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)

Three approaches

1. PROBABILISTIC "guarantee"
2. AMORTIZED "guarantee"
3. WORST-CASE GUARANTEE

Randomized BSTs

IDEA: new node should be root with probability 1/(N+1)

DO IT!

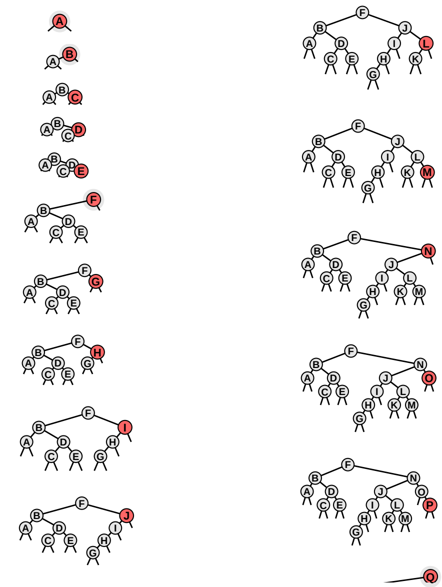
```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if (rand() < RAND_MAX/(h->N+1))
      return insertT(h, item);
    if less(v, t) h->l = insertR(h->l, item);
               else h->r = insertR(h->r, item);
    (h->N)++; return h;
  }
void STinsert(Item item)
  { head = insertR(head, item); }
```

Trees have same shape as random BSTs FOR ALL INPUTS

Random BSTs: exponentially small chance of bad balance

Randomized BST example

Insert keys in order: tree shape still 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

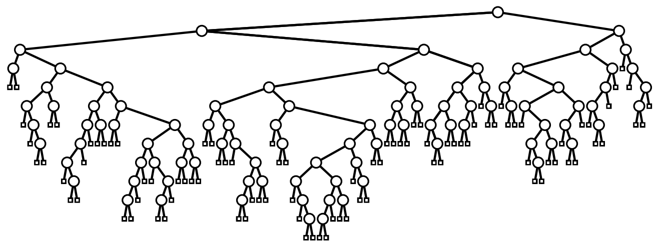
DELETE

- remove the node, do join (above)

THM: Trees still random after delete (!!)

Randomized BSTs

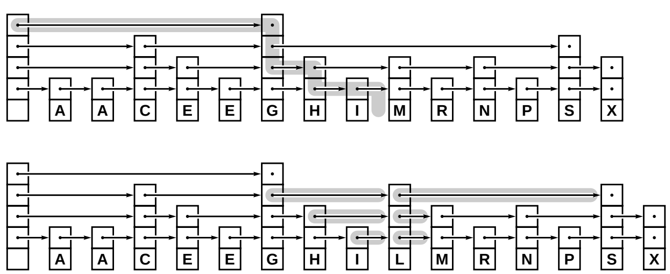
Always look like random BSTs



- implementation straightforward
- support all symbol-table ADT ops
- O(log N) average case
- bad cases provably unlikely

Skip lists

Idea: Add links to linked-list nodes to make ``fast tracks``



Challenges (see Section 13.5 for details):

- how to maintain structure under insertion
- how many links in a particular node?

Bottom line: similar to randomized BSTs

- plus: easier to understand
- minus: more pointer-chasing

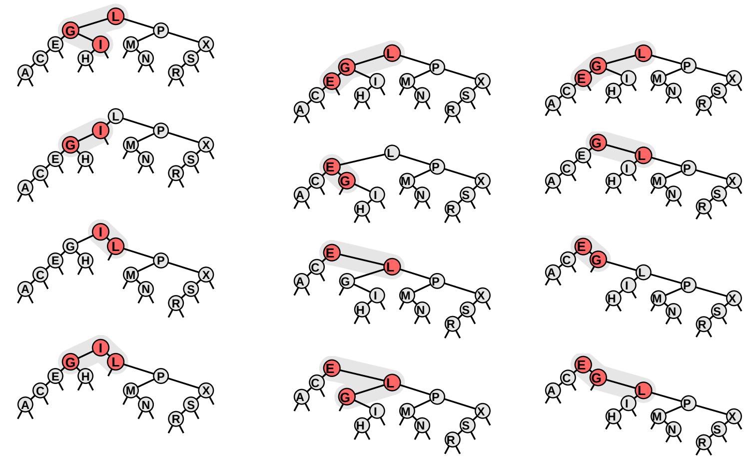
Splay trees

Idea: slight modification to root insertion

Check two links above current node

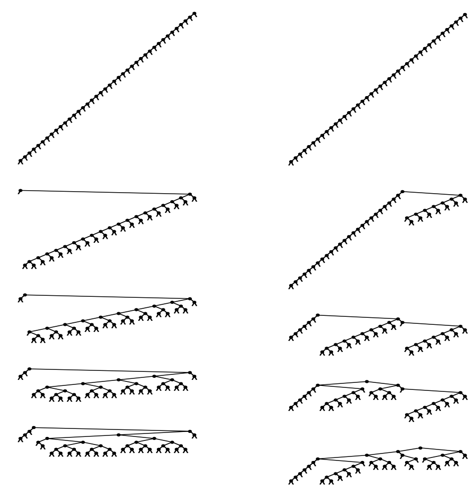
Orientations differ: same as root insertion

Orientations match: do top rotation first



Splay tree balance

THM: Splay rotations halve the search path



guaranteed performance over SEQUENCE of operations

Splay tree implementation

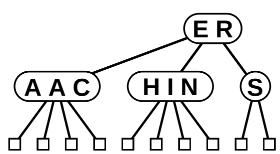
```
link splay(link h, Item item)
{ Key v = key(item);
  if (h == z) return NEW(item, z, z, 1);
  if (less(v, key(h->item)))
  {
    if (hl == z) return NEW(item, z, h, h->N+1);
    if (less(v, key(hl->item)))
      { hll = splay(hll, item); h = rotR(h); }
    else
      { hlr = splay(hlr, item); hl = rotL(hl);}
    return rotR(h);
  }
  else
  {
    if (hr == z) return NEW(item, h, z, h->N+1);
    if (less(key(hr->item), v))
      { hrr = splay(hrr, item); h = rotL(h); }
    else
      { hrl = splay(hrl, item); hr = rotR(hr);}
    return rotL(h);
  }
}
```


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

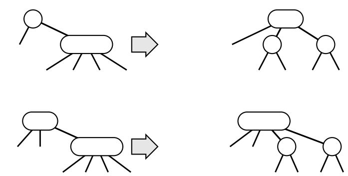
- search to bottom for key
- 2-node at bottom: convert to a 3-node
- 3-node at bottom: convert to a 4-node
- 4-node at bottom: ??

Top-down 2-3-4 trees

Transform tree on the way DOWN

- to ensure that last node is not a 4-node

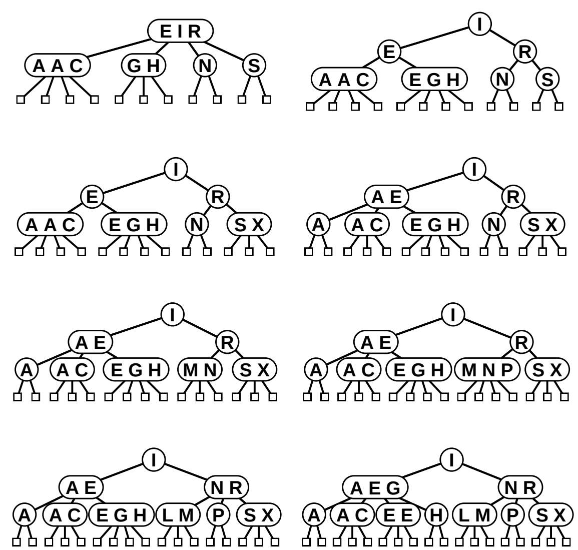
Local transformations to split 4-nodes:



Invariant: "current" node is not a 4-node

- One of two local transformations must apply at next node
- Insertion at bottom is easy (not into a 4-node)

Top-down 2-3-4 tree construction

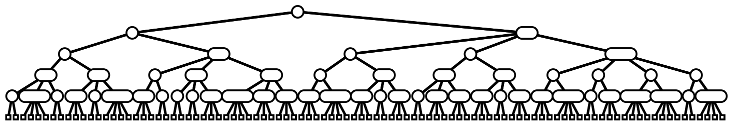


Trees grow up from the bottom

Balance in 2-3-4 trees

In top-down 2-3-4 trees,

- all paths from top to bottom are the same length



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):

```
link insertR(link h, Item item)
  { Key v = key(item);
    link x = h;
    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

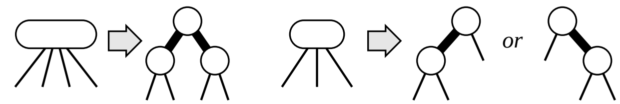
- "therightlink(x, v)"
- maintaining multiple node types
- large number of cases for "split"

Search also more complicated than for BST

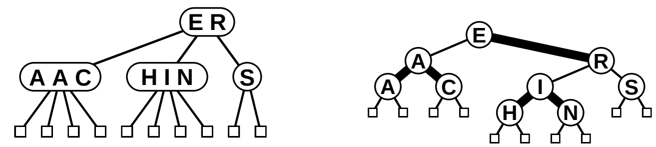
Red-black trees

Represent 2-3-4 trees as binary trees

- with "internal" edges for 3- and 4-nodes



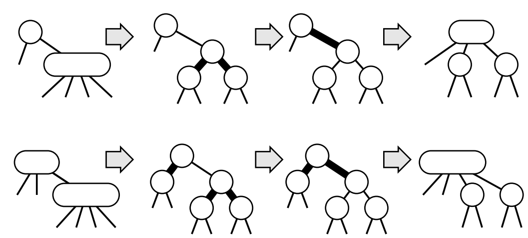
Correspondence between 2-3-4 and RB trees



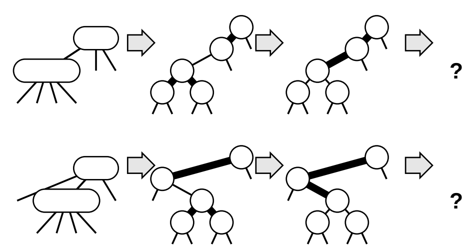
Not 1-1 because 3-nodes swing either way

Splitting nodes in red-black trees

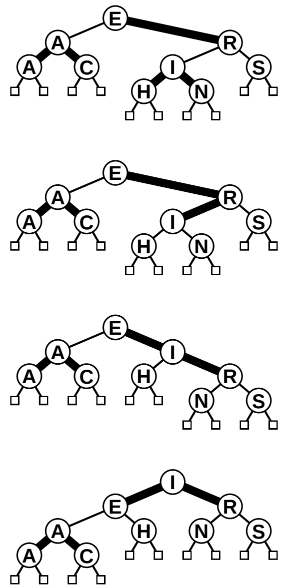
Two cases are easy (need only to switch colors)



Two cases require ROTATIONS



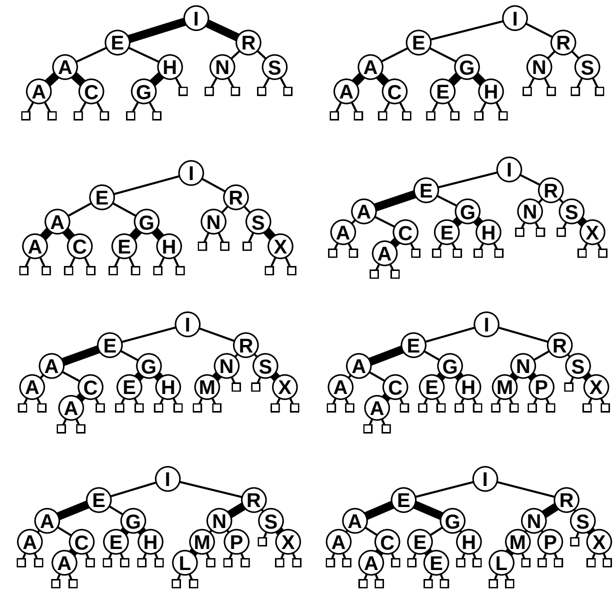
RB tree node split example



Red-black tree implementation

```
link RBinsert(link h, Item item, int sw)
{ Key v = key(item);
  if (h == z) return NEW(item, z, z, 1, 1);
  if ((hl->red) && (hr->red))
    { h->red = 1; hl->red = 0; hr->red = 0; }
  if (less(v, key(h->item)))
    {
      hl = RBinsert(hl, item, 0);
      if (h->red && hl->red && sw) h = rotR(h);
      if (hl->red && hll->red)
      { h = rotR(h); h->red = 0; hr->red = 1; }
    }
  else
    {
      hr = RBinsert(hr, item, 1);
      if (h->red && hr->red && !sw) h = rotL(h);
      if (hr->red && hrr->red)
      { h = rotL(h); h->red = 0; hl->red = 1; }
    }
  return h;
}
void STinsert(Item item)
  { head=RBinsert(head,item,0); head->red=0; }
```

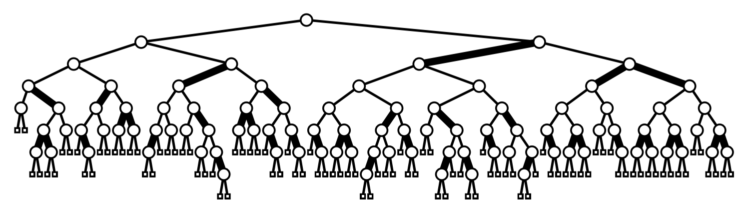

Red-black tree construction



Balance in red-black trees

In red-black trees,

- LONGEST path at most twice as long as SHORTEST path



worst case: less than 2lgN

Comparisons within nodes *are* counted

B-trees

Generalize 2-3-4 trees: up to M links per node

Split full nodes on the way down

Red-black abstraction still works

- BUT might use binary search instead of internal links

B-trees for external search

- node size = page size
- typical: M = 1000, N < 1,000,000,000,000

Main advantage: flexibility to do fast insert/delete

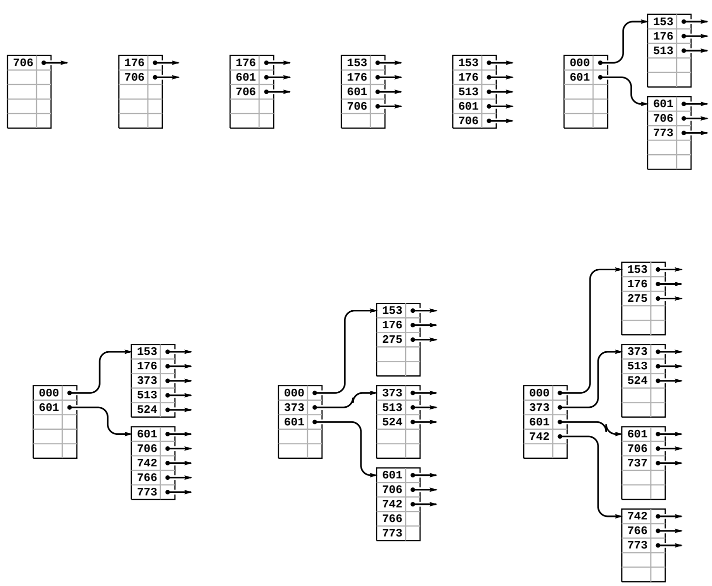
Space-time tradeoff

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

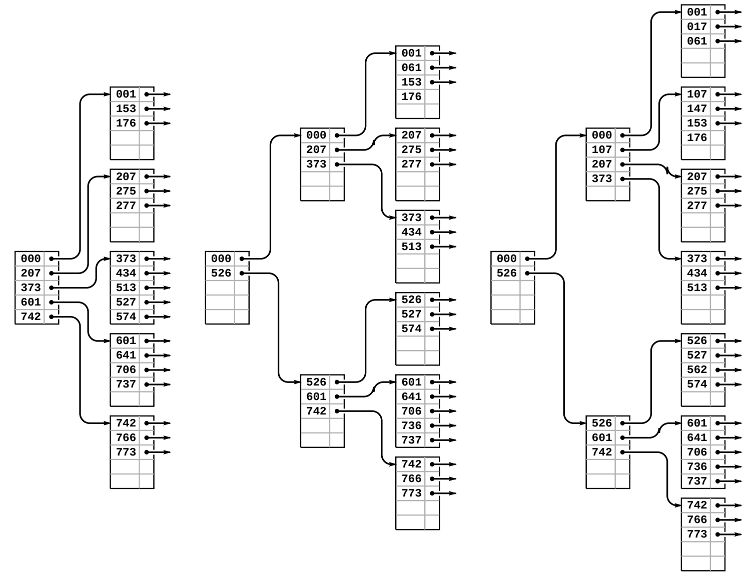
Bottom line:

- log_M N page accesses (3 or 4 in practice)

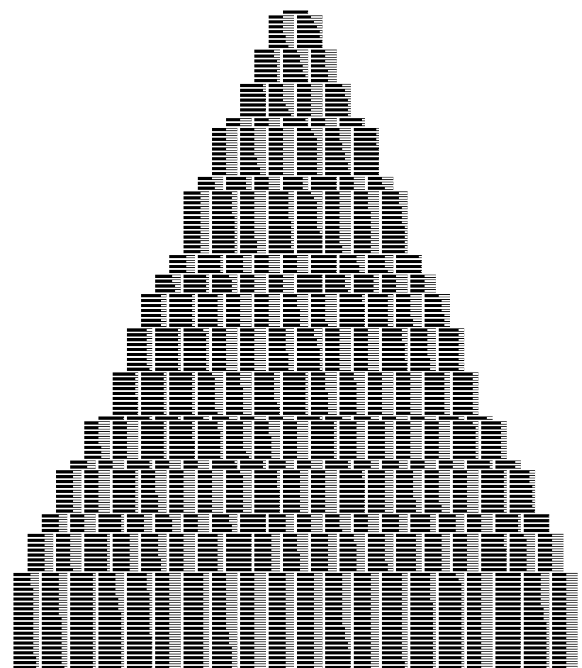
B tree example



B tree example (continued)



B tree growth



Summary

GOAL: ST implementation with O(lgN) GUARANTEE for all ops

probabilistic guarantee: random BSTs, skip lists

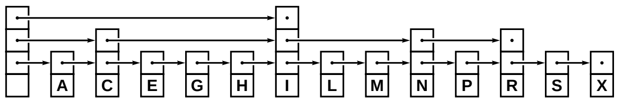
amortized guarantee: splay trees

optimal guarantee: red-black trees

Algorithms are varations on a theme (rotations when inserting)

Different abstractions, but equivalent

Ex: skip-list representation of 2-3-4 tree



Are balanced trees OPTIMAL?

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

Abstraction extends to give search algs for huge files

- B-trees