Van Emde Boas Tree with application to Kruskal and Compare with Union Find and Fibonacci Heap

Team Member :

Apurvi Mansinghka (2019201093)

Himanshu Kumar (2019201094)

Problem Statement

Study and implement Van Emde Boas Tree data structure with application to kruskal algorithm (minimum spanning tree algorithm) and comparison with other data structures Binary Heap and Fibonacci Heap.

Goals

- ➢ Comparative study of dictionary operations such as insertion, search, delete, extract minimum among different data structures.
- ➢ Achieve minimum time complexity to implement kruskal algorithm by using suitable data structure.

1.INTRODUCTION

1.1 Kruskal Algorithm

Kruskal algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.

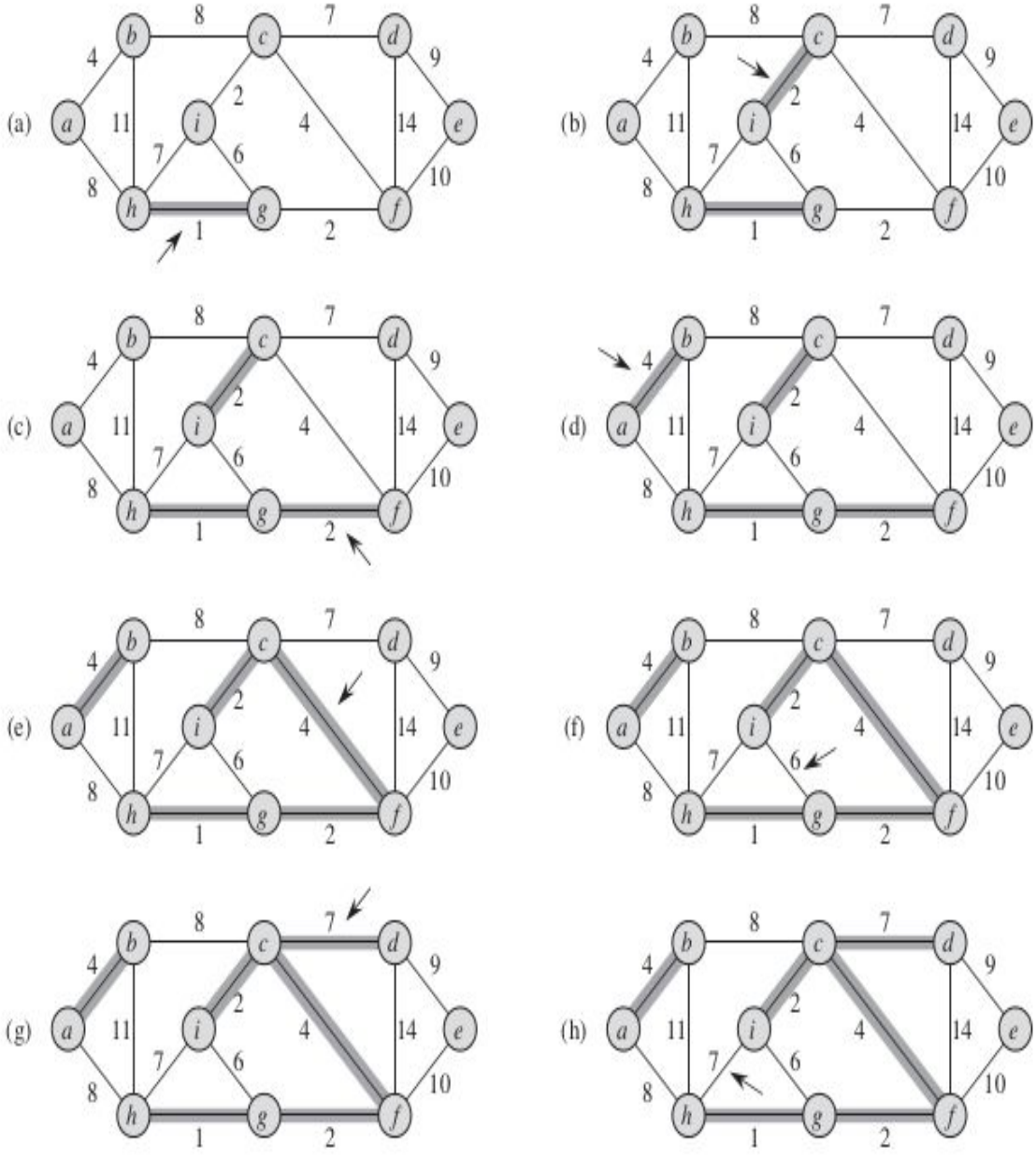
This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the WEIGHTED UNION procedure.

Algorithm:

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Example to show execution of Kruskal's Algorithm



1.2 Data Structures

1.2.1 Sorted and Unsorted Array

Unsorted Array

In case of unsorted array insertion of an element takes O(1), for E edges total time for insertion takes O(E) time. To perform extract minimum operation of kruskal's algorithm,finding and deleting minimum in unsorted array can take O(E). Therefore,

Worst and Average case : O(E^2)

Sorted Array

To reduce complexity of extract minimum operation one approach is to sort the array leading to O(1) time. Insertion of elements and sorting the array can result in O(E+ElogE) overall time complexity for implementing kruskal's algorithm.

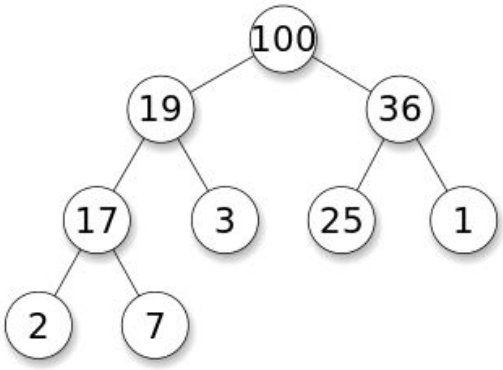
Overall time for Kruskal's Algorithm

All Cases : O(E+ElogE)

1.2.2 Binary Heap

A binary heap is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing dictionary operation such as delete min, adjust key and insertion. A binary heap is defined as a binary tree with two additional constraints:

- Shape property: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children.



Complexity Analysis of kruskal's algorithm using Binary Heap :

Building min Heap for E edges : O(E)

Extract minimum in Min Heap : O(logE)

In best case, V-1 minimum edges needs to be extracted to form minimum spanning tree. However, in the worst case all E edges might be extracted.

Best Case : O(E+VlogE)

Worst and Average Case : O(E+ElogE)

1.2.3 Fibonacci Heap

A Fibonacci heap is a specific implementation of the heap data structure that makes use of Fibonacci numbers. Fibonacci heaps are used to implement the priority queue element in Kruskal's and Dijkstra's algorithm, giving the algorithm a very efficient running time.

Fibonacci heaps have a faster amortized running time than other heap types. Fibonacci heaps are similar to binomial heaps but Fibonacci heaps have a less rigid structure. Binomial heaps merge heaps immediately but Fibonacci heaps wait to merge until the extract-min function is called.

TIME COMPLEXITY COMPARISON

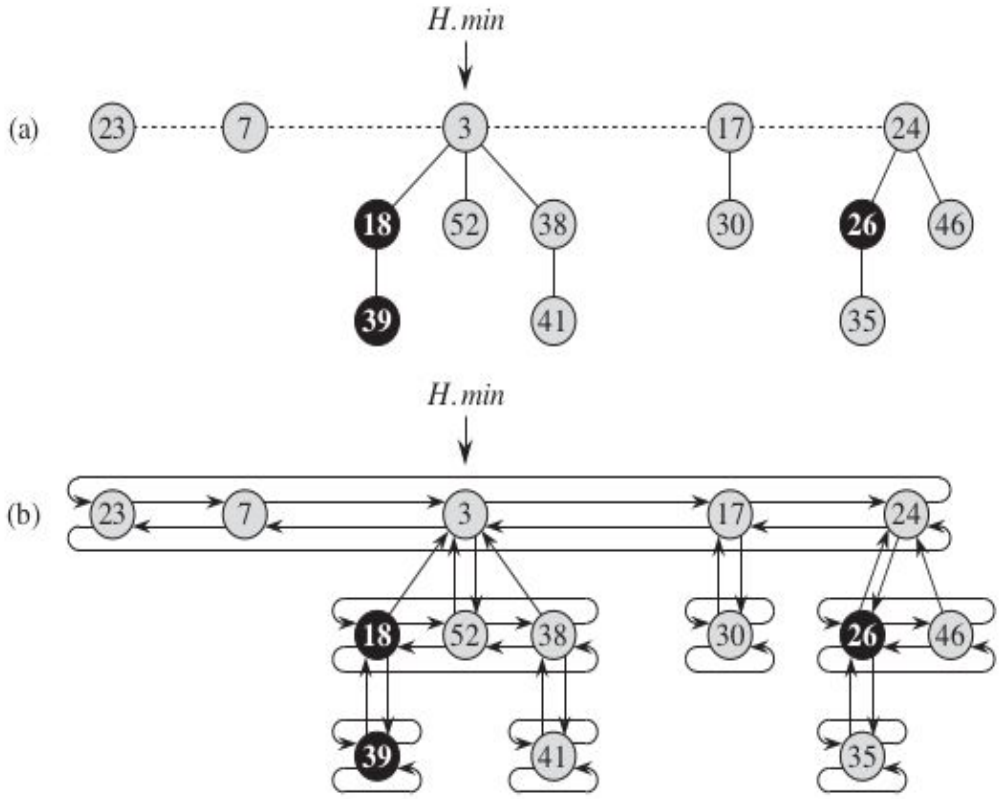
Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

A Fibonacci heap node has following attributes :

- X.degree : Number of children in the child list of node x.
- X.key : Stores the actual value.
- X.left & X.right : Pointers to the sibling nodes.
- X.parent : Pointer to the parent node.
- X.child : Each node points to any one child node of itself.
- X.mark : Indicates whether node x has lost a child since the last time x was made the child of another node.

MEMORY REPRESENTATION OF FIBONACCI HEAP

A Fibonacci Heap is a collection of rooted trees that are min-heap ordered ,i.e., each tree itself follows the basic heap and space property.



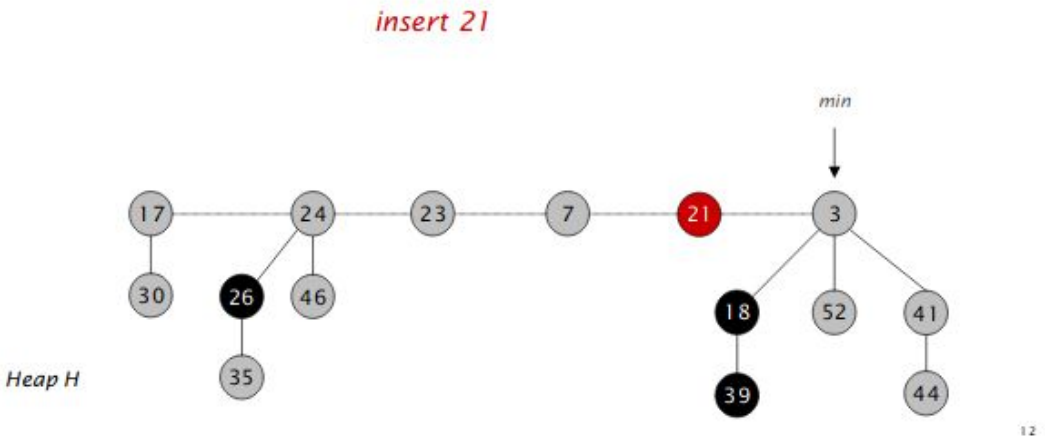
ROOT LIST : The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.

MINIMUM POINTER : H.min points to the minimum node in the root list and if required, gets updated on insertion or deletion of elements.

USED OPERATION ON FIBONACCI HEAP :

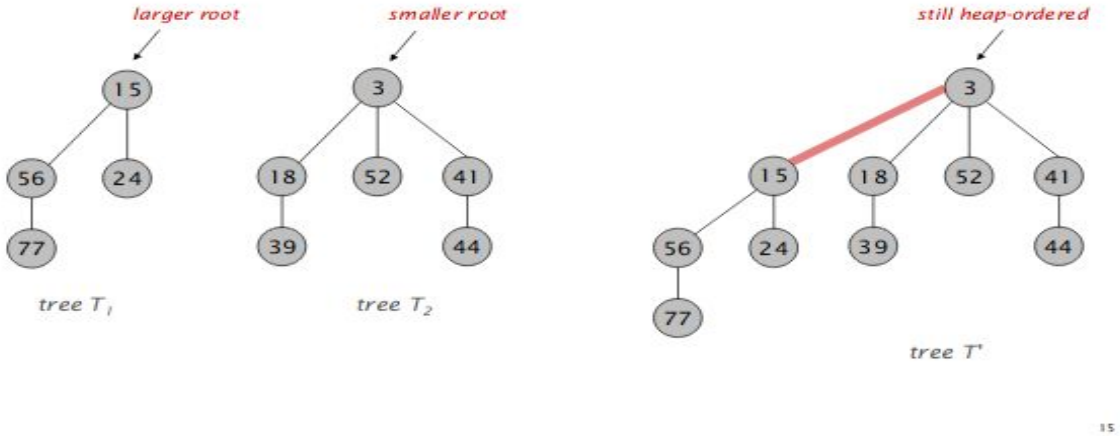
A. Insertion

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



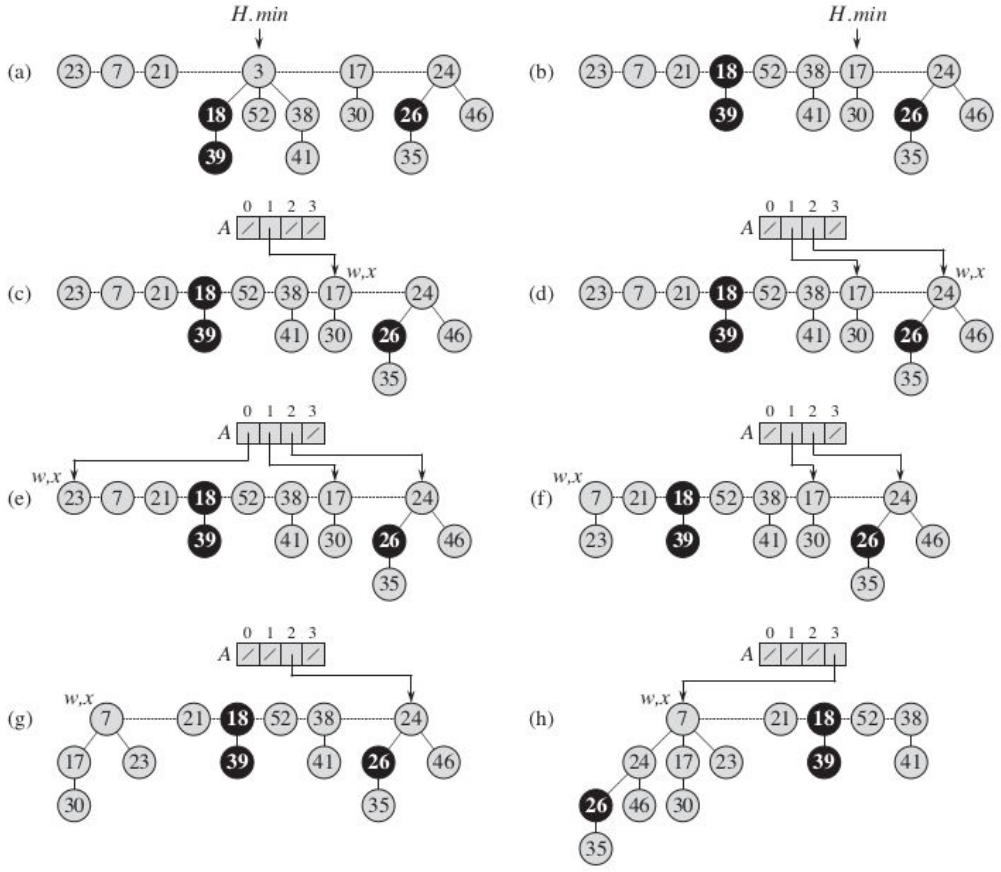
B. LINKING

Make larger root be a child of smaller root.



C. Delete Minimum

- Meld the child nodes of min pointer node to root list.
- Delete node referred by H.min.
- Update H.min pointer to point to next node.
- Consolidate trees so that no two trees have the same rank.



Complexity Analysis of Kruskal's Algorithm using Fibonacci Heap :

Insertion of E edges : O(E)

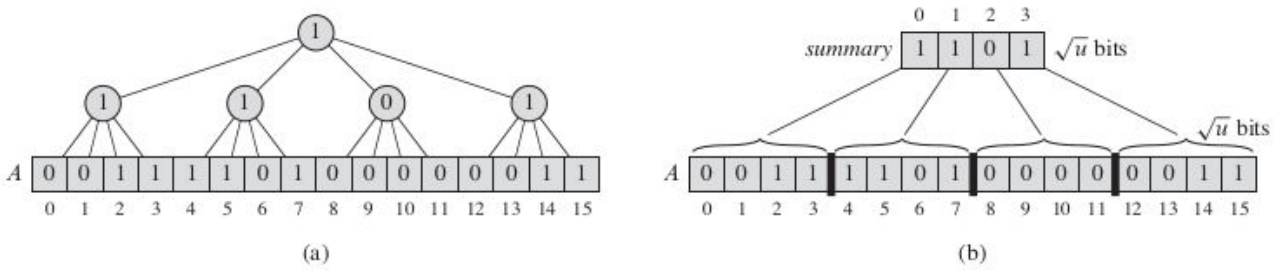
Deletion of a Minimum edge : O(logE)

Overall Complexity : O(E+ElogE)

1.2.3 Van Emde Boas Tree

Before going to VEB let us analyse different approaches to store dynamic data set and the complexity of implementing dictionary using these approaches .

Superimposing a tree of constant height on bit Vector



Let us assume that the size of the universe is u= 2^{2^k} for some integer k, so that u is an integer. We will superimpose a tree of degree $u^{1/2}$, so that the height of the tree will always remains 2 .

- To find the minimum (maximum) value, find the leftmost (rightmost) entry in summary that contains a 1, say summary[i], and then do a linear search within the ith cluster for the leftmost (rightmost) 1.
- To find the successor (predecessor) of x, first search to the right (left) within its cluster. If we find a 1, that position gives the result. Otherwise, search in the summary vector and find first occurrence of 1 and search that particular block.
- To delete the value x, let i= x/ $\sqrt{u}$. Set A[i] to 0 and then set summary[i] to the logical-or of the bits in the ith cluster.

  In each of the above operations, we search through at most two clusters of u bits plus the summary array, and so each operation takes O($\sqrt{u}$) time.

Recursive Approach

Now, we make the structure recursive, shrinking the universe size by the square root at each level of recursion. Starting with a universe of size u, we make structures holding $\sqrt{u}$ items, which themselves hold structures of $u^{1/4}$ items, which hold structures of $u^{1/8}$ items, and so on, down to a base size of 2 which will lead to the recursion relation :

$$T(u) = T(\sqrt{u}) + O(1)$$

Using master's theorem above recurrence results in

$T(u) = O(loglog(u))$ solution.

The cluster number in which x value resides will be given by $floor(x/\sqrt{u})$

$$high(x) = floor(x/\sqrt{u})$$

$$low(x) = x \bmod \sqrt{u}$$

$$index(x, y) = x\sqrt{u} + y$$

The function high(x) gives the most significant bits of x, producing the number of x's cluster. The function low(x) gives the least significant bits of x and provides x's position within its cluster. The function index(x,y) builds an element number from x and y, treating x as the most significant bits of the element number and y as the least significant bits.

Proto Van Emde Boas Structure

Using the above recursive approach we discuss our recursive data structure Proto Veb which will fail to achieve our goal of O(loglog(u)) time but serves as a basis for the vEB structure.

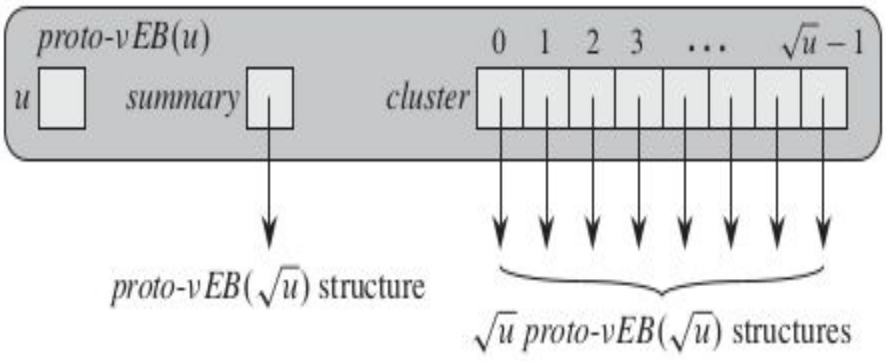
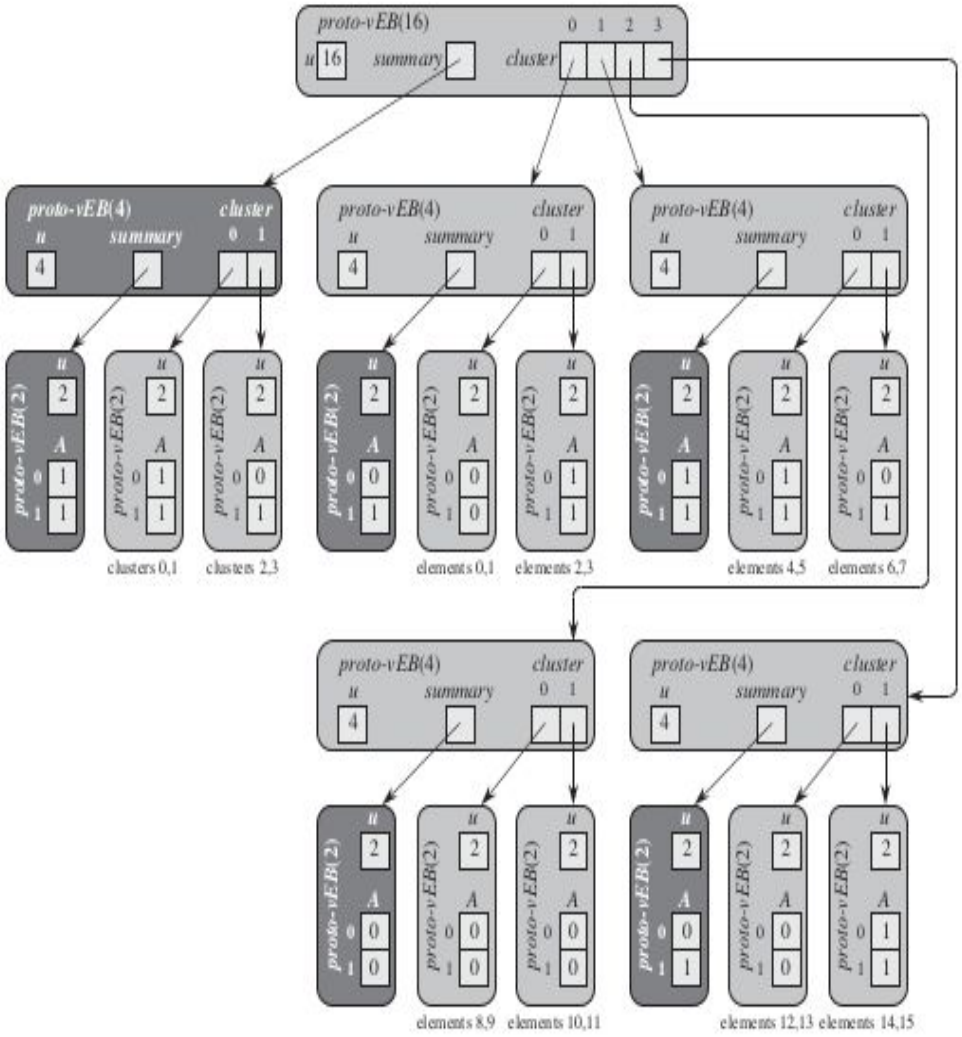


Diagram of proto-vEB structure

Attributes of the structure:

- *u* is the universe size of the bit vector. If u=2 then it is the base size which contains an array A[0..1] of two bits .
- A pointer name summary to a proto-vEB($\sqrt{u}$) structure.
- An array cluster [0.... $\sqrt{u}$ -1] of $\sqrt{u}$ pointers each pointing to some other proto-vEB($\sqrt{u}$) structure.



An example of proto-vEB(16)

proto-vEB(16) structure representing the set {2,3,4,5,7,14,15}. If the value i is in the proto-vEB structure pointed to by summary, then the ith cluster contains some value in the set being represented. u through As in the Tree of constant height, cluster[i] represents the values i $\sqrt{u}$ through (i+1)($\sqrt{u}$ -1) which form the ith cluster.

At the base level, the elements of the actual dynamic sets are stored in some of the proto-vEB(2) structures, and the remaining proto-vEB(2) structures store summary bits. Beneath each of the non-summary base structures, the figure indicates which bits it stores.

Complexity analysis on proto-vEB structure:

Insertion :

It involves two recursive insertion into summary and cluster which leads to the following recurrence relation.

T(u)=2T($\sqrt{u}$)+O(1))

Overall Insertion time becomes: O(logu))

Finding minimum :

Finding minimum also requires two recursive calls ,first to find minimum in summary and then in respective cluster.

T(u)=2T($\sqrt{u}$)+O(1)

Overall Insertion time becomes: O(logu)

Deletion:

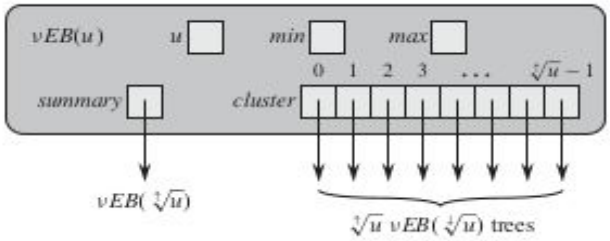
The DELETE operation is more complicated than insertion we cannot always reset the same summary bit to 0 when deleting. We need to determine whether any bit in the appropriate cluster is 1. As we have defined proto-vEB structures, we would have to examine all $\sqrt{u}$ bits within a cluster to determine whether any of them are 1.

T(n)=2T($\sqrt{u}$)+O($\sqrt{u}$)

Implementing kruskal require operations insertion and deleting minimum which would lead to a very high complexity using proto vEB, we improve the deletion and find minimum complexity using vEB structure.

Van Emde Boas Data Structure:

In case of proto-vEB we have to recurse multiple times to implement many operations due to which we fail to achieve O(loglog(u)) complexity . To reduce the need of some recursion, we store a little more information using vEB-structure.



A vEB tree contains two attributes not found in a proto-vEB structure:

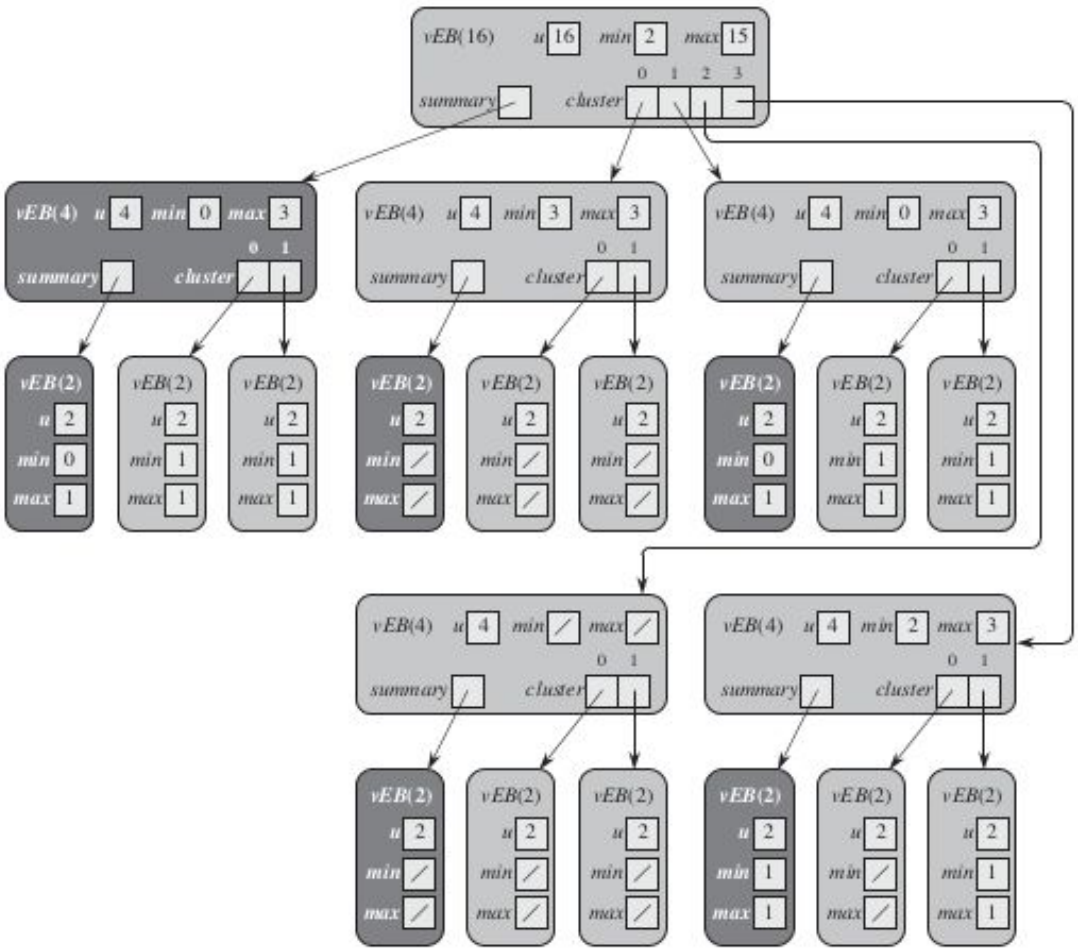
- Min stores the minimum element in the vEB tree, and
- Max stores the maximum element in the vEB tree.

The element stored in min does not appear in any of the recursive vEB trees that the cluster array points to.

Min and max attributes will help to reduce time in the following way:

1. For finding the min and max element we need not to recurse down in cluster or summary, hence constant time will be required.
2. For determining the no. of elements in the vEB tree we can use the min and max values. If min and max both NIL then vEB Tree is empty. If min and max are equal to each other and not NIL then vEB Tree has only one element otherwise the tree has two or more elements. The above determination can be done in constant time providing help in insert and delete operation.
3. If vEB tree is empty we can insert an element by just updating the min and max values in constant time. Similarly deletion of single element can be done in constant time by setting the min and max to null.

An example of vEB(16) that stores the set {2, 3, 4, 5, 7, 14, 15}.The value stored in the min attribute of a vEB tree does not appear in any of its clusters.



Operations in vEB :

A. Insertion :

proto-vEB structure makes two recursive call, one to insert element and another to update summary. vEB structure reduces this to one recursive call as follows:-

- If cluster of the element to be inserted already has some other element then there is no need to update summary structure because the cluster information must be already present in summary.
- If the cluster is empty then to insert element we only need to update min of both cluster and summary in constant time .

Time Complexity :

$T(u)=T(\sqrt{u})+O(1)$

$T(u)=O(loglogu)$

B. Finding minimum :

As the minimum element is present in the root itself. It can be accessed in O(1) time by using the min field of root vEB structure.

C. Deletion :

Deletion from base vEB(2)

- If max!=min then set max=min to the remaining element in vEB(2) and return.
- If max==min then set max=min=nil and return

Deletion in vEB(u)

- If max==min then set max=min=nil and delete entry from summary
- If element to be deleted(x) is equal to min then retrieve min from the node pointed by the cluster array of current node and recursively update the min of all the nodes with the min value retrieved from cluster array.
- Similarly if element to be deleted is max then we will update it recursively .

2. RESULTS

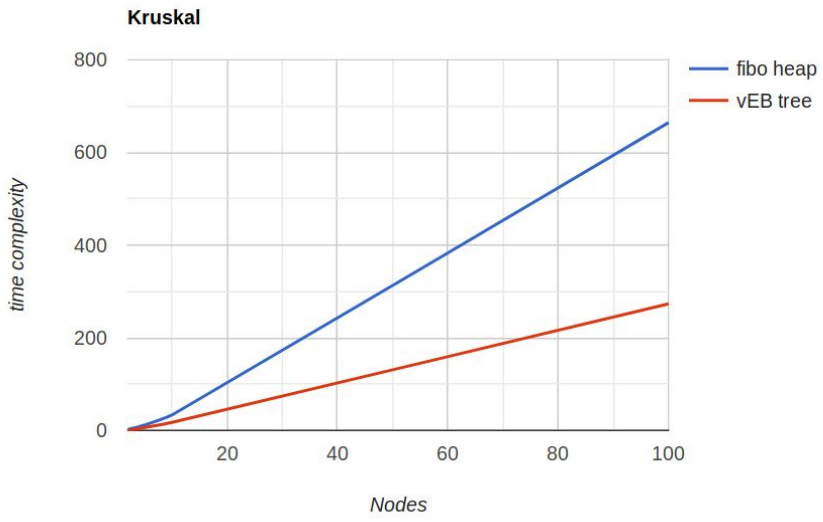
Kruskal algorithm require operations : Insertion, finding and deleting minimum.

E : Total Edges in graph G(V,E)

u : nearest power of 2>E

Operation	Binary Heap	Fibonacci Heap	vEB
E elements insertion(dynamic data set)	O(ElogE)	O(E)	O(Eloglog(u))
Finding Minimum	O(1)	O(1)	O(1)
Deleting minimum	O(logE)	O(logE)	O(loglog(u))
Overall complexity for Kruskal	O(ElogE)	Best Case: O(E+VlogE) Worst & Avg Case: O(E+ElogE)	O(Eloglog(u))

Time observation:



3.CODE

Language used for implementation : C++

GITHUB Repository Link:

https://github.com/HimanshuIIITH/Kruskal-s_Implementation

4.REFERENCES

- ***Introduction to Algorithms*** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- http://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/14/Small14.pdf
- https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf
- https://en.wikipedia.org/wiki/Fibonacci_heap
- https://en.wikipedia.org/wiki/Van_Emde_Boas_tree