Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dmytro Chibisov
Sandeep Sadanandan

Winter Semester 2007/08
Solution Sheet 8
December 17, 2007

Fundamental Algorithms

ab/B - Trees

Problem 1

Explain *ab*-trees and B-trees. What are the differences?

Solution

B-trees and *ab*-trees are advanced binary search trees. Instead of having only one key in every node and a maximum of two children per node in BST, these advanced trees can have a variable number of keys/children for every node.

ab-trees [class definition]	B-trees (parameter `t`)
Number of children is limited by `a` and `b`, Where the minimum number of children is `a` and maximum is `b`.	The number of children is limited by `t` and `2t`.
The minimum value of `a` is 2 and $b \leq 2a-1$.	$t \geq 2$.
All keys in leaf nodes	Keys in every node
$k_{i-1} <$ keys in i^{th} subtree $\leq k_i$	$k_{i-1} <$ keys in i^{th} subtree $< k_i$

The restrictions on minimum number of children for root node is not very strict. There has to be atleast 2 children for the root node, but it could be less than `a`. At the same time, the upper limit of number of children is applicable to root node too.

The number of data-keys stored in every node is one less than the number of children. i.e, if a node has k keys, then the number of children is $k + 1$ – Or zero, if the node is a leaf.

Problem 2

For an *ab*-tree of height h (root node is at level zero) and n leaves, prove:

1. $2a^{h-1} \leq n \leq b^h$
2. $\lg_b(n) \leq h \leq \lg_a(\frac{n}{2}) + 1$

Solution

1. $2a^{h-1} \leq n \leq b^h$

   An ab-tree will have minimum number of nodes for a given height, when

   - the root node has only two children and
   - all the other nodes have the minimum number of children (i.e, `a` children)

   So, at height = 1 the number of nodes = 2, and at height = 2, the number of nodes is = $2a$. Likewise, at height = h, the number of nodes is $2a^{h-1}$. Since this is the minimum possible, the first part of the inequality is satisfied.

   The tree will have maximum number of nodes at a given height, when all the nodes above that level has the maximum number of childres (i.e, `b` children). It is clear that the value is b^h for height h. Since this is the maximum possible value, the second part of the inequality also is satisfied.

2. $\lg_b(n) \leq h \leq \lg_a(\frac{n}{2}) + 1$

   The inequalities can be derived from the first part of the problem.

Problem 3

Starting from scratch, build a 2-3-tree with the following keys. [60, 76, 57, 48, 85, 55, 19, 56, 52]. Once the tree is created, delete the following elements from the tree. [48]

Solution

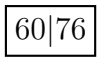
For reference: wikipedia has a good (not extensive) explanation on B-Trees. 2-3 trees could be read from
`http://www.cs.ucr.edu/cs14/cs14_06win/slides/2-3_trees_covered.pdf`

CLR[1] does the complete treatment of BTrees - also with quite a number of good exercises.

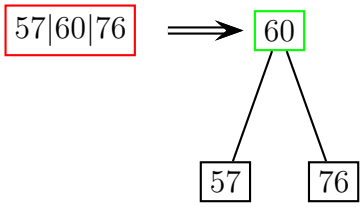
[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Inserting 60

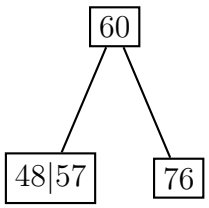
Inserting 76



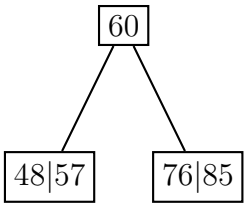
Inserting 57



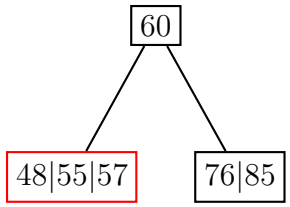
Inserting 48



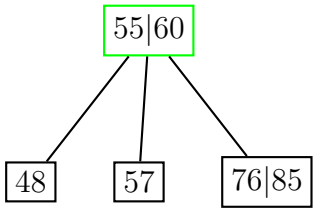
Inserting 85



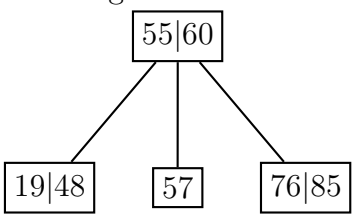
Inserting 55



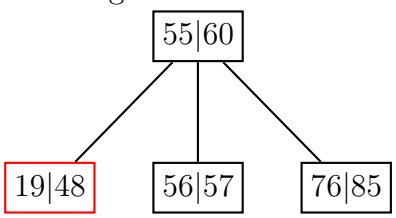
Split at 55



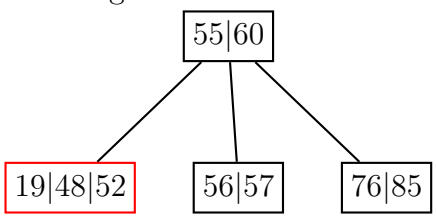
Inserting 19



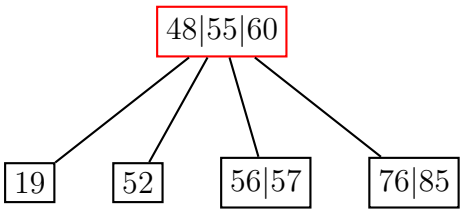
Inserting 56



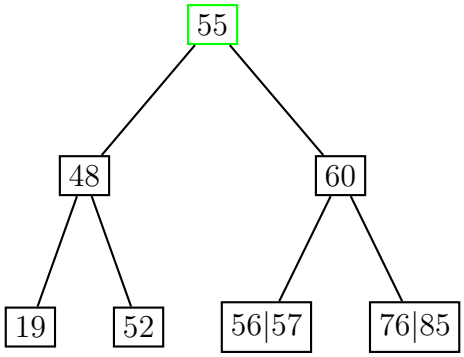
Inserting 52



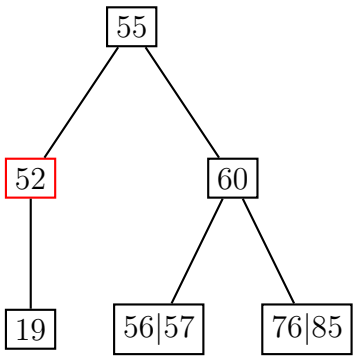
48 up



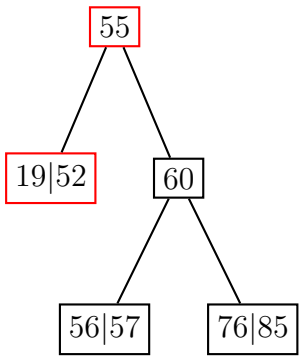
55 up



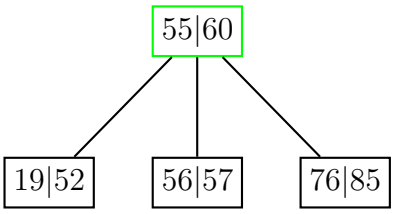
Deleting 48



Join 19 and 52



Merge 55 and 60



(Balanced) Binary (Search) Trees

Problem 4

Explain the three different traversals which could be done on a binary tree. Give the algorithem (pseudo) for all of them.

1. To print the elements of the BINARY SEARCH tree in a non-decreasing order, which traversal must be used?
2. In destroying a tree which traversal should be used? Why?

Solution

A binary search tree can be traversed in three different ways.

- Pre-Order Traversal - The nodes are visited in the order **parent - left child - right child**

  Algorithm *PreOrder*(TreeNode node)
  (∗ The algorithm for pre-order traversal on a binary search tree ∗)
  1. $PreOrder(TreeNode.LeftChild)$
  2. $PreOrder(TreeNode.RightChild)$

- In-Order Traversal - The nodes are visited in the order **left child - parent - right child**

  Algorithm *InOrder*(TreeNode node)
  (∗ The algorithm for in-order traversal on a binary search tree ∗)
  1. $InOrder(TreeNode.LeftChild)$
  2. $Print(TreeNode)$
  3. $InOrder(TreeNode.RightChild)$

- Post-Order Traversal - The nodes are visited **left child - right child - parent**

  Algorithm *PostOrder*(TreeNode node)
  (∗ The algorithm for post-order traversal on a binary search tree ∗)
  1. $PostOrder(TreeNode.LeftChild)$
  2. $PostOrder(TreeNode.RightChild)$
  3. $Print(TreeNode)$

1. To print the elements of the Binary Search tree in a non-decreasing order, which traversal must be used?

   In-Order traversal. In indorder traversal, the left side is traversed first before you reach the parent. And the right side is traversed only after the parent is traversed. This makes sure that any node is traversed only after all the elements which are smaller than that are traversed. This makes sure that the order of nodes traversed/printed will be non decreasing.

2. In destroying a tree which traversal should be used? Why?

   Post-Order traversal. In both the other traversals, the parent node will be destroyed even before the child nodes are destroyed. Hence the link to the child nodes will be lost.

   In postorder, when the traversal reaches the parent node for destroying, the child nodes are already destroyed and hence the node is just a leaf. Hence nothing is lost by destroying it.

Problem 5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child. (Successor and Predecessor - both in-order)

Solution

For any node, the successor is the left most element of the right subtree (if it exists). It is given in the problem that right child exists. So the successor is the left most child in the right subtree. Evidently, left most child cannot have a left child. (if node x has one left child, then node x is no more the left most)

In the similar way, predecessor is the right most child in the left subtree. And rightmost child cannot have a right child.

Problem 6

Suppose n pair-wise unique keys are stored in a sorted array (ascending order). Let $Rang(k)$ be the index of key k in this sorted order. Consider the following implementation of `is_element(k)`:

$i = 0$;
while $2^i \leq n$ and $A[2^i] < k$ **do**
  $i++$;
end while
if $2^i \leq n$ **then**
  $binary\_search(A, 2^{i-1}, 2^i, k)$;
else
  $binary\_search(A, 2^{i-1}, n, k)$;
end if

Why does this approach work, and what is its complexity?

Solution

Let's assume that the algorithm works.

- Proof that it really works

  The while loop breaks on three different cases. Let's consider them separately.

  1. $2^i > n$ (First condition violated)
     $2^{(i-1)} \leq n$ and $A[2^{(i-1)}] < k$ was the last true statement for which while loop was executed. That means that k happens to be on the right side of 2^{i-1}. And the coming out of the loop means that the size of the array, n, is less than 2^i. Hence the search has to be happened on the range greater than 2^{i-1} upto n.
  2. $A[2^i] \geq k$ (Second condition violated)
     When the while loop was executed the last time, $A[2^{i-1}] < k$ was true. So from the last-loop executed and the first-loop violated conditions we can see that $A[2^i] \geq k$ and $A[2^{i-1}] < k$.
     i.e., $A[2^{i-1}] < k \leq A[2^i]$. Hence we search in that range.
  3. $2^i \leq n$ and $A[2^i] \geq k$ (Both conditions violated)
     This case is a mix of the cases discussed above. From the first case we know that $n < 2^i$. From the second case we know k lies in the range 2^{i-1} and 2^i.
     Hence we search in the range 2^{i-1} to n which is the same as the first case.

  So, from the three cases and after looking at the ranges at which we do the binary search, we can see that the algorithm works fine.

- Complexity of the algorithm

  The maximum number of times the while loop will execute is $\lg n$. (Starting from 2^0 until $2^i > n$, there can be a maximum of $\lg n$ iterations)

The maximum size of the range in which we do the binary search can be $\frac{n}{2}$ (The worst case is when we search from $A[2^{i-1}]$ to $A[2^i]$ with $n = 2^i$. Then the number of elements in the range is $2^{i-1} = \frac{n}{2}$. So the complexity of binary search on $\frac{n}{2}$ sized array is $O(\lg n)$)

So considering both the *while* statement and the *binary search*, we can see the complexity is $O(\lg n)$.

Problem 7

Suppose we implement an AVL tree with an additional pointer to the leftmost node (containing the smallest key). This allows us to execute a `find_min()` operation in constant time. Show how the implementations of the AVL tree operations can be modified to keep this pointer current, without increasing their asymptotic complexity by more than a constant factor.

Solution

Since we are considering the leftmost node of an AVL tree, first let's check what happens to the node when a rotation happens. There are four different types of rotations in an AVL tree. We can see that none of the rotations change the left-most position of the minimum element. So rotations have nothing to do with the pointer problem.

Now since we need a method which can take us from any node to the left most node, let us try to have a pointer in every node, which points to the left most node.

In this case, when another element is added on the left side of the leftmost node, we have to visit every node and update the pointer. And this method is too expensive to do.

So we make a place-holder for the pointer to the leftmost node and make every other node point to the place-holder. In this situation, whenever something happens to the leftmost node (deleting it or adding another element to the left), we have to update only in the place-holder. The `find_min()` can always go to the place-holder node which always contains the updated pointer to the leftmost node. Hence the `find_min()` takes only two operations - going to the place-holder and going to the leftmost node from the place-holder.

Problem 8

Given a Binary Tree, how do you decide it's a binary search tree? Give explanation.

Solution

Do a inorder traversal of the tree and print the nodes. If the nodes are in non-decreasing order, the tree is binary search tree.

Proof:

If the nodes are NOT in non-decreasing order, it is evident that the tree is not a binary search tree.

If the tree is not a binary search tree, then it violates the binary search tree property atleast at one node. So the inorder traversal of the tree at that node cannot give a non-decreasing sequence.

so the non-decreasing sequence is possible iff the tree is binary search tree.

Problem 9

Describe an algorithm to select the k^{th} key in a binary search tree

Given a tree with n nodes, $k = 0$ selects the smallest key, $k = n - 1$ selects the largest key, and $k = \lfloor \frac{n}{2} \rfloor - 1$ selects the median key.

Solution

The naive solution will be to do an inorder search and find out the k^{th} element in the sequence.

A more efficient method will be to decide the traversal depending on the value of k. If k is smaller than or equal to $k = \lfloor \frac{n}{2} \rfloor - 1$, then an inorder search could be done and take the k^{th} element. Else if k is larger than $k = \lfloor \frac{n}{2} \rfloor - 1$, then a reverse-inorder search could be done and take the $(n - k)^{th}$ element. Where as in reverse-inorder search, the order of visiting will be **right child - parent - left child**