Study implementation of the van Emde Boas Tree with application to Kruskal and compare wrt RB and AVL Trees

Team Members

- Aman Joshi - 2018201097
- Shubham Rawat - 2018201098

Deliverables

- van Emde Boas Tree
- Kruskal's algorithm
- Red-Black Tree
- AVL Tree
- Comparison of time and space complexities
- An API for vEB, AVL, RB trees

Project Delivery Plan

- Van Emde Boas tree that supports Insertion, Deletion, Lookup in $\boldsymbol{\theta}$ ***(log log n)*** Minimum and Maximum element in $\boldsymbol{\theta}$ ***(1)***.
- AVL Tree that supports Insertion, Deletion, K^{th} element, Search in $\boldsymbol{\theta}$ ***(log n)*** time.
- Red Black trees that support Insertion, Deletion, Search in $\boldsymbol{\theta}$ ***(log n)*** time.
- Kruskal's Algorithm for finding the minimum spanning trees of the given Graph. It uses Disjoint-Set Union data structure for purpose of union and finding set.
- Implementation of Disjoint-set data structure

Technologies to be used

- C++, Python for testing.

Online Resources

- https://en.wikipedia.org/wiki/Van_Emde_Boas_tree
- https://en.wikipedia.org/wiki/AVL_tree
- https://en.wikipedia.org/wiki/Red-black_tree
- https://mcdtu.files.wordpress.com/2017/03/introduction-to-algorithms-3rd-edition-sep-2010.pdf

Repository where work is being committed

- https://github.com/amanjoshi668/Trees

Van Emde Boas Tree:

A Van Emde Boas tree is a tree which implements an associative array of m-bit integer keys. It performs all operations in ***θ (log m)*** time or equivalently in ***θ (log log n)*** where **n = 2^m**, is the maximum number of elements that can be stored in the tree. The vEB tree has good space efficiency when the number of elements is comparable to n.

Given a fixed integer n, the best way to represent a subset S of {0, . . ., n − 1} in memory, here space is not our concern. We use an n-bit array A, setting A[i] = 1 if and only if i ∈ S. This solution gives $\theta(1)$ running time for insertions, deletions, and lookups (i.e., testing whether a given number x is in S). We want to be able not just to insert, delete and lookup, but also to find the minimum and maximum elements of S. On a bit array, these operations would all take time $\theta(|S|)$ in the worst case, as we might need to examine all the elements of S. We have to do extract min, so first we have to find the minimum element by using MIN algorithm and then delete that element using delete function.

Van Emde Boas tree can perform insertion, deletion and lookup in θ *(log log n)*. While it can perform Min and Max operations in θ *(1)*.

Operation	Bit array	van Emde Boas
INSERT	$\theta(1)$	$\theta(log\ log\ n)$
DELETE	$\theta(1)$	$\theta(loglog\ n)$
LOOKUP	$\theta(1)$	$\theta(loglog\ n)$
MAX, MIN	$\theta(n)$	$\theta(1)$

Structure of Node used for vEB Tree.

```
struct vEB_Node
{
    long long u;                    // the size of universe for this node
    long long min_elem;             //minimum element of the node
    long long min_cnt;              //count of the minimum element
    long long max_cnt;              //count of the maximum element
    long long max_elem;             //maximum element of the node
    vEB_Node *summary;              //summary of the cluster
    vector<vEB_Node *> cluster;  //pointers to cluster nodes
}
```

The u defines the number of unique elements the current node can represent. Min elem for storing minimum element. Min count is for storing the count of minimum element. Similarly, $\sqrt{u}$ clusters. Each cluster then recursively store the information about the elements. The minimum element is stored only in parent node. It is not propagated in the tree.
The summary is a vEB node that store the summary of the clusters of the current node. If any element is present in a cluster than its entry in summary is marked as one irrespective of the numbers of elements in the cluster.

Features of vEB Tree:

- Minimum element is stored only at root.
- Rest of the elements are stored in one of the $\sqrt{u}$ clusters.
- We are also storing the count of the elements at each level.
- Each element is present only at one place either leaf node or the node where it is minimum among all the entries.
- However, in spite of having count summary node contain only one entry for a cluster.
- The size of universe is chosen of form $u = 2^{2^k}$
- The base case is when the node size is $u = 2$ then the node contains only min and max element with their respective counts.

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 ***θ (log n)*** time. We have also implemented the K^{th} element in ***θ (log n)*** time.

Structure of AVL Tree Node:

```
struct AVL_Node{
   long long value;          //Data of the node
   long long height;         //Height of the node
   long long weight;         //Weight of the node
   AVL_Node* left;           //Pointer to the Left Subtree
   AVL_Node* right;          //Pointer to right subtree
   AVL_Node* parent;         //Pointer to parent of the node
   AVL_Node (long long k);   //Constructor
};
```

The above node structure was used for implementing AVL Trees. Weight was used for finding the K^{th} element in ***θ (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 ***θ (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

and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

```
struct RB_Node
{
   long long data;          // Data of the node
   bool leaf;               //Whether the node is leaf or not
   COLOR color;             //Color of the Node
   RB_Node *par;            //Parent of the Node
   RB_Node *left;           //Left Children of the Node
   RB_Node *right;          //Right Children of the Node
   RB_Node(long long);      //Constructor
};
```

Properties of Red Black Tree:

- Every node is either red or black.
- Root is Black
- Every leaf (`NIL`) is black.
- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes.
- During Insertion and Deletion these properties may get violated which we restore with the help of rotations.

Kruskal's Algorithm

Kruskal is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It finds a minimum spanning tree for a connected weighted graph. At every step it keeps on adding edges in increasing orders. It only adds the edge to the tree if both the vertices connected by this edge are in different sets i.e. they are not already connected. For this we have used Disjoint-set Data Structure. 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).

This algorithm can be shown to run in $O(E\ logE)$ time, or equivalently, $O(E\ logE)$ time, where E is the number of edges in the graph and V is the number of vertices, all with simple data structure. These running times are equivalent because:

- E is at most V^2 and $log\ V^2 = 2log\ V$ is $O(log\ V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V <= 2E$, so $log\ V$ is $O(log\ E)$.

Disjoint-Set Data Structure:

A disjoint-set data structure is a data structure that tracks set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides near constant-time operations. The operations supported are Union-Set, Find-Set, Same-Set. They have a wort case time complexity of $\boldsymbol{\theta}(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function, it grows very slowly thus providing a near constant time working.

```
struct UnionFind

{

   vl pset; // It is adjustable in constructor

   UnionFind(long long _size){ pset.resize(_size + 2); REP(0, _size+2)pset[i] = i;}

   long long find_set(long long i) { return (pset[i] == i) ? i : (pset[i] = find_set(pset[i])); }

   void union_set(long long i, long long j) { pset[find_set(i)] = find_set(j); }

   bool is_sameSet(long long i, long long j) { return find_set(i) == find_set(j); }

};
```

We have used Path compression technique for the Set. Merging by rank doesn't lead to any significant improvement on the graph size we are working on. Rather it works same because of the extra operations it needs to perform.

Challenges Faced:

Maintaining Multiple values:

- By default, vEB tree support only unique values. Modifying it for storing multiple values took lot of debugging time. For that we were storing count of the elements as well.
- The minimum element is not propagated in the tree. So whenever the new element with smaller value is inserted.
- The minimum element of the node needs to be updated. By default only swapping the min value of the node with the inserted value will complete the task.
- But with multiple values support we have to propagate the count of the element as well.
- However, the real challenge faced was in figuring out that summary node will contain only one entry for any number of occurrences of any element in a cluster. We were constantly updating the count of element in summary by the count we were propagating in the tree.
- The other challenge was that we were not updating the max count of node. We kind of figured this one out later.

Deletion in RED-BLACK Tree:

- Deletion in red black tree took a lot of time.
- It has multiple cases to handle.
- The problem faced was that the rotation algorithms were sometimes changing the parent pointer of Leaf node.
- We used only a single leaf node. So, it was affecting the insertion and deletion on whole.

Testing:

Submitting on online platforms:

- This was for proof of the correctness of the implementations of each of the data structures.
- We submitted our Kruskal's implementation which were using these data structures for choosing the minimum weighted edge.
- All three of the Solutions got accepted.

Comparison Among the three data structures:

For comparison we are randomly generating graphs. We have written a C++ code which takes two parameters (n, m) number of vertices and edges in graph. Then it is generating the graphs randomly. After doing all the preprocessing like initializing the Union Set, adjacency list. The time is calculated for all three implementations.

- AVL Trees performed better than both Red Black Tree and Van Emde Boas tree on smaller graphs.
- Our Red Black tree Implementation is working slow. So, we also compared it with already implemented red black trees in C++, i.e. Multisets.
- Multisets outperforms other data structures up to 10^5 edges.
- With increasing the number of edges vEB tree's implementation outperform every other data structure.
- They were almost 10 times faster than other data structure for 10^6 edges.
- They were almost 80 times faster for 10^7 edges.

C++ file is used to generate 150 different combinations and result are stored in a text file. Later graph is plotted using python script.