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:

- probabilistic "guarantee"

Approach 2:

- amortized "guarantee"

Approach 3:

- worst-case guarantee

Randomized BSTs

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

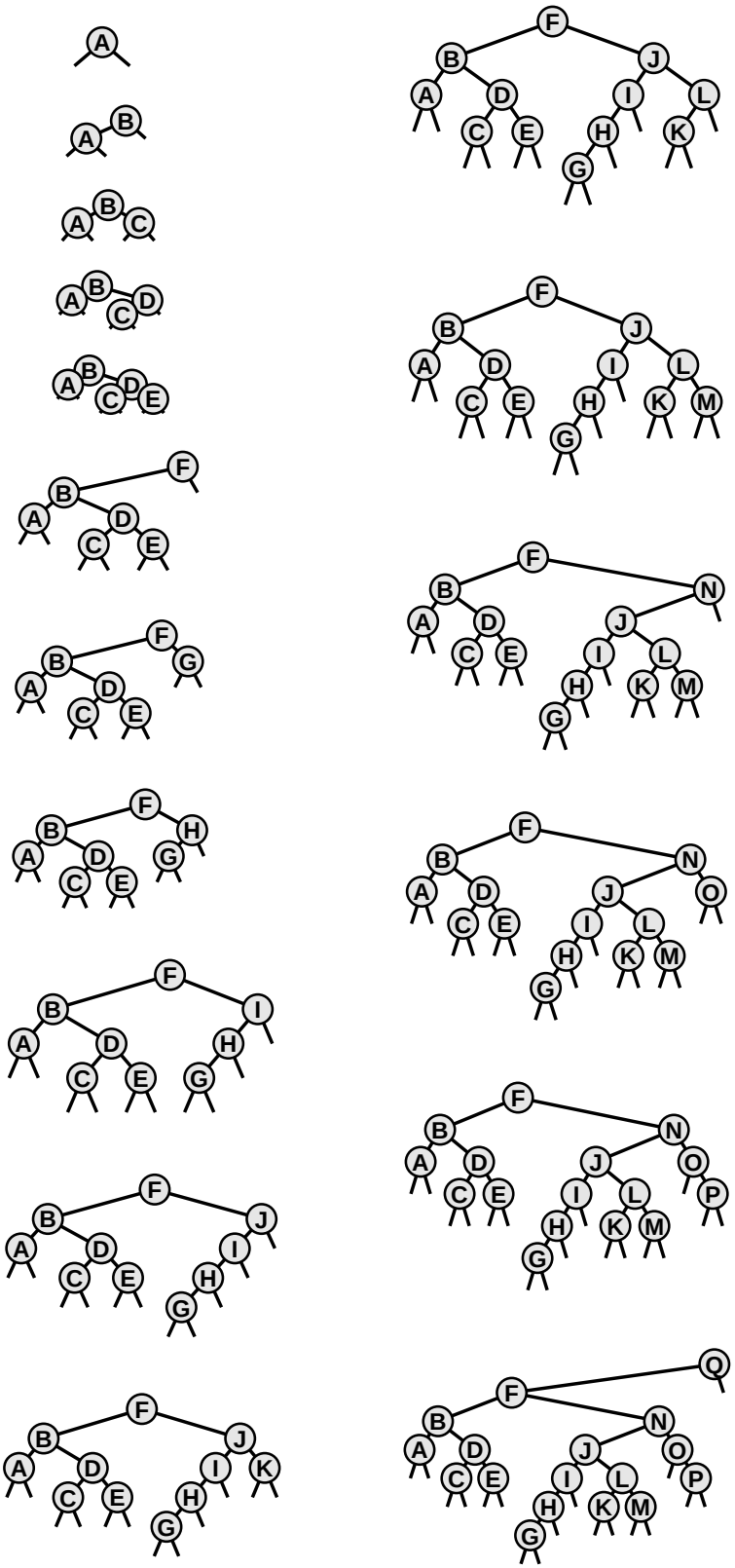
```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if (rand()< RAND_MAX/(h->N+1))
      return insertT(h, item);
    if less(v, t) h->l = insertR(h->l, item);
                  else h->r = insertR(h->r, item);
    (h->N)++; return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

Trees have same shape as random BSTs
** 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)

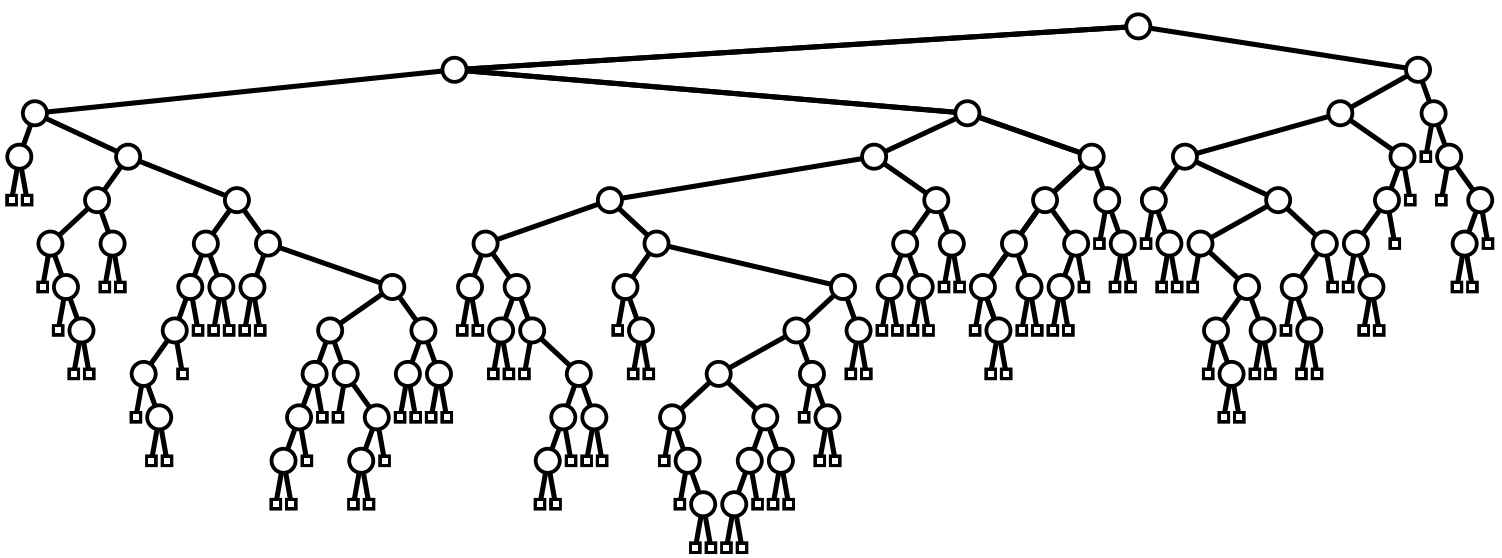
use B root with probability N/(M+N)

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

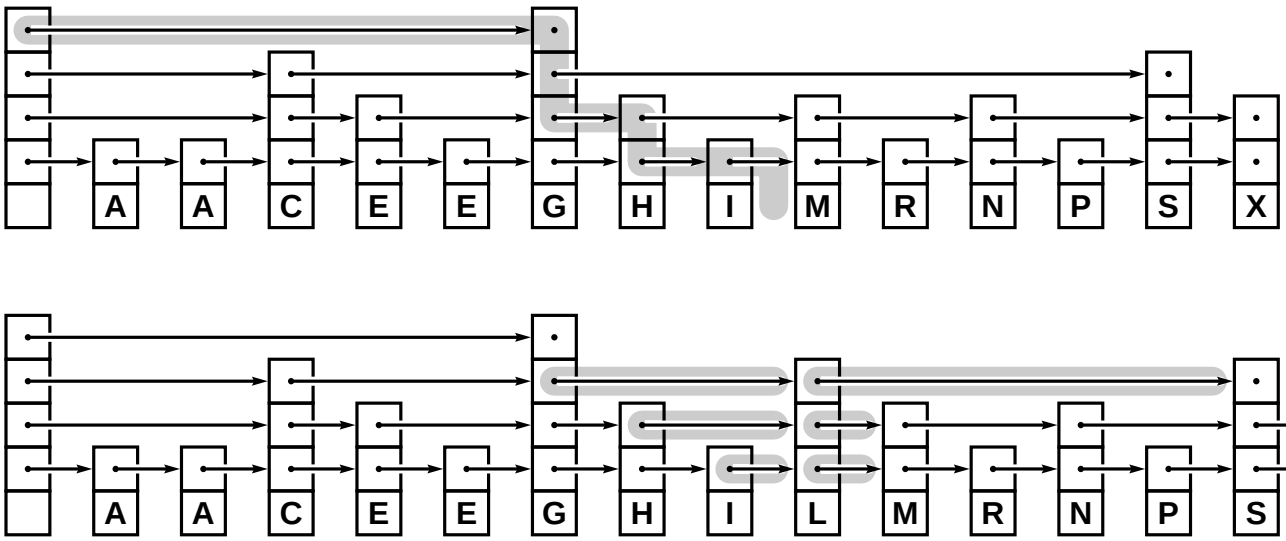
straightforward implementation

O(log N) average case

bad cases provably unlikely

Skip lists

Idea: Add fast tracks to linked lists



Nodes have variable number of links

```
typedef struct STnode* link;
struct STnode { Item item; link* next; int sz; };
link NEW(Item item, int k)
  { int i; link x = malloc(sizeof *x);
    x->next = malloc(k*sizeof(link));
    x->item = item; x->sz = k;
    for (i = 0; i < k; i++) x->next[i] = z;
    return x;
  }
```

Problems:

how to maintain structure under insertion

how many links in a particular node?

Skip list implementation

Idea: put a RANDOM number of links in each node

j links with probability 1/2^j

```
void STinit(int max)
  {
    N = 0; lgN = 0;
    z = NEW(NULLitem, 0);
    head = NEW(NULLitem, lgNmax);
  }
void insertR(link t, link x, int k)
  { Key v = key(x->item);
    if (less(v, key(t->next[k]->item)))
      {
        if (k < x->sz)
          { x->next[k] = t->next[k];
            t->next[k] = x; }
        if (k == 0) return;
        insertR(t, x, k-1); return;
      }
    insertR(t->next[k], x, k);
  }
void STinsert(Key v)
  { insertR(head, NEW(v, randX()), lgN); N++; }
```

Bottom line:

same properties as randomized BSTs

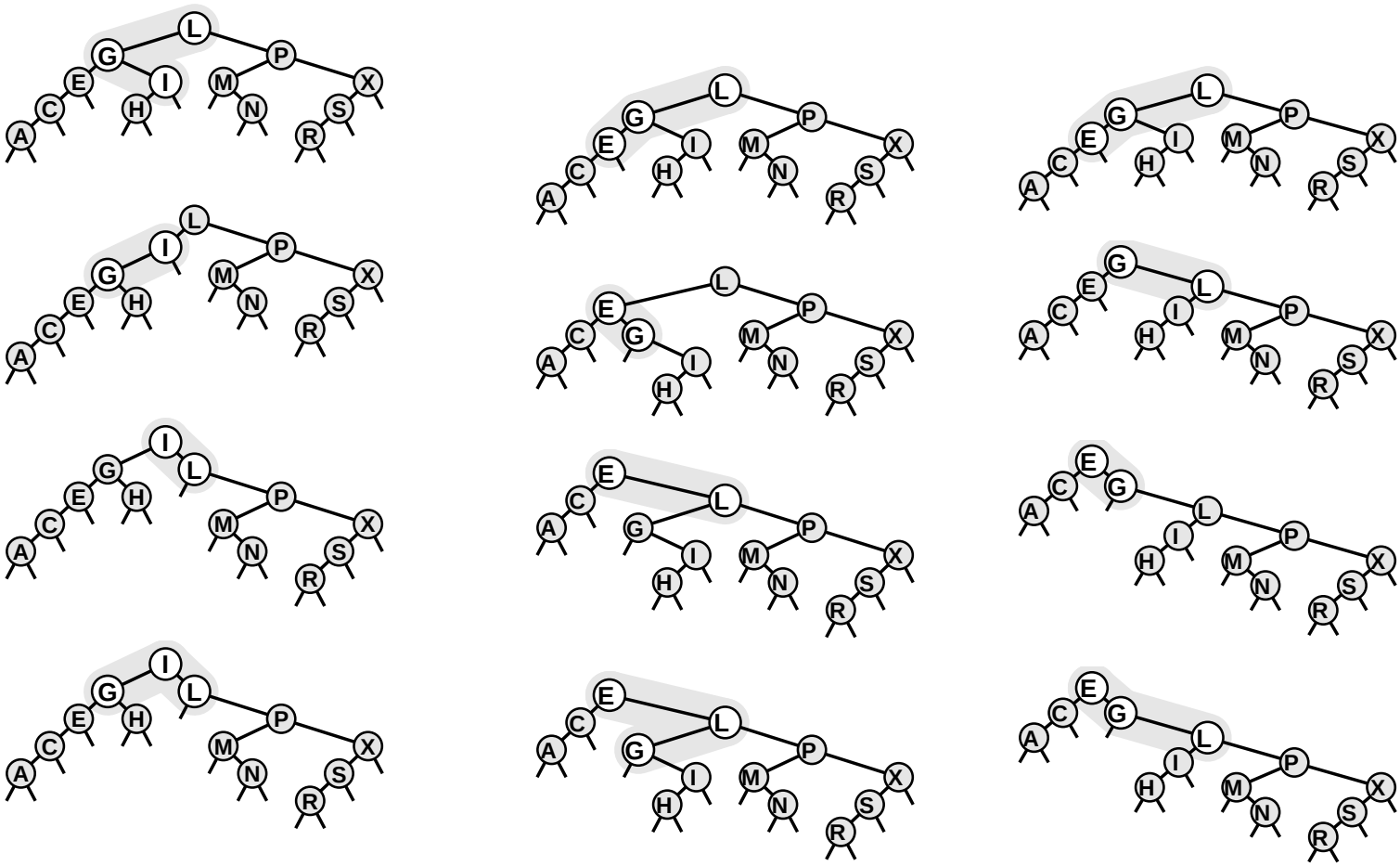
plus: easier to understand

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

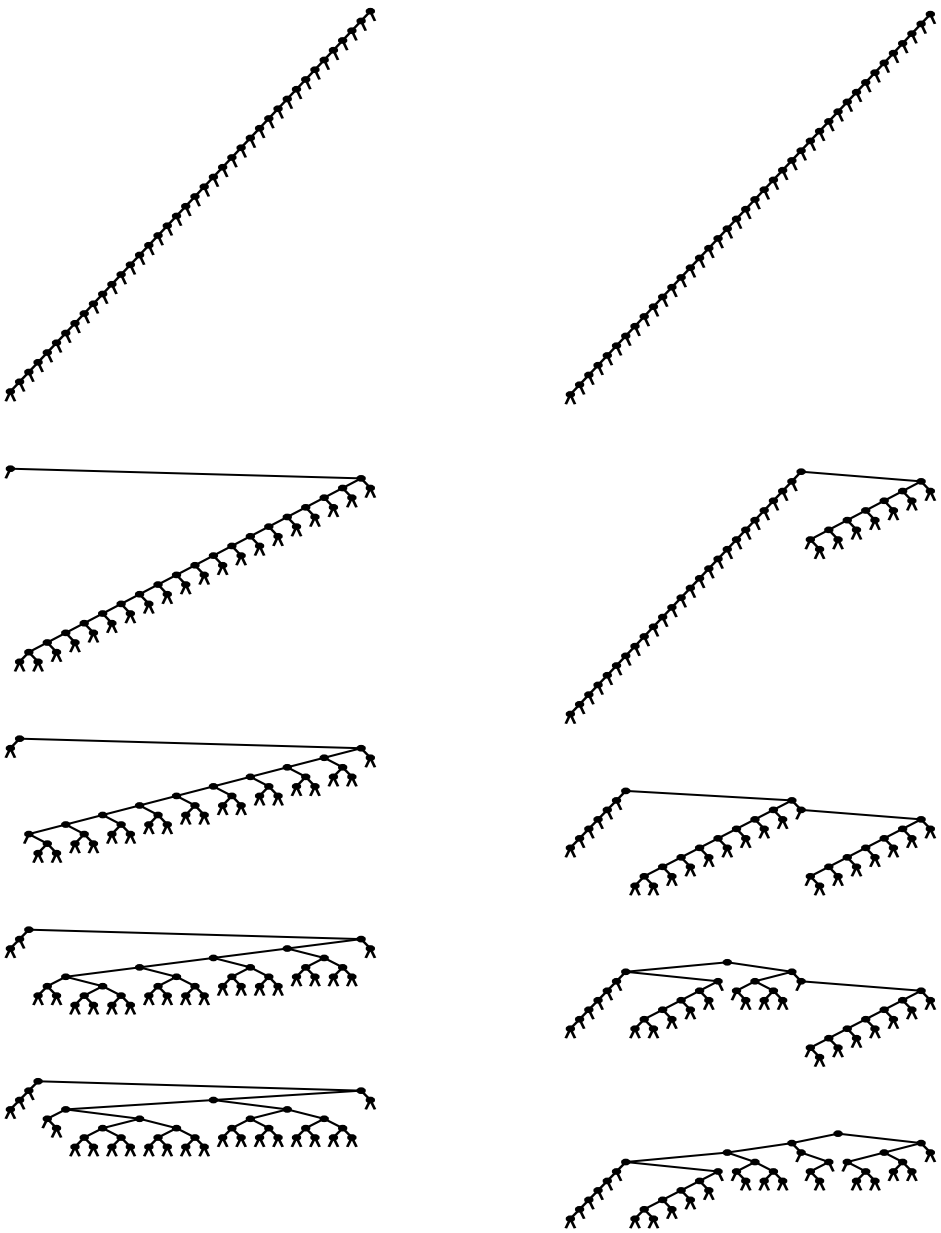


Brings new node to root
Also brings all nodes on search path
closer to root

Splay tree example

Splay rotations halve the search path

Long paths can build up, BUT
tree rebalances whenever we traverse them



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);
  }
}
```

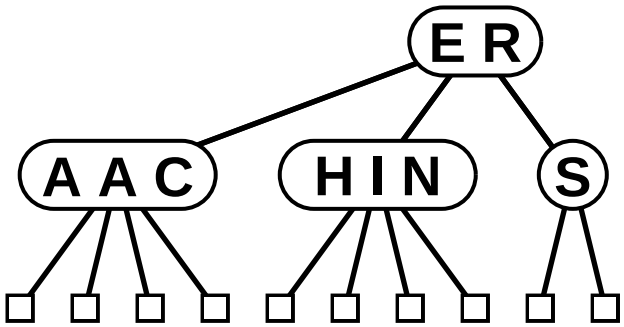
Guaranteed performance over SEQUENCE of ops

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

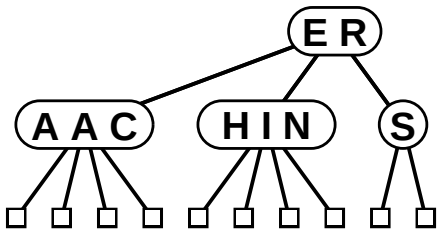
- compare search key against keys in node
- find interval containing search key
- follow associated link (recursively)

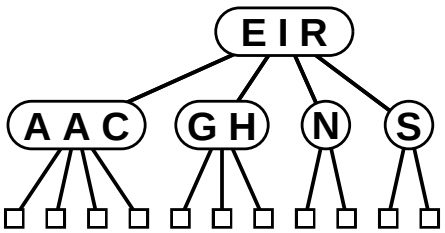
INSERT

- 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: ??

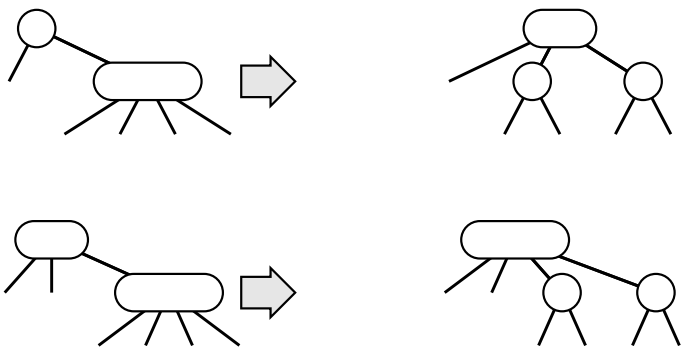
Top-down 2-3-4 trees

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



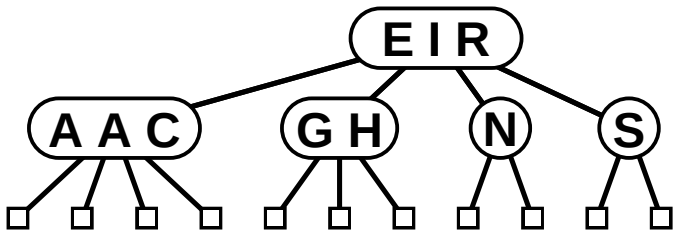


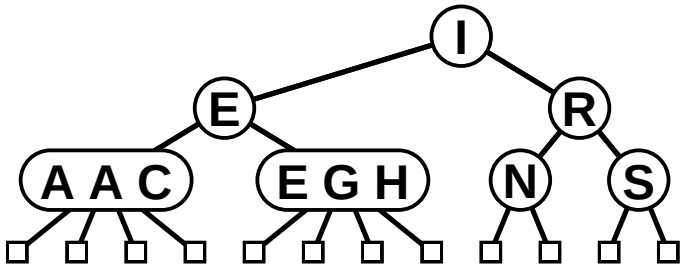
"Split" any 4-nodes encountered using
one of two transformations

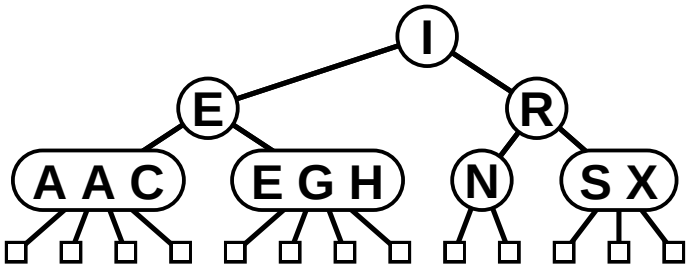


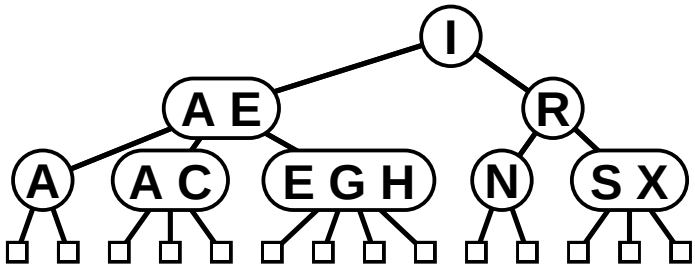
Invariant: "current" node is not a 4-node
Insertion at bottom therefore easy.

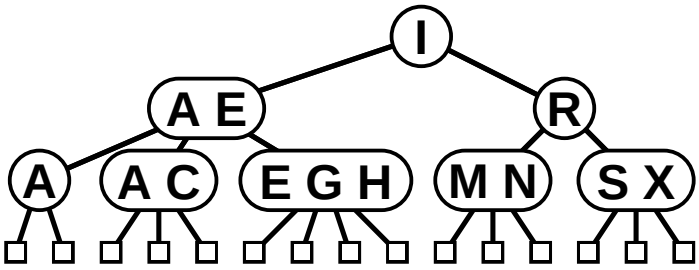
Top-down 2-3-4 tree construction

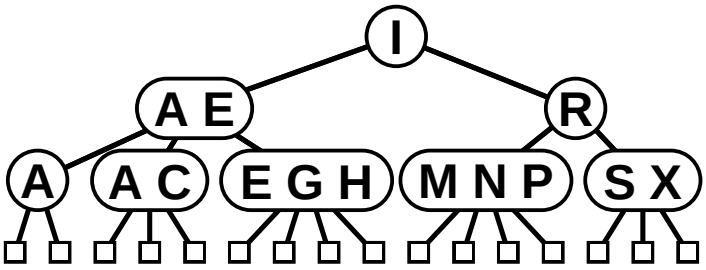


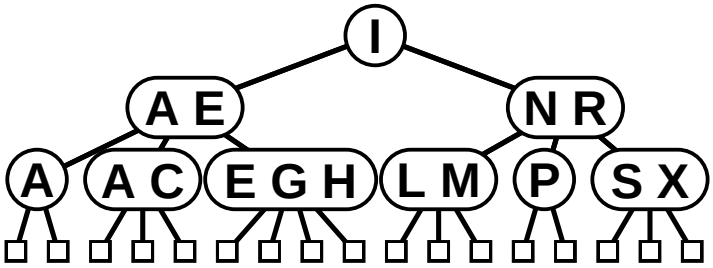


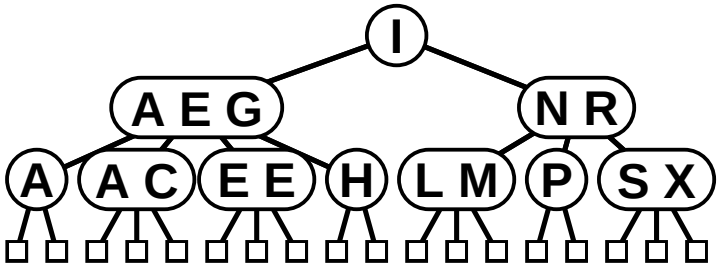












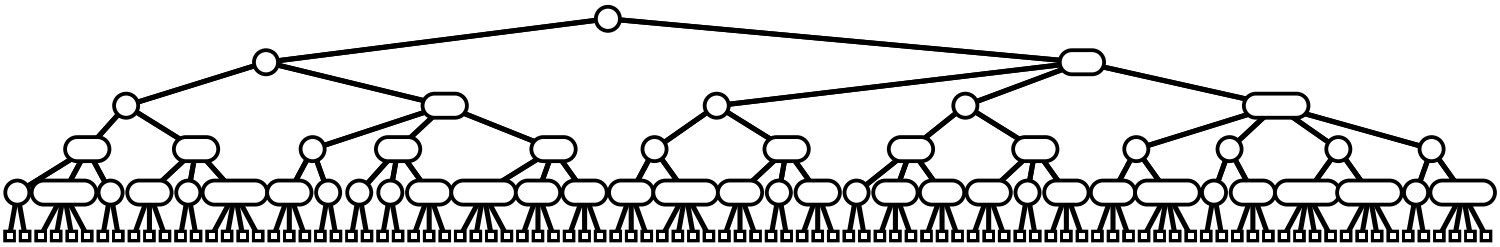
Trees grow up from the bottom

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)
   { Key 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

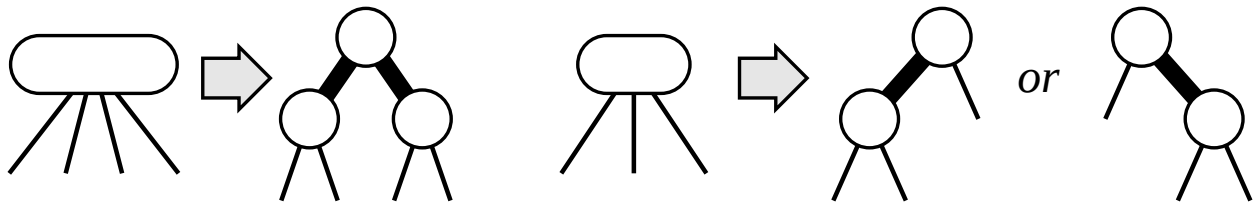
* "therightlink(x, v)"
* maintaining multiple node types
* large number of cases for "split"

Search also more complicated than for BST

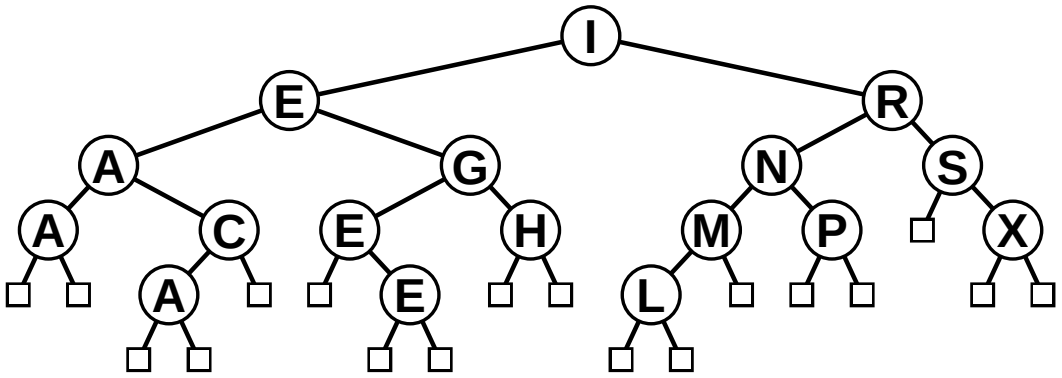
Red-black trees

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

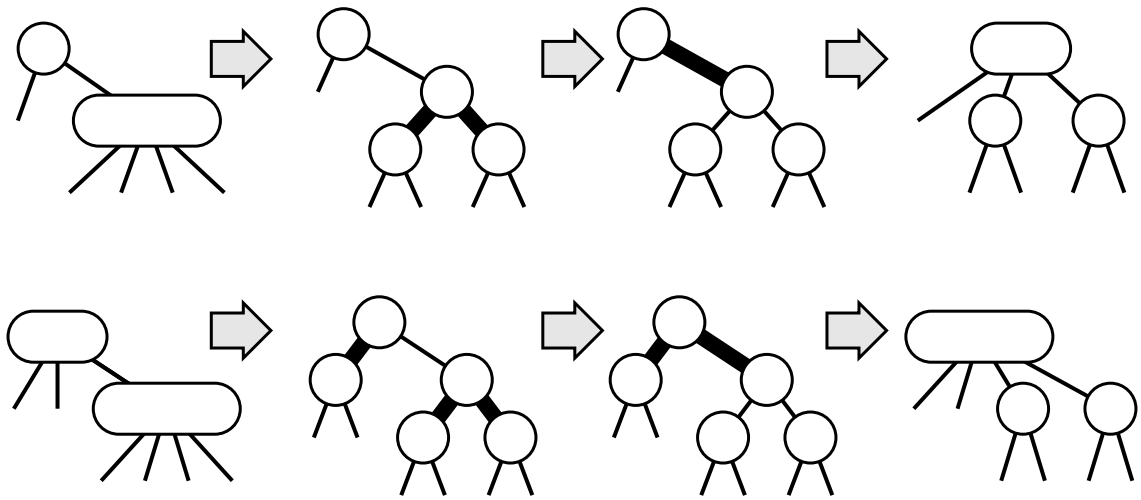
INSERT:

split reduces to fewer cases

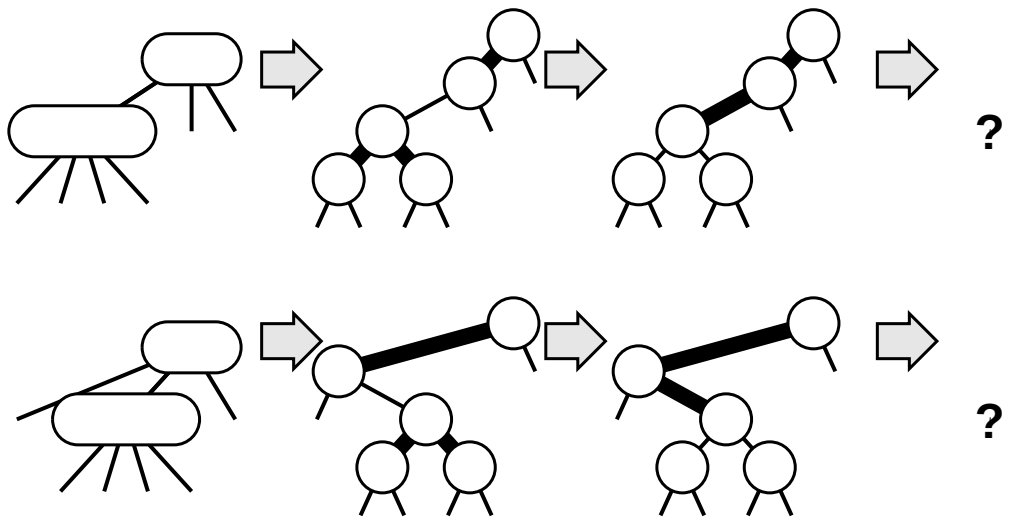
Splitting nodes in red-black trees

Two cases are easy

(need only to switch colors)



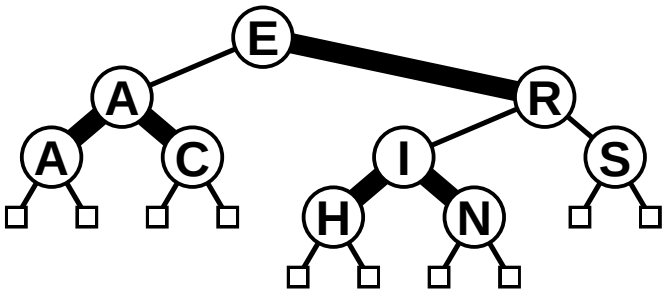
Two cases require ROTATIONS

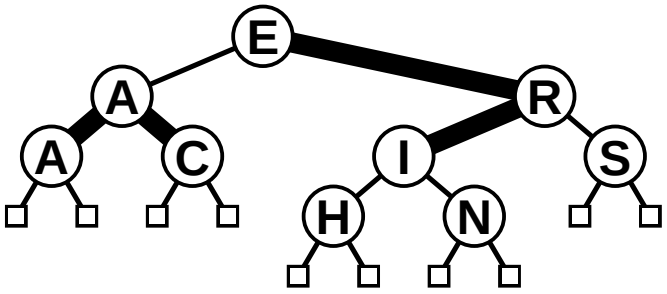


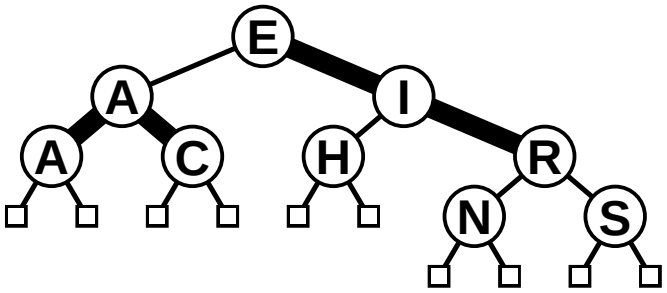
Can use red-black abstraction directly

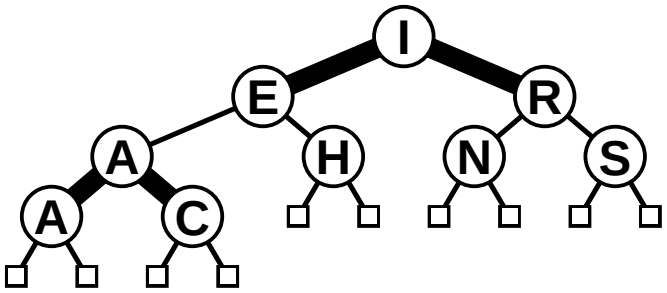
Invariant: never two consecutive red nodes on any path.

RB tree node split example







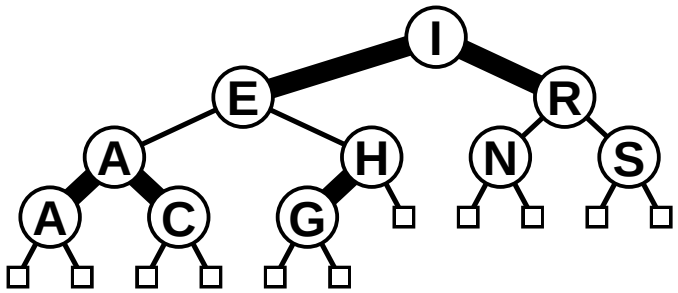


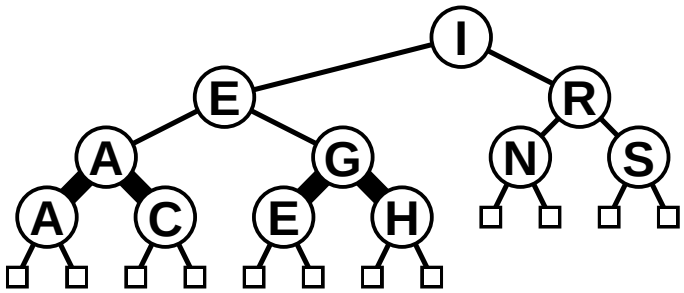
Red–black tree implementation

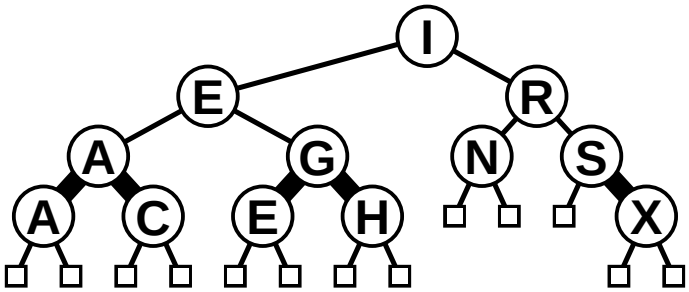
Search code same as BST; insert code balances

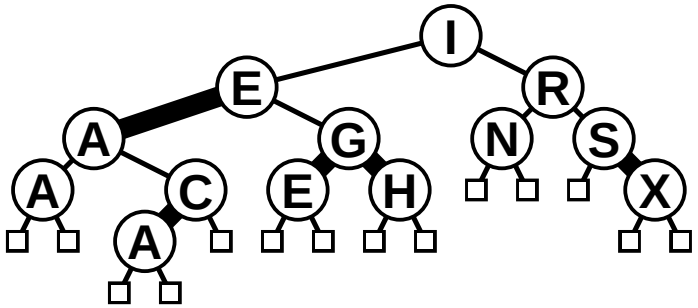
```
link RBinsert(link h, Item item, int sw)
{ Key v = key(item);
  if (h == z) return NEW(item, z, z, 1, 1);
  if ((hl->red) && (hr->red))
    { h->red = 1; hl->red = 0; hr->red = 0; }
  if (less(v, key(h->item)))
    {
      hl = RBinsert(hl, item, 0);
      if (h->red && hl->red && sw) h = rotR(h);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

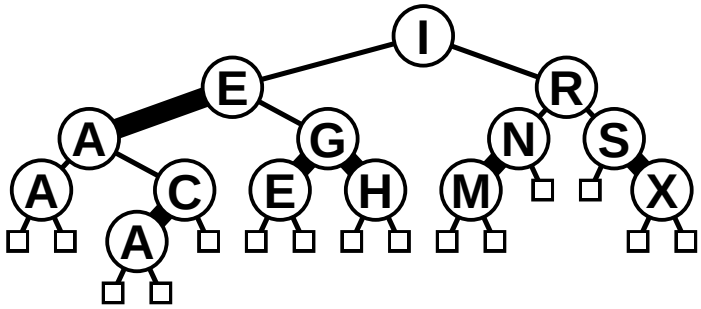
Red-black tree construction

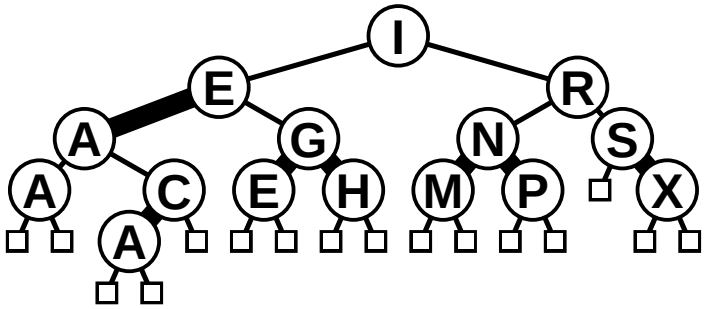


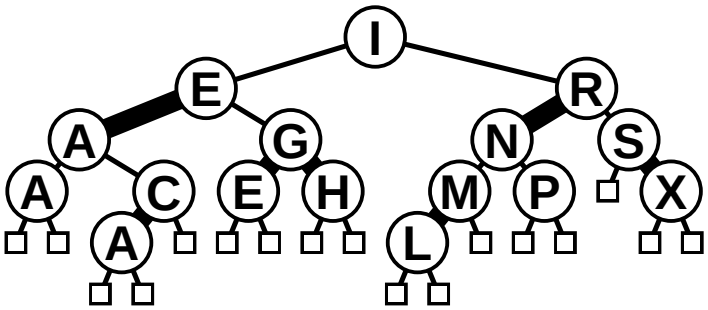


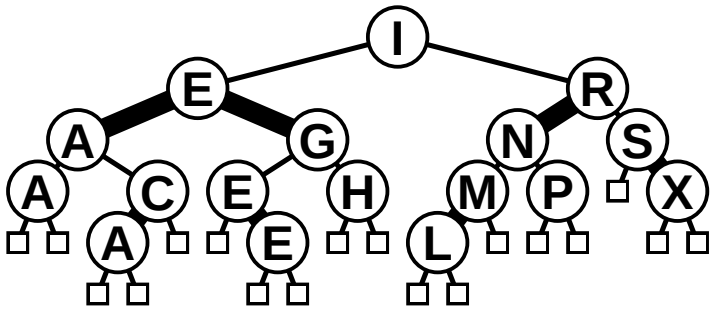










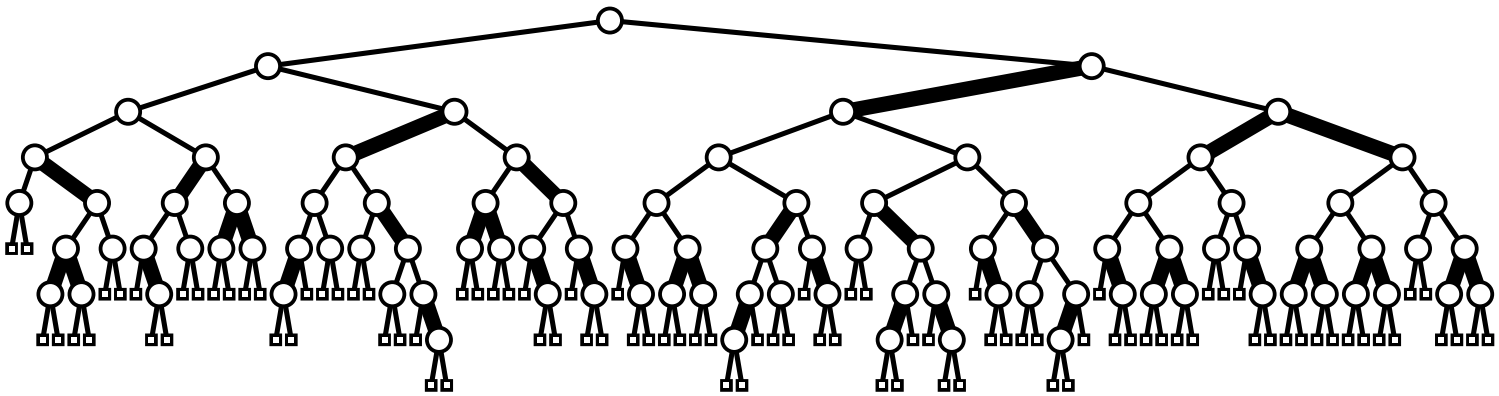


Balance in red-black trees

In red-black trees,

the longest path is at most

twice the length of the shortest path



worst case: less than 2lgN

Comparisons within nodes are accounted for

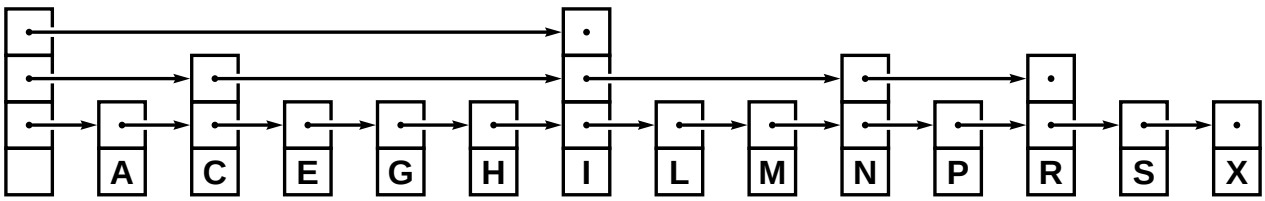
Summary

Goal: Symbol table implementation
with O(lgN) GUARANTEED performance
for both search and insert

probablistic guarantee: random BSTs, skip lists
amortized guarantee: splay trees
optimal guarantee: red-black trees

Algorithms are varations on a theme
rotations during insertion

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

Extensions to search structures for huge files
(stay tuned)