Contemporary Algorithms T.2/2019-20

Basic Information/Logistics

- Website: https://cs.muic.mahidol.ac.th/courses/calgo
- This course: highlights of useful data structures & algorithmic ideas from the past 50 years.
- Schedule will be made as we go along.
- No lecture notes. You'll scribe!

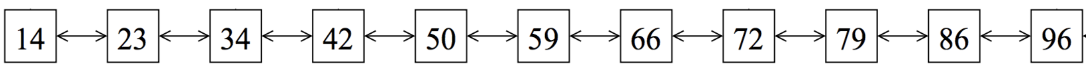
Week 1: Ordered Maps

- Think TreeMap in Java
- The keys are ordered supporting add/remove/update/lookup
- Goal: for most operations, take O(logn) time or faster

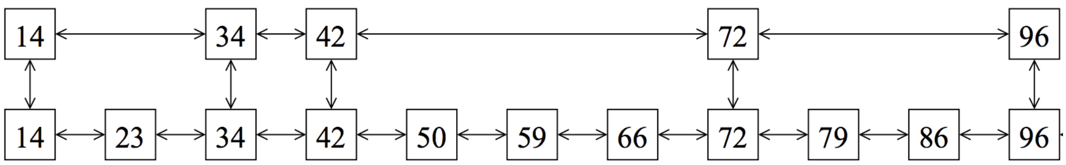
The themes: approximating a perfectly balanced structure

Skip Lists

Starting simple,



Sorted Linked List: $O(n)$
Can we do better?



2-level Linked List: $m + \frac{n}{m} \leq \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$
3-LL: $3\sqrt[3]{n}$
k-LL: k$\sqrt[k]{n}$
Pick k so that search time is minimized, we can get search time = 2lg$n$ and becomes a perfect binary tree.
Maintaing this exactly is too rigid, we relax by flipping a coin.

For each item, we toss a fair coin. If the coin turns head, we promote that item up. By this, we also get approximately lg$n$ layers.
To insert: find right spot, promote the new element (by 0.5 prob) and cut up necessary siblings to remain the structure.

$\mathbb{E}$[search cost] = ? Think backwards from bottom to top.

For one layer, $\mathbb{E}$[walk across] = $\mathbb{E}$[cannot walk up] = $\mathbb{E}$[# toss tails until heads] = $\frac{1}{p} = 2$
We expect lg$n$ layers, so search cost = 2lg$n$.

Definition: An even E_α occurs with **high probability (whp)** if for any α, $P[E_\alpha] \geq 1 - \frac{c_\alpha}{n^\alpha}$ where c_α is a constant that depends only on α.

Thm: Every search costs $\Theta(lg n)$ whp.
Proof ideas:
Analyze search backwards from bottom to top layers.

- Start at the found element at the bottommost layer.
- If we get tail, walk across to the left. It we get head, walk up the tree.
- Stop when reach the topmost layer.

Need to show two things

(i) Walk up is $O(lgn)$.

(ii) Walk across is $O(lgn)$.

Lemma (i): Skip list has $O(lgn)$ levels whp. (Showing walk up is $O(lgn)$)
Proof:

$$Pr[\text{a key } k_i \text{ grows taller than } 100 \;\; lgn] = \frac{1}{2^{100lgn}} = \frac{1}{n^{100}}$$

$$\begin{aligned} Pr[\text{a skiplist has} \geq 100lgn \text{ levels}] &= Pr[\exists_i, k_i \text{ grows to height} \geq 100lgn] \\ &\leq Pr[k_1 \text{ is too tall}] + Pr[k_2 \text{ is too tall}] + ... \\ &= n \times \frac{1}{n^{100}} \\ &= \frac{1}{n^{99}} \end{aligned}$$

Therefore,

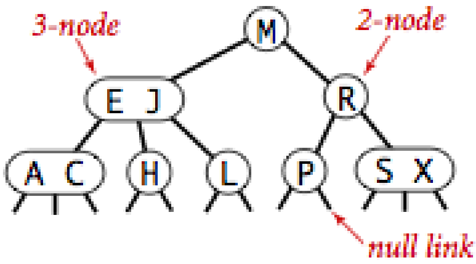
$$\begin{aligned} Pr[\text{a skiplist has} < 100lgn \text{ levels}] &= 1 - Pr[\text{a skiplist has} \geq 100lgn \text{ levels}] \\ &\geq 1 - \frac{1}{n^{99}} \end{aligned}$$

Lemma(ii) : Out of $1000lgn$ flips, you get $\geq 100lgn$ heads up. (Showing walk across is $\Theta(\lg n)$)
Proof:

$$Pr[\text{getting exactly } 100lgn \text{ heads}] = \binom{1000lgn}{100lgn} \underbrace{\frac{1}{2}^{900lgn}}_{\text{Tail}} \underbrace{\frac{1}{2}^{100lgn}}_{\text{Head}}$$

$$\begin{aligned} Pr[\text{getting } \; < 100lgn \text{ heads}] &= \binom{1000lgn}{100lgn} \underbrace{\frac{1}{2}^{900lgn}}_{\text{Tail}} \\ &\leq \frac{e1000lgn}{100lgn} \left(\frac{1}{2}^{900lgn}\right) \\ &= 2^{(lg(10e))100lgn - 900lgn} \\ &= 2^{100lg(10e) - 900} \\ &= n^{-\alpha} \\ &= \frac{1}{n^\alpha} \end{aligned}$$

2-3 search trees



Anatomy of a 2-3 search tree

In **BST**, a node has 1 key and 2 links, we call this a **2-node**.

In **2-3 search tree**, we also allow a node with 2 keys and 3 links which we call a **3-node**.

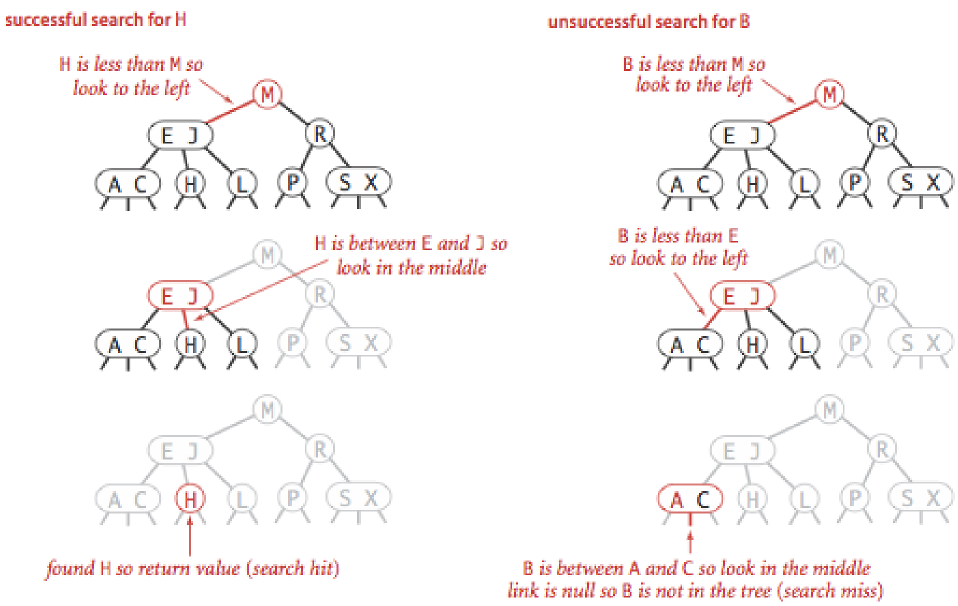
Definition: A *2-3 search tree* is either empty or

- A *2-node* (with one key, two links: a left link with a 2-3 search tree with smaller keys, a right link with a 2-3 search tree with larger keys.)
- A *3-node* (with two keys, three links: a left link with smaller keys, a middle link with keys in between, a right link with larger keys.)

  note: a *null link* is a link to an empty tree.
  Null links of a *perfectly balanced* 2-3 search tree are at the same distance from the root.

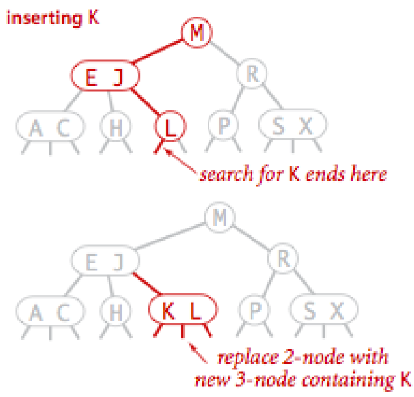
Goal: Any insert operation must still make a 2-3 search tree perfectly balance. (From now, we'll use the term *2-3 tree* to refer to a perfectly balanced 2-3 tree.)

Search: the search algorithm is the same as that of BSTs. We either get a search hit or miss.



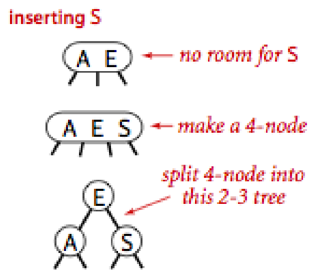
Search hit (left) and search miss (right) in a 2-3 tree

Insert into a 2-node: if we do an unsuccessful search and terminate at a 2-node at the bottom, just replace the node with a 3-node.



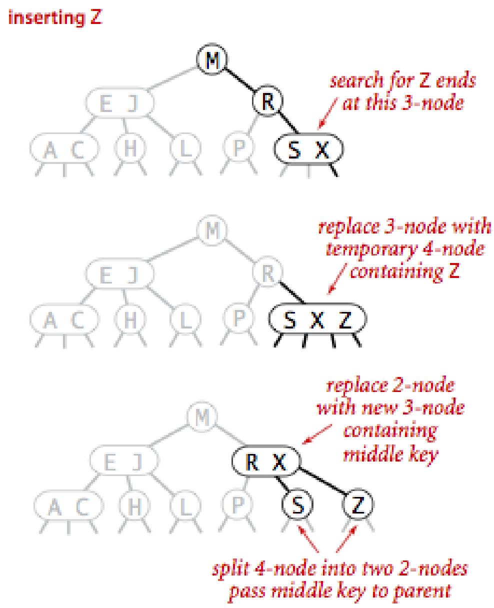
Insert into a 2-node

Insert into a tree of a single 3-node:



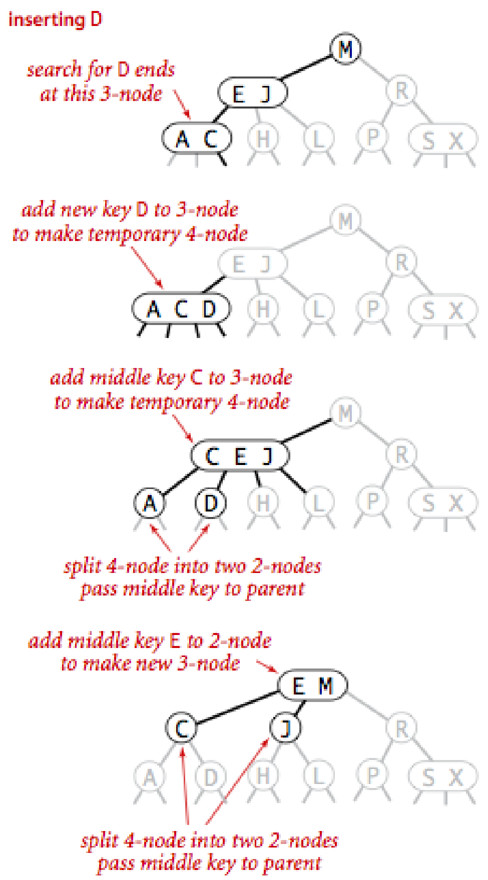
Insert into a single 3-node

Insert into a 3-node whose parent is a 2-node:



Insert into a 3-node whose parent is a 2-node

Insert into a 3-node whose parent is a 3-node:



Insert into a 3-node whose parent is a 3-node

Proposition: Search and insert in a 2-3 tree with N keys are guaranteed to visit at most $\log(N)$ nodes.

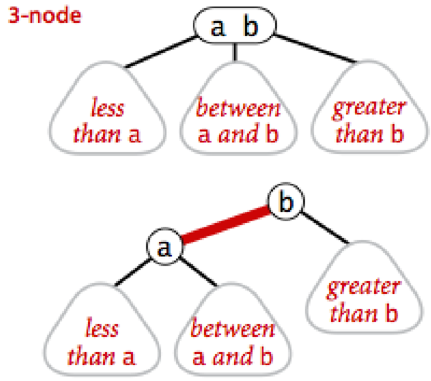
Proof: The height of an N-node 2-3 tree is between $\lfloor log_3 N \rfloor$ (if the tree is all 3-nodes) to $\lfloor log N \rfloor$ (if the tree is all 2-nodes).

Red-black BSTs

We'll implement a 2-3 tree using a *red-black BST* representation. We have two types of links: red and black.

- *Red* links bind two 2-nodes to represent 3-nodes. Red links **lean left**.
- *Black* links bind together the 2-3 tree.

  [see a figure below.]



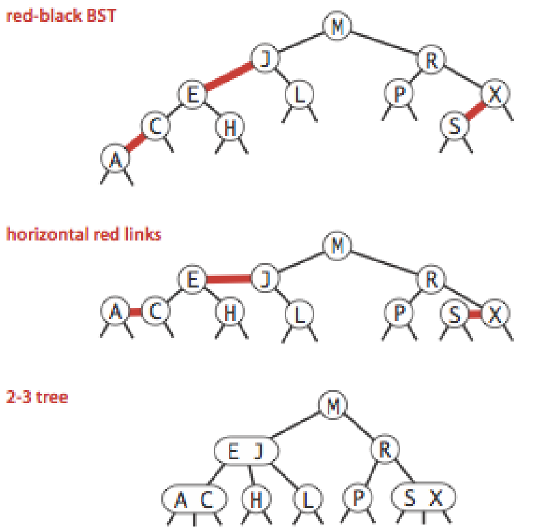
We can see that this representation above (the one with red link) allows us to use code from the standard BST search without modification.

Definition: red-black BSTs are BSTs that have red and black links that satisfy:

- Red links lean left.
- No node has 2 red links connected to it. (no consecutive red links)
- Every path from the root to the null link (a link to an empty tree) has the same number of black links. (**perfect black balance**)

1-1 correspondence between red-black BSTs and 2-3 trees:

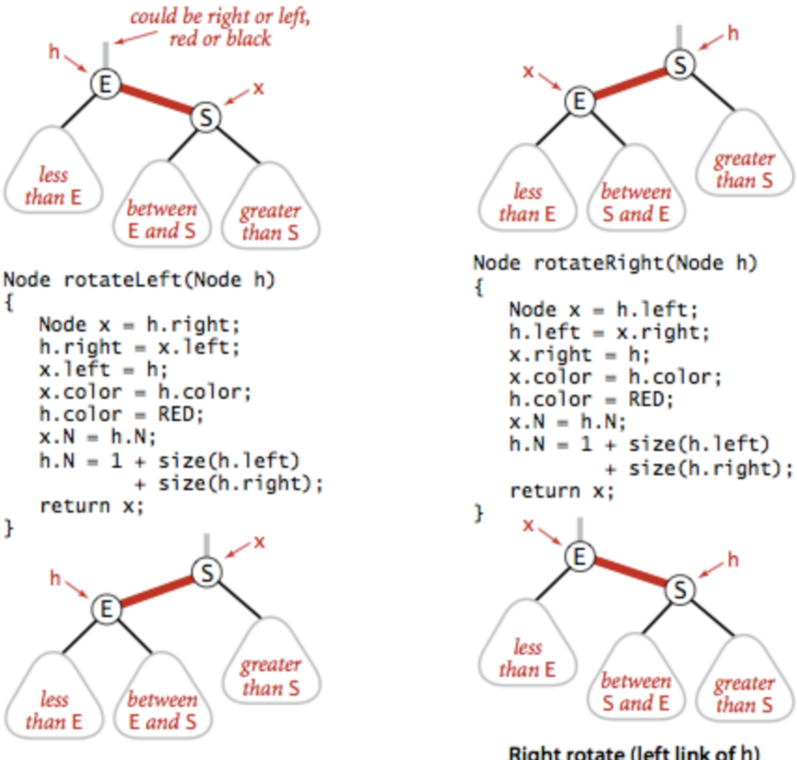
- If we draw the red links horizontally in a read-black BST, all the null links are the same distance from the root.
- If we collapse the nodes connected in red links, we get a 2-3 tree.



1-1 correspondence between red-black BSTs and 2-3 trees

Rotations:

- We use rotations to maintain **the 1-1 correspondence between red-black BSTs and 2-3 trees**.



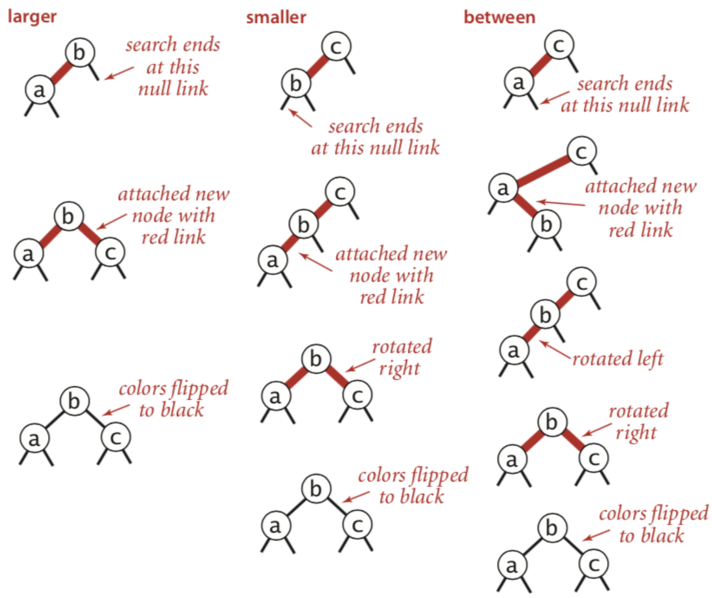

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

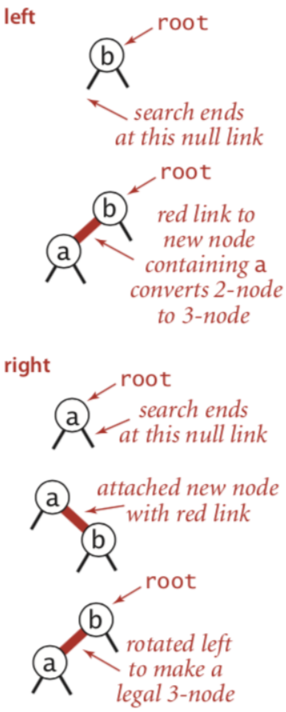
Left rotate (right link of h)

Right rotate (left link of h)

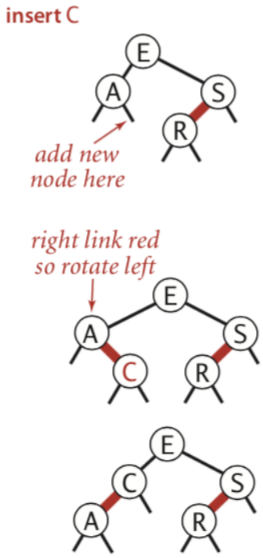
- We also use rotations to maintains the other 2 properties: no consecutive red links and red links must only lean left.



Insert into a single 3-node (three cases)



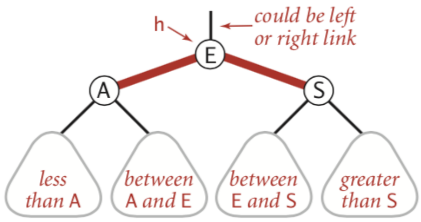
Insert into a single
2-node (two cases)



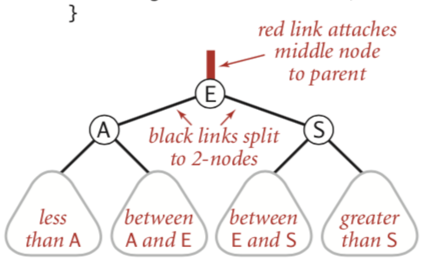
Insert into a 2-node
at the bottom

Flipping colors:

- we flip color to preserve the property **perfect black balance** in the red-black BSTs.

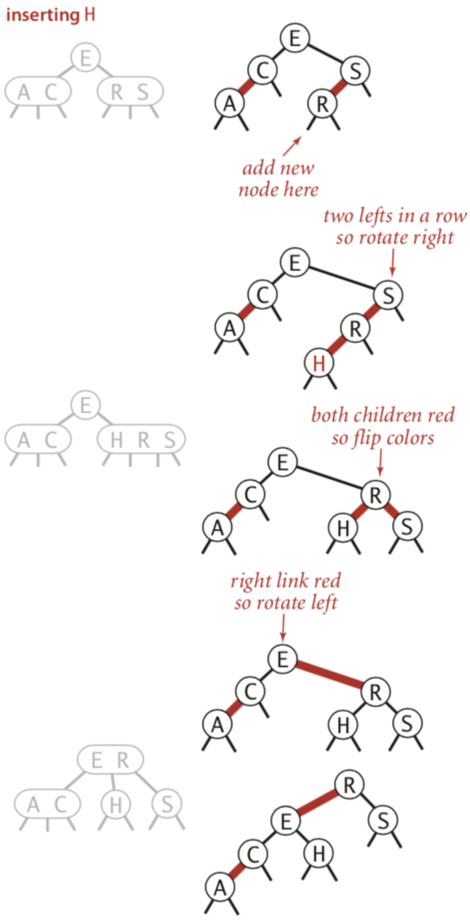



```
void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

Insert into a 3-node at the bottom:



Insert into a 3-node at the bottom