# Project Report

### Team Name

Grey Matter

### Team Members

1) Shafiya Naaz (2018201062)
2) Shivam Singh (2018201015)

### Project Link

https://github.com/shivam496/APS-Project (https://github.com/shivam496/APS-Project)

---

## 1. Introduction

Implementation of van Emde Boas tree with application to Kruskal algorithm and compare its performance with respect to Fibonacci and Binomial heaps.

---

## 2. van Emde Boas tree

### 2.1. Introduction

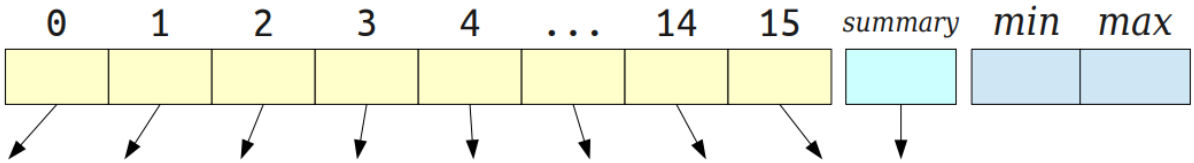
A van Emde Boas tree, is a tree data structure which implements an associative array with m-bit integer keys, where m is of the form $2^n$. It supports priority-queue operations, and a few others, each in $O(\log \log n)$ worst-case time.

### 2.2. vEB tree data structure

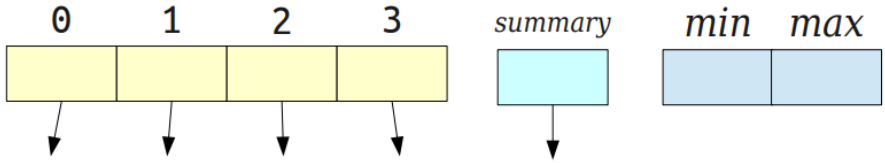
```
A vEB tree is either
    null or
    a node

A node is a structure with five fields:
    cluster: which is pointer to an array of type vEB and size u,
    summary: a pointer of type vEB,
    min: stores the minimum element in that vEB tree,
    max: stores maximum element in that vEB tree,
    u: size of the array to which cluster points
```

In [ ]:
```
struct vEB
{       long long u;
        long long *min, *max;
        vEB *summary;
        vEB **cluster;
};
```

- The top-level structure looks like this:




- Each structure one level below (and the summary) looks like this:




## 2.3. Test Cases for vEB tree

vEB tree should give the following outputs for the inputs provided :

In [ ]:
```
> vEB = new class vEB(256)
> vEB->Insert(3)
> vEB->Insert(10)
> vEB->Insert(100)
> vEB->max
    100
> vEB->min
    3
> vEB->successor(20)
    100
> vEB->predecessor(20)
    3
> vEB->successor(200)
    -1
> vEB->extractMin()
    3
> vEB->Remove(10)
> vEB->max
    100
> vEB->min
    100
```

## 2.4. Some Operations on vEB tree

### 2.4.1. Member

Checks whether an element is present in vEB or not.

In [ ]:
```
// Pseudo code to check whether a key is present in vEB or not
1 vEB_Member(V,x)
2    if x == V.min or x == V.max // Comparing with max and min and returning
3        return TRUE
4    else if V.u == 2 // Base Case
5        return FALSE
6    else return Member(V,cluster[high(x)],low(x))
```

One recursive call is done on a vEB tree with universe size of $\sqrt{u}$. Hence, vEB_Member runs in $O(\log \log u)$ worst-case time.

### 2.4.2. Finding Minimum or Maximum

We store the minimum and maximum in the attributes min and max, so these two of the operations are one-liners, taking constant time.

In [ ]:
```
// Pseudo code for finding maximum and minimum in vEB tree

1 vEB_Minimum(V)
2     return V.min

3 vEB_Maximum(V)
4     return V.max
```

### 2.4.3. Inserting an Element

The vEB_Insert procedure will make only one recursive call. 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.

In [ ]:
```
// Pseudo Code to insert an element assuming x is not already present in vEB
1 vEB_Insert(V,x)
2    if V.min == NULL // checking tree is empty or not
3        vEB_Empty_Tree-Insert.(V,x)
4    else
5        if x < V.min
6            exchange x with V.min // Exchanging x with min
             so we don't lose previous min
7        if V.u > 2 // Inserting x in vEB tree
8            if vEB_Minimum(V.cluster[high(x)]) == NULL
9            vEB_Insert(V.summary,high(x))
10            vEB_Empty_Tree-Insert(V.cluster[high(x)],low(x))
11       else
12            vEB_Insert(V.cluster[high(x)],low(x))
13       if x > V.max // Comparing with max and updating it if required
14            V.max = x
```

We only make at most one “real” recursive call:
- If we don't recurse into the summary, we only made one recursive call down into a substructure.
- If we make a recursive call into the summary, we did so because the other call was on an empty subtree, which isn't a “real” recursive call.

So recurrence relation:

$T(2) = \Theta(1)$

$T(u) \leq T(u/2) + \Theta(1)$ which makes $T(u)$ for insertion as $O(\log \log u)$

### 2.4.4. Finding Successor of an Element

This Procedure finds the next greater element than the input x present in vEB tree.

If not the base case and x ≥ minimum value, let max-low be the maximum in x’s cluster. If there is a greater element in the cluster then assign it to offset and return the index of the successor. Otherwise we have to search for the next non-empty cluster. If any, offset gives the minimum in that cluster.

In [ ]:
```
// Pseudo code to find successor of an element of vEB tree
1 vEB_Successor(V,x)
2    if V.u == 2
3        if x == 0 and V.max == 1
4            return 1
5        else
6            return NULL
7    else if V.min != NULL and x < V.min
8        return V.min
9    else max-low = vEB_Maximum(V.cluster[high(x)])
10       if max-low != NULL and low(x) < max-low
11           offset = vEB_Successor(V.cluster[high(x)],low(x))
12           return index(high(x),offset)
13       else succ-cluster = vEB_Successor(V.summary, high(x))
14           if succ-cluster == NULL
15               return NULL
16           else offset = vEB_Minimum(V.cluster[succ-cluster])
17              return index(succ-cluster,offset)
```

Depending on the result of the test in line 7, the procedure calls itself recursively in either line 9 (on a vEB tree with universe size $\sqrt{u}$) or line 11 (on a vEB tree with universe size $\sqrt{u}$). In either case, the one recursive call is on a vEB tree with universe size at most $\sqrt{u}$. The remainder of the procedure, including the calls to vEB_Minimum and vEB_Maximum, takes $O(1)$ time. Hence, vEB_Successor runs in $O(\log \log u)$ worst-case time.

### 2.4.5. Deleting an Element

Deletion precedure can be done as follows:
- If the tree has just one element, update min and max appropriately and stop.
- If min or max are being deleted, replace them with the min or max of the first or last non-empty tree, then proceed as if deleting that element instead.
- Delete x mod u/2 from its subtree.
- If that subtree is empty, delete ⌊x / u/2⌋ from the summary bitvector.

In [ ]:
```
// Pseudo code to delete an element from vEB tree
1 vEB_Delete(V,x)
2    if V.min == V.max
3        V.min = NULL
4        V.max = NULL
5    else if V.u == 2
6        if x == 0
7            V.min = 1
8        else V.min = 0
9        V.max = V.min
10    else
11        if x == V.min
12            first-cluster = vEB_Minimum(V.summary)
13            x = index(first-cluster,vEB_Minimum(V.cluster[first-cluster]))
14            V.min = x
15        vEB_Delete(V.cluster[high(x)],low(x))
16        if vEB_Minimum(V.cluster[high(x)]) == NULL
17            vEB_Delete(V.summary, high(x))
18            if x == V.max
19                summary-max = vEB_Maximum(V.summary)
20            if summary-max == NULL
21                V.max = V.min
22            else V.max = index(summary-max;vEB_Maximum(V.cluster[summary-m
23        else if x == V.max
24            V.max = index(high(x),vEB_Maximum(V.cluster[high(x)]))
```

The worst case running time is $O(\log \log u)$ and can be seen from the above pseudo code as at each case exactly one recursive call is done on a vEB tree with universe size at most $\sqrt{u}$.

### 2.4.6. Finding Predecessor of an element

This Procedure finds the greatest element present in vEB tree smaller than the input x. It can be implemented similar to the successor procedure and it's complexity can be explained similarly.

In [ ]:
```
// Pseudo code to find predecessor of an element of vEB tree
1 vEB_Predecessor(V,x)
2    if V.u == 2
3        if x == 1 and V.min == 0
4            return 0
5        else return NIL
6    else if V.max != NIL and x > V:max
7        return V.max
8    else min-low = vEB_Minimum(V.cluster[high(x)])
9        if min-low != NULL and low(x) > min-low
10           offset = vEB_Predecessor(V.cluster[high(x)], low(x))
11           return index(high(x), offset)
12       else pred-cluster = vEB_Predecessor(V.summary, high(x))
13           if pred-cluster == NULL
14               if V.min != NULL and x > V.min
15                   return V.min
16               else return NULL
17           else offset = vEB_Maximum(V.cluster[pred-cluster])
18               return index(pred-cluster, offset)
```

## 2.5. Space Requirement for vEB tree

vEB tree has the number of keys in the form of $2^{2^k}$ where k is an integer because at each level the universe size has to become square root of previous universe size,

So the recurrence relation becomes,

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

so, $P(u) = O(u)$

---

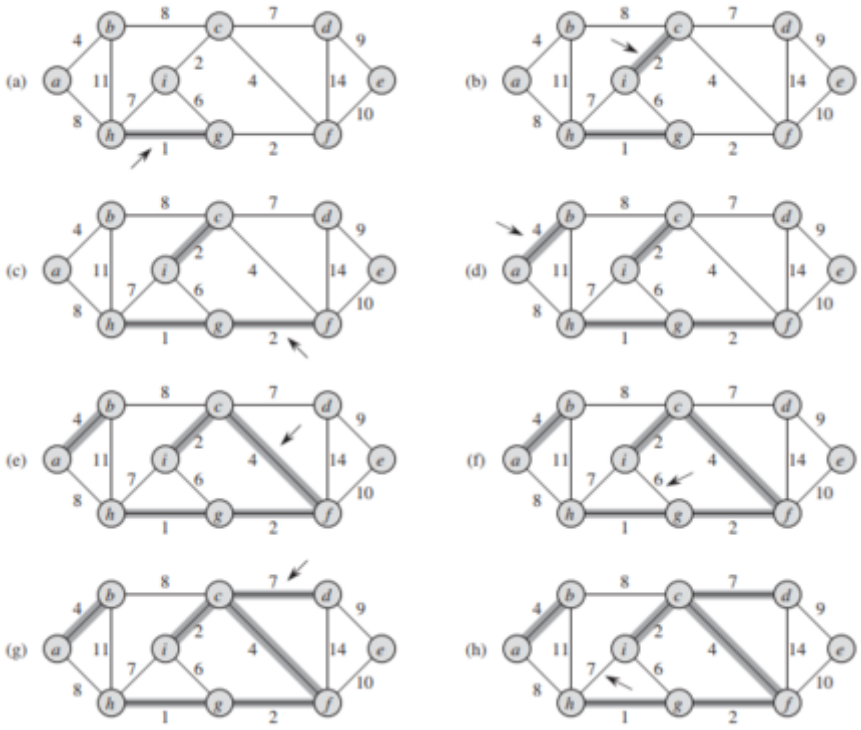
# 3. Kruskal's Algorithm

## 3.1. Introduction

Kruskal's 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).

## 3.2. Algorithm for Kruskal's

1) create a graph F (a set of trees), where each vertex in the graph is a separate tree
2) create a set S containing all the edges in the graph
3) while S is nonempty and F is not yet spanning
    3.1) remove an edge with minimum weight from S
    3.2) if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree
4) At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree




The execution of Kruskal’s algorithm on the graph. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

## 3.3. Pseudo Code

We have implemented kruskal's algorithm using vEB tree as well as disjoint set data structure

In [ ]:
```
Kruskal(G):
    A = NULL
    foreach v in G.V:
        Make-Set(v)
    foreach (u,v) in G.E ordered by weight(u,v), increasing:
        if Find-Set(u) ≠ Find-Set(v):
            A = A U {(u,v)}
            Union(u,v)
    return A
```

## 3.4. Disjoint-Set data structure

A Disjoint-Set data structure (also called a union–find data structure or merge–find set) is a data structure that tracks a set of elements partitioned into a number of disjoint subsets. It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set or not.

### 3.4.1. Make-Set procedure

The MakeSet operation makes a new set by creating a new element with a unique id, a rank of 0, and a parent pointer to itself. The parent pointer to itself indicates that the element is the representative member of its own set.

The MakeSet operation has $O(1)$ time complexity, so initializing n sets has $O(n)$ time complexity.

In [ ]:
```
MakeSet(x)
  if x is not already present:
    add x to the disjoint-set tree
    x.parent = x
    x.rank   = 0
    x.size   = 1
```

### 3.4.2. Find-Set procedure

Find-Set(x) follows the chain of parent pointers from x up the tree until it reaches a root element, whose parent is itself. This root element is the representative member of the set to which x belongs, and may be x itself.

In [ ]:
```
Find(x)
  if x.parent != x
    x.parent = Find(x.parent)
  return x.parent
```

### 3.4.3. Union by rank procedure

Union(x,y) uses Find to determine the roots of the trees x and y belong to. If the roots are distinct, the trees are combined by attaching the root of one to the root of the other.

Union by rank always attaches the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the originals unless they were of equal height, in which case the resulting tree is taller by one node.

In [ ]:
```
Union(x, y)
    xRoot = Find(x)
    yRoot = Find(y)
    if xRoot == yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot, yRoot = yRoot, xRoot // swap xRoot and yRoot
    yRoot.parent = xRoot //merge yRoot into xRoot
    if xRoot.rank == yRoot.rank
         xRoot.rank = xRoot.rank + 1
```

### 3.5. Complexity of Kruskal's Algorithm

We are using the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known.

Sorting the edges takes $O(E \log E)$ time and Make-Set operations take a total of $O(V + E)$ time.

So, the total running time of Kruskal’s Algorithm is $O(E \log E)$. Observing that $|E| < |V|^2$, we have $\log |E| = O(\log V)$, and so we can restate the running time of Kruskal’s algorithm as $O(E \log V)$ when we don't use vEB data strucutre.

---

## 4. Binomial Heap

### 4.1. Introduction

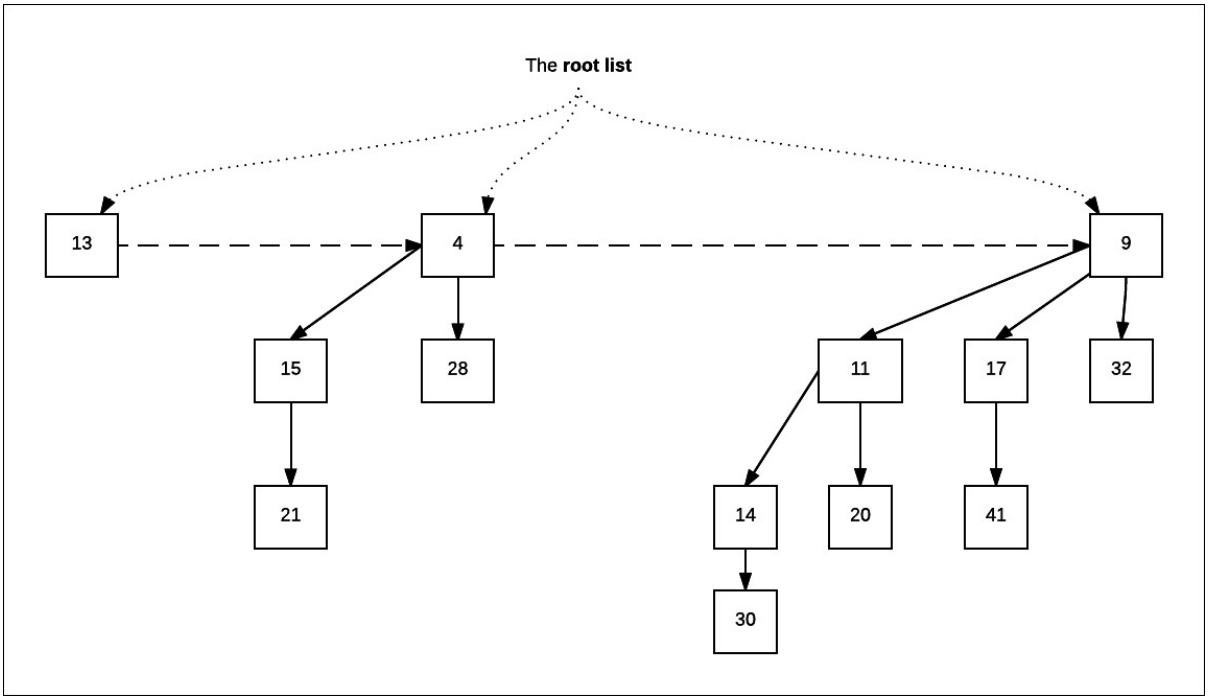
A binomial heap H is a set of binomial trees that satisfies the following binomial-heap properties.
    1) Each binomial tree in H obeys the min-heap property. The key of a node is greater than or equal to the key of its parent. We say that each such tree is min-heap-ordered.
    2) For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

### 4.2. Structure of Binomial Heap

Each binomial tree within a binomial heap is stored in the leftchild. Each node has a key field. In addition, each node x contains pointers p[x] to its parent, child[x] to its leftmost child, and sibling[x] to the sibling of x immediately to its right. If node x is a root, then p[x] = NULL. If

node x has no children, then child[x] = NULL, and if x is the rightmost child of its parent, then sibling[x] = NULL. Each node x also contains the field degree[x], which is the number of children of x.

The roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the root list. The degrees of the roots strictly increase as we traverse the root list.




## 4.3. Test Cases for Binomial Heap

Binomial Heap should give the following outputs for the inputs provided :

In [ ]:
```
> bh = new BinHeap();
> fh->insert(new BinNode(3))
> fh->insert(new BinNode(10))
> fh->insert(new BinNode(100))
> fh->first
    3
> fh->isEmpty()
    0
> fh->Delete(3)
> fh->extractMin()
    10
> fh->first()
    10
> fh->Delete(10)
> fh->Delete(100)
> fh->isEmpty()
    1
```

## 4.4. Some operations on Binomial Heap which are implemented in our project

### 4.4.1. Make Heap

To make an empty binomial heap, the Make-Binomial-Heap procedure simply allocates and returns an object H, where head[H] = NULL. The running time is $\Theta(1)$.

### 4.4.2. Insertion in Binomial Heap

The following procedure inserts node x into binomial heap H , assuming that x has already been allocated and key[x] has already been filled in.

In [ ]:
```
// Pseudo code to insert an element in Binomial heap
1 Binomial-Heap-Insert(H,x)
2     H' = Make-Binomial-Heap()
3     p[x] = NULL
4     child[x] = NULL
5     sibling[x] = NULL
6     degree[x] = 0
7     head[H'] = x
8     H = Binomial-Heap-Union(H,H')
```

The procedure simply makes a one-node binomial heap H' in $O(1)$ time and unites it with the n-node binomial heap H in $O(\log n)$ time. The call to Binomial-Heap-Union takes care of freeing the temporary binomial heap H'.

### 4.4.3. Extract min in Binomial Heap

The process of extracting the minimum node is the most complicated of the operations in Binomial heap. This method finds the minimum element in heap, removes it. balances the root list such that no two elements in that list remains with same degree and points the min to the minimum element.

In [ ]:
```
// Pseudo code to extract the minimum node in Binomial Heap
1 Binomial-Heap-Exract-Min(H)
2     find the root x with the minimum key in the root list of H, and
      remove x from the root list of H
3     H' = Make-Binomial-Heap()
4     reverse the order of the linked list of x's children, and set head[H']
      to point to the head of the resulting list
5     H = Binomial-Heap-Union(H,H')
6     return x
```

Since each of lines 1-4 takes $O(\log n)$ time if H has n nodes, Binomial-Heap-Exract-Min runs in $O(\log n)$ time.

## 4.5. Complexities for various operations of Binomial Heap

```
Make-Heap    --   θ(1)
Insert       --   O(log n)
Minimum      --   O(log n)
Extract-in   --   θ(log n)
Union        --   O(log n)
Decrease-Key --   θ(log n)
Delete       --   θ(log n)
```

---

# 5. Fibonacci Heap

## 5.1. Introduction

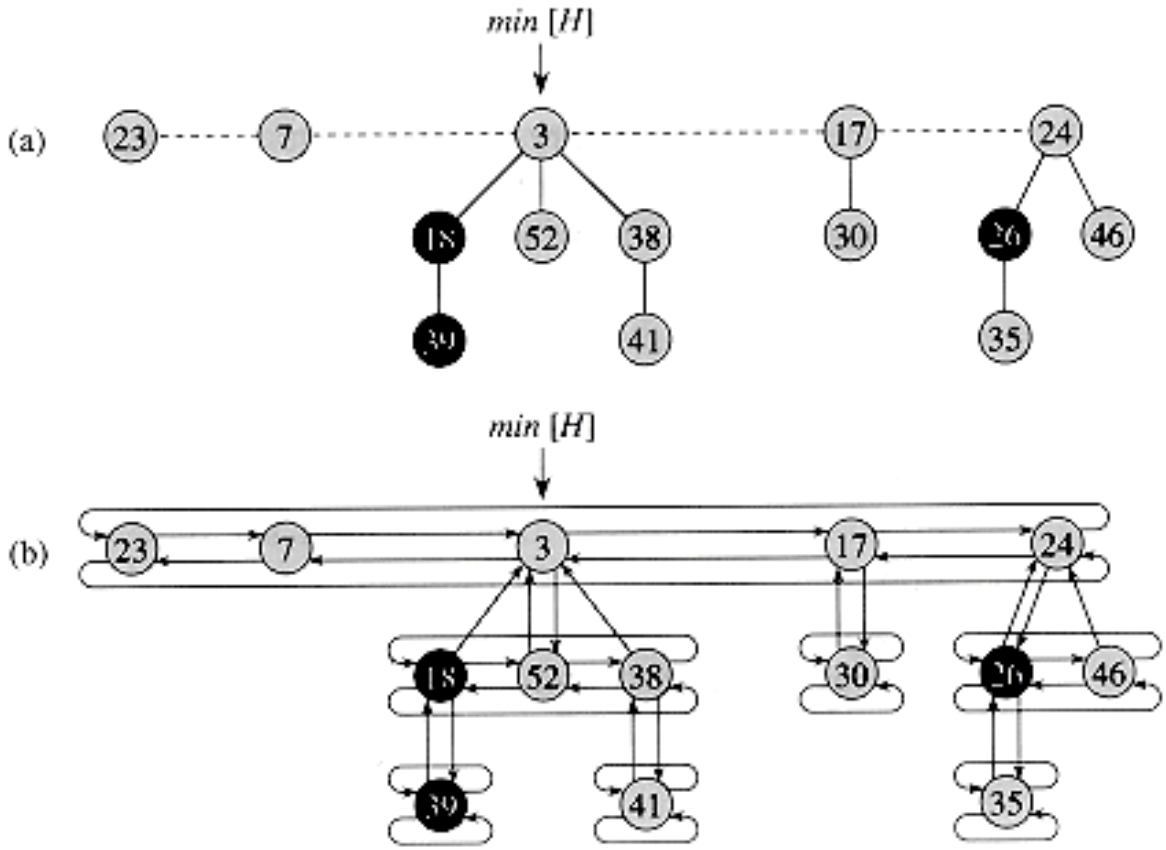
A Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap.

A Fibonacci heap is generalization of Binomial heap.

## 5.2. Structure of Fibonacci Heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees.

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. The pointer H.min thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list.




## 5.3. Test Cases for Fibonacci Heap

Fibonacci Heap should give the following outputs for the inputs provided :

In [ ]:
```
> fh = new FibHeap();
> fh->insert(new FibNode(3))
> fh->insert(new FibNode(10))
> fh->insert(new FibNode(100))
> fh->first
    3
> fh->isEmpty()
    0
> fh->Delete(3)
> fh->extractMin()
    10
> fh->first()
    10
> fh->Delete(10)
> fh->Delete(100)
> fh->isEmpty()
    1
```

## 5.4. Some operations on Fibonacci Heap which are implemented in our project

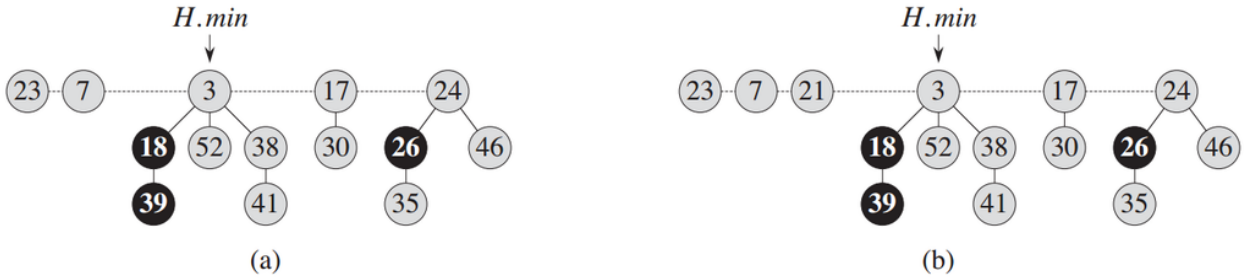
### 5.4.1. Make Heap

To make an empty Fibonacci heap, the Make_Heap procedure allocates and returns the Fibonacci heap object H, where H.n = 0 and H.min = NULL. There are no trees in H. Because $t(H)$ = 0 and $m(H)$ = 0. The amortized cost of Make_Heap is thus equal to its $O(1)$ actual cost.

### 5.4.2. Insertion in Fibonacci Heap

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that x.key has already been filled in.

In [ ]:
```
// Pseudo code to insert an element in Fibonacci Heap
1 Fibonacci_Heap_Insert(H,x)
2    x.degree = 0
3    x.p = NULL
4    x.child = NULL
5    x.mark = FALSE
6    if H.min == NULL
7        create a root list for H containing just x
8        H.min = x
9    else insert x into H's root list
10        if x.key < H.min.key
11            H.min = x
12    H.n = H.n + 1
```




To determine the amortized cost of Fibonacci_Heap_Insert, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H')$ = $t(H)$ + 1 and $m(H')$ = $m(H)$, and the increase in potential is (($t(H)$ + 1) + 2$m(H)$) - ($t(H)$ + 2$m(H)$) = 1

Since the actual cost is $O(1)$, the amortized cost is $O(1)$ + 1 = $O(1)$.

### 5.4.3 Extract min in Fibonacci Heap

The process of extracting the minimum node is the most complicated of the operations in fibonacci heap. This method finds the minimum element in heap, removes it. balances the root list such that no two elements in that list remains with same degree and points the min to the minimum element.

In [ ]:
```
// Pseudo code to extract the minimum node in Fibonacci Heap
1 Fibonacci-Heap-Extract-Min(H)
2    z = H.min
3    if z != NULL
4        for each child x of z
5            add x to the root list of H
6            x.p = NULL
7        remove z from the root list of H
8        if z == z.right
9            H.min = NULL
10        else H.min = z.right
11            CONSOLIDATE(H)
12        H.n = H.n - 1
13    return z
```

## 5.5 Comparision of Fibonacci Heap and Binomial with respect to Binary Heap

Below image shows the performance comparision of Binary heap with respect to Fibonacci heap and Binomial for all the priority queue operations. Which shows fibonacci heap is better than both implementations in most operations.

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

---

# 6. Implementation of Kruskal using van Emde Boas Tree

1) van Emde Boas data structure doesn't support duplicate keys or any related data except a bit representing that number. So to include multiple edges of same weight we have taken an unordered_map in which each key points to a vector which saves our edges.

2) Accessing map for a specific and the element from the vector takes an amortized $O(1)$ time

3) van Emde Boas tree takes a space of $O(u)$ so due to our machine's limitation we have restricted the edge weights to a max of $4 * 10^9$ i.e. $2^{25}$.

4) for Kruskal Algorithm we have used Union-By-Rank and Path-Compression heuristics, since it is the asymptotically fastest implementation known.

---

# 7. Complexity Analysis

1) Insertion in vEB tree takes $O(\log \log u)$ time and insertion in a vector takes an amortized $O(1)$ time. So, insertion of graph ( building vEB ) takes a total of $O(E \log \log u)$ time where $(\log \log u)$ can take a max value of 5. So input time is $O(E)$.

2) In Kruskal Alrorithm, for each edge we are making a set initially, and Make-Set takes $O(1)$ time. So, overall $O(E)$ time.

3) Again for each edge we are finding set if they are same we are looking for next edge otherwise we are merging them and including that edge in our result.

We know both operations of Disjoint-set data structure i.e. Find and Merge are bounded by Ackermann's Functions i.e. takes near constant time operation, i.e. $\alpha(n)$.

So, Overall complexity of our implemented Kruskal Algorithm is $O(E \log \log u)$ or $O(E)$ where E is the number of edges in the graph provided.

---

# 8. Comparison of vEB implementation with respect to Binomial and Fibonacci Heaps

Using Complexity analysis we have seen that the time complexity of Kruskal Algorithm implemented using below mentioned data structures are

-> Binomial Heap - $O(E \log E + E \log E)$

$E \log E$ due to insertion and another $E \log E$ due to extracting min V times and in worst case E times.

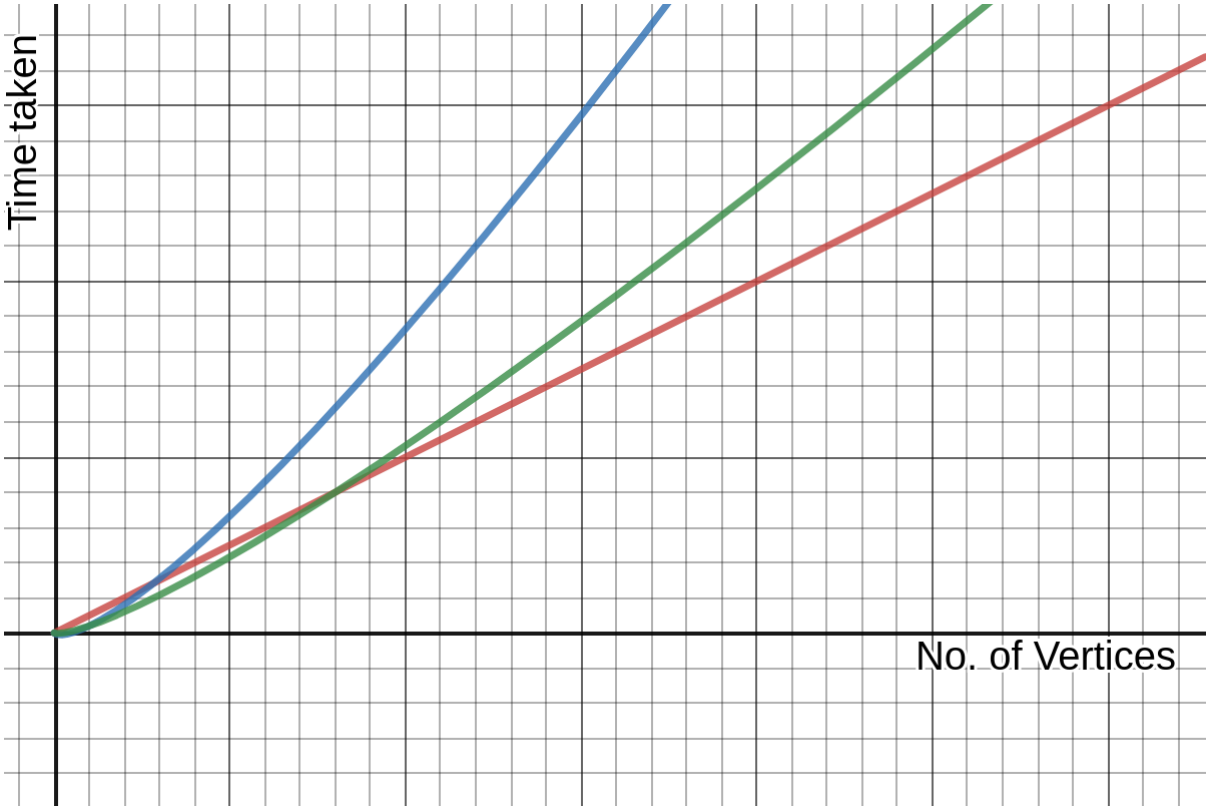
-> Fibonacci Heap - $O(E + E \log E)$

similar to Binomial Heap but insertion happens in $O(1)$, so $O(E)$.

-> vEB tree - $O(E \log \log u)$

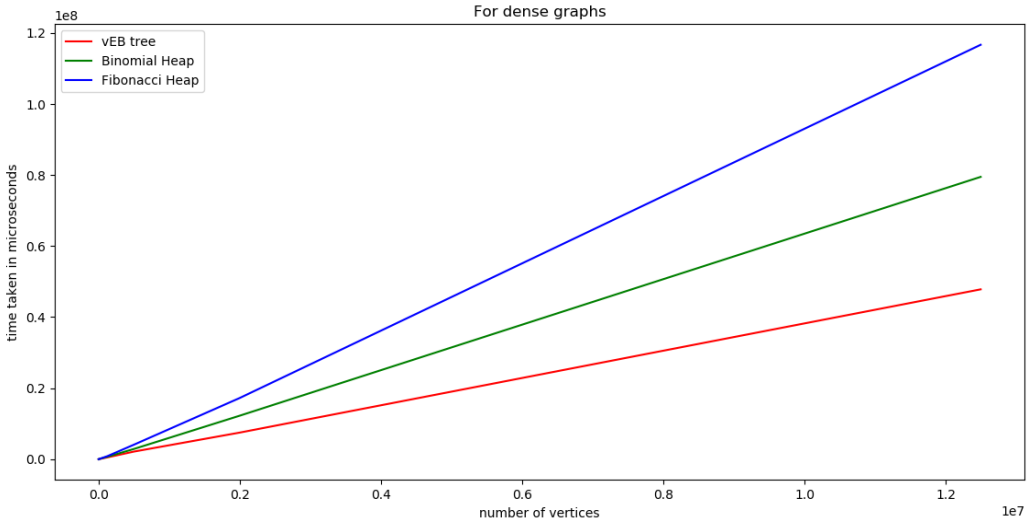
as explained above

So, using above equations graph should be obtained like this.




Graph Comparing Fibonacci heap, Binomial heap and vEB tree with respect to Kruskal's Implementation using above equation

By our implementations the graph obtained is as follows




Graph Comparing Fibonacci heap, Binomial heap and vEB tree with respect to Kruskal's Implementation

# 9. Limitations of vEB tree

vEB tree is definitely a very fast data strucutre if we look for priority queue operations and is used for various uses like Network Routers etc. but it as many limitations too.

- First being the space it takes, vEB tree takes almost exponential space which is very high than any other data structure of same type. For eg. even if we have only 2 elements i.e. 6000 and 67000 even then it will take a space of 4294967296 bits.

- Second it doesn't support duplicate keys.

- Third, We can't have negative keys in vEB tree

---

# 10. Improvements

- We can solve the space limitation by improving our data structure. We can upgrade it to Y-fast trie. It takes amortized same time for all the operations as taken by vEB and takes linear space.

- To allow duplicate elements we have to take support of some another data strucutre like we have taken unordered map into account.

---

# 11. References


- Introduction to Algorithms- Thomas H. Cormen
- MIT 6.046J Design and Analysis of Algorithms, Spring 2015 (https://www.youtube.com/watch?v=hmReJCupbNU)
- University of Cambridge - Algorithms II (https://www.cl.cam.ac.uk/teaching/1314/AlgorithII/2013-stajano-algs2-students-handout.pdf)
- Stanford CS 166 lectures (http://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/14/Small14.pdf)