Lecture 11: March 20, 2001

Admin:

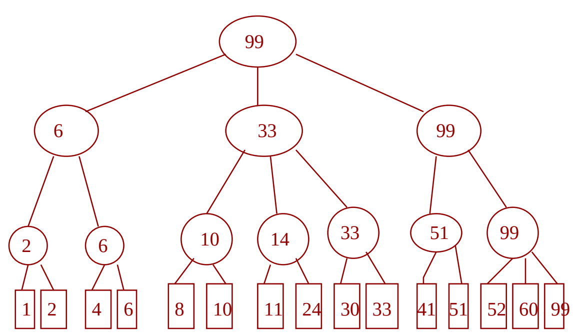
- Quiz 1 Stats
  - Mean 40.4, Std. Dev. 14.4 (Total = 80).
  - Problem 3 "too hard". Effective total = 56.
- Lots of handouts go online today/tomorrow.
  - Lecture notes
  - Practice Problem Set + Soln. (attempt to solve before recitation Friday).

Today:

- Deletion in 2-3 Trees
- Red-Black Trees vs. 2-3 Trees
- Augmenting Data Structures:
  - Dynamic Order Statistics
  - Interval Trees

Recall 2-3 Trees

- Every internal node has 2 or 3 children.
- All leaves at same level.
- All data at leaves (sorted).
- Internal node stores max. key in subtree.

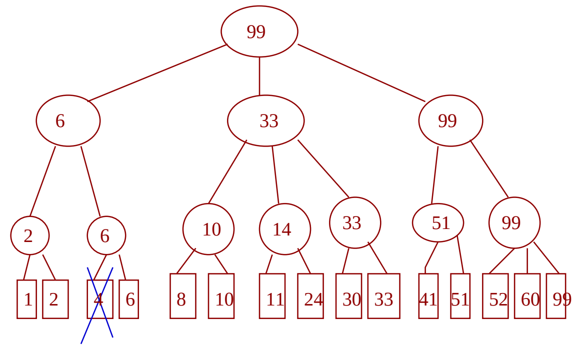


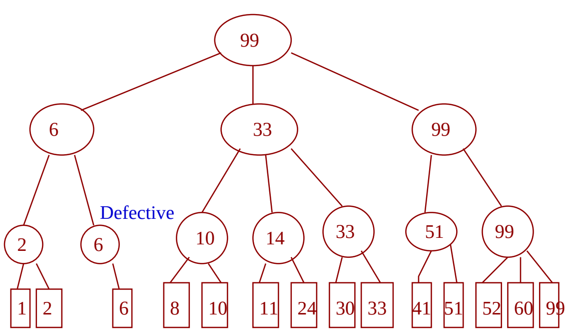
From Last lecture

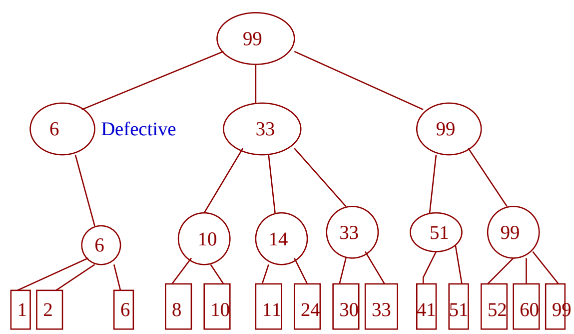
- Depth of 2-3 tree storing n keys $O(\log n)$.
- How to Search and Insert in $O(\log n)$ time.
- Today: How to delete in $O(\log n)$ time per operation.

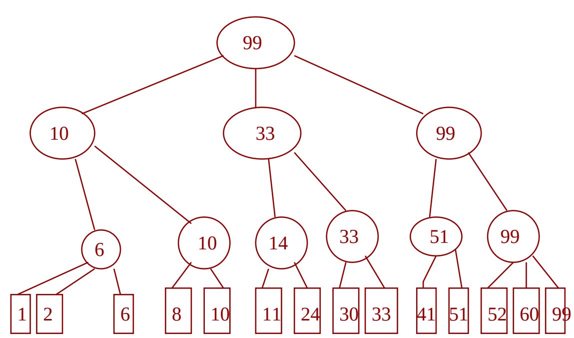
Basic Idea

- Similar to Insert.
- First, delete the leaf.
- May result in a *defective* internal node:
  - I.e., Internal node with only one child.
- Idea to fix defective node v:
  - If Parent(v) has enough grandchildren, can fix the problem at that level.
  - Else, move the *defect* upwards.
  - Will eventually stop at root level.
- Also fix max value along root-leaf path.









Deletion Details

- When can fix be finished at Parent(v)?
  - If Parent(v) has 4 or more grandchildren ... then can fix it! (In some big, but $O(1)$, time).
- So what to do if Parent(v) has 3 grandchildren only?
  - Make all three children of v.
  - Now Parent(v) is *defective*.
- Note defect is moving up. If root becomes defective, just throw it away!

Pros & Cons

- 2-3 Trees conceptually simple. All ideas natural.
- Actual implementation/code is hard. Too many cases of the form "If node has three children, then ...."
- Sometimes, we just like binary trees!
- Can we convert a 2-3 tree into a binary one?
- Yes ....

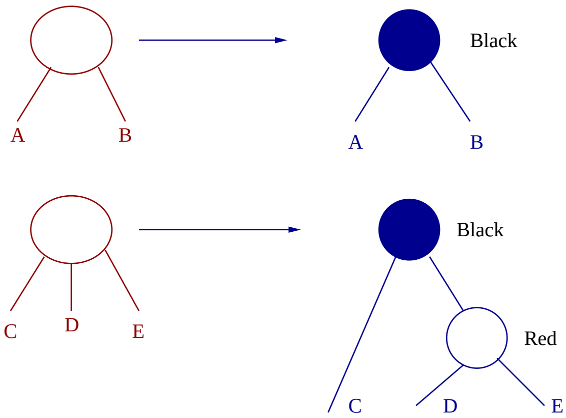
Red Black Trees

- (Somewhat) balanced *binary* search trees.
- Every node colored Red/Black.
- Every internal node has two children.
- No Red parent with Red child.
- Every root-leaf path has same number of Black nodes.
- Keys stored in leaves. Leaves are sorted. Internal nodes store Max.
- (In some texts data stored at internal nodes.)

- Fact: Red-Black trees with n leaves have depth $O(\log n$

2-3 Trees to Red Black Trees

- Can easily implement 2-3 trees + algorithms on 2-3 trees via Red-Black trees.
- Main idea: convert each node of 2-3 tree into a Red-Black subtree (subgraph?).



Augmenting Data Structures

Start "design" phase of course:

- So far, we have used one design technique: *divide and conquer.*
- Today, we look at the technique of *augmenting data structures.*

Dynamic Order Statistics

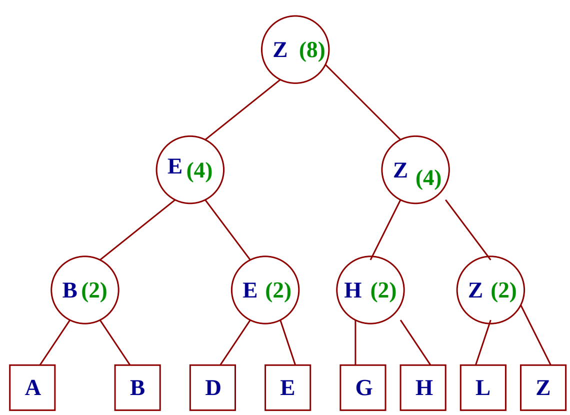
Want to support ordinary dynamic set operations *plus:*

- OS-SELECT(x, i) — returns ith smallest key in subtree rooted at x
- OS-RANK(T, x) — returns rank (position) of x in linear order of tree T

Idea: Keep subtree sizes in nodes of a 2-3 tree.

Dynamic Order Statistics

Sample Tree:
New field *size*$[x]$:



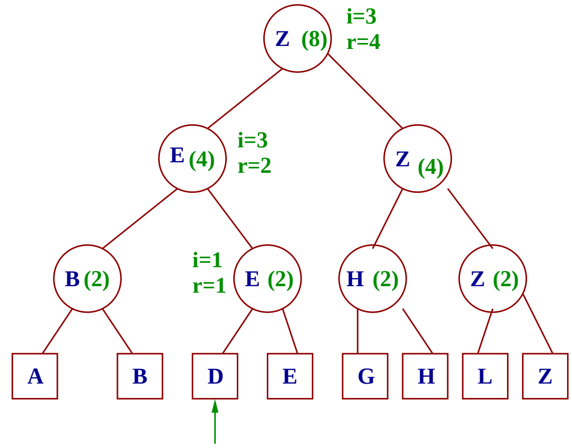
Dynamic Order Statistics: OS-Select

- OS-SELECT — Retrieve an element with specified rank
- Pseudocode: (Assume for simplicity all nodes have 2 children - saves messy pseudocode)

OS-SELECT(x, i)
$r \leftarrow size[left[x]]+1$
if $i = r$
  then return x
elseif $i < r$
  then return OS-SELECT$(left[x], i)$
else return OS-SELECT$(right[x], i - r)$

- Idea: Knowing the size of the left subtree tells you in which subtree the sought key must lie.

Dynamic Order Statistics



Example :

Find OS-SELECT$(root, 5)$ on the sample tree.

Dynamic Order Statistics: OS-Rank

- OS-RANK — Given a pointer to an element, determine its rank in linear order
- Pseudocode in CLR (§15.1, p. 283)
- Runs in $O(\lg n)$ time.

Dynamic Order Statistics: Subtree Structure

Maintaining subtree sizes:

- Can data structure be maintained during tree-modifying operations (INSERT, DELETE)?

Dynamic Order Statistics: Subtree Structure

Maintaining subtree sizes:

- Update subtree sizes when inserting/deleting.
- INSERT, DELETE still $O(\lg n)$ time.

Augmenting Data Structures: Methodology

Methodology:
(with "e.g." for our order-statistic trees)

1. Choose underlying data structure
   (e.g. 2-3 trees)
2. Determine additional information to maintain
   (e.g. subtree sizes)
3. Verify that the information can be maintained for all operations that modify the data structure
   (e.g. INSERT, DELETE — don't forget rotations!)
4. Develop new operations
   (e.g. OS-RANK, OS-SELECT)

Order of steps may vary and be intermixed in real design.

Augmenting Data Structures: Interval Trees

Interval Trees:

The problem:

- Maintain a set of intervals (e.g., time intervals).
- Query: Find an interval in the set that overlaps a given query point.

Interval Trees: Example

Intervals: $[4, 8]$, $[5, 11]$, $[7, 19]$, $[15, 18]$, $[17, 19]$, $[21, 23$
Queries:

Query point 22: answer $[21, 23]$.
Query point 9: answer $[5, 11]$ or $[7, 19]$.
Query point 20: answer NONE.

Augmenting Data Structures: Methodology

Interval Trees:

Following the methodology:

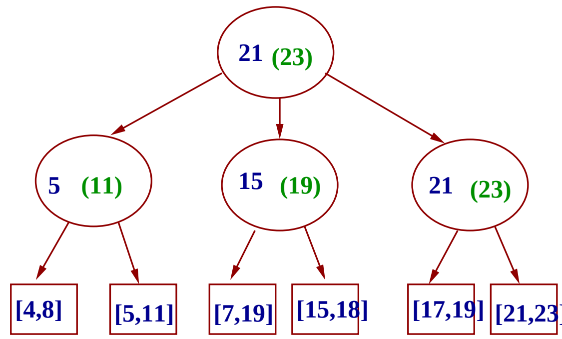
Underlying data structure

2-3 tree containing intervals:

- Leaves store pairs of integers:
  - *high* and *low* endpoint.
- Are keyed on *low* (left) endpoint

Augmenting Data Structures: Methodology

- Usual information at internal nodes:
  - Max left endpt. in subtree.
- Additional information at internal node:
  - Max right endpt. in subtree.



- Warning! Must check that additional info can be maintained during updates. (Not a problem in this case.)

Augmenting Data Structures: Methodology

Develop new operations

Interval-Search
(to find aninterval overlapping a given point)
Basic idea:

- If (Max-right-endpt(leftchild) $<$ query pt.) search in right subtree, else search in left subtree.
- If at leaf, report interval if it overlaps query.

Correctness:

- Clearly doing the right thing in the true case of the if condition.
- Why is it ok to ignore the right subtree, if Max-right-endpt(leftchild) $\geq$ query pt.?
- Exercise!

Correctness

Claim: If $q \leq$Max-right-endpt(leftsubtree), and an overlapping interval exists in tree, then one exists in left subtree.

Proof:

- Let $[a, b]$ be overlapping interval and say it is in right subtree. Then $a \leq q \leq b$. If $[a, b]$ is in left subtree then we are done. So assume $[a, b]$ is in right subtree.
- Let b' be Max-right-endpt(left subtree). Let $[a', b']$ be the interval in the left subtree which has b' as the right endpt.
- Then $a' \leq a$ (since tree is keyed on left endpts), and thus $a' \leq q$.
- Also $b' \geq q$ (given by hypothesis of Claim).
- Thus $a' \leq q \leq b'$, and thus the interval $[a', b']$ is an interval in the left subtree that overlaps q, as required.