DATE: 13/04/2022

Advanced Data Structures & Algorithms
Subject Code: *CSPE43*

TOPIC: Analysis on the Optimization of Graph Storage using different data-structures.

Submitted By – Group No 3

- Arpon Kapuria
  Roll No: 106120014
  arpkapuria@gmail.com
- Ashwin A
  Roll No: 106120018
  Ashwinskap65@gmail.com

Table of Contents:

ABSTRACT: Graph is a widely used data structure and it is used mostly in data mining, social networking sites comprising of a lot of vertices and edges. To store this large amount of data, it takes a lot of storage. This analysis is on optimizing this graph storage on large social networking sites. We'll be discussing on different data structures and how they are used to store the graph and how they can be optimized. We need to optimize this because of a lot of data are there and they are connected through vertices and edges. If we look at a normal adjacency matrix representation of graph, it takes a lot of storage and also it uses 0 and 1's where 0 simply denies the connection. So If we optimize them we can find a better solution and can store them using less space. But there are drawbacks to this system. We'll be starting from Adjacency List then we will optimize it to treap and then to binary tree by improving the large scale treap problems. The different algorithms showed in each data structure section proves that they are efficient and improved than previous one. We'll be evaluating these improvements using time complexity, space complexity, advantages, disadvantages, accuracy, precision & implementations different metrices.

Keywords: Adjacency matrix, Adjacency List, Treap, Binary tree

Introduction

The internet and social media are two rapidly growing sectors of the tech world, with billions of users utilizing it for various purposes every second. On the other hand, we have computer systems doubling their processing capabilities on a yearly basis and require the corresponding software components to maximize the availability of resources with optimal scheduling algorithms.

Problem

The point of commonality here is the usage of Graphs as a means to represent and store data in its nodes, vertices and edges. Graphs are nearly used in every field of computer science these days, and hence it has become a necessity to optimize Graph Storage.

Existing Solutions & their limitations

The data structure most commonly used for storage of data through Graphs, was the Adjacency Matrix, which is a nxn matrix for a graph containing n-nodes. If an edge exists between any two given vertices, then it is marked as a 1 in the matrix, else, a 0. They usually comprise of two components – an array to store all the entries of a matrix and a pointer to an array that takes care of larger elements.

This may work on a small scale, but when we scale this up on to an extent where there are billions of social media/internet users, the graphs become extremely big, and it is not feasible to store this data in the form of an adjacency matrix. If a user has a link with a hundred people and no links with the remaining billion users, then the adjacency matrix would contain a hundred 1s and a billion 0s, which seems like an excessive wastage of memory. Also, other than the waste of memory, adjacency matrices are static arrays, and so insertion and deletion operations [deleting one node would mean shifting billions of other nodes], which need to be done dynamically for something like social media, is hindered greatly on an adjacency matrix.

Solutions

One way of optimizing this is to map the graph itself onto a different data structure, one that requires less memory and is faster. We analyzed three data structures that fit this criterion and improved the efficiency with which we stored Graph data. They are:

- **Adjacency List**
- **Treap**
- **Binary Trees**

Adjacency List representation of a graph is one of the closest alternatives to an Adjacency Matrix, where only the edges that are linked to each vertex are stored in the form of a list, and the entire array of lists serves as a mapping for the entire graph. It is widely used in the networking sector.

The graphs that were mapped onto a binary tree data structure, were found to have reduced the amount of memory required and would be ideal for storing data related to social networking, railway networks, and chemical compounds.

The graphs that were mapped onto the treap data structure which is a random binary tree with two factors, a key value, and a priority value, for tree and heap respectively; it avoided storing the 0s in an adjacency matrix and was hence ideal for social networking sites. It also grew dynamically in response to the addition of new elements, without affecting the existing structure.

The three data structures mentioned above have been tested and analyzed in the following pages.

Adjacency List

Adjacency Lists are a form of graph representation in which each vertex is allotted a separate list which is a chain of all the nodes that have an edge with the particular vertex.

Real-Time Application

Adjacency Lists are frequently used as a representation of networks, and since networks are generally undirected [their connections are mostly two-way], the order in which they are listed becomes irrelevant and thus ideal to be used as an adjacency list. Adjacency Lists also perform considerably better than Adjacency Matrices in many scenarios.

Requirements & Limitations

The requirements that an adjacency list is supposed to follow are:

- Each vertex that is connected to a vertex u must be shown in the adjacency list of vertex u.
- The length of a list cannot exceed V-1, where V is the number of vertices.

Queries like whether an edge exists between two given vertices, take O(V) time, which is inefficient.

Why is this a solution to our problem?

Take, for instance, this graph with 12 nodes in figure 2.1:

The adjacency matrix, as shown above, has numerous 0s that ae stored at the cost of memory. This can be mapped onto a smaller scale by ignoring the 0s in the matrix and simply taking the existing edges. That has been noted down in the form of an adjacency list.

Similarly, any large and sparse adjacency matrix can be reduced to an adjacency list and consequently into a smaller and more efficient data structure representation. Hence this is valid to serve as a solution to our objective to optimizing graph storage.

Algorithm

```
insertion(source,dest){
if(source list does not exist){
        newnode->next=source_list.head;
        source_list.head=newnode;
}else{
check=source_list.head;
while(check->next NOT EQUAL TO NULL){
        check=check->next;
}
check->next=newnode;
}
if(dest list does not exist){
        newnode->next=dest_list.head;
        dest_list.head=newnode;
}else{
check=dest_list.head;
while(check->next NOT EQUAL TO NULL){
        check=check->next;
}
check->next=newnode;
}
}
```

Example

Take, for instance, this graph with 12 nodes:

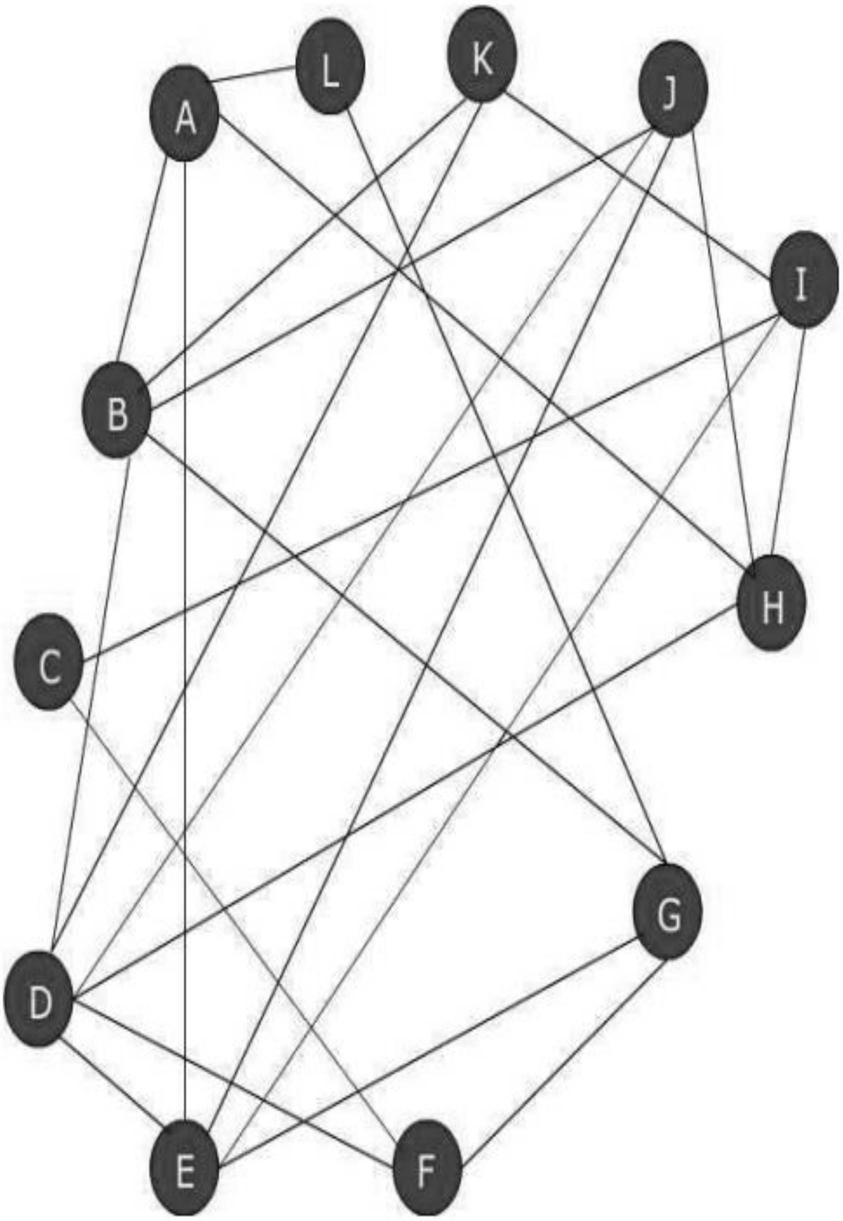
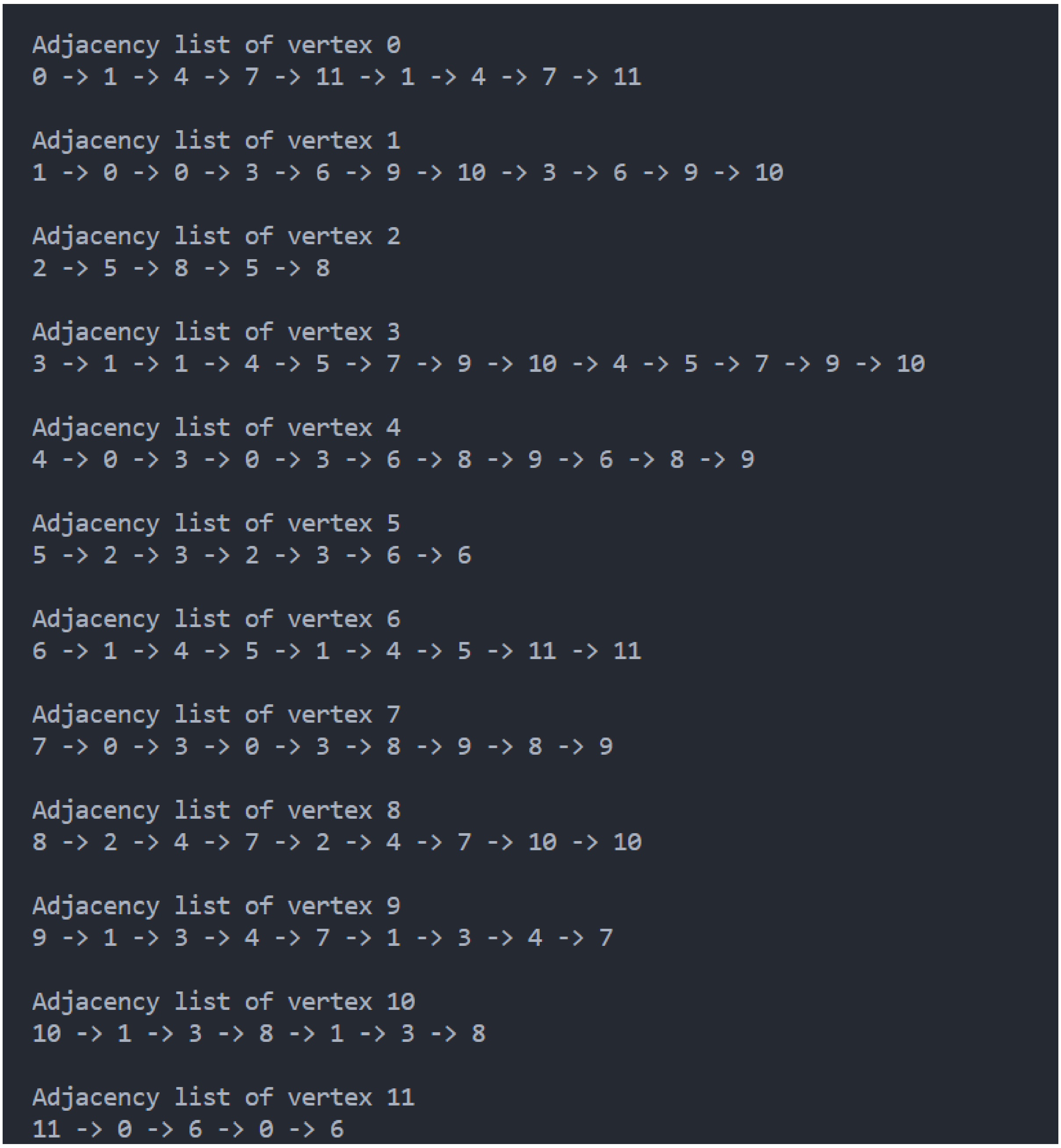


Figure 2.1

Given below is an implementation of the graph mapped through an adjacency list that we saw above. **[Here A to L = 0 to 11]**



```
Adjacency list of vertex 0
0 -> 1 -> 4 -> 7 -> 11 -> 1 -> 4 -> 7 -> 11

Adjacency list of vertex 1
1 -> 0 -> 0 -> 3 -> 6 -> 9 -> 10 -> 3 -> 6 -> 9 -> 10

Adjacency list of vertex 2
2 -> 5 -> 8 -> 5 -> 8

Adjacency list of vertex 3
3 -> 1 -> 1 -> 4 -> 5 -> 7 -> 9 -> 10 -> 4 -> 5 -> 7 -> 9 -> 10

Adjacency list of vertex 4
4 -> 0 -> 3 -> 0 -> 3 -> 6 -> 8 -> 9 -> 6 -> 8 -> 9

Adjacency list of vertex 5
5 -> 2 -> 3 -> 2 -> 3 -> 6 -> 6

Adjacency list of vertex 6
6 -> 1 -> 4 -> 5 -> 1 -> 4 -> 5 -> 11 -> 11

Adjacency list of vertex 7
7 -> 0 -> 3 -> 0 -> 3 -> 8 -> 9 -> 8 -> 9

Adjacency list of vertex 8
8 -> 2 -> 4 -> 7 -> 2 -> 4 -> 7 -> 10 -> 10

Adjacency list of vertex 9
9 -> 1 -> 3 -> 4 -> 7 -> 1 -> 3 -> 4 -> 7

Adjacency list of vertex 10
10 -> 1 -> 3 -> 8 -> 1 -> 3 -> 8

Adjacency list of vertex 11
11 -> 0 -> 6 -> 0 -> 6
```

Treap

As the name suggests, treaps are a combination of tree and heaps that can be used as a replacement for the inefficient adjacency matrices. Each node in a treap contain both a key and a priority, and a factor of randomness, together achieve a binary search tree that has a high probability of being balanced. The height of the treap itself is proportional to the logarithm of the number of keys, so any insertion or deletion operation takes (log n) time to perform.

Real-Time Application

Treaps are used in a large number of fields, especially in the construction of a randomized binary search tree. It is also useful in the creation of a Cartesian tree, which are very useful in range queries and range searching operations. They can be used in a sorting algorithm called as the randomized quicksort, which takes O(nlogn) time, or they can be used to implement skip lists, which are a form of sorted linked lists.

Requirements & Limitations

The requirements that a treap is supposed to follow are:

- If the key value of the child is less than the parent, then it should be placed to the left of the parent, and right if not.
- The nodes should also be arranged in order of priority; in case of a min heap, elements with the smallest priority should occupy the highest positions of the treap.
- Every sub-tree and sub-sub-tree itself will satisfy the property of a treap.

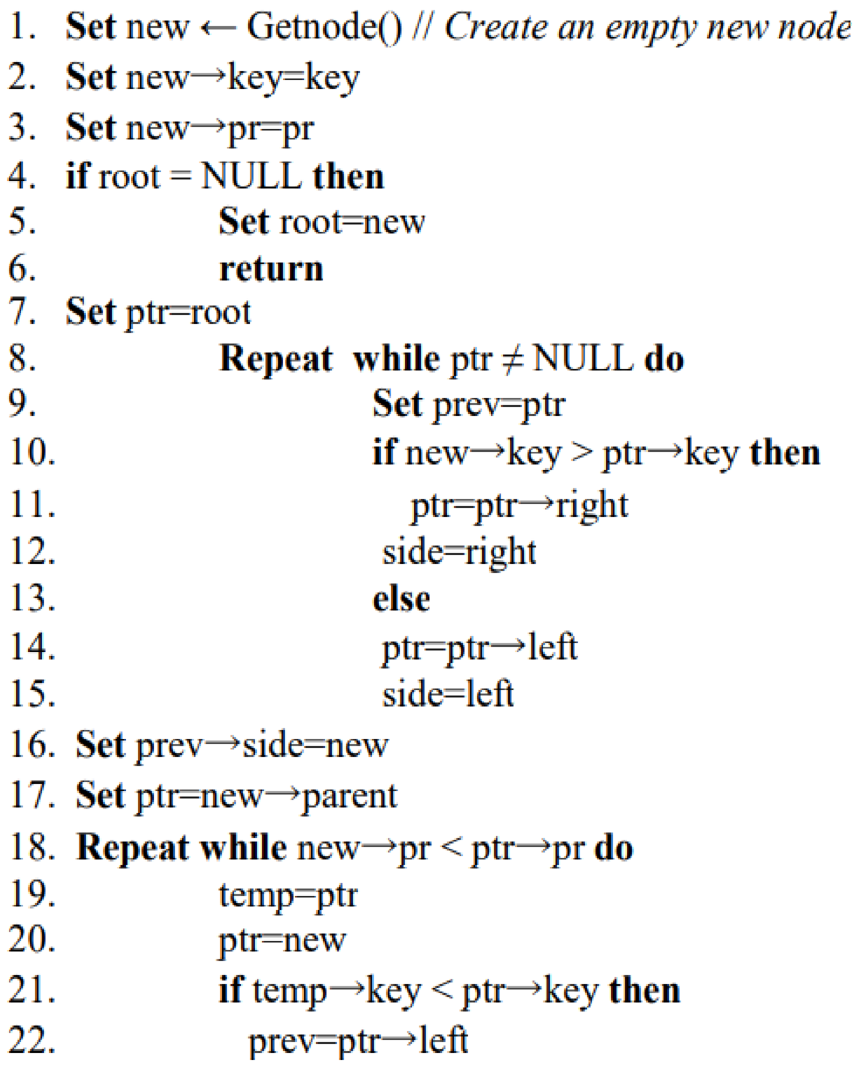
The limitation of a treap is that by removing the storage that is wasted in holding the 0s of an adjacency matrix, it is forced to use extra storage to save the nodes that are required for the process.

Why is this a solution to our problem?

The adjacency matrix, as shown in the example section, has numerous 0s that ae stored at the cost of memory. This can be mapped onto a smaller scale by taking the name of each node as the key and the number of edges it has as its priority, i.e, the number of 1s in their adjacency matrix row. That has been noted down in the form of an adjacency table and it is now very easy to represent this table in efficient time and memory through a treap.

Similarly, any large graph can be reduced to an adjacency table and consequently into a smaller and more efficient data structure representation.

Algorithm

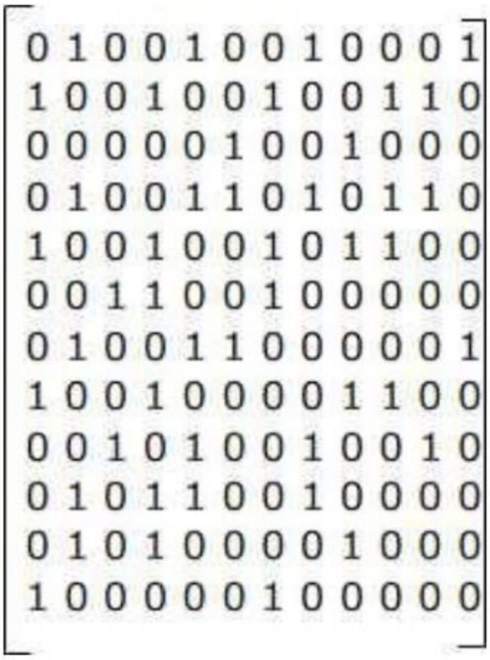


```
Set new ← Getnode() // Create an empty new node
Set new→key=key
Set new→pr=pr
if root = NULL then
        Set root=new
        return
Set ptr=root
        Repeat  while ptr ≠ NULL do
                Set prev=ptr
                if new→key > ptr→key then
                  ptr=ptr→right
                 side=right
                else
                 ptr=ptr→left
                 side=left
Set prev→side=new
Set ptr=new→parent
Repeat while new→pr < ptr→pr do
        temp=ptr
        ptr=new
        if temp→key < ptr→key then
           prev=ptr→left
```

```
23.             ptr→left=temp
24.             if prev→key < temp→key then
25.                 temp→left=prev
26.             else
27.                 temp→right=prev
28.         else
29.             prev=ptr→right
30.             ptr→right=temp
31.             if prev→key < temp→key then
32.                 temp→left=prev
33.             else
34.                 temp→right=prev
35. go to step 16 until new→key < new→parent→key
36. return
```

Example

Take, for instance, this graph of figure 2.1. The adjacency matrix is given below:



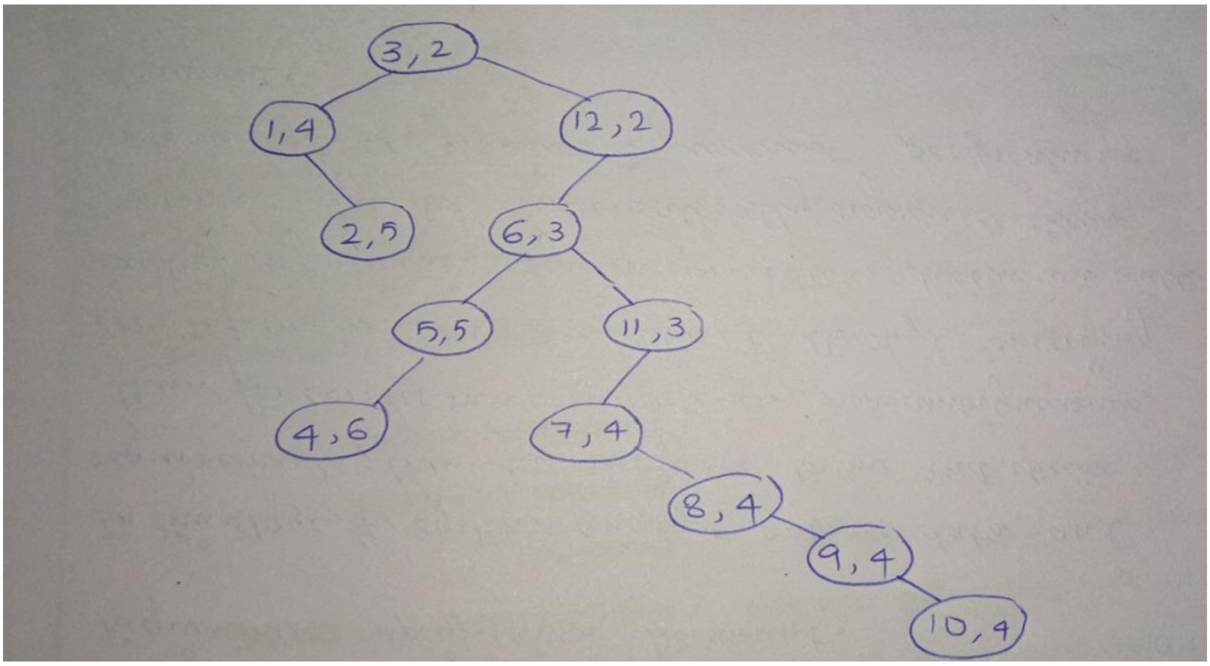
$$\begin{bmatrix}
0&1&0&0&1&0&0&1&0&0&0&1\\
1&0&0&1&0&0&1&0&0&1&1&0\\
0&0&0&0&0&1&0&0&1&0&0&0\\
0&1&0&0&1&1&0&1&0&1&1&0\\
1&0&0&1&0&0&1&0&1&1&0&0\\
0&0&1&1&0&0&1&0&0&0&0&0\\
0&1&0&0&1&1&0&0&0&0&0&1\\
1&0&0&1&0&0&0&0&1&1&0&0\\
0&0&1&0&1&0&0&1&0&0&1&0\\
0&1&0&1&1&0&0&1&0&0&0&0\\
0&1&0&1&0&0&0&0&1&0&0&0\\
1&0&0&0&0&0&1&0&0&0&0&0
\end{bmatrix}$$

Figure 3.1

Priority and adjacency nodes are:

A	B	C	D	E	F	G	H	I	J	K	L
4	5	2	6	5	3	4	4	4	4	3	2
B,E,H,L	A, D, G, J, K	F,I	B, E, F, H, J, K	A, D, G, I,J	C,D,G	B,E,F,L	A,D,I,J	C,E,H,K	B,D,E,H	B,D,J	A,G

This is the final tree that we get as output. It is ordered based on both priority and key values. **[Here A to L = 1 to 12]**



Binary Tree

A binary tree is a tree data structure in which each node has up to two child nodes that create the branches of the tree. The two children are usually referred to as the left and right nodes. Parent nodes are nodes with children, while child nodes can contain references to their parents. The topmost node of the tree is called the root node, the node to the left of the root is the left node which can serve as the root for the left sub-tree and the node to the right of the root is the right node which cans serve as the root for the right sub-tree.

Real-Time Applications

We can use a binary tree to store data in hierarchical order. Trees are faster to insert and delete than arrays and linked lists. The binary tree can contain any number of nodes and is dynamic in size. These are typically used for sorting, but in this case, it is a faster form of search. Almost all database (and database-like) programs use binary trees (or similar) to implement and indexing system. The other is a quite a decision tree that has been used frequently. In computing, binary trees are mainly used for searching and sorting as they provide a means to store data hierarchically. Some common operations that can be conducted on binary trees include insertion, deletion, and traversal. Other real before applications are like binary space partition, heap sort, Huffman coding, virtual memory management, and indexing.

Requirements and Limitations

Some basic properties of a binary tree are given below:

- If the root level is zero, the binary tree can have up to 2^l at level l.
- If each node in the binary tree has one or two children, the number of leaf nodes (nodes without children) is one more than the number of nodes with two children.

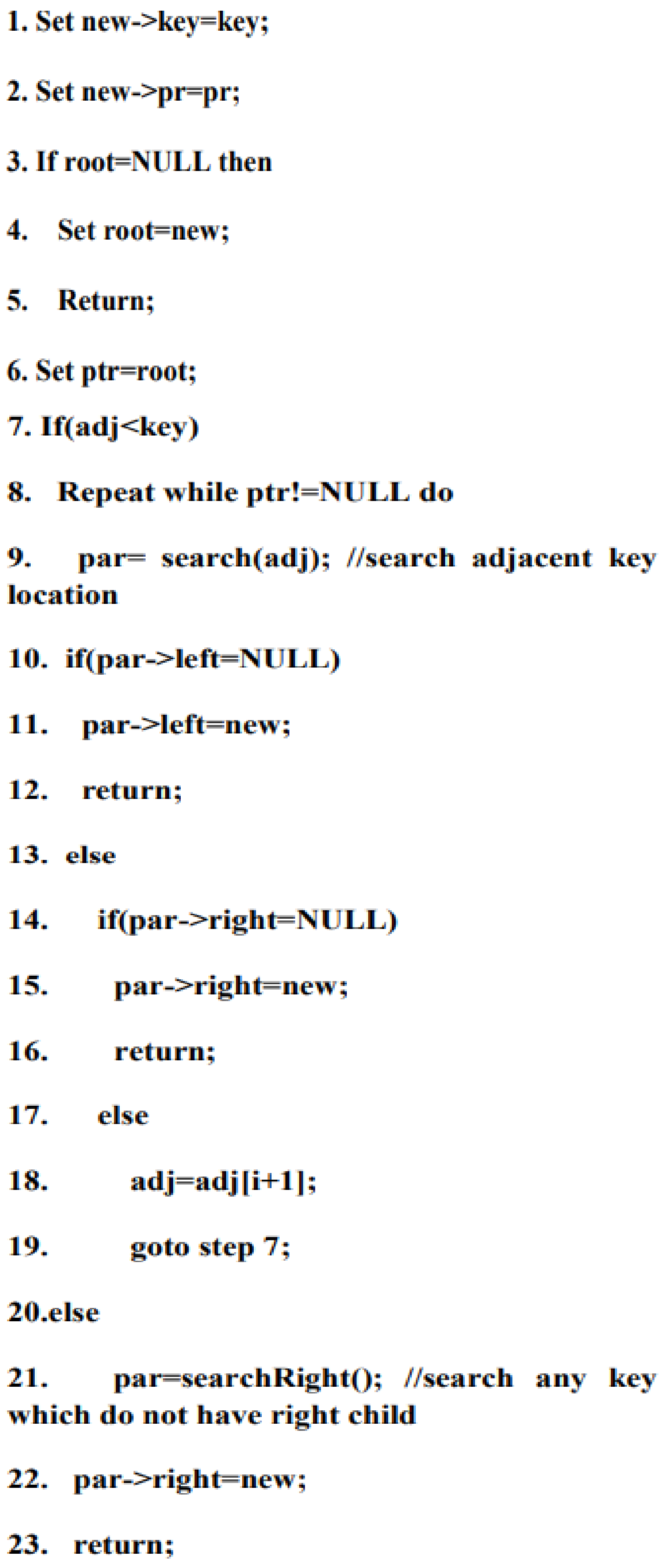
- If the height is h and the height of the leaf node is 1, then binary tree has up to $(2^h - 1)$ nodes.
- If the binary tree has L leaf nodes, then there are at least L+1 levels.
- A binary tree with *n* nodes has a minimum number of levels or a minimum height of $log_2(n + 1)$.
- The minimum and maximum heights of *n* node binary tree are ceiling value of $log_2 n$ and n respectively.
- A binary tree of *n* nodes has *(n+1)* null references.

Limitations are like deleting nodes is a complex procedure, is overkill for every small number of elements. Insertion, deletion, and search operations are dependent on the height of the tree.

Why is this a solution to our problem?

When we use adjacency matrix to store graph, the entry in matrix is 1 if there is an edge between the nodes and is 0 if there is no edge is present. It takes very large space in main memory and database for storing a large graph of thousands of nodes as these are present currently in the social networking sites. So, it is wastage of memory to store 0's for no data present. Also in adjacency matrix, insertion in static arrays requires changing the size of the array dynamically. Deletion requires moving all elements to shrink the array. Then there is another data structure Treaps, which we showed in previous section. Here the problem is if mapping the complete tree using data structure then it takes more space than adjacency matrix also. It is specially designed for social networking sites in which it is always true that the probability of complete graph is negligible. So, a new approach is mapping the graph into a binary tree which reduces space almost half of the treaps. Basically, treaps removes only zeros from adjacency matrix but also require more space to store the nodes.

Algorithm



```
1. Set new->key=key;
2. Set new->pr=pr;
3. If root=NULL then
4.    Set root=new;
5.    Return;
6. Set ptr=root;
7. If(adj<key)
8.    Repeat while ptr!=NULL do
9.       par= search(adj); //search adjacent key location
10.   if(par->left=NULL)
11.      par->left=new;
12.      return;
13.   else
14.      if(par->right=NULL)
15.         par->right=new;
16.         return;
17.      else
18.         adj=adj[i+1];
19.         goto step 7;
20.else
21.      par=searchRight(); //search any key which do not have right child
22.   par->right=new;
23.   return;
```

Here, Key value of node is ‘key’.

Priority of node as ‘pr’.

Root node as ‘root’.

Adjacent key as ‘adj’ = adjacent[i] where i approach from 0 to number of adjacent nodes to that key.

Example

Let G be a graph having V a set of vertices and E a set of edges. Let’s consider this graph.

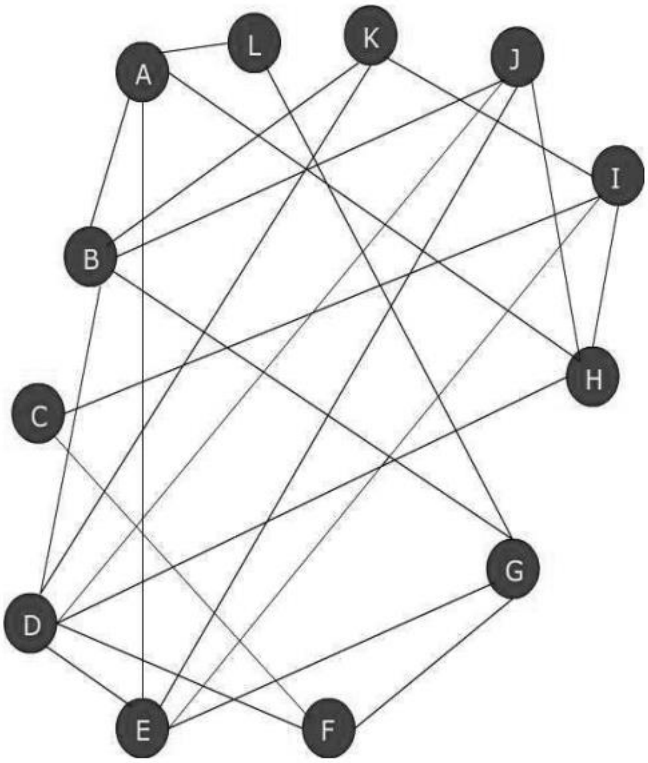


Figure 4.1

The corresponding adjacency matrix for above graph would be 12*12 matrix as there are 12 nodes in V and which is a static array very difficult to insert node and delete the existing one.

it.

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Figure 4.2

Structure of proposed node:

Each node stores the information of key, it's priority, pointer to the array of adjacent nodes, pointer to the sibling node, pointer to the left child and right child as shown in figure 4.3.

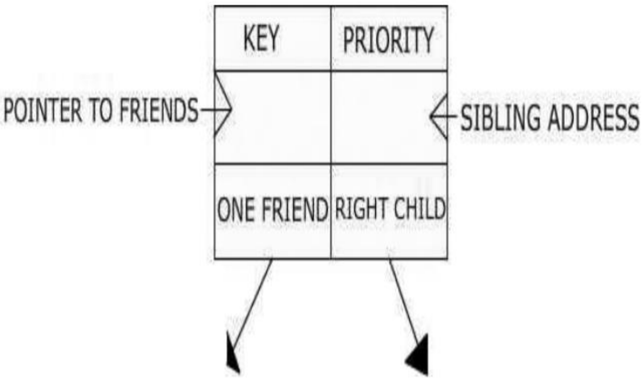


Figure 4.3

Below table shows priority of different nodes and adjacent nodes to it.

A	B	C	D	E	F	G	H	I	J	K	L
4	5	2	6	5	3	4	4	4	4	3	2
B,E,H,L	A, D, G, J, K	F,I	B, E, F, H, J, K	A, D, G, I,J	C,D,G	B,E,F,L	A,D,I,J	C,E,H,K	B,D,E,H	B,D,J	A,G

Insertion of nodes:

From the table, key value is 'A' and priority is 4. So as root is NULL so 'A' is inserted in the root as shown below.

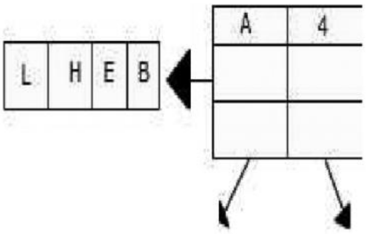


Figure 4.4

Next node is B and is friend of A which have available left pointer so it will be left of A and B deleted from list of A as shown below.

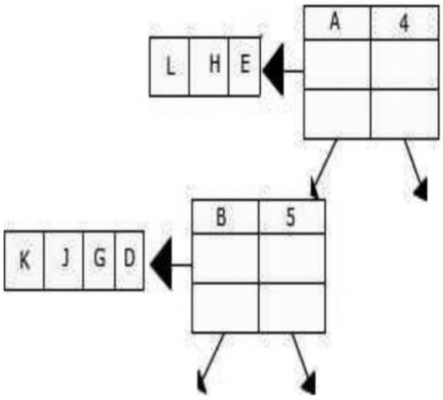


Figure 4.5

Now next node is C have friends which have higher key value so it can be in right of A.

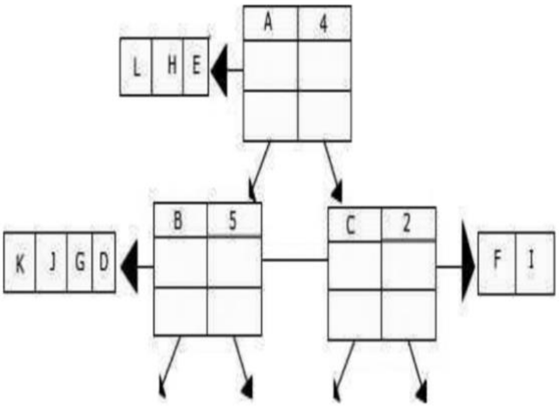


Figure 4.6

Similarly, all the nodes inserted in the tree the conditions are only as following:

- If a node has to insert and root is empty, then insert it on the root.
- Tree is Min heap with key values Minimum value of key is at the top and others at the bottom, it also helps in searching the node.
- If a node has friend have all the child then check for next friend else it can be inserted first in left then in right.
- For checking the list of friend first go to list of adjacent friends, if it has less number of friends then priority, then check its left child, if it is present then it also it's friend. If again it is less in number then go check whether it is left of its parent, if it is than parent is also a friend, if again the number of friends is less than priority then check it's right friend. Calculation priority and finding the friends in the tree is as follows:

  1. Priority = Number of elements in adjacent friend list of node.

  If it is less then,

  2. Priority = Number of friends in adjacent list + Left child.

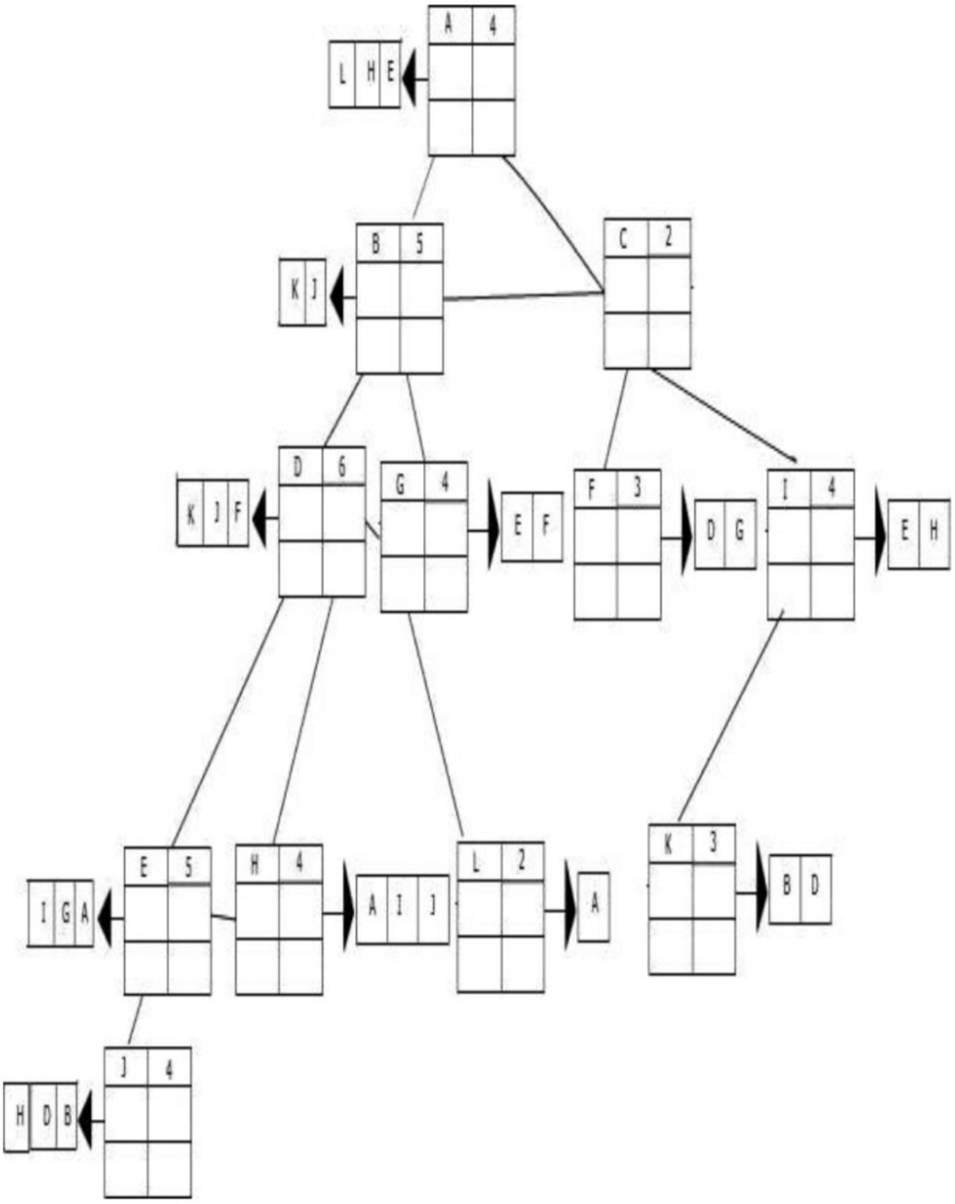
  If again it is less then,

3. Priority = Number of friends in adjacent list + Left child + Parent node (if node is left child of its parent).

If again it is less,

4. Priority = Number of friends in adjacent list + Left child + Parent node (if node is left child of its parent) + Right child.

The final tree is mapped as shown below -



Comparison

Comparing the three data structures we have on a basis of different metrics:

Time Complexity

The time complexity of the adjacency list mapping of a graph is O(E), where E is the number of edges present in the graph. The worst-case scenario would be a complete graph, which has the maximum number of operations. Insertion operations take O(1) time, and hence it is very efficient. Deletion of a vertex in an adjacency list is inefficient, as all the other lists need to be iterated through, and hence it takes $O(V^2)$ time. Deletion of an edge is relatively less time-consuming, taking O(V) time as it is required to traverse the entire list to find the edge. Searching for the presence of an edge, on the other hand, while being constant in an adjacency matrix, is linear in an adjacency list, O(V), and hence is expensive.

The time complexity of the binary tree mapping of a graph, has insertion operations that take O(V) time, since it is needed to traverse the entire tree to perform the comparisons and find the right position. Searching operation also requires the traversal of all its elements, and hence it takes O(V) time. Deletion of a node requires that the node be found initially and later deleted, and since searching operations take O(V) time, this too has a worst-case time complexity of O(V) time.

The treap mapping of a graph has a search operation that is very similar to BST searching of a node, and hence it has a time complexity of O(logV). Insertion operations deal with the creation of the node, performing comparisons similar to BST insertion to find the right position of the node, and finally rotating the graph accordingly in order to balance the graph, and so this too has a worst-case time complexity of O(logV). Deletion of a node is relatively straightforward for a leaf node, but if the node in question is an internal node, then it must be replaced with -∞, and must be rotated accordingly. This too takes O(logV) time.

Space Complexity

The main focus of this project is the optimization of graph storage and hence space complexity is the metric of highest importance. Adjacency Lists have a space complexity of an O(V+E), where V is the number of vertices and E is the number of edges. In a binary tree, our space complexity is O(V) in worst- and best-case scenarios. Just thinking about our previous example would validate this, where an adjacency matrix stored a total of 12x12=144 numbers, while the corresponding binary tree required only half that value. And finally, a treap has O(V) space complexity in both worst and average case scenarios.

Advantages

Adjacency List representation is very useful when space becomes a factor, and even though it requires more time for certain operations when compared to an Adjacency Matrix, the storage saved is massive.

Treaps have a high probability of ending up with a balanced binary tree, and a balanced binary tree requires less resources to locate/insert/delete a particular node when compared to an unbalanced binary tree. Treaps also have the benefit of being easy to implement and scale it to a large scale network. Since it uses linked lists and pointers, it vastly reduces the memory occupied, compared to the array structure of an Adjacency Matrix. They are also considerably faster than Red-Black trees.

Binary Tree, on the other hand, require even less storage than Adjacency Matrices and Treaps, and the given algorithm can be scaled to match any chemical compound or large scale railway network graph.

Disadvantages

The primary disadvantage of adjacency list representation is that simple operations like removing a vertex or searching for an edge take a lot of time, and this is not ideal when the number of nodes stretch to billions for large scale graphs.

While Treaps do have a high probability of acquiring a balanced binary tree through randomization processes, it cannot be ignored that there is a chance that the final tree is unbalanced, and when all its operations depend on the height of the tree, an unbalanced tree could lead to very expensive processing.

Binary Trees, while being one of the best solutions, also have the disadvantage of having a very complex procedure for deleting a node and having insertion/deletion/searching operations dependent on the height of the tree.

Implementation

Adjacency Lists are more suited towards being implemented on a small-scale, as its disadvantages can proportionate to disastrous levels on a larger scale. Treaps and Binary Trees on the other hand can be used for large graphs and their mappings.

Summary Table

	Adjacency List	Treap	Binary Tree
Time Complexity	O(E) Insertion – O(1) Searching – O(n) Deletion – O(n^2)	O(logV)	O(V)
Space Complexity	O(V+E)	O(V)	O(V)
Advantages	very efficient for sparse graphs with lots of 0s present in their adjacency matrices	high probability of ending up with a balanced tree easy to implement and scale up to large scale projects and networks faster and more space-efficient than red-black trees	requires even less storage than adjacency matrices and treaps can be scaled and implemented to any large scale network
Disadvantages	simple operations like deletion of vertices take a lot of time	there is a chance that the tree obtained is not balanced and this becomes inefficient on a large scale	deletion operations are very complex and all operations depend greatly on the height of the tree
Implementation	small-scale	large-scale	large-scale

Common Applications

The common applications of graph that can be implemented as a result of these three data structure mapping include: Social Graphs [such as Facebook's Graph API], Knowledge Graphs [Google's Knowledge Graph], Recommendation Engines, Google Maps, Flight Network Optimization, and Scientific Computations.

Conclusion

We have thus analyzed and tested three possible data structures that could serve as a replacement for Adjacency Matrices, and represent graphs in a much more efficient and optimized manner. Adjacency Lists, although a good alternative for Adjacency Matrices on a small-scale, are too time consuming to be implemented on a larger scale. Treaps vastly optimized the memory required to store large graphs through keys and priorities, and also had reasonable time complexities. Binary Trees have been found to be the best alternative among the three, requiring even less memory than a treap, as a treap requires more space to store its nodes as compared to a binary tree. It also has all the other benefits of Treaps, being efficient to implement and scalable. Hence, we have arrived at the conclusion that a Binary Tree implementation of Graphs gives us the most optimized form of storage of data.

References

[1] https://opendatastructures.org/ods-java/7_2_Treap_Randomized_Binary.html

[2] http://csjournals.com/IJCSC/PDF7-2/31. Sandeep.pdf

[3]https://www.researchgate.net/publication/258650448_Using_Treaps_for_Optimization_of_Graph_Storage

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms(3rd ed.).* McGraw-Hill Higher Education, (2009).