Trees

In graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Tree is a hierarchical structure where every father has N >= children and every child has exactly one father in such a way that there are no cycles in the structure.

Node which is grandfather of all other nodes is called a root (from graph theory view point we thereby create oriented tree).

Node which has no descendants is called a leaf.

To be a tree is a recursive property – every subtree of a tree S is also a tree.

Tree is very popular for its simplicity and applicability. Examples are search trees of heaps.

Common tree type is a **dichotomous** tree, which has 2 pointers at each node: to the left subtree and to the right subtree.

Binary tree = dichotomous tree, for which also applies that all keys in the left subtree are smaller- and all keys in the right subtree is greater than the key.

(Prerequisite: the keys must be unique.)

Main features: easy and fast search using bisection.

If N is the number of nodes then the number of steps ≈ $\log_2 n$

Example

As an example, we will search for the key 28 in binary tree.

If you are looking for a nonexistent key, easy to verify, that does not really exist.

Implementation

- Binary (and also dichotomous, AVL and other) trees can be implemented in many ways. Most common ones are:
- Using an array (a heap)
- dynamic structures linked by references (remote similarity with a simple linked list)

Heap is a way to convert a binary tree into an array (in effect saving a room of references).

The basis is that binary tree has 1 node in the root, 2 nodes at first level, 4 nodes in second level etc, so at level N there are 2^N nodes.

This is why in an array we allocate 2N positions for i-th level (room for nodes which are missing in a tree, remain empty).

Relatively simple algorithm can be used to find a relation between node position in a binary tree and its index in an array=heap

Because positions of all nodes are known, many algorhithms that cannot be achieved using pointers can be realized using a heap.

Heap – calculation of index

Index calculation is very simple.

- Suppose the to root be at index 1.

If we are at node with index K, then (if they exist):

- its parent has index **K div 2**
- left subtree has index **2*K**
- right subtree has index **2*K + 1**

Implementation by references

```
public class Node {
    public static Node root = null;
    public int key;
    public String data;
    public Node left = null;
    public Node right = null;

public Node(int aKey, String aData){
        key   = aKey;
        data  = aData;
        left  = null;
        right = null;
    }
```

```
public Node add(int aKey, String aData) {
        if (aKey == key){
            System.err.println
            ("duplicate key " + aKey);
            return null;
        } else if (aKey< key) {
            if (left == null) {
                left =
                  new Node(aKey, aData);
                return left;
            } else
                return left.add
                  (aKey, aData);
        } else if (aKey >key) {
                if (right == null) {
```

```
                    right =
                    new Node(aKey, aData);
                    return right;
                } else
                return right.add
                  (aKey, aData);
          }
          return null; // for sure
        }

public Node find(int aKey){
          Node marker = root;
          while(marker != null){
              if (marker.key == aKey){
            break; // found
```

```
            } else if (marker.key< aKey){
                marker = marker.left;
            } else {
                marker = marker.right;
            }
        }
        return marker;
    }
```

Disadvantage of binary tree

In case the nodes are inserted in an improper sequence, branches could grow unevenly, even the tree can degenerate into a linear list.

AVL Trees

Eliminate the main weakness of binary tree.

AVL tree is a binary tree in which is filled the additional condition that in each node, the heights of the left subtree and the right subtree differ by no more than 1

abs(High(L) - High(P)) ≤ 1

or **Imbalance ≤ 1**

Using imbalance it is defined what AVL tree is, but it doesn't solve how to create it.

Lemma: every binary tree can be retransformed into AVL tree using finite number of rotations applied **to the propper nodes of the tree**.

Rotating the tree

Suppose the following initial status. The root is marked by an asterisk.

The first step is to change the root to node "26". If it were a subtree rotation, change the reference, that led to the node 32 to point to the node 26.

In the second step, invert the direction of the branch to fill the condition that no branch leads to the root.

Unfortunately, this arises a situation that the node has three branches, which is unacceptable. Therefore, node 28 will be joined under the node 32.

The resulting tree is shown here.

*Note: The tree that came to us after the rotation, is **not** AVL. This is because we did not apply the rotation to the correct node.*

Rotation - notes

Attention, nowhere it is written that every rotation (and over each node) must result into an AVL tree! Improper application can even worsen the situation.

Like the right rotation, left **rotation exists**. It is completely symmetrical (and will be examined!)

Rotation – application principles

The principle according to which is applied rotation, is based on the fact that the node can be added to the AVL in 6 ways

Principle by which rotations are applied derives from the fact that there are basically 6 ways to add node to the AVL tree:

2-3-4 Trees

Motivation

Problems are with too frequent balancing during tree creation.
The only way – to release the rules.

Possibility to tie in multiple children to the node - multiple path nodes.

2-3-4 trees can have 1,2,3 data items in a node and therefore 2,3,4 children.

What 2-3-4 tree means

2,3,4 is number of child nodes:

- node with 1 data item must have 2 children
- node with 2 data item must have 3 children
- node with 3 data item must have 4 children

Node mustn’t have a single son.

List mustn’t be ever empty.

Duplicate values aren’t allowed in a tree.

Result: List can contain 1,2 or 3 data items.

Rules

- Let’s denote: A,B,C ... data items, a,b,c ... children.

Rules are very similar tot the binary search tree:

- All keys of subtree a must be smaller then key A
- All keys of subtree b must be greater then kdy A and smaller then key B
- All keys of subtree c must be greater then key B and smaller then key C
- All keys of subtree d must be greater then key C

Searching

- Very similar to the binary trees:
- we start at the root
- if node doesn’t contain the key we are looking for the suitable descendant will be chosen and the search goes on

- if the key we are searching is not in the list, it is not in a tree at all

Inserting

- **New item is always added into the list**, because only in the list we can be certain that no such value exists.

Two possible scenarios for searching a place to insert:

- there is no filled node in a path
- there is a filled node

Node is not filled

- Scenario where there is no filled node in a path:
- Addition of an item will not affect tree structure

- Insert operation is simple

Example

Example: we need to insert 7 into this tree.

As 7 is less than 11, we follow the left branch.

In this node, while for the key 7 is enough space, but it is not a leaf. Therefore, it can not be inserted here. We proceed right branch

Resulting tree is shown on figure.

Node is filled

- Scenario where there is a filled node anywhere in a path means that the node must be split, with balancing a tree as a consequence.

How to split a node:

- create a new node and move the greatest item into it
- medium item move into a parent node (=push up)
- adjust linkage

Example

Insert key 24 into this tree.

When we try to insert the key 24, so right in the root of the tree we find completely cluttered node. Therefore, this node must be divided.

Detail division shown here. 11-40-60 node is divided into three nodes 11, 40 and 60. Nodes 11 and 60 remain at the original level, whereas 40 is pushed upward.

After splitting our tree has one floor more.

Notes

Why didn't we here insert 24 next to 11?

New element is always inserted into the list, because only in a list we can be certain that the value doesn't exist.

Resulting tree is here.

- Why to split on a way down?

It makes the job easier:

- on a higher floor the filled node is not likely, because if there was one, it is split by now
- so the node split (moving the item up) won't cause a problem

2-3 Trees

Overview

2-3 tree is a tree typle whose every inner node contains either two children and one key, or it contains three children and two keys. All lists belong to the same depth.

2-3 trees can be considered to be B-trees containing inner nodes with two or three descendants exclusively.

Due to equal depth of lists belongs the 2-3 tree to the balanced trees.

Dept of 2-3 tree with **n** elements is in the interval of **log3n** and **log2n**, depending on the stucture used.

This corresponds with operation heftiness, like searching, inserting and removing data from the 2-3 tree.

2-3 trees are very similar to the 2-3-4 trees. Differences:

- nodes can contain 1 or 2 items
- nodes point to 2 or 3 descendants

Search operation

- same as with 2-3-4 tree
- only the number of items searched will change

Insert operation

- **is totally different. . .**

Inserting into 2-3 trees

- What changed?

During split three items are needed:

- one which will stay
- one which will move into a sibling
- one which will move into a parent

But 2-3 tree has only two items in the node. Where to get the third one?

- item being inserted must be used
- **so it is not possible to split on a way down**

Inserting into an non-full leaf

Inserting into a leaf that is not full is without any problem:

Inserting into a filled node with unfilled parent

We must split the leaf, move central item into a parent.

Inserting into a filled leaf with filled parent

parent splitting must take place

eventually parent's parent must be split . . . etc.

Recursively on and on, until unfilled node is found.

Can lead to root shifting “one floor up”.

Example

Splitting of a parent must take place.

Eventually parent’s parent must be split ... etc.

But the parent is filled, so we must split it as well.

As 34 is in the middle once again, so we push it...

...and it becomes a new root.

B-Trees

B-trees of K order

Beware- „B“ doesn’t mean binary, but „Bayer’s“.

Structures we examined so far assumed equally fast access to all elements, which isn’t the case in the real world (memory:disk up to 106) and unlimited memory size (which isn’t the case either).

B-trees give a good guideline how to optimize memory paging (=moving between memory and disk) in light of speed.

every page contains at most 2K and at least (except for root) K nodes.

every page is either leaf (=no branches lead from it) or M+1 branches lead from it, where M=current node count on a page.

Tree grows “upward”, meaning at the root.

Every page contains at most 2K and at least (except for root) K nodes.

Every page is either leaf (=no branches lead from it) or M+1 branches lead from it, where M=current node count on a page.

Tree grows “upward”, meaning at the root.

B-trees paging

Paging model (if only 10 nodes fitted into a memory).

We can assume that data access “near by” will be required.

We will held top page in memory + from each level the page that was accessed last.

Red-Black Trees

Red-black trees are similar to AVL trees and working with them is also similar.

During insert into AVL tree we must make @ rotations at the most, but during removing nodes from AVL tree we can run into situation when we have to rotate nearly everything.

This disadvantage is not inherent to red-black trees, because **we never do more than 2 rotations**.

Red-black tree is a binary searching tree. It is a data structure that is often used for implementation of associative array.

Rules

Nodes of red-black trees are marked by color.

Each node is either red or black.

Each leaf (leaves we consider null pointers) is black.

On each path from root to the leaf there is equal count of black nodes.

Results

There are never two red nodes atop of each other.

Depth of tree is logarithmic $2.\log_2(N+1)$.

Rotation

After Insertu and Delete we repair colors by recoloring on a way to the root and by rotation.

But it is necessary to discuss considerably more cases than in case of AVL trees.

Detailed description is here.

Hufmann tree

Statistical method.

Look up the frequency of characters in the string, and these characters are encoded by frequency so that the shortest code corresponded to the most frequent character.

The codes are binary, the most frequent character is therefore represented by 1 bit unlike eight (16) bits required to store an ASCII (UTF) character.

This is a prefix code, so character is not a prefix of another character → codes for the individual characters are of different

length and there is no need to mark the end of them (each character kreprezentuje one leaf of the tree).

- Codes are created from Huffmann tree:
- Individual characters denote like leaves of the tree.
- These leafy put into the list named S (S contains nodes of the tree, therefore in the beginning only leaves).
- Sort the list S by frequency.
- Select from two elements S M, N with the lowest frequencies m, n, m <n.
- Create a node p, where in the left is M and in the right is N, and its frequency will be m + n.
- Put p to S and repeat until the S is not just one node.

Here is an example.

Strom, of course, we transmit along with compressed data.

The tree can represent the string so that it can go through the DFS algorithm and store for each vertex 0 and 1 for each sheet and sign this sheet.

When retrieving, read zeros and create nodes (to the left). If you encounter one, write the letter and return to the nearest right node (right node of parent, or most left subtree element - for parent at right).

When decompressing the tree we go through and reconstructs the message.

Content