);
            return;
        }
        cout << "List is Empty" << endl;
    }

    // displays the elements in deque
    void display()
    {
        // if list is not empty
        if (!isEmpty()) {
            DQueNode* temp = head;
            while (temp != NULL) {
                cout << temp->value << " ";
                temp = temp->next;
            }
            cout << endl;
            return;
        }
        cout << "List is Empty" << endl;
    }
};

// Class to implement stack using Deque
class Stack : public deque {
public:
    // push to push element at top of stack
    // using insert at last function of deque
    void push(int element)
    {
        insert_last(element);
    }

    // pop to remove element at top of stack
    // using remove at last function of deque
    void pop()
    {
        remove_last();
    }
};

// class to implement queue using deque
```

```
class Queue : public deque {
public:
    // enqueue to insert element at last
    // using insert at last function of deque
    void enqueue(int element)
    {
        insert_last(element);
    }

    // dequeue to remove element from first
    // using remove at first function of deque
    void dequeue()
    {
        remove_first();
    }
};

// Driver Code
int main()
{
    // object of Stack
    Stack stk;

    // push 7 and 8 at top of stack
    stk.push(7);
    stk.push(8);
    cout << "Stack: ";
    stk.display();

    // pop an element
    stk.pop();
    cout << "Stack: ";
    stk.display();

    // object of Queue
    Queue que;

    // insert 12 and 13 in queue
    que.enqueue(12);
    que.enqueue(13);
    cout << "Queue: ";
    que.display();

    // delete an element from queue
    que.dequeue();
    cout << "Queue: ";
    que.display();
```

```
    cout << "Size of Stack is " << stk.size() << endl;
    cout << "Size of Queue is " << que.size() << endl;
    return 0;
}
```

Output:

```
Stack: 7 8
Stack: 7
Queue: 12 13
Queue: 13
Size of Stack is 1
Size of Queue is 1
```

Source

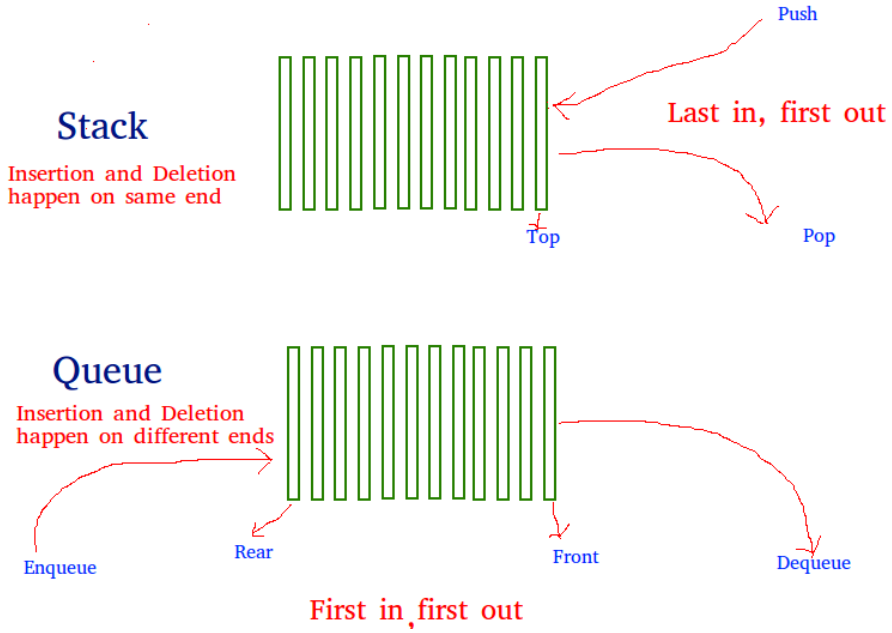
https://www.geeksforgeeks.org/implement-stack-queue-using-deque/

Chapter 29

Implement Stack using Queues

Implement Stack using Queues - GeeksforGeeks

The problem is opposite of thispost. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.



A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)
This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

```
push(s, x) // x is the element to be pushed and s is stack
  1) Enqueue x to q2
  2) One by one dequeue everything from q1 and enqueue to q2.
  3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
  1) Dequeue an item from q1 and return it.
```

```
 /* Program to implement a stack using
two queue */
#include<bits/stdc++.h>
using namespace std;

class Stack
{
    // Two inbuilt queues
    queue<int> q1, q2;

    // To maintain current number of
    // elements
    int curr_size;

    public:
    Stack()
    {
        curr_size = 0;
    }

    void push(int x)
    {
        curr_size++;

        // Push x first in empty q2
        q2.push(x);

        // Push all the remaining
        // elements in q1 to q2.
        while (!q1.empty())
        {
            q2.push(q1.front());
            q1.pop();
        }

        // swap the names of two queues
        queue<int> q = q1;
```

```
        q1 = q2;
        q2 = q;
    }

    void pop(){

        // if no elements are there in q1
        if (q1.empty())
            return ;
        q1.pop();
        curr_size--;
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size()
    {
        return curr_size;
    }
};

// driver code
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;

    cout << "current size: " << s.size()
         << endl;
    return 0;
}
// This code is contributed by Chhavi
```

Output :

```
current size: 3
3
2
1
current size: 1
```

Method 2 (By making pop operation costly)
In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

```
push(s,  x)
  1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)
  1) One by one dequeue everything except the last element from q1 and enqueue to q2.
  2) Dequeue the last item of q1, the dequeued item is result, store it.
  3) Swap the names of q1 and q2
  4) Return the item stored in step 2.
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.
```

```
 /* Program to implement a stack
using two queue */
#include<bits/stdc++.h>
using namespace std;

class Stack
{
    queue<int> q1, q2;
    int curr_size;

    public:
    Stack()
    {
        curr_size = 0;
    }

    void pop()
    {
        if (q1.empty())
            return;

        // Leave one element in q1 and
```

```
    // push others in q2.
    while (q1.size() != 1)
    {
        q2.push(q1.front());
        q1.pop();
    }

    // Pop the only left element
    // from q1
    q1.pop();
    curr_size--;

    // swap the names of two queues
    queue<int> q = q1;
    q1 = q2;
    q2 = q;
}

void push(int x)
{
    q1.push(x);
    curr_size++;
}

int top()
{
    if (q1.empty())
        return -1;

    while( q1.size() != 1 )
    {
       q2.push(q1.front());
       q1.pop();
    }

    // last pushed element
    int temp = q1.front();

    // to empty the auxiliary queue after
    // last operation
    q1.pop();

    // push last element to q2
    q2.push(temp);

    // swap the two queues names
    queue<int> q = q1;
    q1 = q2;
```

```
        q2 = q;
        return temp;
    }

    int size()
    {
        return curr_size;
    }
};

// driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    cout << "current size: " << s.size()
         << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    cout << "current size: " << s.size()
         << endl;
    return 0;
}
// This code is contributed by Chhavi
```

Output :

```
current size: 4
4
3
2
current size: 2
```

References:
Implement Stack using Two Queues

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/implement-stack-using-queue/

Chapter 30

Implement a stack using single queue

Implement a stack using single queue - GeeksforGeeks

We are given queue data structure, the task is to implement stack using only given queue data structure.

We have discussed a solution that uses two queues. In this article, a new solution is discussed that uses only one queue. This solution assumes that we can find size of queue at any point. The idea is to keep newly inserted element always at front, keeping order of previous elements same. Below are complete steps.

```
// x is the element to be pushed and s is stack
push(s, x)
  1) Let size of q be s.
  1) Enqueue x to q
  2) One by one Dequeue s items from queue and enqueue them.

// Removes an item from stack
pop(s)
  1) Dequeue an item from q
```

Below is implementation of the idea.

C++

```
 // C++ program to implement a stack using
// single queue
#include<bits/stdc++.h>
using namespace std;
```

```
// User defined stack that uses a queue
class Stack
{
    queue<int>q;
public:
    void push(int val);
    void pop();
    int top();
    bool empty();
};

// Push operation
void Stack::push(int val)
{
    //  Get previous size of queue
    int s = q.size();

    // Push current element
    q.push(val);

    // Pop (or Dequeue) all previous
    // elements and put them after current
    // element
    for (int i=0; i<s; i++)
    {
        // this will add front element into
        // rear of queue
        q.push(q.front());

        // this will delete front element
        q.pop();
    }
}

// Removes the top element
void Stack::pop()
{
    if (q.empty())
        cout << "No elements\n";
    else
        q.pop();
}

// Returns top of stack
int  Stack::top()
{
    return (q.empty())? -1 : q.front();
```

```
}

// Returns true if Stack is empty else false
bool Stack::empty()
{
    return (q.empty());
}

// Driver code
int main()
{
    Stack s;
    s.push(10);
    s.push(20);
    cout << s.top() << endl;
    s.pop();
    s.push(30);
    s.pop();
    cout << s.top() << endl;
    return 0;
}
```

Java

```
 // Java program to implement stack using a
// single queue

import java.util.LinkedList;
import java.util.Queue;

public class stack
{
    Queue<Integer> q = new LinkedList<Integer>();

    // Push operation
    void push(int val)
    {
        // get previous size of queue
        int size = q.size();

        // Add current element
        q.add(val);

        // Pop (or Dequeue) all previous
        // elements and put them after current
        // element
        for (int i = 0; i < size; i++)
        {
```

```
            // this will add front element into
            // rear of queue
            int x = q.remove();
            q.add(x);
        }
    }

    // Removes the top element
    int pop()
    {
        if (q.isEmpty())
        {
            System.out.println("No elements");
            return -1;
        }
        int x = q.remove();
        return x;
    }

    // Returns top of stack
    int top()
    {
        if (q.isEmpty())
            return -1;
        return q.peek();
    }

    // Returns true if Stack is empty else false
    boolean isEmpty()
    {
        return q.isEmpty();
    }

    // Driver program to test above methods
    public static void main(String[] args)
    {
        stack s = new stack();
        s.push(10);
        s.push(20);
        System.out.println("Top element :" + s.top());
        s.pop();
        s.push(30);
        s.pop();
        System.out.println("Top element :" + s.top());
    }
}

// This code is contributed by Rishabh Mahrsee
```

Output :

```
20
10
```

Source

https://www.geeksforgeeks.org/implement-a-stack-using-single-queue/

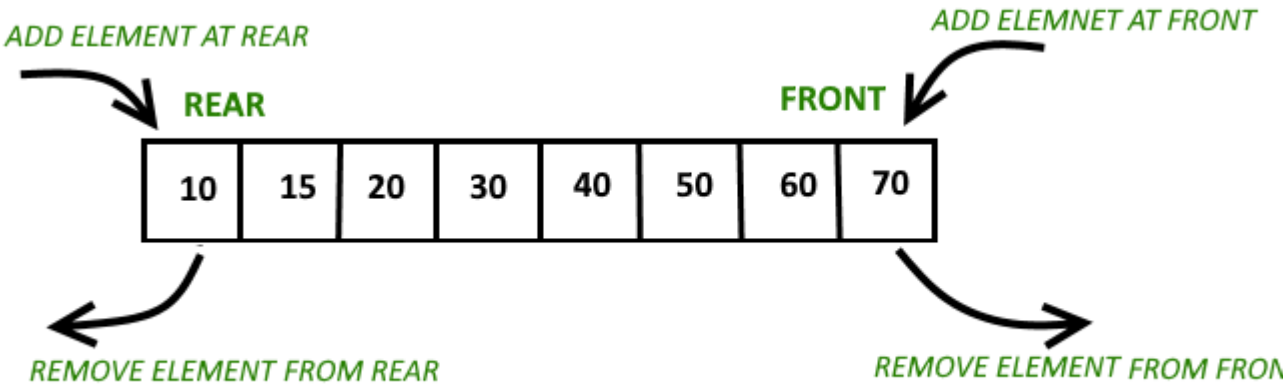
Chapter 31

Implementation of Deque using circular array

Implementation of Deque using circular array - GeeksforGeeks

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.In previous post we had discussed introduction of deque. Now in this post we see how we implement deque Using circular array.
Operations on Deque:
Mainly the following four basic operations are performed on queue:
insetFront(): Adds an item at the front of Deque.
insertRear(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteRear(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported
getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.



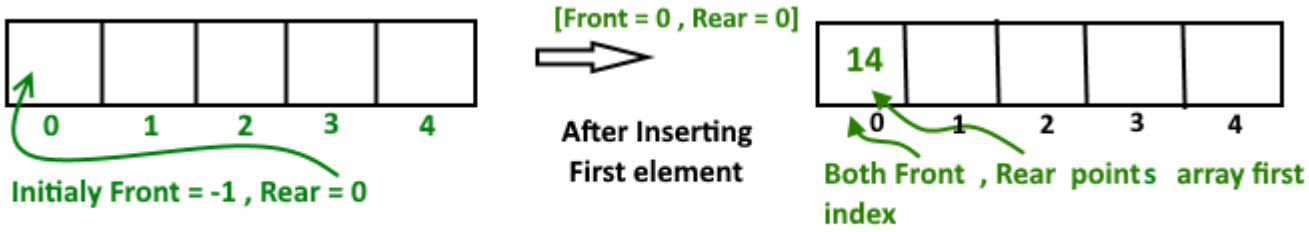
Circular array implementation deque

For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.

Working

1. Create an empty array 'arr' of size 'n'
initialize **front = -1** , **rear = 0**

Inserting First element in deque, at either front or rear will lead to the same result.



After insert **Front** Points = 0 and **Rear** points = 0

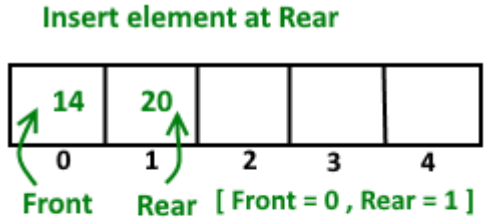
Insert Elements at Rear end

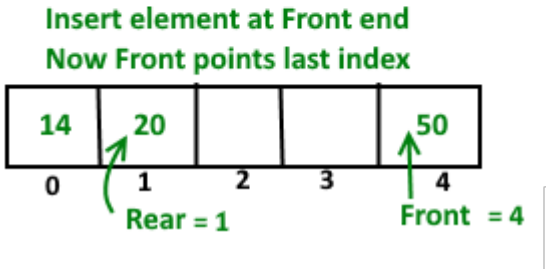
```
a). First we check deque if Full or Not
b). IF Rear == Size-1
       then reinitialize Rear = 0 ;
    Else increment Rear by '1'
    and push current key into Arr[ rear ] = key
Front remain same.
```

Insert Elements at Front end

```
a). First we check deque if Full or Not
```

```
b). IF Front == 0 || initial position, move Front
                     to points last index of array
       front = size - 1
    Else decremented front by '1' and push
         current key into Arr[ Front] = key
Rear remain same.
```





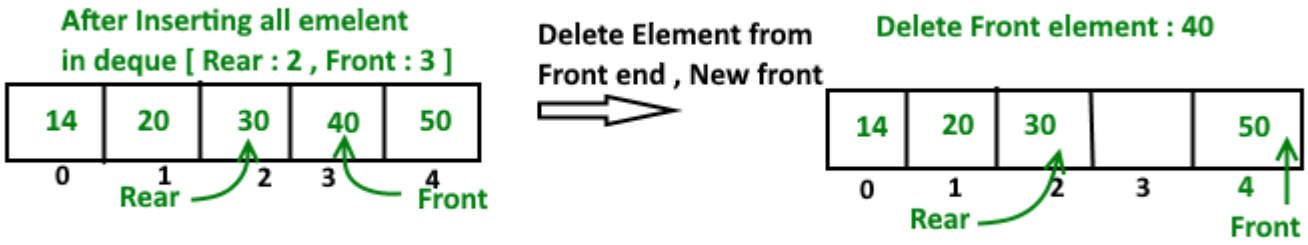
Delete Element From Rear end

```
a). first Check deque is Empty or Not
b).  If deque has only one element
        front = -1 ; rear =-1 ;
    Else IF Rear points to the first index of array
         it's means we have to move rear to points
         last index [ now first inserted element at
         front end become rear end ]
            rear = size-1 ;
    Else || decrease rear by '1'
            rear = rear-1;
```

Delete Element From Front end

```
a). first Check deque is Empty or Not
b).  If deque has only one element
            front = -1 ; rear =-1 ;
    Else IF front points to the last index of the array
```

```
        it's means we have no more elements in array so
         we move front to points first index of array
           front = 0 ;
    Else || increment Front by '1'
           front = front+1;
```



Below is the implementation of above idea.

C++

```
 // C++ implementation of De-queue using circular
// array
#include<iostream>
using namespace std;

// Maximum size of array or Dequeue
#define MAX 100

// A structure to represent a Deque
class Deque
{
    int  arr[MAX];
    int  front;
    int  rear;
    int  size;
public :
    Deque(int size)
    {
        front = -1;
        rear = 0;
        this->size = size;
    }
```

```
    // Operations on Deque:
    void  insertfront(int key);
    void  insertrear(int key);
    void  deletefront();
    void  deleterear();
    bool  isFull();
    bool  isEmpty();
    int  getFront();
    int  getRear();
};

// Checks whether Deque is full or not.
bool Deque::isFull()
{
    return ((front == 0 && rear == size-1)||
            front == rear+1);
}

// Checks whether Deque is empty or not.
bool Deque::isEmpty ()
{
    return (front == -1);
}

// Inserts an element at front
void Deque::insertfront(int key)
{
    // check whether Deque if  full or not
    if (isFull())
    {
        cout << "Overflow\n" << endl;
        return;
    }

    // If queue is initially empty
    if (front == -1)
    {
        front = 0;
        rear = 0;
    }

    // front is at first position of queue
    else if (front == 0)
        front = size - 1 ;

    else // decrement front end by '1'
        front = front-1;
```

```
    // insert current element into Deque
    arr[front] = key ;
}

// function to inset element at rear end
// of Deque.
void Deque ::insertrear(int key)
{
    if (isFull())
    {
        cout << " Overflow\n " << endl;
        return;
    }

    // If queue is initially empty
    if (front == -1)
    {
        front = 0;
        rear = 0;
    }

    // rear is at last position of queue
    else if (rear == size-1)
        rear = 0;

    // increment rear end by '1'
    else
        rear = rear+1;

    // insert current element into Deque
    arr[rear] = key ;
}

// Deletes element at front end of Deque
void Deque ::deletefront()
{
    // check whether Deque is empty or not
    if (isEmpty())
    {
        cout << "Queue Underflow\n" << endl;
        return ;
    }

    // Deque has only one element
    if (front == rear)
    {
        front = -1;
        rear = -1;
```

```
    }
    else
        // back to initial position
        if (front == size -1)
            front = 0;

        else // increment front by '1' to remove current
            // front value from Deque
            front = front+1;
}

// Delete element at rear end of Deque
void Deque::deleterear()
{
    if (isEmpty())
    {
        cout << " Underflow\n" << endl ;
        return ;
    }

    // Deque has only one element
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (rear == 0)
        rear = size-1;
    else
        rear = rear-1;
}

// Returns front element of Deque
int Deque::getFront()
{
    // check whether Deque is empty or not
    if (isEmpty())
    {
        cout << " Underflow\n" << endl;
        return -1 ;
    }
    return arr[front];
}

// function return rear element of Deque
int Deque::getRear()
{
    // check whether Deque is empty or not
```

```
    if(isEmpty() || rear < 0)
    {
        cout << " Underflow\n" << endl;
        return -1 ;
    }
    return arr[rear];
}

// Driver program to test above function
int main()
{
    Deque dq(5);
    cout << "Insert element at rear end  : 5 \n";
    dq.insertrear(5);

    cout << "insert element at rear end : 10 \n";
    dq.insertrear(10);

    cout << "get rear element " << " "
         << dq.getRear() << endl;

    dq.deleterear();
    cout << "After delete rear element new rear"
         << " become " << dq.getRear() << endl;

    cout << "inserting element at front end \n";
    dq.insertfront(15);

    cout << "get front element " << " "
         << dq.getFront() << endl;

    dq.deletefront();

    cout << "After delete front element new "
       << "front become " << dq.getFront() << endl;
    return 0;
}
```

Java

```
 // Java implementation of De-queue using circular
// array

// A structure to represent a Deque
class Deque
{
    static final int MAX = 100;
    int  arr[];
```

```
int  front;
int  rear;
int  size;

public Deque(int size)
{
    arr = new int[MAX];
    front = -1;
    rear = 0;
    this.size = size;
}

/*// Operations on Deque:
void  insertfront(int key);
void  insertrear(int key);
void  deletefront();
void  deleterear();
bool  isFull();
bool  isEmpty();
int  getFront();
int  getRear();*/

// Checks whether Deque is full or not.
boolean isFull()
{
    return ((front == 0 && rear == size-1)||
        front == rear+1);
}

// Checks whether Deque is empty or not.
boolean isEmpty ()
{
    return (front == -1);
}

// Inserts an element at front
void insertfront(int key)
{
    // check whether Deque if  full or not
    if (isFull())
    {
        System.out.println("Overflow");
        return;
    }

    // If queue is initially empty
    if (front == -1)
    {
```

```
        front = 0;
        rear = 0;
    }

    // front is at first position of queue
    else if (front == 0)
        front = size - 1 ;

    else // decrement front end by '1'
        front = front-1;

    // insert current element into Deque
    arr[front] = key ;
}

// function to inset element at rear end
// of Deque.
void insertrear(int key)
{
    if (isFull())
    {
        System.out.println(" Overflow ");
        return;
    }

    // If queue is initially empty
    if (front == -1)
    {
        front = 0;
        rear = 0;
    }

    // rear is at last position of queue
    else if (rear == size-1)
        rear = 0;

    // increment rear end by '1'
    else
        rear = rear+1;

    // insert current element into Deque
    arr[rear] = key ;
}

// Deletes element at front end of Deque
void deletefront()
{
    // check whether Deque is empty or not
```

```
    if (isEmpty())
    {
        System.out.println("Queue Underflow\n");
        return ;
    }

    // Deque has only one element
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else
        // back to initial position
        if (front == size -1)
            front = 0;

        else // increment front by '1' to remove current
            // front value from Deque
            front = front+1;
}

// Delete element at rear end of Deque
void deleterear()
{
    if (isEmpty())
    {
        System.out.println(" Underflow");
        return ;
    }

    // Deque has only one element
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (rear == 0)
        rear = size-1;
    else
        rear = rear-1;
}

// Returns front element of Deque
int getFront()
{
    // check whether Deque is empty or not
    if (isEmpty())
```

```
    {
        System.out.println(" Underflow");
        return -1 ;
    }
    return arr[front];
}

// function return rear element of Deque
int getRear()
{
    // check whether Deque is empty or not
    if(isEmpty() || rear < 0)
    {
        System.out.println(" Underflow\n");
        return -1 ;
    }
    return arr[rear];
}

// Driver program to test above function
public static void main(String[] args)
{

     Deque dq = new Deque(5);

     System.out.println("Insert element at rear end  : 5 ");
     dq.insertrear(5);

     System.out.println("insert element at rear end : 10 ");
     dq.insertrear(10);

     System.out.println("get rear element : "+ dq.getRear());

     dq.deleterear();
     System.out.println("After delete rear element new rear become : " +
                           dq.getRear());

     System.out.println("inserting element at front end");
     dq.insertfront(15);

     System.out.println("get front element: " +dq.getFront());

     dq.deletefront();

     System.out.println("After delete front element new front become : " +
                               +  dq.getFront());

}
```

```
}
```

Output:

```
insert element at rear end  : 5
insert element at rear end : 10
get rear element : 10
After delete rear element new rear become : 5
inserting element at front end
get front element : 15
After delete front element new front become : 5
```

Time Complexity: Time complexity of all operations like insertfront(), insertlast(), deletefront(), deletelast()is O(1).

In mext post we will discuss deque implementation using Doubly linked list.

Improved By : programmer2k17

Source

https://www.geeksforgeeks.org/implementation-deque-using-circular-array/

Chapter 32

Implementation of Deque using doubly linked list

Implementation of Deque using doubly linked list - GeeksforGeeks

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. In previous post Implementation of Deque using circular array has been discussed. Now in this post we see how we implement Deque using Doubly Linked List.

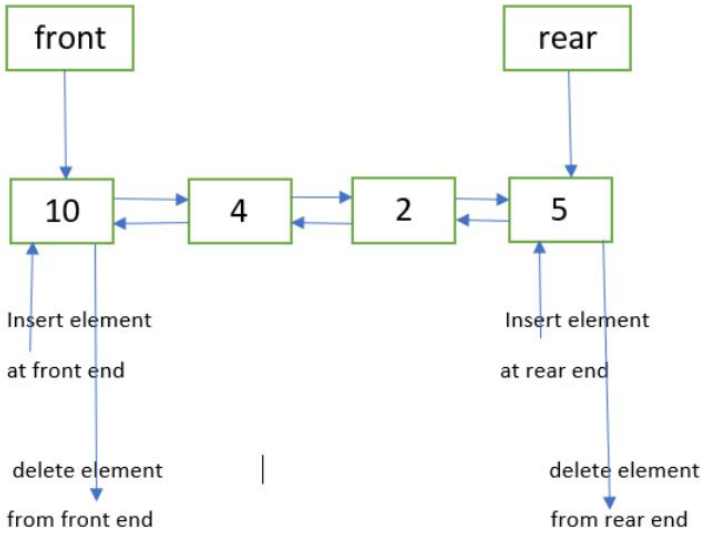
Operations on Deque :

Mainly the following four basic operations are performed on queue :

```
insertFront() : Adds an item at the front of Deque.
insertRear()  : Adds an item at the rear of Deque.
deleteFront() : Deletes an item from front of Deque.
deleteRear()  : Deletes an item from rear of Deque.
```

In addition to above operations, following operations are also supported :

```
getFront() : Gets the front item from queue.
getRear()  : Gets the last item from queue.
isEmpty()  : Checks whether Deque is empty or not.
size()     : Gets number of elements in Deque.
erase()    : Deletes all the elements from Deque.
```



Doubly Linked List Representation of Deque :
For implementing deque, we need to keep track of two pointers, **front** and **rear**. We **enqueue (push)** an item at the rear or the front end of deque and **dequeue(pop)** an item from both rear and front end.

Working :
Declare two pointers **front** and **rear** of type **Node**, where **Node** represents the structure of a node of a doubly linked list. Initialize both of them with value NULL.

Insertion at Front end :

```
1. Allocate space for a newNode of doubly linked list.
2. IF newNode == NULL, then
3.     print "Overflow"
4. ELSE
5.     IF front == NULL, then
6.         rear = front = newNode
7.     ELSE
8.         newNode->next = front
9.       front->prev = newNode
10.        front = newNode
```

Insertion at Rear end :

```
1. Allocate space for a newNode of doubly linked list.
2. IF newNode == NULL, then
3.     print "Overflow"
```

```
4. ELSE
5.     IF rear == NULL, then
6.         front = rear = newNode
7.     ELSE
8.         newNode->prev = rear
9.       rear->next = newNode
10.        rear = newNode
```

Deletion from Front end :

```
1. IF front == NULL
2.     print "Underflow"
3. ELSE
4.     Initalize temp = front
5.     front = front->next
6.     IF front == NULL
7.         rear = NULL
8.     ELSE
9.         front->prev = NULL
10     Deallocate space for temp
```

Deletion from Rear end :

```
1. IF front == NULL
2.     print "Underflow"
3. ELSE
4.     Initalize temp = rear
5.     rear = rear->prev
6.     IF rear == NULL
7.         front = NULL
8.     ELSE
9.         rear->next = NULL
10     Deallocate space for temp
```

```
 // C++ implementation of Deque using
// doubly linked list
#include <bits/stdc++.h>

using namespace std;

// Node of a doubly linked list
struct Node
{
    int data;
    Node *prev, *next;
```

```
    // Function to get a new node
    static Node* getnode(int data)
    {
        Node* newNode = (Node*)malloc(sizeof(Node));
        newNode->data = data;
        newNode->prev = newNode->next = NULL;
        return newNode;
    }
};

// A structure to represent a deque
class Deque
{
    Node* front;
    Node* rear;
    int Size;

public:
    Deque()
    {
        front = rear = NULL;
        Size = 0;
    }

    // Operations on Deque
    void insertFront(int data);
    void insertRear(int data);
    void deleteFront();
    void deleteRear();
    int getFront();
    int getRear();
    int size();
    bool isEmpty();
    void erase();
};

// Function to check whether deque
// is empty or not
bool Deque::isEmpty()
{
    return (front == NULL);
}

// Function to return the number of
// elements in the deque
int Deque::size()
{
    return Size;
```

```
}

// Function to insert an element
// at the front end
void Deque::insertFront(int data)
{
    Node* newNode = Node::getnode(data);
    // If true then new element cannot be added
    // and it is an 'Overflow' condition
    if (newNode == NULL)
        cout << "OverFlow\n";
    else
    {
        // If deque is empty
        if (front == NULL)
            rear = front = newNode;

        // Inserts node at the front end
        else
        {
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
        }

        // Increments count of elements by 1
        Size++;
    }
}

// Function to insert an element
// at the rear end
void Deque::insertRear(int data)
{
    Node* newNode = Node::getnode(data);
    // If true then new element cannot be added
    // and it is an 'Overflow' condition
    if (newNode == NULL)
        cout << "OverFlow\n";
    else
    {
        // If deque is empty
        if (rear == NULL)
            front = rear = newNode;

        // Inserts node at the rear end
        else
        {
```

```
            newNode->prev = rear;
            rear->next = newNode;
            rear = newNode;
        }

        Size++;
    }
}

// Function to delete the element
// from the front end
void Deque::deleteFront()
{
    // If deque is empty then
    // 'Underflow' condition
    if (isEmpty())
        cout << "UnderFlow\n";

    // Deletes the node from the front end and makes
    // the adjustment in the links
    else
    {
        Node* temp = front;
        front = front->next;

        // If only one element was present
        if (front == NULL)
            rear = NULL;
        else
            front->prev = NULL;
        free(temp);

        // Decrements count of elements by 1
        Size--;
    }
}

// Function to delete the element
// from the rear end
void Deque::deleteRear()
{
    // If deque is empty then
    // 'Underflow' condition
    if (isEmpty())
        cout << "UnderFlow\n";

    // Deletes the node from the rear end and makes
    // the adjustment in the links
```

```
    else
    {
        Node* temp = rear;
        rear = rear->prev;

        // If only one element was present
        if (rear == NULL)
            front = NULL;
        else
            rear->next = NULL;
        free(temp);

        // Decrements count of elements by 1
        Size--;
    }
}

// Function to return the element
// at the front end
int Deque::getFront()
{
    // If deque is empty, then returns
    // garbage value
    if (isEmpty())
        return -1;
    return front->data;
}

// Function to return the element
// at the rear end
int Deque::getRear()
{
    // If deque is empty, then returns
    // garbage value
    if (isEmpty())
        return -1;
    return rear->data;
}

// Function to delete all the elements
// from Deque
void Deque::erase()
{
    rear = NULL;
    while (front != NULL)
    {
        Node* temp = front;
        front = front->next;
```

```
        free(temp);
    }
    Size = 0;
}

// Driver program to test above
int main()
{
    Deque dq;
    cout << "Insert element '5' at rear end\n";
    dq.insertRear(5);

    cout << "Insert element '10' at rear end\n";
    dq.insertRear(10);

    cout << "Rear end element: "
        << dq.getRear() << endl;

    dq.deleteRear();
    cout << "After deleting rear element new rear"
        << " is: " << dq.getRear() << endl;

    cout << "Inserting element '15' at front end \n";
    dq.insertFront(15);

    cout << "Front end element: "
        << dq.getFront() << endl;

    cout << "Number of elements in Deque: "
        << dq.size() << endl;

    dq.deleteFront();
    cout << "After deleting front element new "
        << "front is: " << dq.getFront() << endl;

    return 0;
}
```

Output :

```
Insert element '5' at rear end
Insert element '10' at rear end
Rear end element: 10
After deleting rear element new rear is: 5
Inserting element '15' at front end
Front end element: 15
Number of elements in Deque: 2
```

```
After deleting front element new front is: 5
```

Time Complexity : Time complexity of operations like insertFront(), insertRear(), deleteFront(), deleteRear()is O(1). Time Complexity of erase() is O(n).

Source

https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/

Chapter 33

Interleave the first half of the queue with second half

Interleave the first half of the queue with second half - GeeksforGeeks

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 a stack can be used as an auxiliary space.

Examples:

```
Input :  1 2 3 4
Output : 1 3 2 4

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

Following are the steps to solve the problem:
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.

```
 // C++ program to interleave the first half of the queue
// with the second half
#include <bits/stdc++.h>
using namespace std;

// Function to interleave the queue
void interLeaveQueue(queue<int>& q)
```

```
{
    // To check the even number of elements
    if (q.size() % 2 != 0)
        cout << "Input even number of integers." << endl;

    // Initialize an empty stack of int type
    stack<int> s;
    int halfSize = q.size() / 2;

    // Push first half elements into the stack
    // queue:16 17 18 19 20, stack: 15(T) 14 13 12 11
    for (int i = 0; i < halfSize; i++) {
        s.push(q.front());
        q.pop();
    }

    // enqueue back the stack elements
    // queue: 16 17 18 19 20 15 14 13 12 11
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }

    // dequeue the first half elements of queue
    // and enqueue them back
    // queue: 15 14 13 12 11 16 17 18 19 20
    for (int i = 0; i < halfSize; i++) {
        q.push(q.front());
        q.pop();
    }

    // Again push the first half elements into the stack
    // queue: 16 17 18 19 20, stack: 11(T) 12 13 14 15
    for (int i = 0; i < halfSize; i++) {
        s.push(q.front());
        q.pop();
    }

    // interleave the elements of queue and stack
    // queue: 11 16 12 17 13 18 14 19 15 20
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
        q.push(q.front());
        q.pop();
    }
}
```

```
// Driver program to test above function
int main()
{
    queue<int> q;
    q.push(11);
    q.push(12);
    q.push(13);
    q.push(14);
    q.push(15);
    q.push(16);
    q.push(17);
    q.push(18);
    q.push(19);
    q.push(20);
    interLeaveQueue(q);
    int length = q.size();
    for (int i = 0; i < length; i++) {
        cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

Output:

```
11 16 12 17 13 18 14 19 15 20
```

Time complexity: O(n).
Auxiliary Space : O(n).

Source

https://www.geeksforgeeks.org/interleave-first-half-queue-second-half/

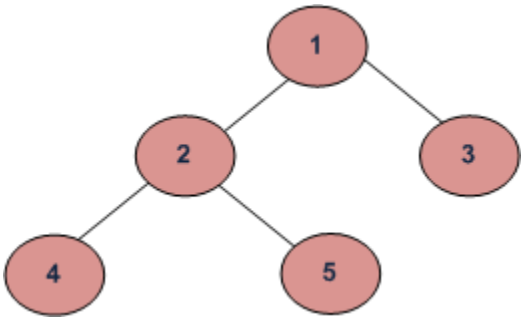
Chapter 34

Iterative Method to find Height of Binary Tree

Iterative Method to find Height of Binary Tree - GeeksforGeeks

There are two conventions to define height of Binary Tree
1) Number of nodes on longest path from root to the deepest node.
2) Number of edges on longest path from root to the deepest node.

In this post, the first convention is followed. For example, height of the below tree is 3.



Example Tree

Recursive method to find height of Binary Tree is discussed here. How to find height without recursion? We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level, stop traversing when count of nodes at next level is 0.
Following is detailed algorithm to find level order traversal using queue.

```
Create a queue.
```

```
Push root into the queue.
height = 0
Loop
    nodeCount = size of queue

        // If number of nodes at this level is 0, return height
    if nodeCount is 0
        return Height;
    else
        increase Height

        // Remove nodes of this level and add nodes of
        // next level
    while (nodeCount > 0)
        pop node from front
        push its children to queue
        decrease nodeCount
       // At this point, queue has nodes of next level
```

Following is the implementation of above algorithm.

C++

```
 /* Program to find height of the tree by Iterative Method */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left;
    int data;
    struct node *right;
};

// Iterative method to find height of Bianry Tree
int treeHeight(node *root)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order tarversal
    queue<node *> q;

    // Enqueue Root and initialize height
    q.push(root);
```

```
    int height = 0;

    while (1)
    {
        // nodeCount (queue size) indicates number of nodes
        // at current lelvel.
        int nodeCount = q.size();
        if (nodeCount == 0)
            return height;

        height++;

        // Dequeue all nodes of current level and Enqueue all
        // nodes of next level
        while (nodeCount > 0)
        {
            node *node = q.front();
            q.pop();
            if (node->left != NULL)
                q.push(node->left);
            if (node->right != NULL)
                q.push(node->right);
            nodeCount--;
        }
    }
}

// Utility function to create a new tree node
node* newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Height of tree is " << treeHeight(root);
```

```
    return 0;
}
```

Java

```
 // An iterative java program to find height of binary tree

import java.util.LinkedList;
import java.util.Queue;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;

    // Iterative method to find height of Bianry Tree
    int treeHeight(Node node)
    {
        // Base Case
        if (node == null)
            return 0;

        // Create an empty queue for level order tarversal
        Queue<Node> q = new LinkedList();

        // Enqueue Root and initialize height
        q.add(node);
        int height = 0;

        while (1 == 1)
        {
            // nodeCount (queue size) indicates number of nodes
            // at current lelvel.
            int nodeCount = q.size();
            if (nodeCount == 0)
                return height;
```

```
            height++;

            // Dequeue all nodes of current level and Enqueue all
            // nodes of next level
            while (nodeCount > 0)
            {
                Node newnode = q.peek();
                q.remove();
                if (newnode.left != null)
                    q.add(newnode.left);
                if (newnode.right != null)
                    q.add(newnode.right);
                nodeCount--;
            }
        }
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        // Let us create a binary tree shown in above diagram
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        System.out.println("Height of tree is " + tree.treeHeight(tree.root));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
 # Program to find height of tree by Iteration Method

# A binary tree node
class Node:

    # Constructor to create new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Iterative method to find height of Binary Tree
```

```
def treeHeight(root):

    # Base Case
    if root is None:
        return 0

    # Create a empty queue for level order traversal
    q = []

    # Enqueue Root and Initialize Height
    q.append(root)
    height = 0

    while(True):

        # nodeCount(queue size) indicates number of nodes
        # at current level
        nodeCount = len(q)
        if nodeCount == 0 :
            return height

        height += 1

        # Dequeue all nodes of current level and Enqueue
        # all nodes of next level
        while(nodeCount > 0):
            node = q[0]
            q.pop(0)
            if node.left is not None:
                q.append(node.left)
            if node.right is not None:
                q.append(node.right)

            nodeCount -= 1


# Driver program to test above function
# Let us create binary tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Height of tree is", treeHeight(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Height of tree is 3
```

Time Complexity: O(n) where n is number of nodes in given binary tree.

This article is contributed by **Rahul Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/iterative-method-to-find-height-of-binary-tree/

Chapter 35

LRU Cache Implementation

LRU Cache Implementation - GeeksforGeeks

How to implement LRU caching scheme? What data structures should be used?
We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide here).

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.
If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

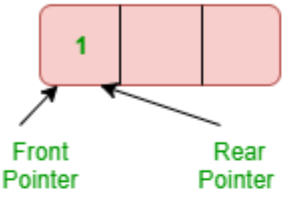
Example – Consider the following reference string :

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

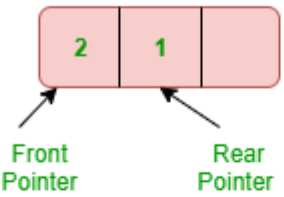
Find the number of page faults using least recently used (LRU) page replacement algorithm with 3 page frames.
Explanation –

Given 3 page frames, so we take size of Queue is 3. Initially, Queue is empty.

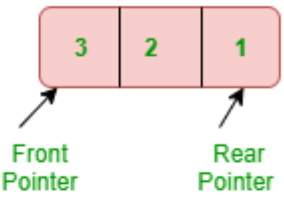
Input : 1



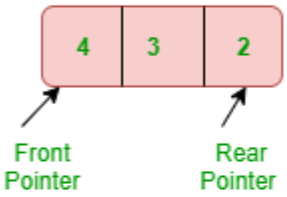
Input : 2 (every new input will be front as defined LRU)



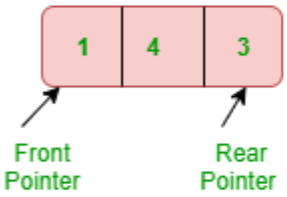
Input : 3



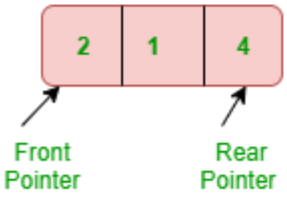
Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



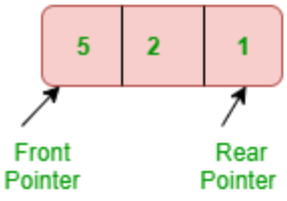
Input : 1 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



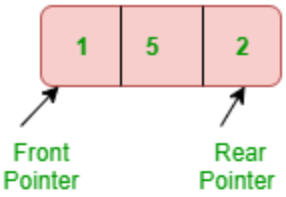
Input : 2 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



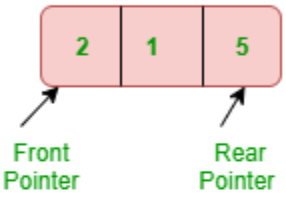
Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



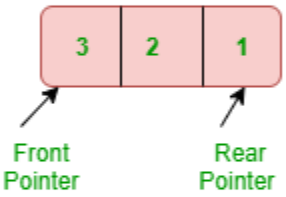
Input : 1 (since present in memory, so bring it to the front of the queue. This is called hit)



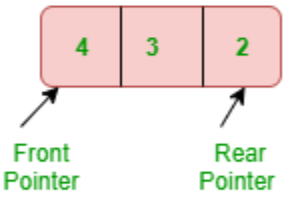
Input : 2 (since present in memory, so bring it to the front of the queue. This is called hit)



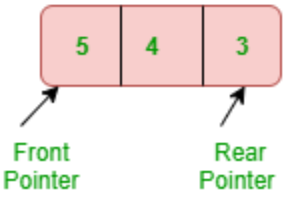
Input : 3 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



So, we have only 2 hits and 10 page faults using LRU page replacement algorithm.

Note: Initially no page is in the memory.

C++ using STL

```
 /* We can use stl container list as a double
   ended queue to store the cache keys, with
   the descending time of reference from front
   to back and a set container to check presence
   of a key. But to fetch the address of the key
   in the list using find(), it takes O(N) time.
    This can be optimized by storing a reference
    (iterator) to each key in a hash map. */
#include <bits/stdc++.h>
using namespace std;

class LRUCache
{
    // store keys of cache
    list<int> dq;

    // store references of key in cache
    unordered_map<int, list<int>::iterator> ma;
    int csize; //maximum capacity of cache

public:
    LRUCache(int);
    void refer(int);
    void display();
};

LRUCache::LRUCache(int n)
{
    csize = n;
}

/* Refers key x with in the LRU cache */
void LRUCache::refer(int x)
{
    // not present in cache
    if (ma.find(x) == ma.end())
    {
        // cache is full
        if (dq.size() == csize)
        {
            //delete least recently used element
            int last = dq.back();
            dq.pop_back();
            ma.erase(last);
        }
```

```
    }

    // present in cache
    else
        dq.erase(ma[x]);

    // update reference
    dq.push_front(x);
    ma[x] = dq.begin();
}

// display contents of cache
void LRUCache::display()
{
    for (auto it = dq.begin(); it != dq.end();
                                        it++)
        cout << (*it) << " ";

    cout << endl;
}

// Driver program to test above functions
int main()
{
    LRUCache ca(4);

    ca.refer(1);
    ca.refer(2);
    ca.refer(3);
    ca.refer(1);
    ca.refer(4);
    ca.refer(5);
    ca.display();

    return 0;
}
// This code is contributed by Satish Srinivas
```

C

```
 // A C program to show implementation of LRU cache
#include <stdio.h>
#include <stdlib.h>

// A Queue Node (Queue is implemented using Doubly Linked List)
typedef struct QNode
{
    struct QNode *prev, *next;
```

```
    unsigned pageNumber;  // the page number stored in this QNode
} QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue
{
    unsigned count;  // Number of filled frames
    unsigned numberOfFrames; // total number of frames
    QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash
{
    int capacity; // how many pages can be there
    QNode* *array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue Node
// will store the given 'pageNumber'
QNode* newQNode( unsigned pageNumber )
{
    // Allocate memory and assign 'pageNumber'
    QNode* temp = (QNode *)malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;

    // Initialize prev and next as NULL
    temp->prev = temp->next = NULL;

    return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfFrames' nodes
Queue* createQueue( int numberOfFrames )
{
    Queue* queue = (Queue *)malloc( sizeof( Queue ) );

    // The queue is empty
    queue->count = 0;
    queue->front = queue->rear = NULL;

    // Number of frames that can be stored in memory
    queue->numberOfFrames = numberOfFrames;

    return queue;
}
```

```
// A utility function to create an empty Hash of given capacity
Hash* createHash( int capacity )
{
    // Allocate memory for hash
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;

    // Create an array of pointers for refering queue nodes
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );

    // Initialize all hash entries as empty
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;

    return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}

// A utility function to check if queue is empty
int isQueueEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void deQueue( Queue* queue )
{
    if( isQueueEmpty( queue ) )
        return;

    // If this is the only node in list, then change front
    if (queue->front == queue->rear)
        queue->front = NULL;

    // Change rear and remove the previous rear
    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );
```

```
    // decrement the number of full frames by 1
    queue->count--;
}

// A function to add a page with given 'pageNumber' to both queue
// and hash
void Enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{
    // If all frames are full, remove the page at the rear
    if ( AreAllFramesFull ( queue ) )
    {
        // remove page from hash
        hash->array[ queue->rear->pageNumber ] = NULL;
        deQueue( queue );
    }

    // Create a new node with given page number,
    // And add the new node to the front of queue
    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    // If queue is empty, change both front and rear pointers
    if ( isQueueEmpty( queue ) )
        queue->rear = queue->front = temp;
    else  // Else change the front
    {
        queue->front->prev = temp;
        queue->front = temp;
    }

    // Add page entry to hash also
    hash->array[ pageNumber ] = temp;

    // increment number of full frames
    queue->count++;
}

// This function is called when a page with given 'pageNumber' is referenced
// from cache (or memory). There are two cases:
// 1. Frame is not there in memory, we bring it in memory and add to the front
//    of queue
// 2. Frame is there in memory, we move the frame to front of queue
void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    // the page is not in cache, bring it
```

```
    if ( reqPage == NULL )
        Enqueue( queue, hash, pageNumber );

    // page is there and not at front, change pointer
    else if (reqPage != queue->front)
    {
        // Unlink rquested page from its current location
        // in queue.
        reqPage->prev->next = reqPage->next;
        if (reqPage->next)
           reqPage->next->prev = reqPage->prev;

        // If the requested page is rear, then change rear
        // as this node will be moved to front
        if (reqPage == queue->rear)
        {
           queue->rear = reqPage->prev;
           queue->rear->next = NULL;
        }

        // Put the requested page before current front
        reqPage->next = queue->front;
        reqPage->prev = NULL;

        // Change prev of current front
        reqPage->next->prev = reqPage;

        // Change front to the requested page
        queue->front = reqPage;
    }
}

// Driver program to test above functions
int main()
{
    // Let cache can hold 4 pages
    Queue* q = createQueue( 4 );

    // Let 10 different pages can be requested (pages to be
    // referenced are numbered from 0 to 9
    Hash* hash = createHash( 10 );

    // Let us refer pages 1, 2, 3, 1, 4, 5
    ReferencePage( q, hash, 1);
    ReferencePage( q, hash, 2);
    ReferencePage( q, hash, 3);
    ReferencePage( q, hash, 1);
    ReferencePage( q, hash, 4);
```

```
    ReferencePage( q, hash, 5);

    // Let us print cache frames after the above referenced pages
    printf ("%d ", q->front->pageNumber);
    printf ("%d ", q->front->next->pageNumber);
    printf ("%d ", q->front->next->next->pageNumber);
    printf ("%d ", q->front->next->next->next->pageNumber);

    return 0;
}
```

Output:

```
5 4 1 3
```

This article is compiled by Aashish Barnwaland reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/lru-cache-implementation/

Chapter 36

Length of the longest valid substring

Length of the longest valid substring - GeeksforGeeks

Given a string consisting of opening and closing parenthesis, find length of the longest valid parenthesis substring.

Examples:

```
Input : (((()
Output : 2
Explanation : ()

Input: )()())
Output : 4
Explanation: ()()

Input:  ()((()))))
Output: 6
Explanation:  ()(())
```

A **Simple Approach** is to find all the substrings of given string. For every string, check if it is a valid string or not. If valid and length is more than maximum length so far, then update maximum length. We can check whether a substring is valid or not in linear time using a stack (See this for details). Time complexity of this solution is $O(n^2$.

An **Efficient Solution** can solve this problem in O(n) time. The idea is to store indexes of previous starting brackets in a stack. The first element of stack is a special element that provides index before beginning of valid substring (base for next valid string).

```
1) Create an empty stack and push -1 to it. The first element
   of stack is used to provide base for next valid string.

2) Initialize result as 0.

3) If the character is '(' i.e. str[i] == '(', push index
   'i' to the stack.

2) Else (if the character is ')')
   a) Pop an item from stack (Most of the time an opening bracket)
   b) If stack is not empty, then find length of current valid
      substring by taking difference between current index and
      top of the stack. If current length is more than result,
      then update the result.
   c) If stack is empty, push current index as base for next
      valid substring.

3) Return result.
```

Below are C++ and Python implementations of above algorithm.

C++

```
 // C++ program to find length of the longest valid
// substring
#include<bits/stdc++.h>
using namespace std;

int findMaxLen(string str)
{
    int n = str.length();

    // Create a stack and push -1 as initial index to it.
    stack<int> stk;
    stk.push(-1);

    // Initialize result
    int result = 0;

    // Traverse all characters of given string
    for (int i=0; i<n; i++)
    {
        // If opening bracket, push index of it
        if (str[i] == '(')
          stk.push(i);
```

```
        else // If closing bracket, i.e.,str[i] = ')'
        {
            // Pop the previous opening bracket's index
            stk.pop();

            // Check if this length formed with base of
            // current valid substring is more than max
            // so far
            if (!stk.empty())
                result = max(result, i - stk.top());

            // If stack is empty. push current index as
            // base for next valid substring (if any)
            else stk.push(i);
        }
    }

    return result;
}

// Driver program
int main()
{
    string str = "((()()";
    cout << findMaxLen(str) << endl;

    str = "()((()))))";
    cout << findMaxLen(str) << endl ;

    return 0;
}
```

Java

```
 // Java program to find length of the longest valid
// substring

import java.util.Stack;

class Test
{
    // method to get length of the longest valid
    static int findMaxLen(String str)
    {
        int n = str.length();

        // Create a stack and push -1 as initial index to it.
        Stack<Integer> stk = new Stack<>();
```

```
        stk.push(-1);

        // Initialize result
        int result = 0;

        // Traverse all characters of given string
        for (int i=0; i<n; i++)
        {
            // If opening bracket, push index of it
            if (str.charAt(i) == '(')
              stk.push(i);

            else // If closing bracket, i.e.,str[i] = ')'
            {
                // Pop the previous opening bracket's index
                stk.pop();

                // Check if this length formed with base of
                // current valid substring is more than max
                // so far
                if (!stk.empty())
                    result = Math.max(result, i - stk.peek());

                // If stack is empty. push current index as
                // base for next valid substring (if any)
                else stk.push(i);
            }
        }

        return result;
    }

    // Driver method
    public static void main(String[] args)
    {
        String str = "((()()";
        System.out.println(findMaxLen(str));

        str = "()((())))";
        System.out.println(findMaxLen(str));

    }
}
```

Python

```
 # Python program to find length of the longest valid
# substring
```

```
def findMaxLen(string):
    n = len(string)

    # Create a stack and push -1 as initial index to it.
    stk = []
    stk.append(-1)

    # Initialize result
    result = 0

    # Traverse all characters of given string
    for i in xrange(n):

        # If opening bracket, push index of it
        if string[i] == '(':
            stk.append(i)

        else:    # If closing bracket, i.e., str[i] = ')'

            # Pop the previous opening bracket's index
            stk.pop()

            # Check if this length formed with base of
            # current valid substring is more than max
            # so far
            if len(stk) != 0:
                result = max(result, i - stk[len(stk)-1])

            # If stack is empty. push current index as
            # base for next valid substring (if any)
            else:
                stk.append(i)

    return result

# Driver program
string = "((()()"
print findMaxLen(string)

string = "()((()))))"
print findMaxLen(string)

# This code is contributed by Bhavya Jain
```

Output:

```
4
```

```
6
```

Explanation with example:

```
Input: str = "(()()"

Initialize result as 0 and stack with one item -1.

For i = 0, str[0] = '(', we push 0 in stack

For i = 1, str[1] = '(', we push 1 in stack

For i = 2, str[2] = ')', currently stack has [-1, 0, 1], we pop
from the stack and the stack now is [-1, 0] and length of current
valid substring becomes 2 (we get this 2 by subtracting stack top
from current index).
Since current length is more than current result, we update result.

For i = 3, str[3] = '(', we push again, stack is [-1, 0, 3].

For i = 4, str[4] = ')', we pop from the stack, stack becomes
[-1, 0] and length of current valid substring becomes 4 (we get
this 4 by subtracting stack top from current index).
Since current length is more than current result, we update result.
```

Thanks to Gaurav Ahirwar and Ekta Goel. for suggesting above approach.

Source

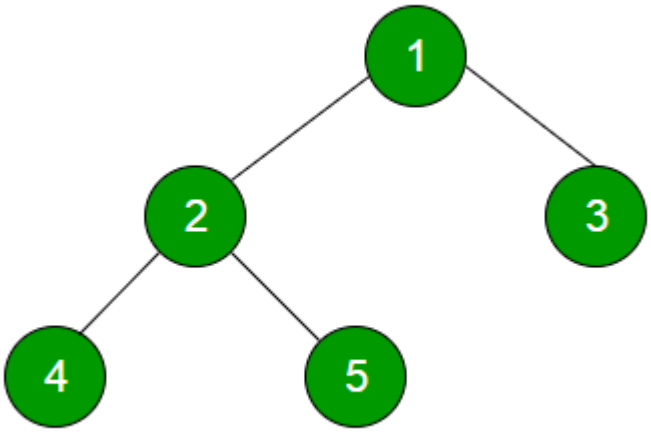
https://www.geeksforgeeks.org/length-of-the-longest-valid-substring/

Chapter 37

Level Order Tree Traversal

Level Order Tree Traversal - GeeksforGeeks

Level order traversal of a tree is breadth first traversal for the tree.



Level order traversal of the above tree is 1 2 3 4 5

METHOD 1 (Use function to print a given level)

Algorithm:
There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
```

```
for d = 1 to height(tree)
   printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Implementation:

C

```
 // Recursive C program for level order traversal of Binary Tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* Function protoypes */
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
```

```
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
```

```
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    return 0;
}
```

Java

```
 // Recursive Java program for level order traversal of Binary Tree

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;
    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    public BinaryTree()
    {
        root = null;
    }

    /* function to print level order traversal of tree*/
    void printLevelOrder()
    {
        int h = height(root);
        int i;
        for (i=1; i<=h; i++)
            printGivenLevel(root, i);
    }

    /* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
```

```
    int height(Node root)
    {
        if (root == null)
           return 0;
        else
        {
            /* compute  height of each subtree */
            int lheight = height(root.left);
            int rheight = height(root.right);

            /* use the larger one */
            if (lheight > rheight)
                return(lheight+1);
            else return(rheight+1);
        }
    }

    /* Print nodes at the given level */
    void printGivenLevel (Node root ,int level)
    {
        if (root == null)
            return;
        if (level == 1)
            System.out.print(root.data + " ");
        else if (level > 1)
        {
            printGivenLevel(root.left, level-1);
            printGivenLevel(root.right, level-1);
        }
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
       BinaryTree tree = new BinaryTree();
       tree.root= new Node(1);
       tree.root.left= new Node(2);
       tree.root.right= new Node(3);
       tree.root.left.left= new Node(4);
       tree.root.left.right= new Node(5);

       System.out.println("Level order traversal of binary tree is ");
       tree.printLevelOrder();
    }
}
```

Python

```
 # Recursive Python program for level order traversal of Binary Tree

# A node structure
class Node:

    # A utility function to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None


# Function to  print level order traversal of tree
def printLevelOrder(root):
    h = height(root)
    for i in range(1, h+1):
        printGivenLevel(root, i)


# Print nodes at a given level
def printGivenLevel(root , level):
    if root is None:
        return
    if level == 1:
        print "%d" %(root.data),
    elif level > 1 :
        printGivenLevel(root.left , level-1)
        printGivenLevel(root.right , level-1)


""" Compute the height of a tree--the number of nodes
    along the longest path from the root node down to
    the farthest leaf node
"""
def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1
```

```
# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level order traversal of binary tree is -"
printLevelOrder(root)

#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -
1 2 3 4 5
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

METHOD 2 (Use Queue)

Algorithm:
For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children (first left then right children) to q
    c) Dequeue a node from q and assign it's value to temp_node
```

Implementation:
Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

C

```
 // Iterative Queue based C program to do level order traversal
// of Binary Tree
#include <stdio.h>
#include <stdlib.h>
#define MAX_Q_SIZE 500
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* frunction prototypes */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
void printLevelOrder(struct node* root)
{
    int rear, front;
    struct node **queue = createQueue(&front, &rear);
    struct node *temp_node = root;

    while (temp_node)
    {
        printf("%d ", temp_node->data);

        /*Enqueue left child */
        if (temp_node->left)
            enQueue(queue, &rear, temp_node->left);

        /*Enqueue right child */
        if (temp_node->right)
            enQueue(queue, &rear, temp_node->right);

        /*Dequeue node and make it temp_node*/
        temp_node = deQueue(queue, &front);
    }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
```

```
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    return 0;
}
```

C++

```
 /* C++ program to print level order traversal using STL */
#include <iostream>
#include <queue>
using namespace std;
```

```
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Bianry Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order tarversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();

        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);

        /*Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
```

```
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}
```

Java

```
 // Iterative Queue based Java program to do level order traversal
// of Binary Tree

/* importing the inbuilt java classes required for the program */
import java.util.Queue;
import java.util.LinkedList;

/* Class to represent Tree node */
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = null;
        right = null;
    }
}

/* Class to print Level Order Traversal */
class BinaryTree {

    Node root;

    /* Given a binary tree. Print its nodes in level order
     using array for implementing queue  */
    void printLevelOrder()
    {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while (!queue.isEmpty())
        {
```

```
            /* poll() removes the present head.
            For more information on poll() visit
            http://www.tutorialspoint.com/java/util/linkedlist_poll.htm */
            Node tempNode = queue.poll();
            System.out.print(tempNode.data + " ");

            /*Enqueue left child */
            if (tempNode.left != null) {
                queue.add(tempNode.left);
            }

            /*Enqueue right child */
            if (tempNode.right != null) {
                queue.add(tempNode.right);
            }
        }
    }

    public static void main(String args[])
    {
        /* creating a binary tree and entering
         the nodes */
        BinaryTree tree_level = new BinaryTree();
        tree_level.root = new Node(1);
        tree_level.root.left = new Node(2);
        tree_level.root.right = new Node(3);
        tree_level.root.left.left = new Node(4);
        tree_level.root.left.right = new Node(5);

        System.out.println("Level order traversal of binary tree is - ");
        tree_level.printLevelOrder();
    }
}
```

Python

```
 # Python program to print level order traversal using Queue

# A node structure
class Node:
    # A utility function to create a new node
    def __init__(self ,key):
        self.data = key
        self.left = None
        self.right = None

# Iterative Method to print the height of binary tree
def printLevelOrder(root):
```

```
    # Base Case
    if root is None:
        return

    # Create an empty queue for level order traversal
    queue = []

    # Enqueue Root and initialize height
    queue.append(root)

    while(len(queue) > 0):
        # Print front of queue and remove it from queue
        print queue[0].data,
        node = queue.pop(0)

        #Enqueue left child
        if node.left is not None:
            queue.append(node.left)

        # Enqueue right child
        if node.right is not None:
            queue.append(node.right)

#Driver Program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level Order Traversal of binary tree is -"
printLevelOrder(root)
#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -
1 2 3 4 5
```

Time Complexity: O(n) where n is number of nodes in the binary tree

References:
http://en.wikipedia.org/wiki/Breadth-first_traversal

Source

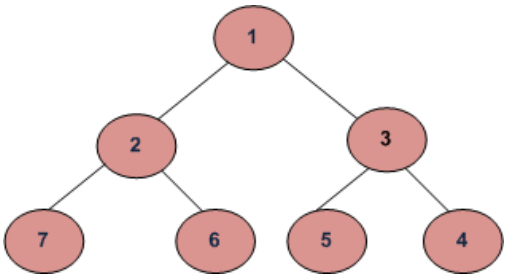
https://www.geeksforgeeks.org/level-order-tree-traversal/

Chapter 38

Level order traversal in spiral form

Level order traversal in spiral form - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



Method 1 (Recursive)
This problem can bee seen as an extension of the level order traversal post.
To print the nodes in spiral order, nodes at different levels should be printed in alternating order. 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.

Function to print level order traversal of tree

```
printSpiral(tree)
  bool ltr = 0;
  for d = 1 to height(tree)
     printGivenLevel(tree, d, ltr);
     ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    if(ltr)
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
    else
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
```

Following is C implementation of above algorithm.

C

```
 // C program for recursive level order traversal in spiral form
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function protoypes */
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(struct node* root)
{
    int h = height(root);
    int i;

    /*ltr -> Left to Right. If this variable is set,
      then the given level is traverseed from left to right. */
    bool ltr = false;
    for(i=1; i<=h; i++)
    {
```

```
        printGivenLevel(root, i, ltr);

        /*Revert ltr to traverse next level in opposite order*/
        ltr = !ltr;
    }
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        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);
        }
    }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}
```

```
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(7);
    root->left->right = newNode(6);
    root->right->left  = newNode(5);
    root->right->right = newNode(4);
    printf("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}
```

Java

```
 // Java program for recursive level order traversal in spiral form

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int d)
    {
        data = d;
        left = right = null;
    }
}
```

```
class BinaryTree
{
    Node root;

    // Function to print the spiral traversal of tree
    void printSpiral(Node node)
    {
        int h = height(node);
        int i;

        /* ltr -> left to right. If this variable is set then the
           given label is transversed from left to right */
        boolean ltr = false;
        for (i = 1; i <= h; i++)
        {
            printGivenLevel(node, i, ltr);

            /*Revert ltr to traverse next level in opposite order*/
            ltr = !ltr;
        }

    }

    /* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {

            /* compute the height of each subtree */
            int lheight = height(node.left);
            int rheight = height(node.right);

            /* use the larger one */
            if (lheight > rheight)
                return (lheight + 1);
            else
                return (rheight + 1);
        }
    }

    /* Print nodes at a given level */
    void printGivenLevel(Node node, int level, boolean ltr)
    {
```

```
        if (node == null)
            return;
        if (level == 1)
            System.out.print(node.data + " ");
        else if (level > 1)
        {
            if (ltr != false)
            {
                printGivenLevel(node.left, level - 1, ltr);
                printGivenLevel(node.right, level - 1, ltr);
            }
            else
            {
                printGivenLevel(node.right, level - 1, ltr);
                printGivenLevel(node.left, level - 1, ltr);
            }
        }
    }
    /* Driver program to test the above functions */
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(7);
        tree.root.left.right = new Node(6);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(4);
        System.out.println("Spiral order traversal of Binary Tree is ");
        tree.printSpiral(tree.root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

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

Method 2 (Iterative)
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 other stack.

C++

```
 // C++ implementation of a O(n) time method for spiral order traversal
#include <iostream>
#include <stack>
using namespace std;

// Binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

void printSpiral(struct node *root)
{
    if (root == NULL)  return;   // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1;  // For levels to be printed from right to left
    stack<struct node*> s2;  // For levels to be printed from left to right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep ptinting while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty())
    {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty())
        {
            struct node *temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty())
```

```
        {
            struct node *temp = s2.top();
            s2.pop();
            cout << temp->data << " ";

            // Note that is left is pushed before right
            if (temp->left)
                s1.push(temp->left);
            if (temp->right)
                s1.push(temp->right);
        }
    }
}

// A utility function to create a new node
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(7);
    root->left->right = newNode(6);
    root->right->left  = newNode(5);
    root->right->right = newNode(4);
    cout << "Spiral Order traversal of binary tree is \n";
    printSpiral(root);

    return 0;
}
```

Java

```
 // Java implementation of an O(n) approach of level order
// traversal in spiral form

import java.util.*;

// A Binary Tree node
```

```
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{

    static Node root;

    void printSpiral(Node node)
    {
        if (node == null)
            return;   // NULL check

        // Create two stacks to store alternate levels
        Stack<Node> s1 = new Stack<Node>();// For levels to be printed from right to left
        Stack<Node> s2 = new Stack<Node>();// For levels to be printed from left to right

        // Push first level to first stack 's1'
        s1.push(node);

        // Keep ptinting while any of the stacks has some nodes
        while (!s1.empty() || !s2.empty())
        {
            // Print nodes of current level from s1 and push nodes of
            // next level to s2
            while (!s1.empty())
            {
                Node temp = s1.peek();
                s1.pop();
                System.out.print(temp.data + " ");

                // Note that is right is pushed before left
                if (temp.right != null)
                    s2.push(temp.right);

                if (temp.left != null)
                    s2.push(temp.left);

            }
```

```
            // Print nodes of current level from s2 and push nodes of
            // next level to s1
            while (!s2.empty())
            {
                Node temp = s2.peek();
                s2.pop();
                System.out.print(temp.data + " ");

                // Note that is left is pushed before right
                if (temp.left != null)
                    s1.push(temp.left);
                if (temp.right != null)
                    s1.push(temp.right);
            }
        }
    }

    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(7);
        tree.root.left.right = new Node(6);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(4);
        System.out.println("Spiral Order traversal of Binary Tree is ");
        tree.printSpiral(root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

Please write comments if you find any bug in the above program/algorithm; or if you want to share more information about spiral traversal.

Source

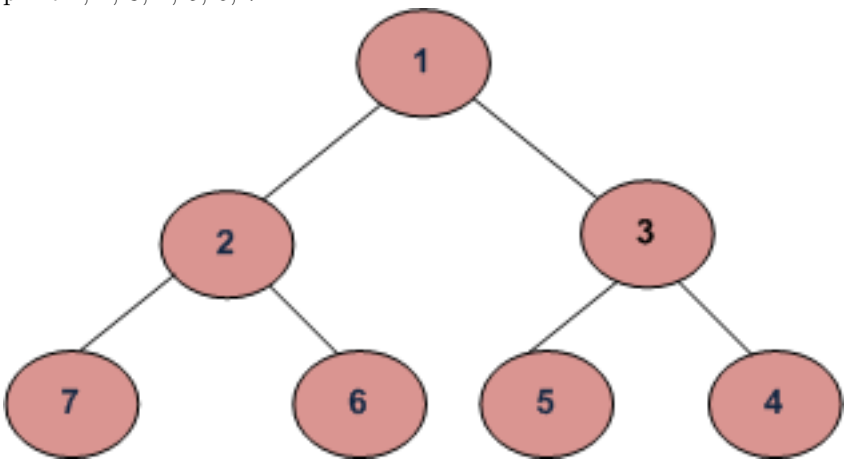
https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/

Chapter 39

Level order traversal in spiral form | Using one stack and one queue

Level order traversal in spiral form | Using one stack and one queue - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



You are allowed to use only one stack.

We have seen recursive and iterative solutions using two stacks. In this post, a solution with one stack and one queue is discussed. The idea is to keep on entering nodes like normal level order traversal, but during printing, in alternative turns push them onto the stack and print them, and in other traversals, just print them the way they are present in the queue.

Following is the CPP implementation of the idea.

```
// CPP program to print level order traversal
```

```
// in spiral form using one queue and one stack.
#include <bits/stdc++.h>
using namespace std;

struct Node {
   int data;
   Node *left, *right;
};

/* Utility function to create a new tree node */
Node* newNode(int val)
{
    Node* new_node = new Node;
    new_node->data = val;
    new_node->left = new_node->right = NULL;
    return new_node;
}

/* Function to print a tree in spiral form
   using one stack */
void printSpiralUsingOneStack(Node* root)
{
    if (root == NULL)
        return;

    stack<int> s;
    queue<Node*> q;

    bool reverse = true;
    q.push(root);
    while (!q.empty()) {

        int size = q.size();
        while (size) {
            Node* p = q.front();
            q.pop();

            // if reverse is true, push node's
            // data onto the stack, else print it
            if (reverse)
                s.push(p->data);
            else
                cout << p->data << " ";

            if (p->left)
                q.push(p->left);
            if (p->right)
                q.push(p->right);
```

```
            size--;
        }

        // print nodes from the stack if
        // reverse is true
        if (reverse) {
            while (!s.empty()) {
                cout << s.top() << " ";
                s.pop();
            }
        }

        // the next row has to be printed as
        // it is, hence change the value of
        // reverse
        reverse = !reverse;
    }
}

/*Driver program to test the above functions*/
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printSpiralUsingOneStack(root);
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Source

https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form-using-one-stack-and-one-queue/

Chapter 40

Level order traversal line by line | Set 2 (Using Two Queues)

Level order traversal line by line | Set 2 (Using Two Queues) - GeeksforGeeks

Given a Binary Tree, print the nodes level wise, each level on a new line.

```
Output:
1
2 3
4 5
```

We have discussed one solution in below article.
Print level order traversal line by line | Set 1

In this post, a different approach using two queues is discussed. We can insert the first level in first queue and print it and while popping from the first queue insert its left and right nodes into the second queue. Now start printing the second queue and before popping insert its left and right child nodes into the first queue. Continue this process till both the queues become empty.

C++

```
 // C++ program to do level order traversal line by
// line
#include <bits/stdc++.h>
using namespace std;

struct Node
{
```

```
    int data;
    Node *left, *right;
};

// Prints level order traversal line by line
// using two queues.
void levelOrder(Node *root)
{
    queue<Node *> q1, q2;

    if (root == NULL)
        return;

    // Pushing first level node into first queue
    q1.push(root);

    // Executing loop till both the queues
    // become empty
    while (!q1.empty() || !q2.empty())
    {
        while (!q1.empty())
        {
            // Pushing left child of current node in
            // first queue into second queue
            if (q1.front()->left != NULL)
                q2.push(q1.front()->left);

            // pushing right child of current node
            // in first queue into second queue
            if (q1.front()->right != NULL)
                q2.push(q1.front()->right);

            cout << q1.front()->data << " ";
            q1.pop();
        }

        cout << "\n";

        while (!q2.empty())
        {
            // pushing left child of current node
            // in second queue into first queue
            if (q2.front()->left != NULL)
                q1.push(q2.front()->left);

            // pushing right child of current
            // node in second queue into first queue
            if (q2.front()->right != NULL)
```

```
                q1.push(q2.front()->right);

            cout << q2.front()->data << " ";
            q2.pop();
        }

        cout << "\n";
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    levelOrder(root);
    return 0;
}
```

Java

```
 //Java program to do level order traversal line by
//line
import java.util.LinkedList;
import java.util.Queue;

public class GFG
{
    static class Node
    {
        int data;
        Node left;
        Node right;
```

```
    Node(int data)
    {
        this.data = data;
        left = null;
        right = null;
    }
}

// Prints level order traversal line by line
// using two queues.
static void levelOrder(Node root)
{
    Queue<Node> q1 = new LinkedList<Node>();
    Queue<Node> q2 = new LinkedList<Node>();

    if (root == null)
        return;

    // Pushing first level node into first queue
    q1.add(root);

    // Executing loop till both the queues
    // become empty
    while (!q1.isEmpty() || !q2.isEmpty())
    {

        while (!q1.isEmpty())
        {

            // Pushing left child of current node in
            // first queue into second queue
            if (q1.peek().left != null)
                q2.add(q1.peek().left);

            // pushing right child of current node
            // in first queue into second queue
            if (q1.peek().right != null)
                q2.add(q1.peek().right);

            System.out.print(q1.peek().data + " ");
            q1.remove();
        }
        System.out.println();

        while (!q2.isEmpty())
        {
```

```
                // pushing left child of current node
                // in second queue into first queue
                if (q2.peek().left != null)
                    q1.add(q2.peek().left);

                // pushing right child of current
                // node in second queue into first queue
                if (q2.peek().right != null)
                    q1.add(q2.peek().right);

                System.out.print(q2.peek().data + " ");
                q2.remove();
            }
            System.out.println();
        }
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {

        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);
        root.left.right = new Node(5);
        root.right.right = new Node(6);

        levelOrder(root);
    }
}
// This code is Contributed by Sumit Ghosh
```

Python

```
 """
Python program to do level order traversal
line by line using dual queue"""
class GFG:

    """Constructor to create a new tree node"""
    def __init__(self,data):
        self.val = data
        self.left = None
        self.right = None

    """Prints level order traversal line by
    line using two queues."""
```

```
def levelOrder(self,node):
    q1 = [] # Queue 1
    q2 = [] # Queue 2
    q1.append(node)

    """Executing loop till both the
    queues become empty"""
    while(len(q1) > 0 or len(q2) > 0):

        """Empty string to concatenate
        the string for q1"""
        concat_str_q1 = ''
        while(len(q1) > 0):

            """Poped node at the first
            pos in queue 1 i.e q1"""
            poped_node = q1.pop(0)
            concat_str_q1 += poped_node.val +' '

            """Pushing left child of current
            node in first queue into second queue"""
            if poped_node.left:
                q2.append(poped_node.left)

            """Pushing right child of current node
            in first queue into second queue"""
            if poped_node.right:
                q2.append(poped_node.right)
        print( str(concat_str_q1))
        concat_str_q1 = ''

        """Empty string to concatenate the
        string for q1"""
        concat_str_q2 = ''
        while (len(q2) > 0):

            """Poped node at the first pos
            in queue 1 i.e q1"""
            poped_node = q2.pop(0)
            concat_str_q2 += poped_node.val + ' '

            """Pushing left child of current node
            in first queue into first queue"""
            if poped_node.left:
                q1.append(poped_node.left)

            """Pushing right child of current node
            in first queue into first queue"""
```

```
                if poped_node.right:
                    q1.append(poped_node.right)
            print(str(concat_str_q2))
            concat_str_q2 = ''

""" Driver program to test above functions"""
node = GFG("1")
node.left = GFG("2")
node.right = GFG("3")
node.left.left = GFG("4")
node.left.right = GFG("5")
node.right.right = GFG("6")
node.levelOrder(node)

# This code is contributed by Vaibhav Kumar 12
```

Output :

```
1
2 3
4 5 6
```

Time Complexity : O(n)

Improved By : ParulShandilya

Source

https://www.geeksforgeeks.org/level-order-traversal-line-line-set-2-using-two-queues/

Chapter 41

Level order traversal with direction change after every two levels

Level order traversal with direction change after every two levels - GeeksforGeeks

Given a binary tree, print the level order traversal in such a way that first two levels are printed from left to right, next two levels are printed from right to left, then next two from left to right and so on. So, the problem is to reverse the direction of level order traversal of binary tree after every two levels.

Examples:

```
Input:
            1
          /   \
        2       3
      /  \     /  \
     4    5    6    7
    / \  / \  / \  / \
   8  9 3   1 4  2 7  2
     /     / \    \
    16    17  18   19
Output:
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
In the above example, first two levels
are printed from left to right, next two
```

```
levels are printed from right to left,
and then last level is printed from
left to right.
```

Approach:
We make use of queue and stack here. Queue is used for performing normal level order traversal. Stack is used for reversing the direction of traversal after every two levels.
While doing normal level order traversal, first two levels nodes are printed at the time when they are popped out from the queue. For the next two levels, we instead of printing the nodes, pushed them onto the stack. When all nodes of current level are popped out, we print the nodes in the stack. In this way, we print the nodes in right to left order by making use of the stack. Now for the next two levels we again do normal level order traversal for printing nodes from left to right. Then for the next two nodes, we make use of the stack for achieving right to left order.
In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
 // CPP program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

// A Binary Tree Node
struct Node {
    struct Node* left;
    int data;
    struct Node* right;
};

/* Function to print the level order of
   given binary tree. Direction of printing
   level order traversal of binary tree changes
   after every two levels */
void modifiedLevelOrder(struct Node* node)
{
    // For null root
    if (node == NULL)
        return;

    if (node->left == NULL && node->right == NULL) {
        cout << node->data;
        return;
    }

    // Maintain a queue for normal level order traversal
    queue<Node*> myQueue;
```

```
/* Maintain a stack for printing nodes in reverse
   order after they are popped out from queue.*/
stack<Node*> myStack;

struct Node* temp = NULL;

// sz is used for storing the count of nodes in a level
int sz;

// Used for changing the direction of level order traversal
int ct = 0;

// Used for changing the direction of level order traversal
bool rightToLeft = false;

// Push root node to the queue
myQueue.push(node);

// Run this while loop till queue got empty
while (!myQueue.empty()) {
    ct++;

    sz = myQueue.size();

    // Do a normal level order traversal
    for (int i = 0; i < sz; i++) {
        temp = myQueue.front();
        myQueue.pop();

        /*For printing nodes from left to right,
        simply print the nodes in the order in which
        they are being popped out from the queue.*/
        if (rightToLeft == false)
            cout << temp->data << " ";

        /* For printing nodes from right to left,
        push the nodes to stack instead of printing them.*/
        else
            myStack.push(temp);

        if (temp->left)
            myQueue.push(temp->left);

        if (temp->right)
            myQueue.push(temp->right);
    }
```

```
        if (rightToLeft == true) {

            // for printing the nodes in order
            // from right to left
            while (!myStack.empty()) {
                temp = myStack.top();
                myStack.pop();

                cout << temp->data << " ";
            }
        }

        /*Change the direction of printing
        nodes after every two levels.*/
        if (ct == 2) {
            rightToLeft = !rightToLeft;
            ct = 0;
        }

        cout << "\n";
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(3);
    root->left->right->right = newNode(1);
    root->right->left->left = newNode(4);
```

```
    root->right->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(2);
    root->left->right->left->left = newNode(16);
    root->left->right->left->right = newNode(17);
    root->right->left->right->left = newNode(18);
    root->right->right->left->right = newNode(19);

    modifiedLevelOrder(root);

    return 0;
}
```

Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

Time Complexity: Each node is traversed at most twice while doing level order traversal, so time complexity would be O(n).

Approach 2:
We make use of queue and stack here, but in a different way. Using macros #define ChangeDirection(Dir) ((Dir) = 1 – (Dir)). In following implementation directs the order of push operations in both queue or stack.
In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
 // CPP program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <stack>
#include <queue>

using namespace std;

#define LEFT 0
#define RIGHT 1
#define ChangeDirection(Dir) ((Dir) = 1 - (Dir))

// A Binary Tree Node
struct node
{
    int data;
```

```
    struct node *left, *right;
};

// Utility function to create a new tree node
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Function to print the level order of
   given binary tree. Direction of printing
   level order traversal of binary tree changes
   after every two levels */
void modifiedLevelOrder(struct node *root)
{
    if (!root)
        return ;

    int dir = LEFT;
    struct node *temp;
    queue <struct node *> Q;
    stack <struct node *> S;

    S.push(root);

    // Run this while loop till queue got empty
    while (!Q.empty() || !S.empty())
    {
        while (!S.empty())
        {
            temp = S.top();
            S.pop();
            cout << temp->data << " ";

            if (dir == LEFT) {
                if (temp->left)
                    Q.push(temp->left);
                if (temp->right)
                    Q.push(temp->right);
            }
            /* For printing nodes from right to left,
            push the nodes to stack instead of printing them.*/
            else {
                if (temp->right)
                    Q.push(temp->right);
```

```
                if (temp->left)
                    Q.push(temp->left);
            }
        }

        cout << endl;

            // for printing the nodes in order
            // from right to left
        while (!Q.empty())
        {
            temp = Q.front();
            Q.pop();
            cout << temp->data << " ";

            if (dir == LEFT) {
                if (temp->left)
                    S.push(temp->left);
                if (temp->right)
                    S.push(temp->right);
            } else {
                if (temp->right)
                    S.push(temp->right);
                if (temp->left)
                    S.push(temp->left);
            }
        }
        cout << endl;

        // Change the direction of traversal.
        ChangeDirection(dir);
    }
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(3);
```

```
    root->left->right->right = newNode(1);
    root->right->left->left = newNode(4);
    root->right->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(2);
    root->left->right->left->left = newNode(16);
    root->left->right->left->right = newNode(17);
    root->right->left->right->left = newNode(18);
    root->right->right->left->right = newNode(19);

    modifiedLevelOrder(root);

    return 0;
}
```

Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

Time Complexity: every node is also traversed twice. There time complexity is still O(n).

Improved By : vhinf2047

Source

https://www.geeksforgeeks.org/level-order-traversal-direction-change-every-two-levels/

Chapter 42

Maximum length of rod for Q-th person

Maximum length of rod for Q-th person - GeeksforGeeks

Given lengths of n rods in an array **a**[]. If any person picks any rod, half of the longest rod (or (max + 1) / 2) is assigned and remaining part (max – 1) / 2 is put back. It may be assumed that sufficient number of rods are always available, answer M queries given in an array q[] to find the largest length of rod available for $\mathbf{q^{ith}}$ person, provided $\mathbf{q^{i}}$ is a valid person number starting from 1.

Examples :

```
Input : a[] = {6, 5, 9, 10, 12}
        q[] = {1, 3}
Output : 12 9
The first person gets maximum length as 12.
We remove 12 from array and put back (12 -1) / 2 = 5.
Second person gets maximum length as 10.
We put back (10 - 1)/2 which is 4.
Third person gets maximum length as 9.

Input : a[] = {6, 5, 9, 10, 12}
        q[] = {3, 1, 2, 7, 4, 8, 9, 5, 10, 6}
Output : 9 12 10 5 6 4 3 6 3 5
```

Approach :
Use a stack and a queue. First sort all the lengths and push them onto a stack. Now, take the top element of stack, and divide by 2 and push the remaining length to queue. Now, from next customer onwards :

1. If stack is empty, pop front queue and push back to queue. It's half (front / 2), if non zero.

2. If queue is empty, pop from stack and push to queue it's half (top / 2), if non zero.
3. If both are non empty, compare top and front, which ever is larger should be popped, divided by 2 and then pushed back.
4. If both are empty, store is empty! Stop here!

At each step above store the length available to i^{th} customer in separate array, say "ans". Now, start answering the queries by giving ans[Q_i] as output.

Below is the implementation of above approach :

```
 // CPP code to find the length of largest
// rod available for Q-th customer
#include <bits/stdc++.h>
using namespace std;

// function to find largest length of
// rod available for Q-th customer
vector<int> maxRodLength(int ar[],
                         int n, int m)
{
    queue<int> q;

    // sort the rods according to lengths
    sort(ar, ar + n);

    // Push sorted elements to a stack
    stack<int> s;
    for (int i = 0; i < n; i++)
        s.push(ar[i]);

    vector<int> ans;

    while (!s.empty() || !q.empty()) {
        int val;

        // If queue is empty -> pop from stack
        // and push to queue it's half(top/2),
        // if non zero.
        if (q.empty()) {
            val = s.top();
            ans.push_back(val);
            s.pop();
            val /= 2;

            if (val)
                q.push(val);
        }
        // If stack is empty -> pop front from
```

```
        // queue and push back to queue it's
        // half(front/2), if non zero.
        else if (s.empty()) {
            val = q.front();
            ans.push_back(val);
            q.pop();
            val /= 2;
            if (val != 0)
                q.push(val);
        }
        // If both are non empty ->
        // compare top and front, whichsoever is
        // larger should be popped, divided by 2
        // and then pushed back.
        else {
            val = s.top();
            int fr = q.front();
            if (fr > val) {
                ans.push_back(fr);
                q.pop();
                fr /= 2;
                if (fr)
                    q.push(fr);
            }
            else {
                ans.push_back(val);
                s.pop();
                val /= 2;
                if (val)
                    q.push(val);
            }
        }
    }

    return ans;
}

// Driver code
int main()
{
    // n : number of rods
    // m : number of queries
    int n = 5, m = 10;

    int ar[n] = { 6, 5, 9, 10, 12 };

    vector<int> ans = maxRodLength(ar, n, m);
```

```
    int query[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int size = sizeof(query) / sizeof(query[0]);
    for (int i = 0; i < size; i++)
        cout << ans[query[i] - 1] << " ";

    return 0;
}
```

Output:

```
12 10 9 6 6 5 5 4 3 3
```

Time complexity : O(N log(N))

Source

https://www.geeksforgeeks.org/maximum-length-rod-q-th-person/

Chapter 43

Minimum number of bracket reversals needed to make an expression balanced

Minimum number of bracket reversals needed to make an expression balanced - GeeksforGeeks

Given an expression with only '}' and '{'. The expression may not be balanced. Find minimum number of bracket reversals to make the expression balanced.

Examples:

```
Input:  exp = "}{"
Output: 2
We need to change '}' to '{' and '{' to
'}' so that the expression becomes balanced,
the balanced expression is '{}'

Input:  exp = "{{{"
Output: Can't be made balanced using reversals

Input:  exp = "{{{{"
Output: 2

Input:  exp = "{{{{}}"
Output: 1

Input:  exp = "}{{}}{{{"
Output: 3
```

One simple observation is, the string can be balanced only if total number of brackets is even (there must be equal no of '{' and '}')

A **Naive Solution** is to consider every bracket and recursively count number of reversals by taking two cases (i) keeping the bracket as it is (ii) reversing the bracket. If we get a balanced expression, we update result if number of steps followed for reaching here is smaller than the minimum so far. Time complexity of this solution is $O(2^n)$.

An **Efficient Solution** can solve this problem in O(n) time. The idea is to first remove all balanced part of expression. For example, convert "}**{{}}**{{{" to "}{{{" by removing highlighted part. If we take a closer look, we can notice that, after removing balanced part, we always end up with an expression of the form }}…}{{…{, an expression that contains 0 or more number of closing brackets followed by 0 or more numbers of opening brackets.

How many minimum reversals are required for an expression of the form "}}..}{{..{" ?. Let m be the total number of closing brackets and n be the number of opening brackets. We need m/2 + n/2 reversals. For example }}}}{{ requires 2+1 reversals.

Below is implementation of above idea.

C++

```
 // C++ program to find minimum number of
// reversals required to balance an expression
#include<bits/stdc++.h>
using namespace std;


// Returns count of minimum reversals for making
// expr balanced. Returns -1 if expr cannot be
// balanced.
int countMinReversals(string expr)
{
    int len = expr.length();

    // length of expression must be even to make
    // it balanced by using reversals.
    if (len%2)
       return -1;

    // After this loop, stack contains unbalanced
    // part of expression, i.e., expression of the
    // form "}}..}{{..{"
    stack<char> s;
    for (int i=0; i<len; i++)
    {
        if (expr[i]=='}' && !s.empty())
        {
            if (s.top()=='{')
                s.pop();
```

```
            else
                s.push(expr[i]);
        }
        else
            s.push(expr[i]);
    }

    // Length of the reduced expression
    // red_len = (m+n)
    int red_len = s.size();

    // count opening brackets at the end of
    // stack
    int n = 0;
    while (!s.empty() && s.top() == '{')
    {
        s.pop();
        n++;
    }

    // return ceil(m/2) + ceil(n/2) which is
    // actually equal to (m+n)/2 + n%2 when
    // m+n is even.
    return (red_len/2 + n%2);
}

// Driver program to test above function
int main()
{
   string expr = "}}{{";
   cout << countMinReversals(expr);
   return 0;
}
```

Java

```
 //Java Code to count minimum reversal for
//making an expression balanced.

import java.util.Stack;

public class GFG
{

    // Method count minimum reversal for
    //making an expression balanced.
    //Returns -1 if expression cannot be balanced
    static int countMinReversals(String expr)
```

```
{
    int len = expr.length();

    // length of expression must be even to make
    // it balanced by using reversals.
    if (len%2 != 0)
    return -1;

    // After this loop, stack contains unbalanced
    // part of expression, i.e., expression of the
    // form "}}..}{{..{"
    Stack<Character> s=new Stack<>();

    for (int i=0; i<len; i++)
    {
        char c = expr.charAt(i);
        if (c =='}' && !s.empty())
        {
            if (s.peek()=='{')
                s.pop();
            else
                s.push(c);
        }
        else
            s.push(c);
    }

    // Length of the reduced expression
    // red_len = (m+n)
    int red_len = s.size();

    // count opening brackets at the end of
    // stack
    int n = 0;
    while (!s.empty() && s.peek() == '{')
    {
        s.pop();
        n++;
    }

    // return ceil(m/2) + ceil(n/2) which is
    // actually equal to (m+n)/2 + n%2 when
    // m+n is even.
    return (red_len/2 + n%2);
}

// Driver method
public static void main(String[] args)
```

```
    {
        String expr = "}}{{";

        System.out.println(countMinReversals(expr));
    }

}
//This code is contributed by Sumit Ghosh
```

Output:

```
2
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Thanks to Utkarsh Trivedi for suggesting above approach.

Source

https://www.geeksforgeeks.org/minimum-number-of-bracket-reversals-needed-to-make-an-expression-balanced/

Chapter 44

Minimum steps to reach target by a Knight | Set 1

Minimum steps to reach target by a Knight | Set 1 - GeeksforGeeks

Given a square chessboard of N x N size, the position of Knight and position of a target is given. We need to find out minimum steps a Knight will take to reach the target position.

Examples:

```
In above diagram Knight takes 3 step to reach
from (4, 5) to (1, 1) (4, 5) -> (5, 3) -> (3, 2)
-> (1, 1)  as shown in diagram
```

This problem can be seen as shortest path in unweighted graph. Therefore we use BFS to solve this problem. We try all 8 possible positions where a Knight can reach from its position. If reachable position is not already visited and is inside the board, we push this state into queue with distance 1 more than its parent state. Finally we return distance of target position, when it gets pop out from queue.
Below code implements BFS for searching through cells, where each cell contains its coordinate and distance from starting node. In worst case, below code visits all cells of board, making worst-case time complexity as O(N^2)

```
 // C++ program to find minimum steps to reach to
// specific cell in minimum moves by Knight
#include <bits/stdc++.h>
using namespace std;

// structure for storing a cell's data
```

```
struct cell
{
    int x, y;
    int dis;
    cell() {}
    cell(int x, int y, int dis) : x(x), y(y), dis(dis) {}
};

// Utility method returns true if (x, y) lies
// inside Board
bool isInside(int x, int y, int N)
{
    if (x >= 1 && x <= N && y >= 1 && y <= N)
        return true;
    return false;
}

// Method returns minimum step to reach target position
int minStepToReachTarget(int knightPos[], int targetPos[],
                                                    int N)
{
    // x and y direction, where a knight can move
    int dx[] = {-2, -1, 1, 2, -2, -1, 1, 2};
    int dy[] = {-1, -2, -2, -1, 1, 2, 2, 1};

    // queue for storing states of knight in board
    queue<cell> q;

    // push starting position of knight with 0 distance
    q.push(cell(knightPos[0], knightPos[1], 0));

    cell t;
    int x, y;
    bool visit[N + 1][N + 1];

    // make all cell unvisited
    for (int i = 1; i <= N; i++)
        for (int j = 1; j <= N; j++)
            visit[i][j] = false;

    // visit starting state
    visit[knightPos[0]][knightPos[1]] = true;

    // loop untill we have one element in queue
    while (!q.empty())
    {
        t = q.front();
        q.pop();
```

```
        // if current cell is equal to target cell,
        // return its distance
        if (t.x == targetPos[0] && t.y == targetPos[1])
            return t.dis;

        // loop for all reachable states
        for (int i = 0; i < 8; i++)
        {
            x = t.x + dx[i];
            y = t.y + dy[i];

            // If reachable state is not yet visited and
            // inside board, push that state into queue
            if (isInside(x, y, N) && !visit[x][y]) {
                visit[x][y] = true;
                q.push(cell(x, y, t.dis + 1));
            }
        }
    }
}

// Driver code to test above methods
int main()
{
    int N = 30;
    int knightPos[] = {1, 1};
    int targetPos[] = {30, 30};
    cout << minStepToReachTarget(knightPos, targetPos, N);
    return 0;
}
```

Output:

```
20
```

Source

https://www.geeksforgeeks.org/minimum-steps-reach-target-knight/

Chapter 45

Minimum sum of squares of character counts in a given string after removing k characters

Minimum sum of squares of character counts in a given string after removing k characters - GeeksforGeeks

Given a string of lowercase alphabets and a number k, the task is to print the minimum value of the string after removal of 'k' characters. The value of a string is defined as the sum of squares of the count of each distinct character. For example consider the string "saideep", here frequencies of characters are s-1, a-1, i-1, e-2, d-1, p-1 and value of the string is 1^2 + 1^2 + 1^2 + 1^2 + 1^1 + 2^2 = 9.

Expected Time Complexity : O(n)

Examples:

```
Input :  str = abccc, K = 1
Output :  6
We remove c to get the value as 11 + 11 + 22

Input :  str = aaab, K = 2
Output :  2
```

Asked In : Amazon

One clear observation is that we need to remove character with highest frequency. One trick is the character ma

A **Simple solution** is to use sorting technique through all current highest frequency reduce up to k times. For After every reduce again sort frequency array.

A **Better Solution** used to Priority Queue which has to the highest element on top.

1. Initialize empty priority queue.
2. Count frequency of each character and Store into temp array.
3. Remove K characters which have highest frequency from queue.
4. Finally Count Sum of square of each element and return it.

Below is the implementation of the above idea.

C++

```
 // C++ program to find min sum of squares
// of characters after k removals
#include <bits/stdc++.h>
using namespace std;

const int MAX_CHAR = 26;

// Main Function to calculate min sum of
// squares of characters after k removals
int minStringValue(string str, int k)
{
    int l = str.length(); // find length of string

    // if K is greater than length of string
    // so reduced string will become 0
    if (k >= l)
        return 0;

    // Else find Frequency of each character and
    // store in an array
    int frequency[MAX_CHAR] = {0};
    for (int i=0; i<l; i++)
        frequency[str[i]-'a']++;

    // Push each char frequency into a priority_queue
    priority_queue<int> q;
    for (int i=0; i<MAX_CHAR; i++)
        q.push(frequency[i]);


    // Removal of K characters
    while (k--)
    {
```

```
        // Get top element in priority_queue,
        // remove it. Decrement by 1 and again
        // push into priority_queue
        int temp = q.top();
        q.pop();
        temp = temp-1;
        q.push(temp);
    }

    // After removal of K characters find sum
    // of squares of string Value
    int result = 0; // Initialize result
    while (!q.empty())
    {
        int temp = q.top();
        result += temp*temp;
        q.pop();
    }

    return result;
}

// Driver Code
int main()
{
    string str = "abbccc";   // Input 1
    int k = 2;
    cout << minStringValue(str, k) << endl;

    str = "aaab";           // Input 2
    k = 2;
    cout << minStringValue(str, k);

    return 0;
}
```

Java

```
 // Java program to find min sum of squares
// of characters after k removals
import java.util.Comparator;
import java.util.PriorityQueue;
public class GFG {

    static final int MAX_CHAR = 26;

    // Defining a comparator class
    static class IntCompare implements Comparator<Integer>{
```

```
    @Override
    public int compare(Integer arg0, Integer arg1) {
        if(arg0 > arg1)
            return -1;
        else if(arg0 < arg1)
            return 1;
        else
            return 0;
    }
}

// Main Function to calculate min sum of
// squares of characters after k removals
static int minStringValue(String str, int k)
{
    int l = str.length(); // find length of string

    // if K is greater than length of string
    // so reduced string will become 0
    if (k >= l)
        return 0;

    // Else find Frequency of each character and
    // store in an array
    int[] frequency = new int[MAX_CHAR];
    for (int i=0; i<l; i++)
        frequency[str.charAt(i)-'a']++;


    // creating object for comparator
    Comparator<Integer> c = new IntCompare();

    // creating a priority queue with comparator
    // such that elements in the queue are in
    // descending order.
    PriorityQueue<Integer> q = new PriorityQueue<>(c);

    // Push each char frequency into a priority_queue
    for (int i = 0; i < MAX_CHAR; i++){
        if(frequency[i] != 0)
            q.add(frequency[i]);
    }


    // Removal of K characters
    while (k != 0)
    {
        // Get top element in priority_queue,
```

```
            // remove it. Decrement by 1 and again
            // push into priority_queue
            int temp = q.peek();
            q.poll();
            temp = temp-1;
            q.add(temp);
            k--;
        }

        // After removal of K characters find sum
        // of squares of string Value
        int result = 0; // Initialize result
        while (!q.isEmpty())
        {
            int temp = q.peek();
            result += temp*temp;
            q.poll();
        }

        return result;
    }

    // Driver Code
    public static void main(String args[])
    {
        String str = "abbccc";   // Input 1
        int k = 2;
        System.out.println(minStringValue(str, k));

        str = "aaab";           // Input 2
        k = 2;
        System.out.println(minStringValue(str, k));
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
6
2
```

Time Complexity : O(n)

Source

https://www.geeksforgeeks.org/minimum-sum-squares-characters-counts-given-string-removing-k-characters/

Chapter 46

Minimum sum of two numbers formed from digits of an array

Minimum sum of two numbers formed from digits of an array - GeeksforGeeks

Given an array of digits (values are from 0 to 9), find the minimum possible sum of two numbers formed from digits of the array. All digits of given array must be used to form the two numbers.

Examples:

```
Input: [6, 8, 4, 5, 2, 3]
Output: 604
The minimum sum is formed by numbers
358 and 246

Input: [5, 3, 0, 7, 4]
Output: 82
The minimum sum is formed by numbers
35 and 047
```

Since we want to minimize the sum of two numbers to be formed, we must divide all digits in two halves and assign half-half digits to them. We also need to make sure that the leading digits are smaller.
We build a Min Heap with the elements of the given array, which takes O(n) worst time. Now we retrieve min values (2 at a time) of array, by polling from the Priority Queue and append these two min values to our numbers, till the heap becomes empty, i.e., all the elements of array get exhausted. We return the sum of two formed numbers, which is our required answer.

C/C++

```
 // C++ program to find minimum sum of two numbers
// formed from all digits in a given array.
#include<bits/stdc++.h>
using namespace std;

// Returns sum of two numbers formed
// from all digits in a[]
int minSum(int arr[], int n)
{
    // min Heap
    priority_queue <int, vector<int>, greater<int> > pq;

    // to store the 2 numbers formed by array elements to
    // minimize the required sum
    string num1, num2;

    // Adding elements in Priority Queue
    for(int i=0; i<n; i++)
        pq.push(arr[i]);

    // checking if the priority queue is non empty
    while(!pq.empty())
    {
        // appending top of the queue to the string
        num1+=(48 + pq.top());
        pq.pop();
        if(!pq.empty())
        {
            num2+=(48 + pq.top());
            pq.pop();
        }
    }

    // converting string to integer
    int a = atoi(num1.c_str());
    int b = atoi(num2.c_str());

    // returning the sum
    return a+b;
}

int main()
{
    int arr[] = {6, 8, 4, 5, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout<<minSum(arr, n)<<endl;
    return 0;
}
```

```
// Contributed By: Harshit Sidhwa
```

Java

```
 // Java program to find minimum sum of two numbers
// formed from all digits in a given array.
import java.util.PriorityQueue;

class MinSum
{
    // Returns sum of two numbers formed
    // from all digits in a[]
    public static long solve(int[] a)
    {
        // min Heap
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

        // to store the 2 numbers formed by array elements to
        // minimize the required sum
        StringBuilder num1 = new StringBuilder();
        StringBuilder num2 = new StringBuilder();

        // Adding elements in Priority Queue
        for (int x : a)
            pq.add(x);

        // checking if the priority queue is non empty
        while (!pq.isEmpty())
        {
            num1.append(pq.poll()+ "");
            if (!pq.isEmpty())
                num2.append(pq.poll()+ "");
        }

        // the required sum calculated
        long sum = Long.parseLong(num1.toString()) +
                   Long.parseLong(num2.toString());

        return sum;
    }

    // Driver code
    public static void main (String[] args)
    {
        int arr[] = {6, 8, 4, 5, 2, 3};
        System.out.println("The required sum is "+ solve(arr));
    }
}
```

Output:

```
The required sum is 604
```

Source

https://www.geeksforgeeks.org/minimum-sum-two-numbers-formed-digits-array-2/

Chapter 47

Minimum time required to rot all oranges

Minimum time required to rot all oranges - GeeksforGeeks

Given a matrix of dimension m*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

```
0: Empty cell

1: Cells have fresh oranges

2: Cells have rotten oranges
```

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

Examples:

```
Input:  arr[][C] = { {2, 1, 0, 2, 1},
                     {1, 0, 1, 2, 1},
                     {1, 0, 0, 2, 1}};
Output:
All oranges can become rotten in 2 time frames.


Input:  arr[][C] = { {2, 1, 0, 2, 1},
                     {0, 0, 1, 2, 1},
```

```
                    {1, 0, 0, 2, 1}};
Output:
All oranges cannot be rotten.
```

The idea is to user Breadth First Search. Below is algorithm.

```
1) Create an empty Q.
2) Find all rotten oranges and enqueue them to Q.  Also enqueue
   a delimiter to indicate beginning of next time frame.
3) While Q is not empty do following
   3.a) While delimiter in Q is not reached
       (i) Dequeue an orange from queue, rot all adjacent oranges.
           While rotting the adjacents, make sure that time frame
           is incremented only once. And time frame is not icnremented
           if there are no adjacent oranges.
   3.b) Dequeue the old delimiter and enqueue a new delimiter. The
        oranges rotten in previous time frame lie between the two
        delimiters.
```

Below is implementation of the above idea.

C++

```
 // C++ program to find minimum time required to make all
// oranges rotten
#include<bits/stdc++.h>
#define R 3
#define C 5
using namespace std;

// function to check whether a cell is valid / invalid
bool isvalid(int i, int j)
{
    return (i >= 0 && j >= 0 && i < R && j < C);
}

// structure for storing coordinates of the cell
struct ele {
    int x, y;
};

// Function to check whether the cell is delimiter
// which is (-1, -1)
bool isdelim(ele temp)
{
    return (temp.x == -1 && temp.y == -1);
```

```
}

// Function to check whether there is still a fresh
// orange remaining
bool checkall(int arr[][C])
{
    for (int i=0; i<R; i++)
       for (int j=0; j<C; j++)
          if (arr[i][j] == 1)
             return true;
    return false;
}

// This function finds if it is possible to rot all oranges or not.
// If possible, then it returns minimum time required to rot all,
// otherwise returns -1
int rotOranges(int arr[][C])
{
    // Create a queue of cells
    queue<ele> Q;
    ele temp;
    int ans = 0;

    // Store all the cells having rotten orange in first time frame
    for (int i=0; i<R; i++)
    {
       for (int j=0; j<C; j++)
       {
            if (arr[i][j] == 2)
            {
                temp.x = i;
                temp.y = j;
                Q.push(temp);
            }
        }
    }

    // Separate these rotten oranges from the oranges which will rotten
    // due the oranges in first time frame using delimiter which is (-1, -1)
    temp.x = -1;
    temp.y = -1;
    Q.push(temp);

    // Process the grid while there are rotten oranges in the Queue
    while (!Q.empty())
    {
        // This flag is used to determine whether even a single fresh
        // orange gets rotten due to rotten oranges in current time
```

```
// frame so we can increase the count of the required time.
bool flag = false;

// Process all the rotten oranges in current time frame.
while (!isdelim(Q.front()))
{
    temp = Q.front();

    // Check right adjacent cell that if it can be rotten
    if (isvalid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)
    {
        // if this is the first orange to get rotten, increase
        // count and set the flag.
        if (!flag) ans++, flag = true;

        // Make the orange rotten
        arr[temp.x+1][temp.y] = 2;

        // push the adjacent orange to Queue
        temp.x++;
        Q.push(temp);

        temp.x--; // Move back to current cell
    }

    // Check left adjacent cell that if it can be rotten
    if (isvalid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x-1][temp.y] = 2;
        temp.x--;
        Q.push(temp); // push this cell to Queue
        temp.x++;
    }

    // Check top adjacent cell that if it can be rotten
    if (isvalid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x][temp.y+1] = 2;
        temp.y++;
        Q.push(temp); // Push this cell to Queue
        temp.y--;
    }

    // Check bottom adjacent cell if it can be rotten
    if (isvalid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x][temp.y-1] = 2;
        temp.y--;
```

```
                Q.push(temp); // push this cell to Queue
            }

            Q.pop();
        }

        // Pop the delimiter
        Q.pop();

        // If oranges were rotten in current frame than separate the
        // rotten oranges using delimiter for the next frame for processing.
        if (!Q.empty()) {
            temp.x = -1;
            temp.y = -1;
            Q.push(temp);
        }

        // If Queue was empty than no rotten oranges left to process so exit
    }

    // Return -1 if all arranges could not rot, otherwise -1.
    return (checkall(arr))? -1: ans;
}

// Drive program
int main()
{
    int arr[][C] = { {2, 1, 0, 2, 1},
                     {1, 0, 1, 2, 1},
                     {1, 0, 0, 2, 1}};
    int ans = rotOranges(arr);
    if (ans == -1)
        cout << "All oranges cannot rotn";
    else
         cout << "Time required for all oranges to rot => " << ans << endl;
    return 0;
}
```

Java

```
 //Java program to find minimum time required to make all
//oranges rotten

import java.util.LinkedList;
import java.util.Queue;

public class RotOrange
{
```

```
public final static int R = 3;
public final static int C = 5;

// structure for storing coordinates of the cell
static class Ele
{
    int x = 0;
    int y = 0;
    Ele(int x,int y)
    {
        this.x = x;
        this.y = y;
    }
}

// function to check whether a cell is valid / invalid
static boolean isValid(int i, int j)
{
    return (i >= 0 && j >= 0 && i < R && j < C);
}


// Function to check whether the cell is delimiter
// which is (-1, -1)
static boolean isDelim(Ele temp)
{
    return (temp.x == -1 && temp.y == -1);
}

// Function to check whether there is still a fresh
// orange remaining
static boolean checkAll(int arr[][])
{
     for (int i=0; i<R; i++)
           for (int j=0; j<C; j++)
              if (arr[i][j] == 1)
                 return true;
     return false;
}

// This function finds if it is possible to rot all oranges or not.
// If possible, then it returns minimum time required to rot all,
// otherwise returns -1
static int rotOranges(int arr[][])
{
    // Create a queue of cells
    Queue<Ele> Q=new LinkedList<>();
    Ele temp;
```

```
int ans = 0;
 // Store all the cells having rotten orange in first time frame
for (int i=0; i < R; i++)
   for (int j=0; j < C; j++)
       if (arr[i][j] == 2)
           Q.add(new Ele(i,j));

// Separate these rotten oranges from the oranges which will rotten
// due the oranges in first time frame using delimiter which is (-1, -1)
Q.add(new Ele(-1,-1));

// Process the grid while there are rotten oranges in the Queue
while(!Q.isEmpty())
{
    // This flag is used to determine whether even a single fresh
    // orange gets rotten due to rotten oranges in current time
    // frame so we can increase the count of the required time.
    boolean flag = false;

    // Process all the rotten oranges in current time frame.
    while(!isDelim(Q.peek()))
    {
        temp = Q.peek();

        // Check right adjacent cell that if it can be rotten
        if(isValid(temp.x+1, temp.y+1) && arr[temp.x+1][temp.y] == 1)
        {
            if(!flag)
            {
                // if this is the first orange to get rotten, increase
                // count and set the flag.
                ans++;
                flag = true;
            }
            // Make the orange rotten
            arr[temp.x+1][temp.y] = 2;

            // push the adjacent orange to Queue
            temp.x++;
            Q.add(new Ele(temp.x,temp.y));

            // Move back to current cell
            temp.x--;
        }

        // Check left adjacent cell that if it can be rotten
        if (isValid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1)
         {
```

```
                if (!flag)
                {
                    ans++;
                    flag = true;
                }
                arr[temp.x-1][temp.y] = 2;
                temp.x--;
                Q.add(new Ele(temp.x,temp.y)); // push this cell to Queue
                temp.x++;
            }

            // Check top adjacent cell that if it can be rotten
            if (isValid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {
                if(!flag)
                {
                    ans++;
                    flag = true;
                }
                arr[temp.x][temp.y+1] = 2;
                temp.y++;
                Q.add(new Ele(temp.x,temp.y)); // Push this cell to Queue
                temp.y--;
            }

            // Check bottom adjacent cell if it can be rotten
            if (isValid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1)
            {
                if (!flag)
                {
                    ans++;
                    flag = true;
                }
                arr[temp.x][temp.y-1] = 2;
                temp.y--;
                Q.add(new Ele(temp.x,temp.y)); // push this cell to Queue
            }
            Q.remove();

        }
        // Pop the delimiter
        Q.remove();

        // If oranges were rotten in current frame than separate the
        // rotten oranges using delimiter for the next frame for processing.
        if (!Q.isEmpty())
        {
            Q.add(new Ele(-1,-1));
        }
```

```
            // If Queue was empty than no rotten oranges left to process so exit
        }

        // Return -1 if all arranges could not rot, otherwise -1.s
        return (checkAll(arr))? -1: ans;

    }

    // Drive program
    public static void main(String[] args)
    {
        int arr[][] = { {2, 1, 0, 2, 1},
                        {1, 0, 1, 2, 1},
                        {1, 0, 0, 2, 1}};
        int ans = rotOranges(arr);
        if(ans == -1)
            System.out.println("All oranges cannot rot");
        else
            System.out.println("Time required for all oranges to rot = " + ans);
    }

}
//This code is contributed by Sumit Ghosh
```

Output:

```
Time required for all oranges to rot => 2
```

Thanks to Gaurav Ahirwar for suggesting above solution.

Source

https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/

Chapter 48

Multi Source Shortest Path in Unweighted Graph

Multi Source Shortest Path in Unweighted Graph - GeeksforGeeks

Suppose there are n towns connected by m bidirectional roads. There are s towns among them with a police station. We want to find out the distance of each town from the nearest police station. If the town itself has one the distance is 0.

Example:

```
Input :
Number of Vertices = 6
Number of Edges = 9
Towns with Police Station : 1, 5
Edges:
1 2
1 6
2 6
2 3
3 6
5 4
6 5
3 4
5 3

Output :
1 0
2 1
3 1
4 1
5 0
```

```
6 1
```

Naive Approach: We can loop through the vertices and from each vertex run a BFS to find the closest town with police station from that vertex. This will take O(V.E).

Naive approach implementation using BFS from each vertex:

```
 // cpp program to demonstrate distance to
// nearest source problem using BFS
// from each vertex
#include <bits/stdc++.h>
using namespace std;
#define N 100000 + 1
#define inf 1000000

// This array stores the distances of the
// vertices from the nearest source
int dist[N];

// a hash array where source[i] = 1
// means vertex i is a source
int source[N];

// The BFS Queue
// The pairs are of the form (vertex, distance
// from current source)
deque<pair<int, int> > BFSQueue;

// visited array for remembering visited vertices
int visited[N];

// The BFS function
void BFS(vector<int> graph[], int start)
{
    // clearing the queue
    while (!BFSQueue.empty())
        BFSQueue.pop_back();

    // push_back starting vertices
    BFSQueue.push_back({ start, 0 });

    while (!BFSQueue.empty()) {

        int s = BFSQueue.front().first;
        int d = BFSQueue.front().second;
        visited[s] = 1;
        BFSQueue.pop_front();
```

```
        // stop at the first source we reach during BFS
        if (source[s] == 1) {
            dist[start] = d;
            return;
        }

        // Pushing the adjacent unvisited vertices
        // with distance from current source = this
        // vertex's distance  + 1
        for (int i = 0; i < graph[s].size(); i++)
            if (visited[graph[s][i]] == 0)
                BFSQueue.push_back({ graph[s][i], d + 1 });
    }
}

// This function calculates the distance of each
// vertex from nearest source
void nearestTown(vector<int> graph[], int n,
                      int sources[], int S)
{

    // reseting the source hash array
    for (int i = 1; i <= n; i++)
        source[i] = 0;
    for (int i = 0; i <= S - 1; i++)
        source[sources[i]] = 1;

    // loop through all the vertices and run
    // a BFS from each vertex to find the distance
    // to nearest town from it
    for (int i = 1; i <= n; i++) {
        for (int i = 1; i <= n; i++)
            visited[i] = 0;
        BFS(graph, i);
    }

    // Printing the distances
    for (int i = 1; i <= n; i++)
        cout << i << " " << dist[i] << endl;
}

void addEdge(vector<int> graph[], int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Driver Code
```

```
int main()
{    // Number of vertices
    int n = 6;

    vector<int> graph[n + 1];

    // Edges
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 6);
    addEdge(graph, 2, 6);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 6);
    addEdge(graph, 5, 4);
    addEdge(graph, 6, 5);
    addEdge(graph, 3, 4);
    addEdge(graph, 5, 3);

    // Sources
    int sources[] = { 1, 5 };

    int S = sizeof(sources) / sizeof(sources[0]);

    nearestTown(graph, n, sources, S);

    return 0;
}
```

Output:

```
1 0
2 1
3 1
4 1
5 0
6 1
```

Time Complexity: *O(V.E)*

Efficient Method A better method is to use the Djikstra's algorithm in a modified way. Let's consider one of the sources as the original source and the other sources to be vertices with 0 cost paths from the original source. Thus we push all the sources into the Djikstra Queue with distance = 0, and the rest of the vertices with distance = infinity. The minimum distance of each vertex from the original source now calculated using the Dijkstra's Algorithm are now essentially the distances from the nearest source.
Explanation: The C++ implementation uses a set of pairs (distance from the source, vertex) sorted according to the distance from the source. Initially, the set contains the sources with distance = 0 and all the other vertices with distance = infinity.

On each step, we will go to the vertex with minimum distance(d) from source, i.e, the first element of the set (the source itself in the first step with distance = 0). We go through all it's adjacent vertices and if the distance of any vertex is > d + 1 we replace its entry in the set with the new distance. Then we remove the current vertex from the set. We continue this until the set is empty.
The idea is there cannot be a shorter path to the vertex at the front of the set than the current one since any other path will be a sum of a longer path (>= it's length) and a non-negative path length (unless we are considering negative edges).
Since all the sources have a distance = 0, in the beginning, the adjacent non-source vertices will get a distance = 1. All vertices will get distance = distance from their nearest source.

Implementation of Efficient Approach:

```
 // cpp program to demonstrate
// multi-source BFS
#include <bits/stdc++.h>
using namespace std;
#define N 100000 + 1
#define inf 1000000

// This array stores the distances of the vertices
// from the nearest source
int dist[N];

// This Set contains the vertices not yet visited in
// increasing order of distance from the nearest source
// calculated till now
set<pair<int, int> > Q;

// Util function for Multi-Source BFS
void multiSourceBFSUtil(vector<int> graph[], int s)
{
    set<pair<int, int> >::iterator it;
    int i;
    for (i = 0; i < graph[s].size(); i++) {
        int v = graph[s][i];
        if (dist[s] + 1 < dist[v]) {

            // If a shorter path to a vertex is
            // found than the currently stored
            // distance replace it in the Q
            it = Q.find({ dist[v], v });
            Q.erase(it);
            dist[v] = dist[s] + 1;
            Q.insert({ dist[v], v });
        }
    }

    // Stop when the Q is empty -> All
```

```
    // vertices have been visited. And we only
    // visit a vertex when we are sure that a
    // shorter path to that vertex is not
    // possible
    if (Q.size() == 0)
        return;

    // Go to the first vertex in Q
    // and remove it from the Q
    it = Q.begin();
    int next = it->second;
    Q.erase(it);

    multiSourceBFSUtil(graph, next);
}

// This function calculates the distance of
// each vertex from nearest source
void multiSourceBFS(vector<int> graph[], int n,
                          int sources[], int S)
{
    // a hash array where source[i] = 1
    // means vertex i is a source
    int source[n + 1];

    for (int i = 1; i <= n; i++)
        source[i] = 0;
    for (int i = 0; i <= S - 1; i++)
        source[sources[i]] = 1;

    for (int i = 1; i <= n; i++) {
        if (source[i]) {
            dist[i] = 0;
            Q.insert({ 0, i });
        }
        else {
            dist[i] = inf;
            Q.insert({ inf, i });
        }
    }

    set<pair<int, int> >::iterator itr;

    // Get the vertex with lowest distance,
    itr = Q.begin();

    // currently one of the souces with distance = 0
    int start = itr->second;
```

```
    multiSourceBFSUtil(graph, start);

    // Printing the distances
    for (int i = 1; i <= n; i++)
        cout << i << " " << dist[i] << endl;
}

void addEdge(vector<int> graph[], int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Driver Code
int main()
{
    // Number of vertices
    int n = 6;

    vector<int> graph[n + 1];

    // Edges
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 6);
    addEdge(graph, 2, 6);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 6);
    addEdge(graph, 5, 4);
    addEdge(graph, 6, 5);
    addEdge(graph, 3, 4);
    addEdge(graph, 5, 3);

    // Sources
    int sources[] = { 1, 5 };

    int S = sizeof(sources) / sizeof(sources[0]);

    multiSourceBFS(graph, n, sources, S);

    return 0;
}
```

Output:

```
1 0
2 1
```

```
3 1
4 1
5 0
6 1
```

Time Complexity: *O(E.logV)*

Source

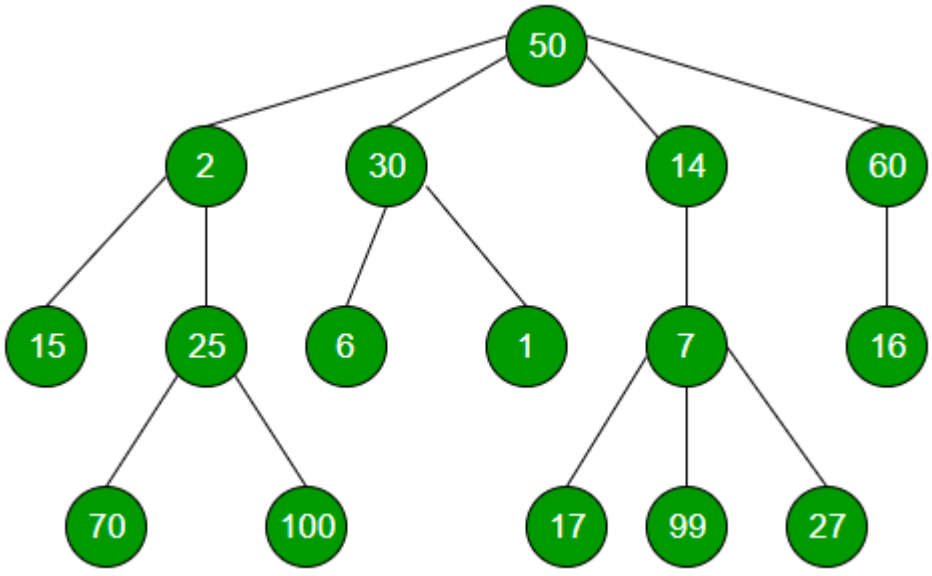
https://www.geeksforgeeks.org/multi-source-shortest-path-in-unweighted-graph/

Chapter 49

Number of siblings of a given Node in n-ary Tree

Number of siblings of a given Node in n-ary Tree - GeeksforGeeks

Given an N-ary tree, find the number of siblings of given node x. Assume that x exists in the given n-ary tree.



Example :

```
Input : 30
Output : 3
```

Approach : For every node in the given n-ary tree, push the children of the current node in the queue. While adding the children of current node in queue, check if any children is equal to the given value x or not. If yes, then return the number of siblings of x.

Below is the implementation of the above idea :

```
 // C++ program to find number
// of siblings of a given node
#include <bits/stdc++.h>
using namespace std;

// Represents a node of an n-ary tree
class Node
{
public:
    int key;
    vector<Node*> child;

    Node(int data)
    {
        key = data;
    }
};

// Function to calculate number
// of siblings of a given node
int numberOfSiblings(Node* root, int x)
{
    if (root == NULL)
        return 0;

    // Creating a queue and
    // pushing the root
    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        int n = q.size();

        // If this node has children
        while (n > 0) {

            // Dequeue an item from queue and
            // check if it is equal to x If YES,
            // then return number of children
            Node* p = q.front();
            q.pop();
```

```
            // Enqueue all children of
            // the dequeued item
            for (int i = 0; i < p->child.size(); i++)
            {
                // If the value of children
                // is equal to x, then return
                // the number of siblings
                if (p->child[i]->key == x)
                    return p->child.size() - 1;

                q.push(p->child[i]);
            }
            n--;
        }
    }
}

// Driver program
int main()
{
    // Creating a generic tree as shown in above figure
    Node* root = new Node(50);
    (root->child).push_back(new Node(2));
    (root->child).push_back(new Node(30));
    (root->child).push_back(new Node(14));
    (root->child).push_back(new Node(60));
    (root->child[0]->child).push_back(new Node(15));
    (root->child[0]->child).push_back(new Node(25));
    (root->child[0]->child[1]->child).push_back(new Node(70));
    (root->child[0]->child[1]->child).push_back(new Node(100));
    (root->child[1]->child).push_back(new Node(6));
    (root->child[1]->child).push_back(new Node(1));
    (root->child[2]->child).push_back(new Node(7));
    (root->child[2]->child[0]->child).push_back(new Node(17));
    (root->child[2]->child[0]->child).push_back(new Node(99));
    (root->child[2]->child[0]->child).push_back(new Node(27));
    (root->child[3]->child).push_back(new Node(16));

    // Node whose number of
    // siblings is to be calculated
    int x = 100;

    // Function calling
    cout << numberOfSiblings(root, x) << endl;

    return 0;
}
```

Output:

```
1
```

Time Complexity : O(N^2), where N is the number of nodes in tree.
Auxiliary Space : O(N), where N is the number of nodes in tree.

Source

https://www.geeksforgeeks.org/number-siblings-given-node-n-ary-tree/

Chapter 50

Print Binary Tree levels in sorted order

Print Binary Tree levels in sorted order - GeeksforGeeks

Given a Binary tree, the task is to print its all level in sorted order

Examples:

```
Input :     7
          /    \
         6       5
        / \     / \
       4  3    2   1
Output :
7
5 6
1 2 3 4

Input :     7
          /    \
         16       1
        / \
       4   13
Output :
7
1 16
4 13
```

Here we can use two Priority queue for print in sorted order. We create an empty queue q and two priority queues, current_level and next_level. We use NULL as a separator between two levels. Whenever we encounter NULL in normal level order traversal, we swap current_level and next_level.

```
 // CPP program to print levels in sorted order.
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Binary Tree
void printLevelOrder(Node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty queue for level order traversal
    queue<Node*> q;

    // A priority queue (or min heap) of integers for
    // to store all elements of current level.
    priority_queue<int, vector<int>, greater<int> > current_level;

    // A priority queue (or min heap) of integers for
    // to store all elements of next level.
    priority_queue<int, vector<int>, greater<int> > next_level;

    // push the root for traverse all next level nodes
    q.push(root);

    // for go level by level
    q.push(NULL);

    // push the first node data in previous_level queue
    current_level.push(root->data);

    while (q.empty() == false) {

        // Get top of priority queue
        int data = current_level.top();

        // Get top of queue
        Node* node = q.front();

        // if node == NULL (Means this is boundary
```

```
        // between two levels), swap current_level
        // next_level priority queues.
        if (node == NULL) {
            q.pop();

            // here queue is empty represent
            // no element in the actual
            // queue
            if (q.empty())
                break;

            q.push(NULL);
            cout << "\n";

            // swap next_level to current_level level
            // for print in sorted order
            current_level.swap(next_level);

            continue;
        }

        // print the current_level data
        cout << data << " ";

        q.pop();
        current_level.pop();

        /* Enqueue left child */
        if (node->left != NULL) {
            q.push(node->left);

            // Enqueue left child in next_level queue
            next_level.push(node->left->data);
        }

        /*Enqueue right child */
        if (node->right != NULL) {
            q.push(node->right);

            // Enqueue right child in next_level queue
            next_level.push(node->right->data);
        }
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
```

```
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node* root = newNode(7);
    root->left = newNode(6);
    root->right = newNode(5);
    root->left->left = newNode(4);
    root->left->right = newNode(3);
    root->right->left = newNode(2);
    root->right->right = newNode(1);

    /*     7
          /    \
         6       5
        / \     / \
       4  3     2  1          */

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}
```

Output:

```
Level Order traversal of binary tree is
7
5 6
1 2 3 4
```

Source

https://www.geeksforgeeks.org/print-binary-tree-levels-sorted-order/

Chapter 51

Print Binary Tree levels in sorted order | Set 2 (Using set)

Print Binary Tree levels in sorted order | Set 2 (Using set) - GeeksforGeeks

Given a tree, print the level order traversal in sorted order.

Examples :

```
Input :     7
          /    \
         6       5
        / \     / \
       4  3    2   1
Output :
7
5 6
1 2 3 4

Input :     7
          /    \
         16       1
        / \
       4   13
Output :
7
1 16
4 13
```

We have discussed a priority queue based solution in below post.

Print Binary Tree levels in sorted order | Set 1 (Using Priority Queue)

In this post, a set (which is implemented using balanced binary search tree) based solution is discussed.

Approach :
1. Start level order traversal of tree.
2. Store all the nodes in a set(or any other similar data structures).
3. Print elements of set.

C++

```
 // CPP code to print level order
// traversal in sorted order
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int dat = 0)
        : data(dat), left(nullptr),
          right(nullptr)
    {
    }
};

// Function to print sorted
// level order traversal
void sorted_level_order(Node* root)
{
    queue<Node*> q;
    set<int> s;

    q.push(root);
    q.push(nullptr);

    while (q.empty() == false) {
        Node* tmp = q.front();
        q.pop();

        if (tmp == nullptr) {
            if (s.empty() == true)
                break;
            for (set<int>::iterator it =
                 s.begin();it != s.end(); ++it)
                cout << *it << " ";
            q.push(nullptr);
            s.clear();
        }
```

```
        else {
            s.insert(tmp->data);

            if (tmp->left != nullptr)
                q.push(tmp->left);
            if (tmp->right != nullptr)
                q.push(tmp->right);
        }
    }
}

// Driver code
int main()
{
    Node* root = new Node(7);
    root->left = new Node(6);
    root->right = new Node(5);
    root->left->left = new Node(4);
    root->left->right = new Node(3);
    root->right->left = new Node(2);
    root->right->right = new Node(1);
    sorted_level_order(root);
    return 0;
}
```

Output:

```
7 5 6 1 2 3 4
```

Source

https://www.geeksforgeeks.org/print-binary-tree-levels-sorted-order-2/

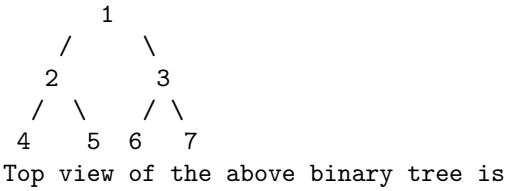
Chapter 52

Print Nodes in Top View of Binary Tree

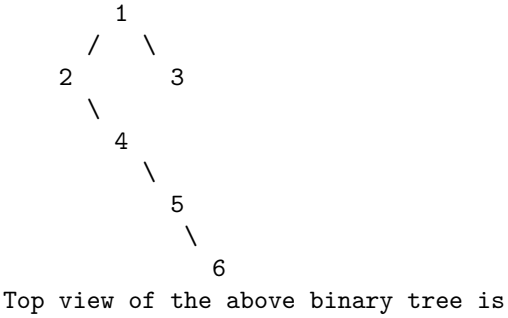
Print Nodes in Top View of Binary Tree - GeeksforGeeks

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.



```
Top view of the above binary tree is
4 2 1 3 7
```



```
Top view of the above binary tree is
2 1 3 6
```

The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to 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.

C++

```
 // C++ program to print top
// view of binary tree
#include <bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node {
    int data;
    struct Node *left, *right;
};

// function should print the topView of
// the binary tree
void topView(struct Node* root)
{
    if (root == NULL)
        return;

    unordered_map<int, int> m;
    queue<pair<Node*, int> > q;

    // push node and horizontal distance to queue
    q.push(make_pair(root, 0));

    while (!q.empty()) {
        pair<Node*, int> p = q.front();
        Node* n = p.first;
        int val = p.second;
        q.pop();

        // if horizontal value is not in the hashmap
        // that means it is the first value with that
        // horizontal distance so print it and store
        // this value in hashmap
        if (m.find(val) == m.end()) {
            m[val] = n->data;
            printf("%d ", n->data);
        }

        if (n->left != NULL)
            q.push(make_pair(n->left, val - 1));
```

```
        if (n->right != NULL)
            q.push(make_pair(n->right, val + 1));
    }
}

// function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->data = key;
    node->left = node->right = NULL;
    return node;
}

// main function
int main()
{
    /* Create following Binary Tree
              1
            /  \
           2    3
            \
             4
              \
               5
                \
                 6*/

    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);

    topView(root);
    return 0;
}

/* This code is contributed by Niteesh Kumar */
```

Java

```
 // Java program to print top view of Binary tree
import java.util.*;

// Class for a tree node
```

```
class TreeNode {
    // Members
    int key;
    TreeNode left, right;

    // Constructor
    public TreeNode(int key)
    {
        this.key = key;
        left = right = null;
    }
}

// A class to represent a queue item. The queue is used to do Level
// order traversal. Every Queue item contains node and horizontal
// distance of node from root
class QItem {
    TreeNode node;
    int hd;
    public QItem(TreeNode n, int h)
    {
        node = n;
        hd = h;
    }
}

// Class for a Binary Tree
class Tree {
    TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // This method prints nodes in top view of binary tree
    public void printTopView()
    {
        // base case
        if (root == null) {
            return;
        }

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Create a queue and add root to it
        Queue<QItem> Q = new LinkedList<QItem>();
        Q.add(new QItem(root, 0)); // Horizontal distance of root is 0
```

```
        // Standard BFS or level order traversal loop
        while (!Q.isEmpty()) {
            // Remove the front item and get its details
            QItem qi = Q.remove();
            int hd = qi.hd;
            TreeNode n = qi.node;

            // If this is the first node at its horizontal distance,
            // then this node is in top view
            if (!set.contains(hd)) {
                set.add(hd);
                System.out.print(n.key + " ");
            }

            // Enqueue left and right children of current node
            if (n.left != null)
                Q.add(new QItem(n.left, hd - 1));
            if (n.right != null)
                Q.add(new QItem(n.right, hd + 1));
        }
    }
}

// Driver class to test above methods
public class Main {
    public static void main(String[] args)
    {
        /* Create following Binary Tree
             1
           /  \
          2    3
           \
            4
             \
              5
               \
                6*/
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);
        root.left.right.right.right = new TreeNode(6);
        Tree t = new Tree(root);
        System.out.println("Following are nodes in top view of Binary Tree");
        t.printTopView();
    }
```

```
}
```

Output:

```
Following are nodes in top view of Binary Tree
1 2 3 6
```

Time Complexity of the above implementation is O(n) where n is number of nodes in given binary tree. The assumption here is that add() and contains() methods of HashSet work in O(1) time.

This article is contributed by **Rohan**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Aarsee

Source

https://www.geeksforgeeks.org/print-nodes-top-view-binary-tree/

Chapter 53

Priority Queue in Python

Priority Queue in Python - GeeksforGeeks

Priority Queue is an extension of the queue with following properties.
1) An element with high priority is dequeued before an element with low priority.
2) If two elements have the same priority, they are served according to their order in the queue.

Below is **simple implementation** of priority queue.

```
 # A simple implementation of Priority Queue
# using Queue.
class PriorityQueue(object):
    def __init__(self):
        self.queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    # for checking if the queue is empty
    def isEmpty(self):
        return len(self.queue) == []

    # for inserting an element in the queue
    def insert(self, data):
        self.queue.append(data)

    # for popping an element based on Priority
    def delete(self):
        try:
            max = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max]:
                    max = i
```

```
            item = self.queue[max]
            del self.queue[max]
            return item
        except IndexError:
            print()
            exit()

if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue)
    while not myQueue.isEmpty():
        print(myQueue.delete())
```

Output:

```
12 1 14 7
14
12
7
1
()
```

Note that the time complexity of delete is O(n) in above code.

A **better implementation** is to use Binary Heap which is typically used to implement priority queue. Note that Python provides **heapq** in library also.

Source

https://www.geeksforgeeks.org/priority-queue-in-python/

Chapter 54

Priority Queue using Linked List

Priority Queue using Linked List - GeeksforGeeks

Implement Priority Queue using Linked Lists.

- push(): This function is used to insert a new data into the queue.
- pop(): This function removes the element with the highest priority form the queue.
- peek() / top(): This function is used to get the highest priority element in the queue without removing it from the queue.

Priority Queues can be implemented using common data structures like arrays, linked-lists, heaps and binary trees.

Prerequisites :
Linked Lists, Priority Queues

The list is so created so that the highest priority element is always at the head of the list. The list is arranged in descending order of elements based on their priority. This allow us to remove the highest priority element in O(1) time. To insert an element we must traverse the list and find the proper position to insert the node so that the overall order of the priority queue is maintained. This makes the push() operation takes O(N) time. The pop() and peek() operations are performed in constant time.

Algorithm :
PUSH(HEAD, DATA, PRIORITY)
Step 1: Create new node with DATA and PRIORITY
Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.
Step 3: NEW -> NEXT = HEAD
Step 4: HEAD = NEW
Step 5: Set TEMP to head of the list

Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY
Step 7: TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: NEW -> NEXT = TEMP -> NEXT
Step 9: TEMP -> NEXT = NEW
Step 10: End

POP(HEAD)
Step 2: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.
Step 3: Free the node at the head of the list
Step 4: End

PEEK(HEAD):
Step 1: Return HEAD -> DATA
Step 2: End

Below is the implementation of the algorithm :

C

```
 // C code to implement Priority Queue
// using Linked List
#include <stdio.h>
#include <stdlib.h>

// Node
typedef struct node {
    int data;

    // Lower values indicate higher priority
    int priority;

    struct node* next;

} Node;

// Function to Create A New Node
Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;

    return temp;
}

// Return the value at head
int peek(Node** head)
```

```
{
    return (*head)->data;
}

// Removes the element with the
// highest priority form the list
void pop(Node** head)
{
    Node* temp = *head;
    (*head) = (*head)->next;
    free(temp);
}

// Function to push according to priority
void push(Node** head, int d, int p)
{
    Node* start = (*head);

    // Create new Node
    Node* temp = newNode(d, p);

    // Special Case: The head of list has lesser
    // priority than new node. So insert new
    // node before head node and change head node.
    if ((*head)->priority > p) {

        // Insert New Node before head
        temp->next = *head;
        (*head) = temp;
    }
    else {

        // Traverse the list and find a
        // position to insert new node
        while (start->next != NULL &&
               start->next->priority < p) {
            start = start->next;
        }

        // Either at the ends of the list
        // or at required position
        temp->next = start->next;
        start->next = temp;
    }
}

// Function to check is list is empty
int isEmpty(Node** head)
```

```
{
    return (*head) == NULL;
}

// Driver code
int main()
{
    // Create a Priority Queue
    // 7->4->5->6
    Node* pq = newNode(4, 1);
    push(&pq, 5, 2);
    push(&pq, 6, 3);
    push(&pq, 7, 0);

    while (!isEmpty(&pq)) {
        printf("%d ", peek(&pq));
        pop(&pq);
    }

    return 0;
}
```

Output:

```
7 4 5 6
```

Time Complexities and Comparison with Binary Heap:

```
               peek()    push()    pop()
-----------------------------------------
Linked List |   O(1)      O(n)      O(1)
            |
Binary Heap |   O(1)    O(Log n)   O(Log n)
```

Source

https://www.geeksforgeeks.org/priority-queue-using-linked-list/

Chapter 55

Priority Queue using doubly linked list

Priority Queue using doubly linked list - GeeksforGeeks

Given Nodes with their priority, implement a priority queue using doubly linked list.

Prerequisite : Priority Queue

- push(): This function is used to insert a new data into the queue.
- pop(): This function removes the element with the lowest priority value form the queue.
- peek() / top(): This function is used to get the lowest priority element in the queue without removing it from the queue.

Approach :
1. Create a doubly linked list having fields info(hold the information of the Node), priority(hold the priority of the Node), prev(point to previous Node), next(point to next Node).
2. Insert the element and priority in the Node.
3. Arrange the Nodes in the increasing order of the priority.

Below is the implementation of above steps :

C++

```
 // CPP code to implement priority
// queue using doubly linked list
#include <bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node {
    int info;
```

```
    int priority;
    struct Node *prev, *next;
};

// Function to insert a new Node
void push(Node** fr, Node** rr, int n, int p)
{
    Node* news = (Node*)malloc(sizeof(Node));
    news->info = n;
    news->priority = p;

    // If linked list is empty
    if (*fr == NULL) {
        *fr = news;
        *rr = news;
        news->next = NULL;
    }
    else {
        // If p is less than or equal front
        // node's priority, then insert at
        // the front.
        if (p <= (*fr)->priority) {
            news->next = *fr;
            (*fr)->prev = news->next;
            *fr = news;
        }

        // If p is more rear node's priority,
        // then insert after the rear.
        else if (p > (*rr)->priority) {
            news->next = NULL;
            (*rr)->next = news;
            news->prev = (*rr)->next;
            *rr = news;
        }

        // Handle other cases
        else {

            // Find position where we need to
            // insert.
            Node* start = (*fr)->next;
            while (start->priority > p)
                start = start->next;
            (start->prev)->next = news;
            news->next = start->prev;
            news->prev = (start->prev)->next;
            start->prev = news->next;
```

```
        }
    }
}

// Return the value at rear
int peek(Node *fr)
{
    return fr->info;
}

bool isEmpty(Node *fr)
{
    return (fr == NULL);
}

// Removes the element with the
// least priority value form the list
int pop(Node** fr, Node** rr)
{
    Node* temp = *fr;
    int res = temp->info;
    (*fr) = (*fr)->next;
    free(temp);
    if (*fr == NULL)
       *rr = NULL;
    return res;
}

// Diver code
int main()
{
    Node *front = NULL, *rear = NULL;
    push(&front, &rear, 2, 3);
    push(&front, &rear, 3, 4);
    push(&front, &rear, 4, 5);
    push(&front, &rear, 5, 6);
    push(&front, &rear, 6, 7);
    push(&front, &rear, 1, 2);

    cout << pop(&front, &rear) << endl;
    cout << peek(front);

    return 0;
}
```

Output:

```
1
2
```

Related Article :
Priority Queue using Singly Linked List

Time Complexities and Comparison with Binary Heap:

```
               peek()    push()    pop()
----------------------------------------
Linked List |   O(1)      O(n)      O(1)
            |
Binary Heap |   O(1)    O(Log n)   O(Log n)
```

Source

https://www.geeksforgeeks.org/priority-queue-using-doubly-linked-list/

Chapter 56

Priority Queue | Set 1 (Introduction)

Priority Queue | Set 1 (Introduction) - GeeksforGeeks

Priority Queue is an extension of queuewith following properties.
1) Every item has a priority associated with it.
2) An element with high priority is dequeued before an element with low priority.
3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
struct item {
   int item;
   int priority;
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items.

Using Heaps:
Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in O(1) time, insert() can be implemented in O(Logn) time and deleteHighestPriority() can also be implemented in O(Logn) time.
With Fibonacci heap, insert() and getHighestPriority() can be implemented in O(1) amortized time and deleteHighestPriority() can be implemented in O(Logn) amortized time.

Applications of Priority Queue:
1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

A priority queue is implemented using Heap. Please refer below articles for our own implementation and library implementations.

1. Binary Heap (The most common implementation of priority queue)
2. Priority Queue in C++.
3. Priority Queue in Java.
4. Priority Queue in Python.
5. Priority Queue in JavaScript.

Useful Links :

1. Recent articles on Priority Queue!
2. Applications of Priority Queue.

References:
http://en.wikipedia.org/wiki/Priority_queue

Source

https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

Chapter 57

Program for Page Replacement Algorithms | Set 2 (FIFO)

Program for Page Replacement Algorithms | Set 2 (FIFO) - GeeksforGeeks

Prerequisite : Page Replacement Algorithms

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

First In First Out (FIFO) page replacement algorithm –
This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example -1. Consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 **Page Faults.**
when 3 comes, it is already in memory so —> 0 Page Faults.
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 **Page Fault.**
Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 **Page Fault.**

So total page faults = **5**.

Example -2. Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1.

Using FIFO page replacement algorithm –

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	0	4	4			4	4	2	
	2	2	2	2	0		hit	0	0	0	
		1	1	1	1	hit		3	3	3	
			6	6	6			6	1	1	hit

So, total number of page faults = 9.

Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.

Implementation – Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

```
1- Start traversing the pages.
 i) If set holds less pages than capacity.
   a) Insert page into the set one by one until
      the size  of set reaches capacity or all
      page requests are processed.
   b) Simultaneously maintain the pages in the
      queue to perform FIFO.
   c) Increment page fault
 ii) Else
   If current page is present in set, do nothing.
   Else
     a) Remove the first page from the queue
        as it was the first to be entered in
        the memory
     b) Replace the first page in the queue with
        the current page in the string.
     c) Store current page in the queue.
     d) Increment page faults.

2. Return page faults.
```

C++

```
 // C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
    // To represent set of current pages. We use
    // an unordered_set so that we quickly check
    // if a page is present in set or not
    unordered_set<int> s;
```

```
// To store the pages in FIFO manner
queue<int> indexes;

// Start from initial page
int page_faults = 0;
for (int i=0; i<n; i++)
{
    // Check if the set can hold more pages
    if (s.size() < capacity)
    {
        // Insert it into set if not present
        // already which represents page fault
        if (s.find(pages[i])==s.end())
        {
            s.insert(pages[i]);

            // increment page fault
            page_faults++;

            // Push the current page into the queue
            indexes.push(pages[i]);
        }
    }

    // If the set is full then need to perform FIFO
    // i.e. remove the first page of the queue from
    // set and queue both and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (s.find(pages[i]) == s.end())
        {
            //Pop the first page from the queue
            int val = indexes.front();

            indexes.pop();

            // Remove the indexes page
            s.erase(val);

            // insert the current page
            s.insert(pages[i]);

            // push the current page into
            // the queue
            indexes.push(pages[i]);
```

```
                // Increment page faults
                page_faults++;
            }
        }
    }

    return page_faults;
}

// Driver code
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                   2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

Java

```
 // Java implementation of FIFO page replacement
// in Operating Systems.

import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;


class Test
{
    // Method to find page faults using FIFO
    static int pageFaults(int pages[], int n, int capacity)
    {
        // To represent set of current pages. We use
        // an unordered_set so that we quickly check
        // if a page is present in set or not
        HashSet<Integer> s = new HashSet<>(capacity);

        // To store the pages in FIFO manner
        Queue<Integer> indexes = new LinkedList<>() ;

        // Start from initial page
        int page_faults = 0;
        for (int i=0; i<n; i++)
        {
```

```
        // Check if the set can hold more pages
        if (s.size() < capacity)
        {
            // Insert it into set if not present
            // already which represents page fault
            if (!s.contains(pages[i]))
            {
                s.add(pages[i]);

                // increment page fault
                page_faults++;

                // Push the current page into the queue
                indexes.add(pages[i]);
            }
        }

        // If the set is full then need to perform FIFO
        // i.e. remove the first page of the queue from
        // set and queue both and insert the current page
        else
        {
            // Check if current page is not already
            // present in the set
            if (!s.contains(pages[i]))
            {
                //Pop the first page from the queue
                int val = indexes.peek();

                indexes.poll();

                // Remove the indexes page
                s.remove(val);

                // insert the current page
                s.add(pages[i]);

                // push the current page into
                // the queue
                indexes.add(pages[i]);

                // Increment page faults
                page_faults++;
            }
        }
    }

    return page_faults;
```

```
    }

    // Driver method
    public static void main(String args[])
    {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                       2, 3, 0, 3, 2};

        int capacity = 4;
        System.out.println(pageFaults(pages, pages.length, capacity));
    }
}
// This code is contributed by Gaurav Miglani
```

Output:

```
7
```

Note – We can also find the number of page hits. Just have to maintain a separate count. If the current page is already in the memory then that must be count as Page-hit.

Belady's anomaly –
Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Improved By : Villan

Source

https://www.geeksforgeeks.org/program-page-replacement-algorithms-set-2-fifo/

Chapter 58

Queue Interface In Java

Queue Interface In Java - GeeksforGeeks

The Queue interface is available in java.util package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.It is to be noted that both the implementations are not thread safe. *PriorityBlockingQueue* is one alternative implementation if thread safe implementation is needed. Few important characteristics of Queue are:

- The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
- The Java Queue supports all methods of Collection interface including insertion, deletion etc.
- LinkedList, ArrayBlockingQueue and PriorityQueue are the most frequently used implementations.
- If any null operation is performed on BlockingQueues, NullPointerException is thrown.
- BlockingQueues have thread-safe implementations.
- The Queues which are available in java.util package are Unbounded Queues
- The Queues which are available in java.util.concurrent package are the Bounded Queues.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

Methods in Queue:

1. **add()-** This method is used to add elements at the tail of queue. More specifically, at the last of linkedlist if it is used, or according to the priority in case of priority queue implementation.

2. **peek()-** This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
3. **element()-** This method is similar to peek(). It throws *NoSuchElementException* when the queue is empty.
4. **remove()-** This method removes and returns the head of the queue. It throws *NoSuchElementException* when the queue is impty.
5. **poll()-** This method removes and returns the head of the queue. It returns null if the queue is empty.

OPERATION	THROWS EXCEPTION	RETURN VALLUES
Insert	add(element)	offer(element)
Remove	remove()	poll()
Examine	element()	peek()

Since it is a subtype of Collections class, it inherits all the methods of it namely *size(), isEmpty(), contains() etc.*
Below is a simple Java program to demonstrate these methods:

```
 // Java orogram to demonstrate working of Queue
// interface in Java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample
{
  public static void main(String[] args)
  {
    Queue<Integer> q = new LinkedList<>();

    // Adds elements {0, 1, 2, 3, 4} to queue
    for (int i=0; i<5; i++)
     q.add(i);

    // Display contents of the queue.
    System.out.println("Elements of queue-"+q);

    // To remove the head of queue.
    int removedele = q.remove();
    System.out.println("removed element-" + removedele);

    System.out.println(q);

    // To view the head of queue
    int head = q.peek();
    System.out.println("head of queue-" + head);
```

```
    // Rest all methods of collection interface,
    // Like size and contains can be used with this
    // implementation.
    int size = q.size();
    System.out.println("Size of queue-" + size);
  }
}
```

Output:

```
Elements of queue-[0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

Applications of queue data structure can be found here

Improved By : Chinmoy Lenka

Source

https://www.geeksforgeeks.org/queue-interface-java/

Chapter 59

Queue based approach for first non-repeating character in a stream

Queue based approach for first non-repeating character in a stream - GeeksforGeeks

Given a stream of characters and we have to find first non repeating character each time a character is inserted to the stream.

Examples:

```
Input  : a a b c
Output : a -1 b b

Input  : a a c
Output : a -1 c
```

We have already discussed a Doubly linked list based approach in the previous post.

Approach-

1. Create a count array of size 26(assuming only lower case characters are present) and initialize it with zero.
2. Create a queue of char datatype.
3. Store each character in queue and increase its frequency in the hash array.
4. For every character of stream, we check front of the queue.
5. If the frequency of character at the front of queue is one, then that will be the first non repeating character.
6. Else if frequency is more than 1, then we pop that element.
7. If queue became empty that means there are no non repeating character so we will print -1.

C++

```
 // C++ program for a Queue based approach to find first non-repeating
// character
#include <bits/stdc++.h>
using namespace std;
const int CHAR_MAX = 26;

// function to find first non repeating
// character of sa stream
void firstnonrepeating(char str[])
{
    queue<char> q;
    int charCount[CHAR_MAX] = { 0 };

    // traverse whole stream
    for (int i = 0; str[i]; i++) {

        // push each character in queue
        q.push(str[i]);

        // increment the frequency count
        charCount[str[i]-'a']++;

        // check for the non pepeating character
        while (!q.empty())
        {
            if (charCount[q.front()-'a'] > 1)
                q.pop();
            else
            {
                cout << q.front() << " ";
                break;
            }
        }

        if (q.empty())
            cout << -1 << " ";
    }
    cout << endl;
}

// Driver function
int main()
{
    char str[] = "aabc";
    firstnonrepeating(str);
    return 0;
```

```
}
```

Java

```
 //Java Program for a Queue based approach to find first non-repeating
//character

import java.util.LinkedList;
import java.util.Queue;

public class NonReapatingCQueue
{
     final static int CHAR_MAX = 26;

    // function to find first non repeating
    // character of stream
     static void firstNonRepeating(String str)
     {
         //count array of size 26(assuming only lower case characters are present)
         int[] charCount = new int[CHAR_MAX];

         //Queue to store Characters
         Queue<Character> q = new LinkedList<Character>();

        // traverse whole stream
         for(int i=0; i<str.length(); i++)
         {
             char c = str.charAt(i);

            // push each character in queue
             q.add(c);

            // increment the frequency count
             charCount++;

            // check for the non repeating character
             while(!q.isEmpty())
             {
                 if(charCount[q.peek() - 'a'] > 1)
                     q.remove();
                 else
                 {
                     System.out.print(q.peek() + " ");
                     break;
                 }
             }
             if(q.isEmpty())
                 System.out.print(-1 + " ");
```

```
            }
            System.out.println();
        }

    // Driver function
    public static void main(String[] args)
    {
        String str = "aabc";
        firstNonRepeating(str);
    }

}
//This code is Contributed by Sumit Ghosh
```

Output:

```
a -1 b b
```

Source

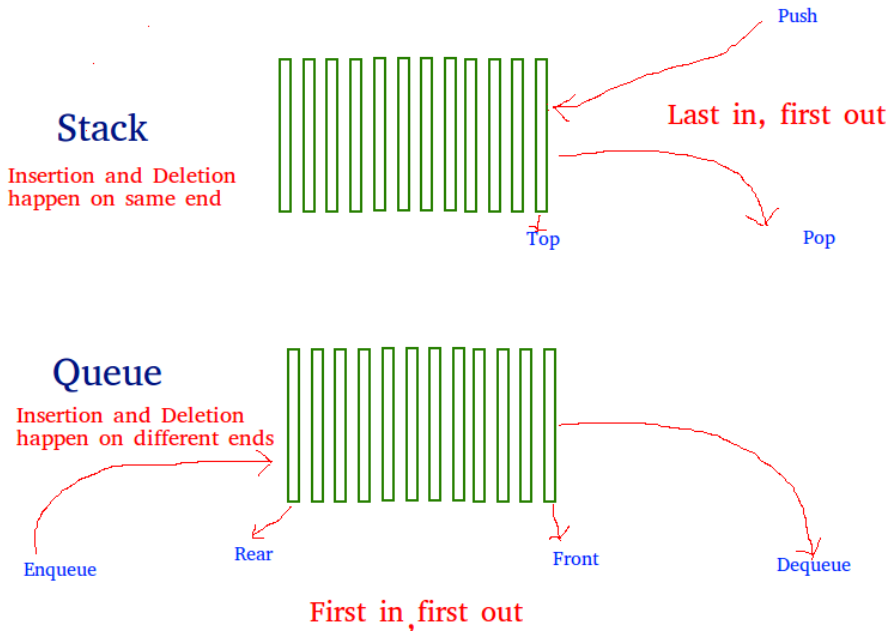
https://www.geeksforgeeks.org/queue-based-approach-for-first-non-repeating-character-in-a-stream/

Chapter 60

Queue using Stacks

Queue using Stacks - GeeksforGeeks

The problem is opposite of this post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly) This method makes sure that 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)
  1) While stack1 is not empty, push everything from stack1 to stack2.
  2) Push x to stack1 (assuming size of stacks is unlimited).
  3) Push everything back to stack1.
Here time complexity will be O(n)

deQueue(q)
  1) If stack1 is empty then error
  2) Pop an item from stack1 and return it
Here time complexity will be O(1)
```

C++

```
 // CPP program to implement Queue using
// two stacks with costly enQueue()
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    void enQueue(int x)
    {
        // Move all elements from s1 to s2
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }

        // Push item into s1
        s1.push(x);

        // Push everything back to s1
        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }

    // Dequeue an item from the queue
    int deQueue()
    {
        // if first stack is empty
        if (s1.empty()) {
            cout << "Q is Empty";
            exit(0);
        }
```

```
        // Return top of s1
        int x = s1.top();
        s1.pop();
        return x;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';

    return 0;
}
```

Method 2 (By making deQueue operation costly) In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q,  x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
Here time complexity will be O(n)
```

Method 2 is definitely better than method 1.
Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.
Implementation of method 2:

C++

```
 // CPP program to implement Queue using
// two stacks with costly deQueue()
```

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

struct Queue {
    stack<int> s1, s2;

    // Enqueue an item to the queue
    void enQueue(int x)
    {
        // Push item into the first stack
        s1.push(x);
    }

    // Dequeue an item from the queue
    int deQueue()
    {
        // if both stacks are empty
        if (s1.empty() && s2.empty()) {
            cout << "Q is empty";
            exit(0);
        }

        // if s2 is empty, move
        // elements from s1
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }

        // return the top item from s2
        int x = s2.top();
        s2.pop();
        return x;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
```

```
    cout << q.deQueue() << '\n';

    return 0;
}
```

C

```
 /* C Program to implement a queue using two stacks */
#include <stdio.h>
#include <stdlib.h>

/* structure of a stack node */
struct sNode {
    int data;
    struct sNode* next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue {
    struct sNode* stack1;
    struct sNode* stack2;
};

/* Function to enqueue an item to queue */
void enQueue(struct queue* q, int x)
{
    push(&q->stack1, x);
}

/* Function to deQueue an item from queue */
int deQueue(struct queue* q)
{
    int x;

    /* If both stacks are empty then error */
    if (q->stack1 == NULL && q->stack2 == NULL) {
        printf("Q is empty");
        getchar();
        exit(0);
    }

    /* Move elements from stack1 to stack 2 only if
```

```
       stack2 is empty */
    if (q->stack2 == NULL) {
        while (q->stack1 != NULL) {
            x = pop(&q->stack1);
            push(&q->stack2, x);
        }
    }

    x = pop(&q->stack2);
    return x;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node = (struct sNode*)malloc(sizeof(struct sNode));
    if (new_node == NULL) {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode* top;

    /*If stack is empty then error */
    if (*top_ref == NULL) {
        printf("Stack underflow \n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
```

```
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test anove functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue* q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enQueue(q, 1);
    enQueue(q, 2);
    enQueue(q, 3);

    /* Dequeue items */
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));

    return 0;
}
```

Java

```
 /* Java Program to implement a queue using two stacks */
// Note that Stack class is used for Stack implementation

import java.util.Stack;

public class GFG {
    /* class of queue having two stacks */
    static class Queue {
        Stack<Integer> stack1;
        Stack<Integer> stack2;
    }

    /* Function to push an item to stack*/
    static void push(Stack<Integer> top_ref, int new_data)
    {
        // Push the data onto the stack
        top_ref.push(new_data);
    }

    /* Function to pop an item from stack*/
    static int pop(Stack<Integer> top_ref)
```

```
{
    /*If stack is empty then error */
    if (top_ref.isEmpty()) {
        System.out.println("Stack Underflow");
        System.exit(0);
    }

    // pop the data from the stack
    return top_ref.pop();
}

// Function to enqueue an item to the queue
static void enQueue(Queue q, int x)
{
    push(q.stack1, x);
}

/* Function to deQueue an item from queue */
static int deQueue(Queue q)
{
    int x;

    /* If both stacks are empty then error */
    if (q.stack1.isEmpty() && q.stack2.isEmpty()) {
        System.out.println("Q is empty");
        System.exit(0);
    }

    /* Move elements from stack1 to stack 2 only if
    stack2 is empty */
    if (q.stack2.isEmpty()) {
        while (!q.stack1.isEmpty()) {
            x = pop(q.stack1);
            push(q.stack2, x);
        }
    }
    x = pop(q.stack2);
    return x;
}

/* Driver function to test above functions */
public static void main(String args[])
{
    /* Create a queue with items 1 2 3*/
    Queue q = new Queue();
    q.stack1 = new Stack<>();
    q.stack2 = new Stack<>();
    enQueue(q, 1);
```

```
        enQueue(q, 2);
        enQueue(q, 3);

        /* Dequeue items */
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
        System.out.println(deQueue(q) + " ");
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
1 2 3
```

Queue can also be implemented using one user stack and one Function Call Stack. Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enQueue(x)
  1) Push x to stack1.

deQueue:
  1) If stack1 is empty then error.
  2) If stack1 has only one element then return it.
  3) Recursively pop everything from the stack1, store the popped item
     in a variable res,  push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in deQueue() and all other items are pushed back in step

3. Implementation of method 2 using Function Call Stack:

C++

```
 // CPP program to implement Queue using
// one stack and recursive call stack.
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s;

    // Enqueue an item to the queue
    void enQueue(int x)
    {
```

```
        s.push(x);
    }

    // Dequeue an item from the queue
    int deQueue()
    {
        if (s.empty()) {
            cout << "Q is empty";
            exit(0);
        }

        // pop an item from the stack
        int x = s.top();
        s.pop();

        // if stack becomes empty, return
        // the popped item
        if (s.empty())
            return x;

        // recursive call
        int item = deQueue();

        // push popped item back to the stack
        s.push(x);

        // return the result of deQueue() call
        return item;
    }
};

// Driver code
int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';

    return 0;
}
```

C

```
 /* Program to implement a queue using one user defined stack
and one Function Call Stack */
#include <stdio.h>
#include <stdlib.h>

/* structure of a stack node */
struct sNode {
    int data;
    struct sNode* next;
};

/* structure of queue having two stacks */
struct queue {
    struct sNode* stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enQueue(struct queue* q, int x)
{
    push(&q->stack1, x);
}

/* Function to deQueue an item from queue */
int deQueue(struct queue* q)
{
    int x, res;

    /* If both stacks are empty then error */
    if (q->stack1 == NULL) {
        printf("Q is empty");
        getchar();
        exit(0);
    }
    else if (q->stack1->next == NULL) {
        return pop(&q->stack1);
    }
    else {
        /* pop an item from the stack1 */
        x = pop(&q->stack1);

        /* store the last deQueued item */
        res = deQueue(q);
```

```
        /* push everything back to stack1 */
        push(&q->stack1, x);
        return res;
    }
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node = (struct sNode*)malloc(sizeof(struct sNode));

    if (new_node == NULL) {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode* top;

    /*If stack is empty then error */
    if (*top_ref == NULL) {
        printf("Stack underflow \n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
```

```
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue* q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;

    enQueue(q, 1);
    enQueue(q, 2);
    enQueue(q, 3);

    /* Dequeue items */
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));

    return 0;
}
```

Java

```
 // Java Program to implement a queue using one stack

import java.util.Stack;

public class QOneStack {
    // class of queue having two stacks
    static class Queue {
        Stack<Integer> stack1;
    }

    /* Function to push an item to stack*/
    static void push(Stack<Integer> top_ref, int new_data)
    {
        /* put in the data */
        top_ref.push(new_data);
    }

    /* Function to pop an item from stack*/
    static int pop(Stack<Integer> top_ref)
    {
        /*If stack is empty then error */
        if (top_ref == null) {
            System.out.println("Stack Underflow");
            System.exit(0);
        }
```

```
    // return element from stack
    return top_ref.pop();
}

/* Function to enqueue an item to queue */
static void enQueue(Queue q, int x)
{
    push(q.stack1, x);
}

/* Function to deQueue an item from queue */
static int deQueue(Queue q)
{
    int x, res = 0;
    /* If the stacks is empty then error */
    if (q.stack1.isEmpty()) {
        System.out.println("Q is Empty");
        System.exit(0);
    }
    // Check if it is a last element of stack
    else if (q.stack1.size() == 1) {
        return pop(q.stack1);
    }
    else {

        /* pop an item from the stack1 */
        x = pop(q.stack1);

        /* store the last deQueued item */
        res = deQueue(q);

        /* push everything back to stack1 */
        push(q.stack1, x);
        return res;
    }
    return 0;
}

/* Driver function to test above functions */
public static void main(String[] args)
{
    /* Create a queue with items 1 2 3*/
    Queue q = new Queue();
    q.stack1 = new Stack<>();

    enQueue(q, 1);
    enQueue(q, 2);
    enQueue(q, 3);
```

```
        /* Dequeue items */
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
        System.out.print(deQueue(q) + " ");
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
1 2 3
```

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

Improved By : iamsdhar, ParulShandilya

Source

https://www.geeksforgeeks.org/queue-using-stacks/

Chapter 61

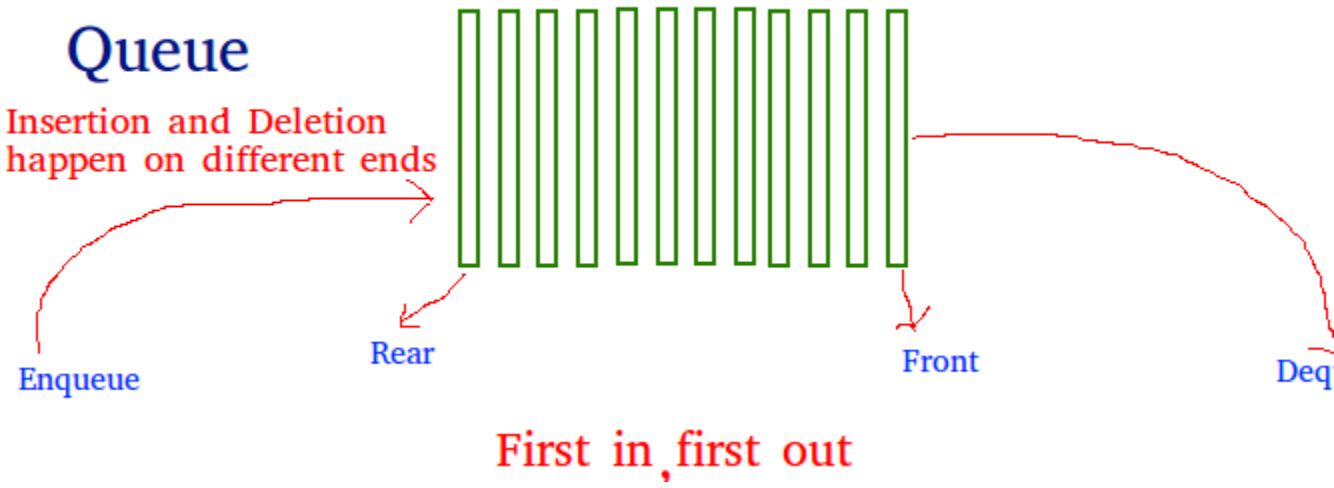
Queue | Set 1 (Introduction and Array Implementation)

Queue | Set 1 (Introduction and Array Implementation) - GeeksforGeeks

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:
Mainly the following four basic operations are performed on queue:

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



Applications of Queue:
Queue is used when things don't have to be processed immediatly, but have to be processed in **F**irst **I**n**F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See this for more detailed applications of Queue and Stack.

Array implementation Of Queue
For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner (See thisfor details)

C

```
 // C program for array implementation of queue
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a queue
struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};
```

```
// function to create a queue of given capacity.
// It initializes size of queue as 0
struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;  // This is important, see the enqueue
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}

// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue)
{  return (queue->size == queue->capacity);  }

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

// Function to add an item to the queue.
// It changes rear and size
void enqueue(struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue.
// It changes front and size
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

// Function to get front of queue
int front(struct Queue* queue)
{
```

```
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

// Driver program to test above functions./
int main()
{
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("%d dequeued from queue\n\n", dequeue(queue));

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}
```

Java

```
 // Java program for array implementation of queue

// A class to represent a queue
class Queue
{
    int front, rear, size;
    int  capacity;
    int array[];

    public Queue(int capacity) {
         this.capacity = capacity;
         front = this.size = 0;
         rear = capacity - 1;
         array = new int[this.capacity];
```

```
}

// Queue is full when size becomes equal to
// the capacity
boolean isFull(Queue queue)
{  return (queue.size == queue.capacity);
}

// Queue is empty when size is 0
boolean isEmpty(Queue queue)
{  return (queue.size == 0); }

// Method to add an item to the queue.
// It changes rear and size
void enqueue( int item)
{
    if (isFull(this))
        return;
    this.rear = (this.rear + 1)%this.capacity;
    this.array[this.rear] = item;
    this.size = this.size + 1;
    System.out.println(item+ " enqueued to queue");
}

// Method to remove an item from queue.
// It changes front and size
int dequeue()
{
    if (isEmpty(this))
        return Integer.MIN_VALUE;

    int item = this.array[this.front];
    this.front = (this.front + 1)%this.capacity;
    this.size = this.size - 1;
    return item;
}

// Method to get front of queue
int front()
{
    if (isEmpty(this))
        return Integer.MIN_VALUE;

    return this.array[this.front];
}

// Method to get rear of queue
int rear()
```

```
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        return this.array[this.rear];
    }
}


// Driver class
public class Test
{
    public static void main(String[] args)
    {
        Queue queue = new Queue(1000);

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.enqueue(40);

        System.out.println(queue.dequeue() +
                    " dequeued from queue\n");

        System.out.println("Front item is " +
                               queue.front());

        System.out.println("Rear item is " +
                                queue.rear());
    }
}

// This code is contributed by Gaurav Miglani
```

Python3

```
 # Python3 program for array implementation of queue

# Class Queue to represent a queue
class Queue:

    # __init__ function
    def __init__(self, capacity):
        self.front = self.size = 0
        self.rear = capacity -1
        self.Q = [None]*capacity
        self.capacity = capacity
```

```
    # Queue is full when size becomes
    # equal to the capacity
    def isFull(self):
        return self.size == self.capacity

    # Queue is empty when size is 0
    def isEmpty(self):
        return self.size == 0

    # Function to add an item to the queue.
    # It changes rear and size
    def EnQueue(self, item):
        if self.isFull():
            print("Full")
            return
        self.rear = (self.rear + 1) % (self.capacity)
        self.Q[self.rear] = item
        self.size = self.size + 1
        print("%s enqueued to queue"  %str(item))

    # Function to remove an item from queue.
    # It changes front and size
    def DeQueue(self):
        if self.isEmpty():
            print("Empty")
            return

        print("%s dequeued from queue" %str(self.Q[self.front]))
        self.front = (self.front + 1) % (self.capacity)
        self.size = self.size -1

    # Function to get front of queue
    def que_front(self):
        if self.isEmpty():
            print("Queue is empty")

        print("Front item is", self.Q[self.front])

    # Function to get rear of queue
    def que_rear(self):
        if self.isEmpty():
            print("Queue is empty")
        print("Rear item is",  self.Q[self.rear])


# Driver Code
if __name__ == '__main__':
```

```
    queue = Queue(30)
    queue.EnQueue(10)
    queue.EnQueue(20)
    queue.EnQueue(30)
    queue.EnQueue(40)
    queue.DeQueue()
    queue.que_front()
    queue.que_rear()
```

C#

```
 // C# program for array implementation of queue
using System;

namespace GeeksForGeeks
{
    // A class to represent a linearqueue
    class Queue
    {
        private int []ele;
        private int front;
        private int rear;
        private int max;

        public Queue(int size)
        {
            ele = new int[size];
            front = 0 ;
            rear = -1;
            max = size;
        }

        // Function to add an item to the queue.
        // It changes rear and size
        public void enqueue(int item)
        {
            if (rear == max-1)
            {
                Console.WriteLine("Queue Overflow");
                return;
            }
            else
            {
                ele[++rear] = item;
            }

        }
```

```
    // Function to remove an item from queue.
    // It changes front and size
    public int dequeue()
    {
        if(front == rear + 1)
        {
            Console.WriteLine("Queue is Empty");
            return -1;
        }
        else
        {
            Console.WriteLine( ele[front]+" dequeued from queue");
            int p = ele[front++];
            Console.WriteLine();
            Console.WriteLine("Front item is {0}",ele[front]);
            Console.WriteLine("Rear item is {0} ",ele[rear]);
    return p;
        }

    }

    // Function to print queue.
    public void printQueue()
    {
        if (front == rear + 1)
        {
            Console.WriteLine("Queue is Empty");
            return;
        }
        else
        {
            for (int i = front; i <= rear; i++)
            {
                Console.WriteLine(ele[i]+ " enqueued to queue" );
            }
        }

    }
}

// Driver code
class Program
{
    static void Main()
    {
        Queue Q = new Queue(5);

        Q.enqueue(10);
```

```
            Q.enqueue(20);
            Q.enqueue(30);
            Q.enqueue(40);
            Q.printQueue();
            Q.dequeue();


        }
    }
}
```

Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40
```

Time Complexity: Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Linked list implementation is easier, it is discussed here: Queue | Set 2 (Linked List Implementation)

Improved By : SoumikMondal

Source

https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/

Chapter 62

Queue | Set 2 (Linked List Implementation)

Queue | Set 2 (Linked List Implementation) - GeeksforGeeks

In the previous post, we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.

deQueue() This operation removes the front node and moves *front* to the next node.

C

```
 // A C program to demonstrate linked list based implementation of queue
#include <stdlib.h>
#include <stdio.h>

// A linked list (LL) node to store a queue entry
struct QNode
{
    int key;
    struct QNode *next;
};

// The queue, front stores the front node of LL and rear stores ths
// last node of LL
struct Queue
{
    struct QNode *front, *rear;
```

```
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
    struct QNode *temp = (struct QNode*)malloc(sizeof(struct QNode));
    temp->key = k;
    temp->next = NULL;
    return temp;
}

// A utility function to create an empty queue
struct Queue *createQueue()
{
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}

// The function to add a key k to q
void enQueue(struct Queue *q, int k)
{
    // Create a new LL node
    struct QNode *temp = newNode(k);

    // If queue is empty, then new node is front and rear both
    if (q->rear == NULL)
    {
       q->front = q->rear = temp;
       return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove a key from given queue q
struct QNode *deQueue(struct Queue *q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
       return NULL;

    // Store previous front and move front one node ahead
    struct QNode *temp = q->front;
    q->front = q->front->next;
```

```
    // If front becomes NULL, then change rear also as NULL
    if (q->front == NULL)
       q->rear = NULL;
    return temp;
}

// Driver Program to test anove functions
int main()
{
    struct Queue *q = createQueue();
    enQueue(q, 10);
    enQueue(q, 20);
    deQueue(q);
    deQueue(q);
    enQueue(q, 30);
    enQueue(q, 40);
    enQueue(q, 50);
    struct QNode *n = deQueue(q);
    if (n != NULL)
      printf("Dequeued item is %d", n->key);
    return 0;
}
```

Java

```
 // Java program for linked-list implementation of queue

// A linked list (LL) node to store a queue entry
class QNode
{
    int key;
    QNode next;

    // constructor to create a new linked list node
    public QNode(int key) {
        this.key = key;
        this.next = null;
    }
}

// A class to represent a queue
//The queue, front stores the front node of LL and rear stores ths
//last node of LL
class Queue
{
    QNode front, rear;

    public Queue() {
```

```
        this.front = this.rear = null;
    }

    // Method to add an key to the queue.
    void enqueue(int key)
    {

        // Create a new LL node
        QNode temp = new QNode(key);

        // If queue is empty, then new node is front and rear both
        if (this.rear == null)
        {
           this.front = this.rear = temp;
           return;
        }

        // Add the new node at the end of queue and change rear
        this.rear.next = temp;
        this.rear = temp;
    }

    // Method to remove an key from queue.
    QNode dequeue()
    {
        // If queue is empty, return NULL.
        if (this.front == null)
           return null;

        // Store previous front and move front one node ahead
        QNode temp = this.front;
        this.front = this.front.next;

        // If front becomes NULL, then change rear also as NULL
        if (this.front == null)
           this.rear = null;
        return temp;
    }
}


// Driver class
public class Test
{
    public static void main(String[] args)
    {
        Queue q=new Queue();
        q.enqueue(10);
```

```
        q.enqueue(20);
        q.dequeue();
        q.dequeue();
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        System.out.println("Dequeued item is "+ q.dequeue().key);
    }
}
// This code is contributed by Gaurav Miglani
```

Python3

```
 # Python3 program to demonstrate linked list
# based implementation of queue

# A linked list (LL) node
# to store a queue entry
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

# A class to represent a queue

# The queue, front stores the front node
# of LL and rear stores ths last node of LL
class Queue:

    def __init__(self):
        self.front = self.rear = None

    def isEmpty(self):
        return self.front == None

    # Method to add an item to the queue
    def EnQueue(self, item):
        temp = Node(item)

        if self.rear == None:
            self.front = self.rear = temp
            return
        self.rear.next = temp
        self.rear = temp

    # Method to remove an item from queue
```

```
    def DeQueue(self):

        if self.isEmpty():
            return
        temp = self.front
        self.front = temp.next

        if(self.front == None):
            self.rear = None
        return str(temp.data)

# Driver Code
if __name__== '__main__':
    q = Queue()
    q.EnQueue(10)
    q.EnQueue(20)
    q.DeQueue()
    q.DeQueue()
    q.EnQueue(30)
    q.EnQueue(40)
    q.EnQueue(50)

    print("Dequeued item is " + q.DeQueue())
```

Output:

```
Dequeued item is 30
```

Time Complexity: Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

Source

https://www.geeksforgeeks.org/queue-set-2-linked-list-implementation/

Chapter 63

Reverse a path in BST using queue

Reverse a path in BST using queue - GeeksforGeeks

Given a binary search tree and a key, your task to reverse path of the binary tree.

Prerequisite : Reverse path of Binary tree

Examples :

```
Input :       50
            /     \
           30      70
          /  \    /  \
        20   40  60   80
k = 70
Output :
Inorder before reversal :
20 30 40 50 60 70 80
Inorder after reversal :
20 30 40 70 60 50 80

Input :       8
            /     \
           3       10
          /  \       \
        1    6         14
            /  \      /
           4    7    13
k = 13
Output :
Inorder before reversal :
```

```
1 3 4 6 7 8 10 13 14
Inorder after reversal :
1 3 4 6 7 13 14 10 8
```

Approach :
Take a queue and push all the element till that given key at the end replace node key with queue front element till root, then print inorder of the tree.

Below is the implementation of above approach :

```
 // CPP code to demonstrate insert
// operation in binary search tree
#include <bits/stdc++.h>
using namespace std;

struct node {
    int key;
    struct node *left, *right;
};

// A utility function to
// create a new BST node
struct node* newNode(int item)
{
    struct node* temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to
// do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// reverse tree path using queue
void reversePath(struct node** node,
            int& key, queue<int>& q1)
{
    /* If the tree is empty,
    return a new node */
    if (node == NULL)
```

```
    return;

// If the node key equal
// to key then
if ((*node)->key == key)
{
    // push current node key
    q1.push((*node)->key);

    // replace first node
    // with last element
    (*node)->key = q1.front();

    // remove first element
    q1.pop();

    // return
    return;
}

// if key smaller than node key then
else if (key < (*node)->key)
{
    // push node key into queue
    q1.push((*node)->key);

    // recusive call itself
    reversePath(&(*node)->left, key, q1);

    // replace queue front to node key
    (*node)->key = q1.front();

    // performe pop in queue
    q1.pop();
}

// if key greater than node key then
else if (key > (*node)->key)
{
    // push node key into queue
    q1.push((*node)->key);

    // recusive call itself
    reversePath(&(*node)->right, key, q1);

    // replace queue front to node key
    (*node)->key = q1.front();
```

```
        // performe pop in queue
        q1.pop();
    }

    // return
    return;
}

/* A utility function to insert
a new node with given key in BST */
struct node* insert(struct node* node,
                              int key)
{
    /* If the tree is empty,
    return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

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

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node* root = NULL;
    queue<int> q1;

    // reverse path till k
    int k = 80;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
```

```
    insert(root, 80);

    cout << "Before Reverse :" << endl;
    // print inoder traversal of the BST
    inorder(root);

    cout << "\n";

    // reverse path till k
    reversePath(&root, k, q1);

    cout << "After Reverse :" << endl;

    // print inorder of reverse path tree
    inorder(root);

    return 0;
}
```

Output:

```
Before Reverse :
20 30 40 50 60 70 80
After Reverse :
20 30 40 80 60 70 50
```

Source

https://www.geeksforgeeks.org/reverse-path-bst-using-queue/

Chapter 64

Reversing a Queue

Reversing a Queue - GeeksforGeeks

Give an algorithm for reversing a queue Q. Only following standard operations are allowed on queue.

1. enqueue(x) : Add an item x to rear of queue.
2. dequeue() : Remove an item from front of queue.
3. empty() : Checks if a queue is empty or not.

Examples:

```
Input : Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Output :Q = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

Input :[1, 2, 3, 4, 5]
Output :[5, 4, 3, 2, 1]
```

C++

```
 // CPP program to reverse a Queue
#include <bits/stdc++.h>
using namespace std;

// Utility function to print the queue
void Print(queue<int>& Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
```

```
}

// Function to reverse the queue
void reverseQueue(queue<int>& Queue)
{
    stack<int> Stack;
    while (!Queue.empty()) {
        Stack.push(Queue.front());
        Queue.pop();
    }
    while (!Stack.empty()) {
        Queue.push(Stack.top());
        Stack.pop();
    }
}

// Driver code
int main()
{
    queue<int> Queue;
    Queue.push(10);
    Queue.push(20);
    Queue.push(30);
    Queue.push(40);
    Queue.push(50);
    Queue.push(60);
    Queue.push(70);
    Queue.push(80);
    Queue.push(90);
    Queue.push(100);

    reverseQueue(Queue);
    Print(Queue);
}
```

Java

```
 // Java program to reverse a Queue
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

// Java program to reverse a queue
public class Queue_reverse {

    static Queue<Integer> queue;

    // Utility function to print the queue
```

```
    static void Print()
    {
        while (!queue.isEmpty()) {
            System.out.print( queue.peek() + ", ");
            queue.remove();
        }
    }

    // Function to reverse the queue
    static void reversequeue()
    {
        Stack<Integer> stack = new Stack<>();
        while (!queue.isEmpty()) {
            stack.add(queue.peek());
            queue.remove();
        }
        while (!stack.isEmpty()) {
            queue.add(stack.peek());
            stack.pop();
        }
    }

    // Driver code
    public static void main(String args[])
    {
        queue = new LinkedList<Integer>();
        queue.add(10);
        queue.add(20);
        queue.add(30);
        queue.add(40);
        queue.add(50);
        queue.add(60);
        queue.add(70);
        queue.add(80);
        queue.add(90);
        queue.add(100);

        reversequeue();
        Print();
    }
}
//This code is contributed by Sumit Ghosh
```

Output

```
100, 90, 80, 70, 60, 50, 40, 30, 20, 10
```

Source

https://www.geeksforgeeks.org/reversing-a-queue/

Chapter 65

Reversing a queue using recursion

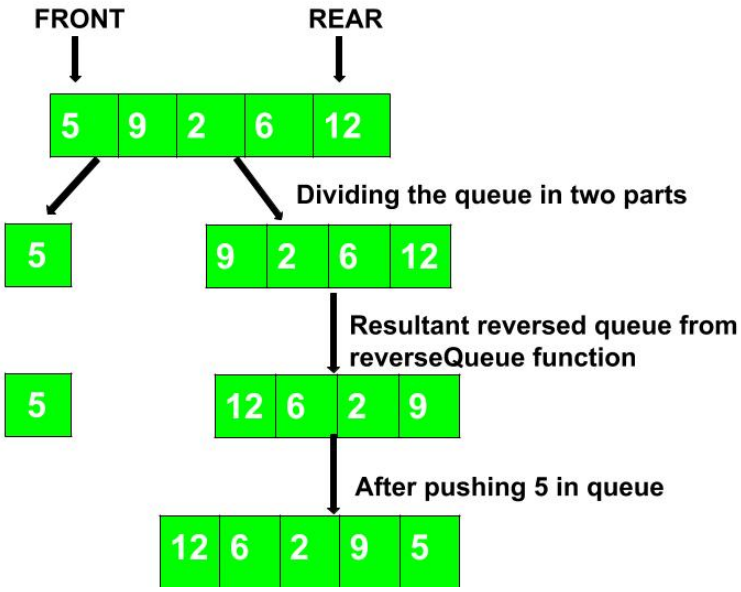
Reversing a queue using recursion - GeeksforGeeks

Given a queue, write a recursive function to reverse it.
Standard operations allowed :
enqueue(x) : Add an item x to rear of queue.
dequeue() : Remove an item from front of queue.
empty() : Checks if a queue is empty or not.

Examples :

```
Input : Q = [5, 24, 9, 6, 8, 4, 1, 8, 3, 6]
Output : Q = [6, 3, 8, 1, 4, 8, 6, 9, 24, 5]

Explanation : Output queue is the reverse of the input queue.

Input : Q = [8, 7, 2, 5, 1]
Output : Q = [1, 5, 2, 7, 8]
```

Recursive Algorithm :
1) Pop element from the queue if the queue has elements otherwise return empty queue.
2) Call reverseQueue function for the remaining queue.
3) Push the popped element in the resultant reversed queue.



Pseudo Code :

```
queue reverseFunction(queue)
{
    if (queue is empty)
       return queue;
    else {
       data = queue.front()
       queue.pop()
       queue = reverseFunction(queue);
       q.push(data);
       return queue;
    }
}
```

C++

```
 // C++ code for reversing a queue
#include <bits/stdc++.h>
using namespace std;

// Utility function to print the queue
void printQueue(queue<long long int> Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
```

```
}

// Recursive function to reverse the queue
void reverseQueue(queue<long long int>& q)
{
    // Base case
    if (q.empty())
        return;

    // Dequeue current item (from front)
    long long int data = q.front();
    q.pop();

    // Reverse remaining queue
    reverseQueue(q);

    // Enqueue current item (to rear)
    q.push(data);
}

// Driver code
int main()
{
    queue<long long int> Queue;
    Queue.push(56);
    Queue.push(27);
    Queue.push(30);
    Queue.push(45);
    Queue.push(85);
    Queue.push(92);
    Queue.push(58);
    Queue.push(80);
    Queue.push(90);
    Queue.push(100);
    reverseQueue(Queue);
    printQueue(Queue);
}
```

Java

```
 // Java program to reverse a Queue by recursion
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

// Java program to reverse a queue recursively
public class Queue_reverse {
```

```
    static Queue<Integer> queue;

    // Utility function to print the queue
    static void Print()
    {
        while (!queue.isEmpty())
        {
            System.out.print(queue.peek() + " ");
            queue.remove();
        }
    }

// Recurrsive function to reverse the queue
static Queue<Integer> reverseQueue(Queue<Integer> q)
{
    // Base case
    if (q.isEmpty())
        return q;

    // Dequeue current item (from front)
    int data = q.peek();
    q.remove();

    // Reverse remaining queue
    q = reverseQueue(q);

    // Enqueue current item (to rear)
    q.add(data);

    return q;
}

// Driver code
public static void main(String args[])
{
    queue = new LinkedList<Integer>();
    queue.add(56);
    queue.add(27);
    queue.add(30);
    queue.add(45);
    queue.add(85);
    queue.add(92);
    queue.add(58);
    queue.add(80);
    queue.add(90);
    queue.add(100);
    queue = reverseQueue(queue);
    Print();
```

```
}
}
```

Python3

```
 # Queue Class
class Queue:
    def __init__(self):
        self.items = []


    def isEmpty(self):
        return self.items == []

    def add(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop(0)

    def front(self):
        return self.items[0]

    def printQueue(self):
        for i in self.items:
            print(i, end =" ")
        print("")


# Recursive Function to reverse the queue
def reverseQueue(q):

    # Base case
    if (q.isEmpty()):
        return

    # Dequeue current item (from front)
    data = q.front();
    q.pop();

    # Reverse remaining queue
    reverseQueue(q)

    # Enqueue current item (to rear)
    q.add(data)


# Driver Code
```

```
q = Queue()
q.add(56)
q.add(27)
q.add(30)
q.add(45)
q.add(85)
q.add(92)
q.add(58)
q.add(80)
q.add(90)
q.add(100)
reverseQueue(q)
q.printQueue()
```

C#

```
 // C# code for reversing a queue
using System;
using System.Collections.Generic;

class GFG
{
    // Utility function
    // to print the queue
    static void printQueue(Queue<long> queue)
    {
        while (queue.Count != 0)
        {
            Console.Write(queue.Peek() + " ");
            queue.Dequeue();
        }
    }

    // Recursive function
    // to reverse the queue
    static void reverseQueue(ref Queue<long> q)
    {
        // Base case
        if (q.Count == 0)
            return;

        // Dequeue current
        // item (from front)
        long data = q.Peek();
        q.Dequeue();

        // Reverse remaining queue
        reverseQueue(ref q);
```

```
        // Enqueue current
        // item (to rear)
        q.Enqueue(data);
    }

    // Driver code
    static void Main()
    {
        Queue<long> queue = new Queue<long>();
        queue.Enqueue(56);
        queue.Enqueue(27);
        queue.Enqueue(30);
        queue.Enqueue(45);
        queue.Enqueue(85);
        queue.Enqueue(92);
        queue.Enqueue(58);
        queue.Enqueue(80);
        queue.Enqueue(90);
        queue.Enqueue(100);
        reverseQueue(ref queue);
        printQueue(queue);
    }
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

```
100 90 80 58 92 85 45 30 27 56
```

Time Complexity : O(n).

Improved By : manishshaw1

Source

https://www.geeksforgeeks.org/reversing-queue-using-recursion/

Chapter 66

Reversing the first K elements of a Queue

Reversing the first K elements of a Queue - GeeksforGeeks

Given an integer k and a queue of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

Only following standard operations are allowed on queue.

- enqueue(x) : Add an item x to rear of queue
- dequeue() : Remove an item from front of queue
- size((: Returns number of elements in queue.
- front() : Finds front item.

Examples:

```
Input : Q = [10, 20, 30, 40, 50, 60,
            70, 80, 90, 100]
        k = 5
Output : Q = [50, 40, 30, 20, 10, 60,
              70, 80, 90, 100]

Input : Q = [10, 20, 30, 40, 50, 60,
            70, 80, 90, 100]
        k = 4
Output : Q = [40, 30, 20, 10, 50, 60,
              70, 80, 90, 100]
```

The idea is to use an auxiliary stack.

1) Create an empty stack.
2) One by one dequeue items from given queue and push the dequeued items to stack.

3) Enqueue the contents of stack at the back of the queue
4) Reverse the whole queue.

C++

```
 // C++ program to reverse first k elements of a queue.
#include <bits/stdc++.h>
using namespace std;

/* Function to reverse the first K elements of the Queue */
void reverseQueueFirstKElements(int k, queue<int>& Queue)
{
    if (Queue.empty() == true || k > Queue.size())
        return;
    if (k <= 0)
        return;

    stack<int> Stack;

    /* Push the first K elements into a Stack*/
    for (int i = 0; i < k; i++) {
        Stack.push(Queue.front());
        Queue.pop();
    }

    /* Enqueue the contents of stack
       at the back of the queue*/
    while (!Stack.empty()) {
        Queue.push(Stack.top());
        Stack.pop();
    }

    /* Remove the remaining elements and
       enqueue them at the end of the Queue*/
    for (int i = 0; i < Queue.size() - k; i++) {
        Queue.push(Queue.front());
        Queue.pop();
    }
}

/* Utility Function to print the Queue */
void Print(queue<int>& Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
}
```

```
// Driver code
int main()
{
    queue<int> Queue;
    Queue.push(10);
    Queue.push(20);
    Queue.push(30);
    Queue.push(40);
    Queue.push(50);
    Queue.push(60);
    Queue.push(70);
    Queue.push(80);
    Queue.push(90);
    Queue.push(100);

    int k = 5;
    reverseQueueFirstKElements(k, Queue);
    Print(Queue);
}
```

Java

```
 // Java program to reverse first k elements
// of a queue.
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Reverse_k_element_queue {

    static Queue<Integer> queue;

    // Function to reverse the first K elements
    // of the Queue
    static void reverseQueueFirstKElements(int k) {
        if (queue.isEmpty() == true || k > queue.size())
            return;
        if (k <= 0)
            return;

        Stack<Integer> stack = new Stack<Integer>();

        // Push the first K elements into a Stack
        for (int i = 0; i < k; i++) {
            stack.push(queue.peek());
            queue.remove();
        }
```

```
        // Enqueue the contents of stack at the back
        // of the queue
        while (!stack.empty()) {
            queue.add(stack.peek());
            stack.pop();
        }


        // Remove the remaining elements and enqueue
        // them at the end of the Queue
        for (int i = 0; i < queue.size() - k; i++) {
            queue.add(queue.peek());
            queue.remove();
        }
    }

    // Utility Function to print the Queue
    static void Print() {
        while (!queue.isEmpty()) {
            System.out.print(queue.peek() + " ");
            queue.remove();
        }
    }

    // Driver code
    public static void main(String args[]) {
        queue = new LinkedList<Integer>();
        queue.add(10);
        queue.add(20);
        queue.add(30);
        queue.add(40);
        queue.add(50);
        queue.add(60);
        queue.add(70);
        queue.add(80);
        queue.add(90);
        queue.add(100);

        int k = 5;
        reverseQueueFirstKElements(k);
        Print();
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
50 40 30 20 10 60 70 80 90 100
```

Source

https://www.geeksforgeeks.org/reversing-first-k-elements-queue/

Chapter 67

Sharing a queue among three threads

Sharing a queue among three threads - GeeksforGeeks

Share a queue among three threads A, B, C as per given norms :

- Thread A generates random integers and pushes them into a shared queue.
- Threads B and C compete with each other to grab an integer from the queue.
- The threads B and C compute the sum of integers that they have grabbed from the queue.
- Compare the sums as computed by B and C. The greatest is the winner.

Prerequisite :- Multithreading

Approach :- Create a global queue which is shared among all three threads. First create all three threads and call the respective functions associated with them.

- **producerFun** generates random numbers and push them into queue
- **add_B** function replicates thread B and consumes the queue for certain numbers.
- **add_C** function replicates thread C and consumes the queue for certain numbers.

Note :- There is mutex lock in every function to avoid any race condition.

```
 // CPP program to demonstrate the given task
#include <iostream>
#include <pthread.h>
#include <queue>
#include <stdlib.h>

#define MAX 10
```

```
using namespace std;

// Declaring global variables
int sum_B = 0, sum_C = 0;
int consumerCount1 = 0;
int consumerCount2 = 0;

// Shared queue
queue<int> Q;

// Fuction declaration of all required functions
void* producerFun(void*);
void* add_B(void*);
void* add_C(void*);

// Getting the mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t dataNotProduced =
                    PTHREAD_COND_INITIALIZER;
pthread_cond_t dataNotConsumed =
                    PTHREAD_COND_INITIALIZER;

// Function to generate random numbers and
// push them into queue using thread A

void* producerFun(void*)
{
    static int producerCount = 0;

    // Initialising the seed
    srand(time(NULL));

    while (1) {
        // Getting the lock on queue using mutex
        pthread_mutex_lock(&mutex);

        if (Q.size() < MAX && producerCount < MAX)
        {

            // Getting the random number
            int num = rand() % 10 + 1;
            cout << "Produced:  " << num << endl;

            // Pushing the number into queue
            Q.push(num);

            producerCount++;
```

```
            pthread_cond_broadcast(&dataNotProduced);
        }

        // If queue is full, release the lock and return
        else if (producerCount == MAX) {
            pthread_mutex_unlock(&mutex);
            return NULL;
        }

        // If some other thread is exectuing, wait
        else {
            cout << ">> Producer is in wait.." << endl;
            pthread_cond_wait(&dataNotConsumed, &mutex);
        }

        // Get the mutex unlocked
        pthread_mutex_unlock(&mutex);
    }
}

// Function definition for consumer thread B
void* add_B(void*)
{

    while (1) {

        // Getting the lock on queue using mutex
        pthread_mutex_lock(&mutex);

        // Pop only when queue has at least 1 element
        if (Q.size() > 0) {
            // Get the data from the front of queue
            int data = Q.front();

            cout << "B thread consumed: " << data << endl;

            // Add the data to the integer variable
            // associated with thread B
            sum_B += data;

            // Pop the consumed data from queue
            Q.pop();

            consumerCount1++;

            pthread_cond_signal(&dataNotConsumed);
        }
```

```
        // Check if consmed numbers from both threads
        // has reached to MAX value
        else if (consumerCount2 + consumerCount1 == MAX) {
            pthread_mutex_unlock(&mutex);
            return NULL;
        }

        // If some other thread is exectuing, wait
        else {
            cout << "B is in wait.." << endl;
            pthread_cond_wait(&dataNotProduced, &mutex);
        }

        // Get the mutex unlocked
        pthread_mutex_unlock(&mutex);
    }
}

// Function definition for consumer thread C
void* add_C(void*)
{

    while (1) {

        // Getting the lock on queue using mutex
        pthread_mutex_lock(&mutex);

        // Pop only when queue has at least 1 element
        if (Q.size() > 0) {

            // Get the data from the front of queue
            int data = Q.front();
            cout << "C thread consumed: " << data << endl;

            // Add the data to the integer variable
            // associated with thread B
            sum_C += data;

            // Pop the consumed data from queue
            Q.pop();
            consumerCount2++;

            pthread_cond_signal(&dataNotConsumed);
        }

        // Check if consmed numbers from both threads
        // has reached to MAX value
```

```
        else if (consumerCount2 + consumerCount1 == MAX)
        {
            pthread_mutex_unlock(&mutex);
            return NULL;
        }

        // If some other thread is exectuing, wait
        else {
            cout << ">> C is in wait.." << endl;
            // Wait on a condition
            pthread_cond_wait(&dataNotProduced, &mutex);
        }

        // Get the mutex unlocked
        pthread_mutex_unlock(&mutex);
    }
}

// Driver code
int main()
{
    // Declaring integers used to
    // identify the thread in the system
    pthread_t producerThread, consumerThread1, consumerThread2;

    // Function to create a threads
    // (pthread_create() takes 4 arguments)
    int retProducer = pthread_create(&producerThread,
                      NULL, producerFun, NULL);
    int retConsumer1 = pthread_create(&consumerThread1,
                       NULL, *add_B, NULL);
    int retConsumer2 = pthread_create(&consumerThread2,
                       NULL, *add_C, NULL);

    // pthread_join suspends execution of the calling
    // thread until the target thread terminates
    if (!retProducer)
        pthread_join(producerThread, NULL);
    if (!retConsumer1)
        pthread_join(consumerThread1, NULL);
    if (!retConsumer2)
        pthread_join(consumerThread2, NULL);

    // Checking for the final value of thread
    if (sum_C > sum_B)
        cout << "Winner is  Thread C" << endl;
    else if (sum_C < sum_B)
        cout << "Winner is  Thread B" << endl;
```

```
    else
        cout << "Both has same score" << endl;

    return 0;
}
```

```
Output:
B is in wait..
Produced:  10
Produced:  1
Produced:  6
Produced:  6
Produced:  4
C thread consumed: 10
C thread consumed: 1
C thread consumed: 6
C thread consumed: 6
Produced:  1
Produced:  9
Produced:  9
Produced:  6
C thread consumed: 4
C thread consumed: 1
C thread consumed: 9
C thread consumed: 9
C thread consumed: 6
>> C is in wait..
Produced:  5
C thread consumed: 5
Winner is  Thread C
```

Note : Output will be different everytime code runs.

Source

https://www.geeksforgeeks.org/sharing-queue-among-three-threads/

Chapter 68

Sliding Window Maximum (Maximum of all subarrays of size k)

Sliding Window Maximum (Maximum of all subarrays of size k) - GeeksforGeeks

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples :

Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
k = 3
Output :
3 3 4 5 5 5 6

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}
k = 4
Output :
10 10 10 15 15 90 90

Method 1 (Simple)
Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

C/C++

```
 #include<stdio.h>

void printKMax(int arr[], int n, int k)
{
```

```
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i+j] > max)
               max = arr[i+j];
        }
        printf("%d ", max);
    }
}


int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Java

```
 //Java Program to find the maximum for each and every contiguous subarray of size k.

public class GFG
{
    // Method to find the maximum for each and every contiguous subarray of size k.
    static void printKMax(int arr[], int n, int k)
    {
        int j, max;

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

            max = arr[i];

            for (j = 1; j < k; j++)
            {
                if (arr[i + j] > max)
                    max = arr[i + j];
            }
            System.out.print(max + " ");
        }
    }
```

```
    // Driver method
    public static void main(String args[])
    {
        int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int k = 3;
        printKMax(arr, arr.length, k);
    }
}

//This code is contributed by Sumit Ghosh
```

Python3

```
 # Python program to find the maximum for
# each and every contiguous subarray of
# size k

# Method to find the maximum for each
# and every contiguous subarray of s
# of size k
def printMax(arr, n, k):
    max = 0

    for i in range(n - k + 1):
        max = arr[i]
        for j in range(1, k):
            if arr[i+j] > max:
                max = arr[i + j]
        print(str(max) + " ", end = "")

# Driver method
if __name__=="__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    n = len(arr)
    k = 3
    printMax(arr, n, k)

# This code is contributed by Shiv Shankar
```

C#

```
 // C# program to find the maximum for
// each and every contiguous subarray of
// size kusing System;
using System;
```

```
class GFG
{
    // Method to find the maximum for
    // each and every contiguous subarray
    // of size k.
    static void printKMax(int []arr, int n, int k)
    {
        int j, max;

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

            max = arr[i];

            for (j = 1; j < k; j++)
            {
                if (arr[i + j] > max)
                    max = arr[i + j];
            }
            Console.Write(max + " ");
        }
    }

    // Driver method
    public static void Main()
    {
        int []arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int k = 3;
        printKMax(arr, arr.Length, k);
    }
}

// This Code is Contributed by Sam007
```

PHP

```
 <?php
// PHP program to find the maximum
// for each and every contiguous
// subarray of size k

function printKMax($arr, $n, $k)
{
    $j; $max;

    for ($i = 0; $i <= $n - $k; $i++)
    {
        $max = $arr[$i];
```

```
        for ($j = 1; $j < $k; $j++)
        {
            if ($arr[$i + $j] > $max)
            $max = $arr[$i + $j];
        }
        printf("%d ", $max);
    }
}

// Driver Code
$arr = array(1, 2, 3, 4, 5,
             6, 7, 8, 9, 10);
$n = count($arr);
$k = 3;
printKMax($arr, $n, $k);

// This Code is Contributed by anuj_67.
?>
```

Output :

```
3 4 5 6 7 8 9 10
```

Time Complexity : The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(nk).

Method 2 (Use Self-Balancing BST)
1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
2) Run a loop for i = 0 to n – k
…..a) Get the maximum element from the BST, and print it.
…..b) Search for arr[i] in the BST and delete it from the BST.
…..c) Insert arr[i+k] into the BST.

Time Complexity: Time Complexity of step 1 is O(kLogk). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). Since steps 2(a), 2(b) and 2(c) are in a loop that runs n-k+1 times, time complexity of the complete algorithm is O(kLogk + (n-k+1)*Logk) which can also be written as O(nLogk).

Method 3 (A O(n) method: use Dequeue)
We create a Dequeue, *Qi* of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain *Qi* to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the *Qi* is the largest and element at rear of *Qi* is the smallest of current window. Thanks to Aashishfor suggesting this method.

Following is the implementation of this method.

C++

```
 #include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maixmum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int>  Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For very element, the previous smaller elements are useless so
        // remove them from Qi
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();  // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ( (!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front();  // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

         // Add current element at the rear of Qi
        Qi.push_back(i);
```

```
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 1, 78, 90, 57, 89, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Java

```
 //Java Program to find the maximum for each and every contiguous subarray of size k.

import java.util.Deque;
import java.util.LinkedList;

public class SlidingWindow
{

    // A Dequeue (Double ended queue) based method for printing maixmum element of
    // all subarrays of size k
    static void printMax(int arr[],int n, int k)
    {
        // Create a Double Ended Queue, Qi that will store indexes of array elements
        // The queue will store indexes of useful elements in every window and it will
        // maintain decreasing order of values from front to rear in Qi, i.e.,
        // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
        Deque<Integer> Qi = new LinkedList<Integer>();

        /* Process first k (or first window) elements of array */
        int i;
        for(i = 0; i < k; ++i)
        {
            // For very element, the previous smaller elements are useless so
            // remove them from Qi
            while(!Qi.isEmpty() && arr[i] >= arr[Qi.peekLast()])
                Qi.removeLast();   // Remove from rear

            // Add new element at rear of queue
            Qi.addLast(i);
        }
```

```
        // Process rest of the elements, i.e., from arr[k] to arr[n-1]
        for( ;i < n; ++i)
        {
            // The element at the front of the queue is the largest element of
            // previous window, so print it
            System.out.print(arr[Qi.peek()] + " ");

            // Remove the elements which are out of this window
            while((!Qi.isEmpty()) && Qi.peek() <= i-k)
                Qi.removeFirst();

            // Remove all elements smaller than the currently
            // being added element (remove useless elements)
            while((!Qi.isEmpty()) && arr[i] >= arr[Qi.peekLast()])
                Qi.removeLast();


            // Add current element at the rear of Qi
            Qi.addLast(i);

        }

        // Print the maximum element of last window
        System.out.print(arr[Qi.peek()]);
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        int arr[]={12, 1, 78, 90, 57, 89, 56};
        int k=3;
        printMax(arr, arr.length,k);
    }

}
//This code is contributed by Sumit Ghosh
```

Python3

```
 # Python program to find the maximum for
# each and every contiguous subarray of
# size k

from collections import deque

# A Deque (Double ended queue) based
# method for printing maximum element
```

```
# of all subarrays of size k
def printMax(arr, n, k):

    """ Create a Double Ended Queue, Qi that
    will store indexes of array elements.
    The queue will store indexes of useful
    elements in every window and it will
    maintain decreasing order of values from
    front to rear in Qi, i.e., arr[Qi.front[]]
    to arr[Qi.rear()] are sorted in decreasing
    order"""
    Qi = deque()

    # Process first k (or first window)
    # elements of array
    for i in range(k):

        # For every element, the previous
        # smaller elements are useless
        # so remove them from Qi
        while Qi and arr[i] >= arr[Qi[-1]] :
            Qi.pop()

        # Add new element at rear of queue
        Qi.append(i);

    # Process rest of the elements, i.e.
    # from arr[k] to arr[n-1]
    for i in range(k, n):

        # The element at the front of the
        # queue is the largest element of
        # previous window, so print it
        print(str(arr[Qi[0]]) + " ", end = "")

        # Remove the elements which are
        # out of this window
        while Qi and Qi[0] <= i-k:

            # remove from front of deque
            Qi.popleft()

        # Remove all elements smaller than
        # the currently being added element
        # (Remove useless elements)
        while Qi and arr[i] >= arr[Qi[-1]] :
            Qi.pop()
```

```
        # Add current element at the rear of Qi
        Qi.append(i)

    # Print the maximum element of last window
    print(str(arr[Qi[0]]))

# Driver programm to test above fumctions
if __name__=="__main__":
    arr = [12, 1, 78, 90, 57, 89, 56]
    k = 3
    printMax(arr, len(arr), k)

# This code is contributed by Shiv Shankar
```

Output:

```
78 90 90 90 89
```

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total 2n operations.
Auxiliary Space: O(k)

Below is an extension of this problem.
Sum of minimum and maximum elements of all subarrays of size k.

Improved By : vt_m

Source

https://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-size-k/

Chapter 69

Smallest multiple of a given number made of digits 0 and 9 only

Smallest multiple of a given number made of digits 0 and 9 only - GeeksforGeeks

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 10^6.

Examples:

```
Input : N = 5
Output : X = 90
Exaplanation: 90 is the smallest number made up
of 9's and 0's which is divisible by 5.

Input : N = 7
Output : X = 9009
Exaplanation: 9009 is smallest number made up
of 9's and 0's which is 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.

We will use the method of generating binary numbers to generate all numbers which can be formed by using digits 0 & 9.

Below is the implementation of above idea:

C++

```
 // CPP program to find smallest multiple of a
// given number made of digits 0 and 9 only
#include <bits/stdc++.h>
using namespace std;

// 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()
{
    // Create an empty queue of strings
    queue<string> q;

    // enque the first number
    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"));
    }
}

// function to find smallest number made up of only
// digits 9's and 0's, which is a multiple of n.
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];
}

// Driver Code
int main()
{
    generateNumbersUtil();
    int n = 7;
    cout << findSmallestMultiple(n);
    return 0;
}
```

C#

```
 // C# program to find smallest
// multiple of a given number
// made of digits 0 and 9 only
using System;
using System.Collections.Generic;

class GFG
{
    // Maximum number of
    // numbers made of 0 and 9
    static int MAX_COUNT = 10000;

    // vector to store all numbers
    // that can be formed using
    // digits 0 and 9 and are
    // less than 10^5
    static List<string> vec = new List<string>();

    /* Preprocessing function
    to generate all possible
    numbers formed by 0 and 9 */
    static void generateNumbersUtil()
    {
        // Create an empty
        // queue of strings
        Queue<string> q = new Queue<string>();
```

```
    // enque the
    // first number
    q.Enqueue("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.Peek();
        q.Dequeue();

        // storing the Peek of
        // queue in the vector
        vec.Add(s1);

        string s2 = s1;

        // Append "0" to s1
        // and enqueue it
        q.Enqueue(s1 + "0");

        // Append "9" to s2 and
        // enqueue it. Note that
        // s2 contains the previous Peek
        q.Enqueue(s2 + "9");
    }
}

// function to find smallest
// number made up of only
// digits 9's and 0's, which
// is a multiple of n.
static string findSmallestMultiple(int n)
{
    // traverse the vector
    // to find the smallest
    // multiple of n
    for (int i = 0; i < vec.Count; i++)

        // stoi() is used for
        // string to int conversion
        if (int.Parse(vec[i]) % n == 0)
            return vec[i];
    return "";
}
```

```
    // Driver Code
    static void Main()
    {
        generateNumbersUtil();
        int n = 7;
        Console.Write(findSmallestMultiple(n));
    }
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

```
9009
```

Time Complexity: O(n)

Improved By : manishshaw1

Source

https://www.geeksforgeeks.org/smallest-multiple-of-a-given-number-made-of-digits-0-and-9-only/

Chapter 70

Sorting a Queue without extra space

Sorting a Queue without extra space - GeeksforGeeks

Given a queue with random elements, we need to sort it. We are not allowed to use extra space. The operations allowed on queue are :

1. enqueue() : Adds an item to rear of queue. In C++ STL queue, this function is called push().
2. dequeue() : Removes an item from front of queue. In C++ STL queue, this function is called pop().
3. isEmpty() : Checks if a queue is empty. In C++ STL queue, this function is called empty().

Examples :

```
Input : A queue with elements
        11  5  4  21
Output : Modified queue with
         following elements
        4 5 11 21

Input : A queue with elements
        3  2  1  2
Output : Modified queue with
         following elements
        1 2 2 3
```

If we are allowed extra space, then we can simply move all items of queue to an array, then sort the array and finally move array elements back to queue.

How to do without extra space?
The idea: 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 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.

Illustration:

```
Input :
-------------
11  5  4  21    min index = 2
-------------
-------------
11  5  21  4    after inserting 4
-------------
-------------
11  5  21  4    min index = 1
-------------
-------------
11  21  4  5   after inserting 5
-------------

-------------
11  21  4  5   min index = 0
-------------

-------------
21  4  5  11   after inserting 11
-------------

-------------
21  4  5  11   min index = 0
-------------

-------------
4  5  11  21   after inserting 21
-------------
Output : 4 5 11 21
```

C++

```
 // C++ program to implement sorting a
// queue data structure
#include <bits/stdc++.h>
using namespace std;

// Queue elements after sortedIndex are
// already sorted. This function returns
// index of minimum element from front to
// sortedIndex
int minIndex(queue<int> &q, int sortedIndex)
{
    int min_index = -1;
    int min_val = INT_MAX;
    int n = q.size();
    for (int i=0; i<n; i++)
    {
        int curr = q.front();
        q.pop();  // This is dequeue() in C++ STL

        // we add the condition i <= sortedIndex
        // because we don't want to traverse
        // on the sorted part of the queue,
        // which is the right part.
        if (curr <= min_val && i <= sortedIndex)
        {
            min_index = i;
            min_val = curr;
        }
        q.push(curr);  // This is enqueue() in
                       // C++ STL
    }
    return min_index;
}

// Moves 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);
    }
}

// driver code
int main()
{
  queue<int> q;
  q.push(30);
  q.push(11);
  q.push(15);
  q.push(4);

  // Sort queue
  sortQueue(q);

  // Print sorted queue
  while (q.empty() == false)
  {
     cout << q.front() << " ";
     q.pop();
  }
  cout << endl;
  return 0;
}
```

Java

```
 // Java program to implement sorting a
// queue data structure
import java.util.LinkedList;
import java.util.Queue;
class GFG
{
    // Queue elements after sortIndex are
    // already sorted. This function returns
    // index of minimum element from front to
    // sortIndex
    public static int minIndex(Queue<Integer> list,
```

```
                                        int sortIndex)
    {
    int min_index = -1;
    int min_value = Integer.MAX_VALUE;
    int s = list.size();
    for (int i = 0; i < s; i++)
    {
        int current = list.peek();

        // This is dequeue() in Java STL
        list.poll();

        // we add the condition i <= sortIndex
        // because we don't want to traverse
        // on the sorted part of the queue,
        // which is the right part.
        if (current <= min_value && i <= sortIndex)
        {
            min_index = i;
            min_value = current;
        }
        list.add(current);
    }
    return min_index;
}

    // Moves given minimum element
    // to rear of queue
    public static void insertMinToRear(Queue<Integer> list,
                                            int min_index)
     {
        int min_value = 0;
        int s = list.size();
        for (int i = 0; i < s; i++)
        {
        int current = list.peek();
        list.poll();
        if (i != min_index)
            list.add(current);
        else
            min_value = current;
        }
        list.add(min_value);
    }

    public static void sortQueue(Queue<Integer> list)
    {
        for(int i = 1; i <= list.size(); i++)
```

```
        {
            int min_index = minIndex(list,list.size() - i);
            insertMinToRear(list, min_index);
        }
    }

    //Driver function
    public static void main (String[] args)
    {
        Queue<Integer> list = new LinkedList<Integer>();
        list.add(30);
        list.add(11);
        list.add(15);
        list.add(4);

        //Sort Queue
        sortQueue(list);

        //print sorted Queue
        while(list.isEmpty()== false)
        {
            System.out.print(list.peek() + " ");
            list.poll();
        }
    }
}

// This code is contributed by akash1295
```

C#

```
 // C# program to implement
// sorting a queue data structure
using System;
using System.Collections.Generic;

class GFG
{
    // Queue elements after sorted
    // Index are already sorted.
    // This function returns index
    // of minimum element from front
    // to sortedIndex
    static int minIndex(ref Queue<int> q,
                        int sortedIndex)
    {
        int min_index = -1;
        int min_val = int.MaxValue;
```

```
    int n = q.Count;
    for (int i = 0; i < n; i++)
    {
        int curr = q.Peek();
        q.Dequeue(); // This is dequeue()
                     // in C++ STL

        // we add the condition
        // i <= sortedIndex because
        // we don't want to traverse
        // on the sorted part of the
        // queue, which is the right part.
        if (curr <= min_val &&
               i <= sortedIndex)
        {
            min_index = i;
            min_val = curr;
        }
        q.Enqueue(curr); // This is enqueue()
                         // in C++ STL
    }
    return min_index;
}

// Moves given minimum
// element to rear of queue
static void insertMinToRear(ref Queue<int> q,
                            int min_index)
{
    int min_val = 0;
    int n = q.Count;
    for (int i = 0; i < n; i++)
    {
        int curr = q.Peek();
        q.Dequeue();
        if (i != min_index)
            q.Enqueue(curr);
        else
            min_val = curr;
    }
    q.Enqueue(min_val);
}

static void sortQueue(ref Queue<int> q)
{
    for (int i = 1; i <= q.Count; i++)
    {
        int min_index = minIndex(ref q,
```

```
                                        q.Count - i);
            insertMinToRear(ref q,
                            min_index);
        }
    }

    // Driver Code
    static void Main()
    {
        Queue<int> q = new Queue<int>();
        q.Enqueue(30);
        q.Enqueue(11);
        q.Enqueue(15);
        q.Enqueue(4);

        // Sort queue
        sortQueue(ref q);

        // Print sorted queue
        while (q.Count != 0)
        {
            Console.Write(q.Peek() + " ");
            q.Dequeue();
        }
        Console.WriteLine();
    }
}
// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

```
 4 11 15 30
```

Time complexity of this algorithm is O(n^2).
Extra space needed is O(1).

Improved By : akash1295, manishshaw1

Source

https://www.geeksforgeeks.org/sorting-queue-without-extra-space/

Chapter 71

Stack Permutations (Check if an array is stack permutation of other)

Stack Permutations (Check if an array is stack permutation of other) - GeeksforGeeks

A **stack permutation** is a permutation of objects in the given input queue which is done by transferring elements from input queue to the output queue with the help of a stack and the built-in push and pop functions.

The well defined rules are:

1. Only dequeue from the input queue.
2. Use inbuilt push, pop functions in the single stack.
3. Stack and input queue must be empty at the end.
4. Only enqueue to the output queue.

There are a huge number of permutations possible using a stack for a single input queue. Given two arrays, both of unique elements. One represents the input queue and the other represents the output queue. Our task is to check if the given output is possible through stack permutation.

Examples:

```
Input : First array: 1, 2, 3
        Second array: 2, 1, 3
Output : Yes
Procedure:
push 1 from input to stack
push 2 from input to stack
```

```
pop 2 from stack to output
pop 1 from stack to output
push 3 from input to stack
pop 3 from stack to output


Input : First array: 1, 2, 3
        Second array: 3, 1, 2
Output : Not Possible
```

The idea to do this is we will try to convert the input queue to output queue using a stack, if we are able to do so then the queue is permutable otherwise not.
Below is the step by step algorithm to do this:

1. Continuously pop elements from the input queue and check if it is equal to the top of output queue or not, if it is not equal to the top of output queue then we will push the element to stack.
2. Once we find an element in input queue such the top of input queue is equal to top of output queue, we will pop a single element from both input and output queues, and compare the top of stack and top of output queue now. If top of both stack and output queue are equal then pop element from both stack and output queue. If not equal, go to step 1.
3. Repeat above two steps until the input queue becomes empty. At the end if both of the input queue and stack are empty then the input queue is permutable otherwise not.

Below is C++ implementation of above idea:

```
 // Given two arrays, check if one array is
// stack permutation of other.
#include<bits/stdc++.h>
using namespace std;

// function to check if input queue is
// permutable to output queue
bool checkStackPermutation(int ip[], int op[], int n)
{
    // Input queue
    queue<int> input;
    for (int i=0;i<n;i++)
        input.push(ip[i]);

    // output queue
    queue<int> output;
    for (int i=0;i<n;i++)
        output.push(op[i]);
```

```
    // stack to be used for permutation
    stack <int> tempStack;
    while (!input.empty())
    {
        int ele = input.front();
        input.pop();
        if (ele == output.front())
        {
            output.pop();
            while (!tempStack.empty())
            {
                if (tempStack.top() == output.front())
                {
                    tempStack.pop();
                    output.pop();
                }
                else
                    break;
            }
        }
        else
            tempStack.push(ele);
    }

    // If after processing, both input queue and
    // stack are empty then the input queue is
    // permutable otherwise not.
    return (input.empty()&&tempStack.empty());
}

// Driver program to test above function
int main()
{
    // Input Queue
    int input[] = {1, 2, 3};

    // Output Queue
    int output[] = {2, 1, 3};

    int n = 3;

    if (checkStackPermutation(input, output, n))
        cout << "Yes";
    else
        cout << "Not Possible";
    return 0;
}
```

Output:

```
Yes
```

Source

https://www.geeksforgeeks.org/stack-permutations-check-if-an-array-is-stack-permutation-of-other/

Chapter 72

Stack and Queue in Python using queue Module

Stack and Queue in Python using queue Module - GeeksforGeeks

A simple python List can act as queue and stack as well. Queue mechanism is used widely and for many purposes in daily life. A queue follows FIFO rule(First In First Out) and is used in programming for sorting and for many more things. Python provides Class queue as a module which has to be generally created in languages such as C/C++ and Java.

1. Creating a FIFO Queue

```
// Initialize queue
Syntax: queue.Queue(maxsize)

// Insert Element
Syntax: Queue.put(data)

// Get And remove the element
Syntax: Queue.get()
```

Initializes a variable to a maximum size of maxsize. A maxsize of zero '0' means a infinite queue. This Queue follows FIFO rule. This module also has a LIFO Queue, which is basically a Stack. Data is inserted into Queue using put() and the end. get() takes data out from the front of the Queue. Note that Both put() and get() take 2 more parameters, optional flags, block and timeout.

```
 import queue

# From class queue, Queue is
# created as an object Now L
```

```
# is Queue of a maximum
# capacity of 20
L = queue.Queue(maxsize=20)

# Data is inserted into Queue
# using put() Data is inserted
# at the end
L.put(5)
L.put(9)
L.put(1)
L.put(7)

# get() takes data out from
# the Queue from the head
# of the Queue
print(L.get())
print(L.get())
print(L.get())
print(L.get())
```

Output:

```
5
9
1
7
```

2. UnderFlow and OverFlow
When we try to add data into a Queue above is maxsize, it is called OverFlow(Queue Full) and when we try removing an element from an empty, it's called Underflow. put() and get() do not give error upon Underflow and Overflow, but goes into an infinite loop.

```
 import queue

L = queue.Queue(maxsize=6)

# qsize() give the maxsize
# of the Queue
print(L.qsize())

L.put(5)
L.put(9)
L.put(1)
L.put(7)

# Return Boolean for Full
```

```
# Queue
print("Full: ", L.full())

L.put(9)
L.put(10)
print("Full: ", L.full())

print(L.get())
print(L.get())
print(L.get())

# Return Boolean for Empty
# Queue
print("Empty: ", L.empty())

print(L.get())
print(L.get())
print(L.get())

print("Empty: ", L.empty())
print("Full: ", L.full())

# This would result into Infinite
# Loop as the Queue is empty.
# print(L.get())
```

Output:

```
0
Full:  False
Full:  True
5
9
1
Empty:  False
7
9
10
Empty:  True
Full:  False
```

3. Stack
This module queue also provides LIFO Queue which technically works as a Stack.

```
import queue
```

```
L = queue.LifoQueue(maxsize=6)

# qsize() give the maxsize of
# the Queue
print(L.qsize())

# Data Inserted as 5->9->1->7,
# same as Queue
L.put(5)
L.put(9)
L.put(1)
L.put(7)
L.put(9)
L.put(10)
print("Full: ", L.full())
print("Size: ", L.qsize())

# Data will be accessed in the
# reverse order Reverse of that
# of Queue
print(L.get())
print(L.get())
print(L.get())
print(L.get())
print(L.get())
print("Empty: ", L.empty())
```

Output:

```
0
Full:  True
Size: 6
10
9
7
1
9
Empty:  False
```

Reference:
https://docs.python.org/3/library/asyncio-queue.html

Source

https://www.geeksforgeeks.org/stack-queue-python-using-module-queue/

Chapter 73

Sudo Placement[1.3] | Final Destination

Sudo Placement[1.3] | Final Destination - GeeksforGeeks

Given an array of integers and a number K with initial and final values. Your task is to find the minimum number of steps required to get final value starting from the initial value using the array elements. You can only do add (add operation % 1000) on values to get the final value. At every step, you are allowed to add any of the array elements with modulus operation.

Examples:

> **Input:** initial = 1, final = 6, a[] = {1, 2, 3, 4}
> **Output:** 2
> Step 1: (1 + 1) % 1000 = 2.
> Step 2: (2 + 4) % 1000 = 6 (which is required final value).
>
> **Input:** start = 998 end = 2 a[] = {2, 1, 3}
> **Output:** 2
> Step 1 : (998 + 2) % 1000 = 0.
> Step 2 : (0 + 2) % 1000 = 2.
> OR
> Step 1 : (998 + 1) % 1000 = 999.
> Step 2 : (999 + 3) % 1000 = 2

Approach: Since in the above problem the modulus given is 1000, therefore the maximum number of states will be 10^3. All the states can be checked using simple BFS. Initialize an ans[] array with -1 which marks that the state has not been visited. ans[i] stores the number of steps taken to reach i from start. Initially push the start to the queue, then apply BFS. Pop the top element and check if it is equal to the end if it is then print the ans[end]. If the element is not equal to the topmost element, then add the top element with every element in the array and perform a mod operation with 1000. If the added element state has not been visited previously, then push it into the queue. Initialize ans[pushed_element]

by ans[top_element] + 1. Once all the states are visited, and the state cannot be reached by performing every possible multiplication, then print -1.

Below is the implementation of the above approach:

```
 // C++ program to find the minimum steps
// to reach end from start by performing
// additions and mod operations with array elements
#include <bits/stdc++.h>
using namespace std;

// Function that returns the minimum operations
int minimumAdditions(int start, int end, int a[], int n)
{
    // array which stores the minimum steps
    // to reach i from start
    int ans[1001];

    // -1 indicated the state has not been visited
    memset(ans, -1, sizeof(ans));
    int mod = 1000;

    // queue to store all possible states
    queue<int> q;

    // initially push the start
    q.push(start % mod);

    // to reach start we require 0 steps
    ans[start] = 0;

    // till all states are visited
    while (!q.empty()) {

        // get the topmost element in the queue
        int top = q.front();

        // pop the topmost element
        q.pop();

        // if the topmost element is end
        if (top == end)
            return ans[end];

        // perform addition with all array elements
        for (int i = 0; i < n; i++) {
            int pushed = top + a[i];
            pushed = pushed % mod;
```

```
            // if not visited, then push it to queue
            if (ans[pushed] == -1) {
                ans[pushed] = ans[top] + 1;
                q.push(pushed);
            }
        }
    }
    return -1;
}

// Driver Code
int main()
{
    int start = 998, end = 2;
    int a[] = { 2, 1, 3 };
    int n = sizeof(a) / sizeof(a[0]);

    // Calling function
    cout << minimumAdditions(start, end, a, n);
    return 0;
}
```

Output:

```
2
```

Time Complexity: O(N)

Source

https://www.geeksforgeeks.org/sudo-placement1-3-final-destination/

Chapter 74

Sum of minimum and maximum elements of all subarrays of size k.

Sum of minimum and maximum elements of all subarrays of size k. - GeeksforGeeks

Given an array of both positive and negative integers, the task is to compute sum of minimum and maximum elements of all sub-array of size k.

Examples:

```
Input : arr[] = {2, 5, -1, 7, -3, -1, -2}
        K = 4
Output : 18
Explanation : Subarrays of size 4 are :
     {2, 5, -1, 7},   min + max = -1 + 7 = 6
     {5, -1, 7, -3},  min + max = -3 + 7 = 4
     {-1, 7, -3, -1}, min + max = -3 + 7 = 4
     {7, -3, -1, -2}, min + max = -3 + 7 = 4
     Sum of all min & max = 6 + 4 + 4 + 4
                          = 18
```

This problem is mainly an extension of below problem.
Maximum of all subarrays of size k

Method 1 (Simple)
Run two loops to generate all subarrays of size k and find maximum and minimum values. Finally return sum of all maximum and minimum elements.

Time taken by this solution is O(nk).

Method 2 (Efficient using Dequeue)
The idea is to use Dequeue data structure and sliding window concept. We create two

empty double ended queues of size k (‘S’ , ‘G’) that only store indexes of elements of current window that are not useless. An element is useless if it can not be maximum or minimum of next subarrays.

```
 a) In deque 'G', we maintain decreasing order of
    values from front to rear
 b) In deque 'S', we maintain increasing order of
    values from front to rear

1) First window size K
  1.1) For deque 'G', if current element is greater
       than rear end element, we remove rear while
       current is greater.
  1.2) For deque 'S', if current element is smaller
       than rear end element, we just pop it while
       current is smaller.
  1.3) insert current element in both deque 'G' 'S'

2) After step 1, front of 'G' contains maximum element
   of first window and front of 'S' contains minimum
   element of first window. Remaining elements of G
   and S may store maximum/minimum for subsequent
   windows.

3) After that we do traversal for rest array elements.
  3.1) Front element of deque 'G' is greatest and 'S'
       is smallest element of previous window
  3.2) Remove all elements which are out of this
       window [remove element at front of queue ]
  3.3) Repeat steps 1.1 , 1.2 ,1.3

4) Return sum of minimum and maximum element of all
   sub-array size k.
```

Below is c++ implementation of above idea

```
 // C++ program to find sum of all minimum and maximum
// elements Of Sub-array Size k.
#include<bits/stdc++.h>
using namespace std;

// Returns sum of min and max element of all subarrays
// of size k
int SumOfKsubArray(int arr[] , int n , int k)
{
    int sum = 0;  // Initialize result
```

```
// The queue will store indexes of useful elements
// in every window
// In deque 'G' we maintain decreasing order of
// values from front to rear
// In deque 'S' we  maintain increasing order of
// values from front to rear
deque< int > S(k), G(k);

// Process first window of size K
int i = 0;
for (i = 0; i < k; i++)
{
    // Remove all previous greater elements
    // that are useless.
    while ( (!S.empty()) && arr[S.back()] >= arr[i])
        S.pop_back(); // Remove from rear

    // Remove all previous smaller that are elements
    // are useless.
    while ( (!G.empty()) && arr[G.back()] <= arr[i])
        G.pop_back(); // Remove from rear

    // Add current element at rear of both deque
    G.push_back(i);
    S.push_back(i);
}

// Process rest of the Array elements
for (  ; i < n; i++ )
{
    // Element at the front of the deque 'G' & 'S'
    // is the largest and smallest
    // element of previous window respectively
    sum += arr[S.front()] + arr[G.front()];

    // Remove all elements which are out of this
    // window
    while ( !S.empty() && S.front() <= i - k)
        S.pop_front();
    while ( !G.empty() && G.front() <= i - k)
        G.pop_front();

    // remove all previous greater element that are
    // useless
    while ( (!S.empty()) && arr[S.back()] >= arr[i])
        S.pop_back(); // Remove from rear
```

```
        // remove all previous smaller that are elements
        // are useless
        while ( (!G.empty()) && arr[G.back()] <= arr[i])
            G.pop_back(); // Remove from rear

        // Add current element at rear of both deque
        G.push_back(i);
        S.push_back(i);
    }

    // Sum of minimum and maximum element of last window
    sum += arr[S.front()] + arr[G.front()];

    return sum;
}

// Driver program to test above functions
int main()
{
    int arr[] = {2, 5, -1, 7, -3, -1, -2} ;
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    cout << SumOfKsubArray(arr, n, k) ;
    return 0;
}
```

Output:

```
16
```

Time Complexity: O(n)

Source

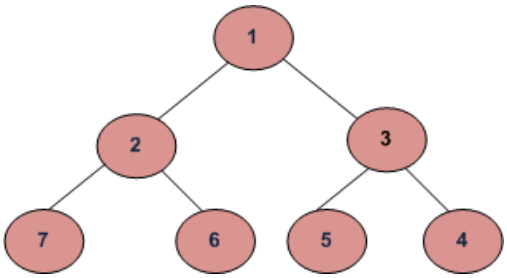
https://www.geeksforgeeks.org/sum-minimum-maximum-elements-subarrays-size-k/

Chapter 75

Zig Zag Level order traversal of a tree using single queue

Zig Zag Level order traversal of a tree using single queue - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



We have discussed naive approach and two stack based approach in Level Order with recursion and multiple stacks

The idea behind this approach is first we have to take a queue, a direction flag and a separation flag which is NULL

1. Insert the root element into the queue and again insert NULL into the queue.
2. For every element in the queue insert its child nodes.
3. If a NULL is encountered then check the direction to traverse the particular level is left to right or right to left. If it's an even level then traverse from left to right otherwise traverse the tree in right to level order i.e., from the front to the previous front i.e., from the current NULL to to the last NULL that has been visited. This continues till the last level then there the loop breaks and we print what is left (that has not printed) by checking the direction to print.

Following is the C++ implementation of the explanation

C++

```
 // C++ program to print level order traversal
// in spiral form using a single dequeue
#include <bits/stdc++.h>

struct Node {
    int data;
    struct Node *left, *right;
};

// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// function to print the level order traversal
void levelOrder(struct Node* root, int n)
{
    // We can just take the size as H+N which
    // implies the height of the tree with the
    // size of the tree
    struct Node* queue[2 * n];
    int top = -1;
    int front = 1;
    queue[++top] = NULL;
    queue[++top] = root;
    queue[++top] = NULL;

    // struct Node* t=root;
    int prevFront = 0, count = 1;
    while (1) {

        struct Node* curr = queue[front];

        // A level separator found
        if (curr == NULL) {

            // If this is the only item in dequeue
            if (front == top)
                break;

            // Else print contents of previous level
            // according to count
```

```
            else {
                if (count % 2 == 0) {
                    for (int i = prevFront + 1; i < front; i++)
                        printf("%d ", queue[i]->data);
                }
                else {
                    for (int i = front - 1; i > prevFront; i--)
                        printf("%d ", queue[i]->data);
                }

                prevFront = front;
                count++;
                front++;

                // Insert a new level separator
                queue[++top] = NULL;

                continue;
            }
        }

        if (curr->left != NULL)
            queue[++top] = curr->left;
        if (curr->right != NULL)
            queue[++top] = curr->right;
        front++;
    }

    if (count % 2 == 0) {
        for (int i = prevFront + 1; i < top; i++)
            printf("%d ", queue[i]->data);
    }
    else {
        for (int i = top - 1; i > prevFront; i--)
            printf("%d ", queue[i]->data);
    }
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
```

```
    levelOrder(root, 7);

    return 0;
}
```

Output:

```
1 2 3 4 5 6 7
```

Time Complexity: O(n)
Auxiliary Space : O(2*n) = O(n)

Source

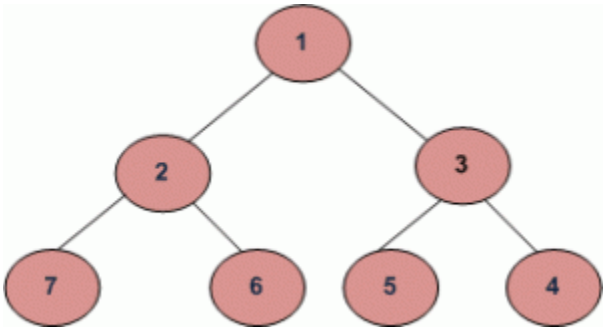
https://www.geeksforgeeks.org/zig-zag-level-order-traversal-of-a-tree-using-single-queue/

Chapter 76

ZigZag Tree Traversal

ZigZag Tree Traversal - GeeksforGeeks

Write a function to print ZigZag order traversal of a binary tree. For the below binary tree the zigzag order traversal will be **1 3 2 7 6 5 4**



This problem can be solved using two stacks. Assume the two stacks are current: **currentlevel and nextlevel.** We would also need a variable to keep track of the current level order(whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)

Below is the implementation of the above approach:

C++

```
 // C++ implementation of a O(n) time method for
// Zigzag order traversal
#include <iostream>
```

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

// Binary Tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// function to print the zigzag traversal
void zizagtraversal(struct Node* root)
{
    // if null then return
    if (!root)
        return;

    // declare two stacks
    stack<struct Node*> currentlevel;
    stack<struct Node*> nextlevel;

    // push the root
    currentlevel.push(root);

    // check if stack is empty
    bool lefttoright = true;
    while (!currentlevel.empty()) {

        // pop out of stack
        struct Node* temp = currentlevel.top();
        currentlevel.pop();

        // if not null
        if (temp) {

            // print the data in it
            cout << temp->data << " ";

            // store data according to current
            // order.
            if (lefttoright) {
                if (temp->left)
                    nextlevel.push(temp->left);
                if (temp->right)
                    nextlevel.push(temp->right);
            }
            else {
                if (temp->right)
                    nextlevel.push(temp->right);
```

```
                if (temp->left)
                    nextlevel.push(temp->left);
            }
        }

        if (currentlevel.empty()) {
            lefttoright = !lefttoright;
            swap(currentlevel, nextlevel);
        }
    }
}

// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// driver program to test the above function
int main()
{
    // create tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "ZigZag Order traversal of binary tree is \n";

    zizagtraversal(root);

    return 0;
}
```

Java

```
 // Java implementation of a O(n) time
// method for Zigzag order traversal
import java.util.*;

// Binary Tree node
class Node
{
```

```
int data;
Node leftChild;
Node rightChild;
Node(int data)
{
    this.data = data;
}
}

class BinaryTree {
Node rootNode;

// function to print the
// zigzag traversal
void printZigZagTraversal() {

    // if null then return
    if (rootNode == null) {
    return;
    }

    // declare two stacks
    Stack<Node> currentLevel = new Stack<>();
    Stack<Node> nextLevel = new Stack<>();

    // push the root
    currentLevel.push(rootNode);
    boolean leftToRight = true;

    // check if stack is empty
    while (!currentLevel.isEmpty()) {

    // pop out of stack
    Node node = currentLevel.pop();

    // print the data in it
    System.out.print(node.data + " ");

    // store data according to current
    // order.
    if (leftToRight) {
        if (node.leftChild != null) {
        nextLevel.push(node.leftChild);
        }

        if (node.rightChild != null) {
        nextLevel.push(node.rightChild);
        }
```

```
    }
    else {
        if (node.rightChild != null) {
        nextLevel.push(node.rightChild);
        }

        if (node.leftChild != null) {
        nextLevel.push(node.leftChild);
        }
    }

    if (currentLevel.isEmpty()) {
        leftToRight = !leftToRight;
        Stack<Node> temp = currentLevel;
        currentLevel = nextLevel;
        nextLevel = temp;
    }
    }
}
}

public class zigZagTreeTraversal {

// driver program to test the above function
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.rootNode = new Node(1);
    tree.rootNode.leftChild = new Node(2);
    tree.rootNode.rightChild = new Node(3);
    tree.rootNode.leftChild.leftChild = new Node(7);
    tree.rootNode.leftChild.rightChild = new Node(6);
    tree.rootNode.rightChild.leftChild = new Node(5);
    tree.rootNode.rightChild.rightChild = new Node(4);

    System.out.println("ZigZag Order traversal of binary tree is");
    tree.printZigZagTraversal();
}
}

// This Code is contributed by Harikrishnan Rajan.
```

Python3

```
 # Python Program to print zigzag traversal
# of binary tree

# Binary tree node
```

```
class Node:
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = self.right = None


# function to print zigzag traversal of
# binary tree
def zizagtraversal(root):

    # Base Case
    if root is None:
        return

    # Create two stacks to store current
    # and next level
    currentLevel = []
    nextLevel = []

    # if ltr is true push nodes from
    # left to right otherwise from
    # right to left
    ltr = True

    # append root to currentlevel stack
    currentLevel.append(root)

    # Check if stack is empty
    while len(currentLevel) > 0:
        # pop from stack
        temp = currentLevel.pop(-1)
        # print the data
        print(temp.data, " ", end="")

        if ltr:
            # if ltr is true push left
            # before right
            if temp.left:
                nextLevel.append(temp.left)
            if temp.right:
                nextLevel.append(temp.right)
        else:
            # else push right before left
            if temp.right:
                nextLevel.append(temp.right)
            if temp.left:
                nextLevel.append(temp.left)
```

```
        if len(currentLevel) == 0:
            # reverse ltr to push node in
            # opposite order
            ltr = not ltr
            # swapping of stacks
            currentLevel, nextLevel = nextLevel, currentLevel


# Driver program to check above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(7)
root.left.right = Node(6)
root.right.left = Node(5)
root.right.right = Node(4)
print("Zigzag Order traversal of binary tree is")
zizagtraversal(root)

# This code is contributed by Shweta Singh
```

Output:

```
ZigZag Order traversal of binary tree is
1 3 2 7 6 5 4
```

Time Complexity: O(n)
Space Complexity: O(n)+(n)=O(n)

Improved By : shweta44

Source

https://www.geeksforgeeks.org/zigzag-tree-traversal/