International Journal Of Engineering Research & Management Technology

ISSN: 2348-4039

September- 2014 Volume 1, Issue-5

Email: editor@ijermt.org www.ijermt.org

Dynamic Trees in Data Structure

Alpana Gupta
Computer Science and Information Technology
Dronacharya College of Engineering
Gurgaon, India

Anupam Lata
Computer Science and Information Technology
Dronacharya College of Engineering
Gurgaon, India

ABSTRACT

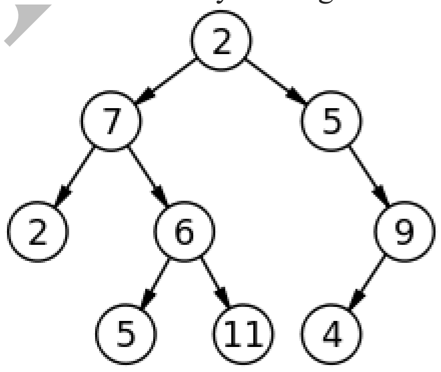
A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a link operation that combines two trees into one by adding an edge, and a cut operation that divides one tree into two by deleting an edge. The tree is one of the most powerful of the advanced data structures and it often pops up in even more advanced subjects such as AI and compiler design. Surprisingly though the tree is important in a much more basic application - namely the keeping of an efficient index. This research paper gives us brief description of importance of tree in data structure, types of trees, implementation with their examples.

Keywords- Treaps, B-tree, R-tree, Variants, query type,

INTRODUCTION

A tree is a widely used abstract data type (ADT) or data structure implementing this ADT that simulates a hierarchical tree structure, with a root value and sub trees of children, represented as a set of linked nodes. A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root. [1]

In this paper we consider the following problem: We are given a collection of vertex-disjoint rooted trees. We want to represent the trees by a data structure that allows us to easily extract certain information about the trees and to easily update the structure to reflect changes in the trees caused by three kinds of operations: link(v, w): If v is a tree root and w is a vertex in another tree, link the trees containing v and w by adding the edge(v, w), making w the parent of v. cuf(v): If node u is not a tree root, divide the tree containing v into two trees by deleting the edge from u to its parent. everf(v): Turn the tree containing vertex u “inside out” by making v the root of the tree.



Tree nodes have many useful properties. The depth of a node is the length of the path (or the number of edges) from the root to that node. The height of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A leaf node has no children -- its only path is up to its parent.

Binary: Each node has zero, one, or two children. This assertion makes many tree operations simple and efficient.

Binary Search: A binary tree where any left child node has a value less than its parent node and any right child node has a value greater than or equal to that of its parent node.

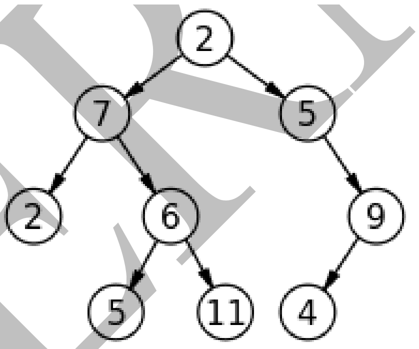
Binary tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the *left* child and the *right* child. A recursive definition using just set theory notions is that a (non-empty) binary tree is a triple (*L*, *S*, *R*), where *L* and *R* are binary trees or the empty set and *S* is a singleton set.[2] Some authors allow the binary tree to be the empty set as well.

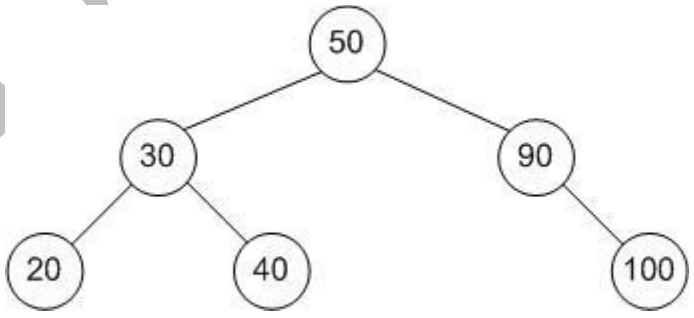
For example, if you construct a binary tree to store numeric values such that each left sub-tree contains larger values and each right sub-tree contains smaller values then it is easy to search the tree for any particular value. The algorithm is simply a tree search equivalent of a binary search:

```
Start at the root
REPEAT until you reach a terminal node
IF value at the node = search value
THEN found
 IF value at node < search value
THEN move to left descendant
ELSE move to right descendant
END REPEAT
```

Of course if the loop terminates because it reaches a terminal node then the search value isn't in the tree, but the fine detail only obscures the basic principles.



Binary Search tree



A binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub tree and smaller than the keys in all nodes in that node's right sub-tree. Each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node. BSTs are also dynamic data structures, and the size of a BST is only limited by the amount of free memory in the operating system. The main advantage of binary search trees is that it remains ordered, which provides quicker search times than many other data structures.
Node- Any item that is stored in the tree. Root-The top item in the tree (50 in the tree above) Child- Node(s) under the current node (20 and 40 are children of 30 in the tree above) Parent- The node directly above the current node (90 is the parent of 100 in the tree above) Leaf -A node which has no children.

Difference between binary tree and binary search tree

Binary tree: In short, a binary tree is a tree where each node has up to two leaves. In a binary tree, a left child node and a right child node contain values which can be either greater, less, or equal to parent node.

```
  3
 / \
4   5
```

Binary Search Tree: In binary search tree, the left child contains nodes with values less than the parent node and where the right child only contains nodes with values greater than the parent node. There must be no duplicate nodes.

```
  4
 / \
3   5
```

Tree Traversal

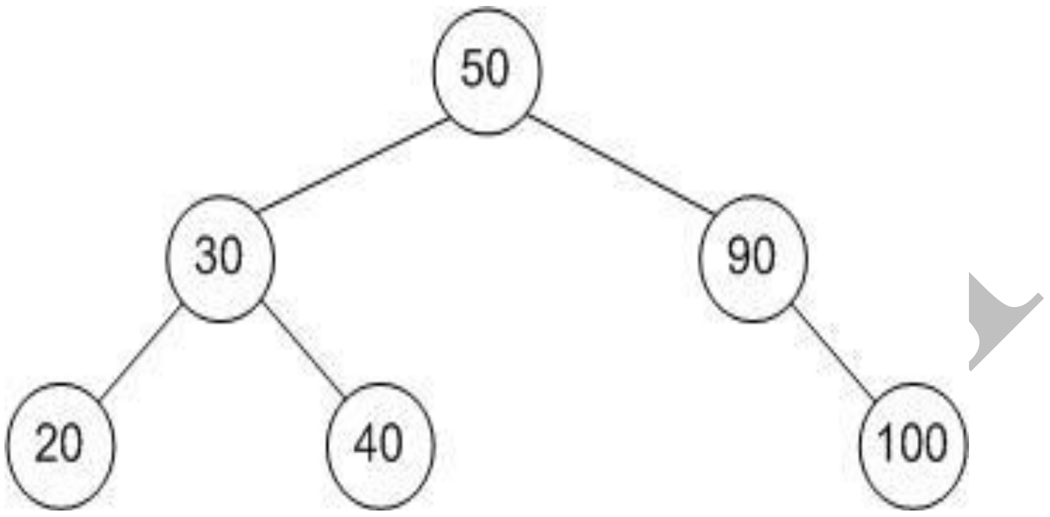
Many problems require we visit the nodes of a tree in a systematic way: tasks such as counting how many nodes exist or finding the maximum element. Three different methods are possible for binary trees: *preorder*, *postorder*, and *in-order*, which all do the same three things: recursively traverse both the left and right subtrees and visit the current node. The difference is when the algorithm visits the current node:
Preorder: Current node, left sub tree, right sub tree (DLR)
Post order: Left sub tree, right sub tree, current node (LRD)
In-order: Left sub tree, current node, right sub tree (LDR)
Level order: Level by level, from left to right, starting from the root node.

Sample implementations for Tree Traversal

```
preorder(node)
  visit(node)
  if node.left ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
inorder(node)
```

```
 if node.left $\neq$ null then inorder (node.left)
  visit(node)
  if node.right $\neq$ null then inorder (node.right)
postorder(node)
  if node.left $\neq$ null then postorder (node.left)
  if node.right $\neq$ null then postorder( node.right)
  visit(node)
```

Example of tree transversal



Red-black trees

A red black tree is a self-balancing tree structure that uses a color attribute , and can be modelled as 2-3-4 tree , which is a sub-class of B tree (below). A black node with one red node can be seen as linked together as a 3-node , and a black node with 2 red child nodes can be seen as a 4-node.

4-nodes are split , producing a two node, and the middle node made red, which turns a parent of the middle node which has no red child from a 2-node to a 3-node, and turns a parent with one red child into a 4-node (but this doesn't occur with always left red nodes).

A in-line arrangement of two red nodes, is rotated into a parent with two red children, a 4-node, which is later split, as described before.

```
     A  right rotate          'split 4-node'   |red
  red / \    -->       B          --->    B
    B              red/ \red              / \
red / \              C   A              C   A
  C   D                /                  /
                     D                  D
```

An optimization mentioned by Sedgwick is that all right inserted red nodes are left rotated to become left red nodes, so that only inline left red nodes ever have to be rotated right before splitting. AA-trees (above) by Arne Anderson , described in a paper in 1993 , seem an earlier exposition of the simplification, however he suggested right-leaning 'red marking' instead of left leaning , as suggested by Sedge wick, but AA trees seem to have precedence over left leaning red black trees. It would be quite a shock if the Linux CFS scheduler was described in the future as 'AA based'.

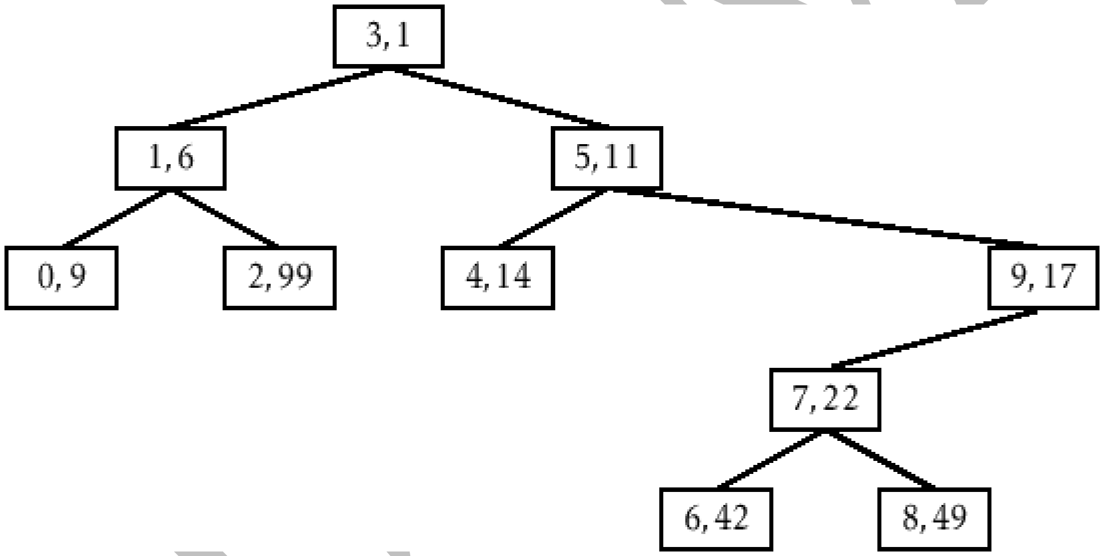
In summary, red-black trees are a way of detecting two insertions into the same side, and leveling out the tree before things get worse. Two left sided insertions will be rotated, and the two right sided insertions, would look like two left sided insertions after left rotation to remove right leaning red nodes. Two balanced insertions for the same parent could result in a 4-node split without rotation, so the question arises as to whether a red black tree could be attacked with serial insertions of one sided triads of a < P < b , and then the next triad's P' < a.

Treaps

The invariant in a binary tree is that left is less than right with respect to insertion keys. e.g. for a key with order, ord(L) < ord(R). This doesn't dictate the relationship of nodes however, and left and right rotation does not affect the above. Therefore another order can be imposed. If the order is randomised, it is likely to counteract any skew ness of a plain binary tree e.g. when inserting an already sorted input in order.

A node in a Treap is like a node in a Binary Search Tree in that it has a data value, x, but it also contains a unique numerical priority, p, that is assigned at random:

```
class Node<T> extends Binary Search Tree. BST Node<Node<T>,T> {
   int p;
}
```



Example of a Treap containing the integers 0,…,9. Each node, u, is illustrated as a box containing u.x,u.p.

CONCLUSION

Many variants of B-tree and R-tree are proposed and some of them are used in the real world for the query and performance optimization. Some index structure have less space complexity, some have less time complexity and support different data types. Most of them support point query and single dimensional data efficiently but for range query and multidimensional data specific structure is required and support specific type of data. B-tree and its variants are support point query and single dimensional data efficiently while R-tree and its variants support multidimensional data and range query efficiently. BR-tree support single dimensional, multi-dimensional and all four type of query. New index structure is proposed by making change in previous structure with use of some other data structure like hash function or use two good property of two different structure. Like BR-tree use hash function and QR+-tree use of Q-tree and R-tree. For optimize space complexity change in existing algorithm is made. Like in Compact B-tree. In future take idea from this and change existing index structure. For new index structure change can be made in algorithm, use two different index structure or use data structure or use of data structure like hash in index construction.

REFERENCES

1. http://en.wikipedia.org/
2. http://www.i-programmer.info/
3. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, On finding lowest common ancestors in trees,
4. S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, Biased 2-3 trees, in “Proc. Twenty-First Annual
5. http://en.wikibooks.org/
6. https://www.google.com