Analysis of Algorithms 1
(Fall 2011)
Istanbul Technical University
Computer Eng. Dept.

Chapter 14

Augmenting Data Structures

Last updated: December 16, 2009

Purpose

Understanding what “augmenting a data structure” means (augmenting=extending)

Go through examples cases where a known data structure is modified to solve a new problem.

Outline

Augmenting a data structure

Red and Black tree for order statistics (SELECT and RANK)

Interval trees

Augmenting a Data Structure

It is unusual to have to design an all-new data structure from scratch.

It is more common to take a data structure that you know and store additional information in it.

With the new information, the data structure can support new operations.

But. . . you have to figure out how to ***correctly maintain the new information without loss of efficiency.***

Augment Red-Black Trees

So that we have

- The usual dynamic-set operations
  - **INSERT(S, x):** inserts element x into set S.
  - **MAXIMUM(S):** returns element of S with largest key.
  - **EXTRACT-MAX(S):** removes and returns element of S with largest key.
  - **INCREASE-KEY(S, x, k):** increases value of element x.s key to k. Assume k ≥ x.s current key value.
- PLUS the following order statistics related operations:
  - **OS-SELECT(x, i):** return pointer to node containing the i th smallest key of the subtree rooted at x.
  - **OS-RANK(T, x):** return the rank of x in the linear order determined by an inorder walk of T .

Red-black trees

REMINDER

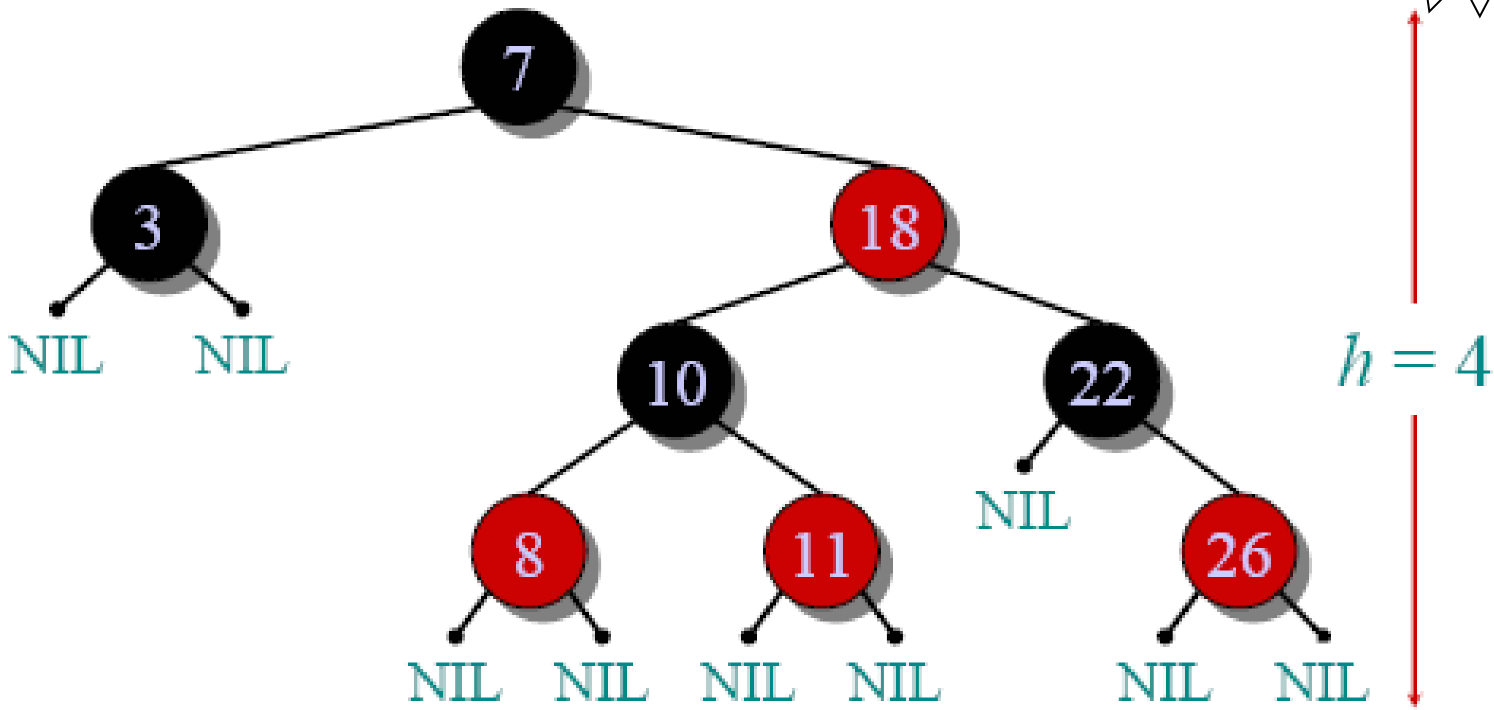
BSTs (Binary Search Tree) with an extra one-bit color field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Red-black tree example

REMINDER



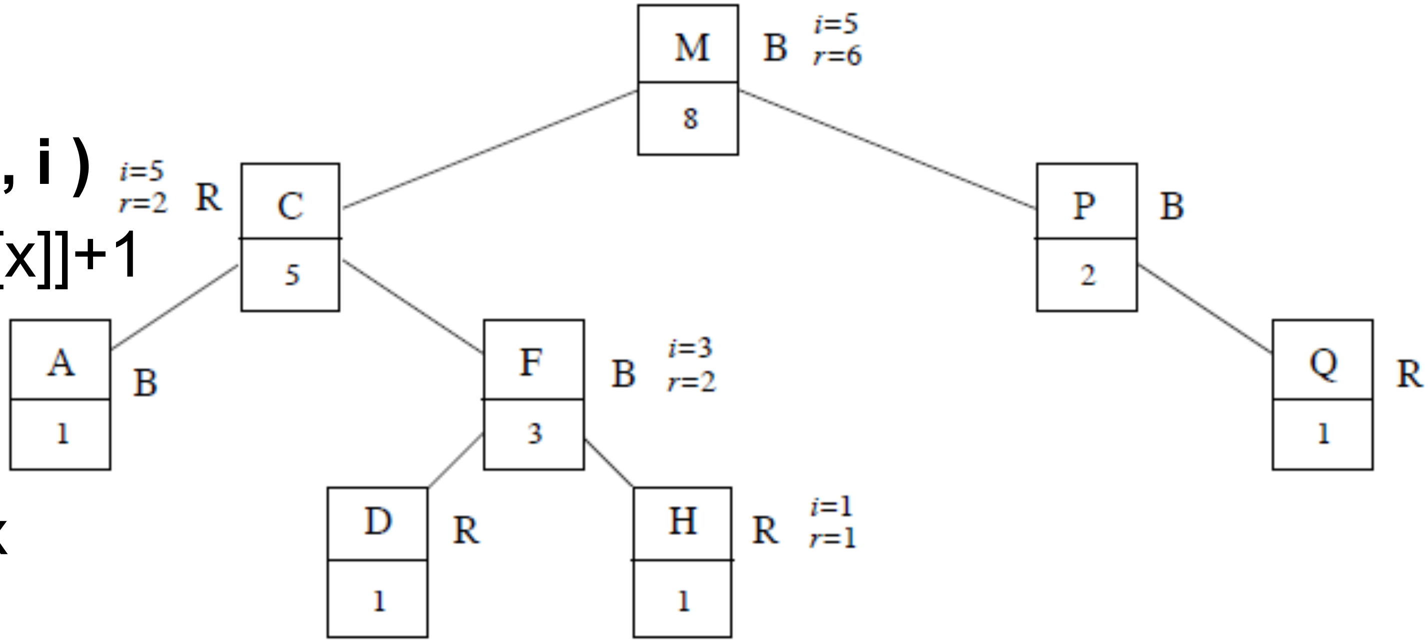
Red-Black Tree Augmenting

- store in each node x:
- size[x] = # of nodes in subtree rooted at x .
  - Includes x itself.
  - Does not include leaves (sentinels).
- Define for sentinel size[nil[T]] = 0.
- Then size[x] = size[left[x]]+size[right[x]]+1

- Note: OK for keys to not be distinct. Rank is defined with respect to position in inorder walk. So if we changed D to C, rank of original C is 2, rank of D changed to C is 3.

OS-SELECT(x, i): return pointer to node containing the i th smallest key of the subtree rooted at x.

OS-SELECT(x, i)



- r ← size[left[x]]+1
- if i = r
- then return x
- elseif i < r
- then return OS-SELECT(left[x], i)
- else return OS-SELECT(right[x], i − r)
- Initial call: OS-SELECT(root[T], i)
- Example: OS-SELECT(root[T], 5) (see figure).
- Note: It is OK for keys to not be distinct. Rank is defined with respect to position in inorder walk. So if we changed D to C, rank of original C is 2, rank of D changed to C is 3.

Proof of Correctness and Efficiency

Correctness: r = rank of x within subtree rooted at x.

- If i = r , then we want x.
- If i < r , then i th smallest element is in x.s left subtree, and we want the i th smallest element in the subtree.
- If i > r , then i th smallest element is in x.s right subtree, but subtract off the r elements in x.s subtree that precede those in x.s right subtree.
- Like the randomized SELECT algorithm!

Analysis: Each recursive call goes down one level. Since R-B tree has O(lg n) levels, have O(lg n) calls ⇒ O(lg n) time.

Randomized Select

Order Statistics Chapter, Finding the element of order i REMINDER

- RANDOMIZED-SELECT(*A, p, r, i)*
- 1 **if *p = r***
- 2 **then return *A[p]***
- 3 *q* ← *RANDOMIZED-PARTITION(A, p, r)*
- 4 *k* ← *q - p + 1*
- 5 **if *i = k*** ▹ ***the pivot value is the answer***
- 6 **then return *A[q]***
- 7 **elseif *i < k***
- 8 **then return RANDOMIZED-SELECT(*A, p, q - 1, i)***
- 9 **else return RANDOMIZED-SELECT(*A, q + 1, r, i - k)***

expected time of RANDOMIZED-SELECT is $\Theta(n)$.

OS-RANK(T,x)

//OS-RANK(T, x): return the rank of x in the linear order determined by an inorder walk of T .

- OS-RANK(T, x)
- r ← size[left[x]] + 1
- y ← x
- while y != root[T]
- do if y = right[p[y]]
- then r ←r + size[left[p[y]]] + 1
- y ← p[y]
- return r

Correctness

Loop invariant: At start of each iteration of while loop, r = rank of key[x] in subtree rooted at y.

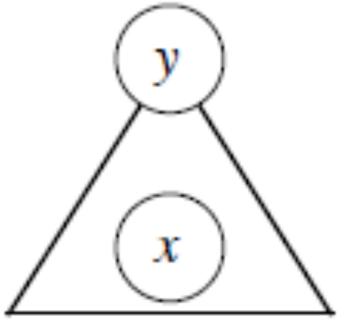
Initialization: Initially, r = rank of key[x] in subtree rooted at x, and y = x.

Termination: Loop terminates when y = root[T] ⇒ subtree rooted at y is entire tree. Therefore, r = rank of key[x] in entire tree.

Maintenance: At end of each iteration, set y ← p[y]. So, show that if r = rank of key[x] in subtree rooted at y at start of

loop body, then r = rank of key[x] in subtree

rooted at p[y] at end of loop body.



[r = # of nodes in subtree rooted at y preceding x in inorder walk]

Correctness (cont)

Maintenance: At end of each iteration, set y ← p[y]. So, show that if r = rank of key[x] in subtree rooted at y at start of

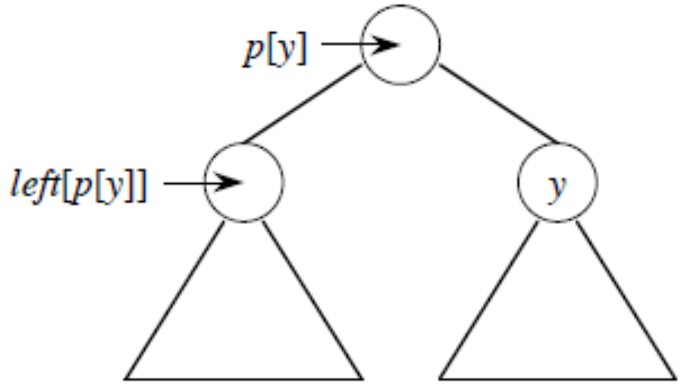
loop body, then r = rank of key[x] in subtree

rooted at p[y] at end of loop body.

Must add nodes in y.s sibling.s subtree.

If y is a left child, its sibling.s subtree follows all nodes in y.s subtree ⇒ don.t change r .

If y is a right child, all nodes in y.s sibling.s subtree precede all nodes in y.s subtree ⇒add size of y.s sibling.s subtree, plus 1 for p[y], into r .



Analysis: y goes up one

level in each iteration ⇒

O(lg n) time.

Efficiency

Maintaining subtree sizes

Need to maintain *size[x] fields during insert and delete operations.*

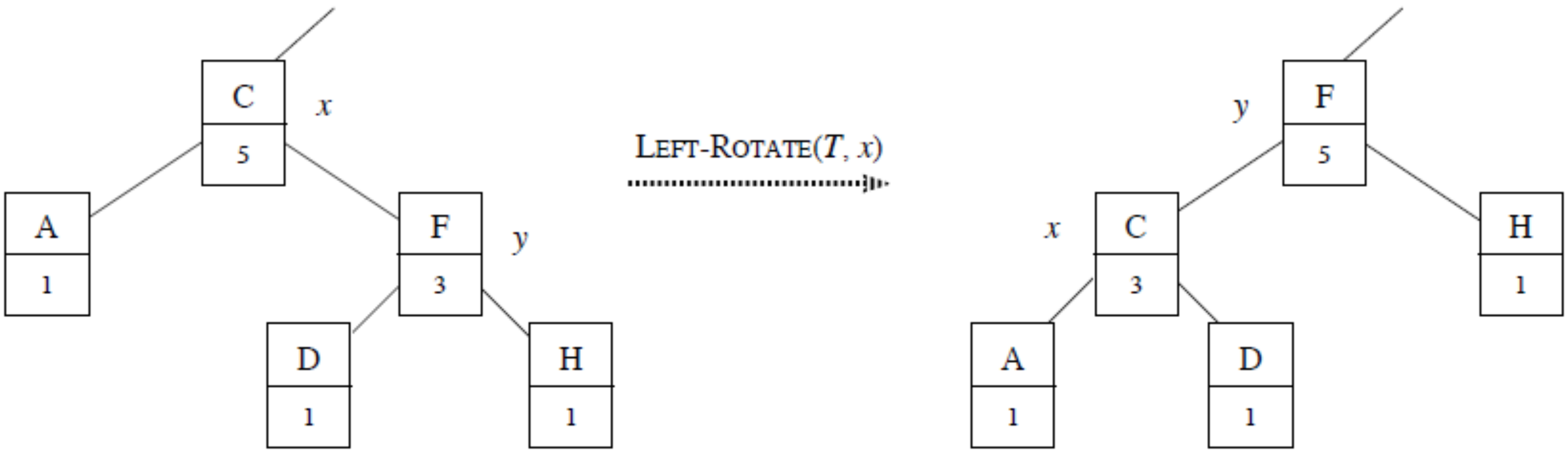
Need to maintain them efficiently. Otherwise, might have to recompute them all, at a cost of *(n).*

Will see how to maintain without increasing *O(lg n) time for insert and delete.*

Efficiency

- ***Insert:***
- During pass downward, we know that the new node will be a descendant of
- each node we visit, and only of these nodes. Therefore, increment *size field of* each node visited.
- Then there is the fixup pass:
- • Goes up the tree.
- • Changes colors *O(lg n) times.*
- • Performs ≤ 2 rotations.
- • Color changes don.t affect subtree sizes.
- • Rotations do!
- • But we can determine new sizes based on old sizes and sizes of children.

Efficiency (cont)



$$size[y] \leftarrow size[x]$$
$$size[x] \leftarrow size[left[x]] + size[right[x]] + 1$$

- Similar for right rotation.
- • Therefore, can update in *O(1) time per rotation ⇒ O(1) time spent updating*
- *size fields during fixup.*
- • Therefore, *O(lg n) to insert.*

Efficiency (cont)

- ***Delete: Also 2 phases:***
- 1. Splice out some node *y.*
- 2. Fixup.
- After splicing out *y, traverse a path y* → *root, decrementing size in each node on* path. *O(lg n) time.*
- During fixup, like insertion, only color changes and rotations.
- • ≤ 3 rotations ⇒ *O(1) time spent updating size fields during fixup.*
- • Therefore, *O(lg n) to delete.*
- Done!

Methodology for augmenting a data structure

- 1. Choose an underlying data structure.
- 2. Determine additional information to maintain.
- 3. Verify that we can maintain additional information for existing data structure
- operations.
- 4. Develop new operations.

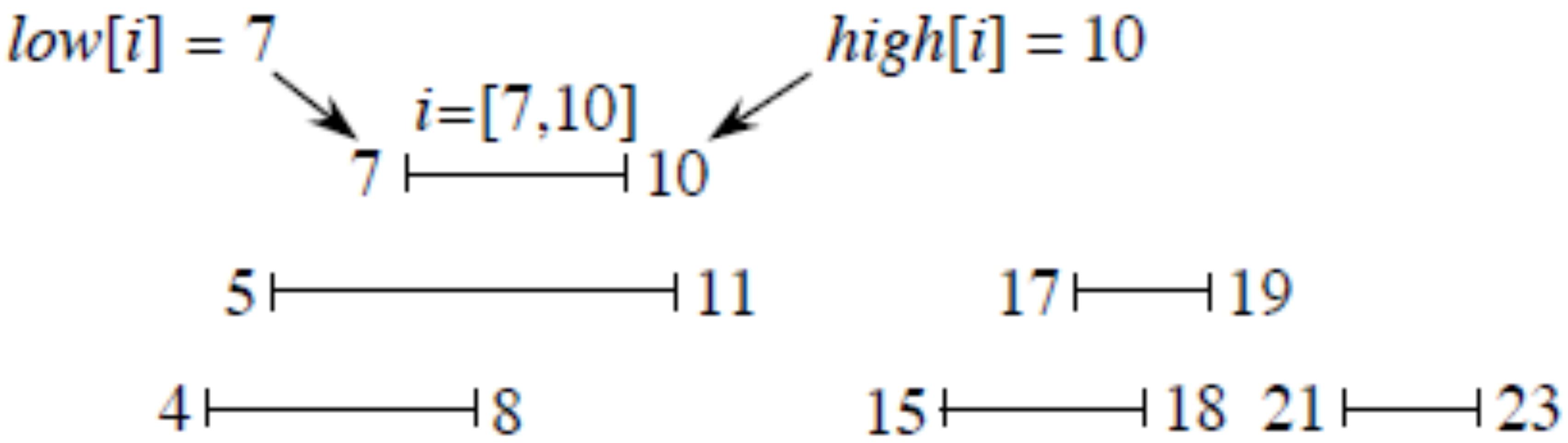
1. R-B tree.
2. *size[x].*
3. Showed how to maintain *size during insert and delete.*
4. Developed OS-SELECT and OS-RANK.

Red-Black Trees Amenable to Augmentation

- ***Theorem***
- Augment a R-B tree with field *f , where f [x] depends only on information in x, left[x], and right[x] (including f [left[x]] and f [right[x]]). Then can maintain* values of *f in all nodes during insert and delete without affecting O(lg n) performance.*
- ***Proof Since f [x] depends only on x and its children, when we alter information*** in *x, changes propagate only upward (to p[x], p[p[x]], . . . , root).*
- Height = *O(lg n)*$\Rightarrow$ *O(lg n) updates, at O(1) each.*
- ***Insertion: see the book***
- ***Delete: see the book***

Interval Trees

Maintain a set of intervals. For instance, time intervals.



Interval Tree Properties

- **Operations**
- • INTERVAL-INSERT*(T, x): int[x] already filled in.*
- • INTERVAL-DELETE*(T, x)*
- • INTERVAL-SEARCH*(T, i): return pointer to a node x in T such that int[x] overlaps* interval *i . Any overlapping node in T is OK. Return pointer to sentinel nil[T] if no overlapping node in T .*
- Interval *i has low[i], high[i].*
- *i and j overlap if and only if low[i] ≤ high[j] and low[j] ≤ high[i].*

Augmenting to Get Interval-Trees

For interval trees

1. Use R-B trees.

Each node *x contains interval int[x].*

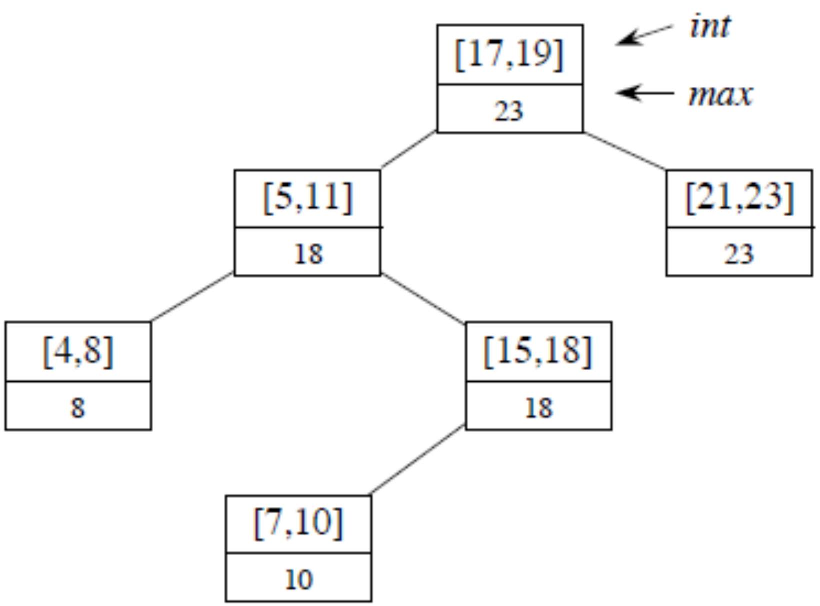
Key is low endpoint (*low[int[x]]).*

Inorder walk would list intervals sorted by low endpoint.

2. Each node *x contains*

max[x] = max endpoint

value in subtree rooted at x .



Summary

Binary Search Tree (BST) review

Red and Black Trees

2-3 and 2-3-4 trees

Operations on Red and Black Trees