CS61B Lecture #29

Public Service Announcement: The CSUA is holding its general meeting on 7 November 5:30PM in the Wozniak lounge.

Administrative

- Test #2 on Friday, 14 November, here. Test will be 1:50 long.

Today:

- Balanced search structures (*DS(IJ)*, Chapter 9

Coming Up:

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

Balanced Search: The Problem

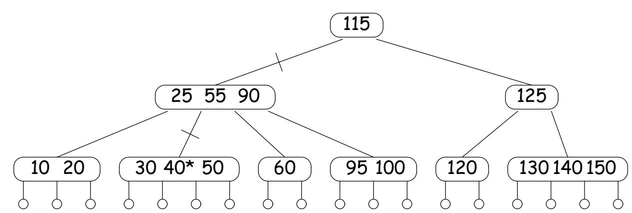
- Why are search trees important?
  - Insertion/deletion fast (on every operation, unlike hash table, which has to expand from time to time).
  - Support range queries, sorting (unlike hash tables)
- But $O(\lg N)$ performance from binary search tree requires remaining keys be divided $\approx$ by some some constant > 1 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.

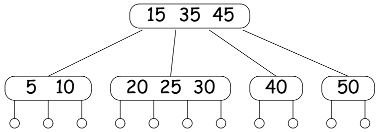
Sample Order 4 B-tree ((2,4) Tree)



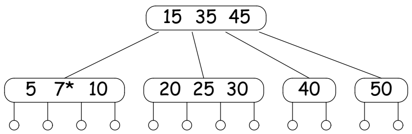
- Crossed lines show path when finding 40.
- Keys on either side of each child pointer in path bracket 40.
- Each node has at least 2 children, and all leaves (little circles) are at the bottom, so height must be $O(\lg N)$.
- In real-life B-tree, order typically much bigger
  - comparable to size of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

Inserting in B-tree (Simple Case)

- Start:

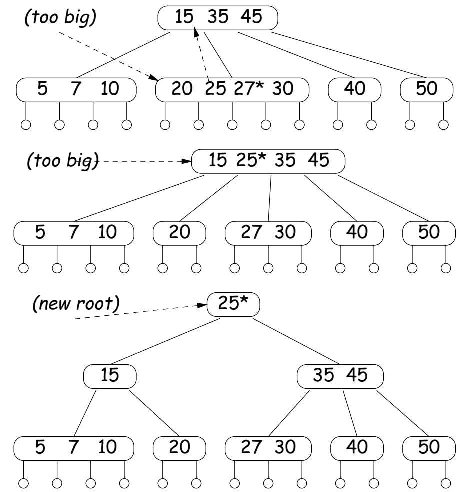


- Insert 7:



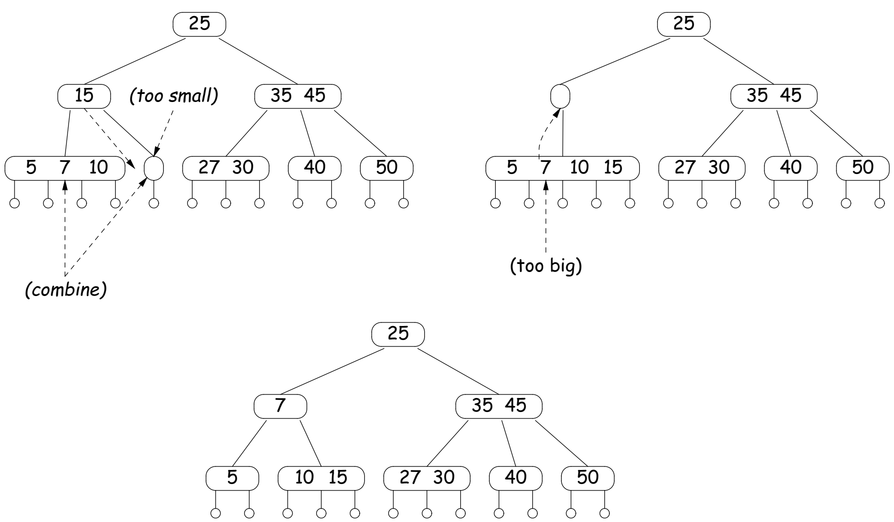
Inserting in B-Tree (Splitting)

- Insert 27:



Deleting Keys from B-tree

- Remove 20 from last tree.

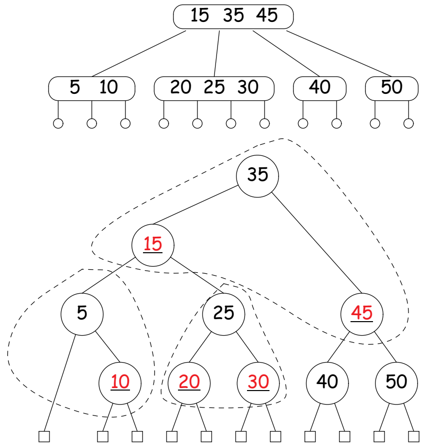


Red-Black Trees

- 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:
  1. Each node is (conceptually) colored red or black.
  2. Root is black.
  3. Every leaf node contains no data (as for B-trees) and is black.
  4. Every leaf has same number of black ancestors.
  5. Every internal node has two children.
  6. Every red node has two black children.
- Conditions 4, 5, and 6 guarantee $O(\lg N)$ searches.

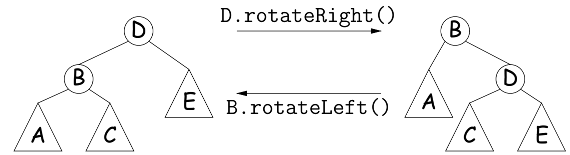
Sample Red-Black Tree

- Every red-black tree corresponds to a (2,4) tree, and the operations on one correspond to those on the other.
- Each node of (2,4) tree corresponds to a cluster of 1–3 red-black nodes in which the top node is black and any others are red.



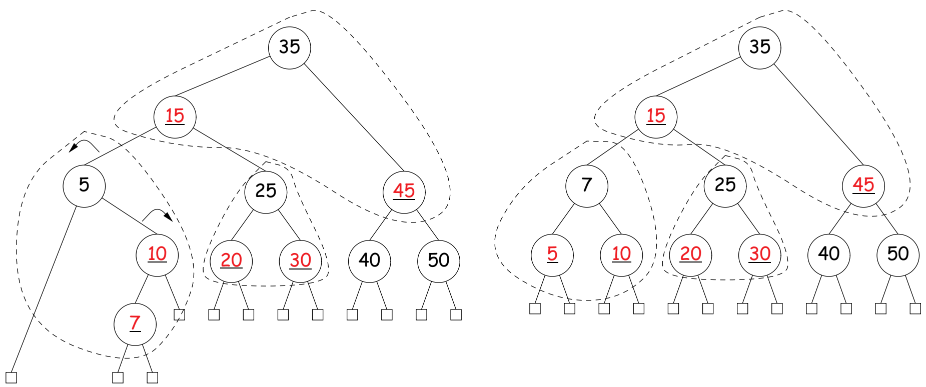
Red-Black Insertion and Rotations

- 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.



Example of Red-Black Insertion (I)

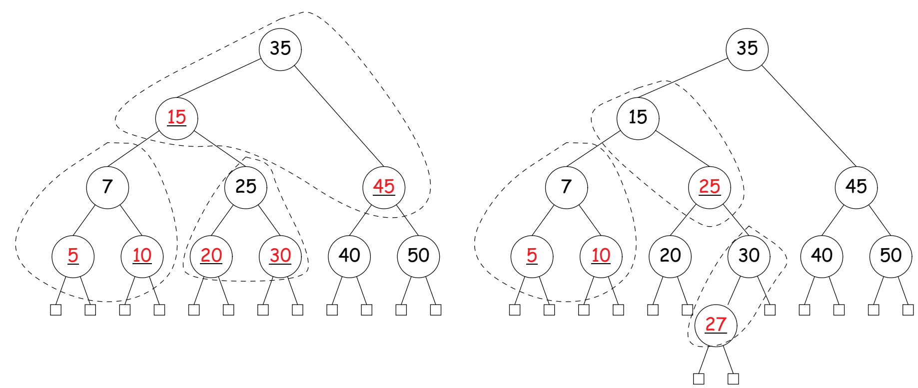
- Insert 7:



  - Here, sibling of offending node (10) is black, so rotate and recolor.
  - In corresponding (2,4) tree, new node fits in existing node.
  - (Dashed lines show groups of tree nodes that correspond to (2,4) tree nodes with > 2 children.)

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.
- In corresponding (2,4) tree, this recoloring splits nodes (adds extra black nodes). We don't have to recolor the root to red, as we did 25, because we are increasing the height of this (2,4) tree.



Really Efficient Use of Keys: the Trie

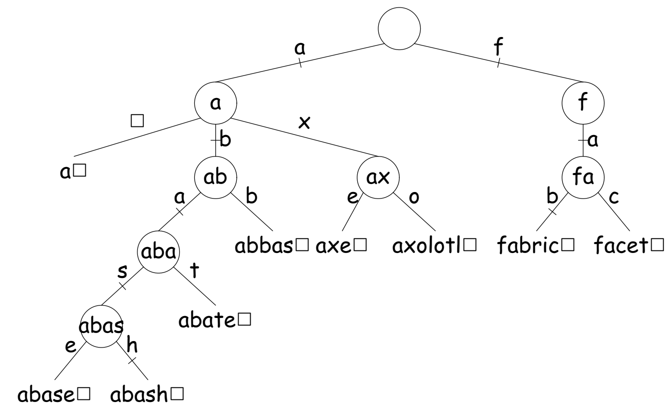
- Have been silent about cost of comparisons.
- For strings, worst case is length of string.
- Therefore should throw extra factor of key length, L, into costs:
  - $\Theta(M)$ comparisons really means $\Theta(ML)$ operations.
  - So to look for key X, keep looking at same chars of X M times.
- Can we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

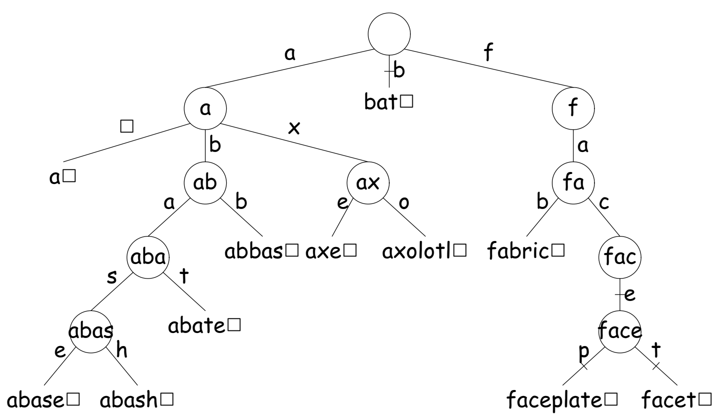
- Set of keys

  {`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.

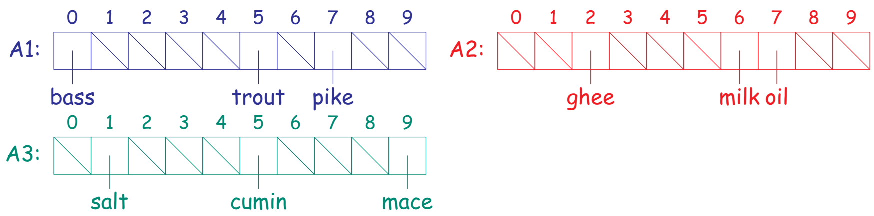
Idea: Put the arrays on top of each other!

- 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.

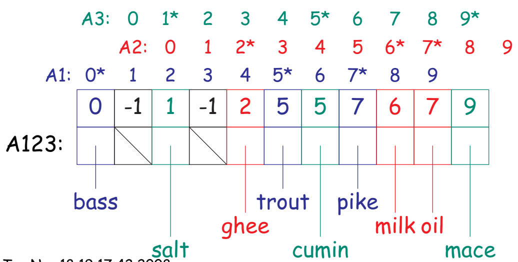
Scrunching Example

Small example: (unrelated to Tries on preceding slides)

- Three leaf arrays, each indexed 0..9

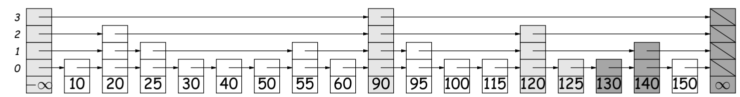


- Now overlay them, but keep track of original index of each item:



Probabilistic Balancing: Skip Lists

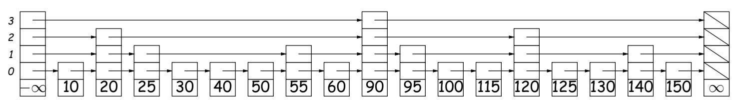
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



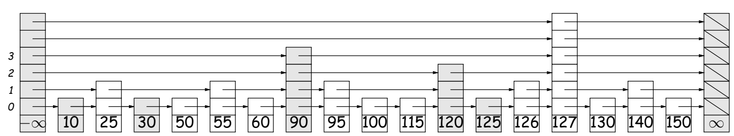
- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability*.

Example: Adding and deleting

- Starting from initial list:



- In any order, we add 126 and 127 (choosing random heights for them), and remove 20 and 40:



- Shaded nodes here have been modified.

Summary

- Balance in search trees allows us to realize $\Theta(\lg N)$ performance.
- B-trees, red-black trees:
  - Give $\Theta(\lg N)$ performance for searches, insertions, deletions.
  - B-trees good for external storage. Large nodes minimize # of I/O operations
- Tries:
  - Give $\Theta(B)$ performance for searches, insertions, and deletions, where B is length of key being processed.
  - But hard to manage space efficiently.
- *Interesting idea:* scrunched arrays share space.
- Skip lists:
  - Give probable $\Theta(\lg N)$ performace for searches, insertions, deletions
  - Easy to implement.
  - Presented for *interesting ideas:* probabilistic balance, randomized data structures.