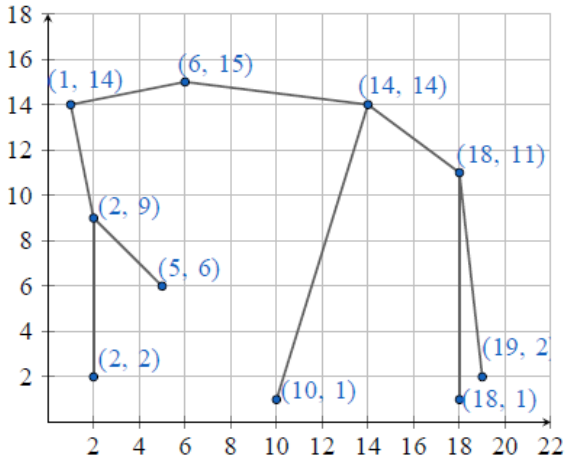
Treaps, Red-Black trees & Others

Treap (Cartesian tree)

- A treap is a data structure which combines binary tree and binary heap (hence the name: tree + heap ⇒ Treap).
- More specifically, treap is a data structure that stores pairs in a binary tree in such a way that it is a binary search tree X by and a binary heap by Y.

If some node of the tree contains values X_0, Y_0, all nodes in the left subtree have $X \le X_0$, all nodes in the right subtree have $X_0 \le X$, and all nodes in both left and right subtrees have $Y \le Y_0$.

A treap is also often referred to as a "cartesian tree", as it is easy to embed it in a Cartesian plane:



2

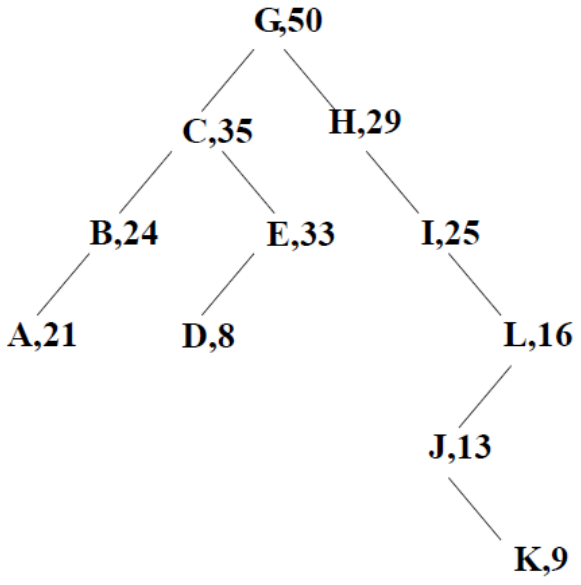
Trees, heaps, and treaps

- A binary search tree (BST) is a binary tree:
  a) Each Node in a BST contains a key; key values are comparable to each other
  b) A BST has the BST ordering property: For every node X, the key in X is greater than all keys in the left subtree of X, and less than all keys in the right subtree of X
- A heap is a binary tree:
  - Each Node in a heap contains a priority; priority values are comparable to each other
  - A heap has the heap ordering property: For every node X, the priority in X is greater
  - than or equal to all priorities in the left and right subtrees of X
- A treap is a binary tree:
  - Nodes in a treap contain both a key, and a priority
  - A treap has the BST ordering property with respect to its keys, and the heap ordering property with respect to its priorities

3

Treaps: an example

Suppose keys are letters, with alphabetic ordering; priorities are integers, with numeric ordering. This tree is a treap:



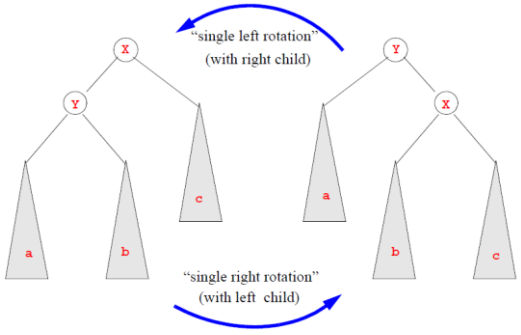
4

Uniqueness of treaps

- Given a set of (key, priority) pairs, with all the key values unique, you could always construct a treap containing those (key, priority) pairs:
  - Start with an empty treap
  - Insert the (key, priority) pairs in decreasing order of priority, using the usual binary search tree insert algorithm that pays attention to the key values only
  - The result is a treap: BST ordering of keys is enforced by the BST insert, heap ordering of priorities is enforced by inserting in priority sorted order
- If the priority values as well as the key values are unique, the treap containing the (key, priority) pairs is unique.
- For example, the treap on the previous page is the unique treap containing these pairs: (G,50),(C,35),(E,33),(H,29),(I,25),(B,24),(A,21),(L,16),(J,13),(K,9),(D,8)
- We are interested in algorithms that create a treap from (key, priority) pairs no matter what order they are inserted in.

Operations on treaps

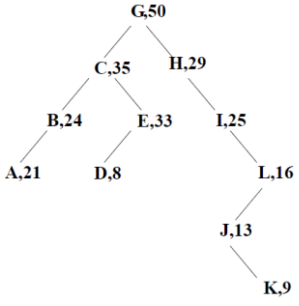
- Treaps permit insert, delete, and find operations (also others but these are basic)
- Finding a key in a treap is very easy: just use the usual BST search algorithm
- Insert and delete of keys are slightly more complicated, since the operations must respect both the BST and heap ordering properties as invariants
- Recall that insert and delete in heaps use "bubble up" and "trickle down" exchanges to restore the heap ordering property
- Those same operations won't work in treaps, because they can destroy the BST ordering property
- The trick is to use AVL rotations instead of exchanges: An AVL rotation always preserves BST ordering, and so it can be used to move a (key,priority) pair up or down the tree to correct a failure of heap ordering without disturbing BST ordering



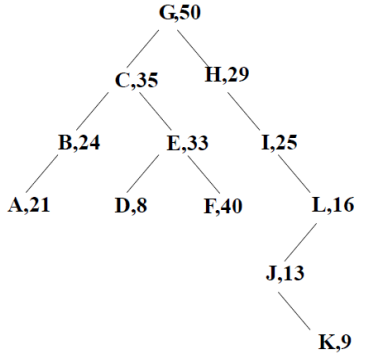
⇐ AVL Single Rotations

Insert in treaps: an example,⇒

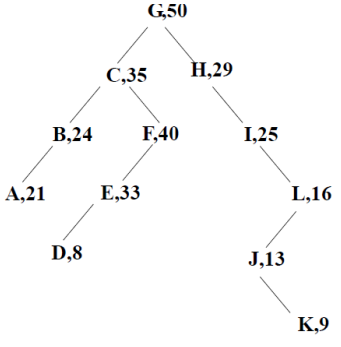
Insert pair (F,40) in this treap

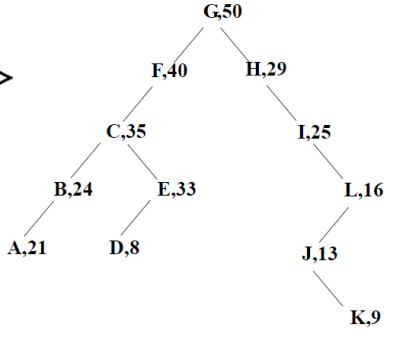


Ordinary BST insert gives this:



One AVL left rotation gives this:





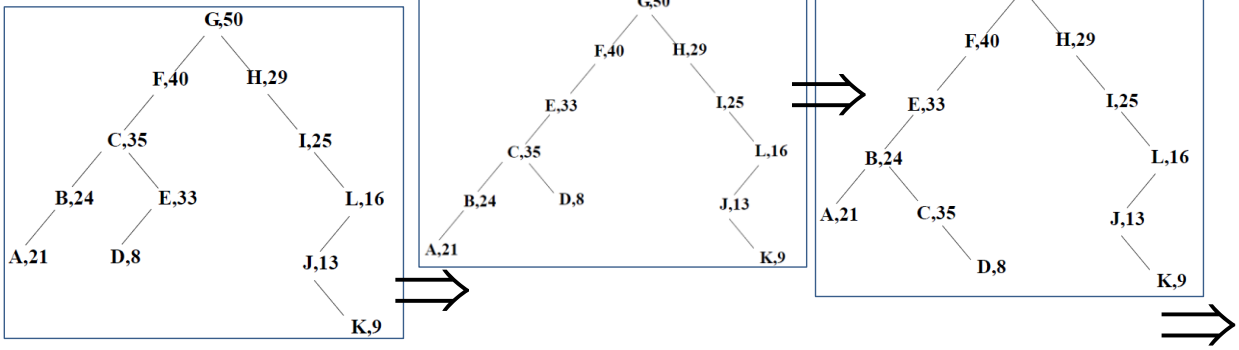
Now the tree is again a treap:

7

Delete in treaps

- To delete a key K, do the following:
  - Search for the node X containing K using the usual BST find algorithm
  - If the node X is a leaf, just delete the node (unlink it from its parent)
  - Otherwise, use AVL rotations to rotate the node down until it becomes a leaf; then delete it
  - (If there are 2 children, always rotate with the child that has the larger priority, to preserve heap ordering: use a left rotation if the right child has larger priority, right rotation otherwise
- Since AVL rotations are constant-time operations, delete in a treap can be performed in time O(H), where H is the height of the treap
- (Note that the AVL-rotation-to-leaf trick also works for delete in ordinary BST's)

Delete in treaps: an example

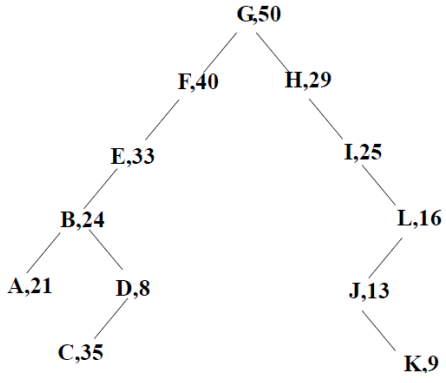


Step 1: Delete the key C from by rotating it down this treap when it is not a leaf

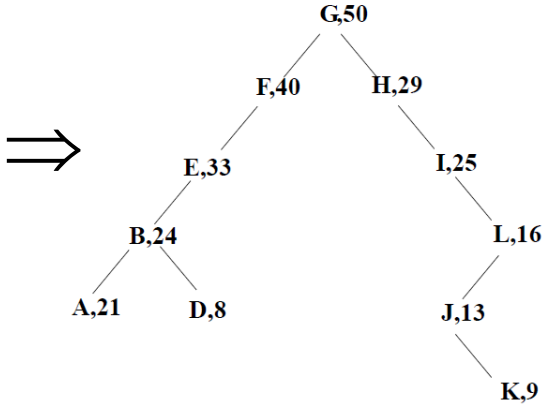
Step 2: Rotating the node with its larger priority child gives this. C is still not a leaf, so need to rotate down again. The node containing C is still not a leaf, so need to rotate down again

Step 3: Rotating the node with its larger priority child now gives this. The node containing C is still not a leaf, so need to rotate down yet again.

Delete in treaps: an example (contd.)



Step 4: Now C is a leaf; so, delete it



Step 5: After clipping off the leaf, the result is again a treap.

Why treaps?

- They permit very easy implementations of **split** and **join** operations, as well as simple implementations of insert, delete, and find.
- They are the basis of randomized search trees, which have performance comparable to balanced search trees but are simpler to implement.
- They also lend themselves well to more advanced tree concepts, such as weighted trees, interval trees, etc. [out of our scope in this class]

Tree Splitting

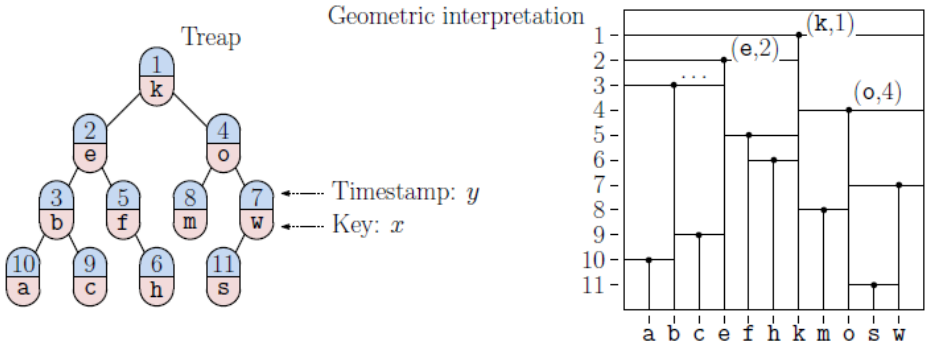
- The **tree splitting** problem: Given a tree and a key value K not in the tree, create two trees: One with keys less than K, and one with keys greater than K.
  - Given a tree and a key value K not in the tree, create two trees: One with keys less than K, and one with keys greater than K
- This is easy to solve with a treap, once the insert operation has been implemented.
  - Insert (K, ∞) in the treap
  - Since this has a higher priority than any node in the heap, it will become the root of the treap after insertion
  - Because of the BST ordering property, the left subtree of the root will be a treap with keys less than K, and the right subtree of the root will be a treap with keys greater than K. These subtrees then are the desired result of the split.
- Since insert can be done in time O(H) where H is the height of the treap, splitting can also be done in time O(H) [this same idea could be used in an ordinary BST as well]

Tree Joining

- **Tree joining or merging**: Given two trees T1, T2, such that each key in T1 is less than all keys in T2, create a new tree T that contains all and only the keys from T1 and T2
  - This is easy to do with a treap, once the delete operation has been implemented:
    - (a) Create a "dummy" node with any key value and any priority;
    - (b) Make the root of T1 be the left child, and the root of T2 be the right child, of this dummy node;
    - (c) Perform a delete operation on the dummy node.
- Since delete can be done in time O(H) where H is the height of the treap, joining can also be done in time O(H) [this same idea could be used in an ordinary BST as well]

Geometric Interpretation of treaps

- Although the treap is usually viewed as a 1-dimensional search structure, the introduction of numeric priorities suggests that we can interpret each key-priority pair as a point in 2-dimensional space.
- We can visualize a treap as a subdivision of 2-dimensional space as follows. (a) Place all the points in rectangle, where the y-coordinates (ordered top to bottom) are the priorities and the x-coordinates are the keys, suitably mapped to numbers. (b) Draw a horizontal line through the root. Because there are no points of lower priority, all the other points lie in the lower rectangle. (c) Shoot a vertical ray down from this point. This splits the rectangle in two, with the points of the left subtree lying in the left rectangle and the points of the right subtree lying in the right subtree. (d) Repeat the process recursively on each of the two halves.



Disadvantages of treaps

- Treaps permit easy implementations of find, insert, delete, split, and join operations
- All these operations take worst case time O(H), where H is the height of the treap
- However, treaps (like BST's) can become very unbalanced, so that H = O(N), and that's bad
- Maintaining a strict balance condition (like the AVL or red-black property) in a treap would be impossible in general, if the user supplies both key values and priorities: remember that treaps with unique keys and priorities are unique
- However, if priorities are generated randomly by the insert algorithm, balance can be maintained with high probability and the operations stay very simple [that is the idea behind Randomized Search Trees (RSTs)]

Randomized search trees

- Randomized search trees were invented by Cecilia Aragon and Raimund Seidel, in early 1990's
- RST's are treaps in which priorities are assigned randomly by the insert algorithm when keys are inserted
- To implement a randomized search tree:
  - start with a treap implementation and its insert method that takes a (key,priority) pair as argument
  - then to implement the RST insert method that takes a key as argument:
    - call a random number generator to generate a uniformly distributed random priority (a 32-bit random int is more than enough in typical applications; fewer bits can also be made to work well) that is independent of the key
    - call the treap insert method with the key value and that priority
- That's all there is to it: none of the other treap operations need to be changed at all
- (The RST implementation should take care to hide the random priorities, however)

16

Analysis of randomized search trees

- We will do an average case analysis of the "successful find" operation: how many steps are required, on average, to find that the key you're looking for is in the tree?
- Suppose you have a RST with N nodes $x_1, \ldots, x_N$, holding keys $k_1, \ldots, k_N$ and priorities $p_1, \ldots, p_N$, such that x_i is the node holding key k_i and priority p_i
- As always when doing average-case analysis, you have to be clear about your probabilistic assumptions. We will make 2:
  - Assumption #1: Each key $k_1, \ldots, k_N$ in the tree is equally likely to be searched for
  - Assumption #2: The priorities $p_1, \ldots, p_N$ are randomly uniformly generated independently of each other and of the keys
- For convenience we will assume that:
  - keys are listed in sorted order: $k_i < k_{i+1}$ for all $0<i<N$ (though keys can be inserted in any order),
  - all priorities are distinct

17

Expected node depth

- Let the depth of node x_i be $d(x_i)$, so the number of comparisons required to find key k_i in this tree is $d(x_i)$
- First, we will find the expected value (i.e. average) of $d(x_i)$, the depth of the node containing the i^{th} smallest key
- For this average-case analysis, we will average (not over all key insertion sequences, but) over all ways of generating random priorities during key insertion
- Let $Pr(p_1, \ldots, p_N)$ be the probability of generating the N priority values $p_1, \ldots, p_N$
  - Note that under the assumption that keys $k_1, \ldots, k_N$ are listed in sorted order, the priority values $p_1, \ldots, p_N$ determine the shape of the treap and the location of every key in it, so in particular they determine the depth of node x_i
- Then we can write the expected value (i.e. average) of $d(x_i)$ as:

$$E[d(x_i)] = \sum_{p_1, \ldots, p_N} Pr(p_1, \ldots, p_N) d(x_i)$$

Expected node depth

- Define A_{ij} to be the "indicator function" for the RST's ancestor relation:

$$A_{ij} = \begin{cases} 1, \text{ if } x_i \text{ is an ancestor of } x_j \\ 0, \text{ otherwise} \end{cases}$$

- (We will take $A_{ii} = 1$ for all i, i.e. we consider a node to be an ancestor of itself.)

- Now note that the depth of a node is just the number of ancestors it has; so we can write

$$d(x_i) = \sum_{m=1}^{N} A_{mi}$$

- ... and so the expected value of the depth of node x_i is (using the fact that the expectation of a sum is the sum of expectations):

$$E[d(x_i)] = \sum_{p_1, \ldots p_N} Pr(p_1, \ldots p_N) \sum_{m=1}^{N} A_{mi} = \sum_{m=1}^{N} E[A_{mi}]$$

- So now what is the expected value of A_{mi} ?

19

Probability of being an ancestor

- A_{mi} is an indicator function: it is a random variable that takes only values 0,1
- The expected value of any indicator function is just equal to the probability that that indicator function has value 1
- So, $E[A_{mi}] = Pr(A_{mi} = 1)$, i.e., the probability that node x_m is an ancestor of x_i
- Seidel & Aragon 1996 prove the following lemma (recall we are assuming that if $i<j$, the key in node x_i is smaller than the key in x_j , and priorities are distinct):

Lemma: x_m is an ancestor of x_i if and only if among all priorities p_h such that h lies between the indices m and i inclusive, p_m is the largest

- So, probability that node x_m is an ancestor of x_i is just the probability that the random priority generated for x_m is higher than the other $|m - i|$ priorities generated for nodes with indexes between m and i inclusive
- But since the priorities are generated randomly and independently, each of those $|m - i| + 1$ nodes have equal probability of having the highest priority! So,

$$E[A_{mi}] = \frac{1}{|m - i| + 1}$$

20

Expected depth of node x_i

- This lets us get what we were seeking first, the expected depth of the node with key k_i in a RST:

$$E[d(x_i)] = \sum_{m=1}^{N} E[A_{mi}] = \sum_{m=1}^{N} \frac{1}{|m-i|+1}$$

- It is interesting to see that this depends on i, the position of the key in the ordered set of keys in the RST. Keys in the middle of the ordering will on average be somewhat deeper than smaller or larger keys. This is true of randomly constructed BST's as well
- For example, if the keys are integers 1,2,...,1000, the expected depth of the node with key 500 is 12.59, while the expected depth of the node with key 1 or 1000 is 7.49

- But we are really interested in the average number of comparisons needed to find a key, assuming that all keys are equally likely to be searched. This is just the expected node depth, averaged over all nodes:

$$D_{avg}(N) = \sum_{i=1}^{N} \frac{1}{N}(E[d(x_i)]) = \frac{1}{N}\sum_{i=1}^{N}\sum_{m=1}^{N} \frac{1}{|m-i|+1}$$

21

Simplifying the double summation

- Let's simplify that double summation

$$\sum_{i=1}^{N} \sum_{m=1}^{N} \frac{1}{|m-i|+1}$$

- Note that there are N^2 terms. We can write the N^2 different values of the denominator $|m-i|+1$ as entries in an NxN matrix, where rows are indexed by m, columns by i. For example, for N = 9:

1	2	3	4	5	6	7	8	9
2	1	2	3	4	5	6	7	8
3	2	1	2	3	4	5	6	7
4	3	2	1	2	3	4	5	6
5	4	3	2	1	2	3	4	5
6	5	4	3	2	1	2	3	4
7	6	5	4	3	2	1	2	3
8	7	6	5	4	3	2	1	2
9	8	7	6	5	4	3	2	1

- You can see that in general there will be N 1's, 2(N-1) 2's, 2(N-2) 3's, ... , 4 (N-1)'s, and 2 N's
- That lets us rewrite the double summation, as shown next

22

The solution

- So we can write

$$\sum_{i=1}^{N}\sum_{m=1}^{N}\frac{1}{|m-i|+1} = 2\sum_{i=1}^{N}\frac{N-i+1}{i} - N = 2(N+1)\sum_{i=1}^{N}\frac{1}{i} - 3N$$

- And thus we have that the average number of comparisons for a successful find (i.e., the average node depth) in a randomized search tree is

$$D_{avg}(N) = \frac{2(N+1)}{N}\sum_{i=1}^{N}\frac{1}{i} - 3$$

- This is exactly the same as the average node depth for a binary search tree, under the assumption that all key insertion sequences are equally likely!
- Think about it, that seems right:
  - The RST's treap structure is identical to a BST with keys inserted in order of their priority
  - With random priorities, all priority orderings in an RST are are equally likely
- So what is the difference in practice between a RST and a BST?

23

Comparing RST's and vanilla BST'

- We have seen that in the average depth of a node in a N-node RST is the same as in a N-node BST: for large N it is approximately $2 \ln(N) = 1.386 \log_2 N$, which is $O(\log N)$
- This seems very good, but the analysis for the BST depended on the assumption that all sequences of key insertions were equally likely
- Often in practice "bad" sequences of BST key insertions (in which the keys are somewhat sorted) can in fact be more likely than others
- Also, if a particular sequence is "bad", it will be bad (leading to much worse than 2lnN average depth) *every* time a BST is built with that sequence
- However, in a randomized search tree, the average case analysis is *independent of the sequence of key insertions*
- If a good random number generator is used to generate the treap priorities, the probability of constructing a "bad" randomized search tree is very low, no matter what the sequence of key insertions is
- That's why RST's are better than vanilla BST's!

More properties of randomized search trees

- The expected value of node depth in a RST is $O(\log N)$, and thus average time cost for successful find is $O(\log N)$
- Similar considerations show that unsuccessful find, insert, delete, split, and join in a RST all have average time cost $O(\log N)$
- It is *possible* for a randomized search tree to be badly unbalanced, with height significantly worse than $\log_2 N$, where N is the number of nodes in the treap; however this is unlikely to happen
  - To be badly unbalanced, the random priorities have to be correlated in a certain way with key values, and with a good random number generator this will be unlikely to occur
- An interesting question is: *How* unlikely is that to happen?

More properties of randomized search trees, cont'd

- The expected time costs are like average time costs, averaged over many constructions of a treap with the same N keys, but different random priorities
- Question: For a single randomized search tree with N keys, how likely is it that its height is much greater than a deterministically balanced search tree such as AVL?
- Aragon and Seidel analyzed this question and showed that the answer is:
  - It is possible, but it can be considered extremely unlikely
- They derive this formula (here e is the natural logarithm base, ln is log base e, and c is any positive constant):

$$Pr(\ H > 2c\ \ln N\ ) \ < \ N\left(\frac{N}{e}\right)^{-c\ln(c/e)}$$

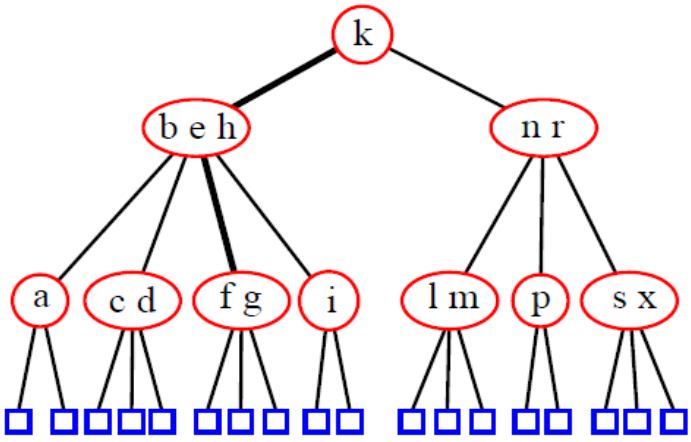
- Example: N=10000, so ln N = 9.210... Pick c = 5.429... so $2c$ ln$N$ = 100. Plugging in numbers, we get $Pr(\ H > 100\ ) \ < 4.081 \cdot 10^{-10}$
- That is: the chance that the height of a randomized search tree with 10,000 nodes will be greater than 100 is less than one in two billion
- So, although randomized search trees do not absolutely guarantee good performance, they will almost certainly provide good performance in practice

26

2-3-4 Trees and Red-Black Trees

2-3-4 Trees

- Nodes store 1, 2, or 3 keys and have 2, 3, or 4 children, respectively
- All leaves have the same depth



$$\frac{1}{2}\log(N+1) \leq \text{height} \leq \log(N+1)$$

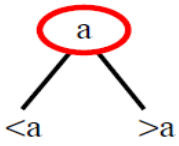
28

2-3-4 Tree Nodes

- Introduction of nodes with more than 1 key, and more than 2 children

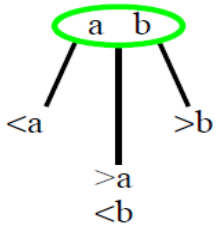
2-node:

- same as a binary node



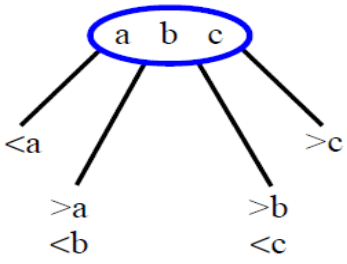
3-node:

- 2 keys, 3 links



4-node:

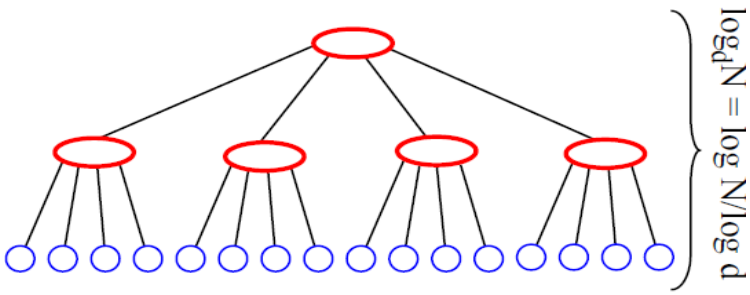
- 3 keys, 4 links



29

Why 2-3-4?

- Why not minimize height by maximizing children in a "d-tree"?
- Let each node have d children so that we get O($\log_d N$) search time! Right?

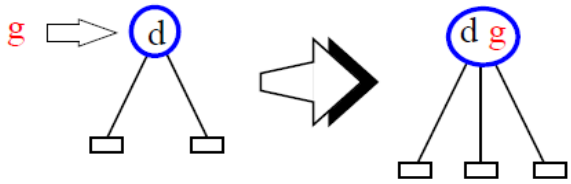


- That means if $d = N^{1/2}$, we get a height of 2
- However, searching out the correct child on each level requires O(log $N^{1/2}$) by binary search
- 2 log $N^{1/2}$ = O(log N) which is not as good as we had hoped for!
- 2-3-4-trees will guarantee O(log N) height using only 2, 3, or 4 children per node

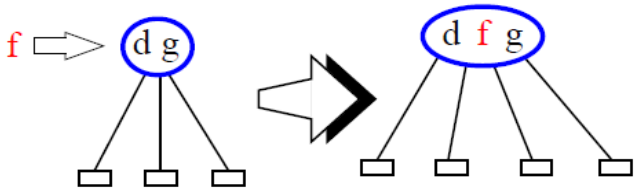
30

Insertion into 2-3-4 Tree

- Insert the new key at the lowest internal node reached in the search
  - ***2-node*** becomes ***3-node***



  - ***3-node*** becomes ***4-node***

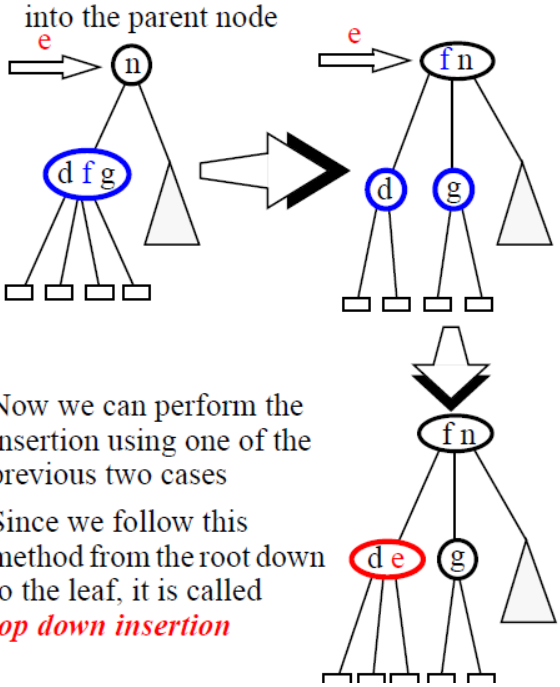


- What about a ***4-node***?
  - We can't insert another key!

31

Top-Down Insertion

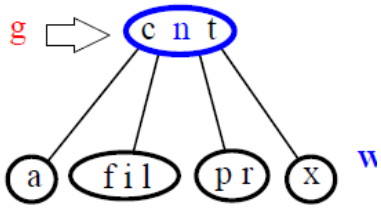
- In our way down the tree, whenever we reach a ***4-node***, we break it up into two ***2-nodes***, and move the middle element up into the parent node

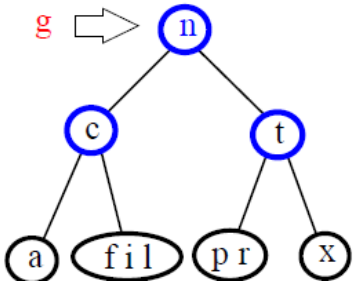


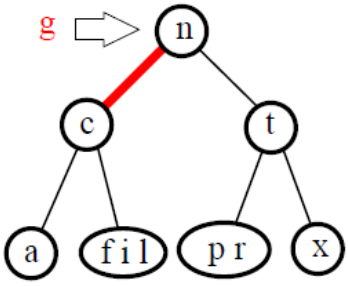
- Now we can perform the insertion using one of the previous two cases
- Since we follow this method from the root down to the leaf, it is called ***top down insertion***

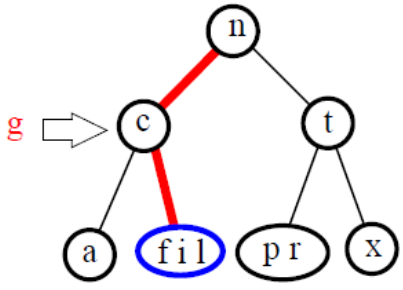
Splitting the Tree

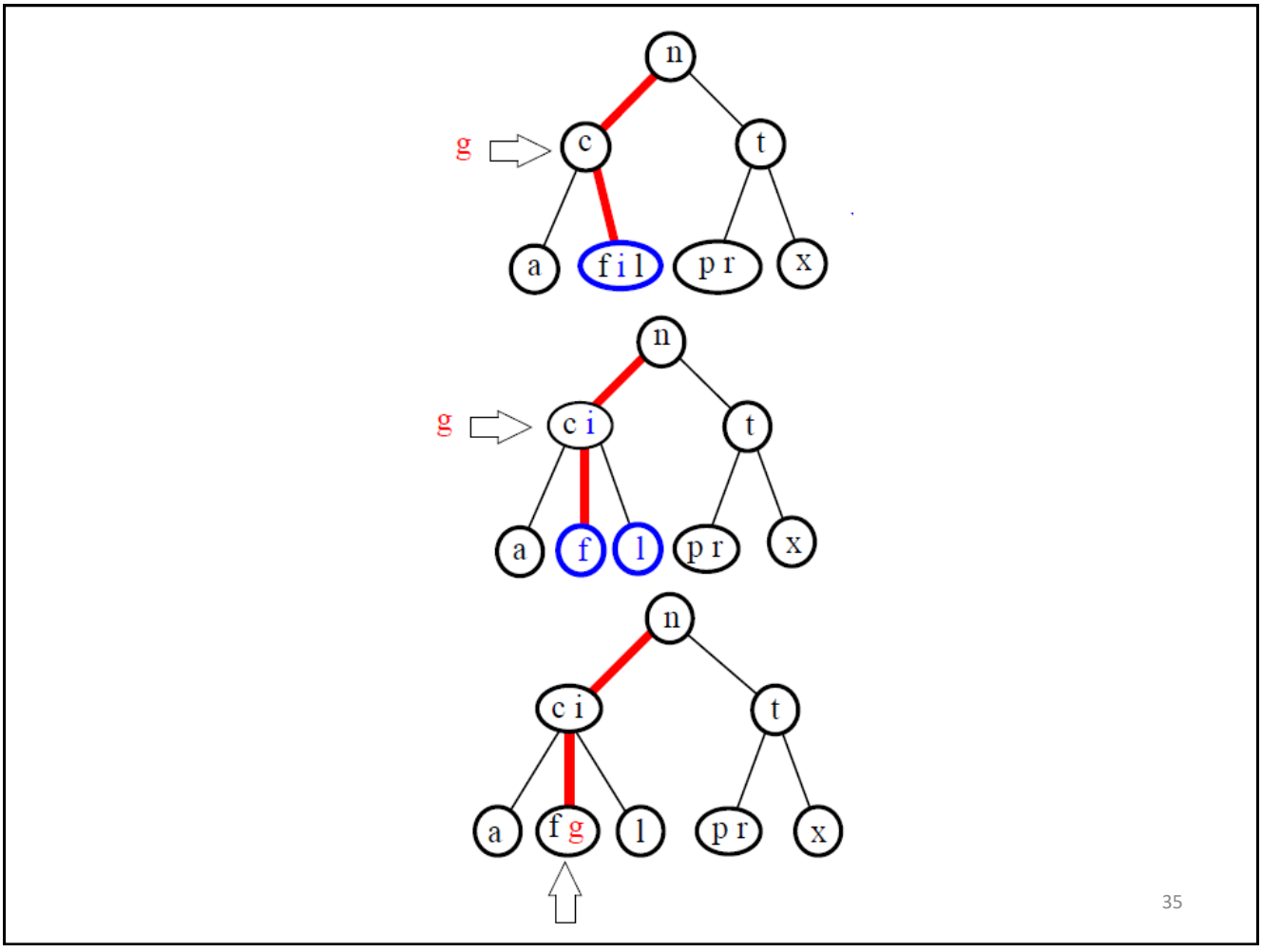
As we travel down the tree, if we encounter any 4-node we will break it up into 2-nodes. This guarantees that we will never have the problem of inserting the middle element of a former 4-node into its parent 4-node.






g
n
c
t
a
f i l
p r
x


n
g
c
t
a
f i l
p r
x



35

Time Complexity of Insertion in 2-3-4 Trees

Time complexity:

- A search visits O(log N) nodes
- An insertion requires O(log N) node splits
- Each node split takes constant time
- Hence, operations Search and Insert each take time O(log N)

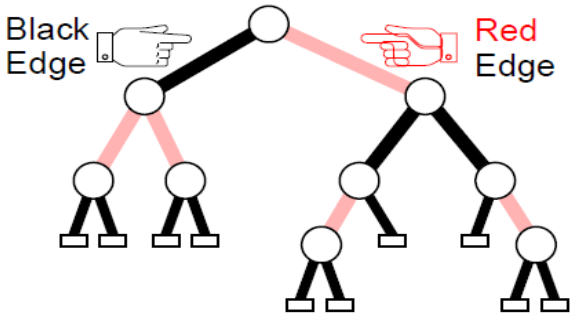
Notes:

- Instead of doing splits top-down, we can perform them bottom-up starting at the insertion node, and only when needed. This is called bottom-up insertion.
- A deletion can be performed by fusing nodes (inverse of splitting), and takes O(log N) time

Red-Black Tree

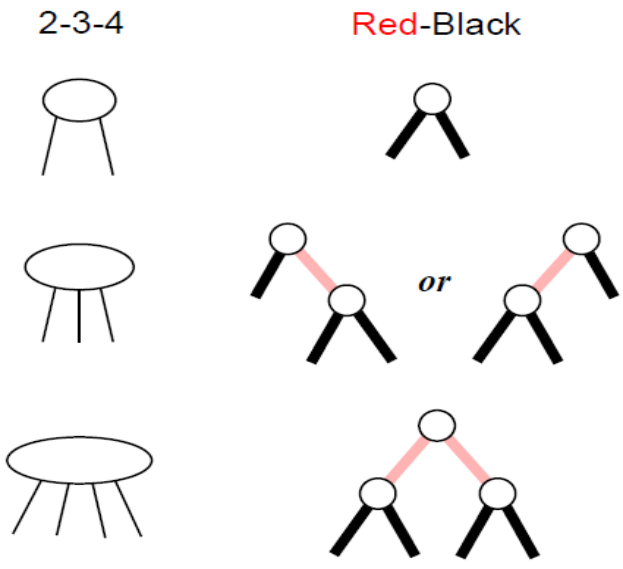
A **red-black tree** is a binary search tree with the following properties:

- edges are colored **red** or **black**
- ***no two consecutive red edges*** on any root-leaf path
- ***same number of black edges*** on any root-leaf path (= ***black height*** of the tree)
- ***edges connecting leaves are black***



2-3-4 Tree Evolution

Note how **2-3-4 trees** relate to **red-black trees**

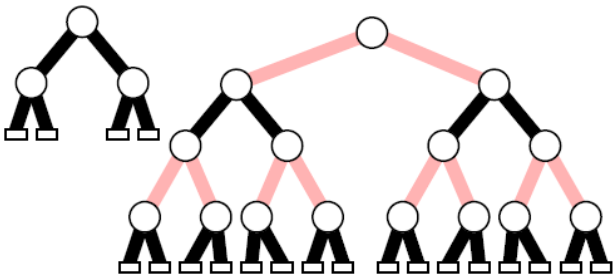


Now we see red-black trees are just a way of representing 2-3-4 trees!

More Red-Black Tree Properties

N # of internal nodes
L # leaves (= N + 1)
H height
B black height

Property 1: $2^B \le N + 1 \le 4^B$



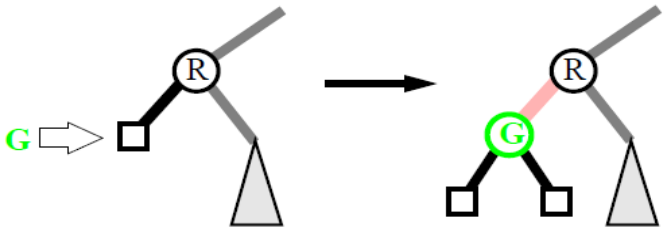
Property 2: $\frac{1}{2}\log(N+1) \le B \le \log(N+1)$

Property 3: $\log(N+1) \le H \le 2\log(N+1)$

This implies that searches take time O(logN)!

Insertion into Red-Black Trees

1. Perform a standard search to find the leaf where the key should be added
2. Replace the leaf with an internal node with the new key
3. Color the incoming edge of the new node red
4. Add two new leaves, and color their incoming edges black
5. If the parent had an incoming red edge, we now have two consecutive red edges! We must reorganize tree to remove that violation. What must be done depends on the sibling of the parent.

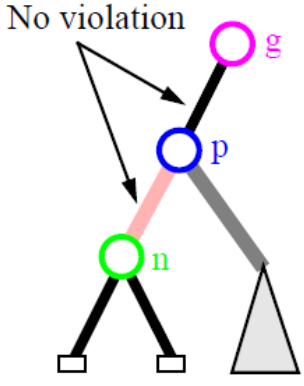


Insertion - Plain and Simple

Let:

- n be the new node
- p be its parent
- g be its grandparent

Case 1: Incoming edge of p is black



STOP!

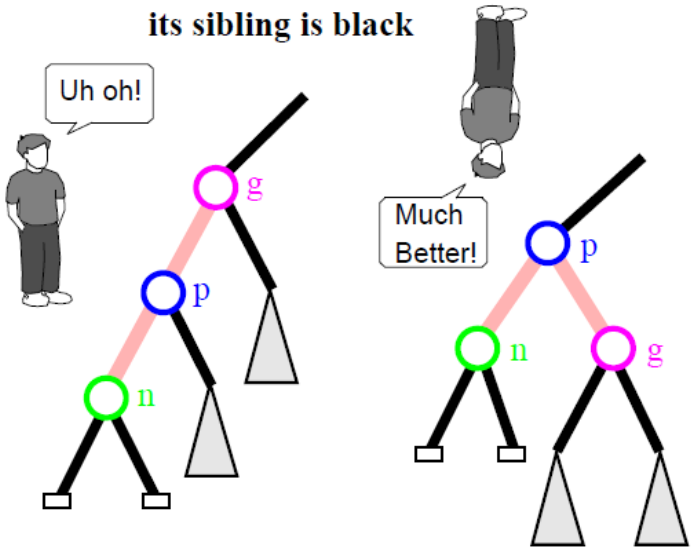
Pretty easy, huh?

Well... it gets messier...

41

Restructuring

Case 2: Incoming edge of p is red, and its sibling is black

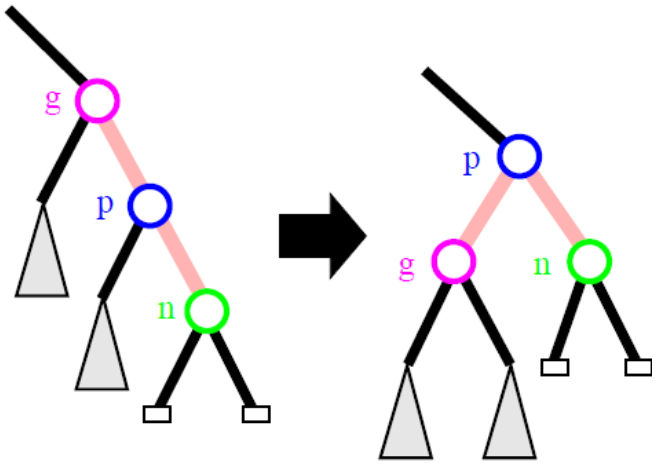


We call this a "*right rotation*"

- No further work on tree is necessary
- Inorder remains unchanged
- Tree becomes more balanced
- No two consecutive red edges!

More Rotations

Similar to a right rotation, we can do a "**left rotation**"...

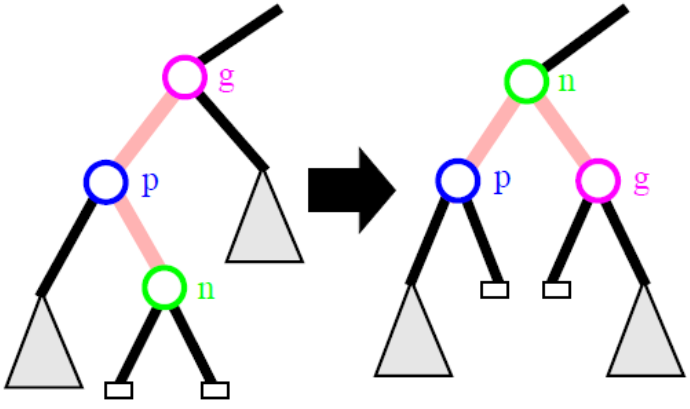


Simple, huh?

Double Rotations

What if the new node is between its parent and grandparent in the inorder sequence?

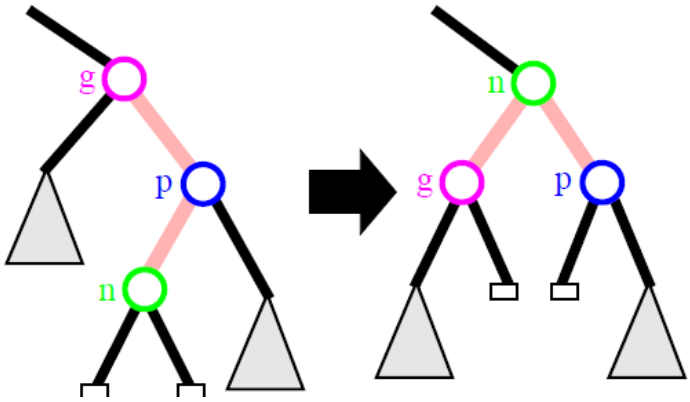
We must perform a "double rotation"
(which is no more difficult than a "single" one)



This would be called a
"left-right double rotation"

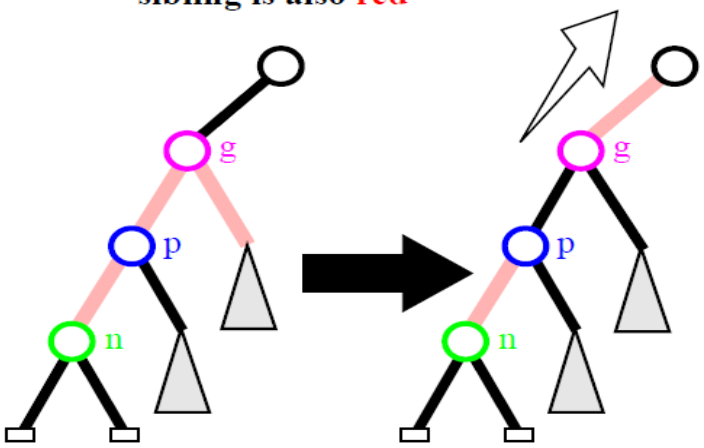
Last of the Rotations

And this would be called a
"right-left double rotation"



Bottom-Up Rebalancing

Case 3: Incoming edge of p is red and its sibling is also red

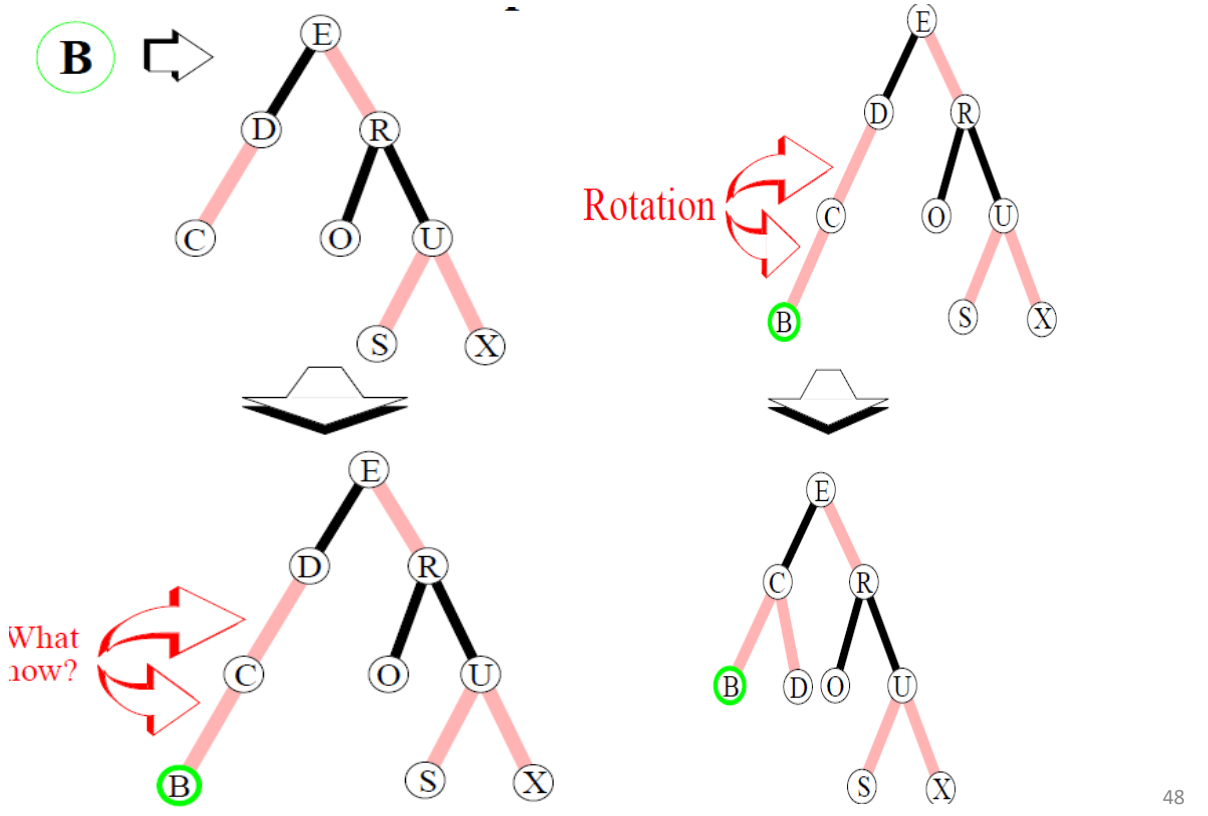


- We call this a "***promotion***"
- Note how the black depth remains unchanged for all of the descendants of **g**
- This process will continue upward beyond **g** if necessary: rename **g** as **n** and repeat.

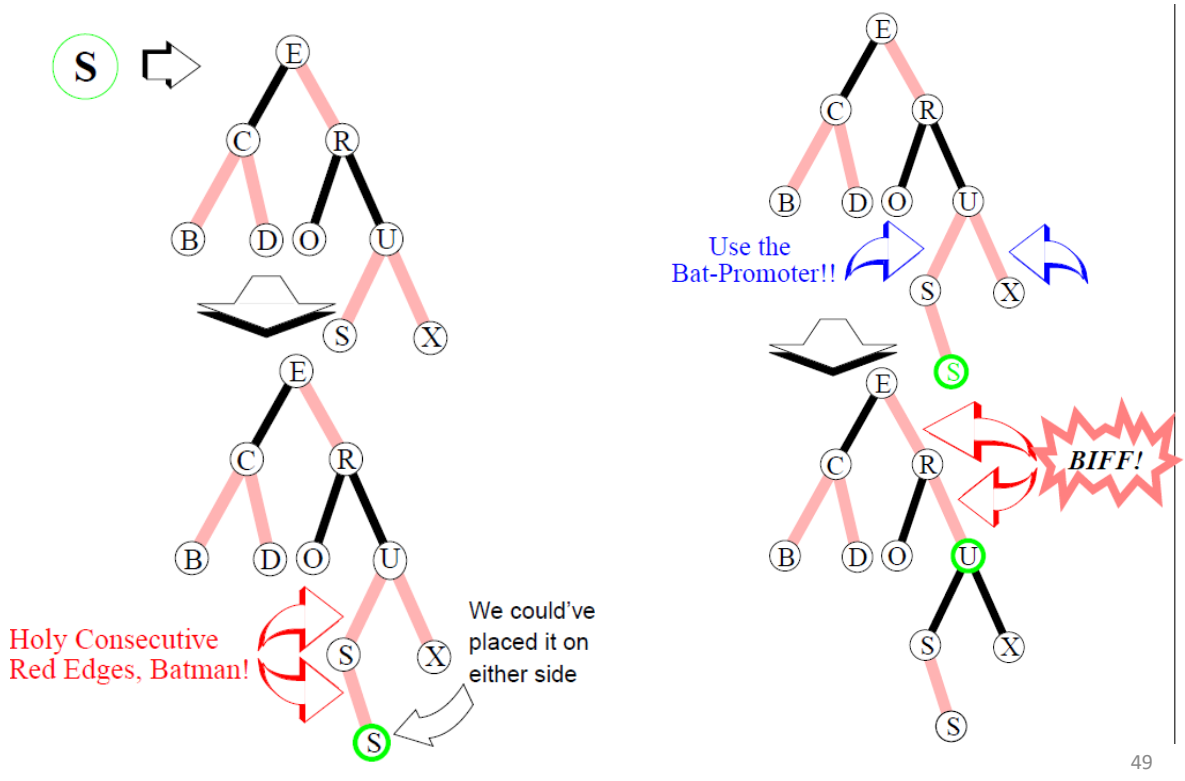
Summary of Insertion

- If two red edges are present, we do either
  - a ***restructuring*** (with a simple or double rotation) and **stop**, or
  - a ***promotion*** and **continue**
- A restructuring takes constant time and is performed at most once. It reorganizes an off-balanced section of the tree.
- Promotions may continue up the tree and are executed O(log N) times.
- The **time complexity** of an insertion is **O(logN).**

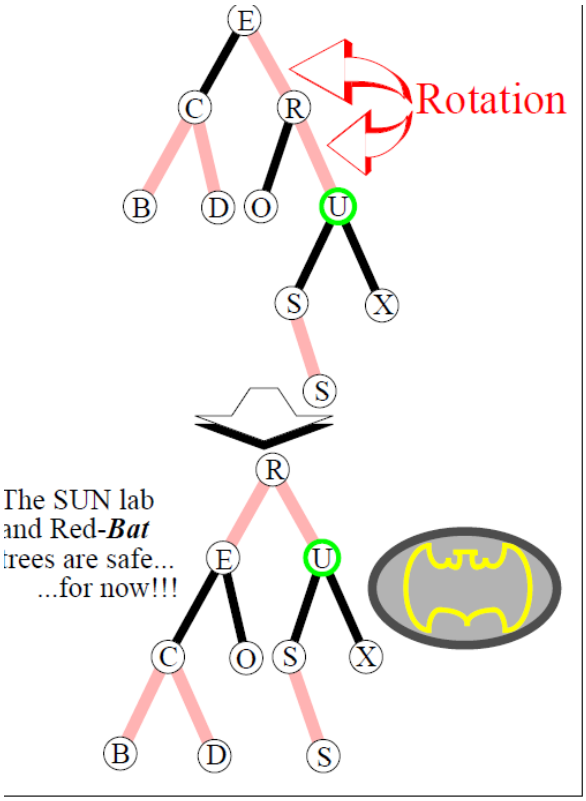
Examples


B
E
D
R
C
O
U
S
X
Rotation
E
D
R
C
O
U
B
S
X
What now?
E
D
R
C
O
U
B
S
X
E
C
R
B
D
O
U
S
X
48

A Super Example



A Super Example contd.



50