Definition

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.

Where it is used?

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 at level .
* 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 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 .
* The minimum and maximum heights of *n* node binary tree are ceiling value of 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 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 for Insertion

Input data:

1. Key value of node is ‘key’.
2. Priority of node as ‘pr’.
3. Root node as ‘root’.
4. Adjacent key as ‘adj’ = adjacent[i] where i approach from 0 to number of adjacent nodes to that key.

Set new->key = key;

Set new->pr = pr;

If root=NULL then

Set root = new;

Return;

Set ptr = root;

If (adj<key)

Repeat while ptr != NULL do

Par = search(adj); // search adjacent key location

If(par->left =NULL)

Par->left = new;

Return;

Else

If(par->right = NULL)

Par->right = new;

Return;

Else

If(par->right = NULL)

Par->right = new;

Return;

Else

Adj = adj[i+1];

Goto step 7;

Else

Par = searchRight(); // search any key which do not have right child

Par->right = new;

Return;

Example

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

Chart

Description automatically generated

Figure 4.1: Graph

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.

Shape

Description automatically generated

Figure 4.2: Adjacency Matrix

Structure of proposed node:

Each node store 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.

Diagram

Description automatically generated

Figure 4.3: Node Structure

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

A picture containing text, crossword puzzle

Description automatically generated

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.

Diagram

Description automatically generated

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.

Diagram

Description automatically generated

Figure 4.5

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

Diagram

Description automatically generated

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,

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

If again it is less then,

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

If again it is less,

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

The final tree of figure 4.1 is mapped as shown in figure 4.7.

Diagram, engineering drawing

Description automatically generated