

Today:

- Balanced search structures (*DS(IJ)*, Chapter 9: *G&T*, §8.6, §9.2, §9.4, §9.5)

Coming Up:

- Pseudo-random Numbers (*DS(IJ)*, Chapter 11)

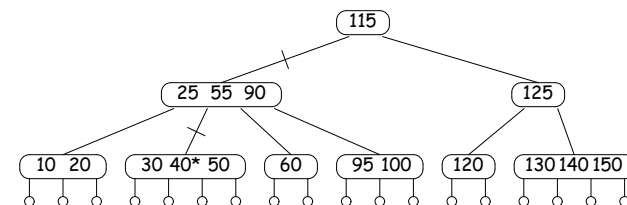
- Why are search trees important?
 - Insertion/deletion fast (on every operation, unlike hash table, which has to expand from time to time).
 - Support range queries, sorting (unlike hash tables)
- But $O(\lg N)$ performance from binary search tree requires remaining keys be divided \approx by 2 at each node.
- In other words, that tree be "bushy"
- "Stringy" trees (many nodes with one side much longer than other) perform like linked lists.
- Suffices that heights of two subtrees always differ by no more than constant K .

Example of Direct Approach: B-Trees

Idea: If tree grows/shrinks only at root, then two sides always have same height.

- Order M B-tree is an M -ary search tree, $M > 2$.
- Each node, except root, has from $\lceil M/2 \rceil$ to M children, and one key "between" each two children.
- Root has from 2 to M children (in non-empty tree).
- Children at bottom of tree are all empty (don't really exist) and equidistant from root.
- Obeys search-tree property:
 - Keys are sorted in each node.
 - All keys in subtrees to left of a key, K , are $< K$, and all to right are $> K$.
- Searching is simple generalization of binary search.
- Insertion: add just above bottom; split overfull nodes as needed, moving one key up to parent.

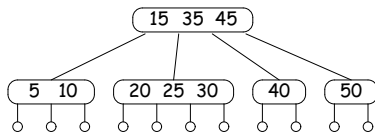
Sample Order 4 B-tree (2-4 Tree)



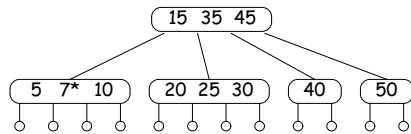
- Crossed lines show path when finding 40.
- Keys on either side of each child pointer in path bracket 40.
- Each node has at least 2 children, so height must be $O(\lg N)$.
- In real-life B-tree, order typically much bigger
 - comparable to size of disk sector, page, or other convenient unit of I/O

Inserting in B-tree (Simple Case)

- Start:

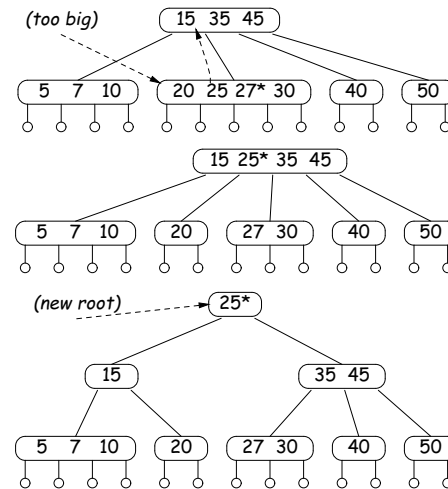


- Insert 7:



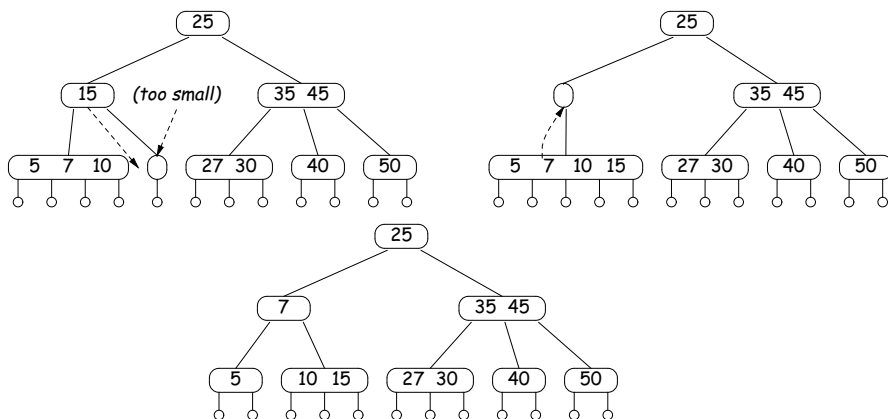
Inserting in B-Tree (Splitting)

- Insert 27:



Deleting Keys from B-tree

- Remove 20 from last tree.

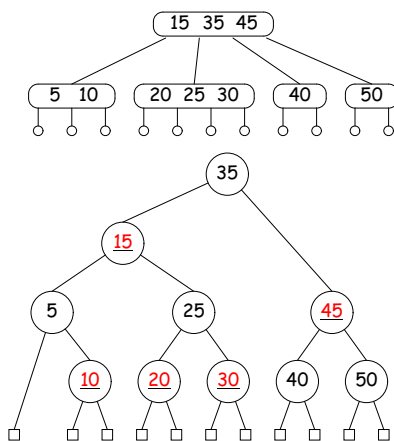


Red-Black Trees

- Red-black tree is a binary search tree with additional constraints that limit how unbalanced it can be.
- Thus, searching is always $O(\lg N)$.
- Used for Java's TreeSet and TreeMap types.
- When items are inserted or deleted, tree is rotated as needed to restore balance.
- Constraints:
 - Each node is (conceptually) colored red or black.
 - Root is black.
 - Every leaf node contains no data (as for B-trees) and is black.
 - Every leaf has same number of black ancestors.
 - Every internal node has two children.
 - Every red node has two black children.
- Conditions 4, 5, and 6 guarantee $O(\lg N)$ searches.

Sample Red-Black Tree

- Every red-black tree corresponds to a 2-4 tree, and the operations on one correspond to those on the other.

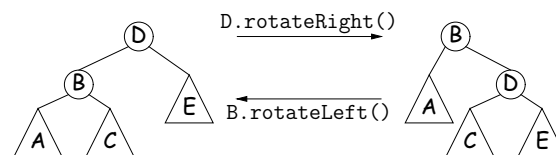


Last modified: Mon Nov 4 11:13:43 2002

CS61B: Lecture #30 9

Red-Black Insertion and Rotations

- Insert at bottom just as for binary tree (color red except when tree initially empty).
- Then rotate (and recolor) to restore red-black property, and thus balance.
- *Rotation of trees preserves binary tree property, but changes balance.*

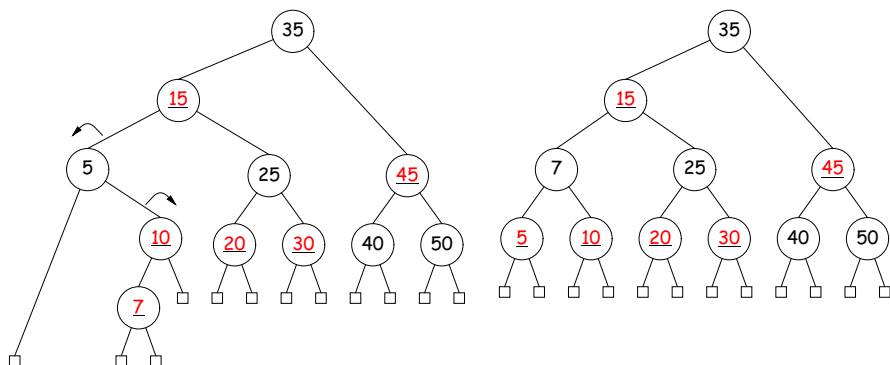


Last modified: Mon Nov 4 11:13:43 2002

CS61B: Lecture #30 10

Example of Red-Black Insertion (I)

- Insert 7:

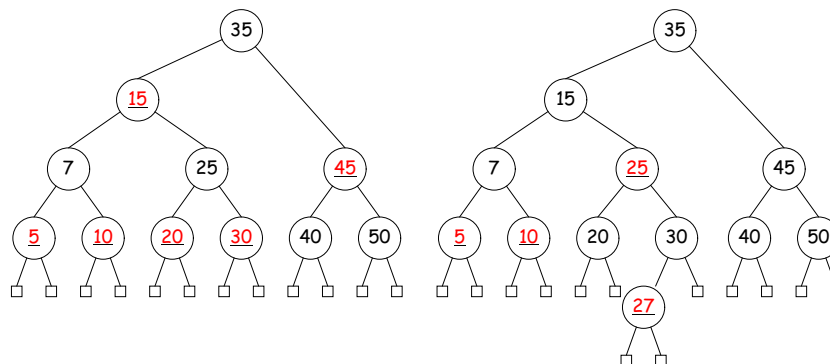


Last modified: Mon Nov 4 11:13:43 2002

CS61B: Lecture #30 11

Example of Red-Black Insertion (II)

- Insert 27, recolor to restore red-black property. Doesn't do any rebalancing, but sets things up to cause future insertions to rebalance.



Last modified: Mon Nov 4 11:13:43 2002

CS61B: Lecture #30 12

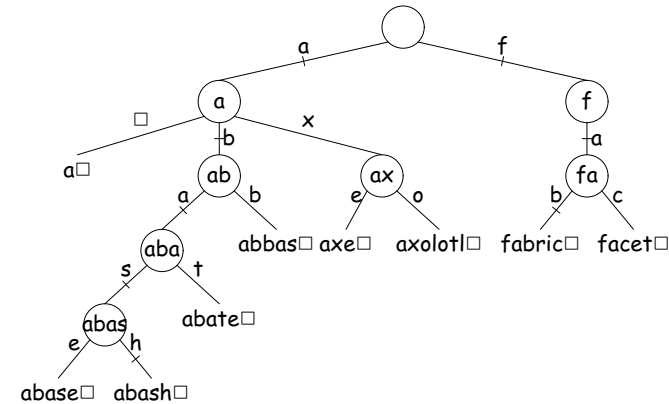
Really Efficient Use of Keys: the Trie

- Have been silent about cost of comparisons.
- For strings, worst case is length of string.
- Therefore should throw extra factor of key length, L , into costs:
 - $\Theta(M)$ comparisons really means $\Theta(ML)$ operations.
 - So to look for key X , keep looking at same chars of X M times.
- Can we do better? Can we get search cost to be $O(L)$?

Idea: Make a multi-way decision tree, with one decision per character of key.

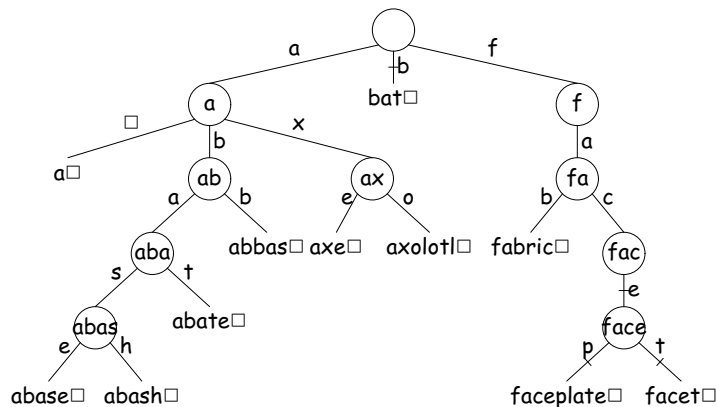
The Trie: Example

- Set of keys
 {a, abase, abash, abate, abbas, axolotl, axe, fabric, facet}
- Ticked lines show paths followed for "abash" and "fabric"
- Each internal node corresponds to a possible prefix.
- Characters in path to node = that prefix.



Adding Item to a Trie

- Result of adding bat and faceplate.
- New edges ticked.



A Side-Trip: Scrunching

- For speed, obvious implementation for internal nodes is array indexed by character.
- Gives $O(L)$ performance, L length of search key.
- [Looks as if independent of N , number of keys. Is there a dependence?]
- **Problem:** arrays are sparsely populated by non-null values—waste of space.

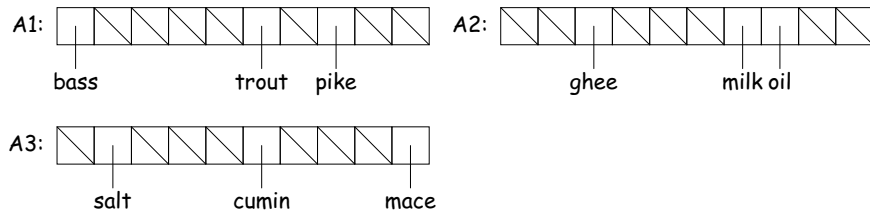
Idea: Put the arrays on top of each other!

- Use null (0, empty) entries of one array to hold non-null elements of another.
- Use extra markers to tell which entries belong to which array.

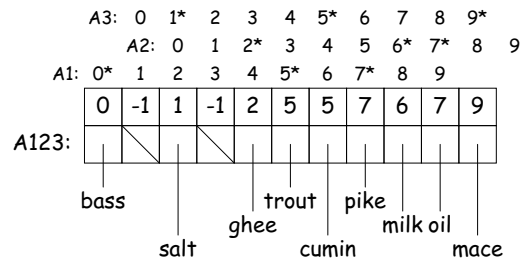
Scrunching Example

Small example: (unrelated to Tries on preceding slides)

- Three leaf arrays, each indexed 0..9



- Now overlay them, but keep track of original index of each item:



Last modified: Mon Nov 4 11:13:43 2002

CS61B: Lecture #30 17

Probabilistic Balancing: Skip Lists

- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

