Lecture Notes for Chapter 14: Augmenting Data Structures

Chapter 14 overview

We'll be looking at methods for *designing* algorithms. In some cases, the design will be intermixed with analysis. In other cases, the analysis is easy, and it's the design that's harder.

Augmenting data structures

- It's unusual to have to design an all-new data structure from scratch.
- It's 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*.

We'll look at a couple of situations in which we augment red-black trees.

Dynamic order statistics

We want to support the usual dynamic-set operations from R-B trees, plus:

- OS-SELECT(x, i): return pointer to node containing the ith 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.

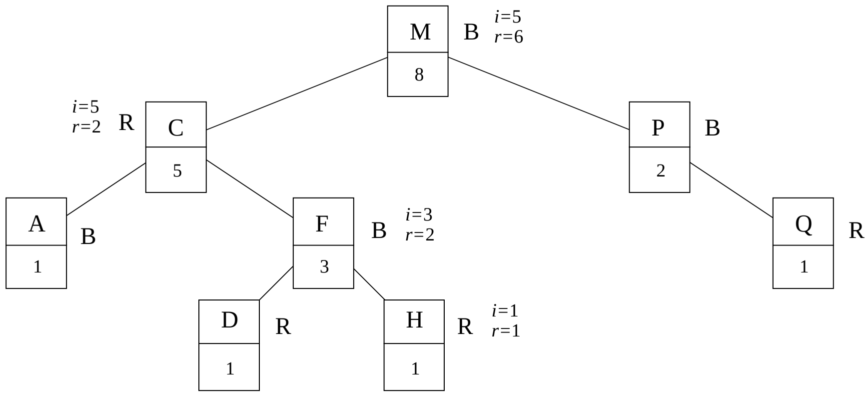
Augment by storing in each node x:

$x.size = \#$ of nodes in subtree rooted at x .

- Includes x itself.
- Does not include leaves (sentinels).

Define for sentinel $T.nil.size = 0$.

Then $x.size = x.left.size + x.right.size + 1$.



*[**Example above:** Ignore colors, but legal coloring shown with "R" and "B" notations. Values of i and r are for the example below.]*

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)
  $r = x.left.size + 1$
  if $i == r$
    return x
  elseif $i < r$
    return OS-SELECT$(x.left, i)$
  else return OS-SELECT$(x.right, i - r)$

Initial call: OS-SELECT$(T.root, i)$

Try OS-SELECT$(T.root, 5)$. *[Values shown in figure above. Returns node whose key is H.]*

Correctness

r = rank of x within subtree rooted at x.

- If $i = r$, then we want x.
- If $i < r$, then ith smallest element is in x's left subtree, and we want the ith smallest element in the subtree.
- If $i > r$, then ith 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 $\Rightarrow O(\lg n)$ time.

OS-RANK(T, x)

$r = x.left.size + 1$
$y = x$
while $y \neq T.root$
  if $y == y.p.right$
    $r = r + y.p.left.size + 1$
  $y = y.p$
return r

Demo: Node D.

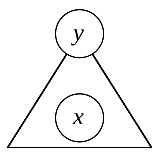
Why does this work?

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

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

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

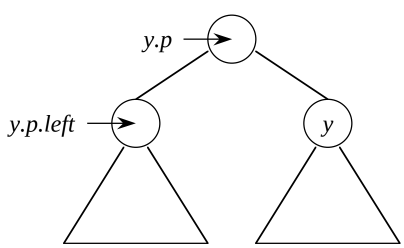
Maintenance: At end of each iteration, set $y = y.p$. So, show that if r = rank of $x.key$ in subtree rooted at y at start of loop body, then r = rank of $x.key$ in subtree rooted at $y.p$ at end of loop body.



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

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 $\Rightarrow$ 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 $\Rightarrow$ add size of y's sibling's subtree, plus 1 for $y.p$, into r.



Analysis

y goes up one level in each iteration $\Rightarrow O(\lg n)$ time.

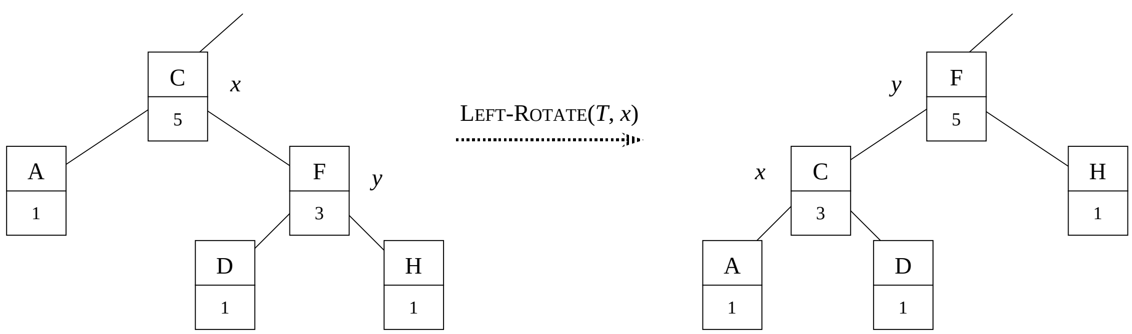
Maintaining subtree sizes

- Need to maintain *size* attributes during insert and delete operations.
- Need to maintain them efficiently. Otherwise, might have to recompute them all, at a cost of $\Omega(n)$.

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

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* attribute of each node visited.
- Then there's 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.



$$\begin{aligned} y.size &= x.size \\ x.size &= x.left.size + x.right.size + 1 \end{aligned}$$

- Similar for right rotation.
- Therefore, can update in $O(1)$ time per rotation $\Rightarrow$ $O(1)$ time spent updating *size* attributes during fixup.
- Therefore, $O(\lg n)$ to insert.

Delete

Also 2 phases:

1. Splice out some node y.
2. Fixup.

After splicing out y, traverse a path $y \to root$, decrementing *size* in each node on path. $O(\lg n)$ time.

During fixup, like insertion, only color changes and rotations.

- ≤ 3 rotations $\Rightarrow O(1)$ time spent updating *size* attributes 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.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

How did we do them for OS trees?

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

Red-black trees are particularly amenable to augmentation.

Theorem

Augment a R-B tree with attribute f, where $x.f$ depends only on information in x, $x.left$, and $x.right$ (including $x.left.f$ and $x.right.f$). Then can maintain values of f in all nodes during insert and delete without affecting $O(\lg n)$ performance.

Proof Since $x.f$ depends only on x and its children, when we alter information in x, changes propagate only upward (to $x.p$, $x.p.p$, $x.p.p.p$, $\ldots$, *root*).

Height = $O(\lg n) \Rightarrow O(\lg n)$ updates, at $O(1)$ each.

Insertion

Insert a node as child of existing node. Even if can't update f on way down, can go up from inserted node to update f. During fixup, only changes come from color changes (no effect on f) and rotations. Each rotation affects f of ≤ 3 nodes (x, y, and parent), and can recompute each in $O(1)$ time. Then, if necessary, propagate changes up the tree. Therefore, $O(\lg n)$ time per rotation. Since ≤ 2 rotations, $O(\lg n)$ time to update f during fixup.

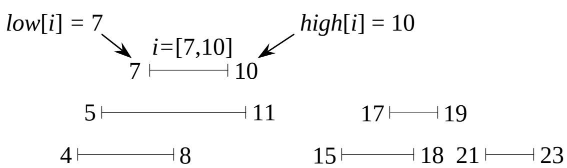
Delete

Same idea. After splicing out a node, go up from there to update f. Fixup has ≤ 3 rotations. $O(\lg n)$ per rotation $\Rightarrow$ $O(\lg n)$ to update f during fixup. ■ (theorem)

For some attributes, can get away with $O(1)$ per rotation. Example: *size* attribute.

Interval trees

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



[leave on board]

Operations

- INTERVAL-INSERT(T, x): $x.int$ already filled in.
- INTERVAL-DELETE(T, x)
- INTERVAL-SEARCH(T, i): return pointer to a node x in T such that $x.int$ overlaps interval i. Any overlapping node in T is OK. Return pointer to sentinel $T.nil$ if no overlapping node in T.

Interval i has $i.low$, $i.high$.

i and j overlap if and only if
$i.low \leq j.high$ and $j.low \leq i.high$.

(Go through examples of proper inclusion, overlap without proper inclusion, no overlap.)

Another way: i and j *don't* overlap if and only if
$i.low > j.high$ or $j.low > i.high$.
[leave this on board]

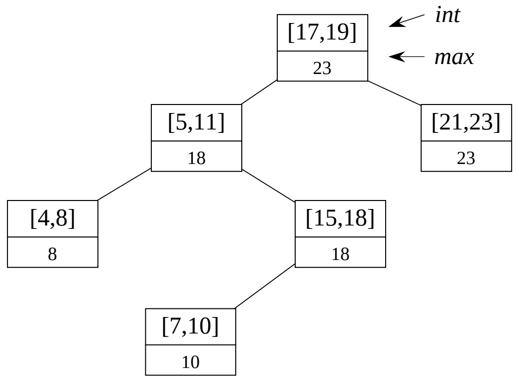
Recall the 4-part methodology.

For interval trees

1. Use R-B trees.
   - Each node x contains interval $x.int$.
   - Key is low endpoint ($x.int.low$).
   - Inorder walk would list intervals sorted by low endpoint.

2. Each node x contains

   $x.max$ = max endpoint value in subtree rooted at x .



   [leave on board]

$$x.max = max \begin{cases} x.int.high\ , \\ x.left.max\ , \\ x.right.max \end{cases}$$

   Could $x.left.max > x.right.max$? Sure. Position in tree is determined only by low endpoints, not high endpoints.

3. Maintaining the information.
   - This is easy—$x.max$ depends only on:
     - information in x: $x.int.high$
     - information in $x.left$: $x.left.max$
     - information in $x.right$: $x.right.max$
   - Apply the theorem.
   - In fact, can update *max* on way down during insertion, and in $O(1)$ time per rotation.

4. Developing new operations.

   INTERVAL-SEARCH(T, i)

   $x = T.root$
   while $x \neq T.nil$ and i does not overlap $x.int$
       if $x.left \neq T.nil$ and $x.left.max \geq i.low$
           $x = x.left$
       else $x = x.right$
   return x

Examples

Search for $[14, 16]$ and $[12, 14]$.

Time

$O(\lg n)$.

Correctness

Key idea: need check only 1 of node's 2 children.

Theorem

If search goes right, then either:

- There is an overlap in right subtree, or
- There is no overlap in either subtree.

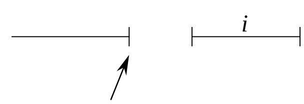
If search goes left, then either:

- There is an overlap in left subtree, or
- There is no overlap in either subtree.

Proof If search goes right:

- If there is an overlap in right subtree, done.
- If there is no overlap in right, show there is no overlap in left. Went right because
  - $x.left = T.nil \Rightarrow$ no overlap in left.

    OR
  - $x.left.max < i.low \Rightarrow$ no overlap in left.



x.left.max = highest endpoint in left

If search goes left:

- If there is an overlap in left subtree, done.
- If there is no overlap in left, show there is no overlap in right.
  - Went left because:

    $$\begin{aligned} i.low &\le x.left.max \\ &= j.high \text{ for some } j \text{ in left subtree} . \end{aligned}$$
  - Since there is no overlap in left, i and j don't overlap.
  - Refer back to: no overlap if

    $$i.low > j.high \text{ or } j.low > i.high .$$
  - Since $i.low \le j.high$, must have $j.low > i.high$.
  - Now consider *any* interval k in *right* subtree.
  - Because keys are low endpoint,

    $$\underbrace{j.low}_{\text{in left}} \le \underbrace{k.low}_{\text{in right}} .$$
  - Therefore, $i.high < j.low \le k.low$.
  - Therefore, $i.high < k.low$.
  - Therefore, i and k do not overlap. ■ (theorem)