INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

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

Team Members

- Gajanan Modi - 2019201049
- Jeevesh Kataria - 2019201058

Professor:
- **Avinash Sharma**

TA:
- **Aman Joshi**

Abstract

van Emde Boas trees support various operations of a Priority Queue such as SEARCH , INSERT , DELETE , MINIMUM, MAXIMUM , SUCCESSOR, and PREDECESSOR in O(lg lg n) time worst case time. The hitch is that the keys must be integers in the range 0 to n -1, with no duplicates allowed. So, implementing operations in Prims using vEB tree helps in improving the time complexity from O(E log V) to O(E log log V).

Preliminary Approaches for vEB Tree

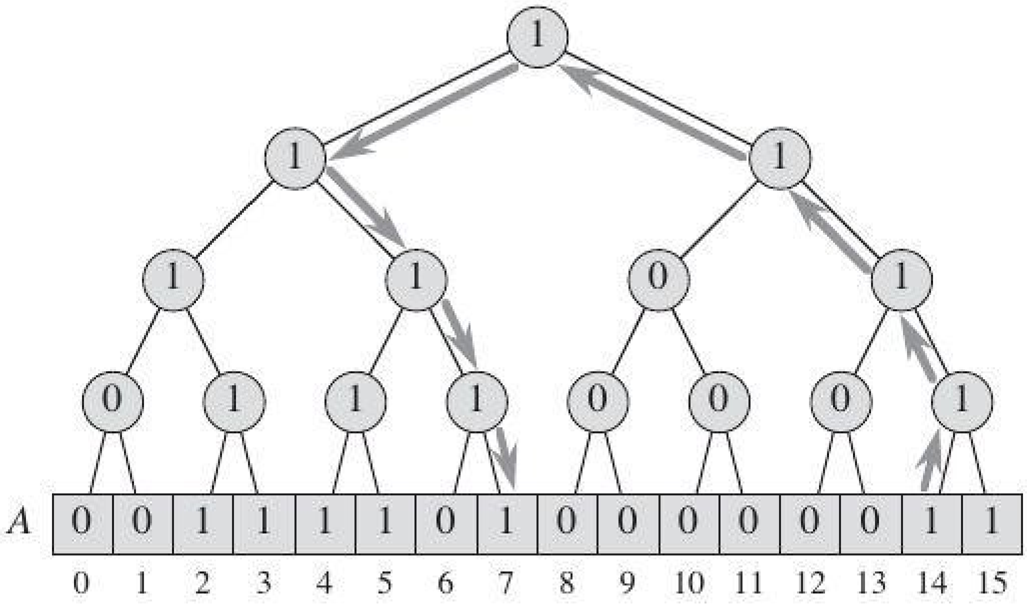
n denotes the number of elements currently in the set and **u** denotes the range of possible values. We call the set {0, 1, 2, … ,u -1} the **universe** of values that can be stored and u the universe size.

We assume throughout that u is an exact power of 2, i.e., u = 2^k for some integer k >= 1.

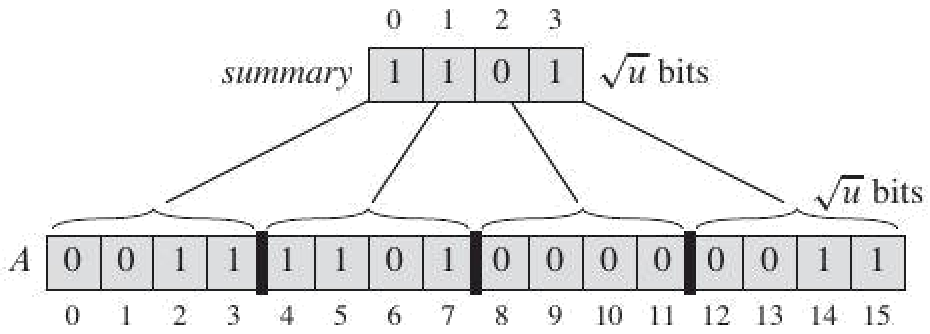
- **Using Bit Vector:** To store a dynamic set of values from the universe {0, 1, 2, …, u-1} we maintain an array A[0, …, u-1] of u bits. The entry A[x] holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise. Although we can perform each of the INSERT, DELETE, and MEMBER operations in O(1) time with a bit vector, the remaining operations —MINIMUM , MAXIMUM, SUCCESSOR, and PREDECESSOR, each take $\Theta($ u) time in the worst case.
- **Superimposing a Binary Tree structure:** The entries of the bit vector form the leaves of the binary tree, and each internal node contains a 1 if and only
- if any leaf in its subtree contains a 1. In other words, the bit stored in an internal node is the logical-or of its two children.
- To find the **minimum** value in the set, start at the root and head down toward the leaves, always taking the leftmost node containing a 1.

- To find the **successor** of 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. Then head down through node z, always taking the leftmost node containing a 1 (i.e., find the minimum value in the subtree rooted at the right child z).

- The below figure represents a binary tree of bits superimposed on top of a bit vector representing the set {2, 3, 4, 5, 7, 14, 15} when u=16. Each internal node contains a 1 if and only if some leaf in its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the set.



1. **Superimposing a tree of constant height:** Instead of superimposing a binary tree on top of the bit vector, we superimpose a tree of degree $\sqrt{u}$. The height of the resulting tree is always 2. Each internal node stores the logical-or of the bits within its sub-tree, so that the $\sqrt{u}$ internal nodes at depth 1 summarize each group of $\sqrt{u}$ values. An array summary[0... $\sqrt{u}$ -1], where summary[i] contains a 1 if and only if the subarray A[i $\sqrt{u}$ ...(i+1) $\sqrt{u}$ -1] contains a 1. We call this $\sqrt{u}$ -bit subarray of A the ith cluster.For a given value of x, the bit A[x] appears in cluster number floor(x/ $\sqrt{u}$).



- 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 pcluster. If we find a 1, that position gives the result. Otherwise, let i floor(x/ $\sqrt{u}$) and search to the right (left) within the summary array from index i. The first position that holds a 1 gives the index of a cluster. Search within that cluster for the leftmost (rightmost) 1. That position holds the successor (predecessor).

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

- **A recursive structure:** 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 $\sqrt{u} = u^{1/2}$ 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. This recurrence relation can be represented as:

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

Let m = lg u, so that u = 2^m
and we have

$$T(2^m) = T(2^{m/2}) + O(1):$$

Now we rename S(m) = T(2^m), giving the new recurrence

$$S(m) = S(m/2) + O(1)$$

By case 2 of the master method, this recurrence has the solution S(m) = O(lg m). We change back from S(m) to T(u), giving T(u) = T(2^m) = S(m) = O(lg m) = O(lg lg u).

Now a given value **x** resides in cluster number floor(x/ $\sqrt{u}$). If we view x as a lg u-bit binary integer, that cluster number, $floor($ x/ $\sqrt{u}$), is given by the most significant (lg u)/2 bits of x. Within its cluster, x appears in position x mod $\sqrt{u}$, which is given by the least significant (lg u)/2 bits of x. We will need to index in this way, and so we define some functions that will help us do so:

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

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

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

The function high(x) gives the most significant (lg u)/2 bits of x, producing the number of x's cluster. The function low(x) gives the least significant (lg u)/2 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 (lg u/2) bits of the element number and y as the least significant (lg u/2) bits.

- **Proto van Emde Boas structures:** It uses a recursive data structure to support the operations.

For the universe {0, 1, 2, …, u-1}, we define a proto van Emde Boas structure, or proto-vEB structure, which we denote as proto-vEB(u), recursively as follows. Each proto-vEB(u) structure contains an attribute u giving its universe size. In addition, it contains the following:

○ If u = 2, then it is the base size, and it contains an array A[0...u-1] of two bits.

○ Otherwise, u = 2^{2^k} for some integer k >= 1, so that u >= 4. In addition to the universe size u, the data structure proto-vEB(u) contains the following attributes:

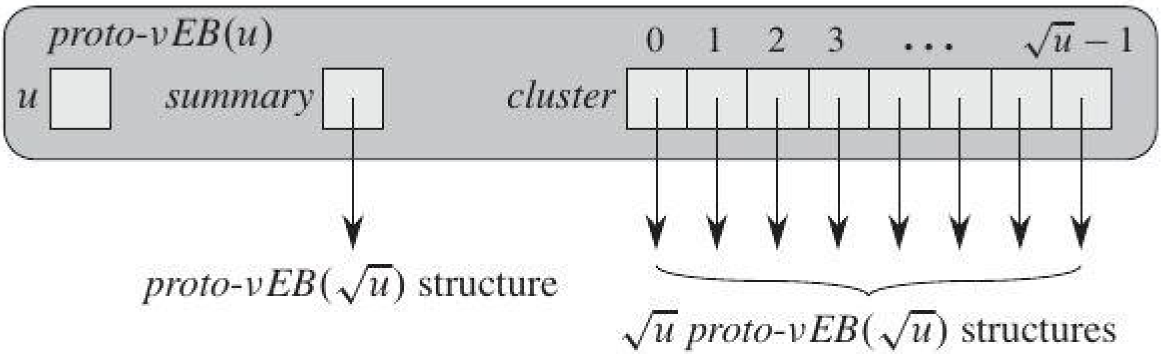
○ a pointer named summary to a proto-vEB($\sqrt{u}$) structure and

● ○ an array cluster[0... $\sqrt{u}$ -1] of $\sqrt{u}$ pointers, each to a proto-vEB($\sqrt{u}$) structure.

●

● The element x, where 0 <= x < u, is recursively stored in the cluster numbered high(x) as element low(x) within that cluster.

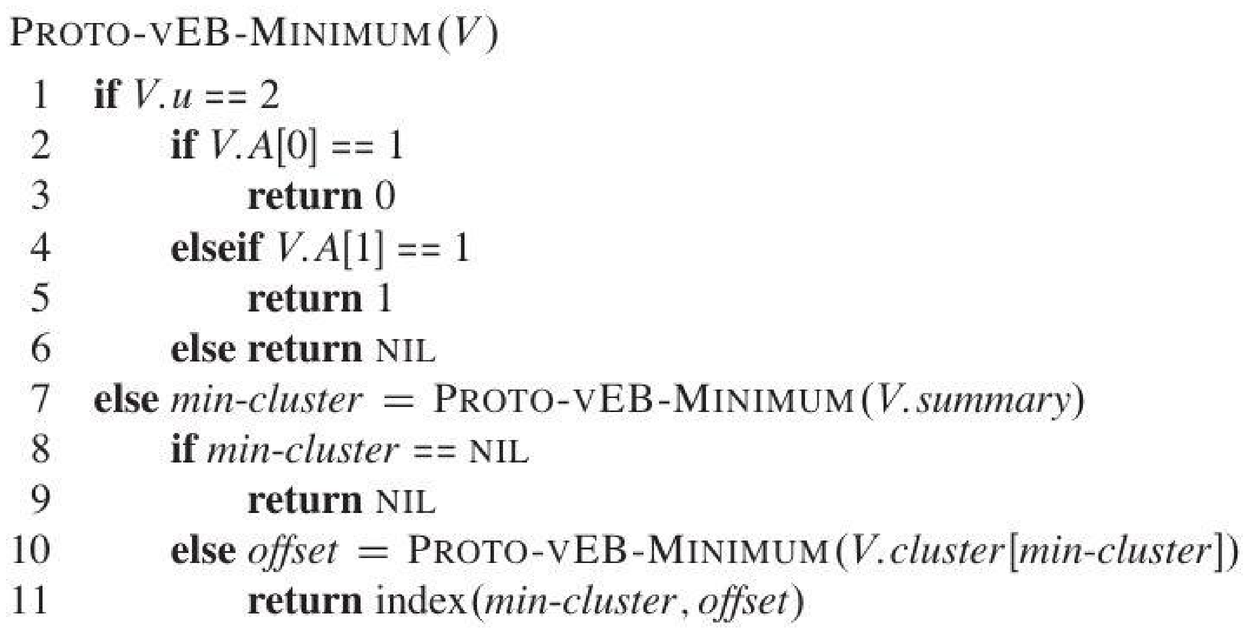
○



proto van Emde Boas Operations

We shall now describe how to perform operations required in the implementation of priority queue for Dijkstra on a proto-vEB structure.

- **Finding the minimum element:** The procedure PROTO-vEB-MINIMUM(V) returns the minimum element in the proto-vEB structure V , or NIL if V represents an empty set.



```
PROTO-vEB-MINIMUM(V)
1  if V.u == 2
2      if V.A[0] == 1
3          return 0
4      elseif V.A[1] == 1
5          return 1
6      else return NIL
7  else min-cluster = PROTO-vEB-MINIMUM(V.summary)
8      if min-cluster == NIL
9          return NIL
10     else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
11         return index(min-cluster, offset)
```

Line 1 tests for the base case, which lines 2–6 handle by brute force. Lines 7–11 handle the recursive case. First, line 7 finds the number of the first cluster that contains an element of the set. It does so by recursively calling PROTO-vEB-MINIMUM on V.summary, which is a proto-vEB($\sqrt{u}$) structure. Line 7 assigns this cluster number to the variable min-cluster. If the set is empty, then the recursive call returned NIL, and line 9 returns NIL. Otherwise, the minimum element of the set is somewhere in cluster number min-cluster. The recursive call in line 10 finds the offset within the cluster of the minimum element in this cluster. Finally, line 11 constructs the value of the minimum element from the cluster number and offset, and it returns this value.

Time Complexity of PROTO-vEB-MINIMUM

Let T(u) denote the worst-case time for PROTO-vEB-MINIMUM on a proto-vEB(u) structure, we have the recurrence:

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

Let m = lg u,

$$T(2^m) = 2T(2^{m/2}) + O(1)$$

Renaming S(m) = T(2^m) gives

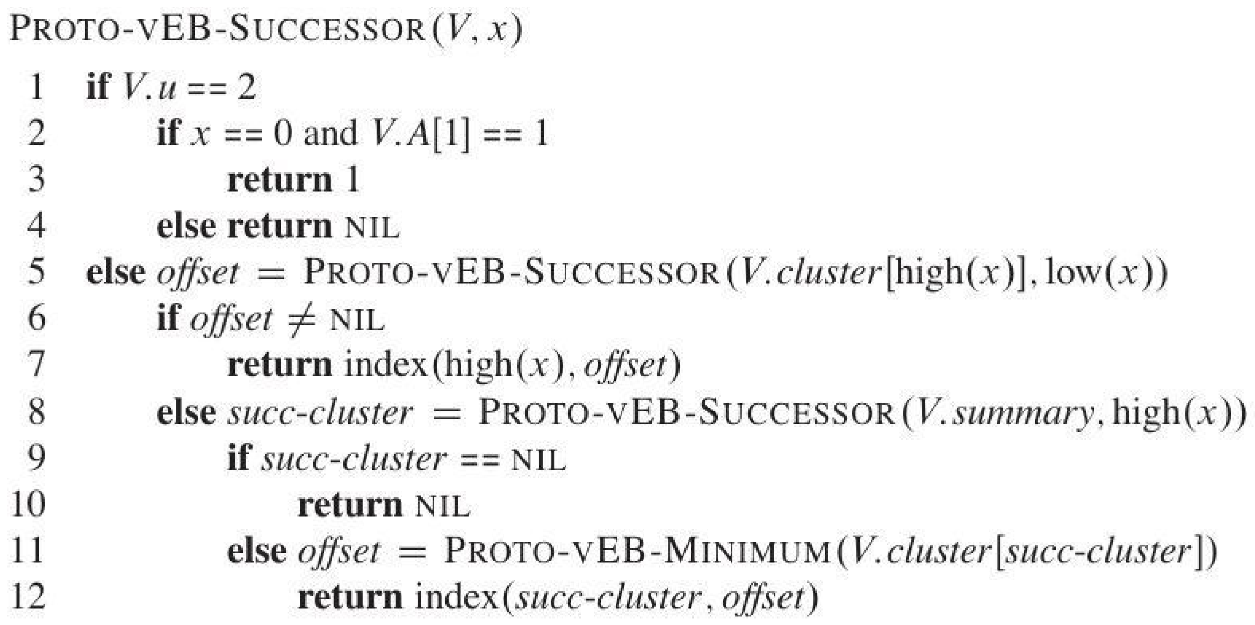
$$S(m) = 2S(m/2) + O(1)$$

which, by case 1 of the master method, has the solution S(m) = Θ (m). By changing back from S(m) to T(u), we have that T(u) = T(2^m) = S(m) = Θ (m) = Θ (lg u).

Because this procedure makes two recursive calls on proto-vEB($\sqrt{u}$) structures, it does not run in O(lg lg u) time in the worst case.

- **Finding the successor:** The procedure PROTO-vEB-SUCCESSOR(V, x) returns the smallest element in the proto-vEB structure **V** that is greater than x, or NIL if no element in V is greater than x. It does not require x to be a member of the set, but it does assume that $0 \leq x <$ V.u.

- The PROTO-vEB-SUCCESSOR procedure works as follows. Line 1 tests for the base case, which lines 2–4 handle by brute force: the only way that x can have a successor within a proto-vEB(2) structure is when x=0 and A[1] is 1. Lines 5–12 handle the recursive case. Line 5 searches for a successor to x within x's cluster, assigning the result to offset. Line 6 determines whether x has a successor within its cluster; if it does, then line 7 computes and returns the value of this successor. Otherwise, we have to search in other clusters. Line 8 assigns to succ-cluster the number of the next non empty cluster, using the summary information to find it. Line 9 tests whether succ-cluster is NIL, with line 10 returning NIL if all succeeding clusters are empty. If succ-cluster is non-NIL , line 11 assigns the first element within that cluster to offset, and line 12 computes and returns the minimum element in that cluster.



```
PROTO-vEB-SUCCESSOR(V, x)
 1  if V.u == 2
 2      if x == 0 and V.A[1] == 1
 3          return 1
 4      else return NIL
 5  else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
 6      if offset ≠ NIL
 7          return index(high(x), offset)
 8      else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
 9          if succ-cluster == NIL
10              return NIL
11          else offset = PROTO-vEB-MINIMUM(V.cluster[succ-cluster])
12              return index(succ-cluster, offset)
```

Time Complexity of PROTO-vEB-SUCCESSOR

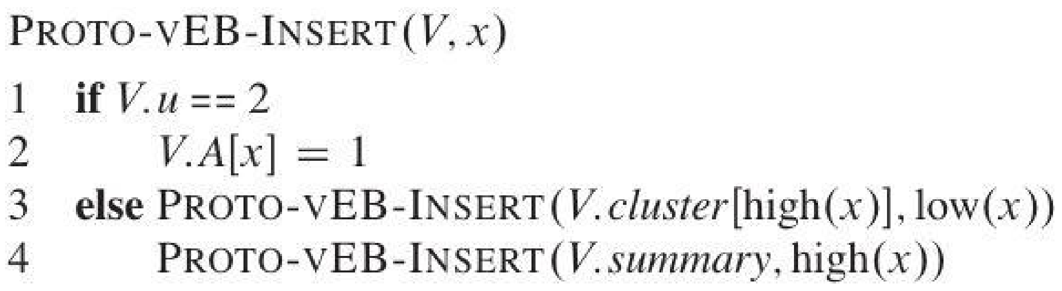
The SUCCESSOR operation is even worse. In the worst case, it makes two recursive calls, along with a call to PROTO-vEB-MINIMUM. Thus,

$T(u) = 2T(\sqrt[\uparrow]{u}) + \Theta(\lg \sqrt[\uparrow]{u})$

$= 2T(\sqrt[\uparrow]{u}) + \Theta(\lg u)$

This recurrence has the solution T(u) = Θ (lg u lg lg u).

- **Inserting an element:** To insert an element, we need to insert it into the appropriate cluster and also set the summary bit for that cluster to 1. The procedure PROTO-vEB-INSERT(V, x) inserts the value x into the proto-vEB structure V.



```
PROTO-vEB-INSERT$(V, x)$
1  if $V.u == 2$
2        $V.A[x] = 1$
3  else PROTO-vEB-INSERT$(V.cluster[\text{high}(x)], \text{low}(x))$
4        PROTO-vEB-INSERT$(V.summary, \text{high}(x))$
```

In the base case, line 2 sets the appropriate bit in the array A to 1. In the recursive case, the recursive call in line 3 inserts x into the appropriate cluster, and line 4 sets the summary bit for that cluster to 1.

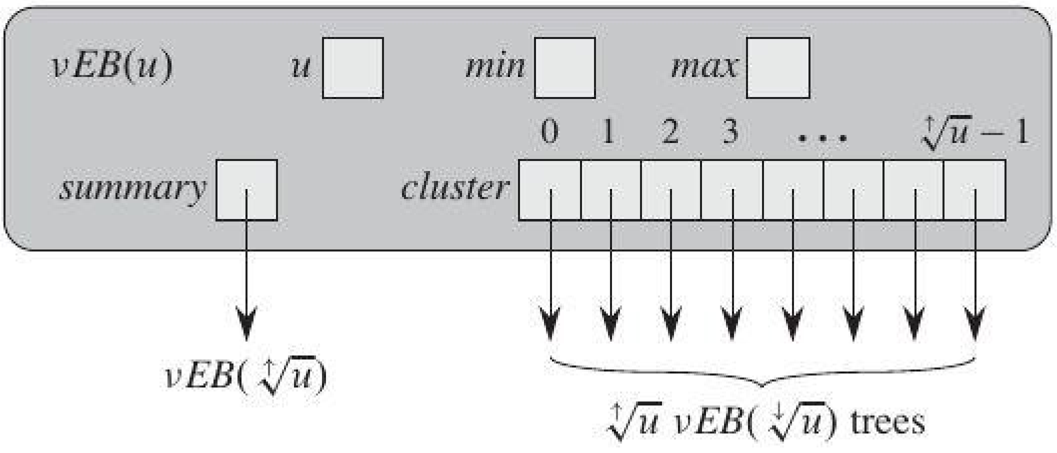
van Emde Boas Trees

The van Emde Boas tree, or vEB tree, modifies the proto-vEB structure. We denote a vEB tree with a universe size of u as vEB(u) and, unless u equals the base size of 2, the attribute summary points to a vEB(uppersqrt(u)) tree and the array cluster [0...uppersqrt(u)

- 1] points to uppersqrt(u) vEB(lowersqrt(u)) trees. 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.

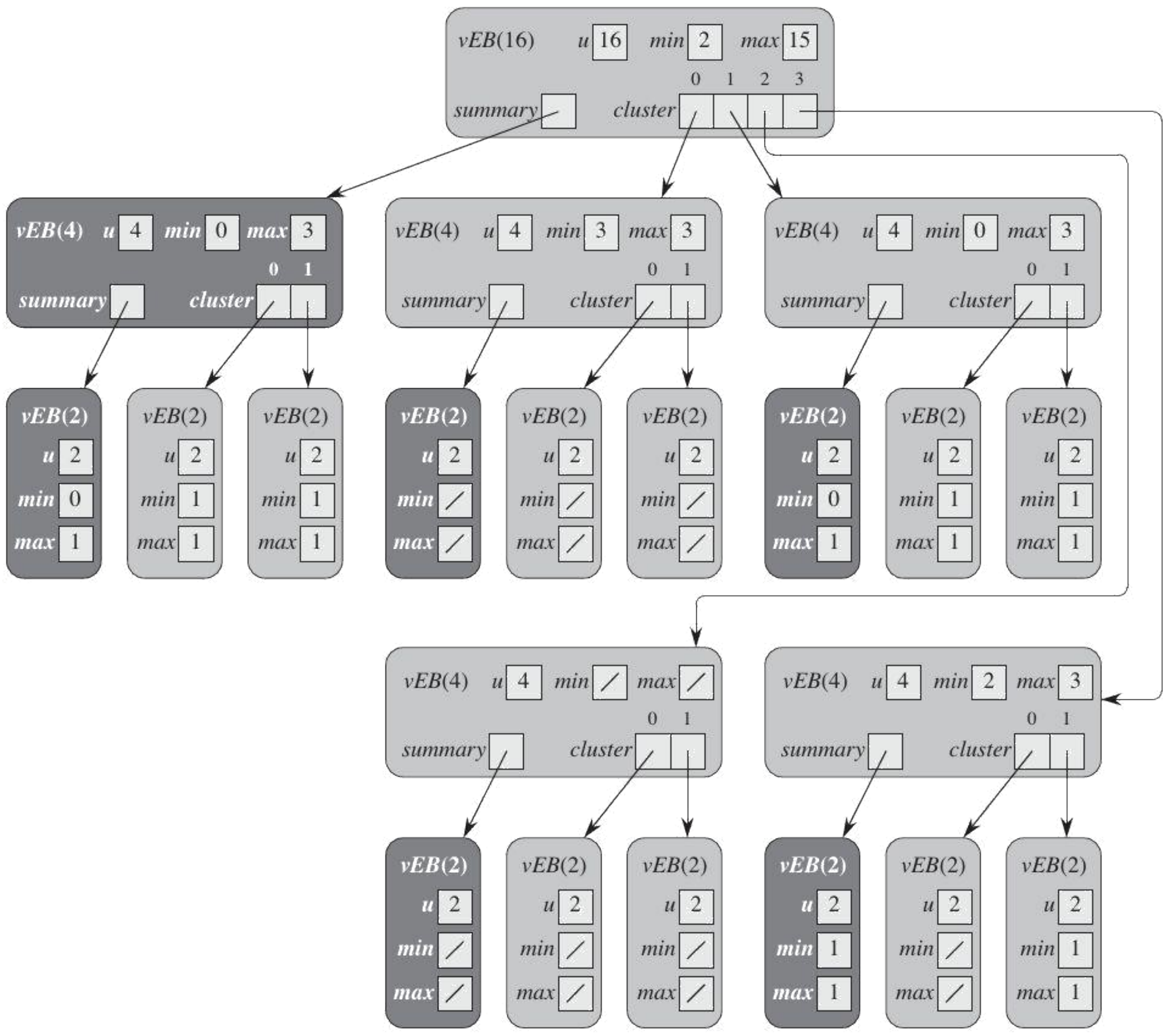
Furthermore, the element stored in min does not appear in any of the recursive vEB(lowersqrt(u)) trees that the cluster array points to. The elements stored in a vEB(u) tree V , therefore, are V.min plus all the elements recursively stored in the vEB(lowersqrt(u)) trees pointed to by V.cluster[0...u-1].



$vEB(u)$
u
min
max
0 1 2 3 … $\sqrt[\uparrow]{u}-1$
$summary$
$cluster$
$vEB(\sqrt[\uparrow]{u})$
$\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ trees

Since the base size is 2, a vEB(2) tree does not need the array A that the corresponding proto-vEB(2) structure has. Instead, we can determine its elements from its min and max attributes. In a vEB tree with no elements, regardless of its universe size u, both min and max are NIL.

The **min and max** attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:


vEB(16) u 16 min 2 max 15
summary
cluster 0 1 2 3
vEB(4) u 4 min 0 max 3
summary cluster 0 1
vEB(4) u 4 min 3 max 3
summary cluster 0 1
vEB(4) u 4 min 0 max 3
summary cluster 0 1
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min max
vEB(2) u 2 min max
vEB(2) u 2 min max
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(4) u 4 min max
summary cluster 0 1
vEB(4) u 4 min 2 max 3
summary cluster 0 1
vEB(2) u 2 min max
vEB(2) u 2 min max
vEB(2) u 2 min max
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min max
vEB(2) u 2 min 1 max 1

A vEB(16) tree V holding the set {2, 3, 4, 5, 7, 14, 15}. Because the smallest element is 2, V.min equals 2, and even though high(2) = 0, the element 2 does not appear in the vEB(4) tree pointed to by V.cluster[0]: notice that V.cluster[0].min equals 3, and so 2 is not in this vEB tree. Similarly, since V.cluster[0].min equals 3, and 2 and 3 are the only elements in V.cluster[0], the vEB(2) clusters within V.cluster[0] are empty.

van Emde Boas Trees Operations

- **Finding the minimum and maximum elements:** Finding minimum and maximum element in vEB tree takes constant time.

```
vEB-TREE-MINIMUM($V$)
return $V.min$
```

```
vEB-TREE-MAXIMUM($V$)
return $V.max$
```

- **Finding the successor and predecessor:** In a vEB tree because we can access the maximum value quickly, we can avoid making two recursive calls, and instead make one recursive call on either a cluster or on the summary, but not on both.
-

```
vEB-TREE-SUCCESSOR($V, x$)
if $V.u == 2$
    if $x == 0$ and $V.max == 1$
        return 1
    else return NIL
elseif $V.min \neq$ NIL and $x < V.min$
    return $V.min$
else $max\text{-}low$ = vEB-TREE-MAXIMUM($V.cluster[\text{high}(x)]$)
    if $max\text{-}low \neq$ NIL and $\text{low}(x) < max\text{-}low$
        $offset$ = vEB-TREE-SUCCESSOR($V.cluster[\text{high}(x)], \text{low}(x)$)
        return index($\text{high}(x), offset$)
    else $succ\text{-}cluster$ = vEB-TREE-SUCCESSOR($V.summary, \text{high}(x)$)
        if $succ\text{-}cluster$ == NIL
            return NIL
        else $offset$ = vEB-TREE-MINIMUM($V.cluster[succ\text{-}cluster]$)
            return index($succ\text{-}cluster, offset$)
```

- **Inserting an element:** 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 that recursive call.

  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:

- **Time Complexity of vEB Tree Operations:**

  The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence:

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

  Letting m = lg u, we rewrite it as:

  $$T(2^m) \leq T(2^{m/2}) + O(1)$$

  Noting that $m/2 \leq 2m=3$ for all $m \geq 2$, we have

  $$T(2^m) \leq T(2^{2m/3}) + O(1)$$

  Let S(m) = T(2^m), we rewrite this last recurrence as

  $$S(m) = S(2m/3) + O(1)$$

  which, by case 2 of the master method, has the solution S(m) = O(lg m). (In terms of the asymptotic solution, the fraction 2/3 does not make any difference compared with the fraction 1/2, because when we apply the master method, we find that $\log_{3/2} 1 = \log_2 1 = 0$) Thus, we

  have T(u) = T(2^m) = S(m) = O(lg m) = O(lg lg u).

```
vEB-EMPTY-TREE-INSERT(V, x)
V.min = x
V.max = x
```

```
vEB-TREE-INSERT(V, x)
if V.min == NIL
    vEB-EMPTY-TREE-INSERT(V, x)
else if x < V.min
        exchange x with V.min
    if V.u > 2
        if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
            vEB-TREE-INSERT(V.summary, high(x))
            vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
        else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
    if x > V.max
        V.max = x
```

AVL Tree

An AVL tree is a balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, it was first dynamically balanced trees to be proposed. They are not completely balanced but the height of two child subtree can differ by at most one.

Whenever this condition in violated the condition is restored with the help of one or more rotations. AVL trees can support Insertion, Deletion and Searching in ***(log n)*** time. We have also implemented the Kth element in ***(log n)*** time.

Structure of AVL Tree Node:

```
struct
  AVL_Node{ long
  long value; long
  long height; long
  long weight;
  AVL_Node* left;
  AVL_Node* right; AVL_Node* parent;
  AVL_Node (long long k);

};
```

//Data of the node

//Height of the node

//Weight of the node

//Pointer to the Left Subtree

//Pointer to right subtree

//Pointer to parent of the node

//Constructor

The above node structure was used for implementing AVL Trees. Weight was used for finding the Kth element in ***(log n)*** time. AVL trees have strict heights compared to other search trees of the same kind. They have more rotations to maintain their strict height. They are more suitable where lookups are more common than Insertions and Deletions.

Red-Black Trees

A red-black tree is a self-balancing binary search tree which perform Insertion, Deletion, Search in ***(log n)*** of worst case time.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either `RED` or `BLACK`. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged

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.

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. Complex-ity = 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.

2. 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
- 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.

GUIDANCE:

Prof: Avinash Sharma

- Complete Discussion of Insertion in case on Van Emde Boas Tree.

Teaching Assistant:

- **Aman Joshi**
  Discussion 1: Introduction to Van Emde Boas tree.
  Discussion 2: Mid-Evaluation : Implement Prim’s Algorithm using (VEB,RB Tree,AVL Tree)