Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe and Sarah Winkler

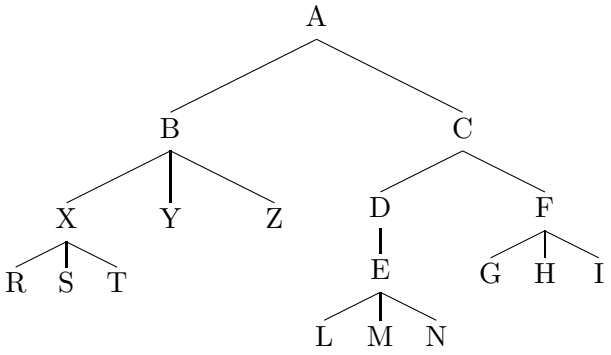
20 May 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 8

Exercise 1 (Trees)

In the lecture the following terms are defined in the context of trees: node, edge, root, degree, level, height, ancestor, descendant, parent, child, path. Consider the tree given in the figure below and answer the following questions.



a) Which node is the root node? Node A is the root.

b) Give the degree of the following Nodes: D, E, C. What is the degree of the tree itself?
The degree of Node D is 1 of E is 3 and C is 2.
The tree has degree 3, because the maximum degree of a single Node is three, in this case of B, X, E and F.

c) What is the level of the following nodes: A, B, Z, D, G, L? What is the height of the tree?
The levels for Nodes A, B, Z, D, G, L are 0, 1, 2, 2, 3, 4.

The height of the tree is equal to the maximum level of a single node of the tree. In this case because of nodes L, M, N it is 4.

d) Categorize the relation between the following pairs of nodes with, ancestor, descendant, parent, child: (Y,A),(Y,B)

- A is an Ancestor of Y. Y is a descendant of A.
- B is the parent node of Y. Y is a child node of B. B is also an ancestor of Y.

e) Give all child nodes, the parent node, all ancestors and all descendants of D.

Node D has

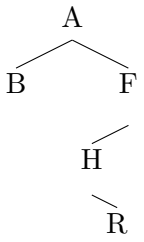
- Node E as child
- Nodes E, L, M, N as descendants
- Node C as parent
- Node C, A as ancestors

f) List all paths that contain Node D.

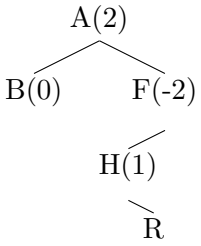
- (A,C)(C,D)(D,E)(E,L)
- (A,C)(C,D)(D,E)(E,M)
- (A,C)(C,D)(D,E)(E,N)
- (C,D)(D,E)(E,L)
- (C,D)(D,E)(E,M)
- (C,D)(D,E)(E,N)
- (D,E)(E,L)
- (D,E)(E,M)
- (D,E)(E,N)
- (A,C)(C,D)(D,E)
- (A,C)(C,D)
- (C,D)(D,E)
- (C,D)
- (D,E)

Exercise 2 (AVL-Trees)

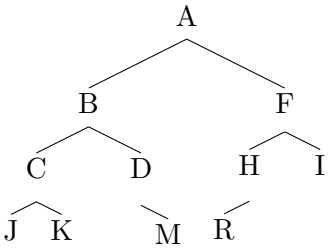
For the following trees decide whether they are balanced or not and specify balance factors for all nodes. If a tree is not balanced identify all roots of smallest unbalanced subtrees and balance them using rotations.

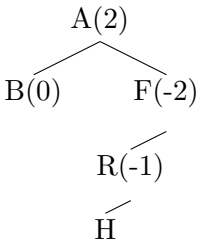


The tree is not balanced. The balance factors are given in brackets at each node. There is one critical node F.



To solve the critical node F we need to perform a double rotation. First do a single Roation to the left.

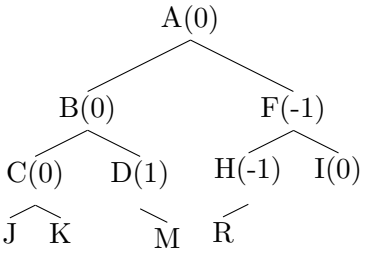


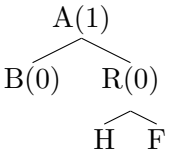


In the second step a single rotation to the right is needed.

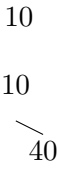
The tree is balanced. The balance factors are given in brackets at each node.

a)



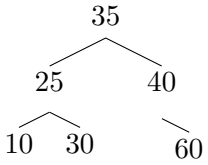


b) Insert the following nodes into an empty AVL tree. Show each step and what rotations are needed. Nodes to insert: 10, 40, 35, 25, 60, 30, 80, 50, 27, 28, 38

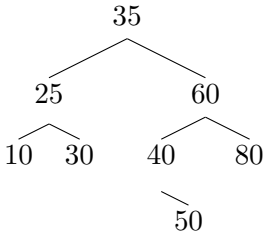


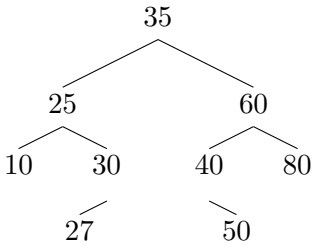
In the next step a double rotation first to the right at 40 and next to the left at 10 is needed.

The insertion of 25 and 60 is straight-forward. In the next step, for the insertion of 30, a single rotation to the left is necessary.

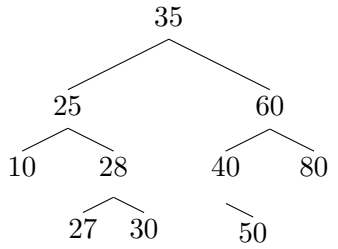


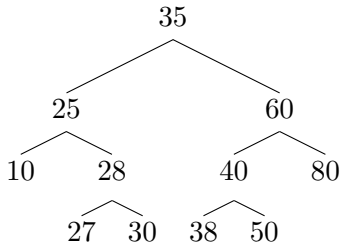
For insertion of 80 a single rotation to the left at 40 is done.





For insertion of 28 a double rotation is done. First part is rotation to the right and the second part a rotation to the left.





c) Consider the record describing binary trees from the last exercise sheet. Modify the record so that it can be used to represent AVL trees.

Listing 1 Record describing an avl tree.

```
1: record avltree =
2: begin
3:     key : integer;
4:     data : ...;
5:     left, right : ^avltree;
6:     balance : integer;
7: end
```

d) Assume that a new node was inserted into the AVL tree and that the balance factors have not yet been updated. Further assume that all keys are distinct. Use pseudo code to describe an algorithm that detects if the tree is in imbalance for the inserted node, and corrects the tree with rotations if necessary. Also update the balance factors.
The head of the algorithm should be as follows:

Listing 2 Rebalance

Input: T: pointer to the root of a tree which is an AVL-tree where a node with key *key* has just been inserted.
key: key of the inserted node.
Output: T: pointer to the restructured AVL tree.

Exercise 3 (Binary Search)

Extend the binary search example from the last exercise sheet such that it is able to work as a telephone book. Create a user interface that allows the user to insert, search and delete telephone numbers. As a simplification assume that each person is identified by the lastname. Also assume that telephone number and lastname are not longer than 50 characters.

Hint: For string comparison use the function **strcmp**. For string assignment use **strcpy**.

Listing 3 Rebalance

Input: T: pointer to the root of a tree which is an AVL-tree where a node with key key has just been inserted.
key: key of the inserted node.
Output: T: pointer to the restructured AVL tree.

```
1:  begin
2:    N := T;                          /* N represents the currently traversed node */
3:    P := nil;    /* P represents the parent node of the currently traversed node */
4:    C := T; /* C is the critical node, it is the node at the path from the root to the
      inserted node with the highest level that is unequal to 0 */
5:    CC := nil;          /* CC the child of the critical node. It is important to decide
      whether a double or single rotation is needed */
6:    while N ≠ nil and N^.key ≠ key do
7:      if N^.balance ≠ 0 then
8:        C := N;
9:      P := N;
10:     if key < N^.key then
11:       N := N^.left;
12:     else
13:       N := N^.right;
14:     if P = C then
15:       CC := N
16:   call updateBalance(C)); /* updates the balance of the tree after the insertion
      of key */
17:   if C ≠ nil then
18:     if CC^.key < key ∧ C^.key > key then
19:       C^.balance := 0;
20:       CC^.balance := −1;
21:       TN := rotateLeft(CC)
22:       TN^.balance := 0;
23:       C^.left = TN;
24:       C = rotateRight(C);
25:     else if CC^.key > key ∧ C^.key < key then
26:       C^.balance := 0;
27:       CC^.balance := 1;
28:       TN := rotateRight(CC)
29:       TN^.balance := 0;
30:       C^.right = TN;
31:       C = rotateLeft(C);
32:     else
33:       C := singleRotation(C);
34: end
```

Listing 4 Rebalance

```
1:  procedure updateBalance(C : ^avltree, key : integer)
2:  begin
3:    while C ≠ nil do
4:      if key < C^.key then
5:        C^.balance := C^.balance − 1;   /* because the left branch of C is one level
          higher after insertion */
6:        C := C^.left;
7:      else if key > C^.key then
8:        C^.balance := C^.balance + 1; /* because the right branch of C is one level
          higher after insertion */
9:        C := C^.right;
10: end

11: function rotateLeft(T : ^avltree) : ^avltree
12: begin
13:   R := T^.right;                                   /* rotate to the left */
14:   TEMP := R^.left;
15:   R^.left := T;
16:   T^.right := TEMP;
17:   return R;
18: end

19: function rotateRight(T : ^avltree) : ^avltree
20: begin
21:   L := T^.left;                                    /* rotate to the right */
22:   TEMP := L^.right;
23:   L^.right := T;
24:   T^.left := TEMP;
25:   return L;
26: end

27: function singleRotation(T : ^avltree) : ^avltree
28: begin
29:   if T^.balance > 1 then
30:     T^.balance := 0;
31:     if T^.left ≠ nil then
32:       T^.left^.balance := 0                  /* null check to avoid nullpointer */
33:     T := rotateLeft(T);
34:   else if T^.balance < −1 then
35:     T^.balance := 0;
36:     if T^.right ≠ nil then
37:       T^.right^.balance := 0                 /* null check to avoid nullpointer */
38:     T := rotateRight(T);
39:   return T
40: end
```