6.046 Recitation 6: Augmenting Data Structures Bill Thies, Fall 2004

Acknowledgements:

Parts of these notes were borrowed from David Liben-Nowell.

My contact information:

Bill Thies

thies@mit.edu

Office hours: Sat 1-3pm, 36-153

Recitation website: http://cag.lcs.mit.edu/~thies/6.046/

Announcements:

- PS 3, due Mon 10/18
- Correction on 2c: give an upper bound, not a lower bound

Today:

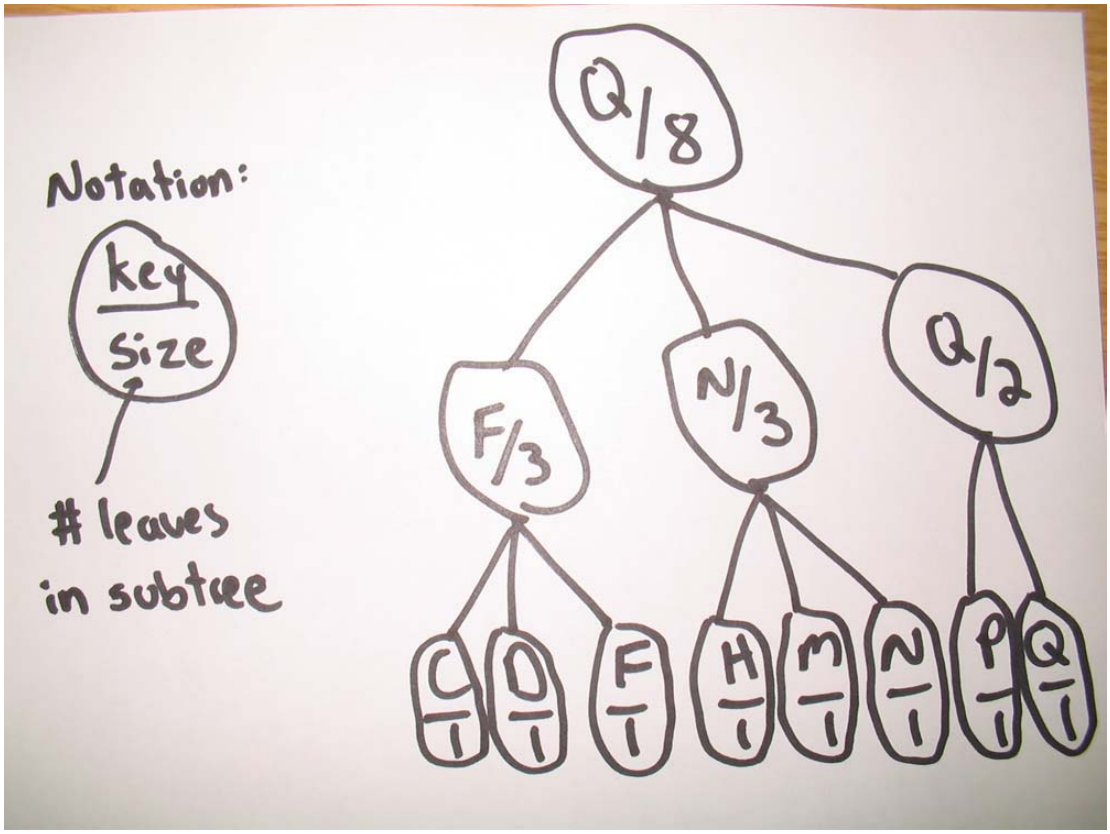
- Augmenting data structures

Dynamic Order Statistics

Insert(X, s)
Delete(S, x)
OS-Select(S, i) – returns i'th smallest element in S

Recall: select on array is $\Theta(n)$
Idea: Use 2-3 tree for S, but keep subtree sizes in nodes

Example OS-Tree



Size(x) = size(left(x)) + size(mid(x)) + size(right(x)) if x is internal node
= 1 if x is leaf
= 0 if x is NIL

OS-Select

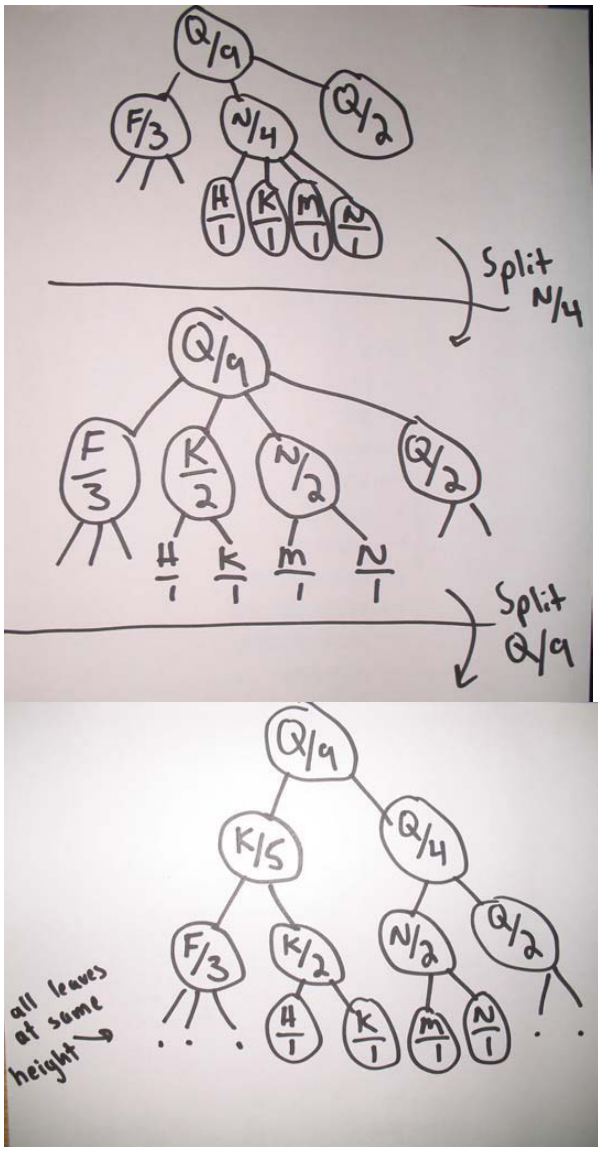
```
OS-Select(x, i)    // returns i'th smallest element in subtree rooted at x
  if i > size(x) or i < 0  then fail
  elseif x is leaf then return x
  elseif i ≤ size(left(x))
        then return OS-Select(left(x), i)
  elseif i ≤ size(left(x)) + size(right(x))
        then return OS-Select(mid(x), i – size(left(x))
  else return OS-Select(right(x), i – size(left(x) – size(mid(x))
```

Runtime? Θ(lg n) because 2-3 tree is balanced

<< can also implement OS-Rank(S, x) in similar way, see textbook >>

Modifying Ops: Insert, Delete

Insert “K”:



Two changes to Insert:

1. Increment size of internal nodes on path from root to leaf
2. Modify Split to recompute new size in parent
   - Time for new split (to run on single node?) = O(1)

Thus, Insert and Delete still work in $\Theta(\lg n)$ time on 2-3 tree

Question: Why not keep ranks in nodes instead of subtree sizes?

Answer: Might have to update all nodes in tree (consider inserting new minimum element); would become $\Theta(n)$ insert time

Methodology (Augmenting Data Structures)

In general		In this case
1.	Choose underlying data structure	2-3 trees
2.	Determine extra info	subtree size
3.	Develop new operations that use info	OS-Select
4.	Verify that info can be maintained for modifying ops	insert, delete, Split!

Memory Allocation

Support two operations:

1. malloc(size) - return free block of given size
   - further assume “First-Fit”: return minimum address x such that (x, x+size) is free
2. free(x) - mark block x as free

Example: Consider free regions: (0,3), (15, 20), (22, 32), (35, 55), (60, 70)

malloc(2) = [0,1]
malloc(12) = [35,46]

Naïve implementation: Linked list of free regions. Worst case runtime? $\Theta(n)$

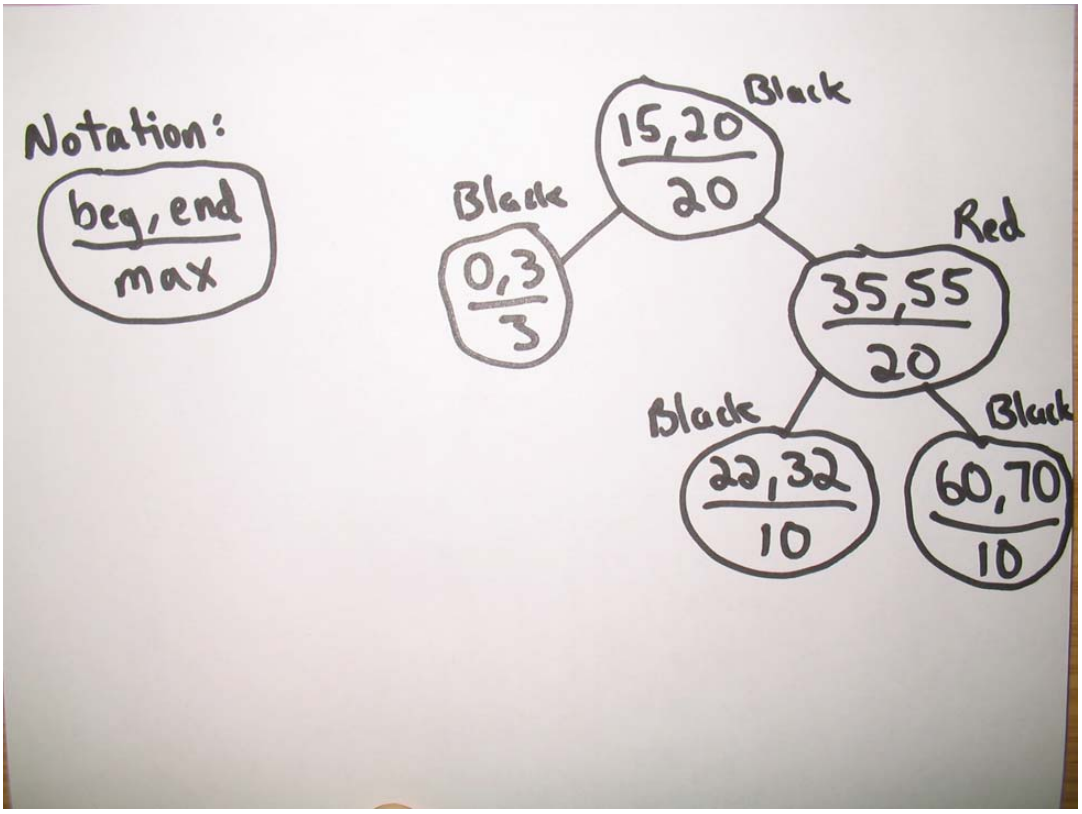
Using a balanced tree? $\Theta(\lg n)$ [n = max number of addresses]

Augmented Data Structure for Memory Allocation

Following methodology from before:

1. Data structure = red / black tree
2. Info for each node x:
    - beg(x) - beginning of block
    - end(x) - end of block
    - max(x) - size of max free block in subtree

Example tree, keeps track of free regions that we listed before:



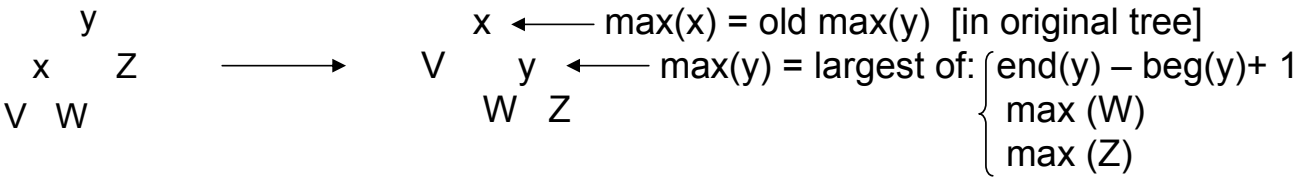
Malloc

In class, we only gave the intuition for malloc.

```
Malloc(size, T) {
  if max(T) < size
        return Nil
  if max(left(T)) ≥ size
        return Malloc(size, left(T))
  if (end(T) – beg(T) + 1 ≥ size)
        block = [beg(T), beg(T) + size – 1]
        delete T
        if (beg(T) + size ≤ end(T))
            then insert [beg(T) + size, end(T)]
        return block
  else return Malloc(size, right(T))
}
```

Maintaining Max Field

In red-black trees, the fundamental maintenance operation is *rotation*. We need to show how to compute the augmented information following a rotation:



beg and end fields remain the same.

Update time? O(1).

Insert and delete still run in Θ(lg n) time

Free

We did not discuss this in section.

```
Free(start, size, T)
  if end(Pred(start+size+1)) ≥ start
        then fail  // block is already free!

   // if there is neighboring free block, need to coalesce blocks into one
  pred = Predecessor(start, T)
  succ = Successor(start+size-1, T)

  // test combination with predecessor
  if end(pred) = start-1
        then start = beg(pred)
              size = size + size (pred)
              delete (pred)

  // test combination with successor
  if beg(succ) = start + size
        then size = size + size(succ)
              delete (succ)

  // add combined node
  insert [start, start + size] into T
```