INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

van Emde Boas Tree with application to Prim's and compare wrt Binomial Heap and AVL.

Nitish Kumar Dwivedi (2018201068)
Dhawal Jain (2018201065)

1 Introduction

A Van Emde Boas tree is a tree data structure which implements an associative array with m-bit integer keys. It performs all operations in O(log m) time, or equivalently in O(log log M) time, where M = 2^m is the maximum number of elements that can be stored in the tree.

1.1 Set of Operations

Given a set S of elements such that the elements are taken from universe { 0, 1, ... u-1 } , perform following operations efficiently.

- insert(x) : Adds an item x to the set+ S.
- isEmpty() : Returns true if S is empty, else false.
- find(x) : Returns true if x is present in S, else false.
- delete(x) : Delete an item x from S.
- max() : Returns maximum value from S.
- min() : Returns minimum value from S.
- successor(x) : Returns the smallest value in S which is greater than x.

There are several ways to perform the above operation such as :

- AVL Tree
- Binomial Heap
- Van Emde Boas Tree

Let us analyze all the above operation corresponding to each data structure listed above.

1.1.1 AVL (Adelson-Velsky and Landis) Tree

An AVL Tree is a self-balancing binary search tree. It was the first such data structure to be invented.
In an AVL Tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

- insert(x) : O(log n).
- isEmpty() : O(1)
- find(x) : O(log n).

- delete(x) : O(log n).
- max() : O(log n).
- min() : O(log n).

1.1.2 BINOMIAL HEAP

Binomial heap is a heap similar to a binary heap but also supports quick merging of two heaps. This is achieved by using a special tree structure. It is important as an implementation of the mergeable heap abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

- insert(x) : O(1).
- isEmpty() : O(1)
- findmin(x) : O(log n).
- deletemin(x) : O(log n).
- union(x) : O(log n).

2 Van Emde Boas Tree

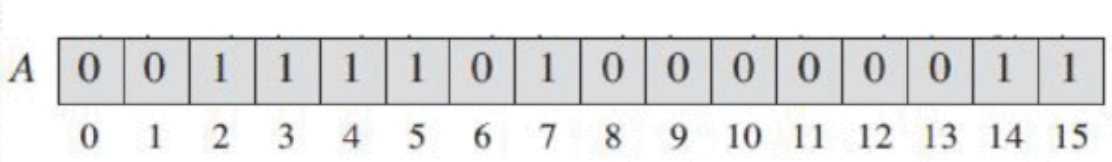
In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- Direct Addressing.
- Superimposing a binary tree structure.
- Superimposing a tree of constant height.

2.1 Direct Addressing

The direct-addressing stores the dynamic set as a bit vector.

- To store a dynamic set of values from the universe { 0, 1, ..., u-1 } we maintain an array A[0 .. u-1] of u bits.
- Entry A[x] holds 1 if the value x is in the dynamic set and 0 otherwise.
- INSERT, DELETE and MEMBER operations can be performed in O(1)time with this bit vector.
- MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR can take O(u) in the worst case since we might have to scan through O(u) elements.



2.2 Superimposing a binary tree structure.

- One can short-cut long scans in the bit vector by superimposing a binary tree on the top of it.
- Entries of the bit vector form the leaves of the binary tree.
- Each internal node contain 1 if and only if in its subtree contains 1.
- Bit stored in an internal node is the logical-or of its children.

MINIMUM:

- Algorithm: start at the root and head down toward the leaves, always taking the leftmost node containing 1.
- Time complexity: O(log u)

MAXIMUM:

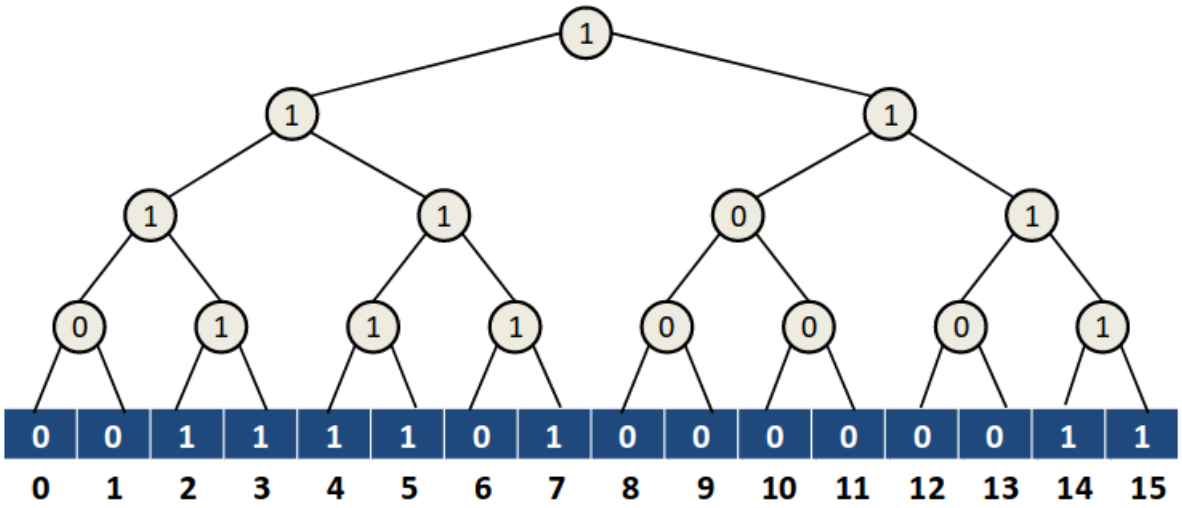
- Algorithm: start at the root and head down toward the leaves, always taking the rightmost node containing 1.
- Time complexity: O(log u)

PREDECESSOR (A, x) :

- Start at the leaf indexed by x and head up toward the root until we enter a node from the right and this node has a 1 in its left child z.
- Head down through node z, always taking the rightmost node containing a 1.
- Time complexity: O(log u).

SUCCESSOR (A, x) :

- Start at the leaf indexed by x and head up toward the root until we enter a node from the left and this node has a 1 in its right child z.
- Head down through node z, always taking the leftmost node containing a 1.
- Time complexity: O(log u)



Summary about above approach:

- This approach is only marginally better than just using a balanced search tree.
- MEMBER operation can be performed in O(1) time, what is better than O(log n) for BST.
- If the number of elements n is much smaller than size of the universe u than BST would be faster for all the other operations.

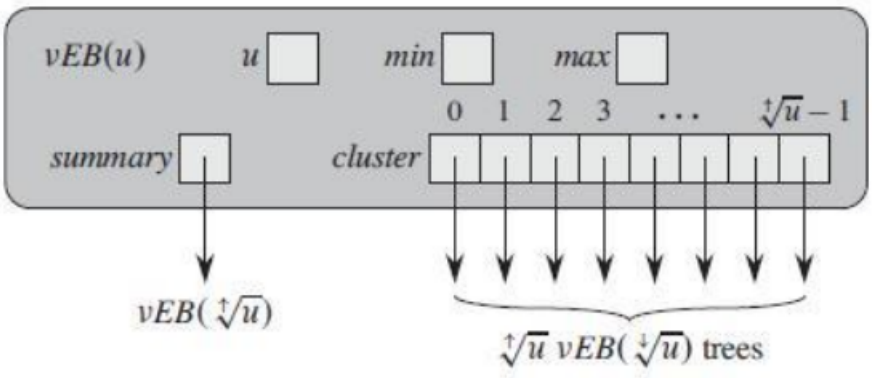
hyperref Prototype of approach that corresponds to the desirable complexity of O(log log n).

- The main idea is superimposing a tree of variable degree $u^{1/k}$.
- Starting with an universe of size u we make structures of $u^{1/2}$ items, which themselves contain structures of $u^{1/4}$ items, which hold structures of $u^{1/8}$ items...
- This subdivision process stops when there are only 2 items.
- For simplicity we assume that for some integer k.

the vEB has two main attributes:

- min stores the minimum element in the vEB.
- max stores the maximum element in the vEB.
- The attribute summary points to a vEB($u^{1/2}$) tree.
- The array cluster points to $u^{1/2}$ vEB($u^{1/2}$) trees.
- Important: The element stored in min does no appear in any of the recursive vEB trees that the cluster array points to.

Structure of vEB Tree Node:



The elements stored in a vEB(u) tree V.min plus all the elements stored in the vEB($u^{1/2}$) trees pointed to by V.cluster[0 .. ($u^{1/2}$) - 1]

- When a vEB contains two or more elements the element stored in min does not appear in any of the clusters but the stored in max does (unless it is equal to min).
- Since the basis size is 2, vEB(2) does not need the array A of the corresponding vEB. The attributes min and max can be used for it.
- Important: In the vEB, all the elements can be determined from the min and max attributes.

Complexities of Respective Functions:

- minimum(V,x) : O(1)
- maximum(V,x) : O(1)
- member(V,x) : O(log log u)
- successor(V,x) : O(log log u)
- insert(V,x) : O(log log u)
- delete(V,x) : O(log log u)

3 PRIM'S ALGORITHM

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.
The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

3.1 IMPLEMENTATION USING AVL TREE

We are using the following methods while Implementing Prim's Algorithm using AVL.

- The structure of AVL Tree node consists of following items:
  - data : Weight of Edge
  - *left : left subtree pointer
  - *right : right subtree pointer
  - height : height of that node
  - vector<pair<int,int> > describing all the vertex pairs that corresponds to the weight stored in data.
- Following are the methods we are using while implementing AVL Tree for Prim's:
  - Kth Minimum : Extracting the minimum weight value from the AVL Tree. Complexity = O(log n).
  - delete : Deletes the node from the AVL Tree. Complexity = O(log n).
  - insert : Insert the node in the AVL Tree. Complexity = O(log n).
- During the whole process, we ensured that the crux of Prim's Algorithm remains intact.

3.2 IMPLEMENTATION USING BINOMIAL HEAP

We are using the following methods while Implementing Prim's Algorithm using Binomial Heap.

- The structure of Binomial heap node consists of following items:
  - data : Weight of Edge
  - *left : left subtree pointer
  - *right : right subtree pointer
  - degree : height of the heap
- Following are the methods we are using while implementing Binomial Heap for Prim's:
  - Getmin : Extracting the minimum weight value from the Binomial Heap. Complexity = O(log n).

  - ExtractMin : Deletes the node from the Binomial Heap. Complexity = O(log n).
  - insert : Insert the node in the Binomial Heap. Complexity = O(1).
  - merge : merges two heaps. Complexity = O(log n)
- During the whole process, we ensure that there is only atmost one binomial tree with any degree.
- During the whole process, we ensured that the crux of Prim's Algorithm remains intact.

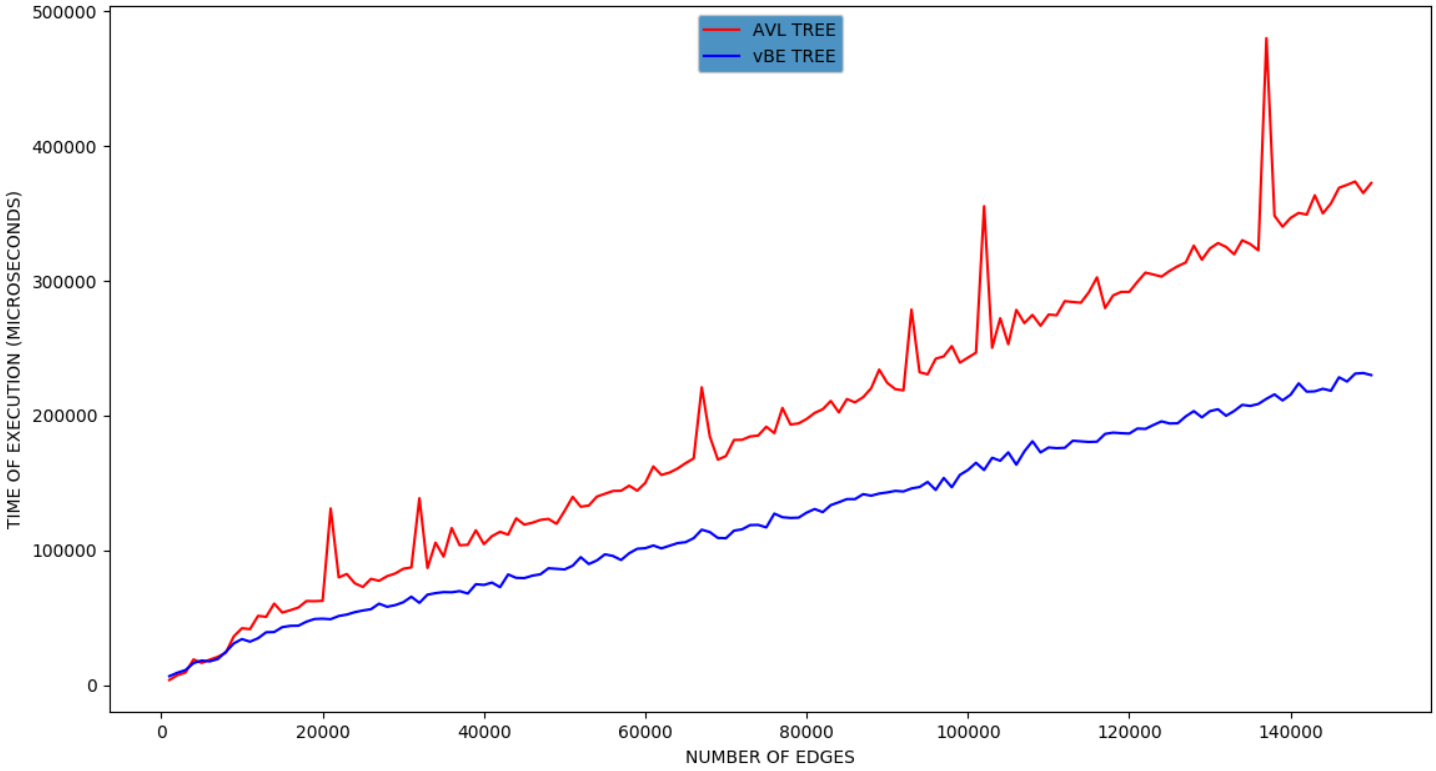
3.3 IMPLEMENTATION USING vAN EMDE BOAS TREE

We are using the following methods while Implementing Prim's Algorithm using Van Emde Boas Tree.

- The structure of Binomial heap node consists of following items:
  - U : Universe Size
  - minimum-value : minimum of each cluster
  - maximum-value : maximum of each cluster
  - *summary : Summary of entire cluster
  - **cluster : maintains the pointers to the nodes of vEB Tree.
- Following are the methods we are using while implementing vEB Tree for Prim's:
  - getmin() : Extracting the minimum weight value from the vEB Tree. Complexity = O(1).
  - getmax() : Extracting the minimum weight value from the vEB Tree. Complexity = O(1).
  - delete : Deletes the node from the vEB Tree. Complexity = O(log log n).
  - insert : Insert the node in the vEB Tree. Complexity = O(log log n).
- We are using an auxillary array to maintain the frequency of the weights.
- During the whole process, we ensured that the crux of Prim's Algorithm remains intact.

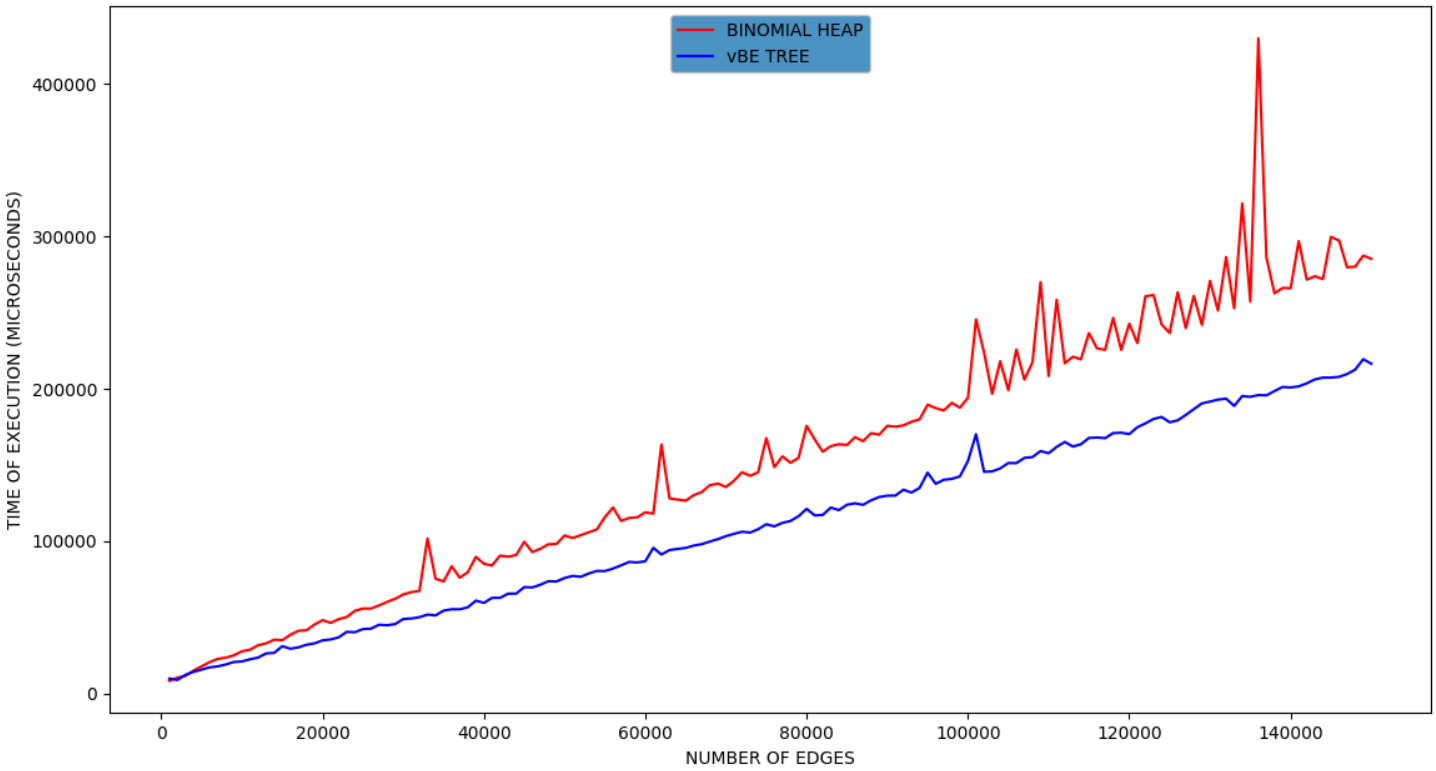
4 COMPARISION OF PERFORMANCE OF DATA STRUCTURES

4.1 vAN EMDE BOAS TREE V/S AVL TREE

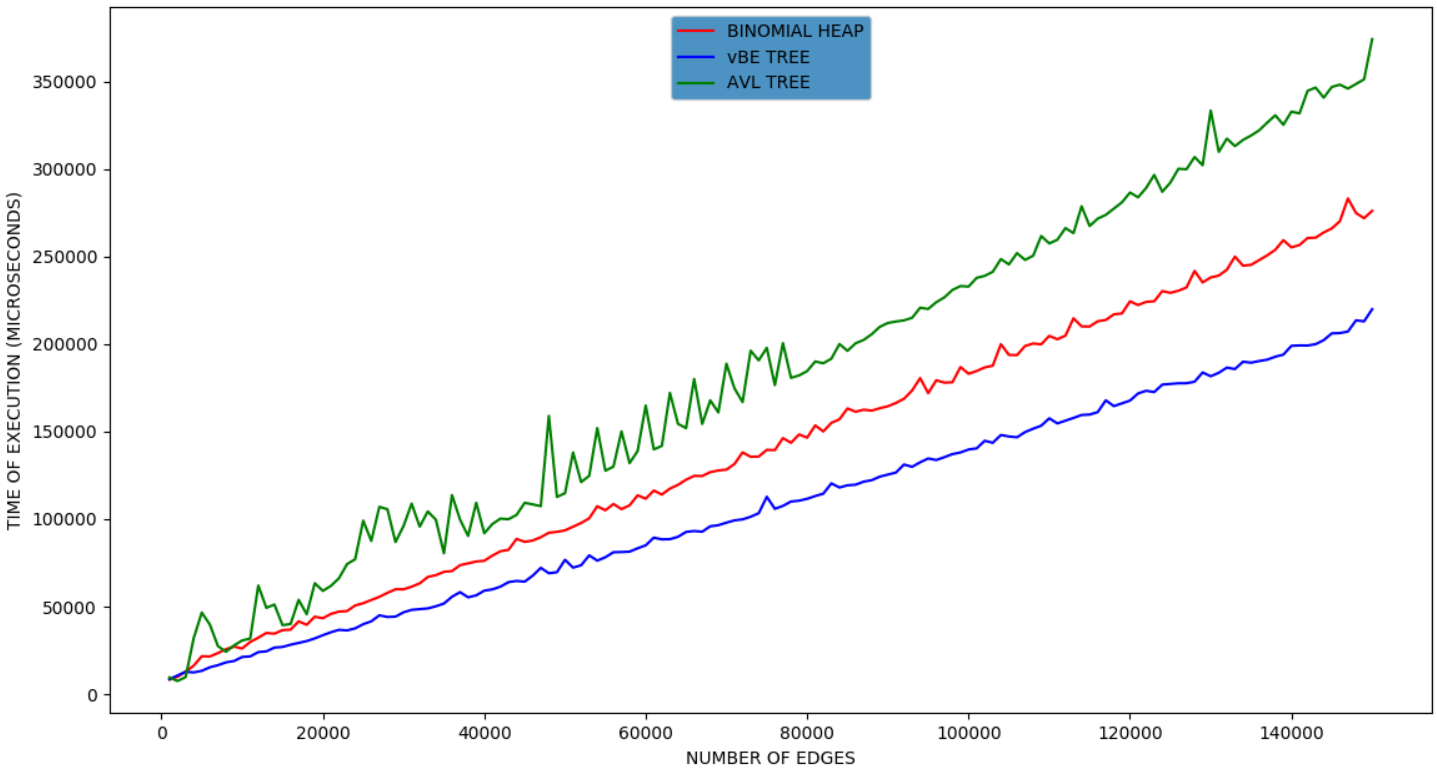


Initially the performance of van Emde Boas tree is poor in comparison with AVL because of the constant factor involved in van Emde Boas build which affects its performance in smaller number of vertices in graph. As the number of vertices and Edges increases, its performance with respect of AVL Tree increases significantly.

4.2 vAN EMDE BOAS TREE V/S BINOMIAL HEAP



van Emde Boas Tree performance WRT Binomial Heap is also poor in the beginning because of the constant factor involved in van Emde Boas build which affects its performance in smaller number of vertices in graph. As the number of vertices and Edges increases, its performance with respect of Binomial Heap increases significantly.



The collective comparision of the three Data Structures on Prim's Algorithm is shown above.

For implementation details, refer:

https://github.com/nitishkd/VAN-EMDE-BOAS-TREE

5 USER MANUAL

- To generate random Test Cases, use python script generate.py. It will generate 150 test cases to execute in different files.
  - python3 gererate.py
- Now, we will use another bash script to run these 150 test cases on our program and record their execution time in a external text file named as outputavl.txt , outputvb.txt and outputbino.txt . (make sure that you have proper permissions to run this script).
  - ./script.sh
- Now, use test.py file to plot the graph.
  - python3 test.py

5.1 COMPILATION AND EXECUTION OF INDIVIDUAL FILES

- AVL TREE:
  - To Compile: g++ AVL.cpp -o AVL
  - To Execute: ./AVL
  - Input Format:
    - Line 1 contains two integer values N and M, where N is number of Nodes and M is number of Edges.
    - Next M lines, each containing 3 integers X, Y, W where there is an Edge between X and Y with weight W.
  - Constraints:
    - $1 \leq N \leq 100000$
    - $1 \leq M \leq 1000000$
    - $0 \leq X, Y \leq N-1$
    - $1 \leq W \leq 65530$
  - NOTE: The graph should always be connected and should not result in a forest.
- BINOMIAL HEAP:
  - To Compile: g++ BINOMIAL.cpp -o BINOMIAL
  - To Execute: ./BINOMIAL
  - Input Format:
  - Input Format:
    - Line 1 contains two integer values N and M, where N is number of Nodes and M is number of Edges.

    - Next M lines, each containing 3 integers X, Y, W where there is an Edge between X and Y with weight W.
  - Constraints:
    - $1 \leq N \leq 100000$
    - $1 \leq M \leq 1000000$
    - $0 \leq X, Y \leq N-1$
    - $1 \leq W \leq 65530$
  - NOTE: The graph should always be connected and should not result in a forest.

- vEB Tree:
  - To Compile: g++ vEB.cpp -o vEB
  - To Execute: ./vEB
  - Input Format:
    - Line 1 contains two integer values N and M, where N is number of Nodes and M is number of Edges.
    - Next M lines, each containing 3 integers X, Y, W where there is an Edge between X and Y with weight W.
  - Constraints:
    - $1 \leq N \leq 100000$
    - $1 \leq M \leq 1000000$
    - $0 \leq X, Y \leq N-1$
    - $1 \leq W \leq 65530$
  - NOTE: The graph should always be connected and should not result in a forest.