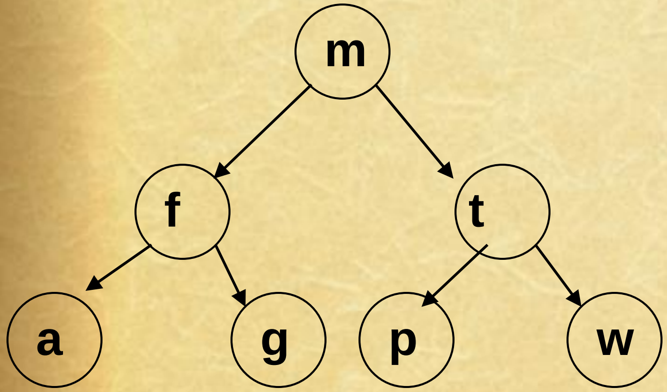
Dynamic data structures for searching

Binary search trees

- Let K_1, K_2, ..., K_n be n distinct keys in ascending order.
- Let T be a binary tree with n nodes, and let N_i be the i'th node visited in a symmetric order traversal of the tree (left, root, right).
- If we store K_i in N_i, then the binary tree has the **binary search property**: At each node N_i, with key K_i, all nodes in the left subtree of N_i have keys less than K_i and all nodes in the right subtree of N_i have keys greater than K_i.

Binary search trees



- To **search** a binary search tree for a key K:

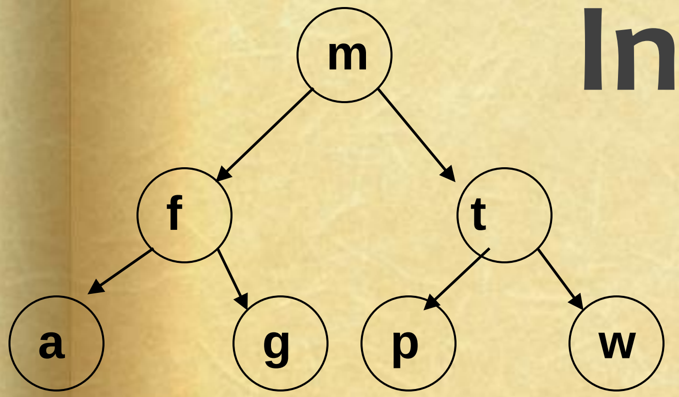
  1) If K matches the key at the root, done.

  2) If K is less than the key at the root, search the left subtree

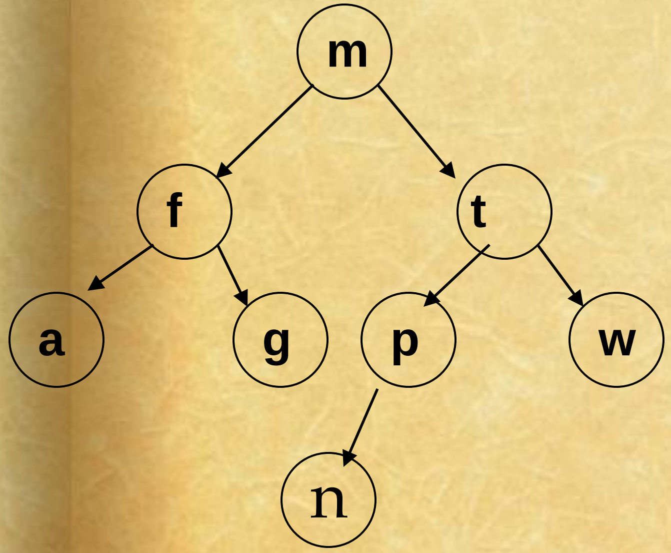
  3) Otherwise, search the right subtree.

  4) Reaching a nil link ends in failure.

Insertion into a BST

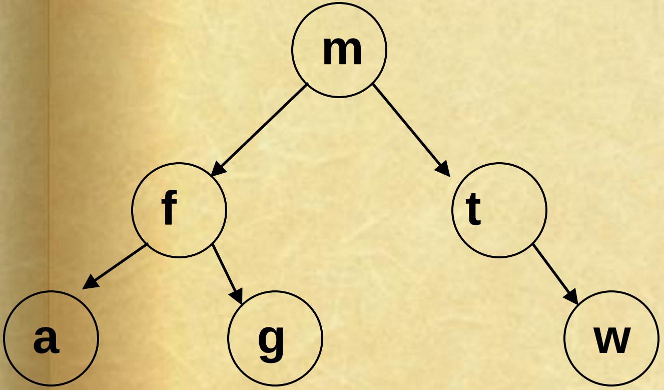


insert n

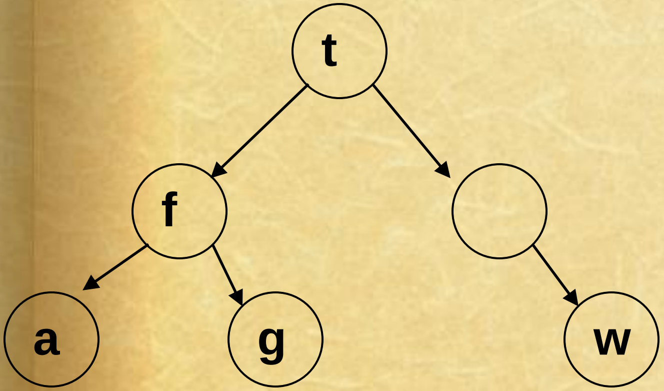


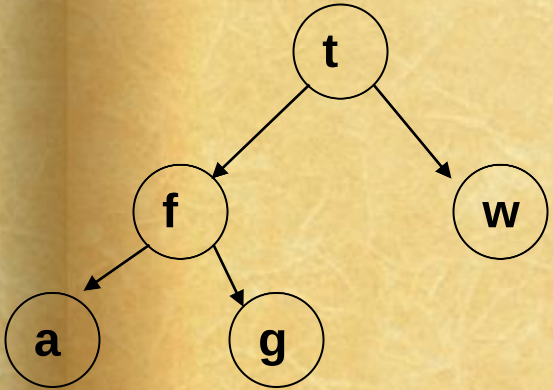
- To insert (K)
  - search for K
  - if search fails, it will fail at a leaf node.
  - insert (K) as the appropriate son of the leaf node at which search fails
- An unlucky sequence of insertions can inbalance the tree badly

Deletion from a BST



delete m



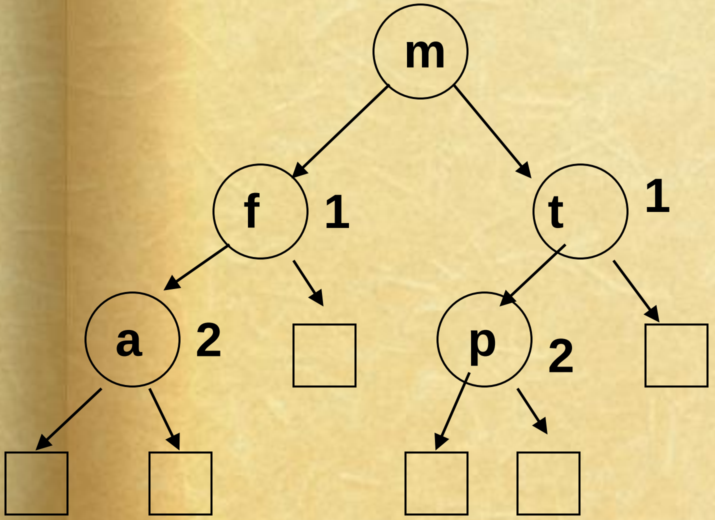


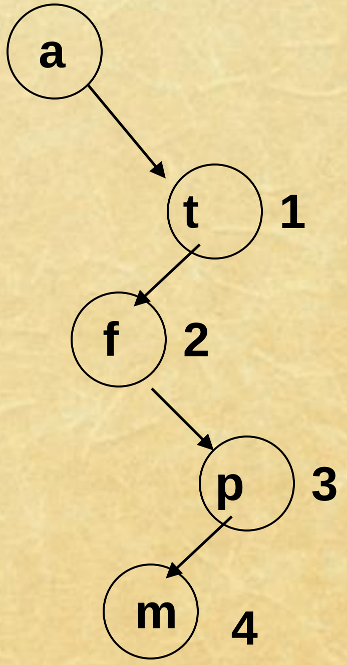
- Search(K) to find location of item to be deleted.
  - Leaf node - delete immediately
  - Node with one child - replace with child and delete empty child node
  - Node with two children:
    - replace with smallest node in the right subtree
    - then recursively delete the replaced node
    - because this is the smallest node, it cannot have a left son, so deleting it is easy

Good and bad binary search trees

- Given a set of keys, there exist many binary search trees that can be built based on those keys.
- Define the **internal path length** of a BST to be the sum of the lengths of the paths from the root to each node. Every successful search will follow one of these paths.
- We can also define an **external path length** by appending dummy square nodes where empty subtrees occur (these are places where unsuccessful searches terminate) and computing the sum of the lengths of the paths to these dummy nodes.

Good and bad BST's





- Internal path length = 6
- external path length = 16
- average successful search = 6/5
- average unsuccessful search = 16/6

internal path length = 10

external path length = 1 +2+3+4+5+5 = 20

Good and bad BST's

- Let's assume that all of the keys in the BST are equally likely candidates for searches

- What is the best BST?
  - The one with minimum internal path length
  - Complete binary tree which has an internal path length of logn +1

- What is the worst BST?
  - One with linear structure
  - Internal path length is n(n+1)/2

- How well does a random BST do?
  - About 1.386 log n +1 - not much worse than optimal!

Dynamic BST's

- Static BST's
  - Suppose keys do not have equal probability of being searched for, but the probabilities are known. Can we construct an optimal BST?
- Dynamic BST's
  - Suppose keys can be inserted and deleted in a BST. How can we keep a BST balanced as keys are added and removed?
  - If we are willing to keep the BST "almost balanced" are better algorithms available? (AVL trees, splay trees and B-trees)

AVL Trees

- Define the left and right height of a nonempty binary tree, T as follows:

  Leftheight(T) = 0 if LSON(T) = Nil

  1 + height(LSON(T)) otherwise

  Rightheight(T) = ...

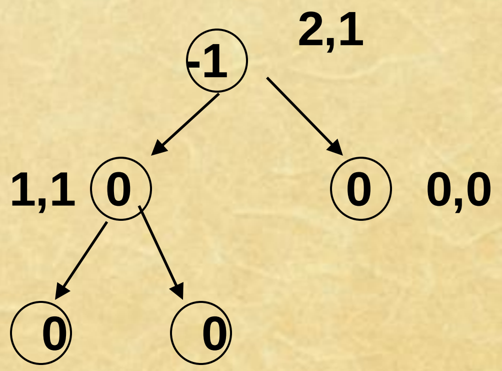
- Height of a node is the maximum of its Leftheight and Rightheight.

- The balance of a node is Rightheight-Leftheight

- T is an AVL tree if every node has balance +1, 0, -1

AVL Trees

- Every AVL tree with n nodes has height O(logn)
  - so all successful and unsuccessful searches take O(logn) time
- A node can be added to or deleted from an AVL tree with n nodes in time O(logn) while preserving the AVL property



Insertion into AVL trees

- Representation of AVL trees
  - add a balance field to each node in a binary tree
  - two bits are sufficient to encode the possible balances of +1, 0, -1

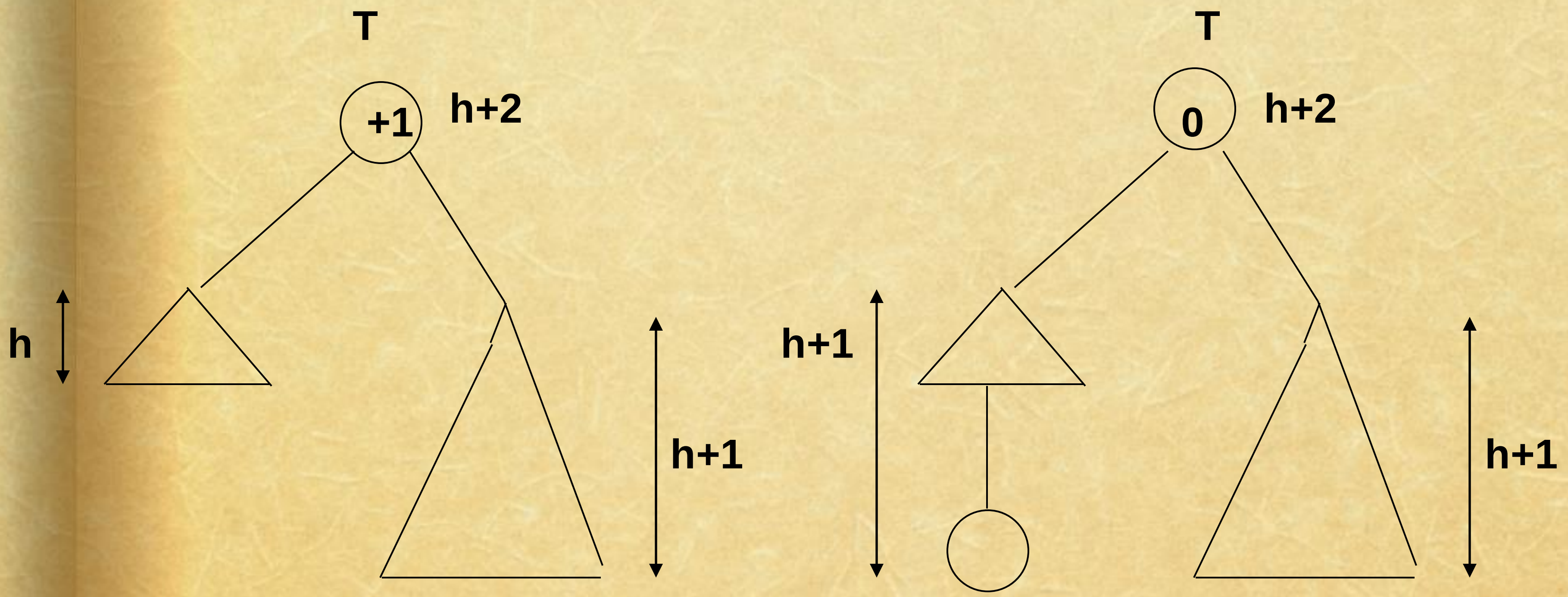
- General insertion algorithm:

  1) Using the binary tree insertion method, trace a path from the root and insert the new node as a leaf. Remember the path

  2) Retrace the path towards the root, updating the balance factors

  3)When encountering a node for which the balance becomes +2 or -2, readjust the subtrees of that node and its descendants to obtain a BST with AVL balances

Insertion into AVL trees



- A node that was out of balance becomes perfectly balanced
- All ancestors are OK since height of node has not changed, and only its height affects ancestor balance

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

Inside Cases (requires double rotation) :

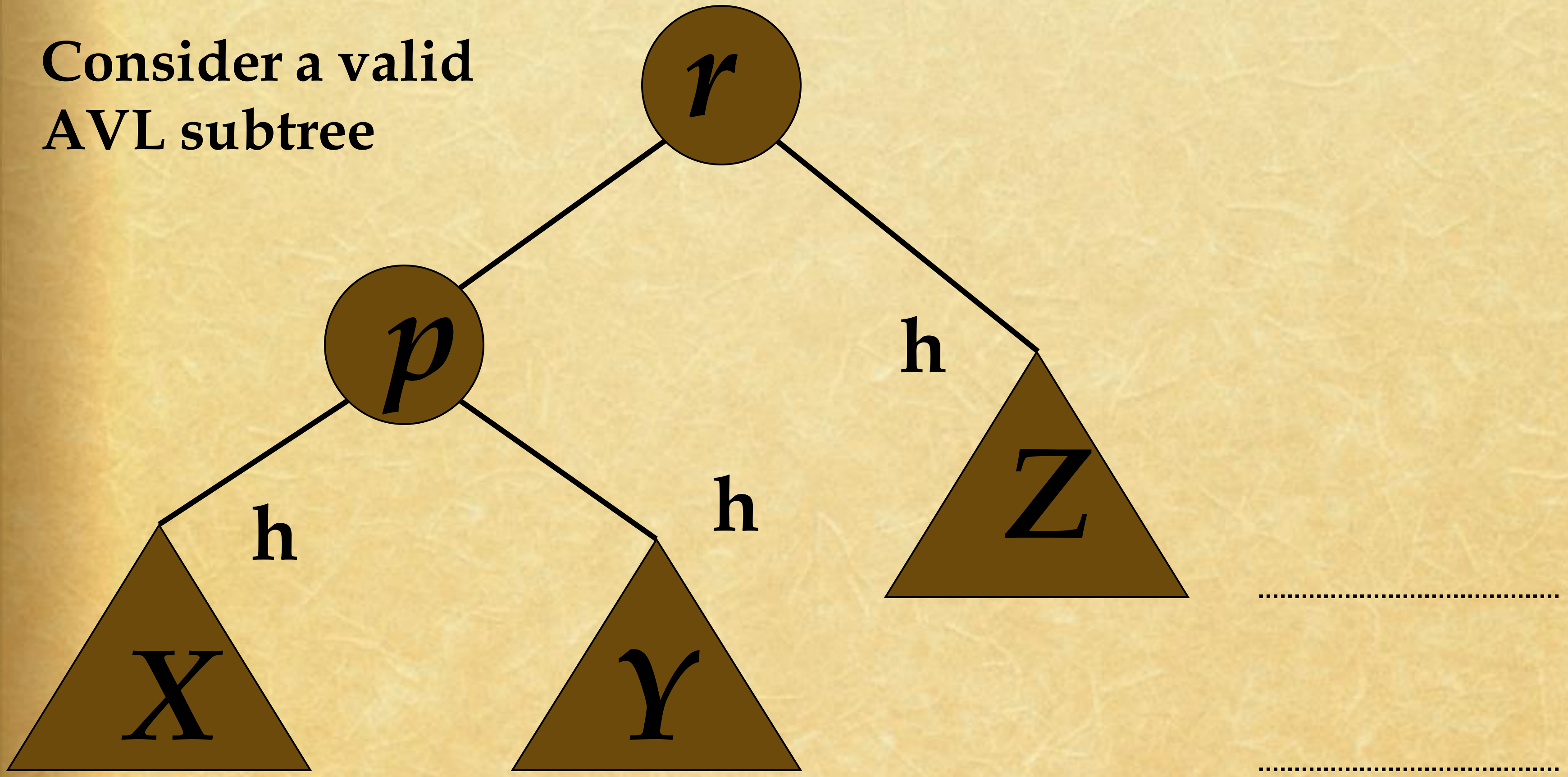
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

The rebalancing is performed through four separate rotation algorithms.

courses.cs.washington.edu/courses/cse373/04wi/sl ides/lecture08.pp

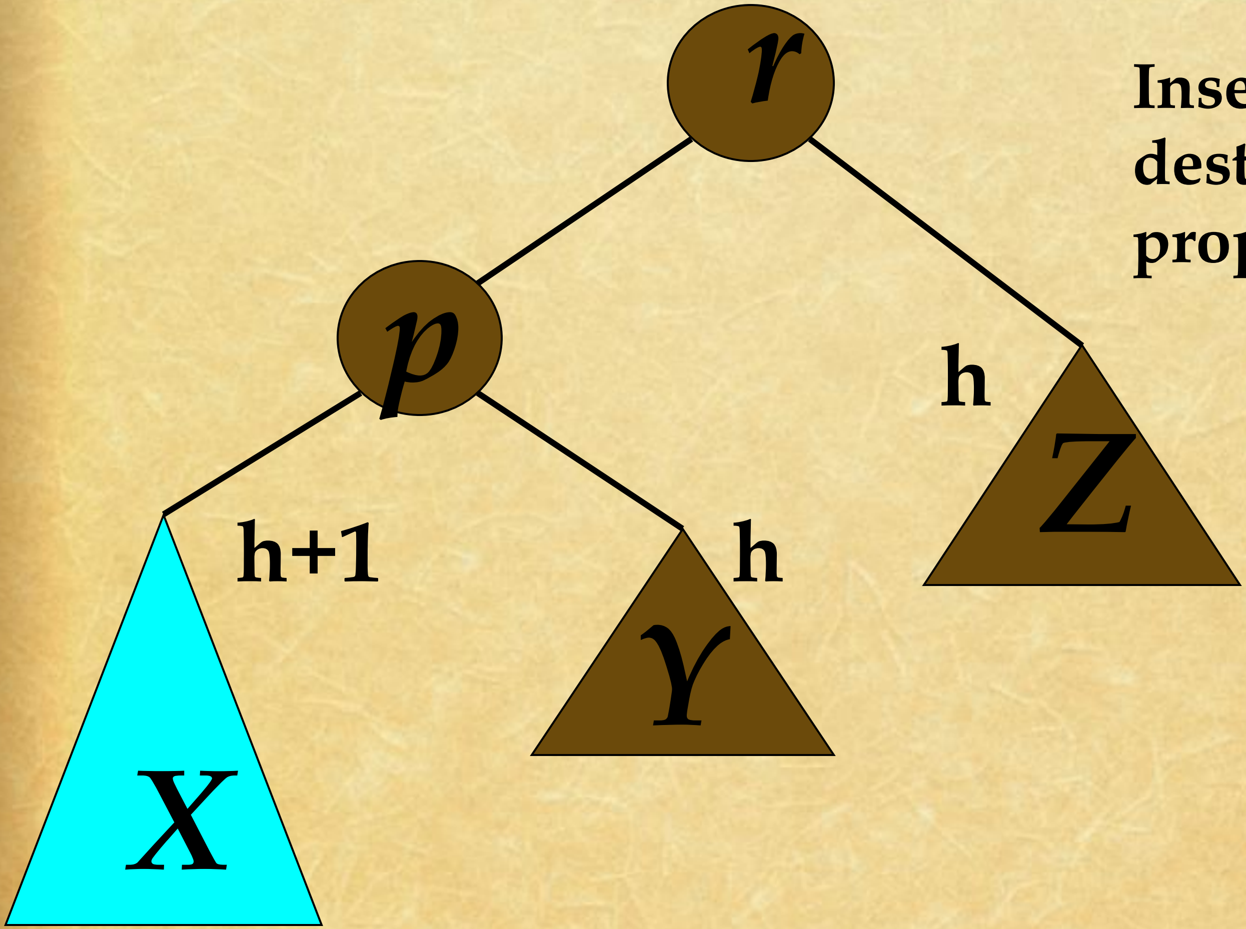
AVL Insertion: Outside Case

Consider a valid AVL subtree



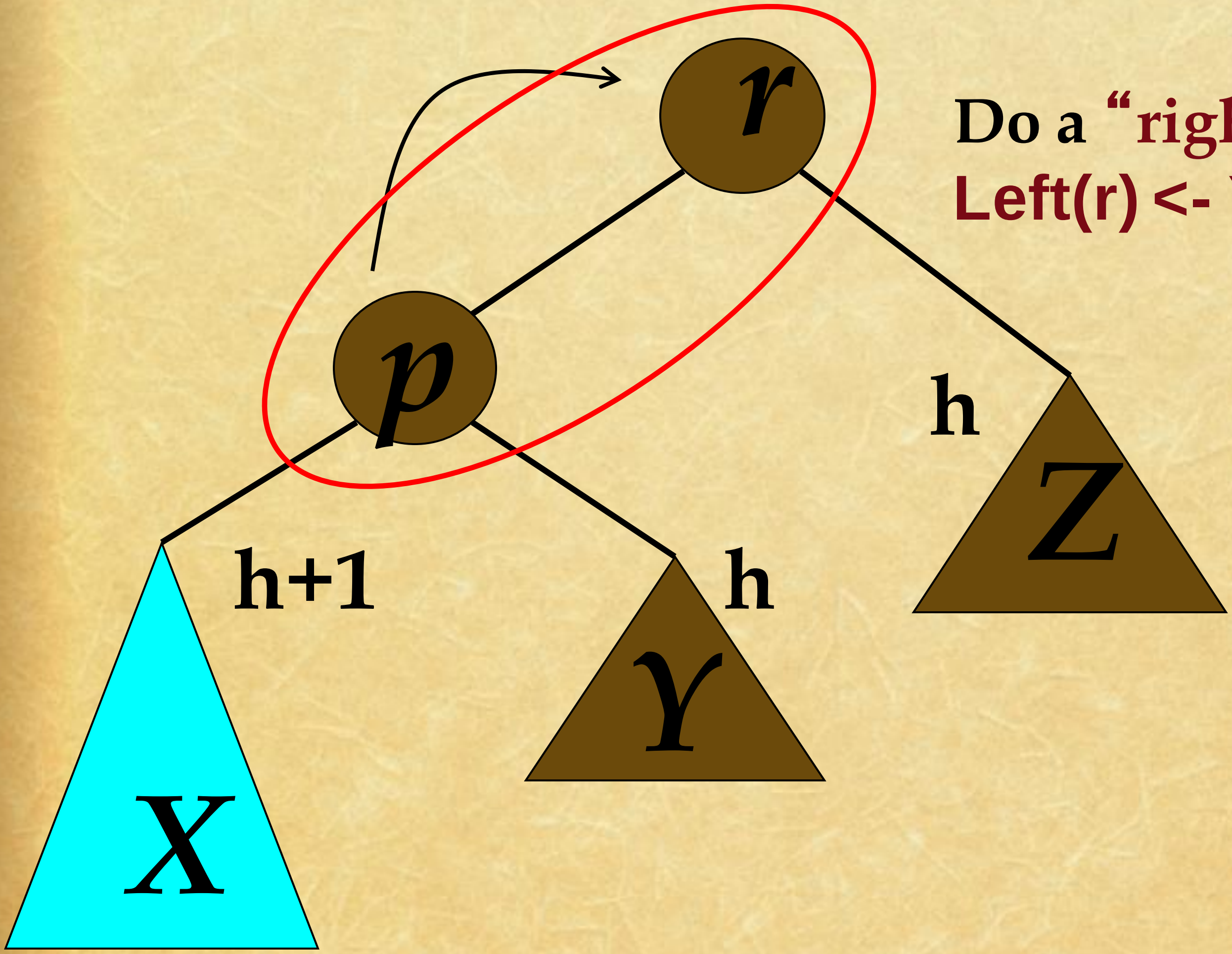
courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

AVL Insertion: Outside Case



Inserting into X destroys the AVL property at node r

AVL Insertion: Outside Case

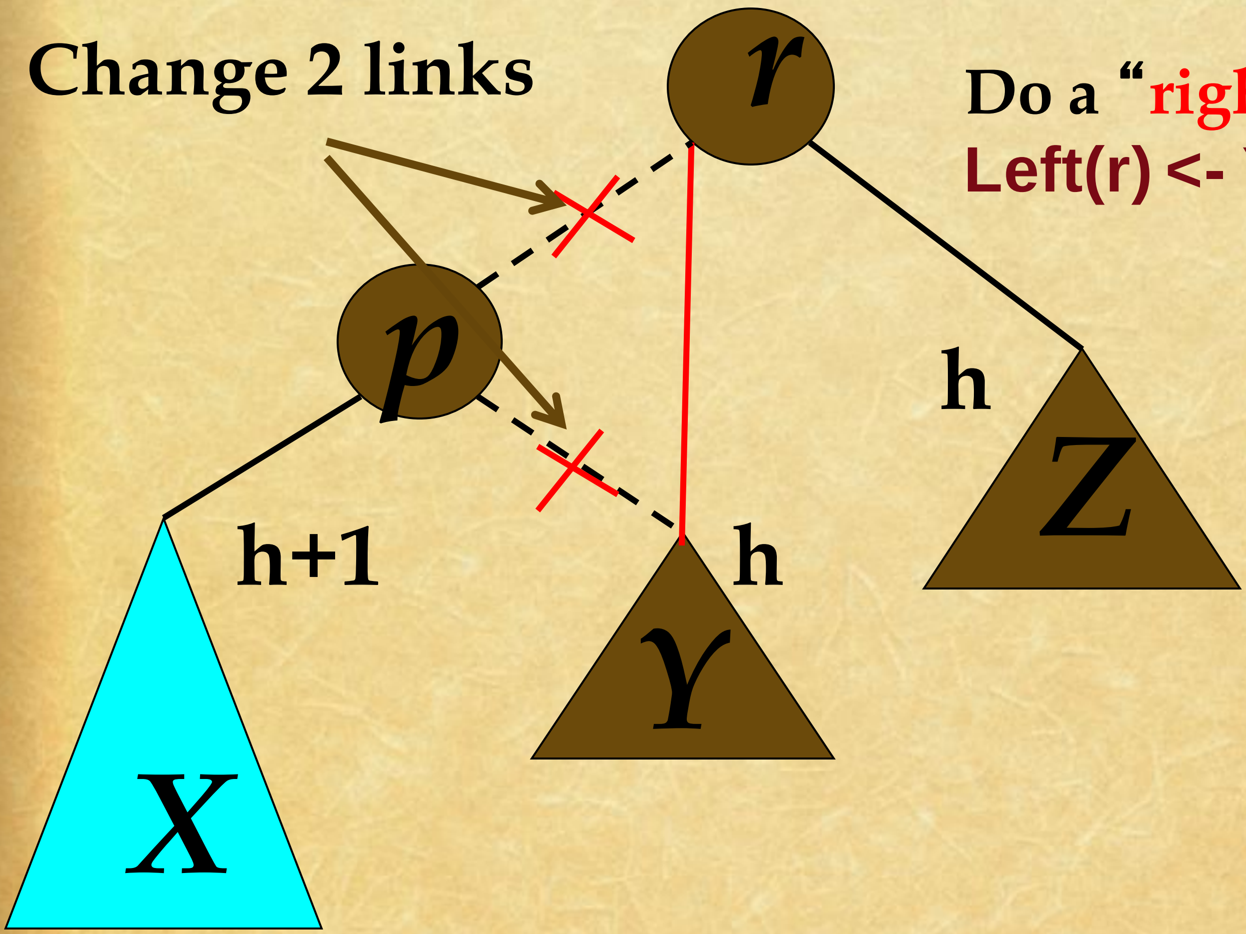


Do a "right rotation"

Left(r) <- Y

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Single right rotation

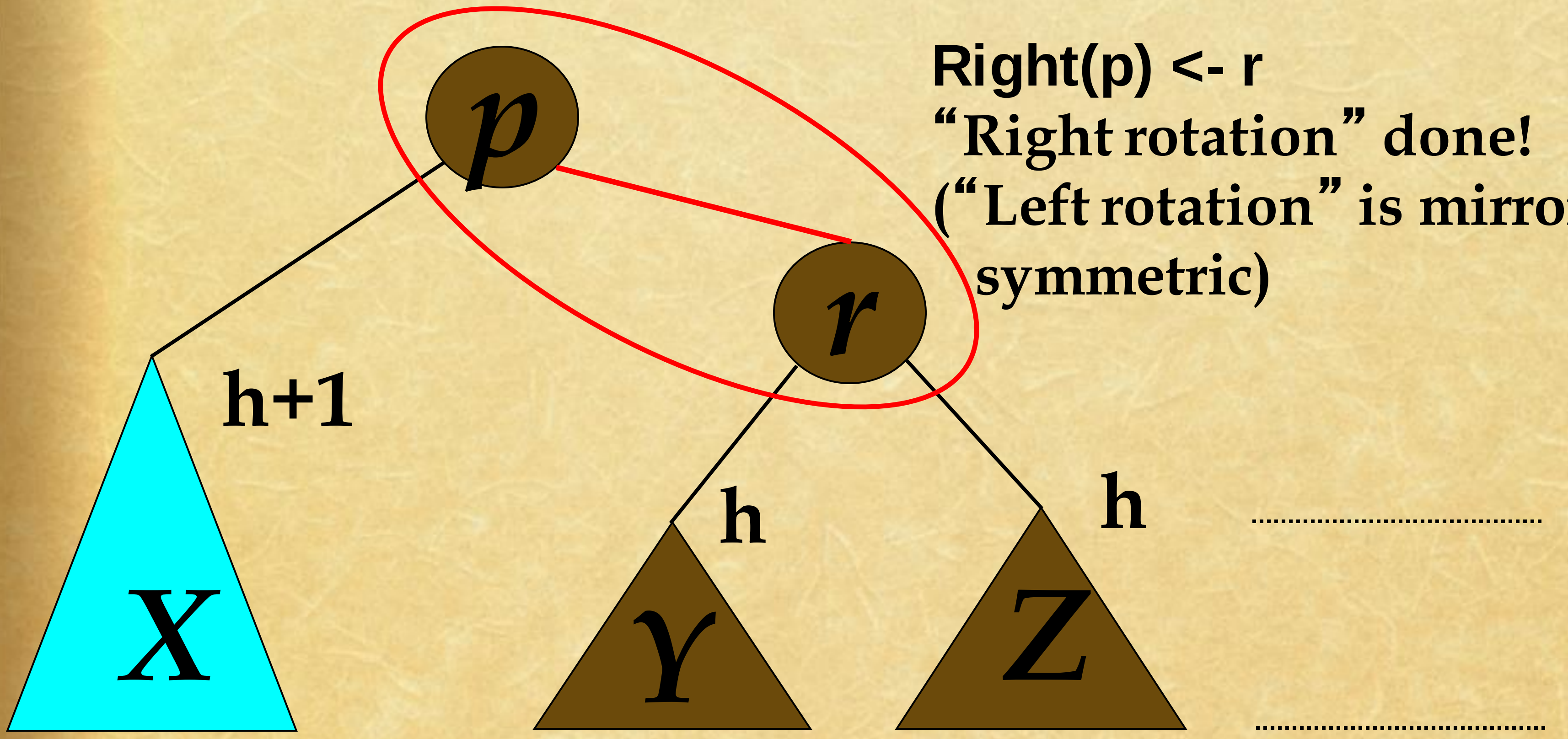


Do a "right rotation"

Left(r) <- Y

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Outside Case Completed

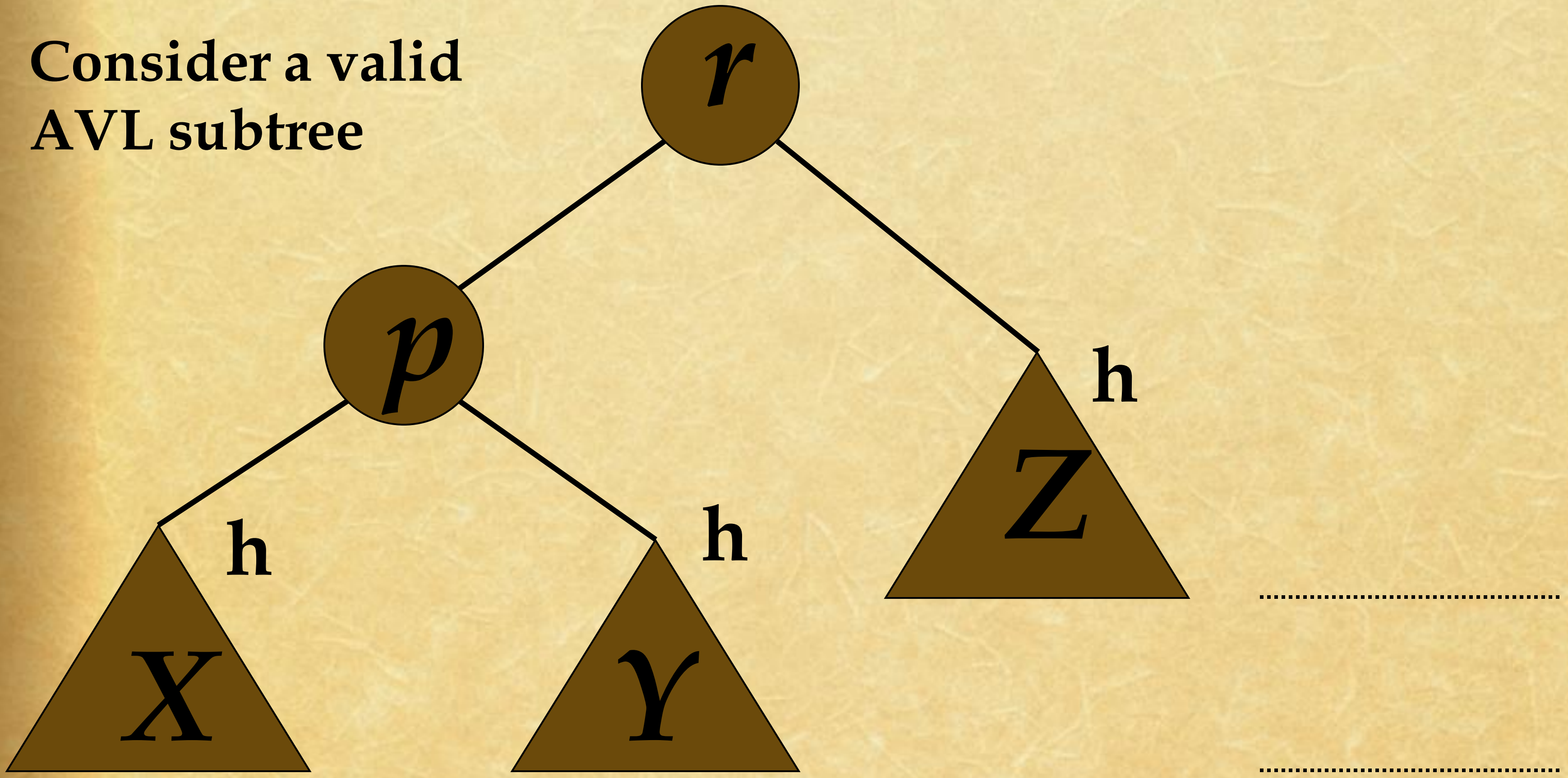


Right(p) <- r
"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored
BST properties still hold

AVL Insertion: Inside Case

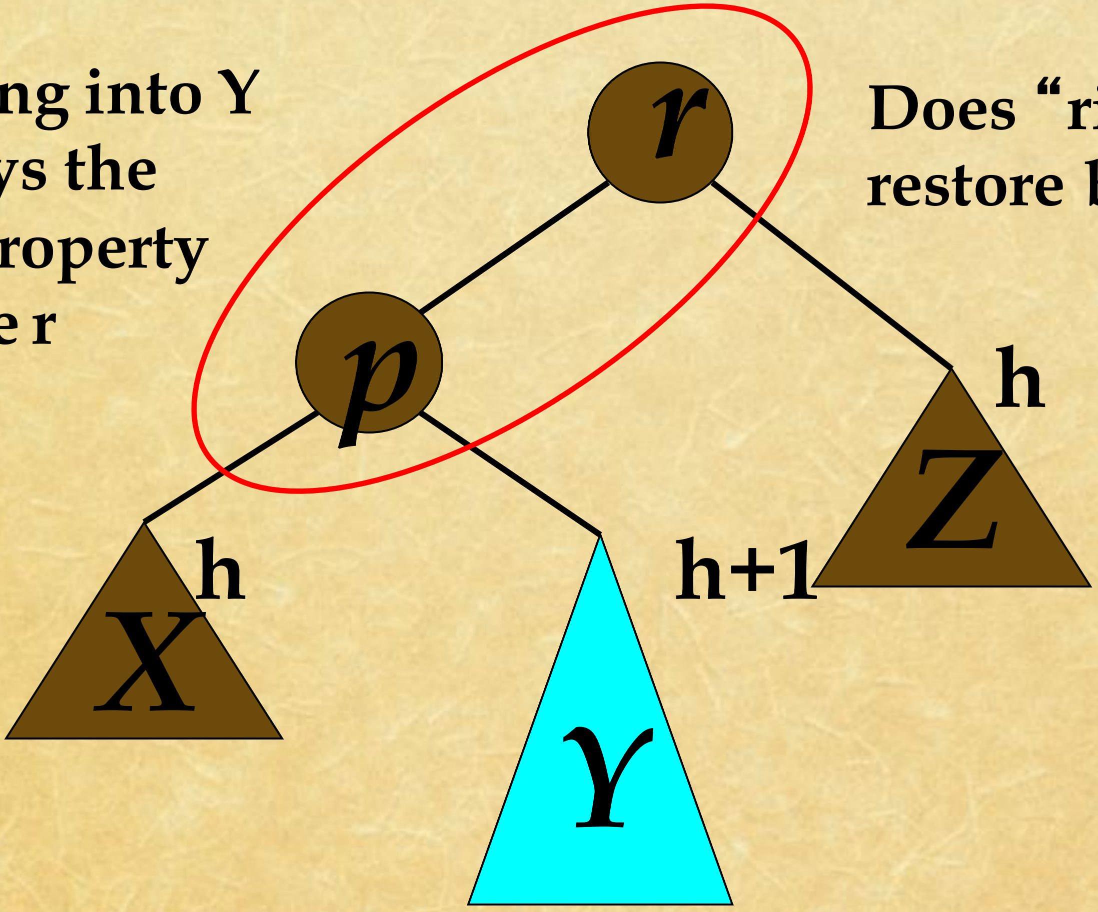
Consider a valid AVL subtree



courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node r



Does "right rotation" restore balance?

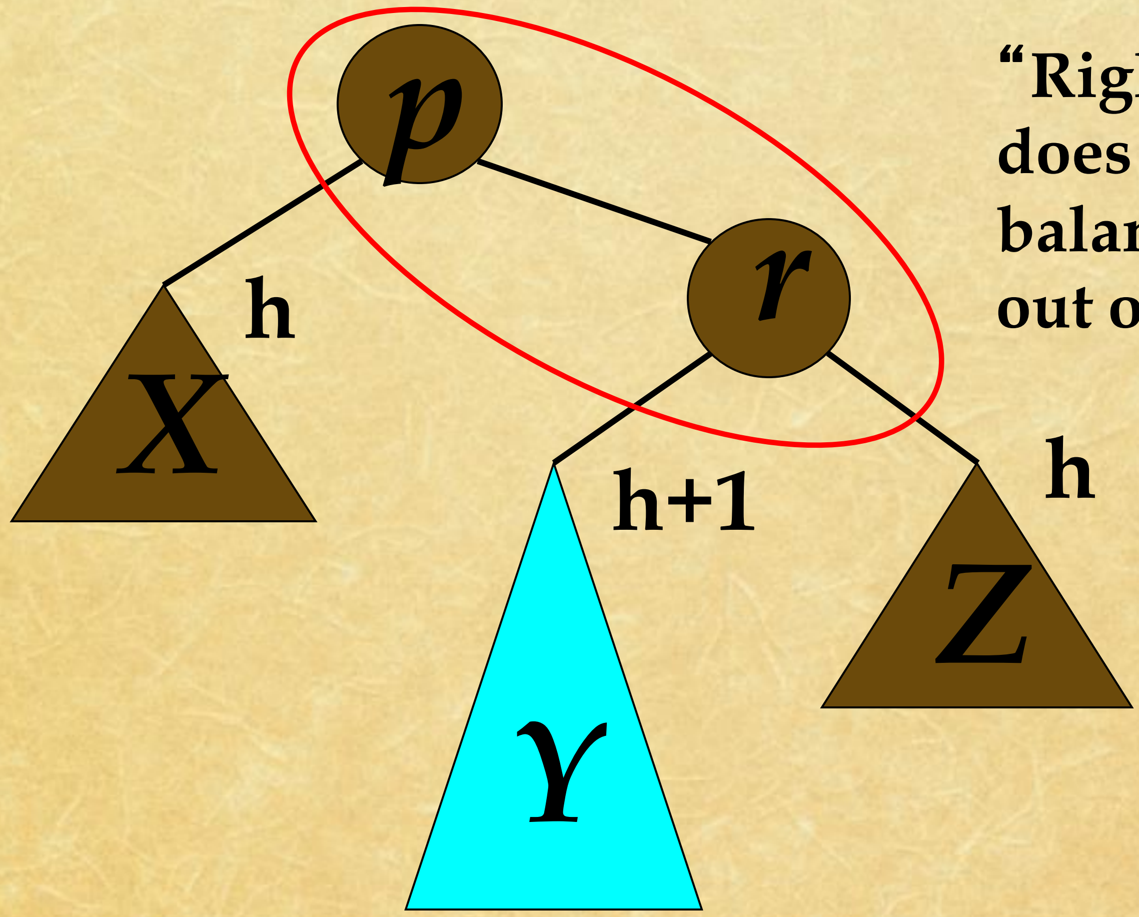
………………………………

………………………………

………………………………

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

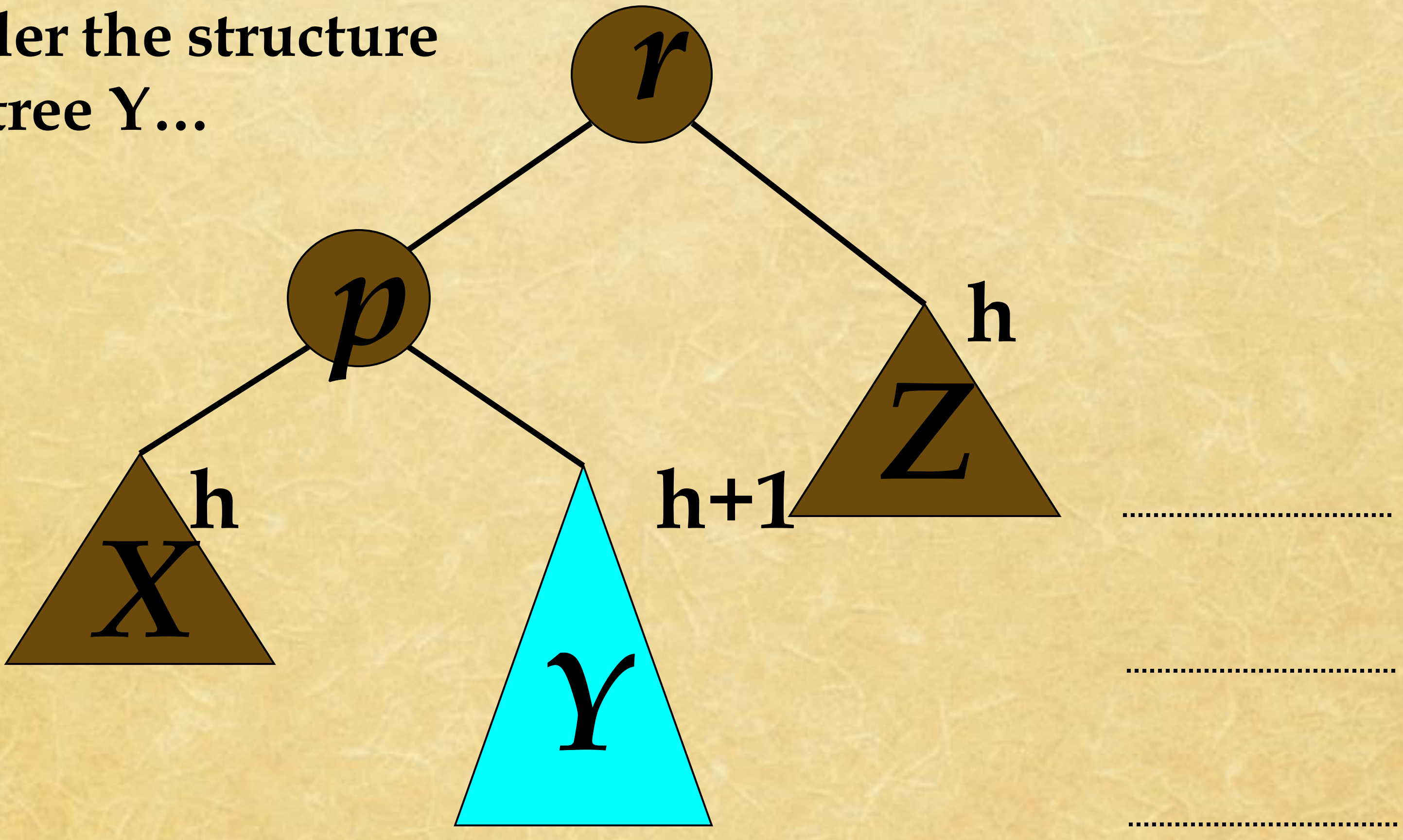
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now p is out of balance

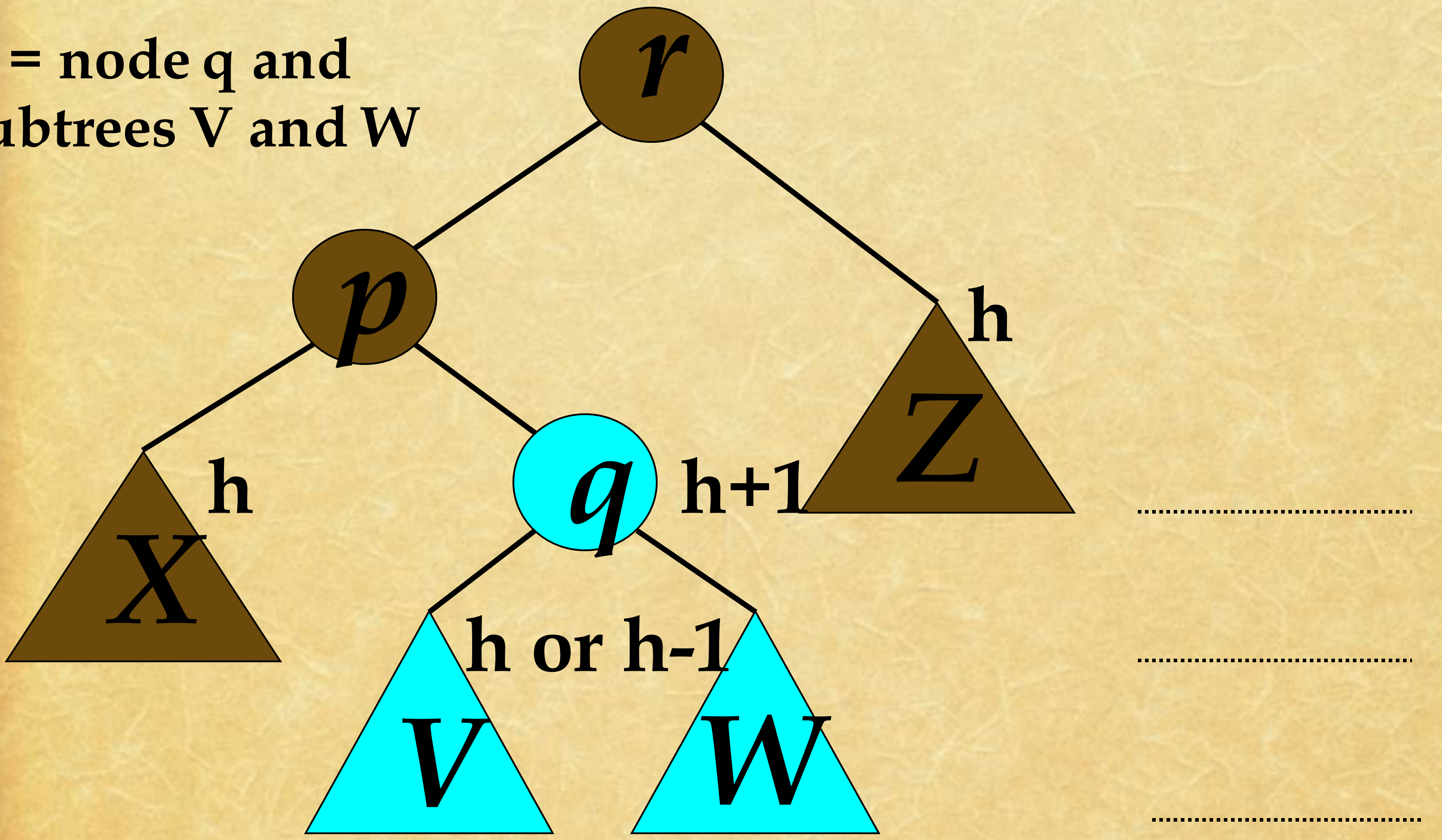
AVL Insertion: Inside Case

Consider the structure of subtree Y…

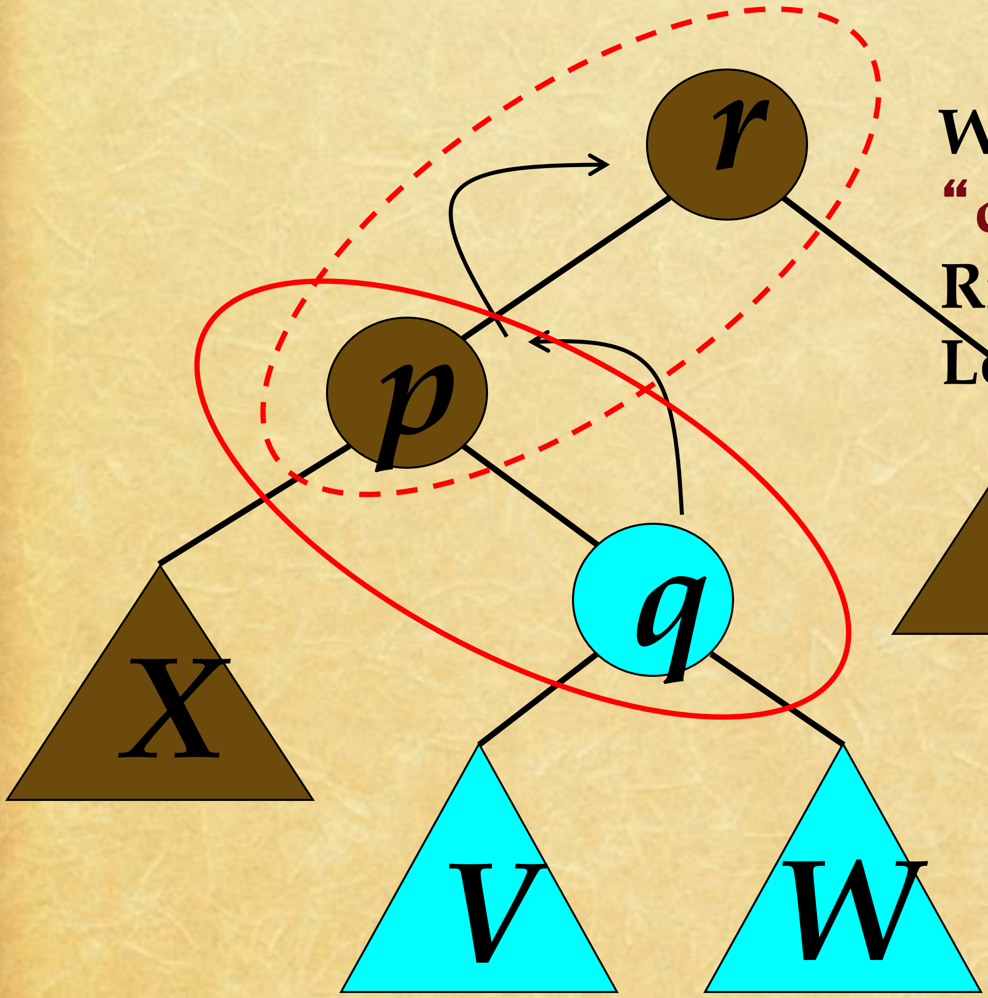


AVL Insertion: Inside Case

Y = node q and subtrees V and W



AVL Insertion: Inside Case



We will do a left-right "double rotation" . . .

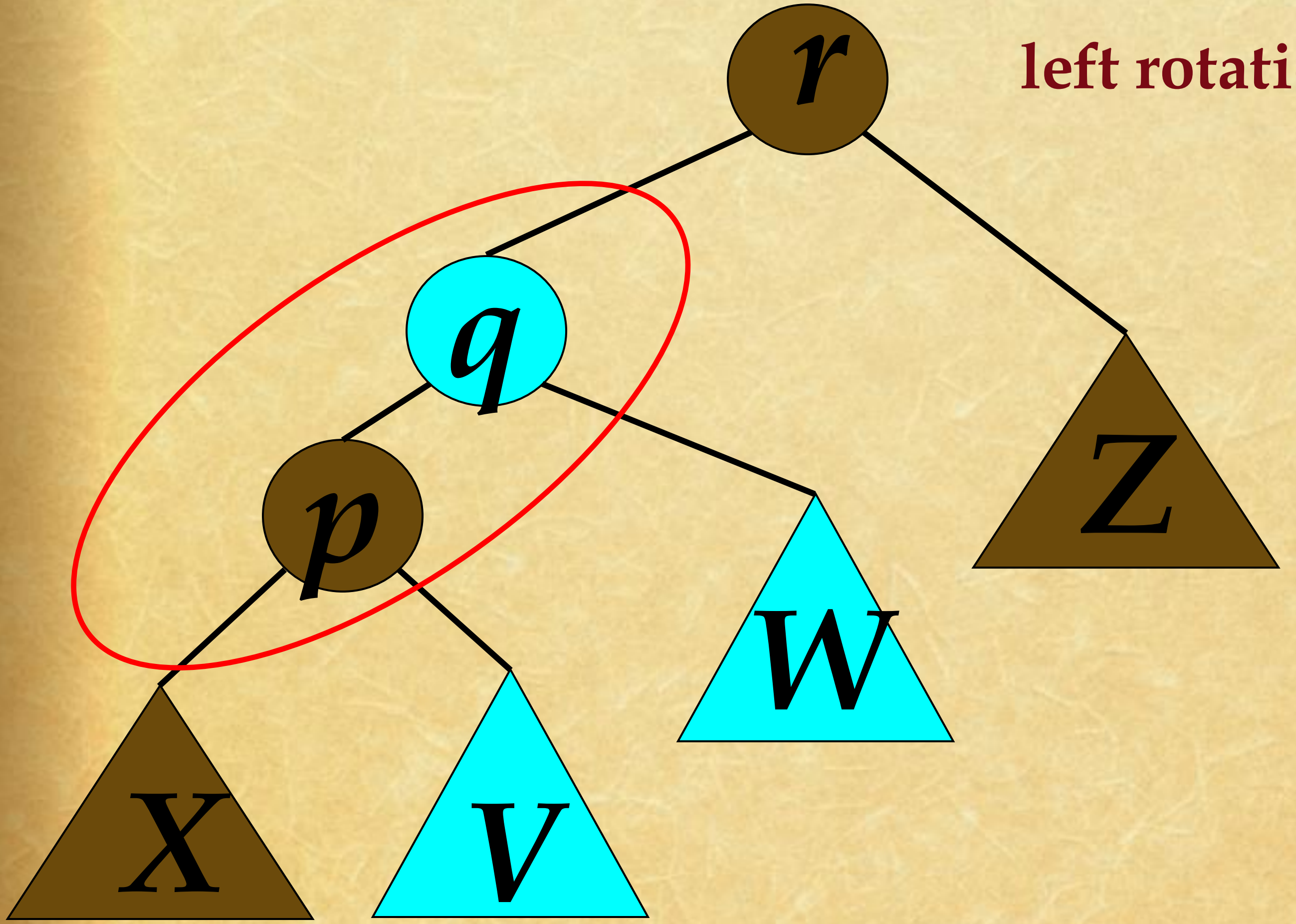
Right(p) <- V

Left(q) <- p

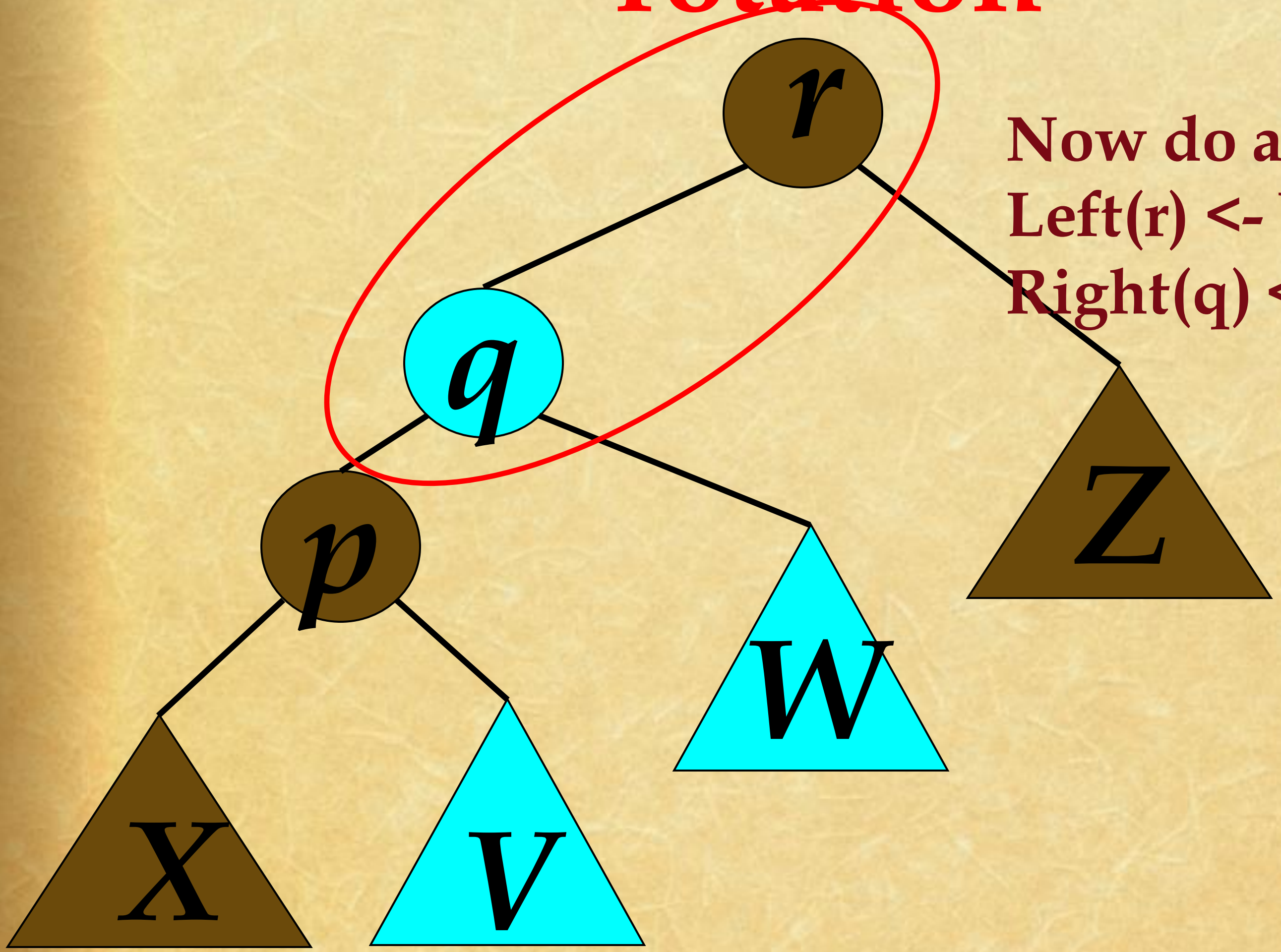
courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Double rotation : first rotation

left rotation complete



Double rotation : second rotation



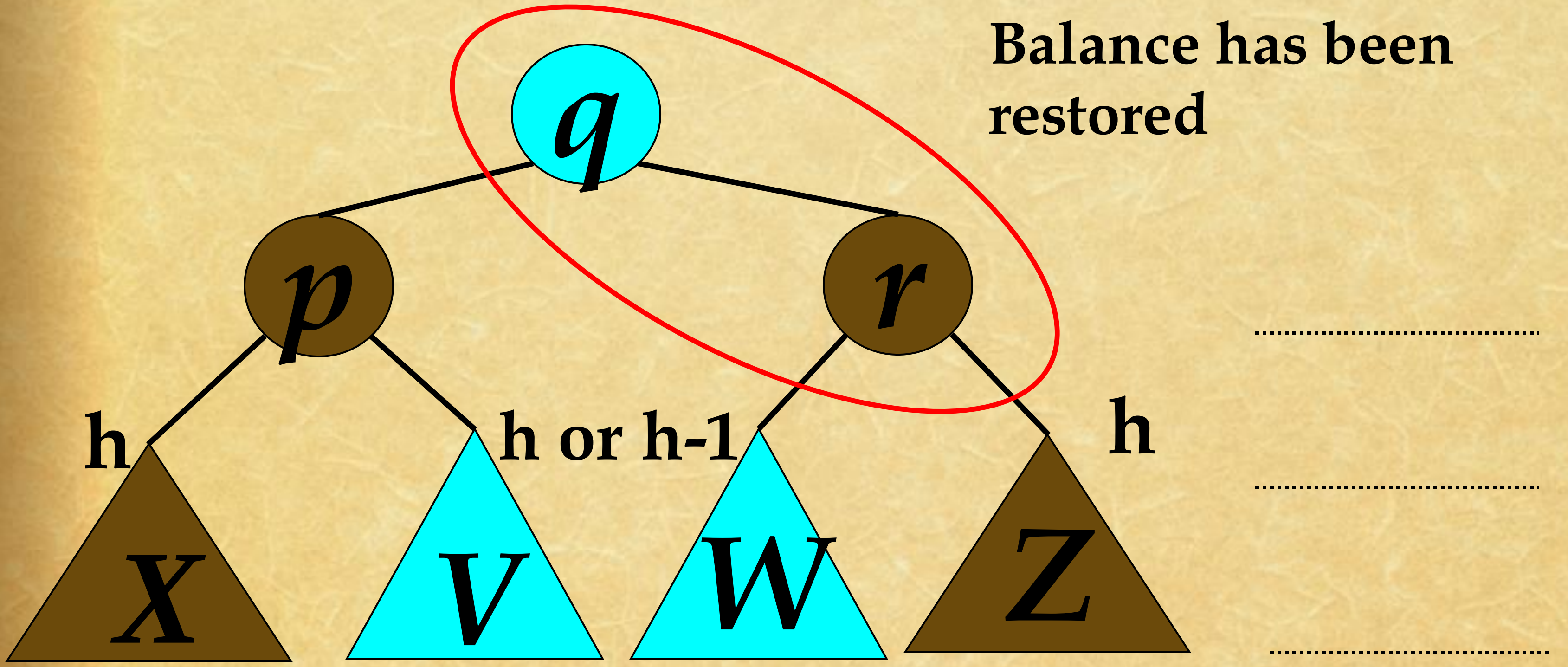
Now do a right rotation
Left(r) <- W
Right(q) <- r

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Double rotation : second rotation

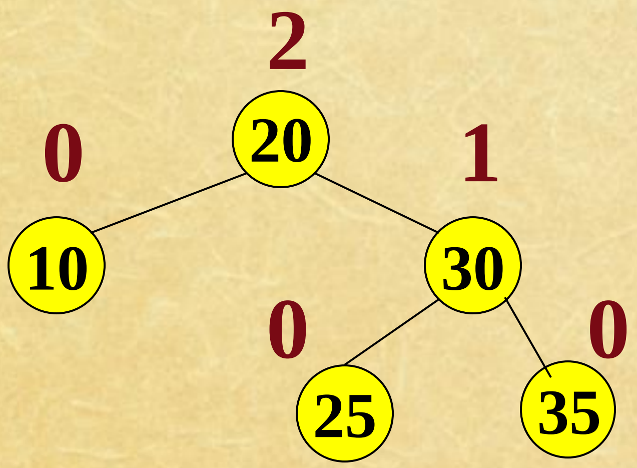
right rotation complete

Balance has been restored



courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

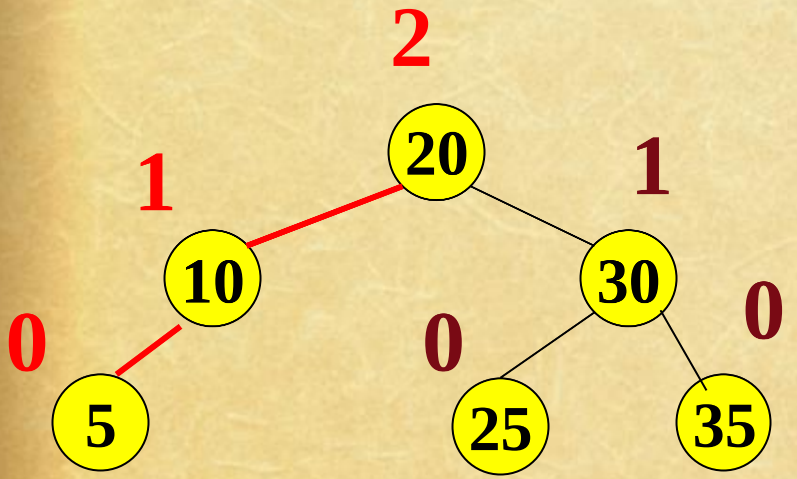
Example of Insertions in an AVL Tree

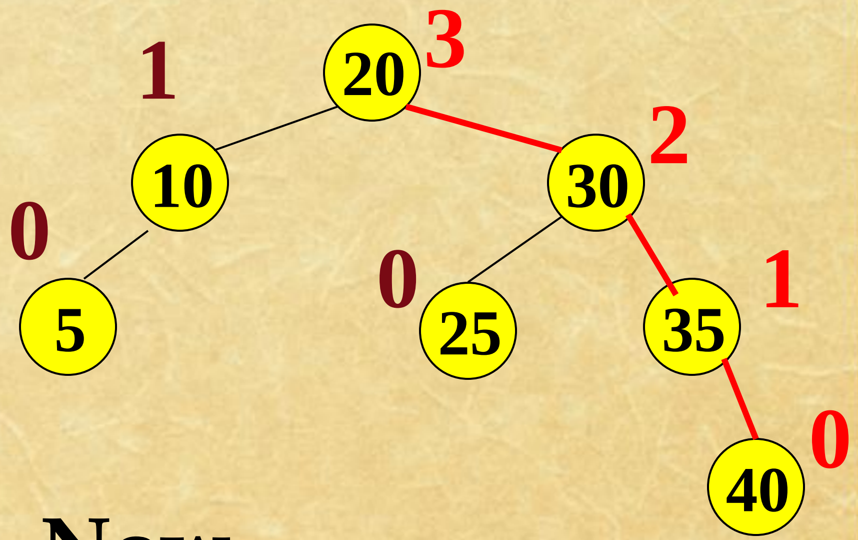


Insert 5, 40

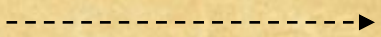
courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Example of Insertions in an AVL Tree



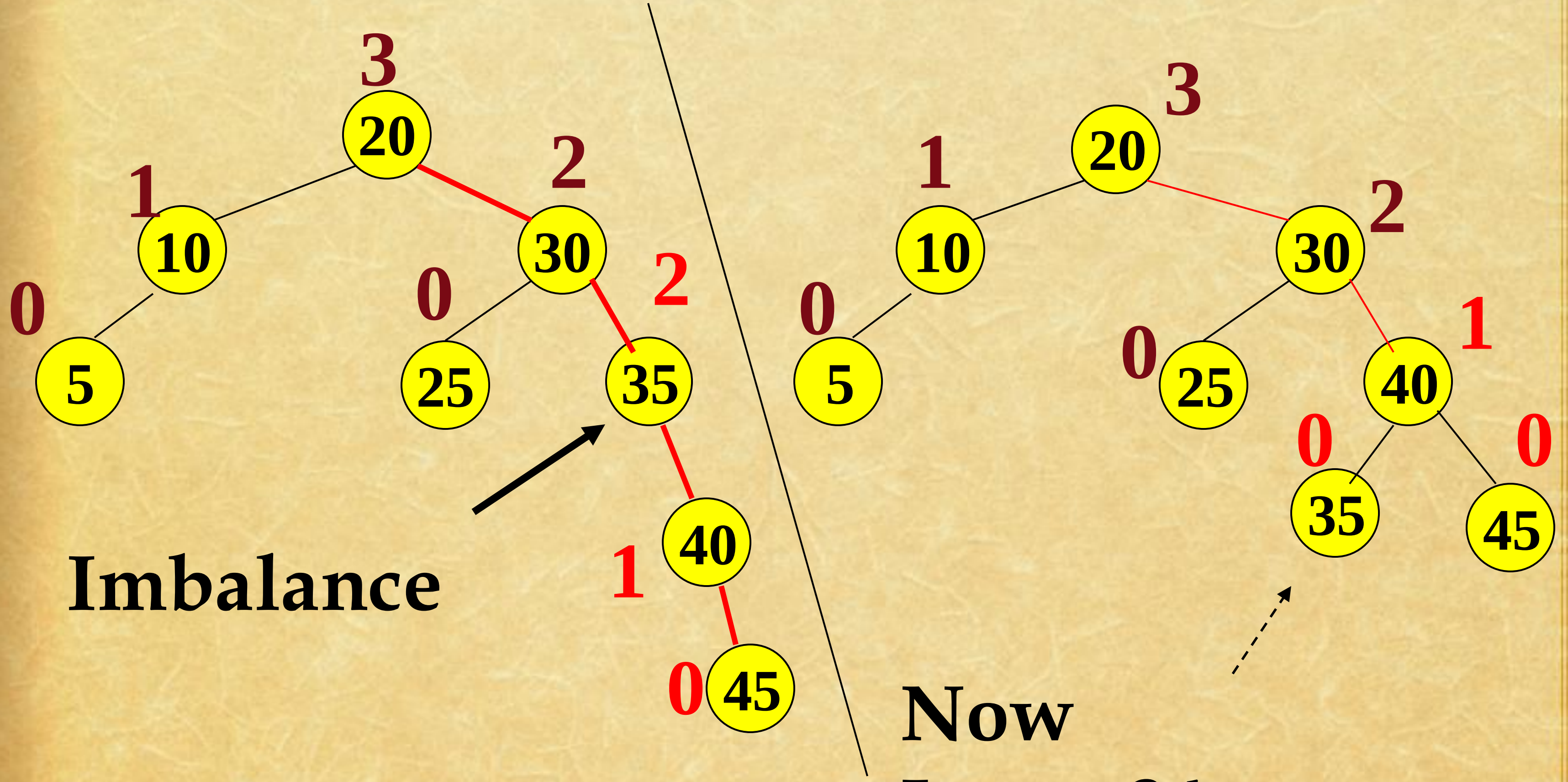


Now Insert 45



courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

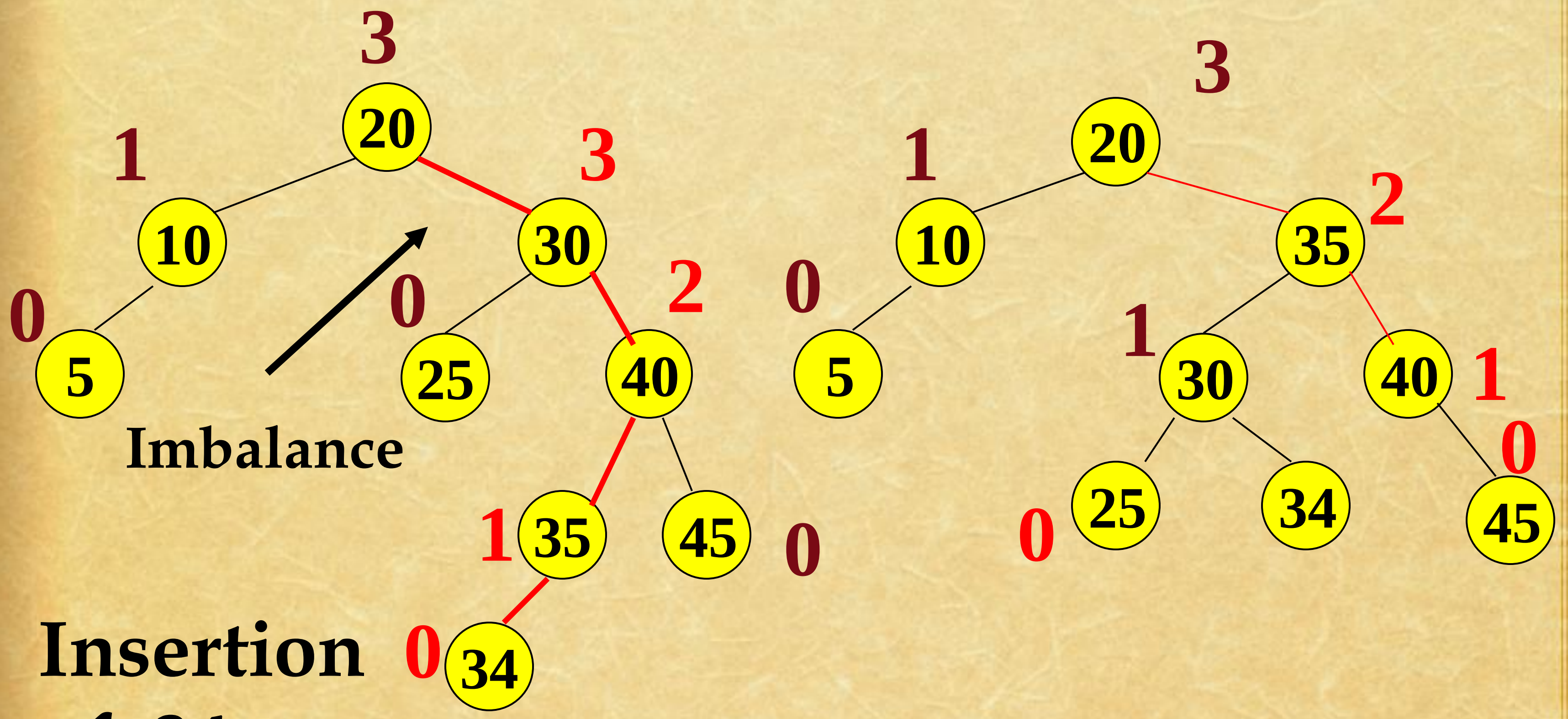
Single rotation (outside case)



Now
Insert 34

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

Double rotation (inside case)



Insertion of 34

courses.cs.washington.edu/courses/cse373/04wi/slides/lecture08.pp

AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Lists and optimal sequential search

- Consider a simple linked list as a representation for a set of keys.
- Standard solution involves
  - sorting keys in list in advance
  - search fails when we reach a key greater than the query key, or reach the end of the table.

Lists and optimal sequential search

- But sorting is the ``optimal'' strategy only if all keys are equally likely to be requested
  - If one key is the subject of 99% of all queries, should place that key at the beginning of the table.
  - But then the list will (probably) not be sorted

Minimizing the average cost of a search

- Intuitively, if we knew the probabilities of keys being accessed beforehand, then we would construct the list by ordering the keys according to their probability to minimize the expected cost of searching the list.
- But, we do not generally know these probabilities ahead of time.
- However, we can do **almost** as well as optimal if we **reorder** the list after each query.

The Move-to-Front heuristic

- After each successful search, the accessed element is moved to the front of the list
  - for a singly linked list this is very efficient to accomplish.

- Suppose we have a long sequence of queries.
  - If we knew the sequence ahead of time, we could use it to compute estimates of the $P(q = e_i)$ and order our list in decreasing order of these probabilities.
  - This is called the **static optimal list,** and there would be a total cost associated with answering all of the queries in the sequence.
  - This cost is mimimum over all possible permutations of the list because if we interchanged two elements in the optimal list, the time saved in searching for the less frequently asked for key is not as much as the extra cost incurred in searching for the more popular key.

The Move-to-Front heuristic

- Fact: Given a sufficiently long sequence of queries to a list containing the keys mentioned in the queries, the cost of answering these queries with the move-to-front heuristic is never more than **twice** the cost of answering the queries using the static optimal list.
- We need to consider a long list of queries, because one query, for example, can take long using the mtf heuristic (just ask for the last element in the list)
- This is proved using a proof method called **amortized analysis**.

Splay trees

- Called self-adjusting binary search trees
- No colors, balance or auxiliary fields - just a vanilla BST
- Lookup, insertion and deletion algorithms do not have O(logn) worst case complexity.
- But they have an amortized cost of log (n) - i.e., a sequence of m operations starting with an empty tree will take exactly mlog(n) operations, where n is the size of the largest tree constructed during the sequence.
- So, while a single operation can have a high cost - Ω(n) - this can only happen if it is preceded by many operations whose total cost is small, since the entire sequence must take mlog(n) operations.

Splay trees

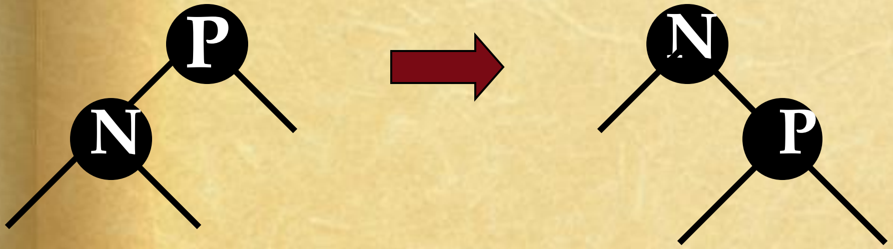
- In order to accomplish this, we must move an item after accessing it
  - otherwise we could continue to access an item at the end of a path of length O(n) and could not guarantee the amortized cost bound
- The key accessed is pushed up to the root of the tree by a sequence of AVL tree like rotations.
- If a key is deep, then as we rotate we will also bring keys on the path closer to the root - the rotations will tend to balance an unbalanced tree.

Splay trees: bottom-up splaying

- Splaying = moving an item to the root via a sequence of rotations
- In bottom-up splaying, we start at the node being accessed, and rotate from the bottom up along the access path until the node is at the root.
- The nodes that are involved in the rotations are
  - the node being accessed (N)
  - its parent (P)
  - its grandparent (G)

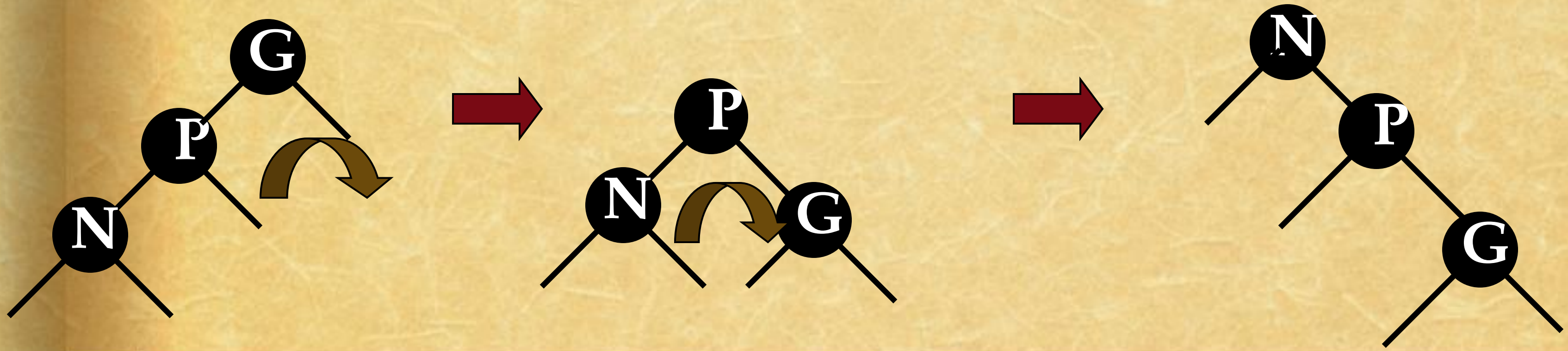
Splay trees: bottom-up splaying

- The rotation depends on the positions of the current node N, its parent P and its grandparent G



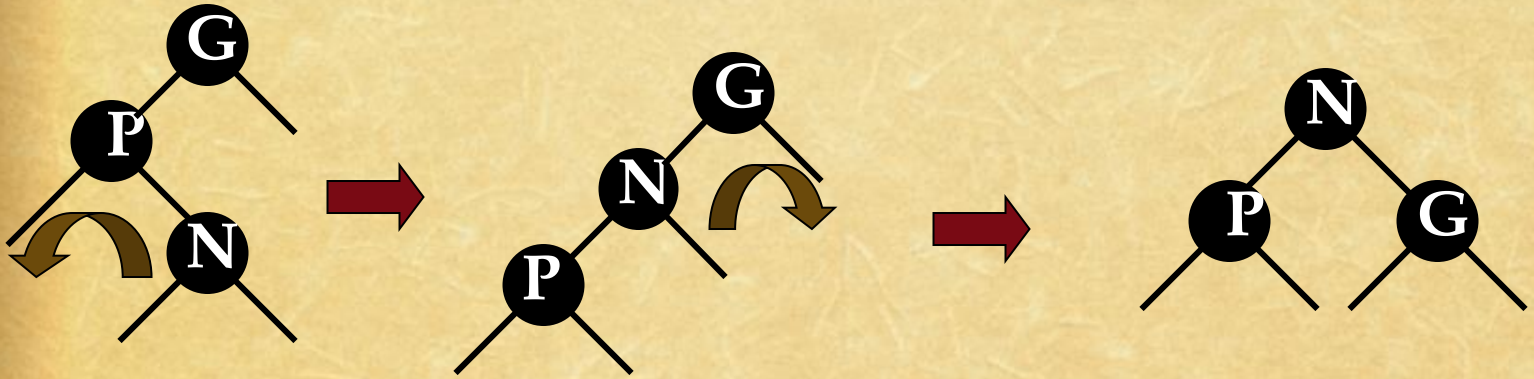
- If N is the root, we are done
- If P is the root, perform a single rotation

- If P and N are both left or both right children, first rotate P and then N as shown below



Splay trees: bottom-up splaying

- If P is a left child and N is a right child (or vice versa), first rotate P and then N as shown below



Splay trees: bottom-up splaying

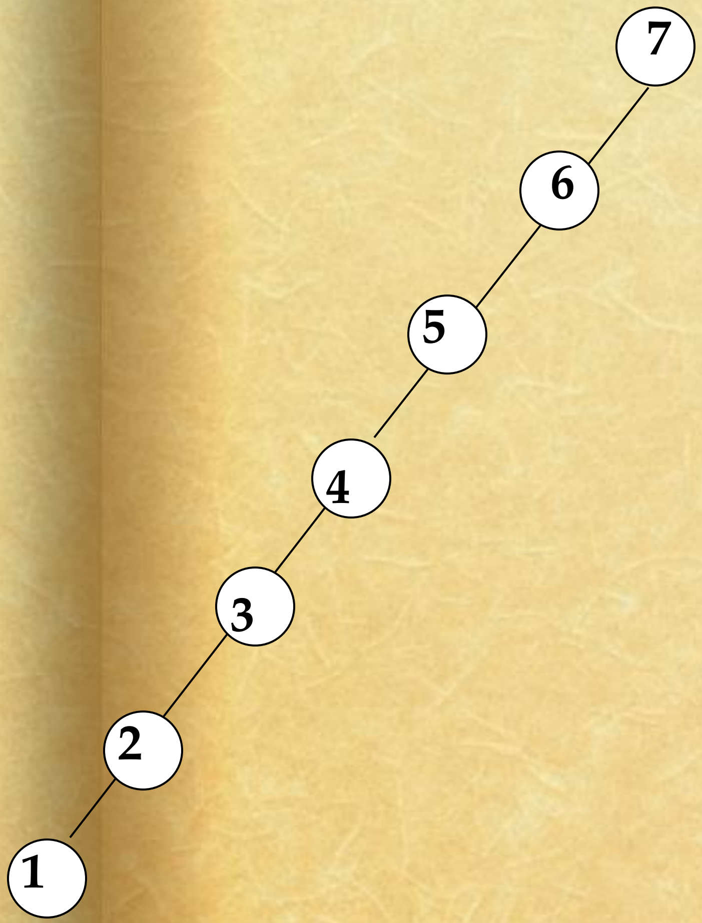
- Search
  - Once the node has been found, splay it
- Insert
  - Insert the new node and immediately splay
- Delete

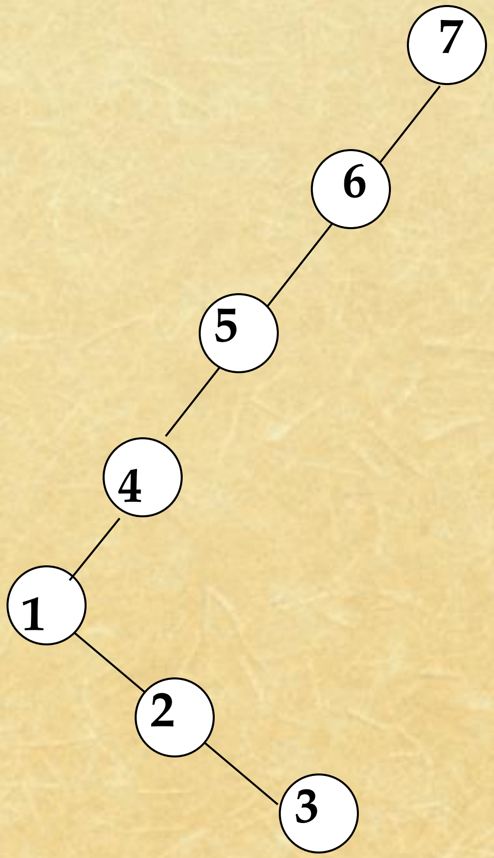
1. Do a Search for the node to be deleted. It will end up at the root. Removing it will split the tree in two subtrees, the left (L) and the right (R) one
2. Find the maximum in L and splay it to the root of L. This root will have no right child
3. Make R a right child of L

Example

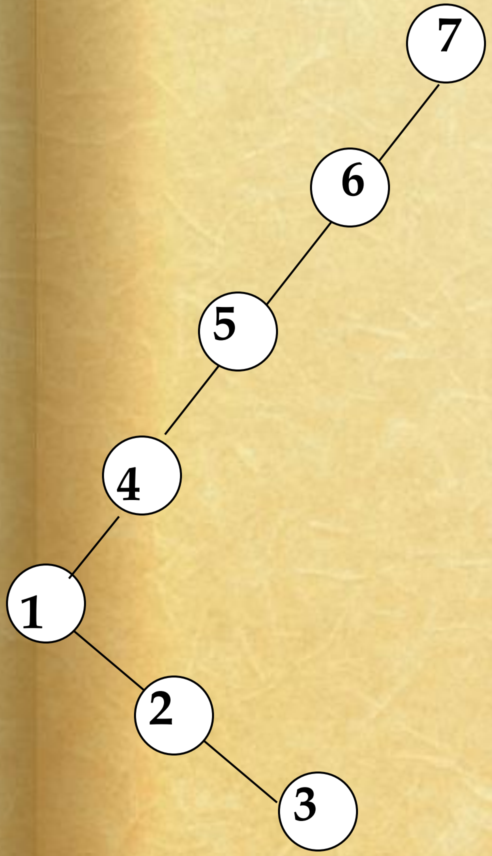
- Look at the effect of
  - inserting keys 1,2,3,4,5,6,7 in order into an initially empty tree
  - then accessing key 1, so we splay on 1.
- Each insertion will take constant time
  - splay (k,T) will fail at the root, since k will always be greater than the key stored at the root, and the right child of the root will be empty
  - so, we'll create a new root for k, and have its left link point to the old splay tree
  - this takes constant time
- The access of key 1 will then halve the path length to each node in this bad tree

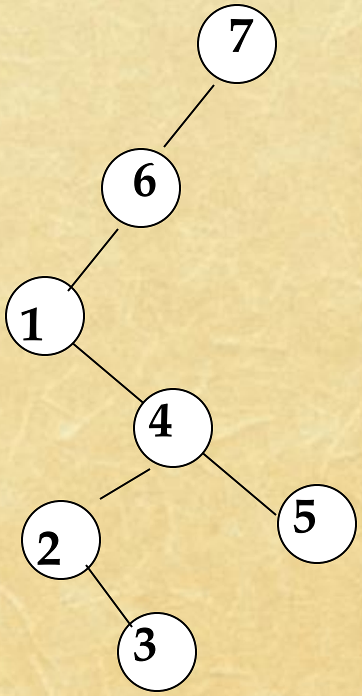
Example



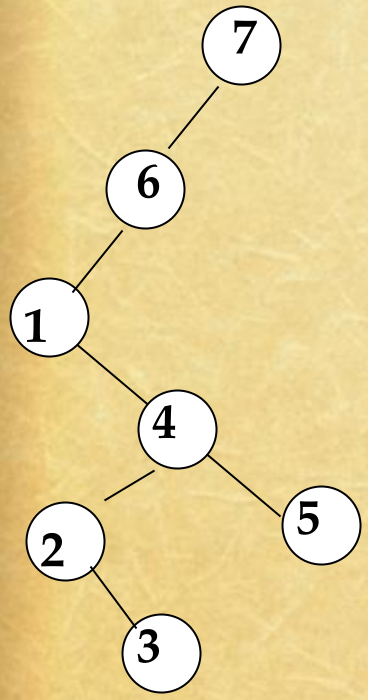


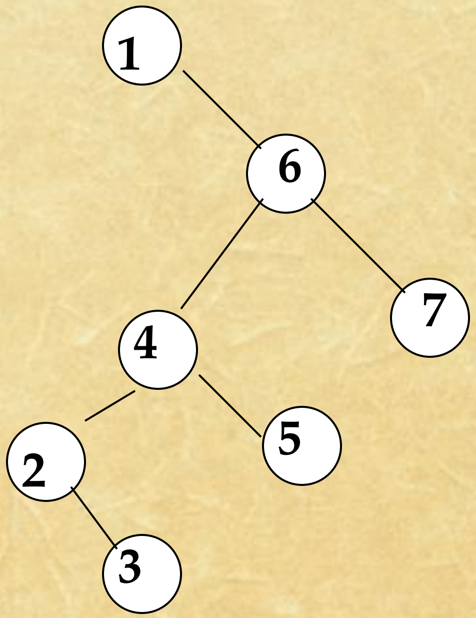
Example





Example





Code for splaying

```
type
    splay_ptr = ^splay_node
    splay_node = record
        element: element_type
        left: splay_ptr
        right: splay_ptr
        parent: splay_ptr
    end
    SEARCH_TREE = ^ splay_node
```

Basic splay routine

```
procedure splay (current:splay_ptr)
    var father : splay_ptr

begin
    father := current^.parent
    while father <> nil do
    begin
        if father^.parent  = nil then
          single_rotate (current)
        else
          double_rotate(current)
        father := current^.parent
    end

end
```

Single rotation

```
procedure single_rotate(x:splay_ptr)
begin
    if x^.parent^.left = x then
        zig_left(x)
    else
        zig_right(x)
end
```

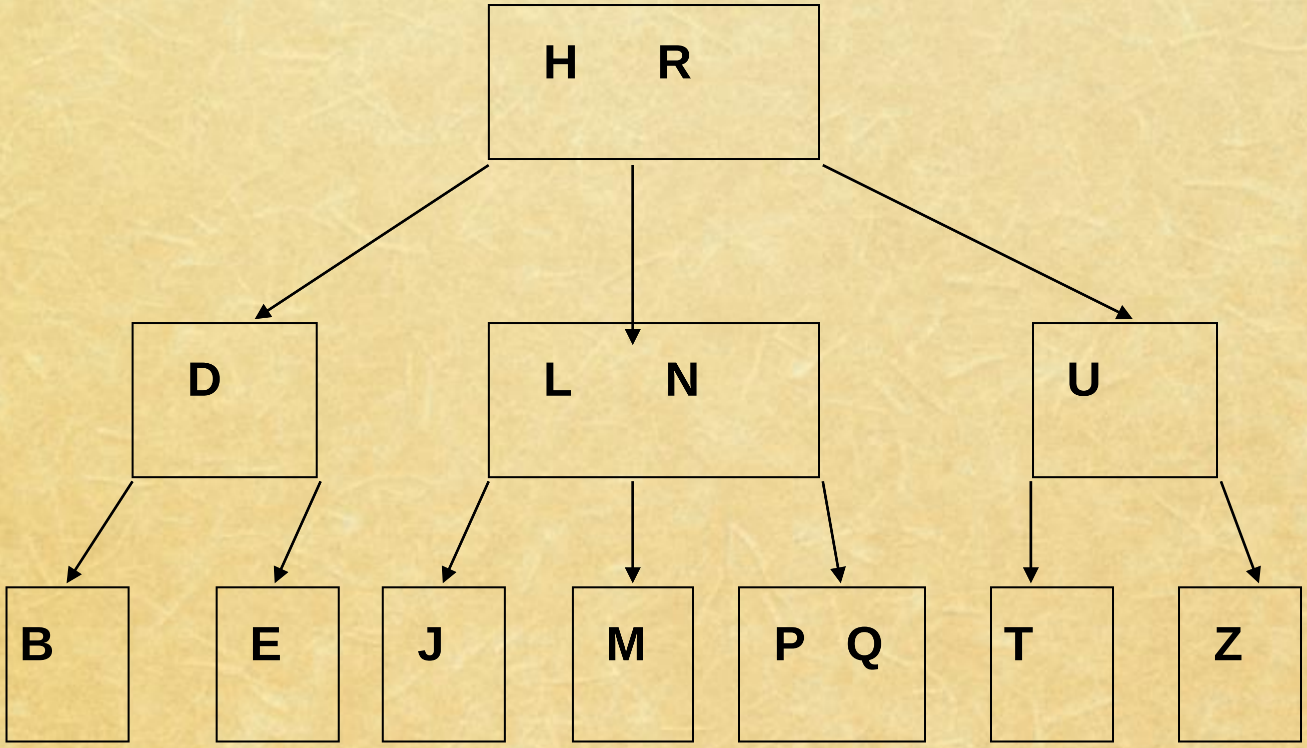
Zig left - single rotation between root and its left child

```
procedure zig_left(x: splay_ptr)
    var p, B :splay_ptr

begin
    p := x^.parent
    B := x^.right
    x^.right := p
    x^.parent := nil
    if B <> nil then B^.parent := p
    p^.left := B
    p^.parent := x

end
```

2-3 Trees

- Tree in which nodes that are not leaves may have either 2 or 3 children
- By arranging both types of nodes, can make a "perfectly balanced" tree in terms of path length
- Definition of a 2-3 tree:
  - all leaves are at the same depth and contain 1 or 2 keys
  - an interior node either contains one key and has two children (a 2-node) or contains 2 keys and 3 children (a 3-node)
  - A key in an interior node is between the keys in the subtrees of its adjacent children. For a 3-node the 2 keys fall between the 3 subtrees.

2-3 Trees



2-3 Trees

- A 2-3 tree having n keys can have height at most $\log_2 n$. This occurs when all of the nodes are 2-nodes, and we have a perfect binary tree
- A 2-3 tree in which all of the nodes are 3-nodes and which contains n keys will have height $\log_3 n$.
- Searching a 2-3 tree involves only trivial modifications to the BST algorithm to handle the 3-nodes. But insertion and deletion are complicated

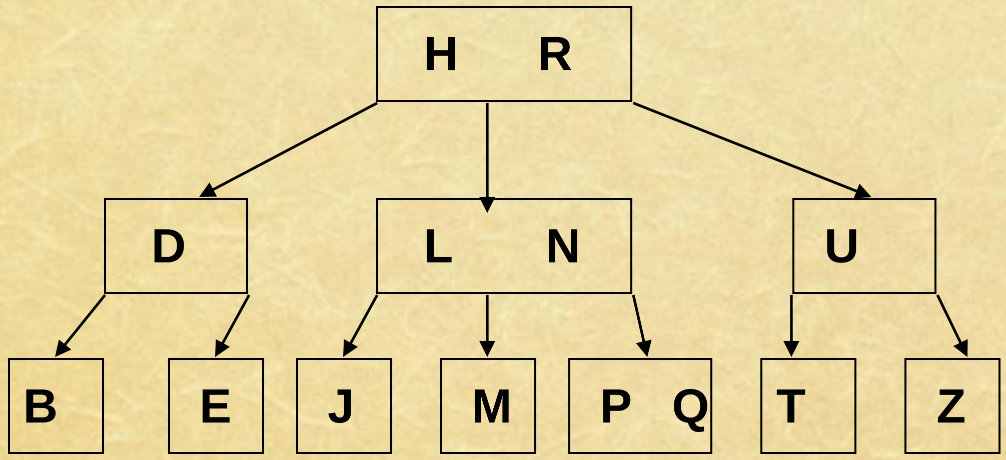
Insertion into 2-3 trees

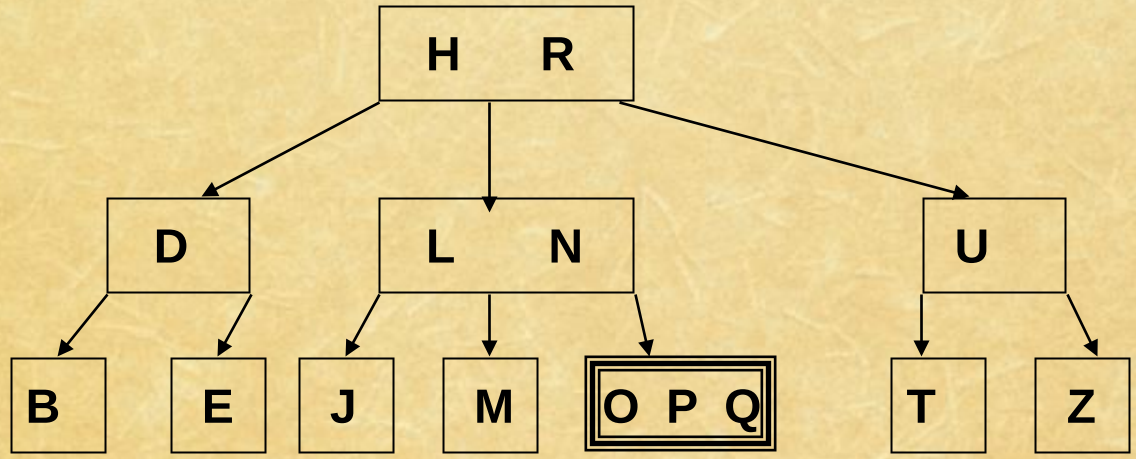
- Search for the leaf where the key belongs, remembering the path to that leaf.
- If the leaf contains only one key, add the key to that leaf and stop (example - add F to the 2-3 tree)

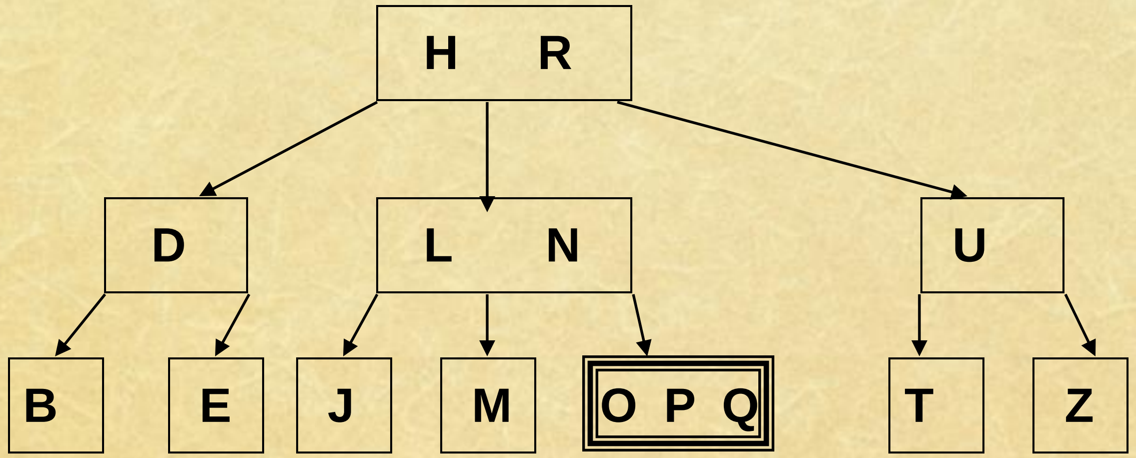
Insertion into 2-3 trees

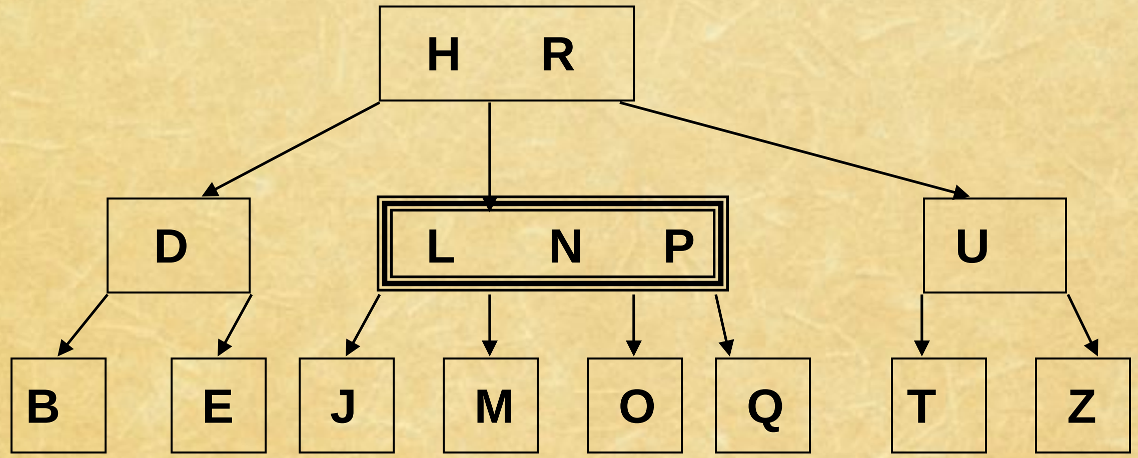
- If the leaf is full, split it into two 2-nodes - using the first and third key - and pass the middle key up to the parent to separate the two keys left in the leaf.
- If the parent was a 2-node, it is changed into a 3-node and we stop. Otherwise, we repeat the splitting step to the parent, promoting one of its keys up another level in the tree.
- If the root must be split, a new root is created and the height of the tree is increased by one.

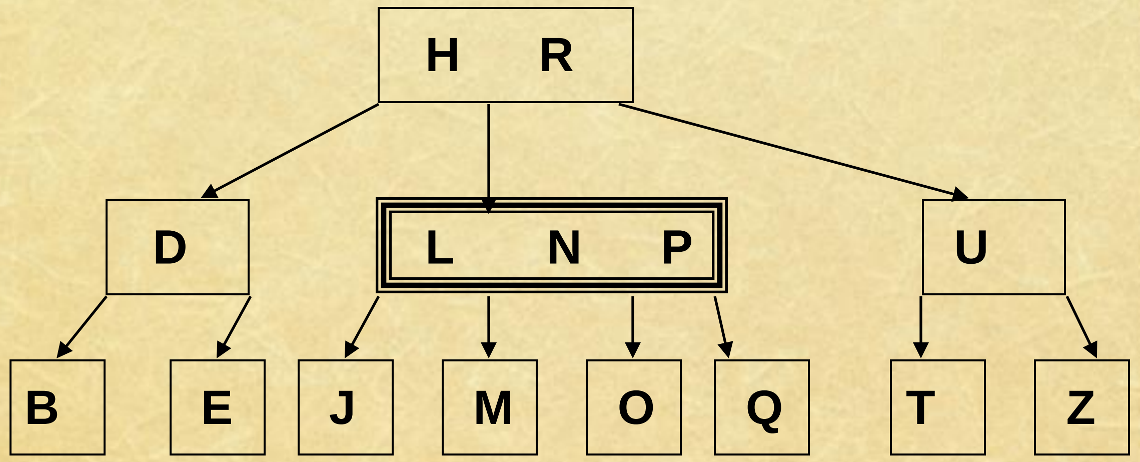
Example - insertion of O

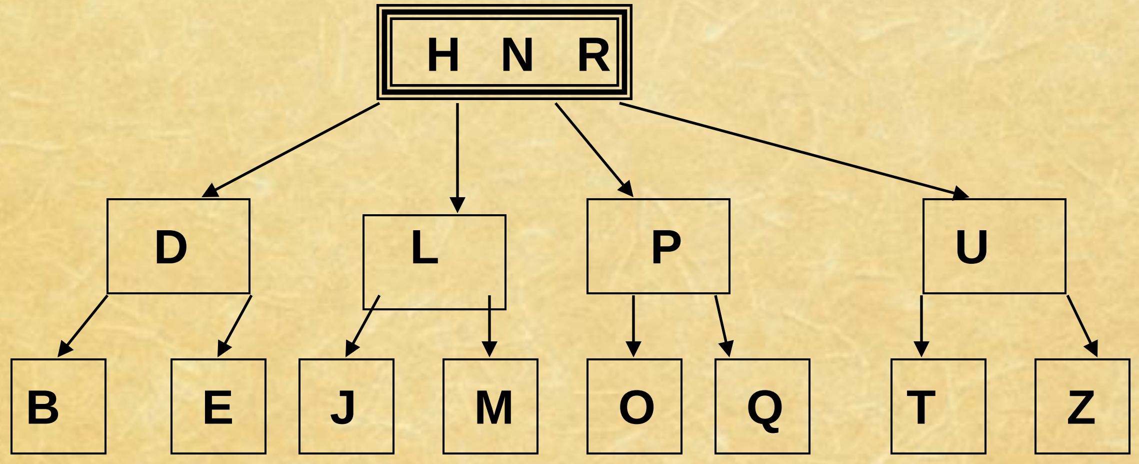


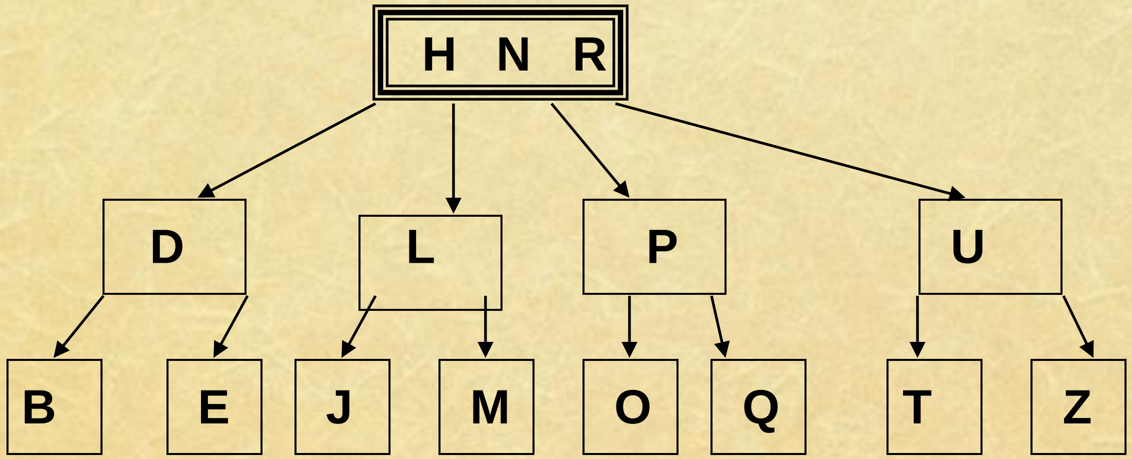


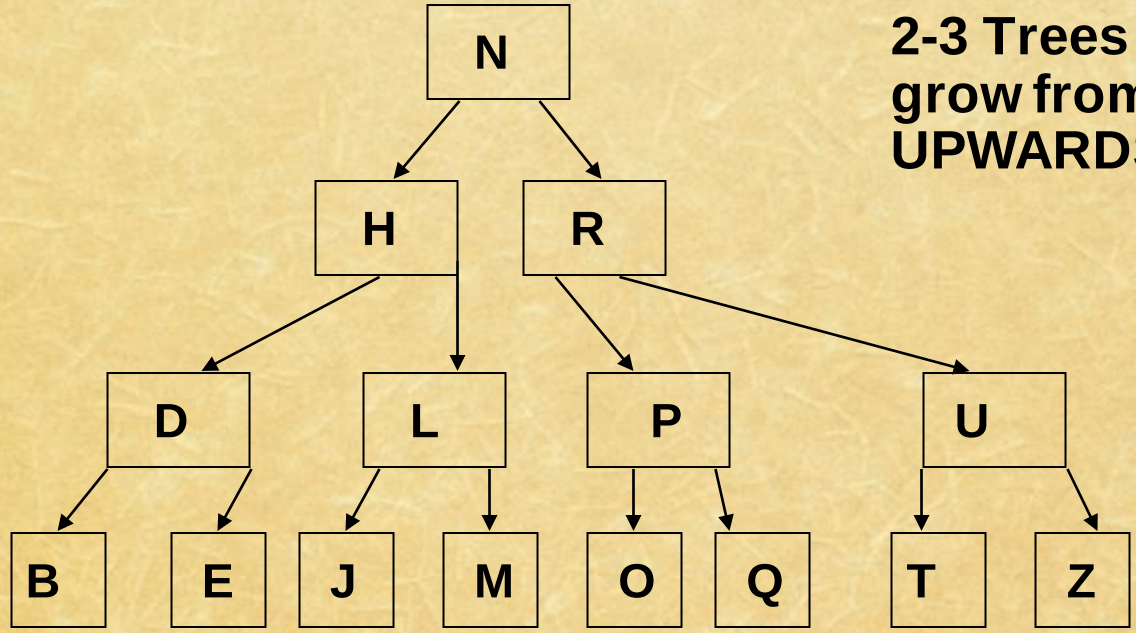

H R
D
L N
U
B
E
J
M
O P Q
T
Z


H R
D
L N P
U
B
E
J
M
O
Q
T
Z


H R
D
L N P
U
B
E
J
M
O
Q
T
Z


H N R
D
L
P
U
B
E
J
M
O
Q
T
Z

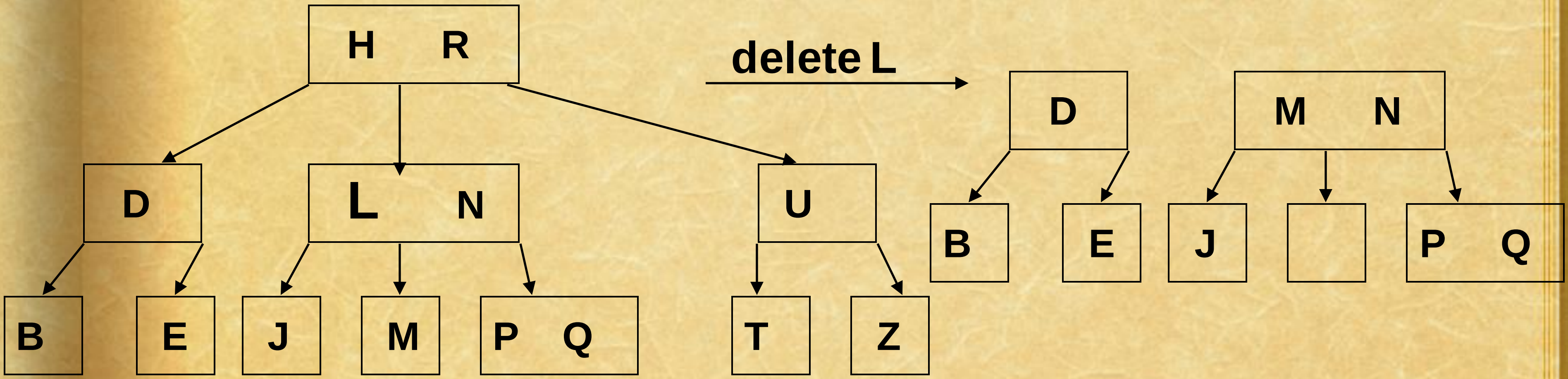




2-3 Trees ALWAYS grow from the root UPWARDS

Deletion from 2-3 trees

- [Always delete from a leaf] If the key to be deleted is in a leaf, remove it. If not, then the key's inorder successor is in a leaf (it is the leftmost node in the "right" subtree of that key). Replace the key by its inorder successor and remove the inorder successor from the leaf in which it was found.



Deletion from 2-3 trees

- Suppose we deleted a key from node α. If α still has one key, stop. If α has no keys:
  - If α is the root, delete it. If α had a child, this child becomes the new root.

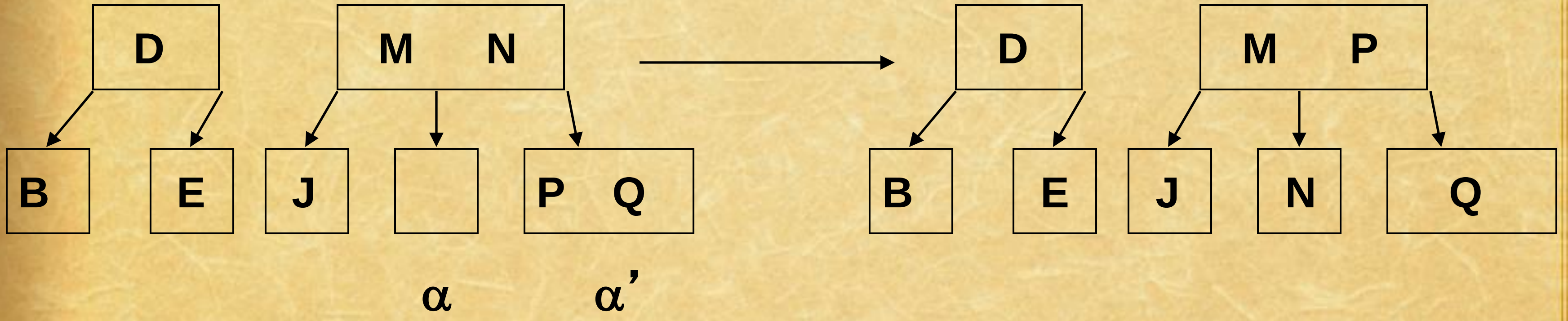
Deletion from 2-3 trees

- α must have one sibling (only the root doesn't).

  If α has a sibling α' immediately to its left or right that has 2 keys, then let S be the key in the parent that separates α and α'. Move S to α and replace it in the parent by the key in α' that is adjacent to α.

  If α and α' are interior nodes, then also move one child of α' to be a child of α. α and α' end up with one key each, rather than 0 and 2, and the algorithm is complete.

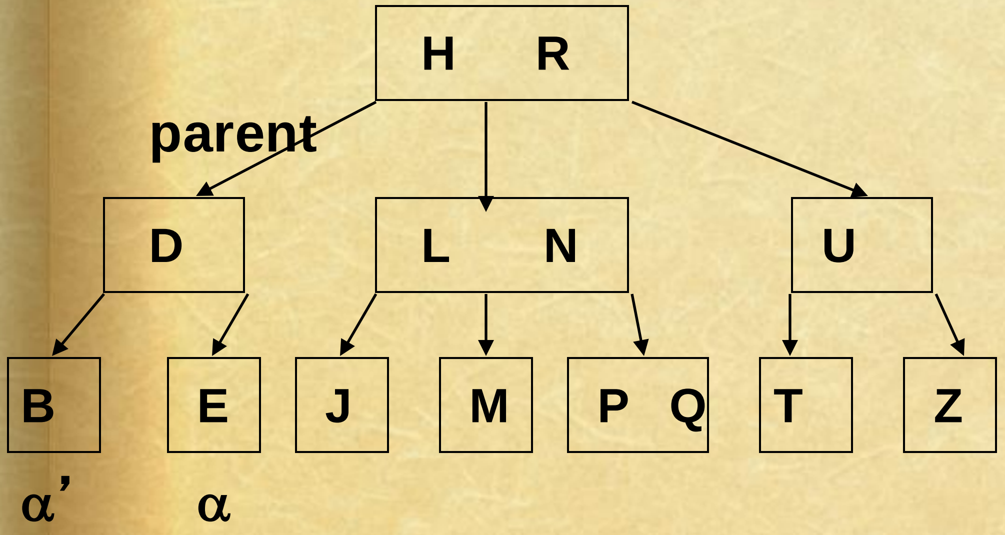
- α has a sibling α' immediately to its left or right that has 2 keys
- N is the key in the parent that separates α and α'
- Move N to α and replace it in the parent by the key in α' that is adjacent to α = P.

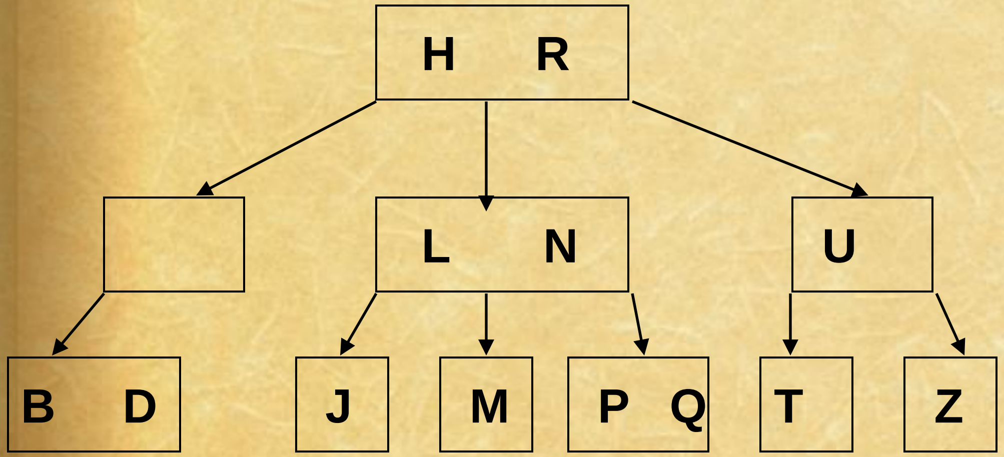


Deletion from 2-3 trees

- Continuing case of α having no keys and no chance of borrowing from sibling
  - [α has a sibling α' to its left or right that has only one key]. Let π be the parent of α and α', and let S be the key in π that separates them. Consolidate S and the one key in α' into a new 3-node which replaces both α and α'. This reduces by one both the number of keys in π and the number of children of π. Set α to π and go back to the second major step of the algorithm.

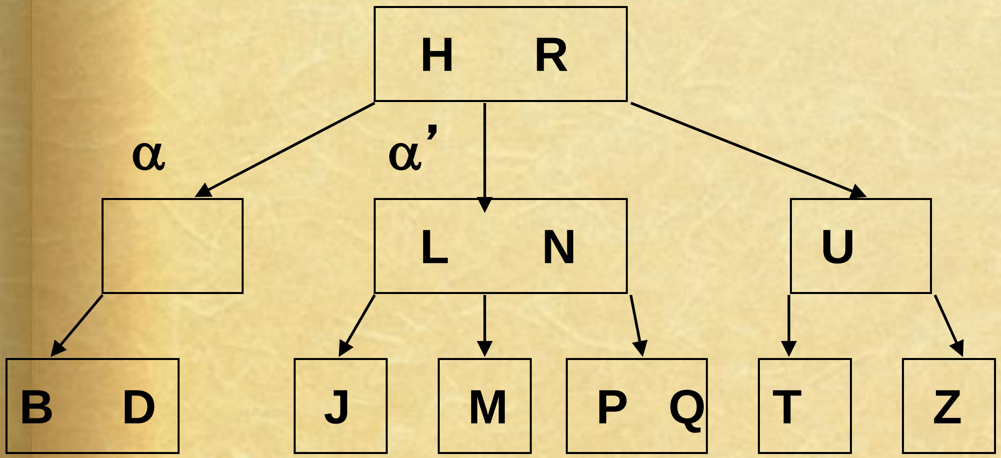
Example - deletion of E

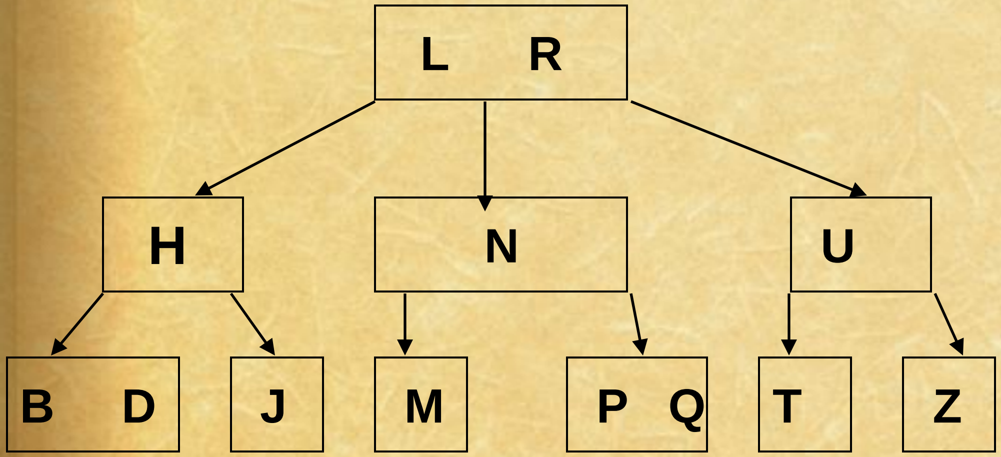




Let π be the parent of α and α′, and let S be the key in π that separates them. Consolidate S and the one key in α′ into a new 3-node which replaces both α and α′.

Example - deletion of E

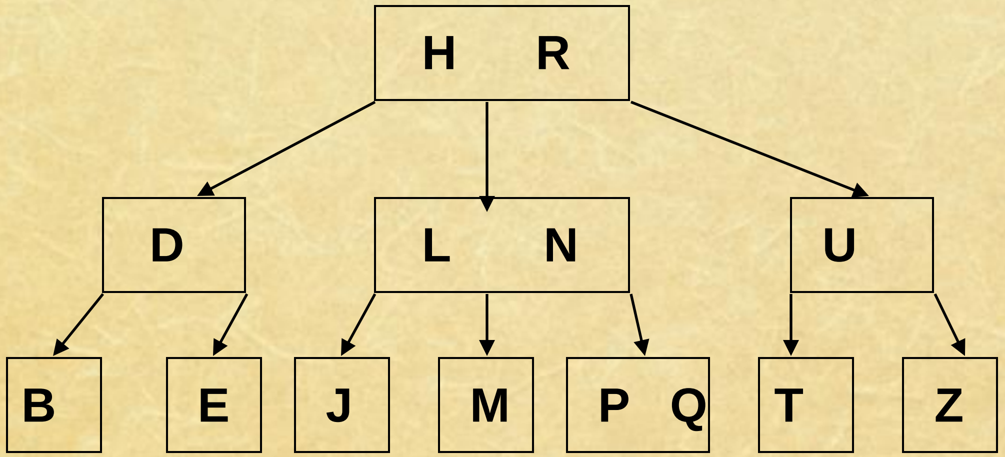


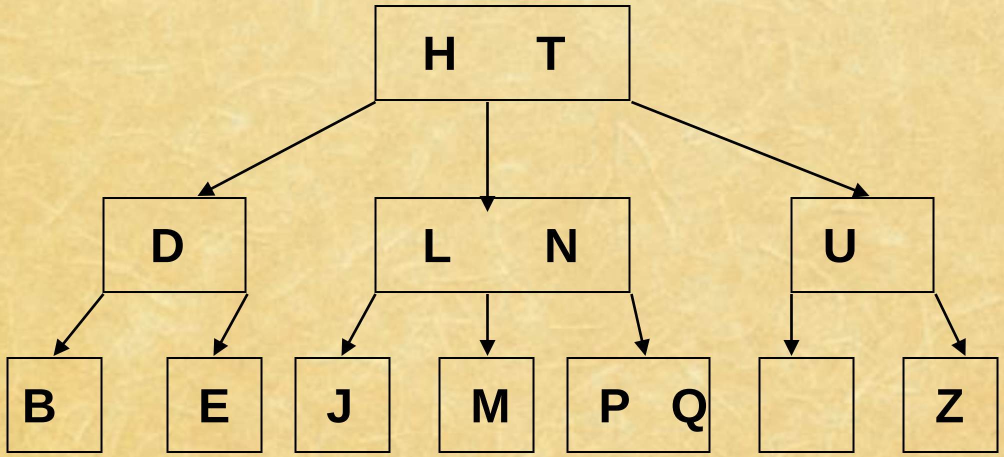


This reduces by one both the number of keys in π and the number of children of π. Set α to π and go back to the second major step of the algorithm. In this case we can move a key from α'

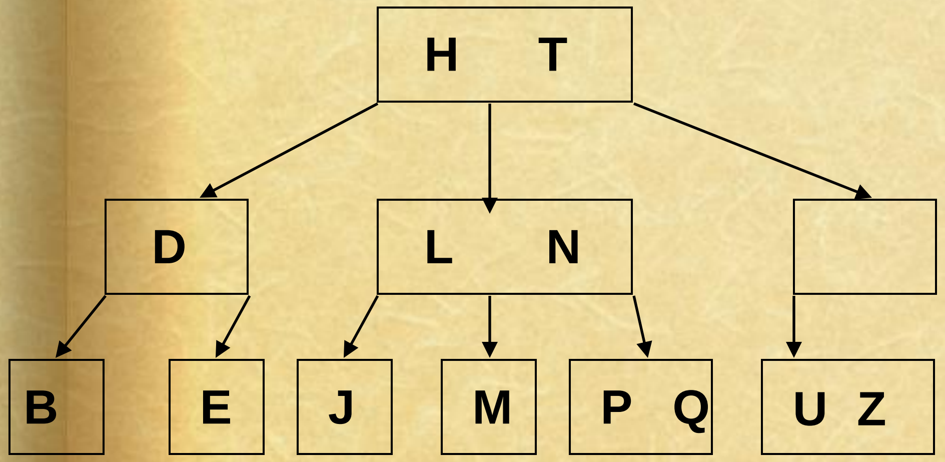
up to the parent and one from the parent down.

More examples - delete R

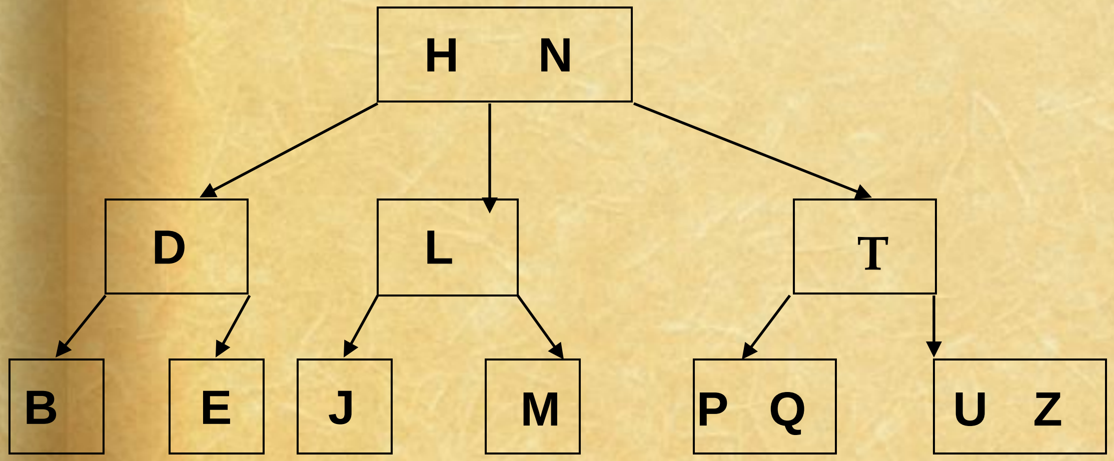




Delete R

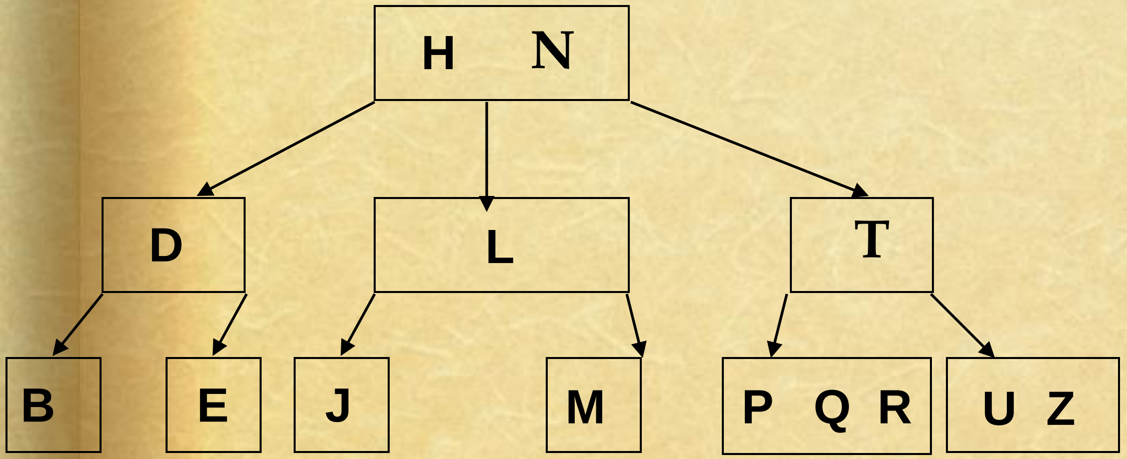


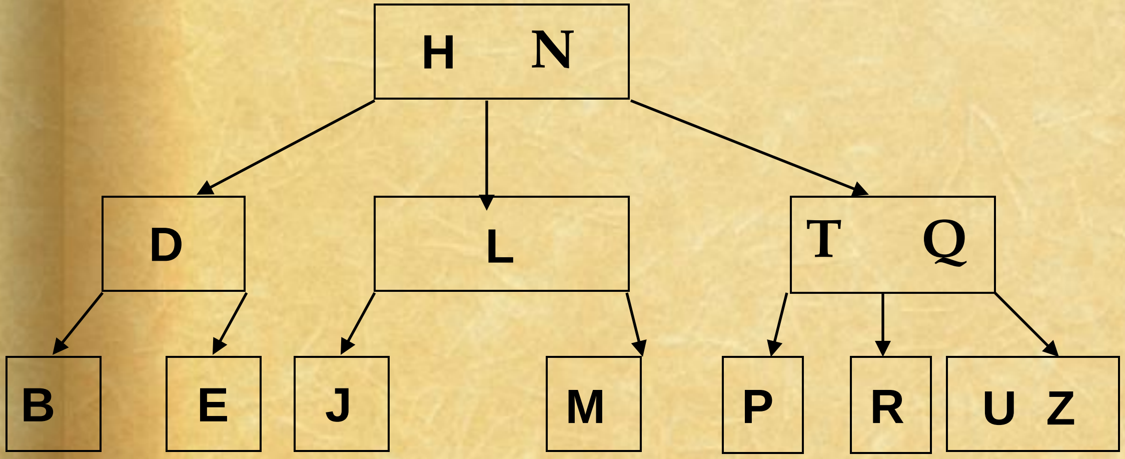
U was the key in parent that separated siblings; combine U with Z and remove U from parent - underflows parent.



Rotate key from sibling to parent, and borrow key from parent; readjust pointers to children

Insert R





Split overflow node into two nodes, promoting middle key to parent.

Difficulties with implementing 2-3 trees

- 2-nodes and 3-nodes have to be handled as separate cases
- have to resort to variant records, for example, but these can waste storage
- can we find a way to model 2-3 trees using regular binary trees?
  - red/black trees

Skip lists

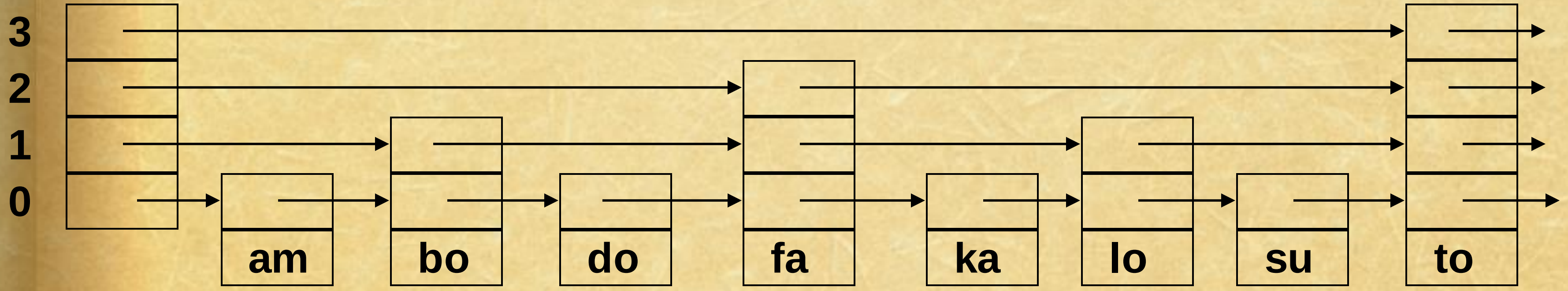
- How can we combine the simplicity of binary search over sequential allocation with the ease of insertions/deletions of linked lists?
- Skip lists - hierarchies of linked lists
- List array (bad solution on the way to skip lists)

 1) linked list at the bottom contains all of the elements of the set

 2) linked list at next level links every other element together.

 3) linked list at i'th level links every 2**(i-1) elements

List array - example



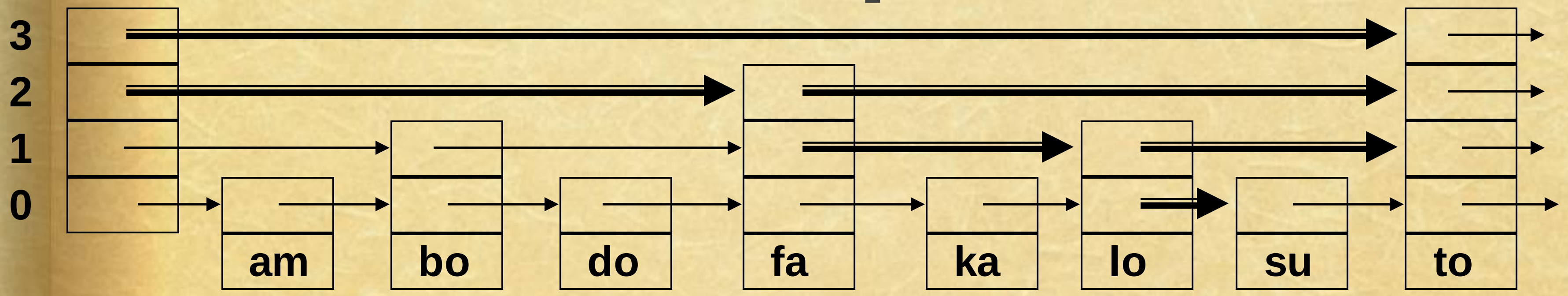
Searching in a list array

- Start at the highest level
- Scan the elements until encountering a node, p, whose value is greater than or equal to the key sought, k.
- If k is the value of node p, done; otherwise descend one level from the predecessor of p.

Searching in a list array

- If we reach the end of the lowest level list, the search fails.
- If list contains n elements, then search takes at most 2 logn operations since there are logn lists and at most 2 elements per list are examined before descending.

Search example



- Searching for RA follows the bold links.

Disadvantages of list arrays

- Consider the problem of updating the list array
- Inserting an element at the beginning of the list involves changing the level of every element in the list
  - Solution - relax the constraint that we skip a constant number of elements at each level of the list array, as long as the average number of nodes skipped is about right.
- Skip lists - list array built by generating the skip increment randomly. Design insertion routine to be twice as likely to generate a skip at level i than at level i+1

Insertion into a skip list

- Goal - insert a node with key k into a skip list

1) Search for k in the skip list, remembering the link fields traversed during the search at each level.

2) If the search is unsuccessful, position where k should be inserted at lowest level has been found - make the insertion

Insertion into a skip list

3) Generate a random number in [0,1] and:

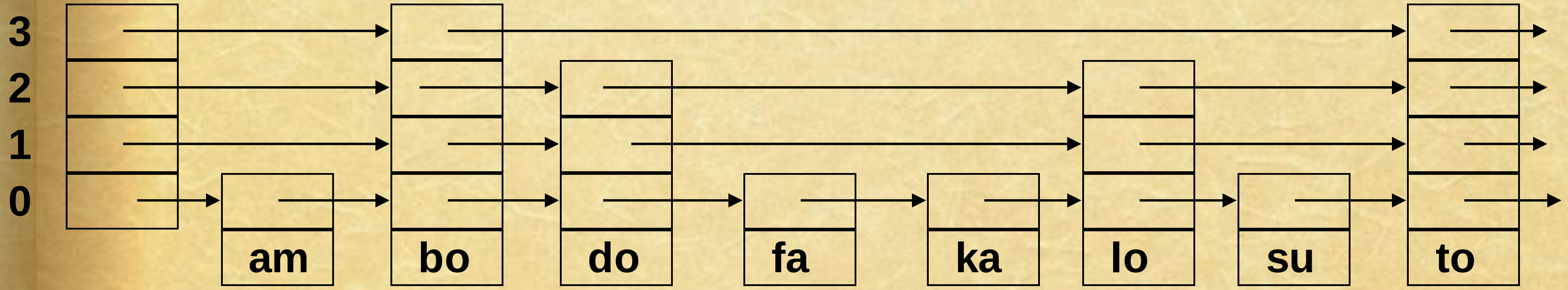
   a) if the number is less than .5, then exit

   b) if the number is greater than or equal to .5, ascend to level i+1 and insert k at this level.

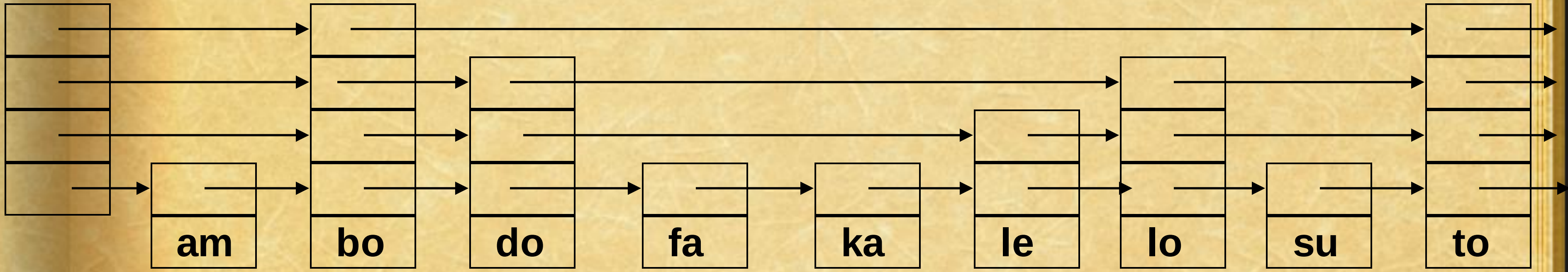
   c) Repeat this step if there are any more levels in the skip list.

4) Note that the probability of ascending n levels is 1/2**n, so that the skip list cannot grow very high. Usually a prior max level is set.

Skip list example – insert "le"



3
2
1
0
am
bo
do
fa
ka
lo
su
to



am
bo
do
fa
ka
le
lo
su
to