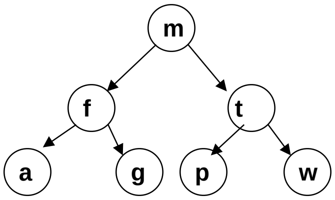
Dynamic data structures for searching

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:

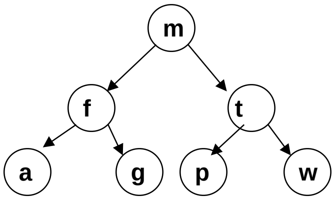
  1) If K matches the key at the root, done.

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

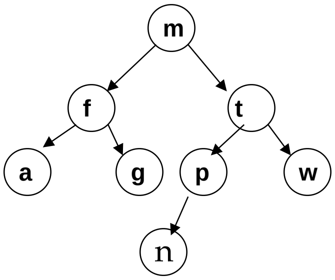
  3) Otherwise, search the right subtree.

  4) Reaching a nil link ends in failure.

Insertion into a BST

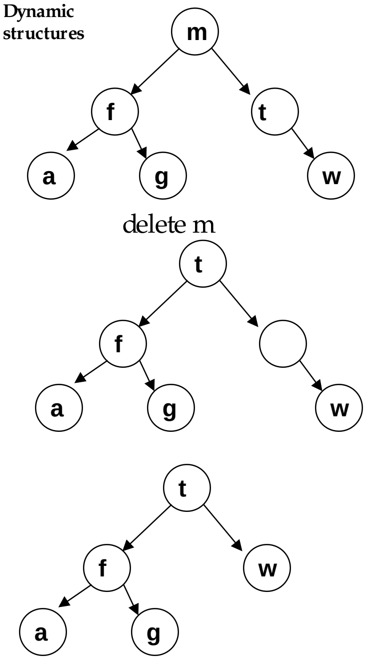


insert n



- 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 unbalance the tree badly

Deletion from a BST



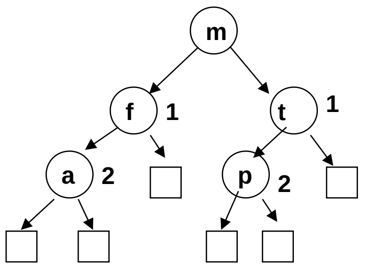
- Search(K) to find location of item to be deleted.
  - Leaf node - delete immediately
  - Node with one child - replace with child and delete empty child node
  - Node with two children:
    - » replace with smallest node in the right subtree
    - » then recursively delete the replaced node
    - »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

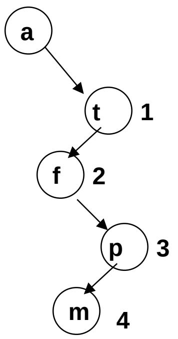
---

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?
  - – Ones with linear structure
  - – Internal path length is (n+1)/2
- ❑ How well does a random BST do?
  - – About 1.386 log n - 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)

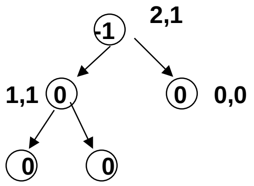
AVL Trees

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

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

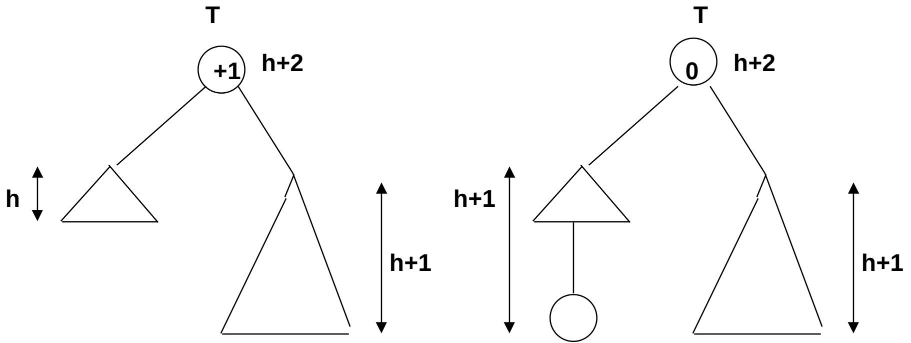
- Representation of AVL trees
  - add a balance field to each node in a binary tree
  - two bits are sufficient to encode the possible balances of +1, 0, -1
- General insertion algorithm:

  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

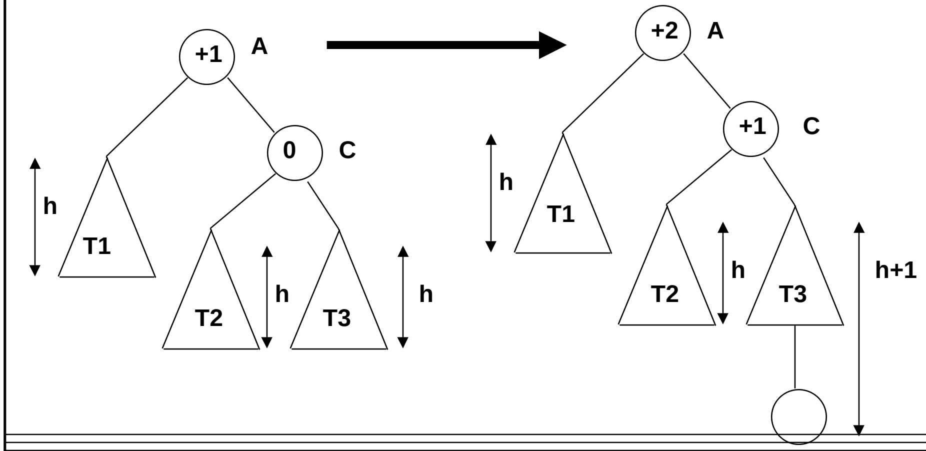

T
+1
h+2
h
h+1
T
0
h+2
h+1
h+1

- 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

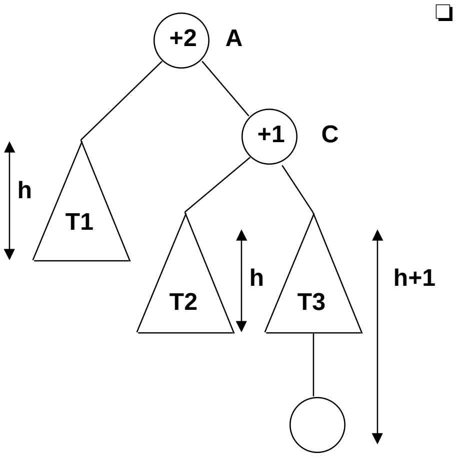
---

Insertion into AVL trees

- Two ways for a node to go out of balance


+1
A
0
C
h
T1
T2
h
T3
h
+2
A
+1
C
h
T1
T2
h
T3
h+1

Insertion into AVL Tree


+2
A
+1
C
h
T1
T2
h
T3
h+1

- How can we rebalance this AVL Tree?
  - Exchange subtrees T1 and T3?

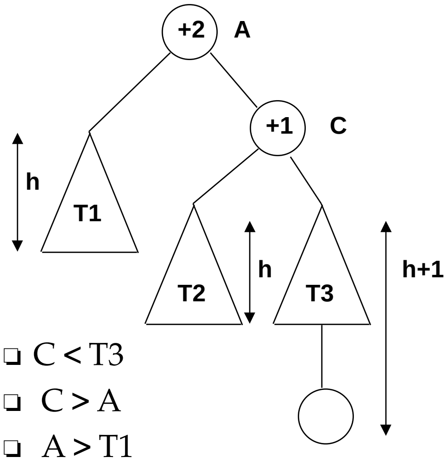
    No. This will balance the tree, but will destroy the BST property
  - Single **left rotation**:

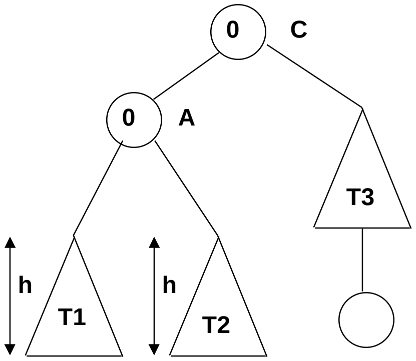
    - Make A the left child of C

    - Move the left subtree of C to become the right subtree of A

    - Makes the balance of A and C both 0

Insertion into AVL tree


+2
A
+1
C
h
T1
T2
h
T3
h+1

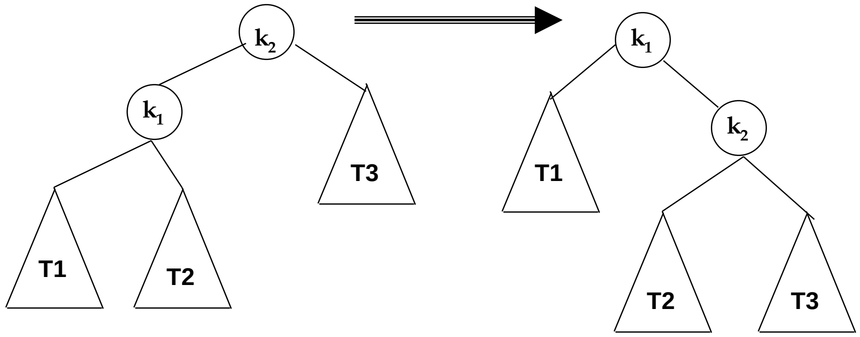

0
C
0
A
T3
h
T1
h
T2

- C < T3
- C > A
- A > T1
- C > T2 > A

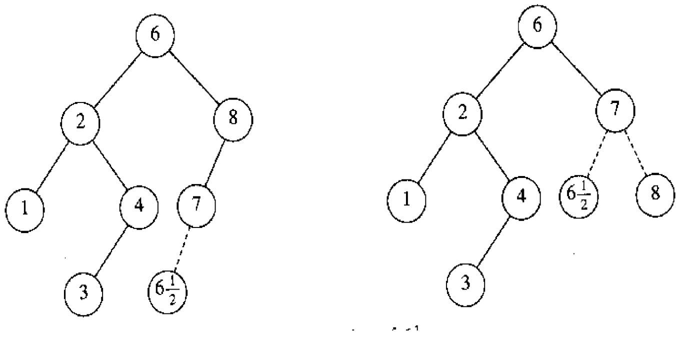
Single Left Rotation

(T1 A (T2 C T3)) ->
((T1 A T2) C T3)

Single Right rotation

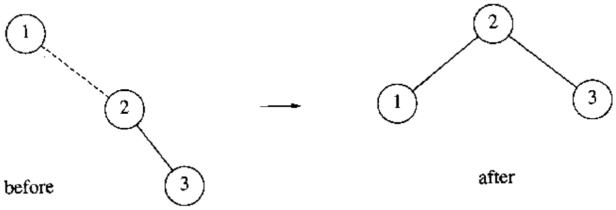

k_2
k_1
T1
T2
T3
k_1
k_2
T1
T2
T3

Single rotation fixes tree

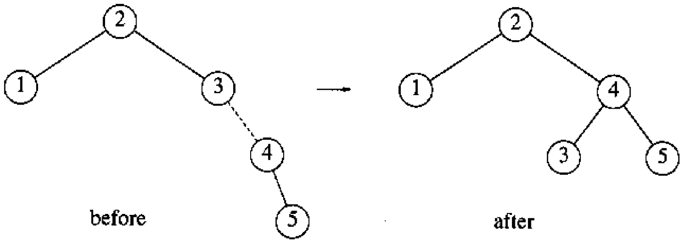

6
2
8
1
4
7
3
$6\frac{1}{2}$
6
2
7
1
4
$6\frac{1}{2}$
8
3

An extended example

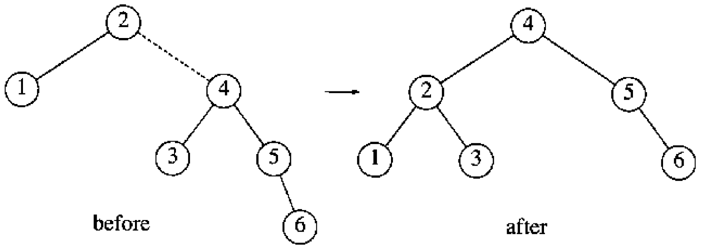
Insert the keys 1,2,...8 into an initially empty tree


1
2
3
before
2
1
3
after

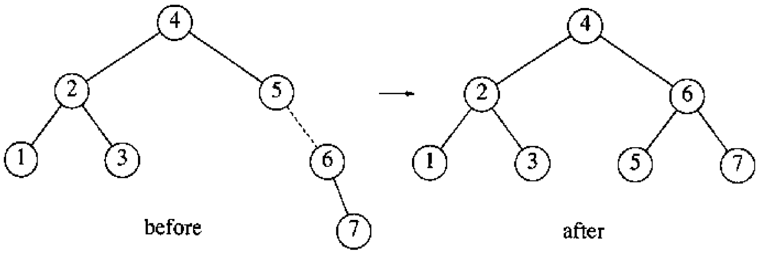
Extended example


2
1
3
4
5
before
2
1
4
3
5
after

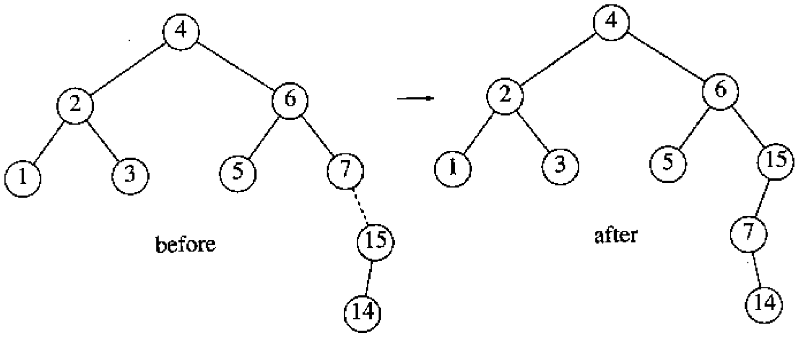
Extended example


2
1
4
3
5
6
before
4
2
5
1
3
6
after

Extended example

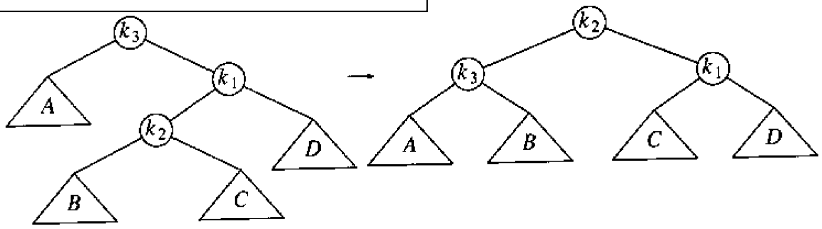

4
2
5
1
3
6
7
before
4
2
6
1
3
5
7
after

Trouble


4
2
6
1
3
5
7
15
14
before
4
2
6
1
3
5
15
7
14
after

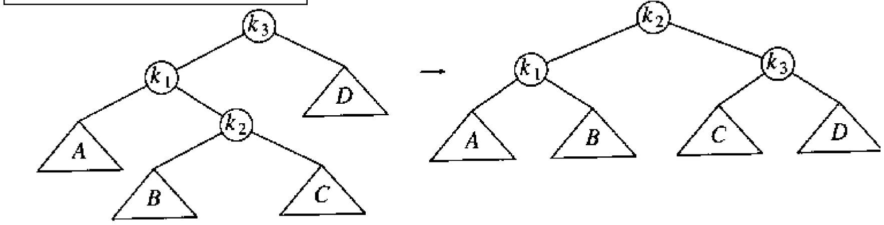
The new key is in the left subtree of the right son of the inbalanced node (i.e, 14 is between 7 and 15)

Solution - double rotations


k_3
k_1
k_2
A
B
C
D
k_2
k_3
k_1
A
B
C
D

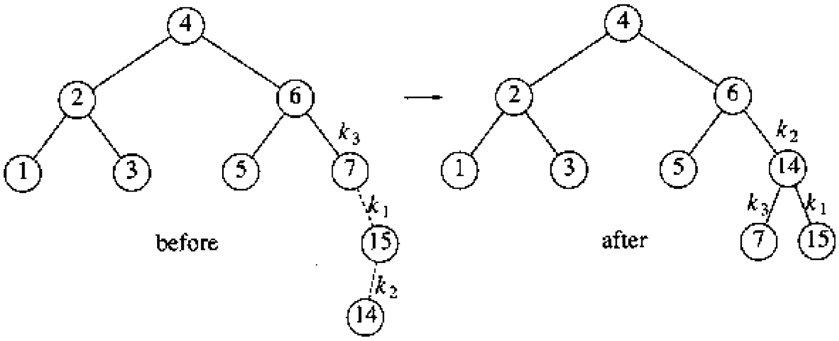
Right--left double rotation

Double rotations

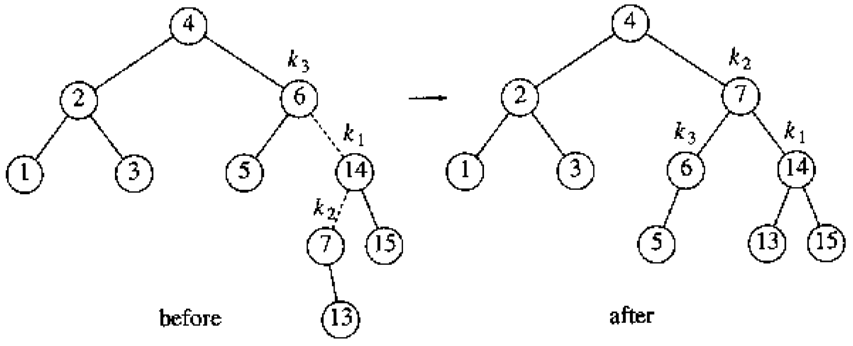

k_3
k_1
k_2
A
B
C
D
k_2
k_1
k_3
A
B
C
D

Left-right double rotation

Double rotations

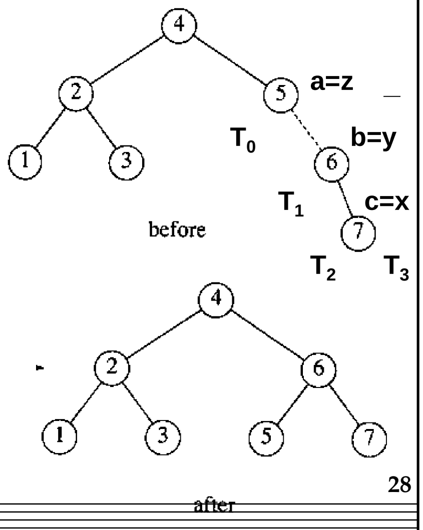

4
2
6
1
3
5
7
15
14
k_3
k_1
k_2
before
4
2
6
1
3
5
14
7
15
k_2
k_3
k_1
after

Double rotations


4
k_3
2
6
k_1
1
3
5
14
k_2
7
15
before
13
4
k_2
2
7
k_3
k_1
1
3
6
14
5
13
15
after

Simplification of rotations

- Let x be the first node encountered whose grandfather, z, is out of balance, and let y be the parent of x
  - Let (a, b, c) be an inorder listing of (x, y, z)
  - Let (T_0, T_1, T_2, T_3) be an inorder listing of the subtrees of (x,y,z)
  - Replace the subtree rooted at z with a new subtree rooted at b
  - Let a be the left child of b, and let T_0 and T_1 be the left and right subtrees of a.
  - Let c be the right child of a, and T_2 and T_3 be the left and right subtrees of c.
- When b = y we have a single rotation; b = x is a double


4
2
5
a=z
T_0
b=y
1
3
6
T_1
c=x
before
7
T_2
T_3
4
2
6
1
3
5
7
after

Some observations

- AVL trees are always fixed by either **one** single or double rotation
- Since only one rotation is needed, a program that is non-recursive is ordinarily much faster than a recursive program
- Deletion is not as simple.
  - Delete key as in BST
  - update balance factors moving back towards the root
  - when an out of balance key is detected, rotate - but this may cause other nodes, higher up, to go out of balance
  - continue up towards the root, making sure no nodes are out of balance.

Insertion procedure

```
procedure insert(x:element_type; var T: SEARCH-TREE):
if T = nil then make a node  ...
if x < T^.element then
    begin
     insert(x, T^.left);
    if  height(T^.left) - height(T^.right) = 2 then
            if x < T^.left.element then
                        single_rotate_left(T)
            else
                        double_rotate_left(left(T)
    else T^.height := max(height(T^.left), height(T^.right)) + 1;
```

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.
- ❑ 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.
- ❑ For the analysis that follows, we assume keys are not sorted, so all unsuccessful queries fail at the end of the list.

A simple model for the expected(average) cost of a search

- ❑ Let e_1, e_2, ..., e_n be the keys in positions 1, ..., n of a table.
- ❑ Let $P(q = e_i)$ be the probability that a member query is for key e_i.
  - – If $P(q = e_i) = 1/n$, then all keys are equally likely to be searched for.
  - – If $P(q = e_i) = 1/2^i$, then keys at the beginning of the table are much more likely to be asked for than keys near the end.
- ❑ The query might be for a key that is **not** in the table. The probability that this happens is P(q fails) =

$$\mathbf{1 - \Sigma\, P(q = e_i)}$$

A simple model for the expected(average) cost of a search

- The costs of search are:
  - for a successful search for e_i, cost is i.
  - for an unsuccessful search, n+1
- The average or expected cost of a query is then $\mathrm{Cost}(e_1, \ldots, e_n) = \Sigma i P(q=e_i) + (n+1) \bullet P(\text{query fails})$
- Examples:
  - If $P(q=e_i) = 1/n$, so that P(query fails) = 0, then $\mathrm{Cost} = 1/n \Sigma i = (n+1)/2$
  - If $P(q = e_i) = 1/2^i$ (so that $P(\mathrm{fail}) = 1/2^{n-1}$) , then we can show that Cost < 2 - most queries succeed in a constant number of comparisons near the beginning of the list.
  - If we reverse the probabilities, so that the most popular key is at the end of the list, then we can show that (n-1) < Cost < (n-2) - we almost always search the entire list

---

A simple model for the expected(average) cost of a search

- Intuitively, if we knew the probabilities 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)
- We can prove this using an **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 - $\Omega(n)$ - 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.

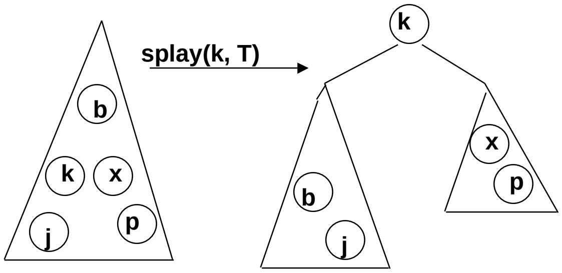
Splay trees

- 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.

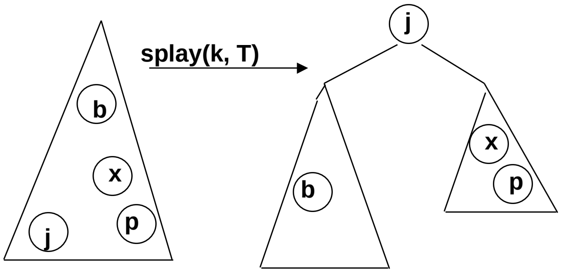
Splay trees

- Basic operation is a tree version of Move to front, and is called Splay.
- Splay (K,T) takes a BST, T, and a key, K, and modifies T so that it is still a BST on the same keys
  - If K was in the original tree, then the new tree has K at the root.
  - If K was not in the original tree, then the new tree has either the inorder successor or predecessor of K at the root.

Examples

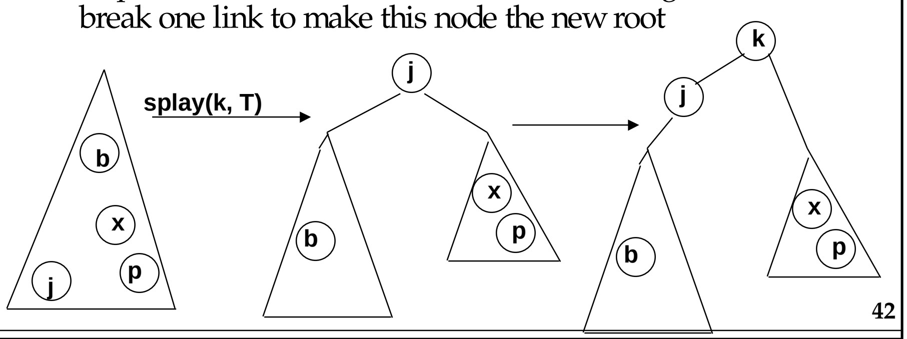


Examples


splay(k, T)
b
x
j
p
j
b
x
p

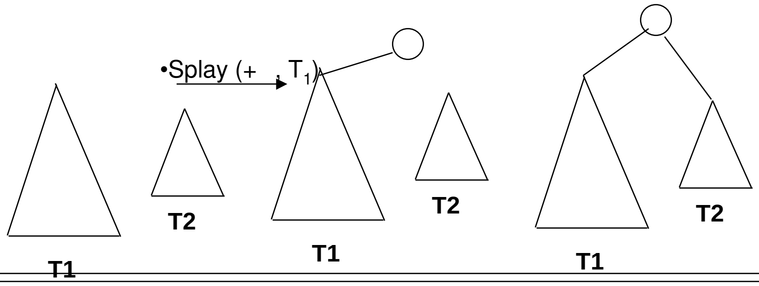
Operations on splay trees

- Suppose that we have an implementation of Splay (to follow)
- Lookup(K,T) - Execute Splay(K,T) and then examine the root of the tree to see if it contains K.
- Insert(K,T) - Execute Splay(K,T). If K is at the root, stop. Otherwise, create a new node containing K and break one link to make this node the new root


splay(k, T)
b
x
j
p
j
b
x
p
k
j
b
x
p

Operations on splay trees

- Deletion requires an operation call Concat(T_1, T_2)
- If T_1 and T_2 are BST's such that every key in T_1 is less than every key in T_2, then concat(T_1,T_2) creates a BST containing all of the keys in T_1 and T_2.
- Concat(T_1,T_2) - first execute Splay (+ , T_1). After this, T_1 has no right subtree; attach the root of T_2 as the right child of the root of T_1.

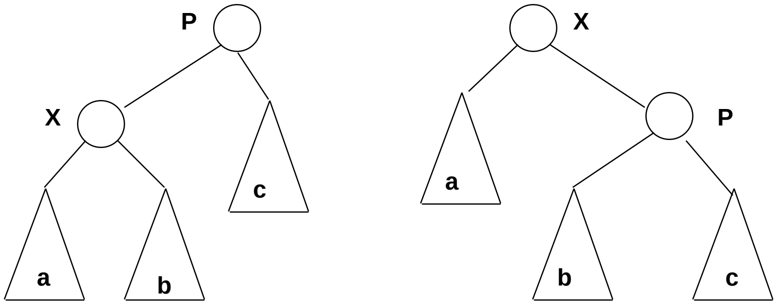


Operations on splay trees

- Now we can define delete(K,T) - Execute Splay(K,T). If the root does not contain K, then there is nothing to do. Otherwise, apply Concat to the two subtrees of the root.
- Now, how do we implement splay(K,T)?
  - first search for K, remembering the path. Let X be the last node inspected.
  - If K is in the tree, then K is in node X; otherwise X has an empty child at which the search terminated unsuccessfully. X will become the new root.
  - Return from X to the root executing rotations

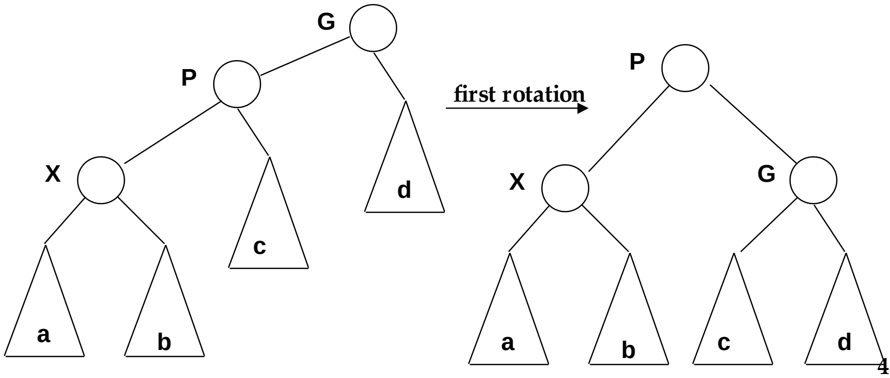
Splay rotations

- ❑ The rotations move X up through the tree. There are 3 cases
  - – X has no grandparent, so parent(P) is the root. Perform a single rotation around P.

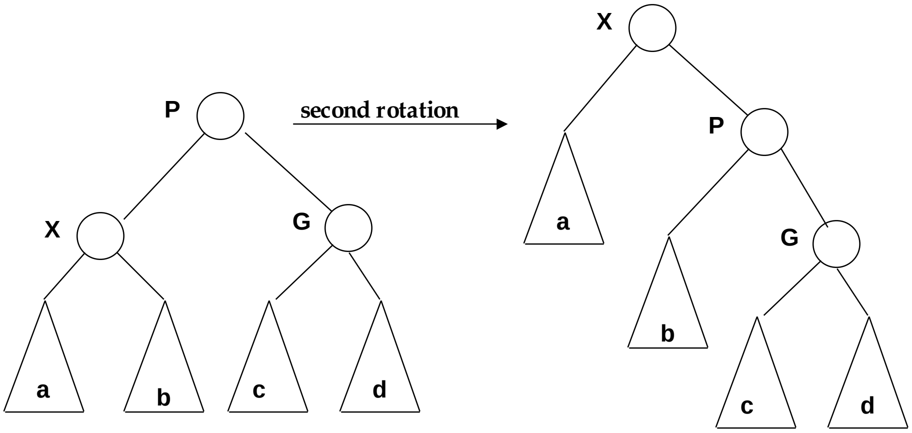


Splay rotations

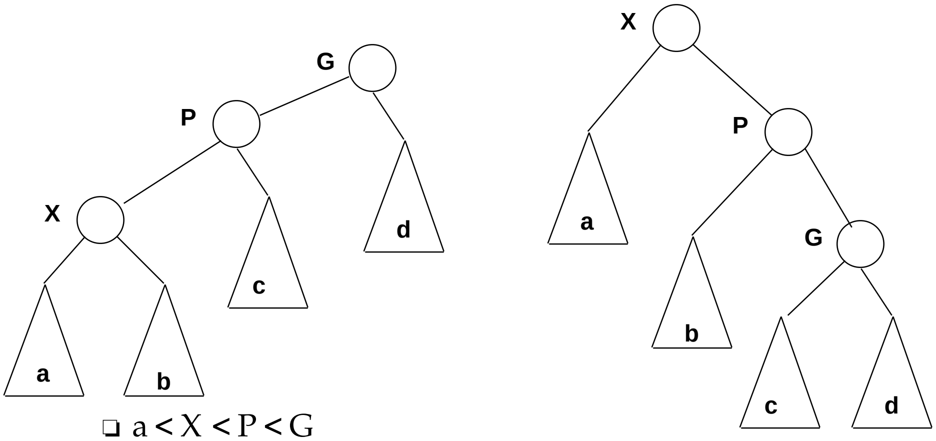
- ❑ Case 2 - P and its parent(G) are both left children or both right children. Here, we perform two single rotations in the same direction - first around the grandparent and then around the parent . Zig-zig or Zag-zag



Splay rotations


P
X
G
a
b
c
d
second rotation
X
P
G
a
b
c
d

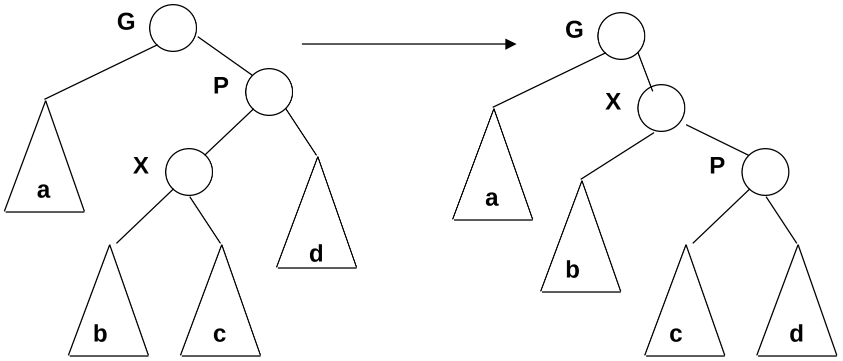
Splay rotations


G
P
X
a
b
c
d
X
P
G
a
b
c
d

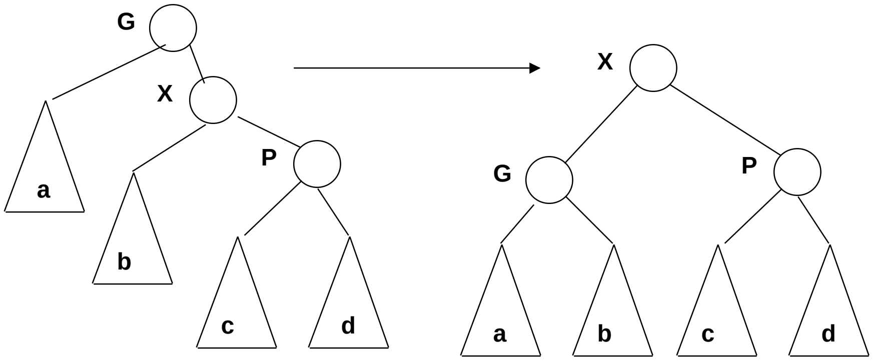
- $a < X < P < G$
- $b > X$, $b < P$
- $c > P$, $c < G$
- $d > G$

Splay rotations

- Case 3: One of P and X is a left child and the other is a right child. Perform single rotations in opposite directions, first around P and then around G.


G
P
X
a
b
c
d
G
X
P
a
b
c
d

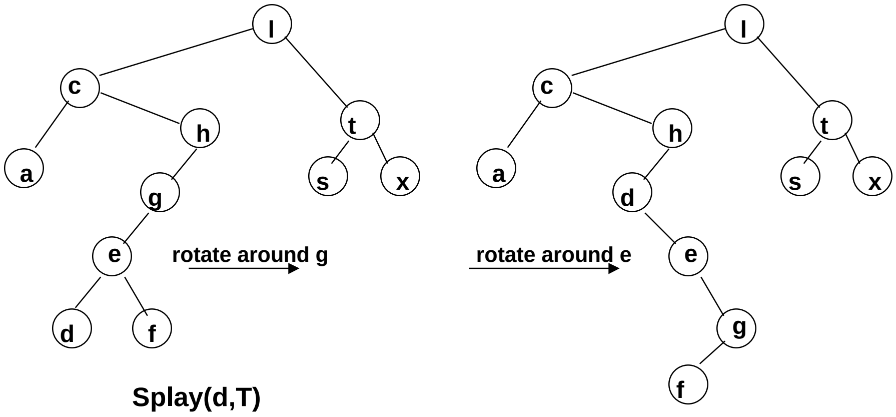
Splay rotations


G
X
P
a
b
c
d
X
G
P
a
b
c
d

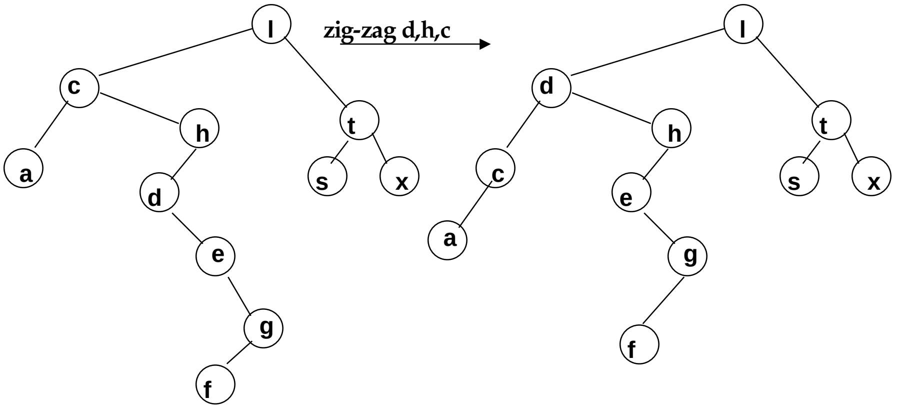
Splaying is mysterious

- The first and third cases are AVL tree single and double rotations, respectively, but case 2 is special to this algorithm
- A splay operation can increase the height of a BST
- A splay operation might decrease the balance of a BST.
- Nevertheless, any sequence of m operations starting from an empty tree and generating a sequence of trees having a maximum of n nodes will take exactly mlogn operations.

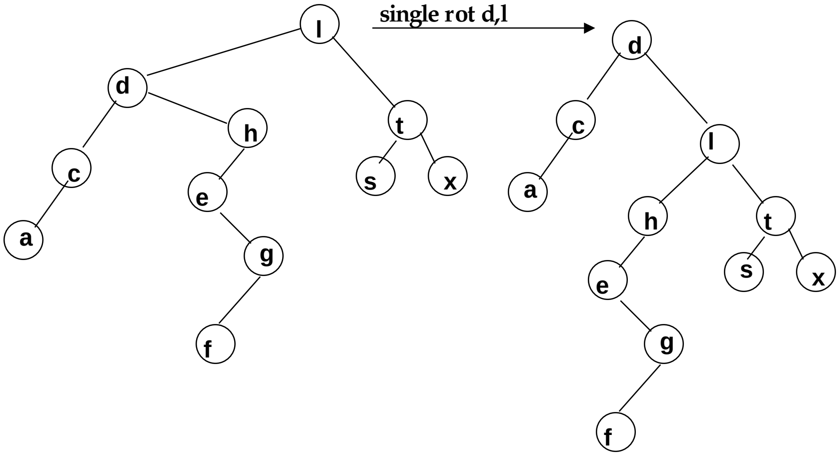
Splaying example


l
c
h
t
a
g
s
x
e
rotate around g
d
f
Splay(d,T)
l
c
h
t
a
d
s
x
rotate around e
e
g
f

Splay example


zig-zag d,h,c
l
c
h
a
d
e
g
f
t
s
x
l
d
h
c
a
e
g
f
t
s
x

Splay example

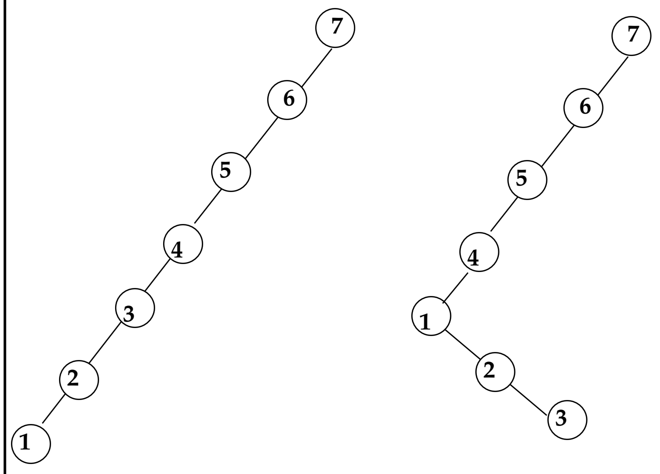

single rot d,l
l
d
h
c
a
e
g
f
t
s
x
d
c
a
l
h
e
g
f
t
s
x

Second example

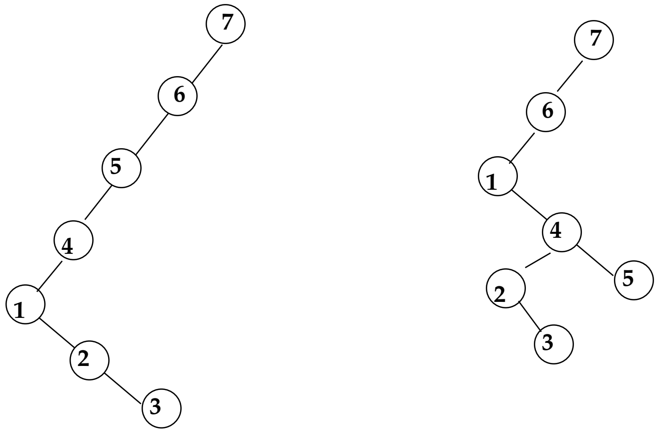
- ❑ Look at the effect of
  - – inserting keys 1,2,3,4,5,6,7 in order into an initially empty tree
  - – then accessing key 1, so we splay on 1.
- ❑ Each insertion will take constant time
  - – splay (k,T) will fail at the root, since k will always be greater than the key stored at the root, and the 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

---

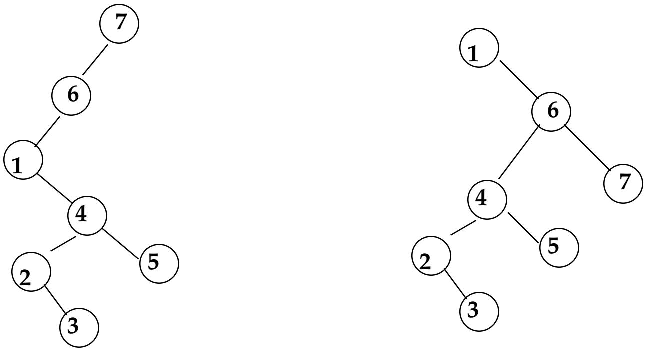
Second example



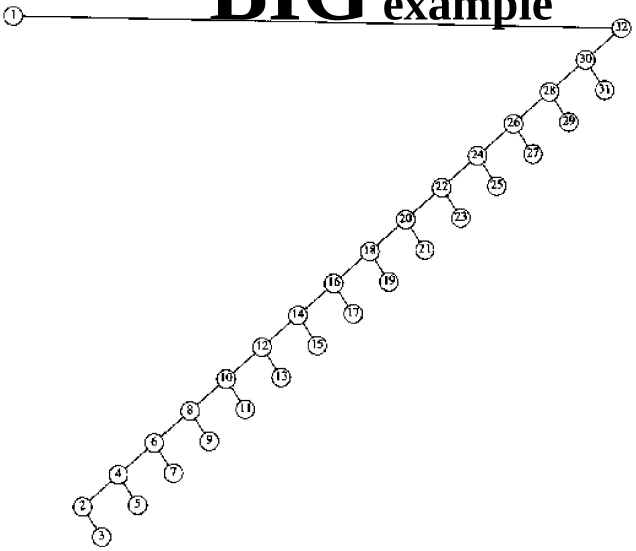
Second example

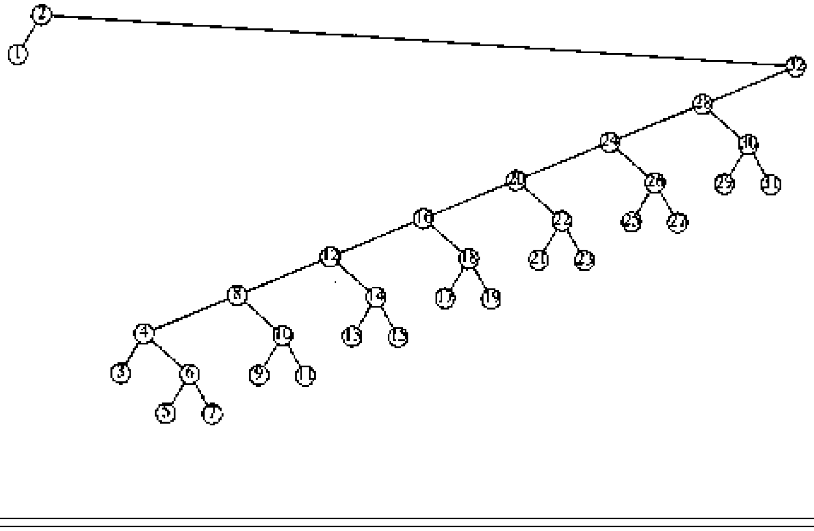

7
6
5
4
1
2
3
7
6
1
4
2
5
3

Second example

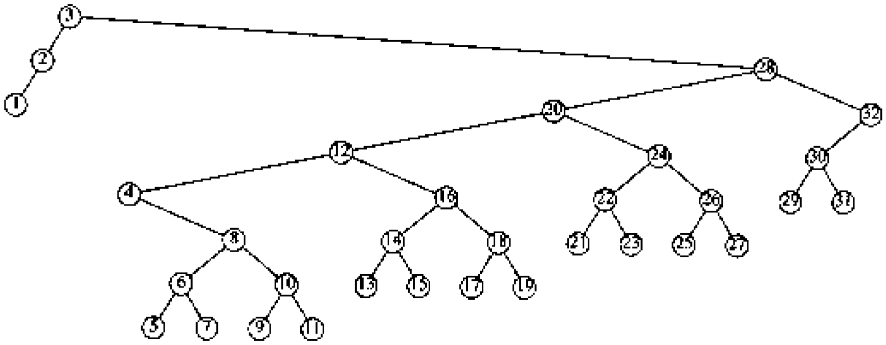

7
6
1
4
2
5
3
1
6
4
7
2
5
3

BIG example





Result of splaying previous tree at node 3


3
28
2
20
32
1
12
24
30
4
16
22
26
29
31
8
14
18
21
23
25
27
6
10
13
15
17
19
5
7
9
11

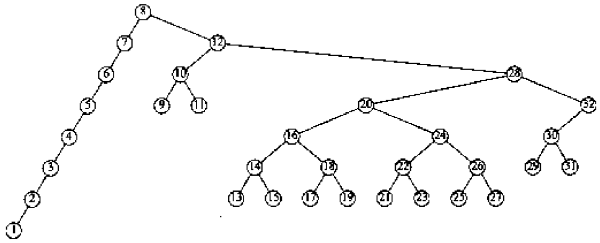

8
7
12
6
10
28
5
9
11
20
32
4
16
24
30
3
14
18
22
26
29
31
2
13
15
17
19
21
23
25
27
1

Figure 4.52 Result of splaying previous tree at node 8

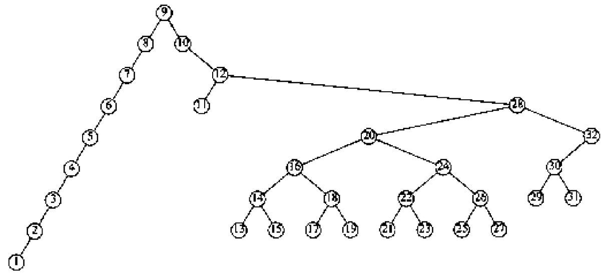

9
8
10
7
12
6
11
28
5
20
32
4
16
24
30
3
14
18
22
26
29
31
2
13
15
17
19
21
23
25
27
1

Figure 4.53 Result of splaying previous tree at node 9

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
```

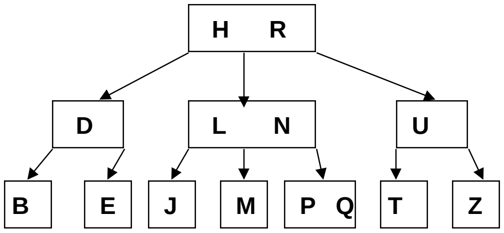
Zig left - single rotation between root and its left child

```
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



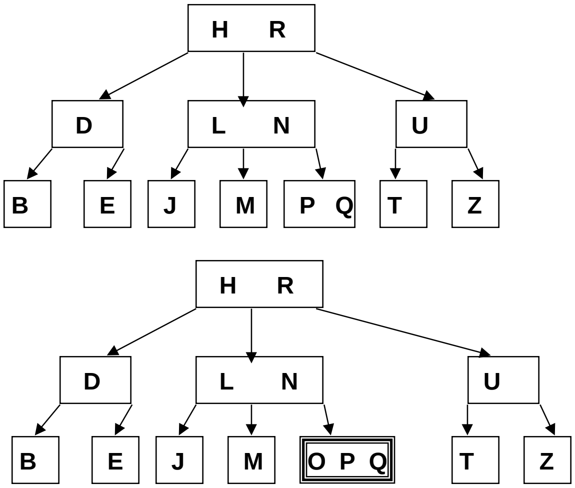
2-3 Trees

- 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 of the BST algorithm to handle the 3-nodes. But insertion and deletion are complicated

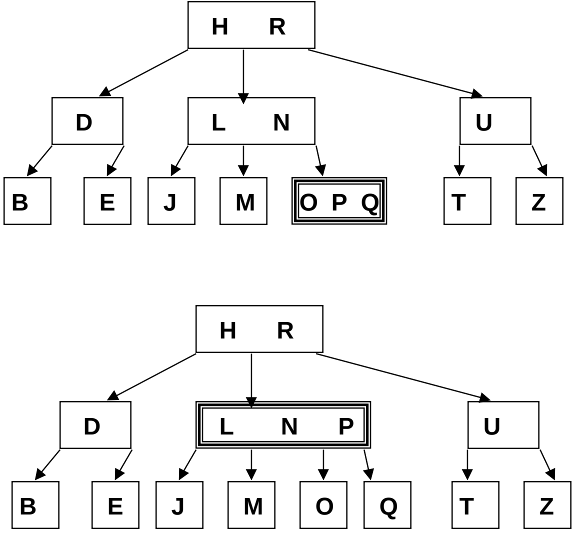
Insertion into 2-3 trees

- 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)
- 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.

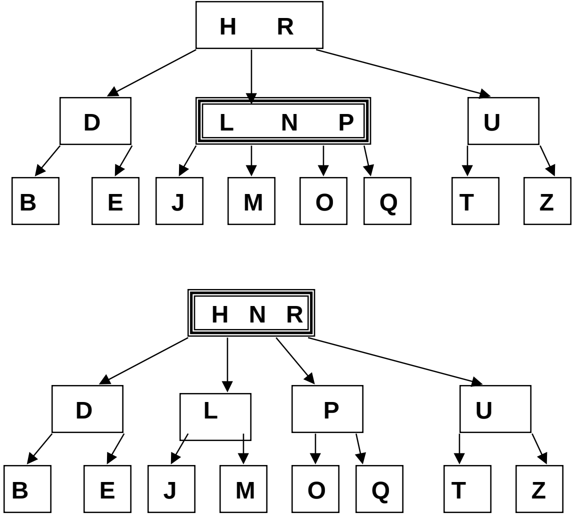
Example - insertion of O


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

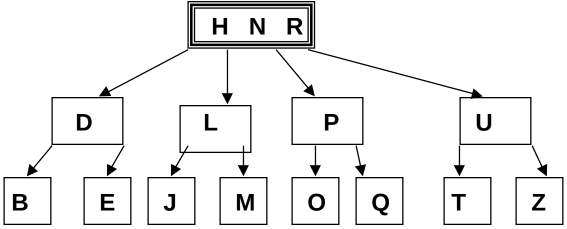
Example

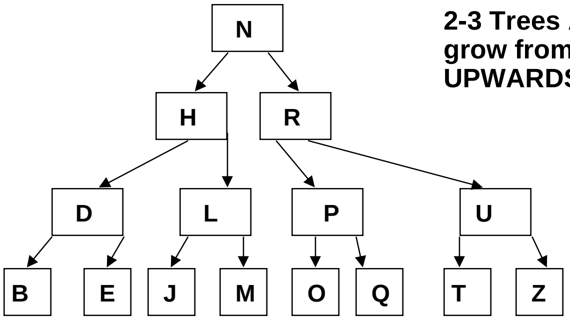

H R
D
L N
U
B
E
J
M
O P Q
T
Z
H R
D
L N P
U
B
E
J
M
O
Q
T
Z

Example


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

Example

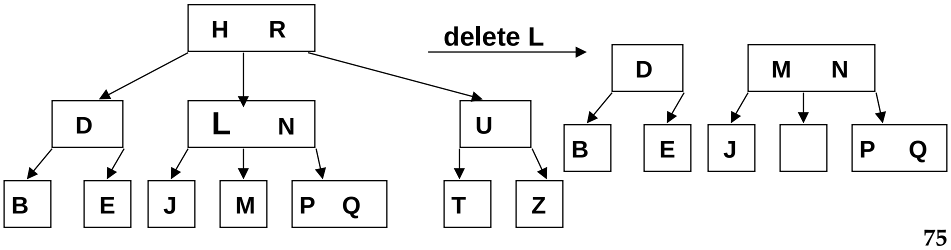

H N R
D
L
P
U
B
E
J
M
O
Q
T
Z


N
H
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). Replace the key by its inorder successor and remove the inorder successor from the leaf in which it was found.

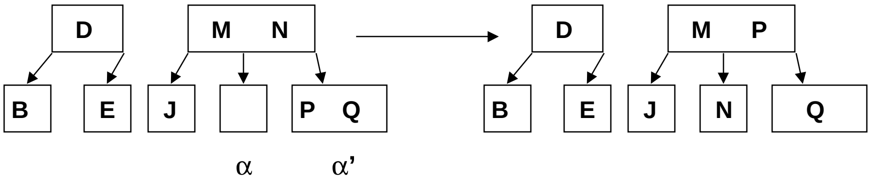


Deletion from 2-3 trees

- 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.
  - N 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.

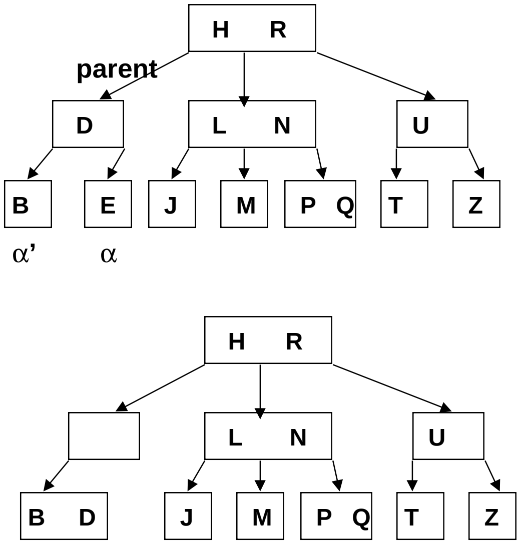
    If α and α′ are interior nodes, then also move one child of α′ to be a child of α. α and α′ end up with one key each, rather than 0 and 2, and the algorithm is complete.



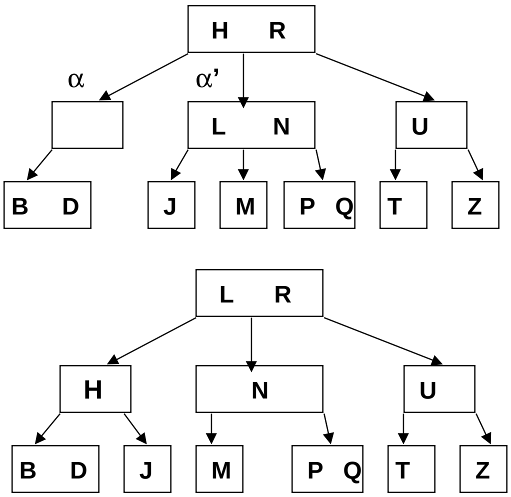
Deletion from 2-3 trees

- Continuing case of α having no keys
  - [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.

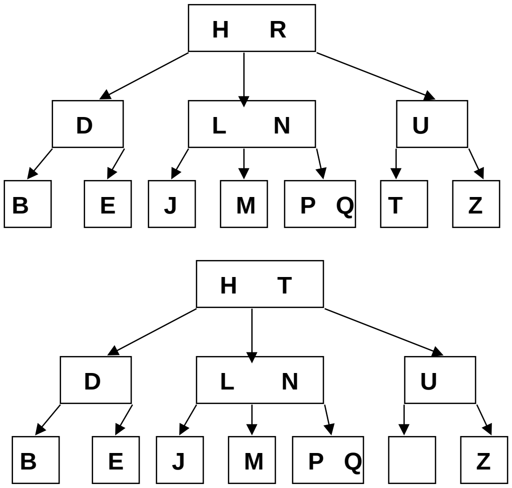
Example - deletion of E



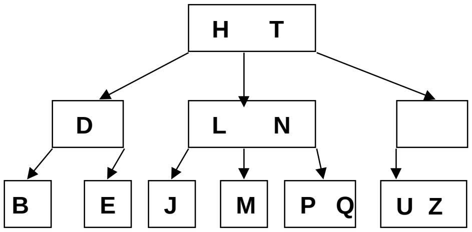
Example - deletion of E


H R
α
α'
L N
U
B D
J
M
P Q
T
Z
L R
H
N
U
B D
J
M
P Q
T
Z

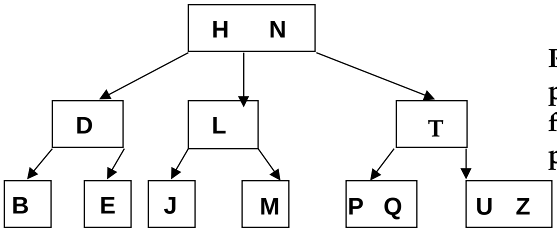
More examples - delete R


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

Delete R


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

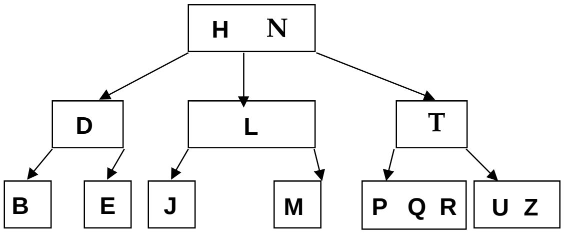
U was the key in parent that separated siblings; combine U with Z and remove U from parent - underflows parent.

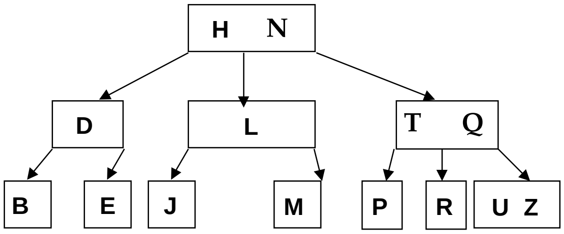

H N
D
L
T
B
E
J
M
P Q
U Z

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

---

Insert R


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


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

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

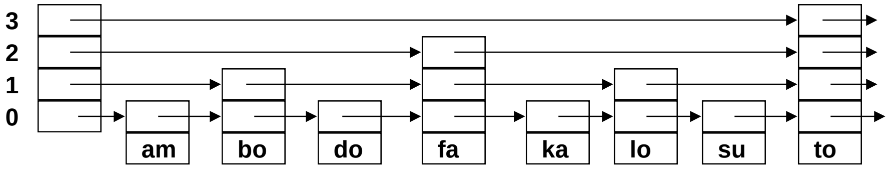
Difficulties with implementing 2-3 trees

- ❑ 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
  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

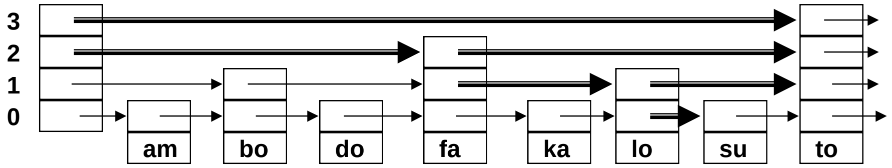
List array - example



Searching in a list array

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