van Emde Boas tree with application to Prim's Algorithm and comparison wrt RB tree and AVL tree

Akanksha Agarwal(2018201061)
Kratika Kothari(2018201060)

November 12, 2018

Abstract

In this project, we have studied van Emde Boas Tree(vEB Tree) and implementation of Prim's Algorithm on vEB tree. Further, we have compared the implementaion of prim's algorithm on vEB tree with AVL tree and Red-Black tree(RB tree). We have also discussed real time applications of vEB tree.

github repository: https://github.com/akanksha212/APS_Project.git

1 Introduction

van Embe Boas Tree
van Emde Boas data structure is a tree data structure which supports dynamic set operations in O(loglogu) time when we need to manipulate values in the range 0,1.. u-1. It was invented by a team led by Dutch computer scientist Peter van Emde Boas in 1975. A dynamic set refers to a set of values that support insertion and/or deletion of elements.
A Veb tree allows queries on such a set using the following operations:-

1. **Insert**
   Insert a key in the range 0,1.. u-1 in the tree where u is the universe size.
   Complexity: O(loglogu)

2. **Delete**
   Delete a key in the tree.
   Complexity: O(loglogu)

3. **Minimum**
   Find the minimum key in the tree.
   Complexity: O(1)

4. **Maximum**
   Find the maximum key in the tree.
   Complexity: O(1)

5. **Successor**
   Find the smallest key in the tree greater than a given value k.
   Complexity: O(loglogu)

6. **Predecessor**
   Find the largest key in the tree smaller than a given value k.
   Complexity: O(loglogu)

7. **Member**
   Find if a given value k is present in the tree.
   Complexity: O(loglogu)

In the context of the current task, we implemented insert, delete and minimum operations on the vEB tree.

Prims Algorithm
Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum.

AVL Tree
An AVL tree is a self-balancing binary search tree. 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.

Red-Black Tree
A Red-Black tree is a self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

2 Theory

How vEB tree works?
van Emde Boas tree is a recursive tree data structure. It divides the given range (universe) 0,1.. u-1 into blocks of size sqrt(n) which are called as clusters. Each cluster is then a vEB tree data structure of size sqrt(n). In addition, there is a summary structure that keeps track of which clusters are nonempty.
For the universe {0,1.. u-1} we define a vEB structure which is denoted by vEB(u) recursively as follows. Each vEB(u) structure V contains the following attributes:-

1. V.size, size of V.

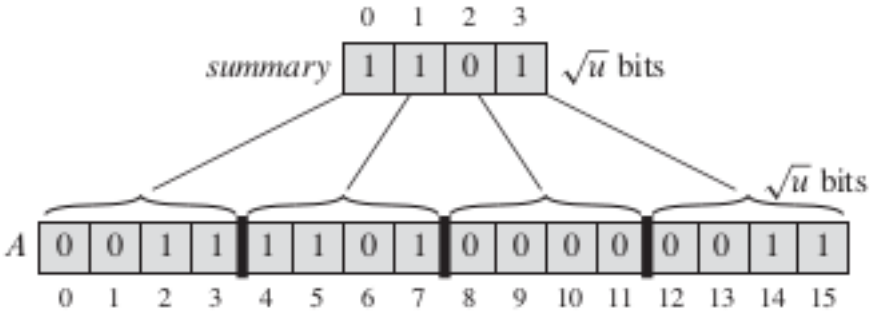
2. V.cluster, array of size sqrt(u) which contains pointers to vEB(sqrt(u)) structures. For performance reasons, the value stored in each entry of V.cluster will initially be NIL we will wait to build each cluster until we have to insert something into it.

3. V.summary, pointer to a a van Emde Boas structure of size sqrt(n) that keeps track of which clusters are nonempty. As with the entries of V.clusters we initially set V.summaryNIL.

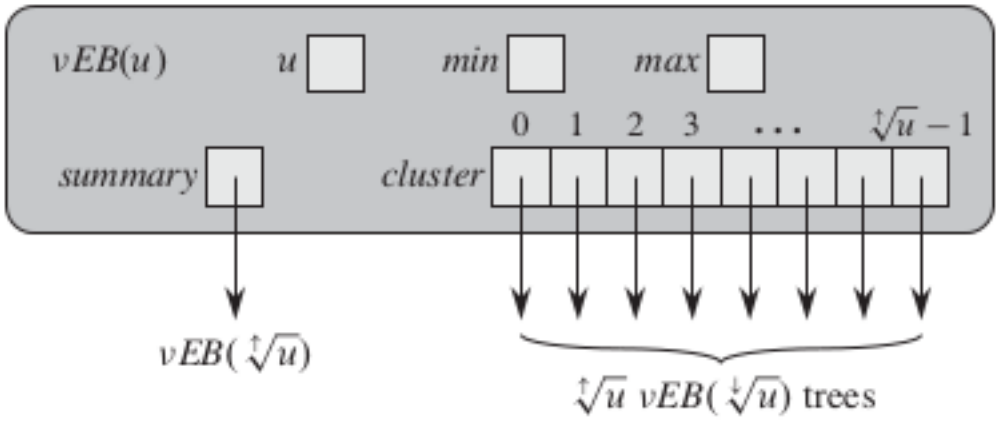
4. V.min, minimum element of V, or NIL if V is empty. The same min value does not appear in any other cluster that the tree points to.

5. V.max, maximum element of V, or NIL if V is empty.

A given value x resides in cluster number floor(x/sqrt(n)) where n = V.size. For example when V.size = 16, we will have sqrt(V.size) clusters each of size sqrt(V.size) i.e. 4. 7 will reside in floor(7/4)th cluster i.e. cluster number 1. Within the cluster, x appears in position (x)mod(sqrt(V.size)). Hence within 1st cluster 7 lies at position 7mod4 i.e. 3. Below we have a set of values 2,3,4,5,7,14,15.



The actual vEB tree does not store bits for absence or presence of values but the above figure gives an idea of the clusters and the summary structure. The actual vEB structure is as follows:-



Here the summary is a pointer to a vEB structure of size ceiling(sqrt(u)) and size of each cluster is floor(sqrt(u)) (Since u may not be an even power of 2).

Operations on vEB tree

We define two operations that aid insertion and deletion in the vEB tree.Suppose x be the key value to be inserted/deleted in the vEB tree structre V.

a) high(x) = ceiling(x/sqrt(V.size))
This gives the cluster number where x must be inserted.

b) low(x) = floor(x/sqrt(V.size))
This gives the position in the cluster where x must be inserted.

1. **Finding minimum**
Since we store the minimum value in each vEB structure, the minimum function just returns min attribute value.

2. **Insertion**
When we insert an element, we perform two tasks:

   a) Insert element in the cluster by updating min/max values.
   b) Update the cluster in the summary structure.

   When we insert an element, either the cluster that it goes into already has another element or it does not. If the cluster already has another element, then the cluster number is already in the summary, and so we do not need to make another call for that.
   If the cluster does not already have another element, then the element being inserted becomes the only element in the cluster, and we do not need to recurse to insert an element into an empty vEB tree.
   The key to the efficiency of this procedure is that inserting an element into an empty vEB tree takes O(1) time.
   Suppose we have to insert x in the vEB tree V.

I) If V.min is NULL, tree is empty.
Insert in an empty vEB tree by setting V.min=V.max=x.

II) If tree is not empty, check if x is smaller than V.min.
If yes, swap V.min and x.
Proceed by inserting the new x in the cluster high(x) since a min value exists only once in a vEB tree.
If the cluster high(x) was previously empty, insert high(x) in the V.summary.

III) If tree is not empty, check if x is greater than V.max.
If yes, set V.max =x.

IV) If tree is not empty and V.min¡x¡V.max then
Find the cluster x should exist in: (high(x)).
If the cluster is empty then update the cluster in the summary recursively and insert x in the empty tree using constant operations of updating min and max values.
If the cluster is not empty, we recurse in the cluster to insert x in the position within the cluster.

3. **Deletion:**
To delete an element, we will have to delete it from its cluster.
If the element to be deleted is min, then a new element becomes the min and now this element is to be deleted from the tree. If the cluster is empty now, then we remove this cluster from the summary.
After updating the summary, we might need to update max. If the deleted element was the maximum element, then summary-max to the number of the highest-numbered nonempty cluster.
If all the clusters are empty then the max is set to min, otherwise max is set to the maximum element in the highest numbered cluster.

I) If V.min is NULL, tree is empty.
Insert in an empty vEB tree by setting V.min=V.max=x.

II) If tree is not empty, check if x is smaller than V.min.
If yes, swap V.min and x.
Proceed by inserting the new x in the cluster high(x) since a min value exists only once in a vEB tree.
If the cluster high(x) was previously empty, insert high(x) in the V.summary.

III) If tree is not empty, check if x is greater than V.max.
If yes, set V.max =x.

IV) If tree is not empty and V.min¡x¡V.max then
Find the cluster x should exist in: (high(x)).
If the cluster is empty then update the cluster in the summary recursively and insert x in the empty tree using constant operations of updating min and max values.

If the cluster is not empty, we recurse in the cluster to insert x in the position within the cluster.

3 Implementation

Data structures used:

1. **Graph:** Graph is stored as a vector whose index is vertex number. At every index of this vector we are storing a vector of pairs in which each pair corresponds to an edge from this vertex. Pair has the other vertex as one value and weight as other. Also it holds pointer to the node in the tree corresponding to the vertex.

2. **vEB tree:** In order to find minimum spanning tree, we are using vEB tree which is a self-balancing binary search tree and a recursive data structure. The structure stores a summary. Also, it stores minimum and maximum separately from the rest of the structure. Each recursive copy of the data structure stores its own min and max storing the min and max value in its substructure.

3. The minimum and maximum values in every cluster hold a key value which itself is a vertex structure. The vertex structure identifies each vertex of the graph not yet included in the MST. The attributes of the vertex structure include:-

   a) v: The vertex number of the concerned vertex.

   b) key: Weight of the edge with the shortest length leading to that vertex.

   c) P: Other end point of the minimum weight edge to this vertex.

4. An array MST[] is maintained which tells whether a node is part of the Minimum Spanning Tree yet.

Operations:

1. **Insert():** Inserts a given node in the tree.

2. **Delete():** Deletes a given node from the tree.

3. **Findminimum():** Returns the node with minimum value for vertexkey(weight).

4. **Find-vertex():** Returns the node for a given vertexv(i.e. searching a vertex with its label).

Procedure:

1. (Initializing)
   Select a vertex at random to be the source of minimum spanning (vertex 0 is selected in our implementation). Insert source vertex with weight=0 and parent=-1.
   Insert all other vertices in the tree with weight = MAX and parent=-1 where MAX = max(all edge weights) + 1.
   Initialize MST[] =0

2. While tree is not empty
   Repeat steps 3 and 4.

3. Find node with smallest key, x. This is done by getting the minimum value in the top most level of vEB tree.

   - Delete that node.
   - Set the vertex as part of the MST by setting MST[x.v]

4. For all the adjacent vertices of x.v:-
   Let the adjacent vertex be a. If MST[a] is unset:-

   - Find the vertex structure of vertex a in the vEB tree say z
   - If the key value of z is greater than edge weight of (x.v, a) then
     - delete z
     - insert new node for this vertex with new edge weight

4 Analysis

Analysis of Prims using vEB tree
Let v be the number of vertices.
e be the number of edges.
Max be the maximum edge length.

I. Add all vertices to the vEB tree.→ $O(v \log\log(max))$

II. For v iterations:

   i. Find node with minimum key. → O(1)

   ii. Delete the node found.→ $O(loglog(max))$

   iii. Let the number of adjacent nodes of this minimum weight node be x. For x iterations:

      a) Find the neighbouring node in the vEB tree. → O(1).

      b) Check if the weight of edge between this node and the node deleted is less than the previous key value. If less, delete this node → $O(loglog(max))$.
      Insert the node with new key value. → $O(loglog(max))$.

Total time:
$vloglog(max) + v(loglog(max) + v(loglog(max)))$
$= O(v(loglog(max)) + v^2(loglog(max)))$
$= O(v^2(loglog(max)))$

Analysis of Prims using AVL tree

I. Add all vertices to the AVL tree.→ $O(vlogv)$

II. For v iterations:

   i. Find node with minimum key. → $O(logv)$

   ii. Delete the node found.→ $O(logv)$

   iii. Let the number of adjacent nodes of this minimum weight node be x. For x iterations:

      a) Find the neighbouring node in the AVL tree. → O(1).

      b) Check if the weight of edge between this node and the node deleted is less than the previous key value. If less, delete this node → $O(logv)$.
      Insert the node with new key value. → $O(logv)$.

Total time:
$vlogv + v(logv + logv + v(logv + logv))$
$= O(vlogv + v(logv + v(logv)))$
$= O(vlogv + v^2(logv)) = O(v^2(logv))$

Analysis of Prims using RB tree

I. Add all vertices to the RB tree.→ $O(vlogv)$

II. For v iterations:

   i. Find node with minimum key. → $O(logv)$

   ii. Delete the node found.→ $O(logv)$

   iii. Let the number of adjacent nodes of this minimum weight node be x. For x iterations:

      a) Find the neighbouring node in the RB tree. → O(1).

      b) Check if the weight of edge between this node and the node deleted is less than the previous key value. If less, delete this node → $O(logv)$.
      Insert the node with new key value. → $O(logv)$.

Total time:
$vlogv + v(logv + logv + v(logv + logv))$
$= O(vlogv + v(logv + v(logv)))$
$= O(vlogv + v^2(logv)) = O(v^2(logv))$

Comparison of well known operations
Suppose we need to manipulate (insert, delete, find etc.) a set of size s consisting of integer keys in the range {0,1.. u-1}.

Operations	VEB tree	RB tree	AVL tree
Insert	$O(loglogu)$	$O(logs)$	$O(logs)$
Delete	$O(loglogu)$	$O(logs)$	$O(logs)$
findMin	$O(1)$	$O(logs)$	$O(logs)$
findMax	$O(1)$	$O(logs)$	$O(logs)$
Successor	$O(loglogu)$	$O(logs)$	$O(logs)$
Predecessor	$O(loglogu)$	$O(logs)$	$O(logs)$

vEB trees are preferred in case we are given a fixed range of integers to query such that all the keys are integer and distinct. Hence they can be used anywhere in place of a normal binary search tree as long as the keys in the search tree satisfy the above conditions. For applications where you need to be able to find the integer in a set that is closest to some other integer, using a vEB-tree can potentially be faster than using a simple balanced binary search tree.
However one should not see vEB as an alternative for BST but rather as an alternative for arrays. An array has O(1) store and look up costs for object keyed by integers. vEB offers $O(loglogM)$, where M is size of the universe of your values. vEB is not better than a regular array for look ups and store but it offers O(1) min, max operations and $O(loglogM)$ previous, next key operations which array does not.

5 Discussion

The vEB tree deals with keys in a specific range {0,1.. u-1} where each key is say, m-bit. At every level of vEB tree, the universe size at that level is the square root of the previous level universe and hence number of bits to represent the keys at that level become half of the previous level. The high(x) and low(x) are nothing but the higher half of the bits of key x and lower half of the bits respectively where high(x) gives the cluster number of x and low(x) gives the position of x in cluster high(x). Hence every recursion of insertion, deletion etc is of the form, $f(V, x) \rightarrow f(V.cluster[high(x)], low(x))$ where f is a generic function, V is the vEB tree and x is the key under consideration.

Advantages of vEB tree:
An optimization of vEB trees is to discard empty subtrees. This makes vEB trees quite compact when they contain many elements, because no subtrees are created until something needs to be added to them.

Disadvantages of vEB tree:
The overhead associated with vEB trees is huge in the case of small trees: on the order of $\sqrt{M}$ (M is the universe size).This is one reason why they are not popular in practice.
One way of addressing this limitation is to use only a fixed number of bits per level, which results in a trie.
One of the reason is that complexity is defined not on the size of the set you store but on the size of the universe of values.
Also keys can't be arbitrary types for which you have comparison operation but must be

integers.

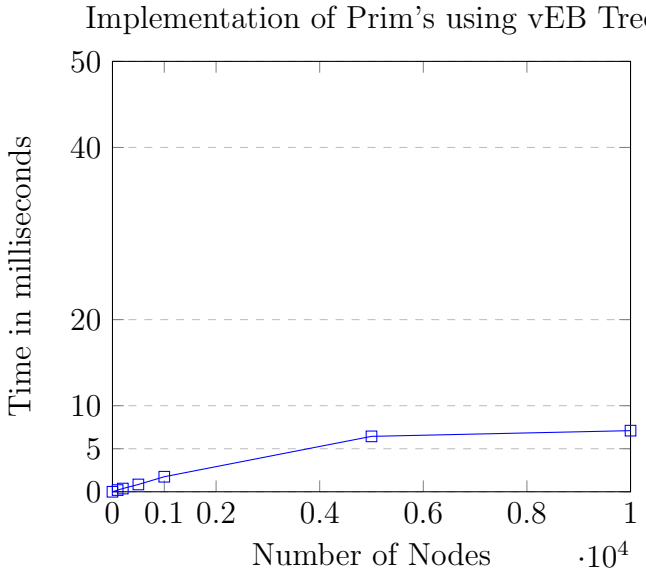
Applications of vEB tree:
One practical application of vEB trees is its use in routing table. vEB trees are used to make routing decisions at the router.
The van Emde Boas layout (based on van Emde Boas tree) is used in cache-oblivious data structures meaning faster data access.

6 Testing

Following are the graphs plotted on the basis of observation on the behaviour of algorithm on different test cases.

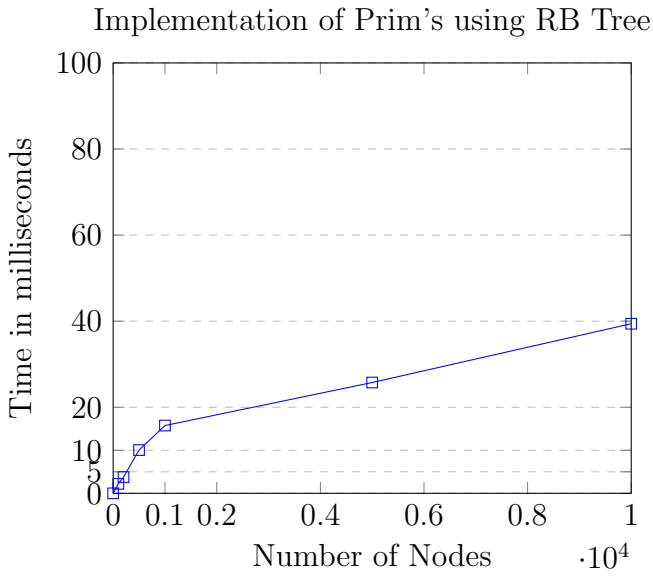
vEB Tree:

No.of Nodes	No.of edges	Time(in seconds)
5	6	0.000111
5	7	0.000120
4	5	0.000114
4	6	0.000122
100	100	0.000130
200	200	0.000296
500	500	0.000704
1000	1000	0.001282
5000	5000	0.006436
10000	10000	0.007009



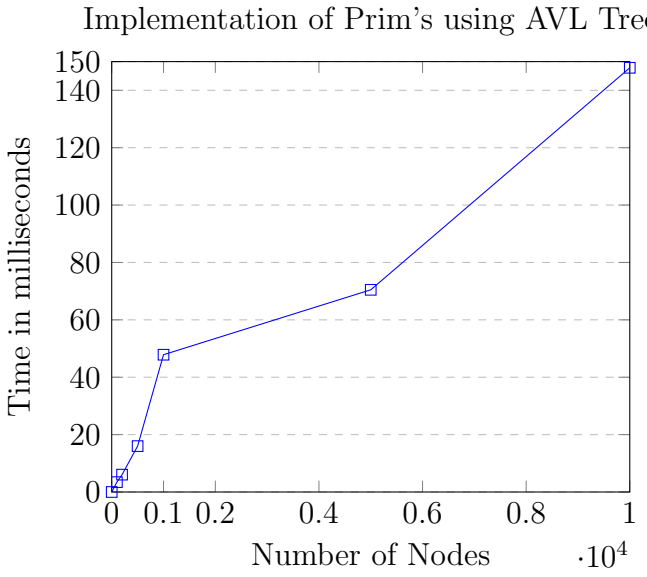
RB Tree:

No.of Nodes	No.of edges	Time(in seconds)
5	6	0.000496
5	7	0.000482
4	5	0.000470
4	6	0.000478
100	100	0.002198
200	200	0.003773
500	500	0.010048
1000	1000	0.015746
5000	5000	0.025742
10000	10000	0.039400



AVL Tree:

No.of Nodes	No.of edges	Time(in seconds)
5	6	0.000476
5	7	0.000477
4	5	0.000468
4	6	0.000434
100	100	0.003440
200	200	0.006046
500	500	0.015994
1000	1000	0.047845
5000	5000	0.073695
10000	10000	0.147845



7 Conclusions

van Emde Boas Tree is a complex data structure but takes reduces the time taken exponentially as compared to AVL and Red-Black trees.