Universitetet i Oslo
Institutt for Informatikk
S.M. Storleer, S. Kittilsen

IN2010: Algoritmer og Datastrukturer

Series 5

Tema: DFS, BFS, topologisk sortering, (balanserte) binære (søke-)trær

Publisert: 14.10.2019

Classroom

Oppgave 1 (Fra boka)

- R-4.7
- R-4.10
- R-4.15 (AVL-trær: for alle noder i treet kan differansen mellom høyden til barna være høyst 1.)
- C-4.14
- R-13.6 c)

  Solution

  A BFS traversal starting at vertex 1 visits the vertices in the following order:

  1, 2, 3, 4, 6, 5, 7, 8.
- R-13.10

  Solution

  Remember that the topological ordering might not be unique.

  (BOS, JFK, MIA, ORD, DFW, SFO, LAX).
- C-13.9

Oppgave 2 (Height of a binary tree)

1. Show that the maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$
2. Show that for a balanced tree with N nodes, the height is $\lfloor log_2(N) \rfloor$

Solution

1. This can be proved by induction.

   Base step $h = 0$: $2^{0+1} - 1 = 2 - 1 = 2$

   Induction step: Assume that a tree with height k has maximum $2^{k+1} - 1$ nodes. Prove that a tree with height $k + 1$ has maximum $2^{(k+1)+1} - 1 = 2^{k+2} - 1$ nodes.

   Proof: For a tree with height $k + 1$ we have:

   - 1 root node
   - a left subtree with maximum height k, having maximum $2^{k+1} - 1$ nodes according to the induction hypothesis
   - a right subtree with maximum height k, having maximum $2^{k+1} - 1$ nodes according to the induction hypothesis

   This gives a total of $(2^{k+1} - 1) + (2^{k+1} - 1) + 1 = (2^{k+1} + 2^{k+1}) - 1 = 2 * 2^{k+1} - 1 = 2^{k+2} - 1$ nodes.

2. Proved by manipulation of the result from the previous exercise: $N = 2^{h+1} - 1 \implies 2^{h+1} = N + 1 \implies \log_2(2^{h+1}) = \log_2(N + 1) \implies h + 1 = \log_2(N + 1) \implies h = \log_2(N + 1) - 1 \implies h = \lfloor \log_2(N) \rfloor$

Oppgave 3 (Trees) Given a binary tree (not necessesarily a binary search tree), with an arbitary number of nodes. Write pseudo-code for a function which

(a) returns the smallest value in the tree

(b) returns the largest value in the tree

(c) returns the length of the longest path from the root to a null pointer

(d) returns the length of the shortest path from the root to a null pointer

Oppgave 4 (Nodes in a binary tree)

1. Show that in a (non-empty) binary tree with n nodes, there are $n + 1$ `null` links representing children.

2. A *full* node is a node with 2 children. Prove that the number of full nodes + 1 is equal to the number of leaves in a non-empty binary tree.

Solution

The exercise is taken from from Weiss p.160, exercise 4.4 and 4.6.

4.4 For an N-node tree, there are $2N$ pointers and $N - 1$ incoming edges. Therefore, we have $2N - (N - 1) = N + 1$ `NULL` pointers.

4.6 We prove it by induction on the number of full nodes.

Base case: Single-node tree → 1 leaf, 0 full node. Tree with one full node → 2 leaves, 1 full node.

Inductive case: Assume that all binary trees with n full nodes have n + 1 leaves.

We need to show that a tree with n+1 full nodes has n + 2 leaves: A tree with n+1 full nodes can be formed from a tree with n full nodes in the following ways. First

we observe: given the tree in the induction case, we have to add 1 full node. That can be done by *turning* an existing node in the given tree from a non-full node into a full node. Alternatively: the tree is extended by a (new) subtree which contains the additional full node. The following is the (simplified) core of the argument:

1. Make a non-full inner node, i.e., a node with 1 child, a full node by adding one leaf node.
2. Add 2 children to a leaf node.
3. Add a tree with one full node to a leaf.

From these cases, the number of full nodes is increased by one and the number of leaves is also increased by one. This induction is slightly simplified (but actually captures intuitively the argument), because we are doing induction on the number full node. Basically what is imprecise is that the cases do not cover all possibilities in that a tree with $n + 1$ full nodes can be turned into one with $n + 2$ such nodes (even if restricting to adding nodes at the leaves, which is ok.) If we want to capture *all* such $n + 2$ trees we must be more careful, but the idea remains the same.

Let's observe: a tree with *no* full nodes is linear (= list-like). A tree with 1 full nodes has *two leaves*. This either is obvious or of course is also follows from the induction.

additonal full node: in old tree

1. Make a non-full inner node, i.e., a node with 1 child, a full node by adding one linear tree (plus append to an arbitrary number of leaf-nodes *one* linear tree).
2. Add 2 linear trees/lists as 2 children to a leaf node (and additionally append to an arbitrary number of leaf-nodes *one* linear tree).

additional full node: in the appended, new tree Add not a leaf a tree with *1* full node (and additionally append to an arbitrary number of leaf-nodes *one* linear tree).

From these cases, the number of full nodes is increased by one and the number of leaves is also increased by one.

As an aside: How would one prove that a tree with 1 full node has 2 leaves? One could say: that's obvious (which it more or less is). If one likes more formal arguments: by induction on the number of nodes does the job. The base case is $n = 3$ which is the smallest tree with one full node. The induction should not restrict itself to add new nodes at the leaves (otherwise one would stick to the form of the tree where the full node is at the root).

Oppgave 5 (Red-black tree) Build, step by step, red-black trees that result from inserting the following sequences of elements:

1. `41 38 31 12 19 8`
2. `A L G O R I T H M`

Solution

Seen in the form of complexity measures ("Big-O") there's no big difference between AVL and red-black trees (and other form of balanced trees). The trick is always the same: balancing gives a garantee of $\log n$ worst case. The problem of the AVL trees is mainly

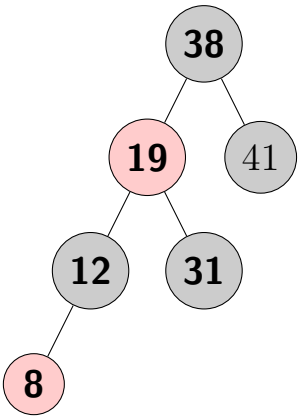

Figure 1: Exercise 5

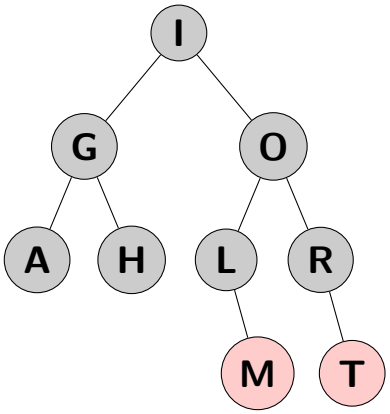

Figure 2: Exercise 5

that they are too strict as far as balancing is concerned. They like to maintain almost perfect balance. It's only "+/- 1". Especially in very large trees and when doing frequent insertion (and/or deletions), the requires frequent rebalance. Red-black trees allow not a constant "+/-1" condition for being balanced but much more loose "factor two". Beside that, they use a clever "coloring scheme" to ensure that condition. It's clever because one does not need to calculate the balance *globally* for all node and compare whether the tree is needs rebalance or not. It suffices to check whether locally the color conditions in the neighborhood of the interted node is satisfied or not. Also the rebalance is done in constant time (which is less easy to see than for the AVLs).

The two resulting red-black trees are given as follows.

Oppgave 6 (De første nodene i et rød-svart tre) Algoritmen for å sette inn en node (X) i et rød-svart tre forutsetter eksistensen av tre noder; mor (P), bestemor (G) og tante (S). I denne oppgaven skal vi se på hvordan treet bygger seg opp før vi er kommet så langt at alle disse eksisterer. Først setter vi inn rotnoden. For å løse denne oppgaven kan det være lurt å tegne (et tre!).

a) Hviken farge skal rota ha?

b) Vi setter inn en node til. Hvor mange forskjellige trær kan vi ha med to noder?

c) Vi setter inn en node til. Hvor mange forskjellige trær kan vi ha med tre noder?

d) Vi setter inn en node til. Hvor mange forskjellige trær kan vi ha med fire noder?

e) Hvor mange noder må treet minst inneholde for at uansett hvor ny node X havner (på grunnlag av invariantene for et binært søketre), så må P eksistere?

f) Hvor mange noder må treet minst inneholde for at G må eksistere?

g) Hvor mange noder må treet minst inneholde for at S må eksistere?

Solution

a) svart

b) 2

c) 1

d) 4

e) Hvis treet ikke er tomt, eksisterer mor (P) til den nye noden.

f) 2 (fra før)

g) Det er aldri sikkert at tante (S for søstra til mor) eksisterer.

Lab

Oppgave 7 (Binary search) Implement binary search in an array of integers.

Oppgave 8 (Rotations) Implement single and double rotation for arbitrary binary search trees (i.e. without the color coding of red-black trees).

Oppgave 9 (Innsetting i rød-svart tre) Implementer innsettingsalgoritmen i Java. Det kan være lurt å gjøre oppgaven *De første nodene...* først.

Oppgave 10 (Red-black trees vs. ordinary BST) In the previous week, you were asked to analyze the play *Vildanden* (se datafil vildanden_utf8.txt sammen med ukesoppgavene) from Henrik Ibsen by building a binary search tree. Instead of building a BST, use a *red-black* tree for the play Vildanden this week. Similarly, all the words which are different from upper case and lower case are considered to be the same. Each node in the tree is corresponding to a unique word in the file. Each node should remember the frequency of the corresponding word appear in the play. Calculate the depth of the left and right subtrees for the red-black tree, and compare with the depth of the BST from last week.