COS 226 Lecture 8: Balanced trees

Symbol Table, Dictionary

- records with keys
- INSERT
- SEARCH

Goal: Symbol table implementation

- with O(lgN) GUARANTEED performance
- for both search and insert
- (and other ST operations)

Approach 1:

- worst-case guarantee

Approach 2:

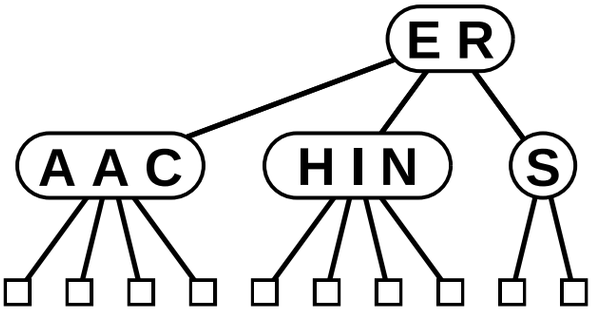
- probabilistic "guarantee"

2-3-4 trees

Allow one, two, or three keys per node

Keep link for every interval beteen keys

- 2-node: one key, two children
- 3-node: two keys, three children
- 4-node: three keys, four children



SEARCH

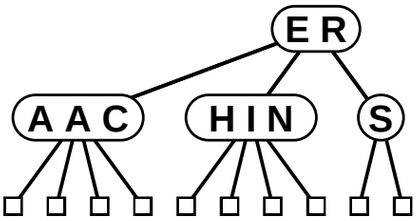
- compare search key against keys in node
- find interval containing search key
- follow associated link (recursively)

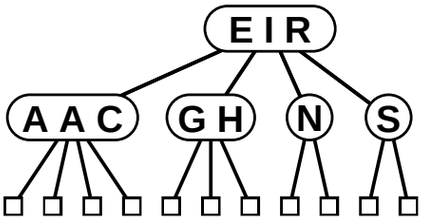
INSERT

- search to bottom for key
- 2-node at bottom: convert to a 3-node
- 3-node at bottom: convert to a 4-node
- 4-node at bottom: ??

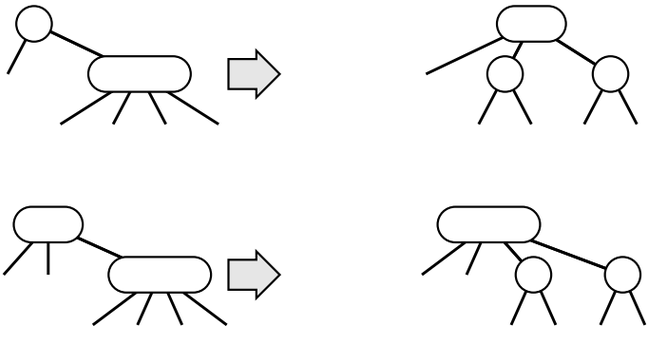
Top-down 2-3-4 trees

Transform tree on the way down
to ensure that last node is not a 4-node



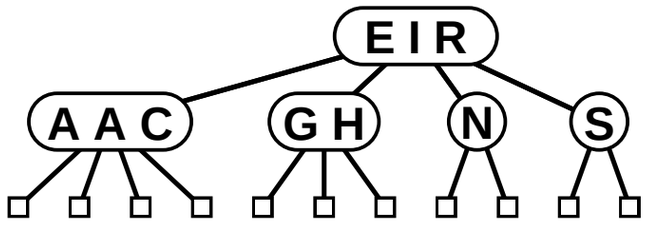


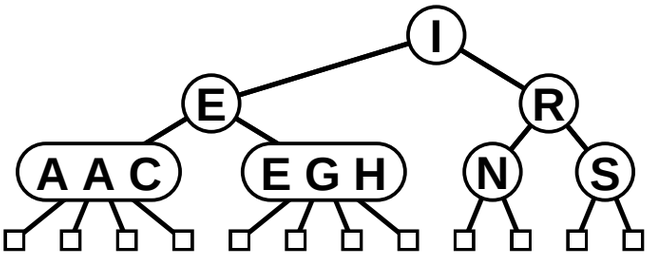
"Split" any 4-nodes encountered using
one of two transformations

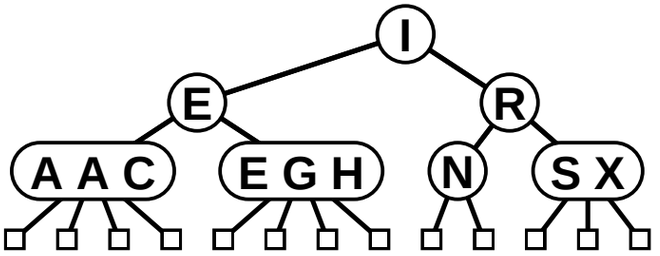


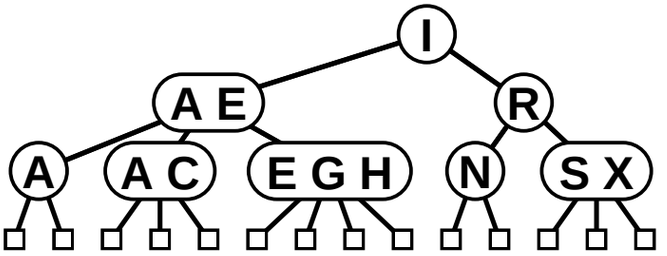
Invariant: "current" node is not a 4-node
Insertion at bottom therefore easy.

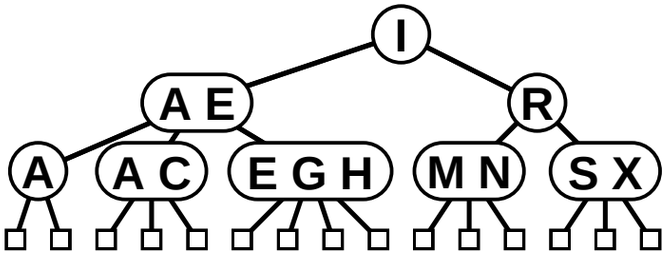
Top-down 2-3-4 tree construction

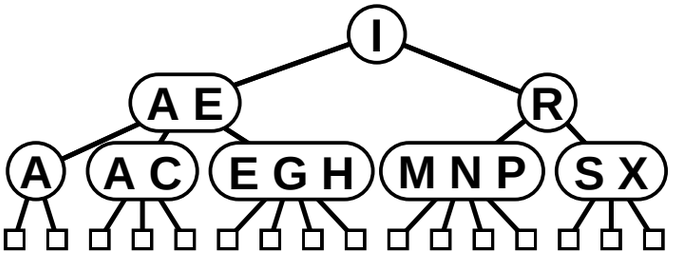


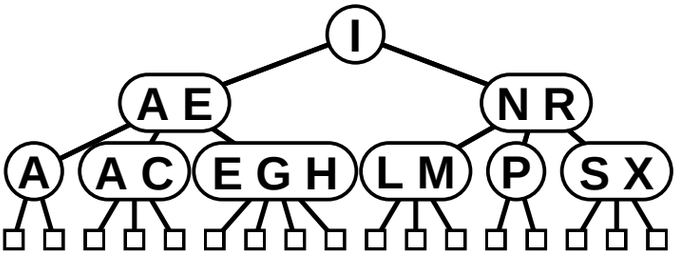


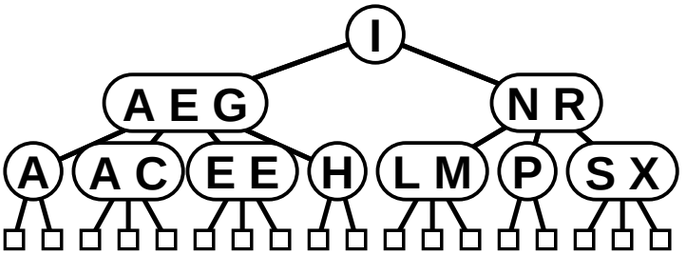












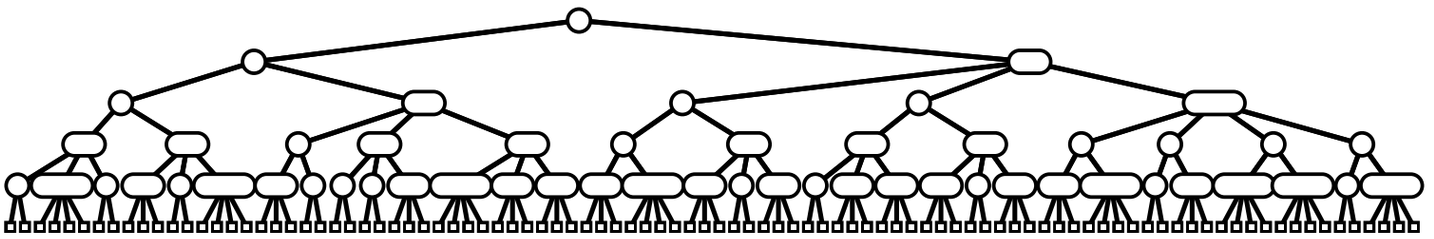
Trees grow up from the bottom

Balance in 2-3-4 trees

In top-down 2-3-4 trees,

all paths from top to bottom

are equal length (tree height)



Tree height:

worst case: lgN (all 2-nodes)

best case: lgN/2 (all 4-nodes)

between 10 and 20 for a million nodes

between 15 and 30 for a billion nodes

Comparisons within nodes not accounted for

Top-down 2-3-4 tree implementation

Fantasy code (sketch)

```
ST BSTinsert(ST head, recordType rec)
   { keyType v = key(rec);
     struct BSTnode *x = head->r;
     while (x != z)
        {
          x = therightlink(x, v);
          if fourNode(x) then split(x);
        }
     if twoNode(x) then makeThree(x, v); else
     if threeNode(x) then makeFour(x, v); else
     return head;
   }
```

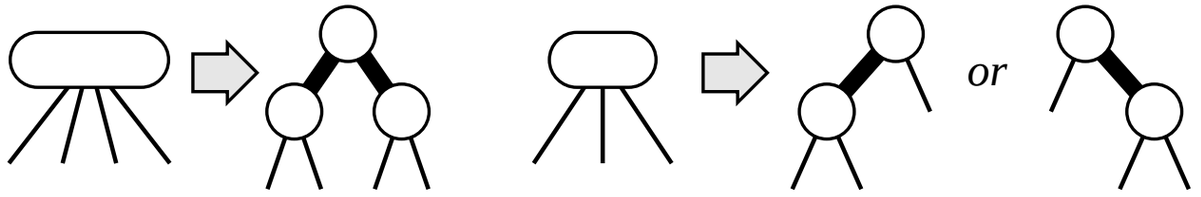
Direct implementation complicated because of

* "therightlink(x, v)"
* maintaining multiple node types
* large number of cases for "split"

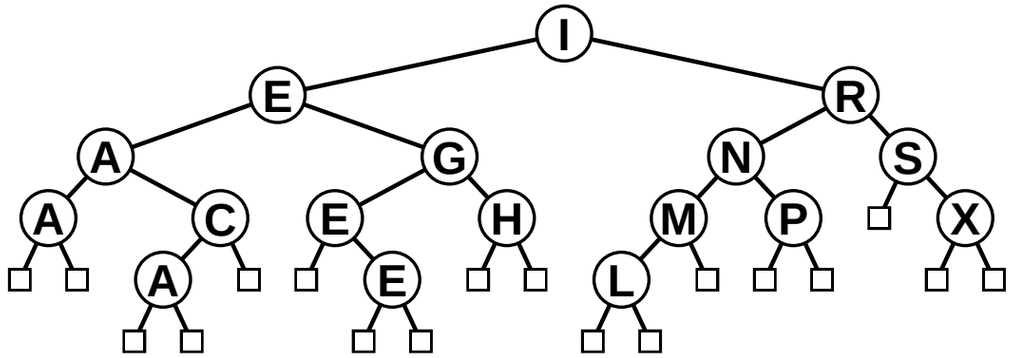
Search also more complicated than for BST

Red-black trees

Represent 2-3-4 trees as binary trees
with "internal" edges for 3- and 4-nodes



Correspondence between 2-3-4 and RB trees



Not 1-1 because 3-nodes swing either way

SEARCH:

use plain BST search (!)

tree balance gives

O(lgN) performance guarantee

all comparisons accounted for

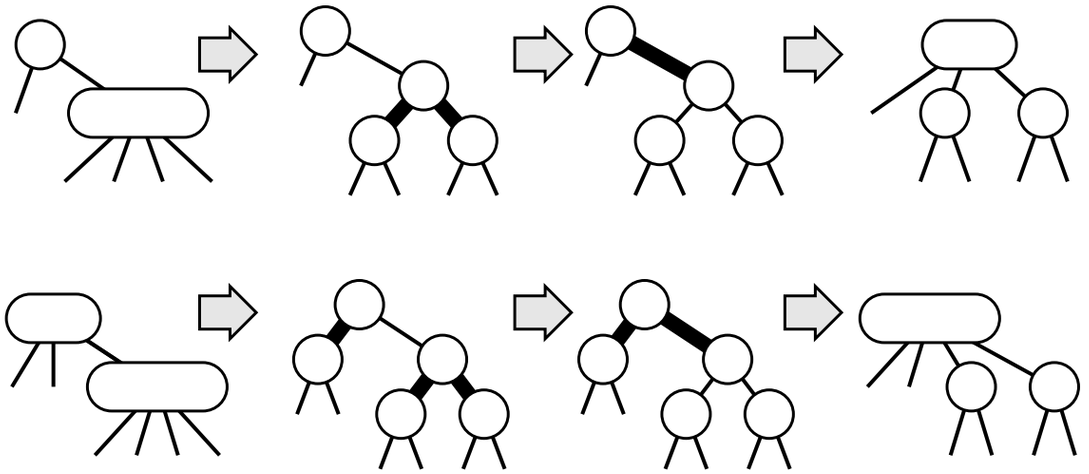
INSERT:

split reduces to fewer cases

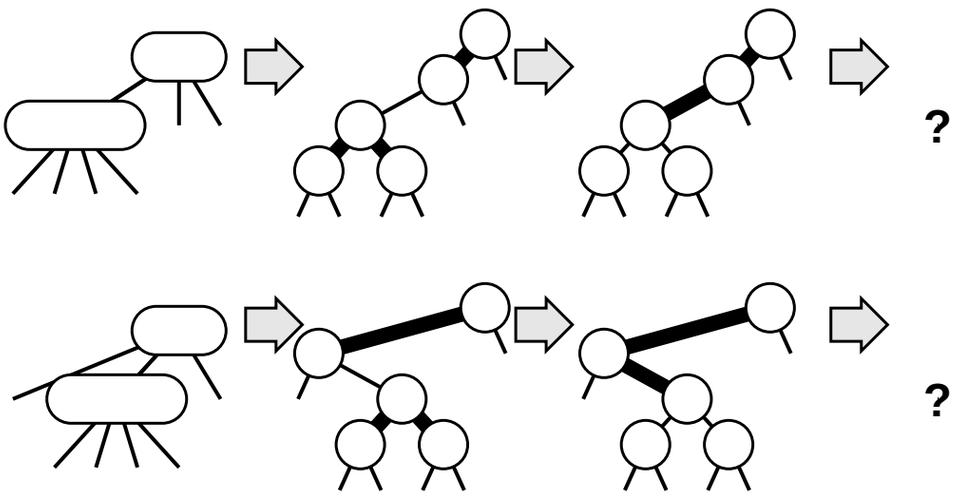
Splitting nodes in red-black trees

Two cases are easy

(need only to switch colors)



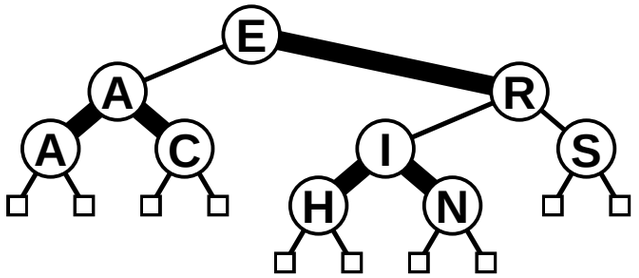
Two cases require "rotations"

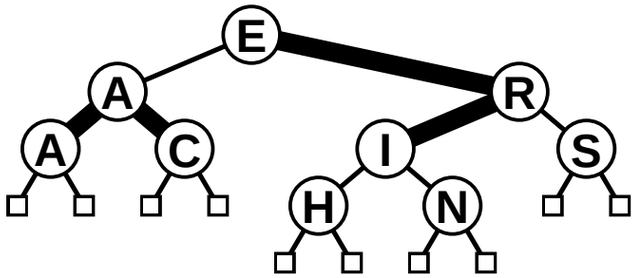


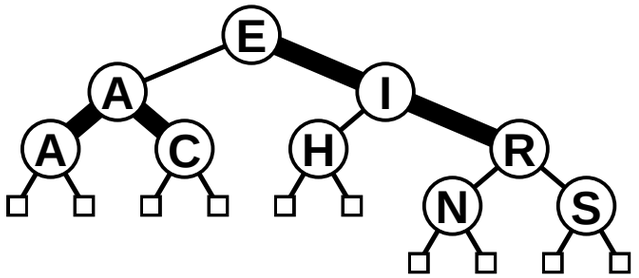
Can use red-black abstraction directly

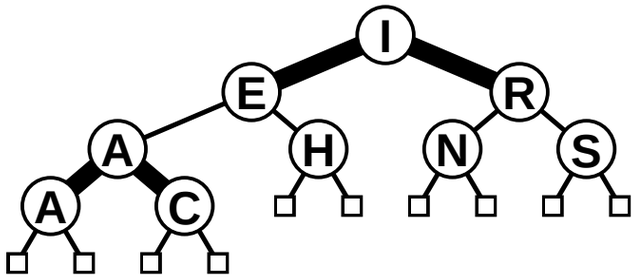
Invariant: never two consecutive red nodes on any path.

RB tree node split example



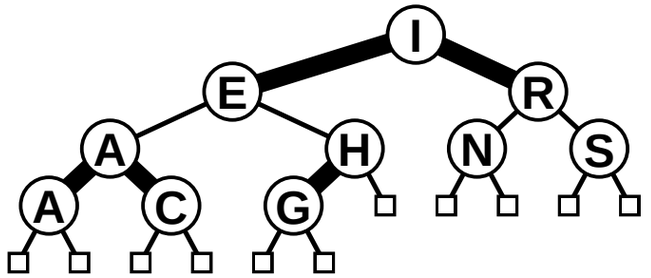


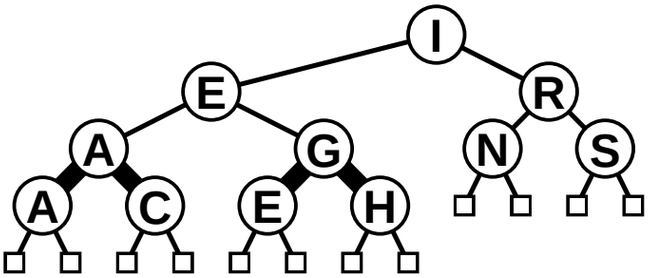


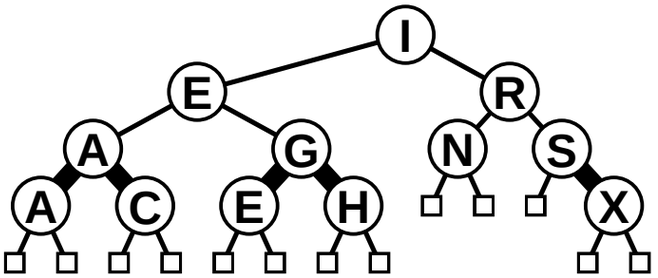


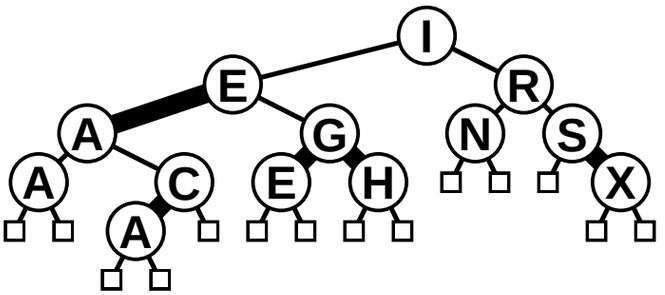
Code: tricky but manageable (see book)

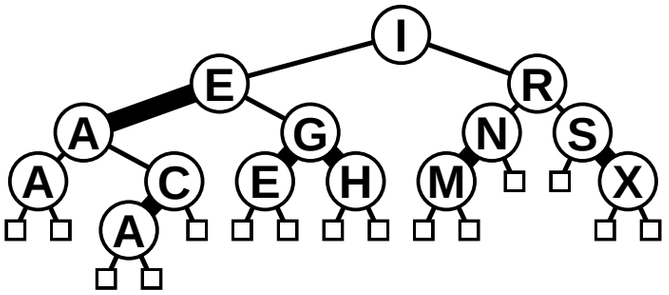
Red-black tree construction

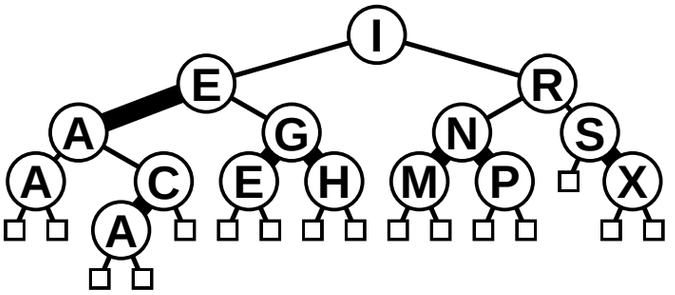


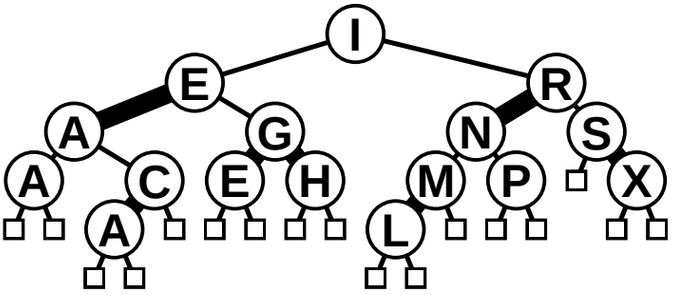


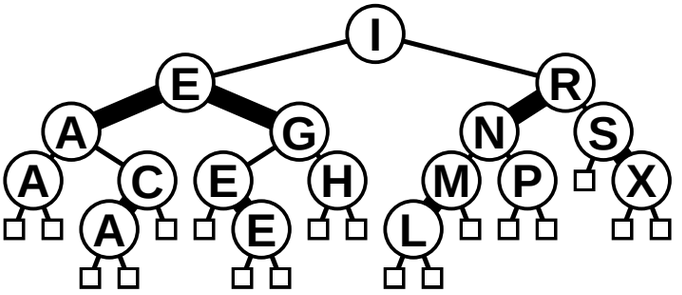








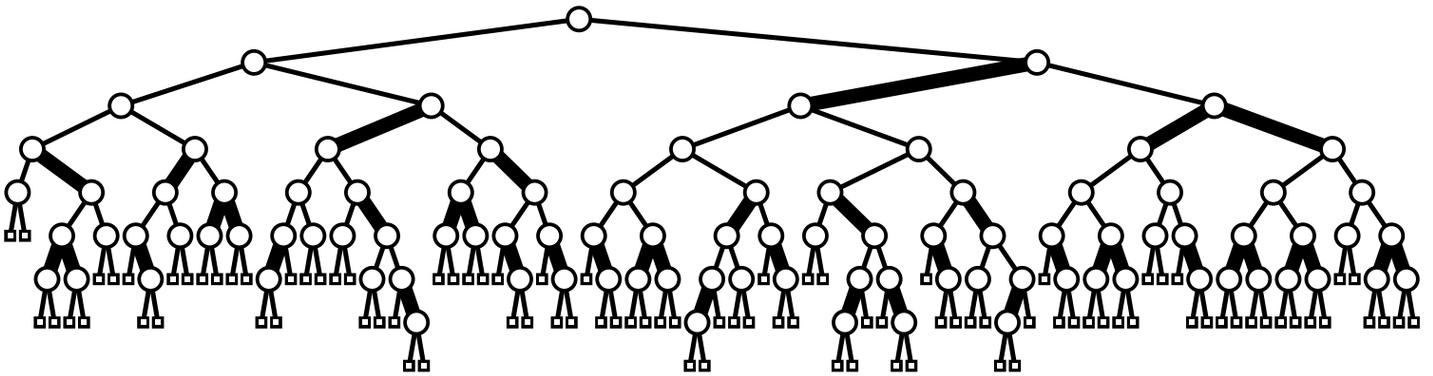




Balance in red-black trees

In red-black trees,

the longest path is at most twice
the length of the shortest path



worst case: less than 2lgN

Comparisons within nodes are accounted for

B-trees

Generalize 2-3-4 trees: up to M links per node
Split full nodes on the way down

Used for external search
    node size = page size
    typical: M=1000
        search and insert on 10^9 records in
        three or four page acceses

Key point: flexibility to do fast insert

Red-black abstraction still works
    BUT
might not use internal links

Space time tradeoff

M large: only a few levels in tree
M small: less wasted space

Randomized BSTs

Invented in 1995 by S. Roura

Idea: new node should be at root with
probability 1/(N+1), so put it there!

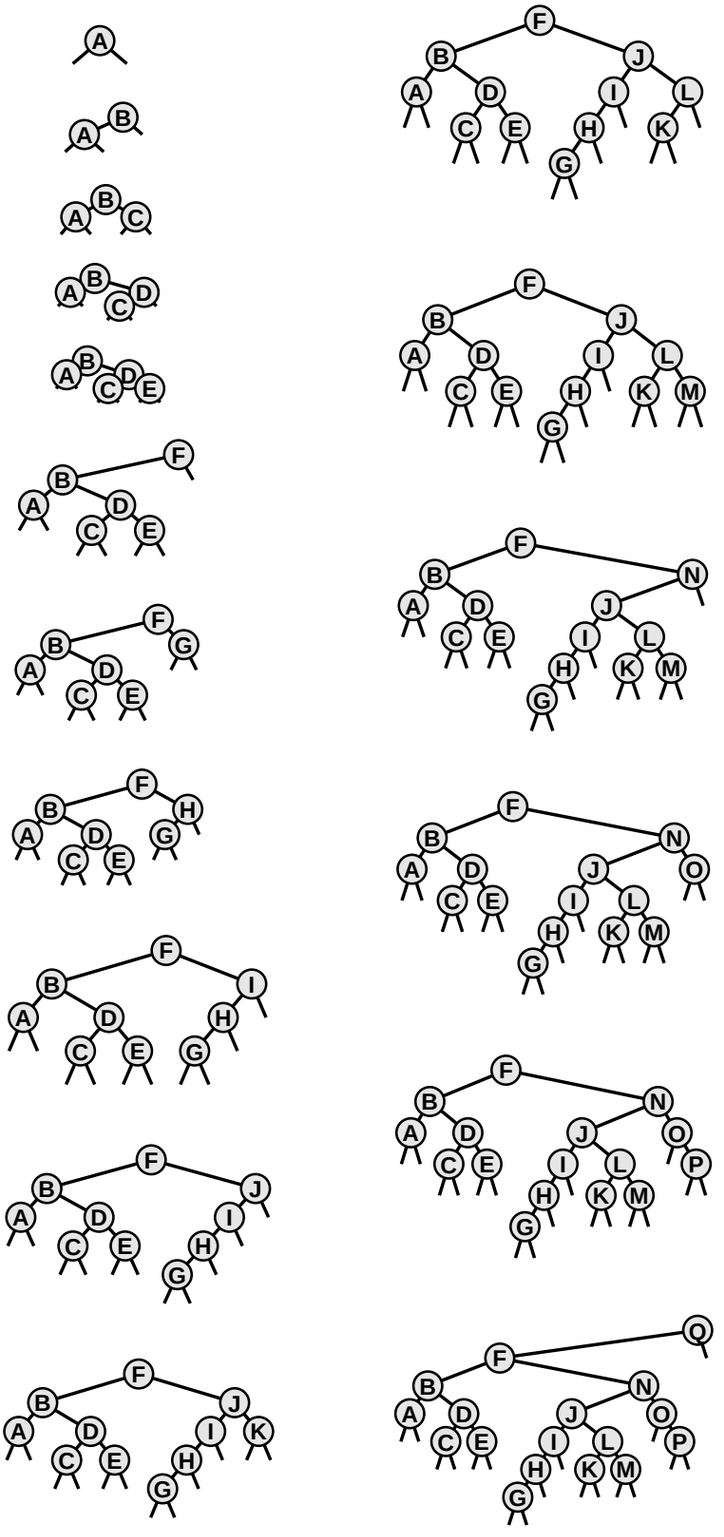
```
ST BSTinsert(ST head, recordType rec)
  { keyType v = key(rec);
    if (head == z) return head = NEW(rec, z, z, 1);
    if (drand48()< 1.0/(head->N+1))
      return rootBSTinsert(head, rec);
    if (less(v, key(head->rec)))
      head->l = BSTinsert(head->l, rec);
    else
      head->r = BSTinsert(head->r, rec);
    (head->N)++;
    return head;
  }
```

Trees have same shape as random BSTs
no matter what the input

Random BSTs are well studied
exponentially small chance of bad balance

Randomized BST example

Insert keys in order: tree shape random!



Other operations in randomized BSTs

Find kth largest

another use of size field already there

Join disjoint STs

straightforward recursive implementation

to join STs A (of size M) and B (of size N)

use A root with probability M/(M+N)

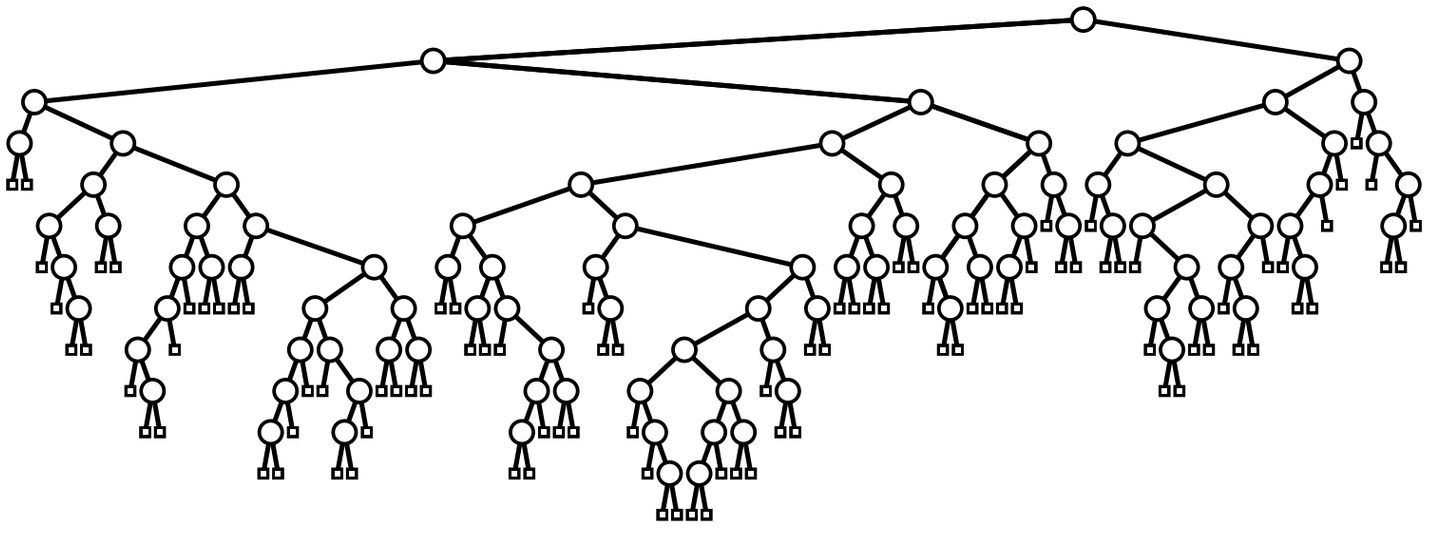
use B root with probability N/(M+N)

join other tree with subtree recursively

Delete

remove the node, do join (above)

THM: Trees still random after delete (!!)



Bottom line:

randomized BSTs provide full symbol table ADT

straightforward implementation

O(log N) average case

bad cases provably unlikely

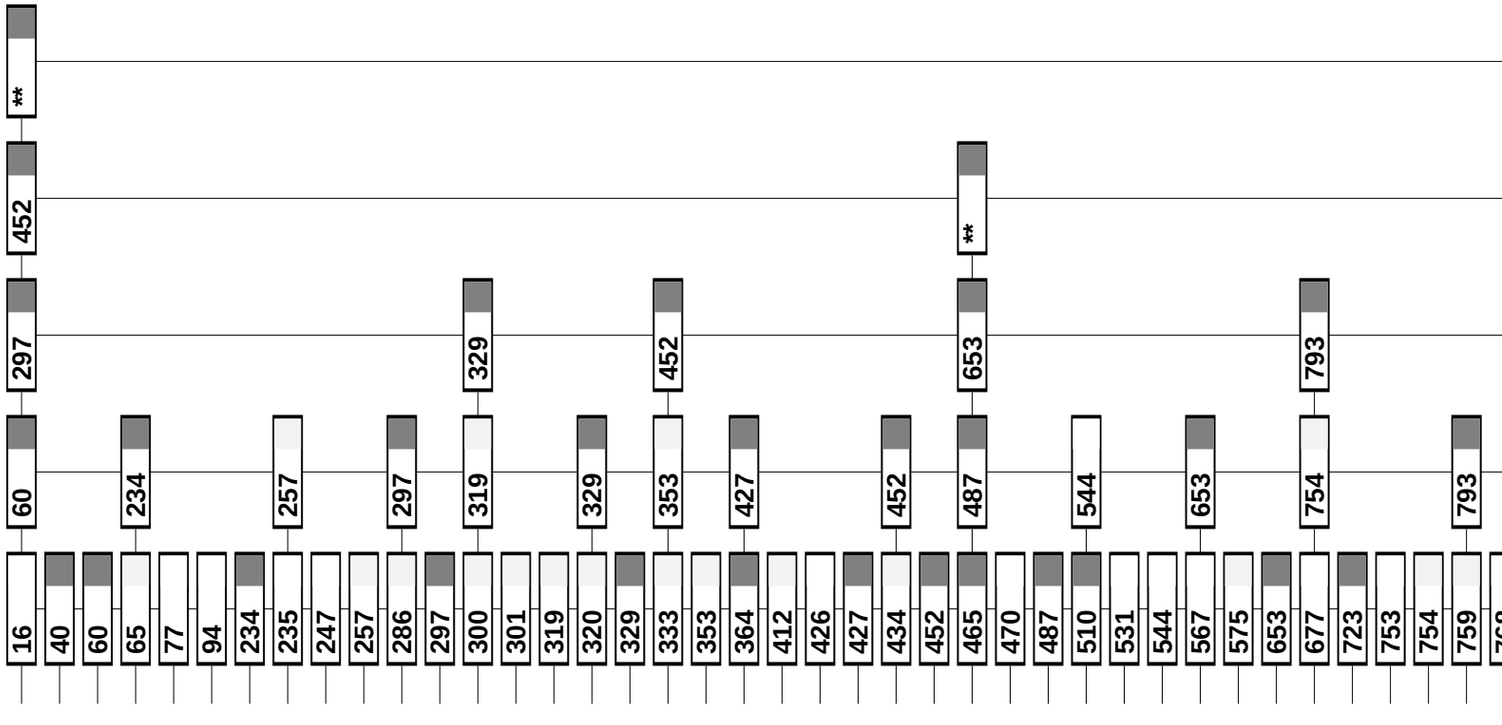
Recursive BST root insertion

Still tricky; still not much code

```
ST rootBSTinsert(ST head, recordType rec)
   { keyType v = key(rec);
     struct randBSTnode *x = NEW(rec, z, z, 1);
     if (head == z) return x;
     if (less(v, key(head->rec)))
     { x->r = head; x->l = BSTstitch(head, v); }
     else
        /* symmetric case */
     fixN(x); /* update count field */
     return x;
   }
struct randBSTnode *BSTstitch(ST head, keyType v)
   { struct randBSTnode *x;
     if (head == z) return head;
     if (less(v, key(head->rec)))
     { if ((x = head->l) == z) return x;
        if (less(v, key(x->rec)))
           return BSTstitch(x, v);
        head->l = BSTstitch(x, v); }
     else
        /* symmetric case */
     fixN(head);
     return x;
   }
```

Other randomized trees

- treaps
- skip lists



Other balanced tree algorithms

- skip lists (!!)
- AVL trees
- 2-3 trees
- ...

Can represent as red-black trees

Are balanced trees optimal?

- worst-case: no (can get ClgN for C>1)
- average-case: open