Lesson 9
Tree Data Structures

Wholeness of the Lesson

In many cases, data structures in which searches, insertions, and deletions rely on a *linear orientation* run into limitations either in performance or in versatility of functionality. It is often possible to enhance performance or functionality in such cases using the *two-dimensional* structure of a *tree.*

- Use a binary search tree to enhance all list operations
- Use a heap to increase versatility of a queue

Science of Consciousness. To enhance performance and eliminate limitation in life itself, SCI suggests: Bring to the requirements of daily living a higher level of consciousness. This is achieved by taking inward dives to the transcendental field.

Overview of Lesson

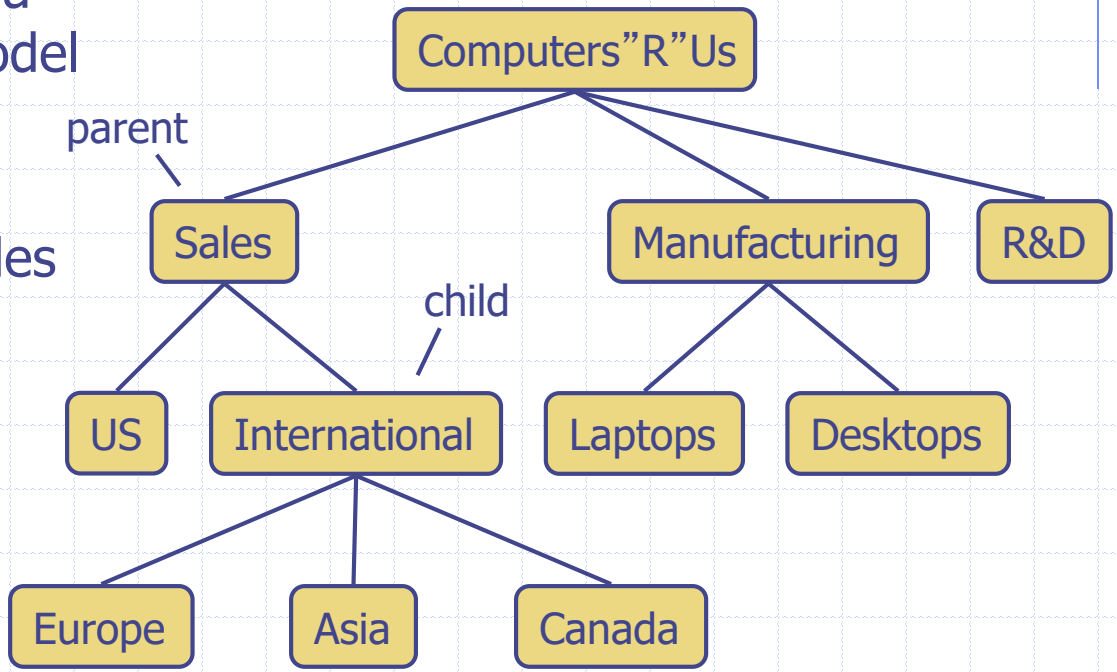
- Binary Search Trees – maintaining data in sorted order
- Red-Black Trees – avoiding worst-case performance of BSTs
- Priority Queues – enhancing functionality of Queues using a tree data structure (heaps)

The Problem of Keeping Sorted Data in Memory

- Question: What is the best data structure to keep data in sorted order in memory?
- How hard is it to implement this strategy: maintain sorted order to optimize searches ?
  - If we are using an ArrayList, then maintaining the list in sorted order is expensive because each time we add a new element, it must be inserted into the correct spot, and this requires array copy routines.
  - If we are using a LinkedList, it is easy to maintain sorted order since insertions are efficient, but searches are not very efficient.
    Exercise: What is the running time to carry out BinarySearch in a LinkedList?
- *The Need.* We need a data structure that performs insertions efficiently in order to maintain sorted order (like a linked list) but that also performs finds efficiently (as in an array list where binary search is highly efficient)
- The solution involves expanding from linear data structures to the two-dimensional structure provided by *trees.*

Review of Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



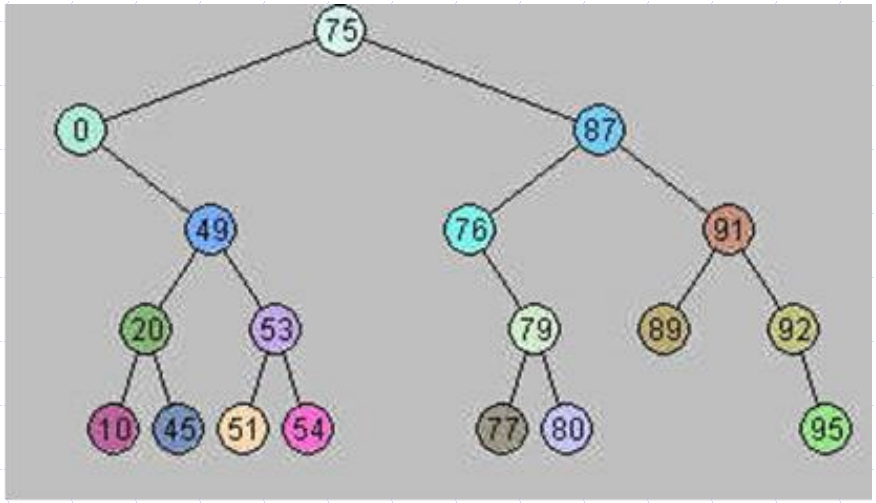
A Solution: Binary Search Trees

- A *binary tree* is a tree that each node has a reference to a left and right child node (though some references may be null).
- A *binary search tree* (BST) is a binary tree in which the BST Rule is satisfied:

  BST Rule:

  At each node N, every value in the left subtree at N is less than the value at N, and every value in the right subtree at N is greater than the value at N.

  For the moment, we assume all values are of type Integer.



Binary Search Trees

- The fundamental operations on a BST are:

```
public boolean find(Integer val)
public void insert(Integer val)
public boolean remove(Integer val)
public void print()
```

- When implemented properly, BSTs perform insertions and deletions faster than can be done on Linked Lists and performs any `find` with as much efficiency as BinarySearch on a sorted array.

- In addition, because of the BST Rule, the BST keeps all data in sorted order, and the algorithm for displaying all data in its sorted order is very efficient.

Insertion into a BST

Recursive algorithm for insertion of Integer x [an iterative implementation is given in the demo code]

- If the root r (of the tree being examined) is null, create a new root having value x.
- Otherwise:
  - If x is less than the value in the root, recursively insert into the left subtree below r
  - If x is greater than the value in the root, recursively insert into the right subtree below r.
- Exercise: Insert the following into an initially empty BST: 1,8,2,3,9,5,4.

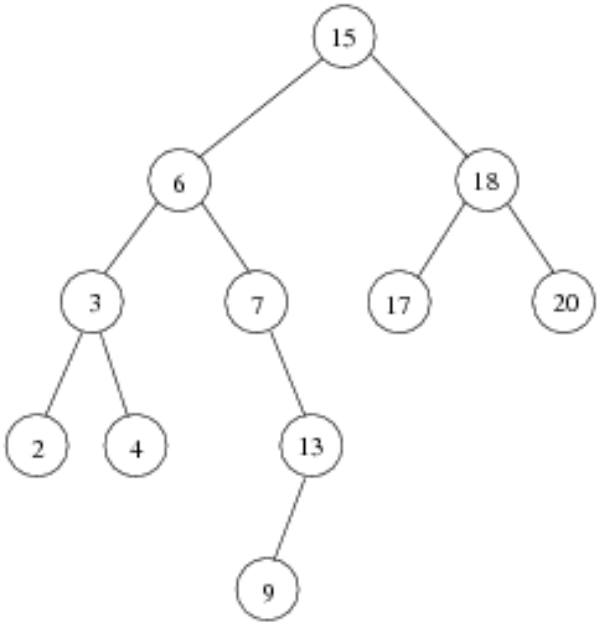
Searching a BST

Recursive algorithm for finding an Integer x

The recursive *find* operation for BSTs is in reality the binary search algorithm in the context of BSTs.

1. If the root is null, return false
2. If the value in the root equals x, return true
3. If x is less than the value in the root, return the result of searching the left subtree
4. If x is greater than the value in the root, return the result of searching the right subtree

Example: search for 13



BST In-Order Traversal

Recursive print algorithm – outputs values in every node in sorted order (or can load into a list)

1. If the root is null, return
2. Print the left subtree of the root
3. Print the root
4. Print the right subtree of the root

Note: There are several commonly used procedures for visiting all nodes in a binary tree: *in-order traversal, pre-order traversal, post-order traversal*, and others. In this course we will only discuss in-order traversal since it is the only one that outputs elements in sorted order.

Note: There is also an iterative algorithm for in-order traversal but it is complicated. Please do not attempt to use an iterative version on exams – we will use only the recursive version given here in this class.

Deletions in a BST

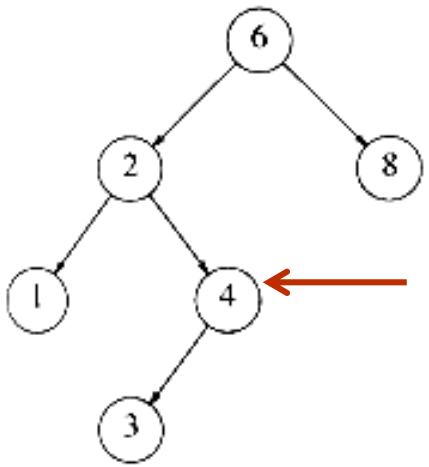
Algorithm to remove Integer x

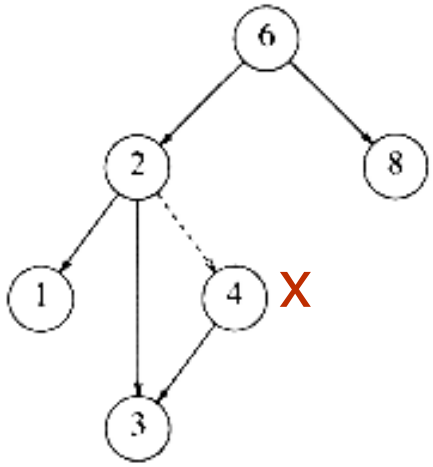
- Case I: Node to remove is a leaf node

  Find it and set to null

- Case II: Node to remove has one child

Create new link from parent to child, and set node to be removed to null

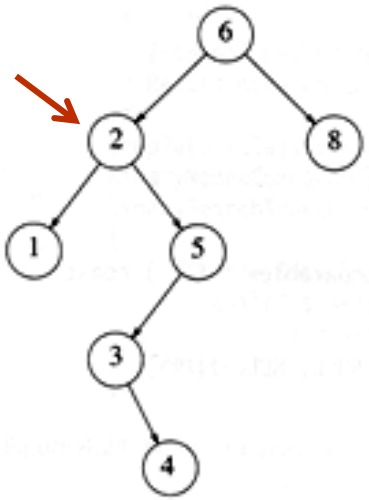


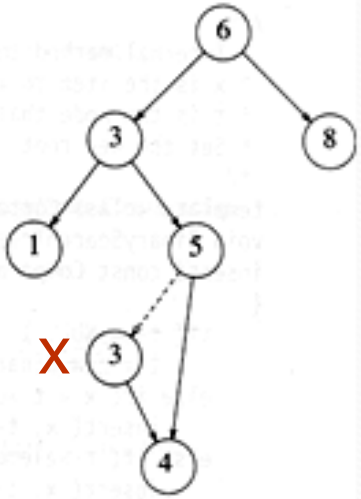


Deletions in a BST - continued

Algorithm to remove Integer x

- Case III: Node to remove has two children
  1. Find smallest node in right subtree – say it stores an Integer y. This node has at most one child
  2. Replace x with y in node to be removed. Delete node that used to store y – this is done as in one of the two cases above





Running Time Analysis for BSTs

- A worst case for BSTs occurs when the insertion sequence is sorted or reverse-sorted.
- In the worst case, a BST containing n elements will have height n-1. Therefore, worst-case performance of each of the operations find, insert, delete is $\Theta(n)$.

Average Case Analysis of BST Operations

- What does "average-case" mean in the case of BSTs? There is more than one interpretation, though in each case the result is the same. A BST with *n* nodes is said to be *randomly built* if *n* distinct integers are randomly chosen and inserted successively into an initially empty BST
- The main result is:

  The average depth of a node in a randomly built BST having n nodes is O(log *n*).

- Therefore, the operations of insert, delete ,and search perform in O(log n) on average.
- It can be shown that in-order-traversal performs in O(n) on average.

Handling Duplicates in a BST

- Running times of find and insert given above, and their implementations, make the assumption that there are no duplicate values.
- To handle duplicate values, store in each Node a List (instead of a value); then when a value is added to a Node, it is instead added to the List.
- Example: Suppose we are storing Employees in a BST, ordered by name. Our BST will be created so that the value in each node is a List<Employee> instance. Then, after insertions, all Employees with the same name will be found in a single List located in a single Node.

Overview of Lesson

- Binary Search Trees – maintaining data in sorted order
- Red-Black Trees – avoiding worst-case performance of BSTs
- Priority Queues – enhancing functionality of Queues using a tree data structure (heaps)

Balanced BSTs

- In order to avoid worst case scenarios for BSTs, several kinds of *balanced* BSTs have been devised. Typically, these work by formulating a *balance condition* that ensures that, as long as the condition is satisfied, the BST (having n nodes) must have height O(log n). The balance condition is forced to be true by performing balancing steps after every insertion and deletion.
- The most efficient kind of balanced BST is called a *red-black tree.*

Introduction to Red-Black Trees

A BST is *red-black* if it has the following 4 properties:

- Every node is colored either red or black
- The root is colored black.
- If a node is red, its children are black.
- For each node *x*, every path from *x* to a NULL reference has the same number of black nodes.

  The number of black nodes on any path from *x* to a NULL is called the *black height* of the tree and is denoted bh(x)

Red-Black Tree Exercise

Num nodes *n*	Does there exist a red-black tree with *n* nodes, all of which are black?
1	Yes
2	
3	
4	
5	
6	
7	

Num nodes *n*	Does there exist a red-black tree with *n* nodes, where exactly *one* of the nodes is red?
1	No
2	
3	
4	
5	
6	
7	

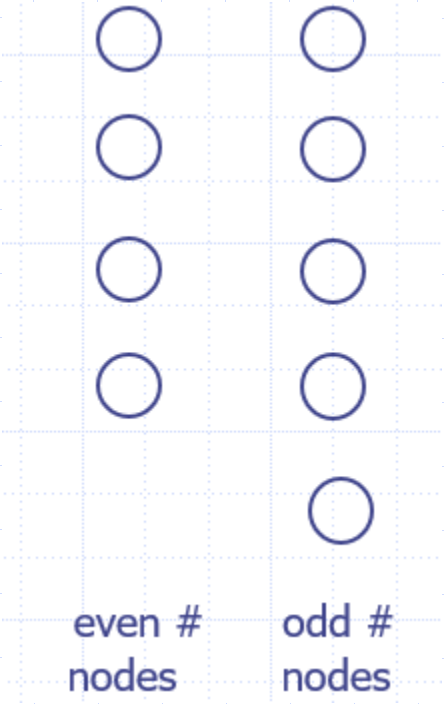
Height of a Red-Black Tree

Suppose T is red-black and has n nodes and height h and root r.

1. Fact 1: $bh(r) \geq h/2$
2. Fact 2: $n \geq 2^{bh(r)} - 1$ (consider some examples)
3. Theorem: $h \leq 2\log(n+1)$

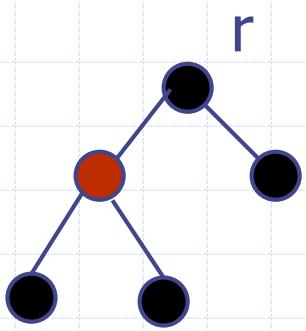
Proof:

$$
\begin{aligned}
n &\geq 2^{bh(r)} - 1 \\
&\geq 2^{h/2} - 1 \\
n + 1 &\geq 2^{h/2} \\
\log(n+1) &\geq h/2 \\
2\log(n+1) &\geq h
\end{aligned}
$$

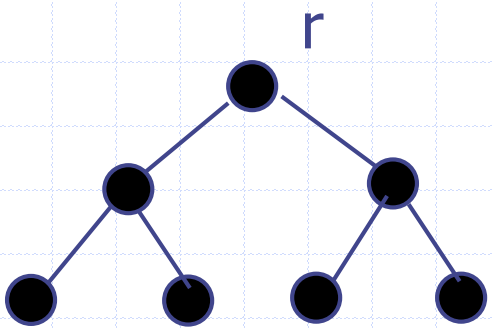


Examples of Fact 2

$$n \geq 2^{bh(r)} - 1$$



n = 5
bh(r) = 2
h = 2
$5 \geq 2^2 - 1$



n = 7
bh(r) = 3
h = 2
$7 \geq 2^3 - 1$

Operations on a Red-Black Tree

- Searches in a red-black tree are performed in the same way as for any BST. Running time for a search is proportional to the height of the tree. Since the height of a red-black tree is O(log n), worst-case running time for the search operation is O(log n).
- Insertions and deletions are also done in the same way as for any BST except that there are additional steps which guarantee that after insertion (or deletion) the tree continues to be red-black.
- It can be shown that the extra work to ensure the tree remains red-black after insertion or deletion is at worst O(log n). Therefore, running time for insertion and deletion is O(log n)

Main Point

Because of their balance condition, red-black trees always have height O(log n), so their primary operations all have running times that are O(log n). Techniques for maintaining the red-black properties after insertions and deletions are techniques that maintain balance in the midst of change.

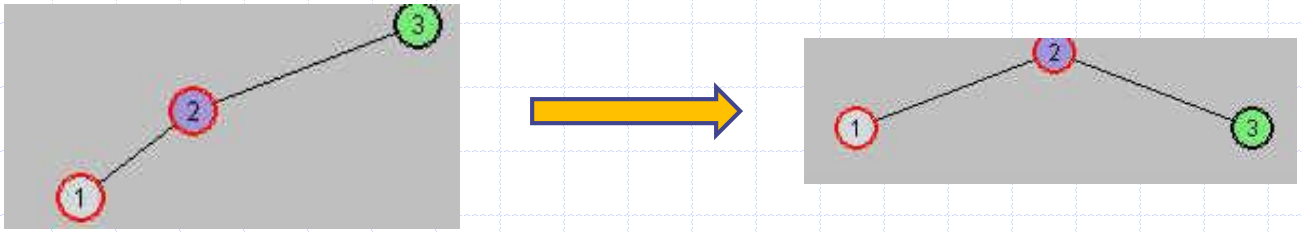
Science of Consciousness: "Far away indeed from the balanced intellect is action devoid of greatness." (Gita, II.49) The "balanced intellect" is a state of life in perfect balance in which each area of life from most expressed to most subtle and refined is spontaneously given due attention.

Preparing to Insert into a Red Black Tree: *Rotations*

- Introducing the primary re-balancing technique: *rotations*
  - Used to keep a red-black tree balanced
  - It has the effect of raising the height of some nodes and lowering others, but preserving the BST and red-black properties

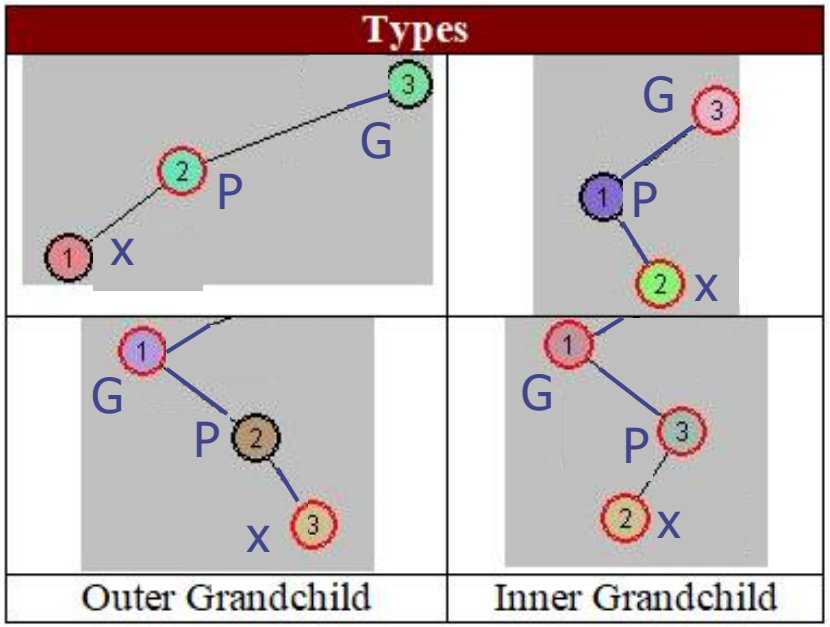
Examples of Rotations on a BST

a. Perform a right rotation of tree obtained from 3,2,1 insertion. One says: *a right rotation of 2, 3 with top 3*



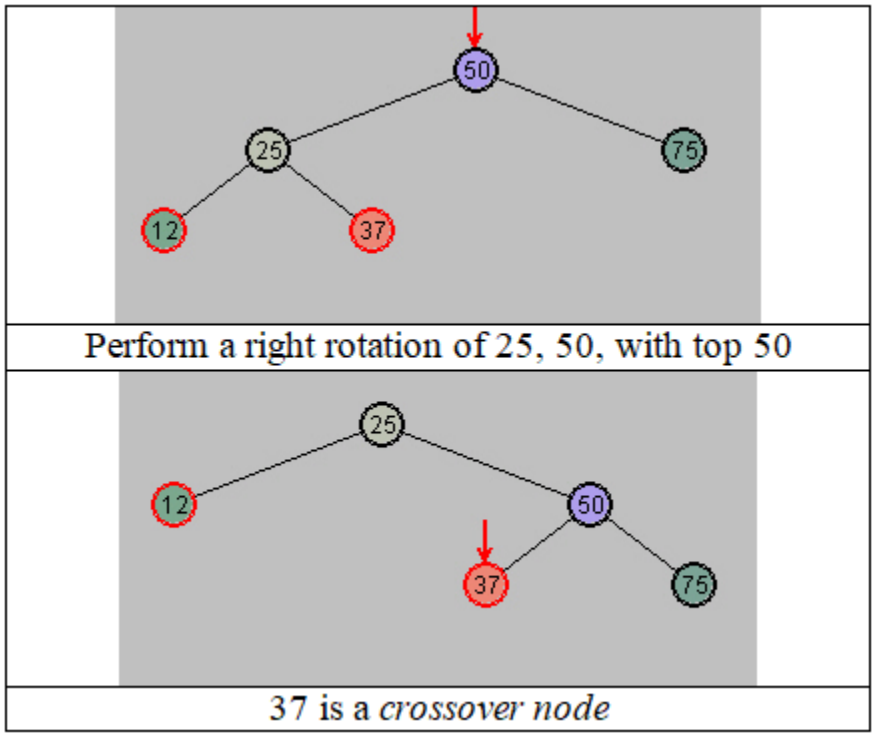
Examples of Rotations

b. *Rotations involving a grandchild.* Often need to move a node X, and the parent P and grandparent G need to be examined. Relative to the grandparent, X can be an *outer grandchild* or an *inner grandchild.*



Examples of Rotations

c) In complicated rotations (for instance, in a right rotation where the child being lifted already has a right child), there may be a *crossover node.*



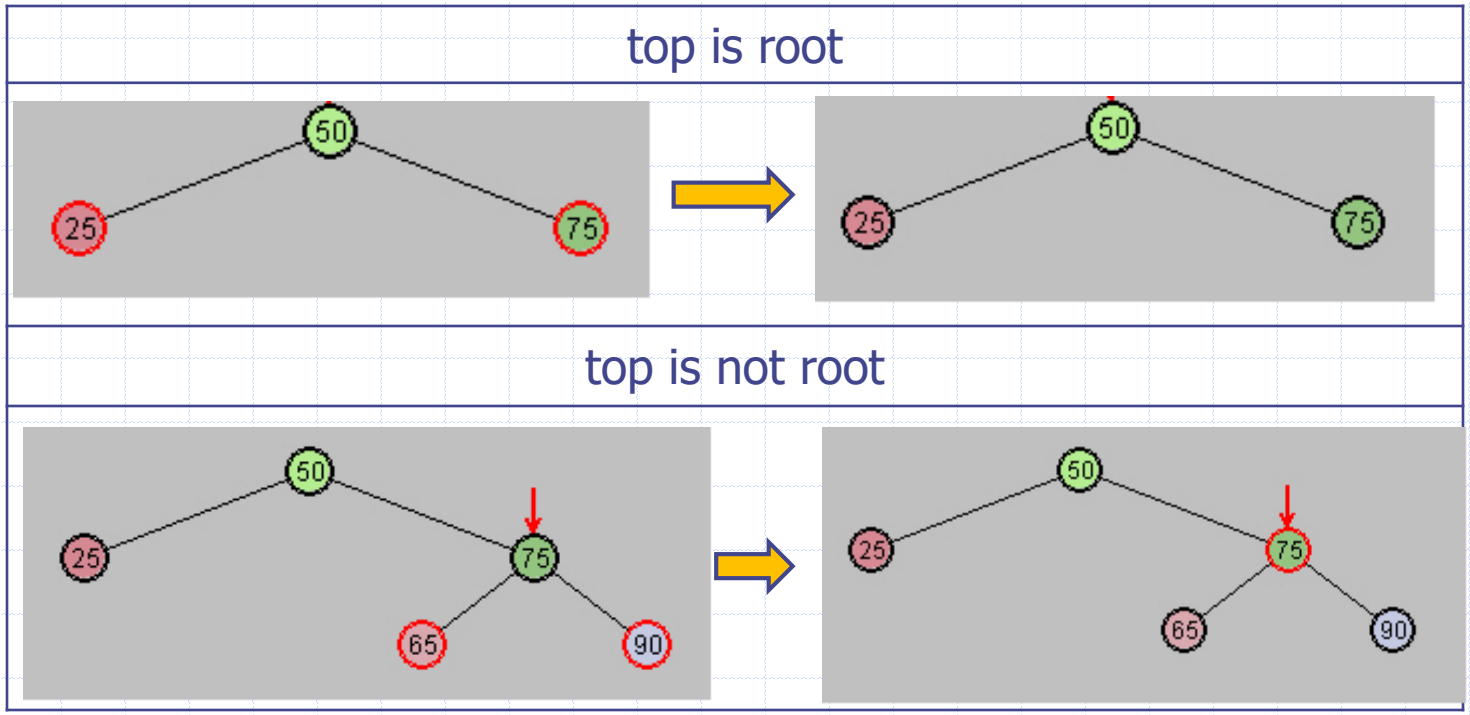
Top-Down Insertion in a Red-Black Tree

1. Basic idea:
   a. To insert a new node, attempt to locate proper location using usual BST algorithm
   b. On the way down the tree, make "preventive" adjustments
   c. Perform the insertion
   d. Correct any red-red violations near the insertion point

2. More detail:
   a. On the way down the tree, perform color flips and rotations
   b. Insert new node
   c. Make further adjustments: color changes and rotations

Color Flips on the Way Down

Color Flip Strategy: During the search for the insertion point, when a black node having two red children is encountered, a color flip is done. A color flip changes colors of all three nodes unless the top node is the root, in which case only the children's nodes change color.



Color Flips on the Way Down

- This color-flip strategy is used as a preventive measure to make sure that adjustments needed after insertion don't propagate up the tree.
- After a color flip, all red-black tree properties continue to hold except for the property that red nodes must have black children.
- In particular, a color flip can introduce a *red-red violation*, which must be corrected to maintain red-black property. Before moving further to the insertion point, the violation must be corrected, using one or two *rotations.*

Rotations on the Way Down

- These are done to correct red-red violations that occur from color flips. It can be shown that after a color flip, followed by one of the color change/rotation combinations mentioned below, the tree is once again red-black.
- Case 1 Outside grandchild causes a red-red violation after color flip. The Rule:

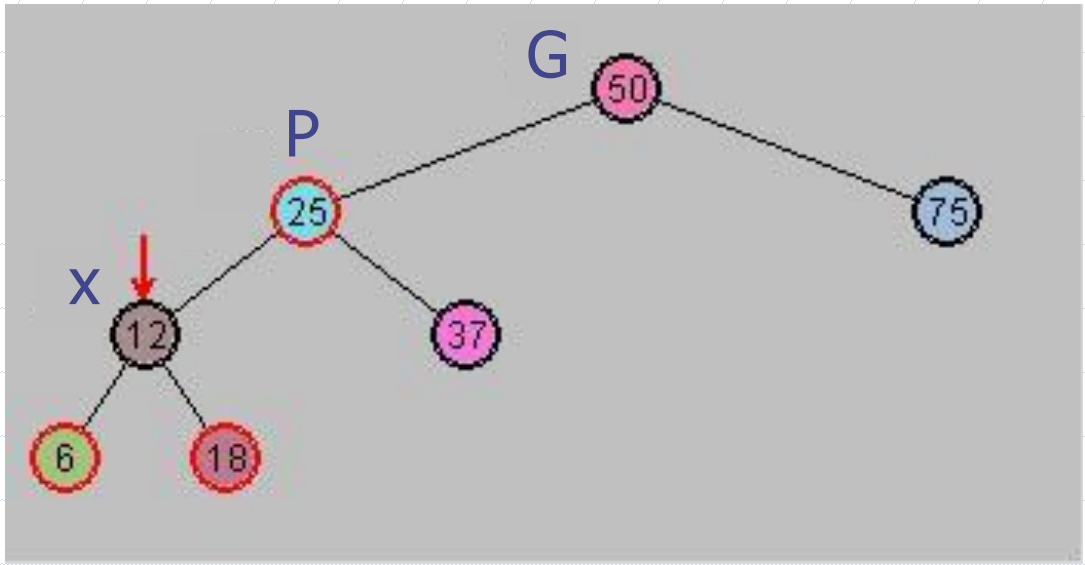
  - Change color of G
  - Change color of P
  - Rotate P, G in the direction that lifts X

- Case 2 The inside grandchild causes a red-red violation. The Rule:

  - Change color of G
  - Change color of X
  - Perform double rotation:
    - P, X, lifting X
    - G, X, lifting X

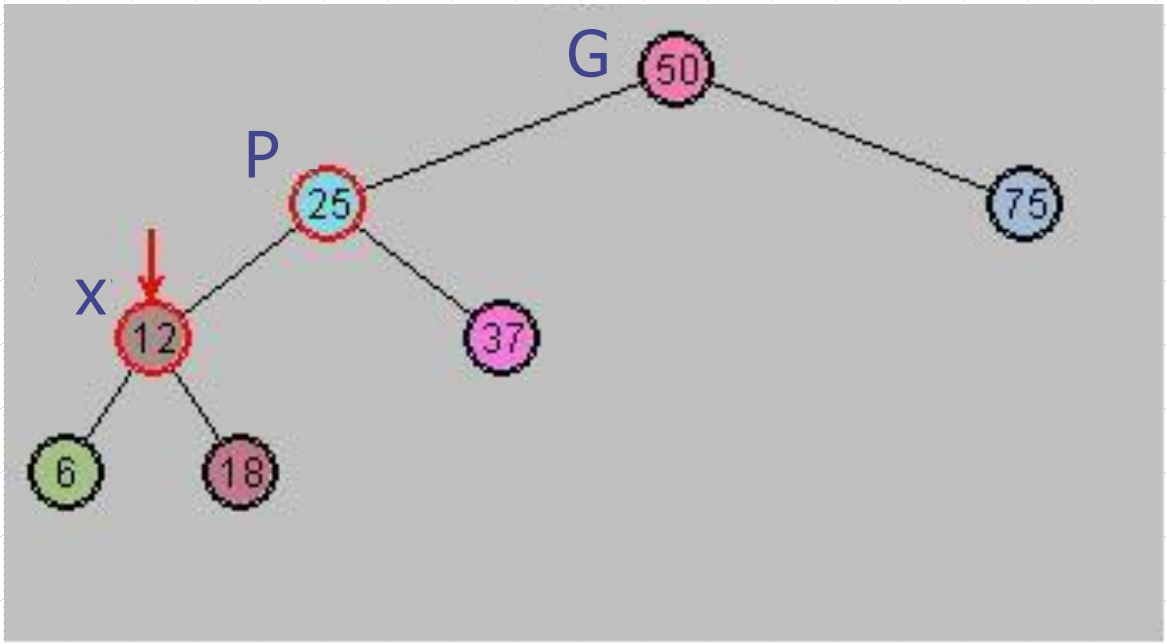
Example for Case 1

- We wish to insert the node 3, but we first notice need for color flip



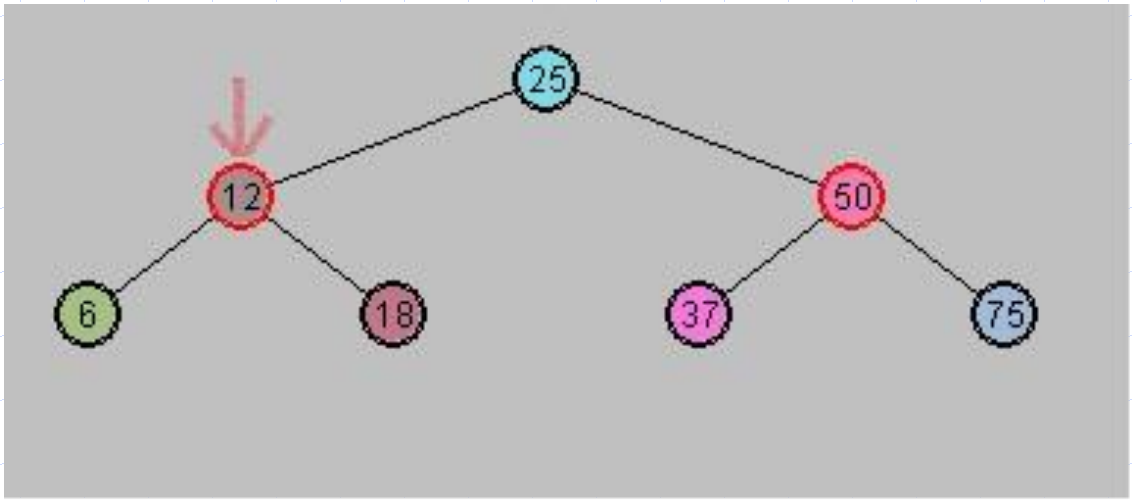
Example for Case 1 - Continued

- After color flip, we encounter red-red violation caused by outer grandchild



Example for Case 1 - Continued

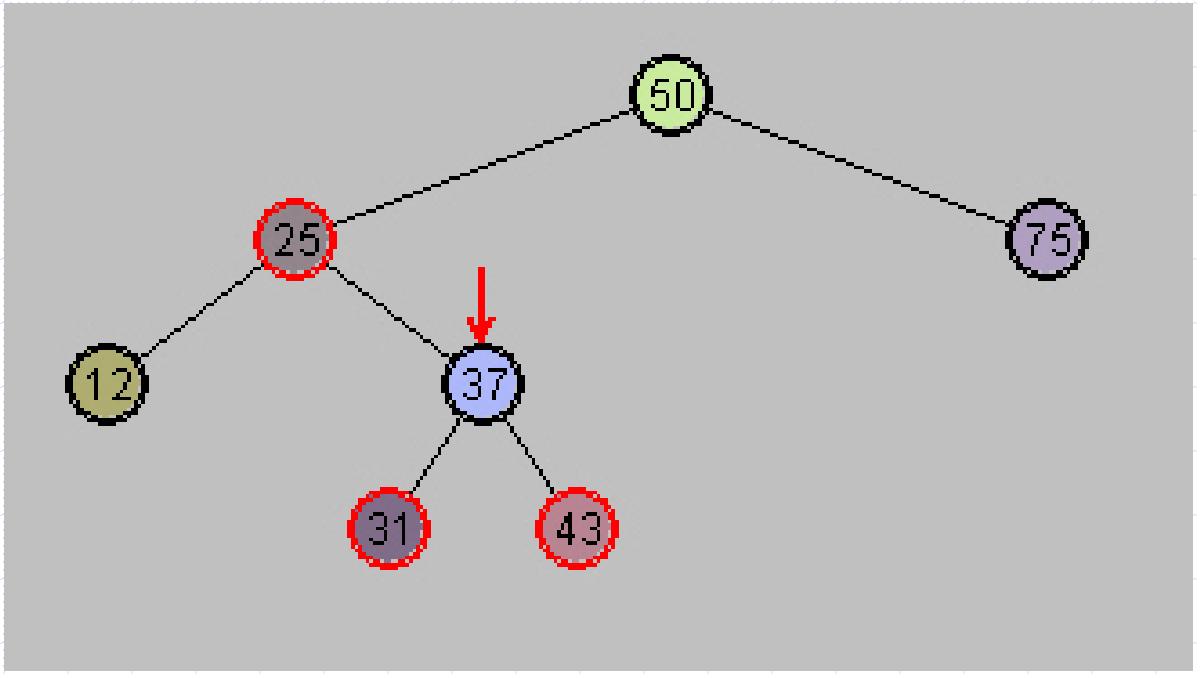
- We apply the rule for this case:



- Red-red violation has been eliminated and tree is now balanced.

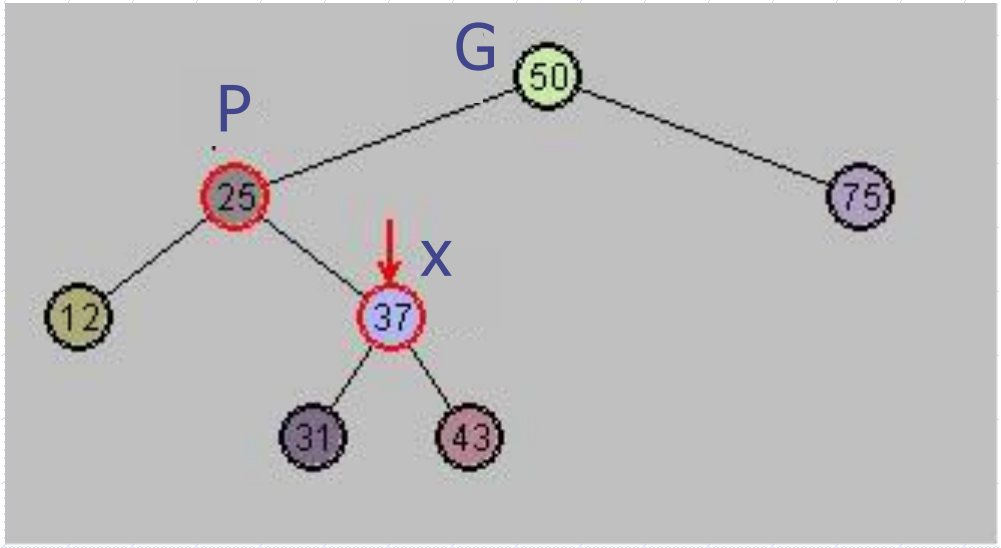
Example for Case 2

- Suppose we are trying to insert the node 28.

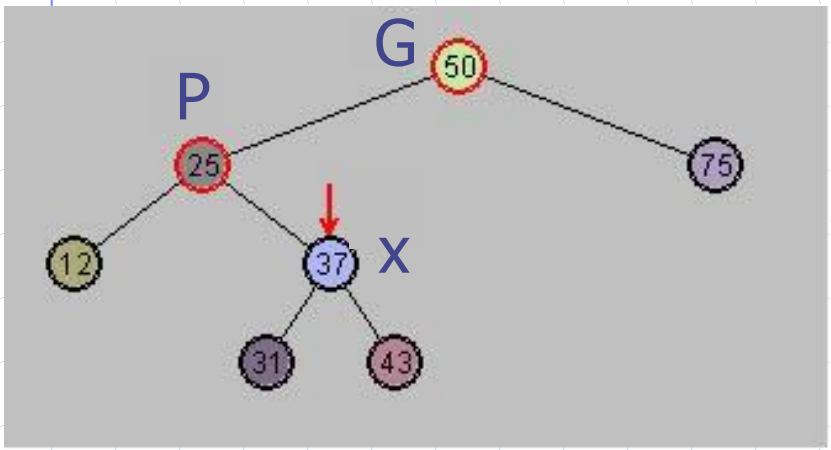


Example for Case 2 - Continued

- Need to do a color flip which leads to a red-red violation caused by inner grandchild.

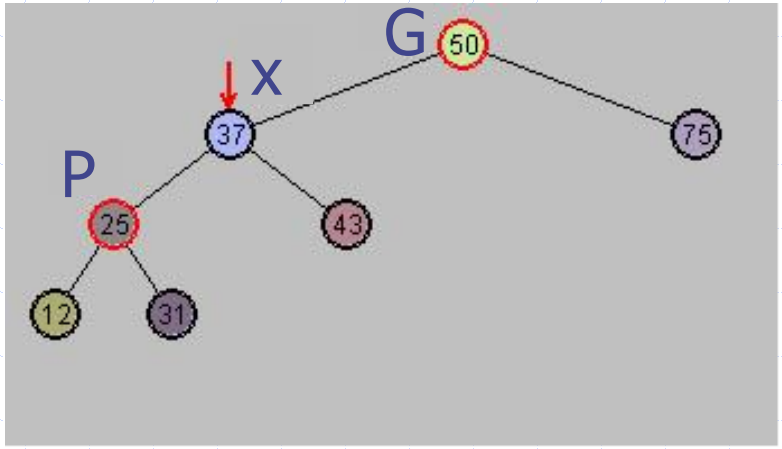


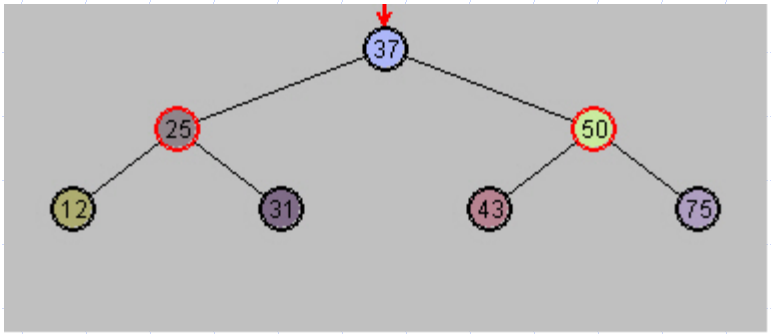
Example for Case 2 - Continued



1. First change colors of G, X

2. Rotate X,P in the direction that lifts X.





3. rotate X, G in the direction that lifts X. Tree is now balanced. Can now insert 28.

Insertion And Corrections

- Nodes are always inserted as red nodes. This can lead to further red-red violations.
- Three cases when new node X is inserted.
  - P is black, so no adjustment necessary (since X is red)
  - P is red and X is an outside grandchild. The Rule:

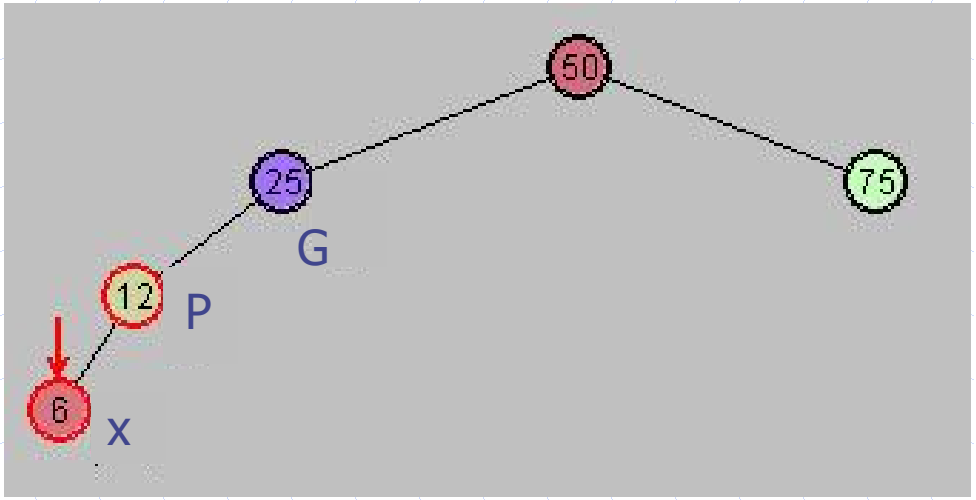
    - Change color of G
    - Change color of P
    - Rotate P, G in direction that lifts X

  - P is red and X is an inside grandchild. The Rule:

    - Change color of G
    - Change color of X
    - Double rotation:
      - P, X, lifting X
      - G, X, lifting X

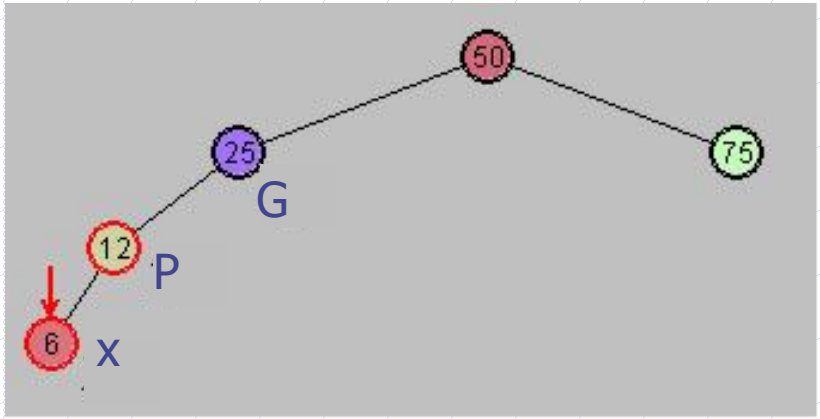
Example

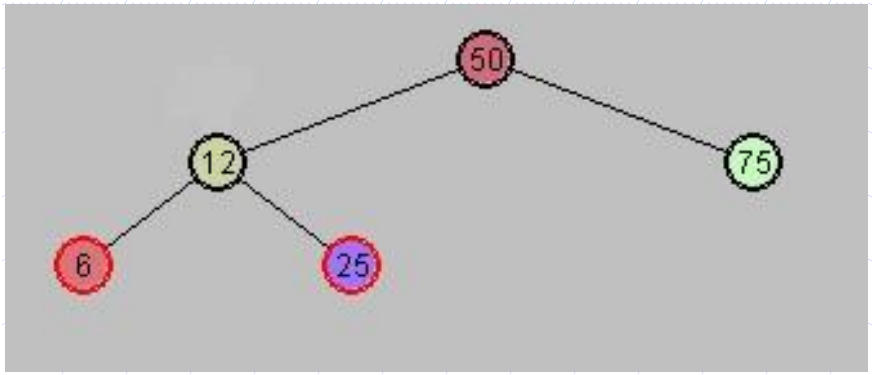
- Insert 6 to the tree. Insertion results in outer grandchild red-red violation.



Example (continued)

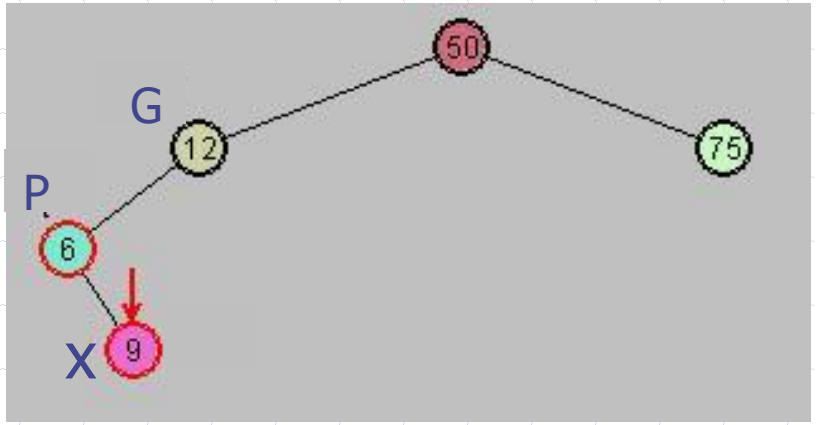
- Insertion results in outer grandchild red-red violation.





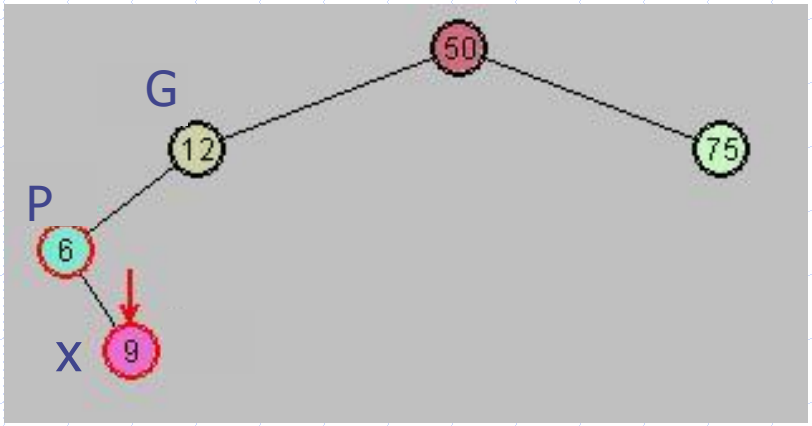
Example

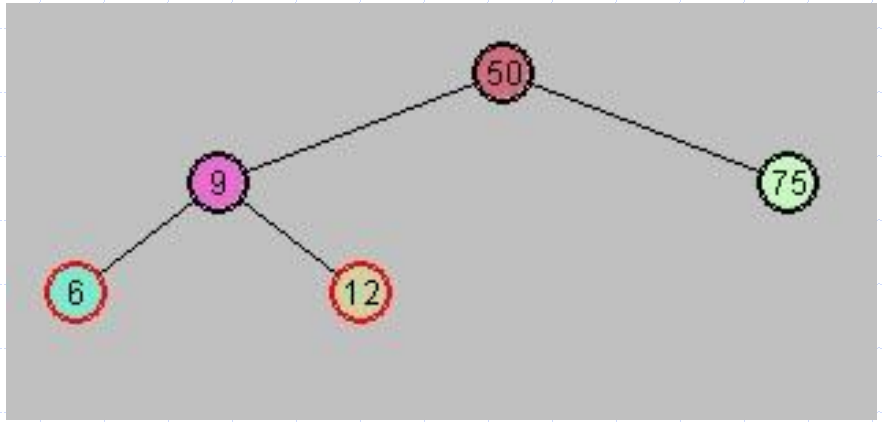
- Insert 9 to the tree. Insertion results in inner grandchild red-red violation.



Example (continued)

- Insertion results in inner grandchild red-red violation.





Exercise

- Build a red-black tree using the insertion sequence 1, 5, 2, 4, 3

Conclusions

- Insertions of a node at depth d require
  - O(d) to make flips and adjustments on the way to insertion point
  - O(1) to make final color changes and adjustments

- Therefore, worst case running time for insertions:

  find insertion point: O(log n) (since height of tree is O(log n)

  corrections via rotations and color changes:

  O(log n)

  Total running time: O(log n)

- Deletions can likewise be shown to take O(log n). Since the algorithm requires several cases and is a bit complicated, we do not go through it here.

Overview of Lesson

- Binary Search Trees – maintaining data in sorted order
- Red-Black Trees – avoiding worst-case performance of BSTs
- Priority Queues – enhancing functionality of Queues using a tree data structure (heaps)

Enhancing Queues Using a Tree Data Structure: *Priority Queues*

- In an ordinary queue, elements are removed from the queue in the order in which they were originally inserted.
- Sometimes, a different order of removal would be preferable. Example: In an operating system, a time slice is given to a requesting process according to priority rather than according to the order in which processes place requests.
- Priority Queues support the notion that highest priority objects are removed first.

Priority Queue ADT

- *Elements of form (k,e).* In order to specify priorities, each object to be inserted into the queue is equipped with a key – its "priority". Therefore, typical elements of the queue are pairs (k,e) where k is the key (like an integer) and e is the element. (Duplicate keys are allowed.)
- *Operations.* A Priority Queue has two main operations:

`insertItem(k,e)` – inserts the pair (k,e)

`removeMin()` – removes a pair (k,e) for which k is smallest and returns e

Auxiliary Operations:

`minElement()` – returns (but does not remove) an element with smallest key

`minKey()` – returns (but does not remove) the smallest key

Implementing Priority Queues with *Heaps*

- A heap is a binary tree that has the following structural property and ordering property.
  - *Heap Structural Property:* Structurally, a heap is a binary tree in which every level except possibly the bottom level is filled completely, and the bottom level is filled from left to right. (Such trees are sometimes called *complete.*)
  - *Min-Heap Order Property:* A key in one node n must always be greater than or equal to the key in the parent of n (if there is a parent).

Note: The Min-Heap Order Property ensures min key is at the root. A variant is the *Max-Heap Order Property* (a key in one node n must always be less than or equal to the parent key) which ensures the *max* key is at the root.

Running Time of Heap Operations

- **Definition**. A binary tree is *completely filled* if, in each nonempty level k of the tree, there are 2^k nodes.
- **Theorem**. Suppose T is a completely filled binary tree having n nodes and having height h. Then
  - $n = 2^{h+1}-1$
  - T has exactly 2^h leaves.

Proof. The fact that T has 2^h leaves follows from the definition of "completely filled" and the fact that level h of the tree consists of the leaves of T. For the first part, proceed by induction on h. The base case $h = 0$ is obvious. Assume the result for completely filled binary trees of height $< h$. T has 2^h leaves (by the first part). If we obtain T' from T by removing its leaves, every nonempty level k of T' still has 2^k nodes, so T' is completely filled. Therefore, by induction hypothesis, T' has $2^h - 1$ nodes, and also 2^{h-1} leaves. If we add $2 * 2^{h-1} = 2^h$ children to the leaves of T', we reconstruct T and have $n = 2^h - 1 + 2^h = 2^{h+1} - 1$ nodes.

(continued)

- The structural property implies that if the heap has height h with n nodes, then

$$2^h \leq n \leq 2^{h+1} - 1$$

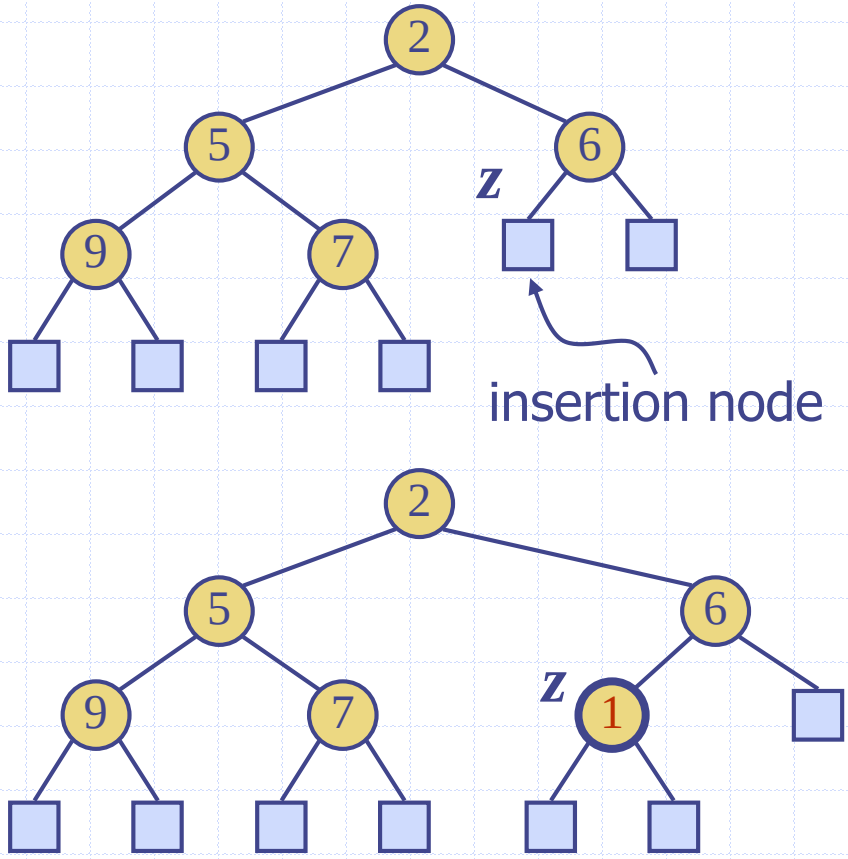
Taking logs establishes:

Theorem. If T is a heap with n nodes and height h, then h is O(log n) (with small constant factor).

- Consequently, when a heap is used to implement a PriorityQueue, removeMin and insertItem run in O(log n) and the constant factors are smaller than if a Red-Black tree is used.

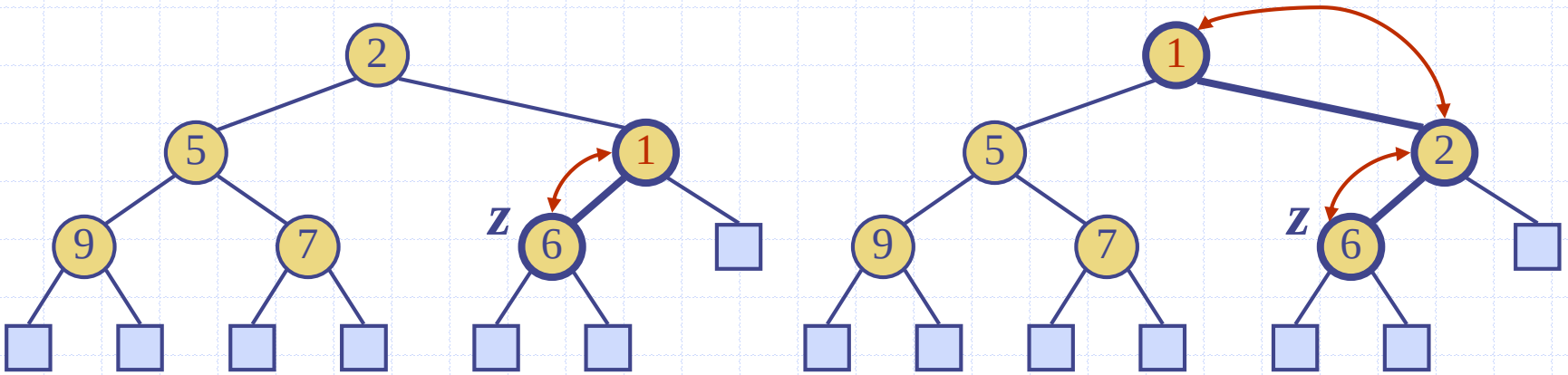
`insertItem` Using Heaps

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z -- the new last node (implementation: maintain a pointer to current last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



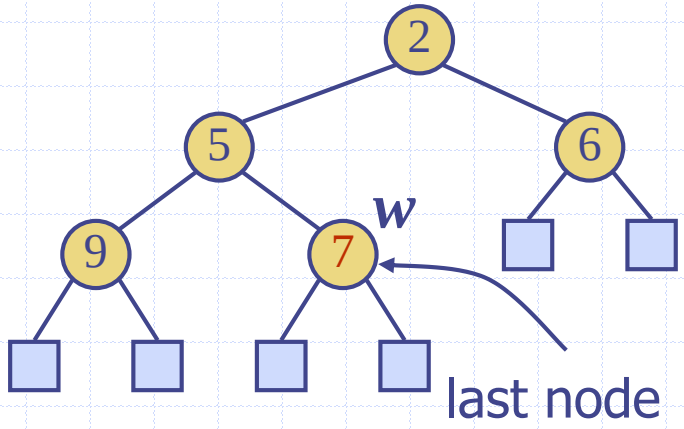
Maintaining Heap Order: Upheap

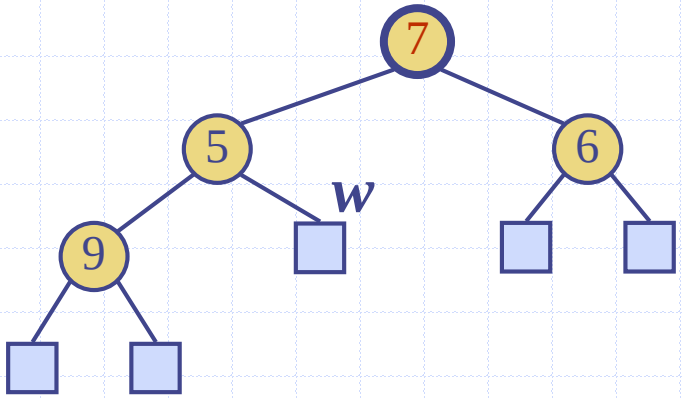
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
- NOTE: During upheap, a swap occurs only if the key k that is being moved upward is *strictly less than* its parent; this fact is important when heaps are used for sorting (discussed later in these slides).



`removeMin` Using Heaps

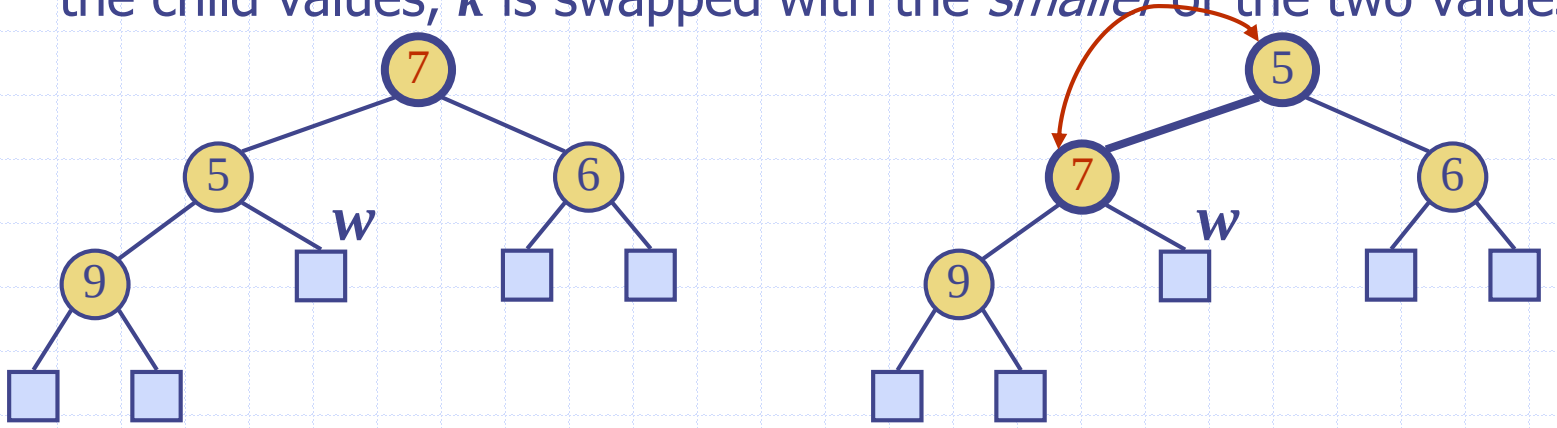
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Make the last node null
  - Restore the heap-order property (discussed next)





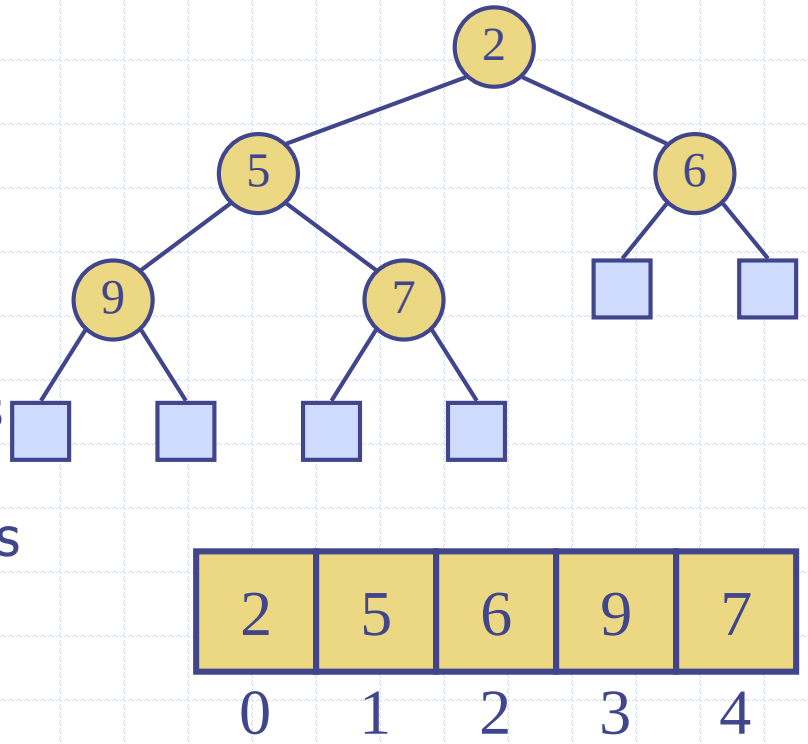
Maintaining Heap Order after Removal: Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time
- NOTE: In the downward path, whenever key k is larger than both the child values, k is swapped with the *smaller* of the two values.



Implementing a Heap in an Array

- We can represent a heap with n keys by means of an array of length n
- For the node at position i
  - the left child is at position $2i+1$
  - the right child is at position $2i + 2$
  - the parent is at position $(i - 1)/2$.
- Operation insertItem corresponds to inserting at next available slot
- Operation removeMin corresponds to removing item in first occupied slot



Main Point

The most efficient implementation of a Priority Queue is via *heaps* which support O(log *n*) worst-case running time for both operations `removeMin` and `insertItem`, with minimal overhead. A heap is a binary tree in which every level is filled except possibly the bottom level, which is as full as possible from left to right. In addition, a binary heap satisfies the *heap order property:* For every node *X*, the key value of *X* is greater than or equal to that of its parent (if it has a parent). A Priority Queue gives a simple model of the principle of the Highest First: Putting attention on the highest value as the top priority results in fulfillment of all lower-priority values as well, in the proper time.

Using a Heap for Sorting

- An algorithm for using a priority queue for sorting:

  Given an array of *n* items (keys) to sort:

  - Use `insertItem` *n* times to load the Priority Queue
  - Output the results by inserting into output array the output of `removeMin`, loading from left to right

- Analysis: When a Heap is used to implement the Priority Queue
  - load the Priority Queue: O(nlog n)
  - load the output array: O(nlog n)

  Therefore *HeapSort* runs in O(nlog n), in worst-case

HeapSort Optimization: *In-Place* HeapSort

1. Use the Max-Heap Order Property, so that data in a node is always *less than or equal to* data in the parent.

HeapSort Optimization: *In-Place* HeapSort

1. Use the Max-Heap Order Property, so that data in a node is always *less than or equal to* data in the parent.
2. In array implementation of Heap, children of node at position j are at positions $2j+1$ and $2j+2$ (so, parent of node at position n is located in position $(j-1)/2$).

HeapSort Optimization: *In-Place* HeapSort

1. Use the Max-Heap Order Property, so that data in a node is always *less than or equal to* data in the parent.
2. In array implementation of Heap, children of node at position j are at positions $2j+1$ and $2j+2$ (so, parent of node at position n is located in position $(j-1)/2$).
3. Phase I: Given an array A, we grow an array-based heap from left to right; before stage i, the heap occupies positions $0..i-1$ and the remaining "unheapified" elements are in remaining slots; at stage i, element at position i is included in heap, and array-based "upheap" is performed.

HeapSort Optimization: *In-Place* HeapSort

1. Use the Max-Heap Order Property, so that data in a node is always *less than or equal to* data in the parent.
2. In array implementation of Heap, children of node at position j are at positions $2j+1$ and $2j+2$ (so, parent of node at position n is located in position $(j-1)/2$).
3. Phase I: Given an array A, we grow an array-based heap from left to right; before stage i, the heap occupies positions $0..i-1$ and the remaining "unheapified" elements are in remaining slots; at stage i, element at position i is included in heap, and array-based "upheap" is performed.
4. Phase 2: Repeatedly perform removeMax on the left part of the array that consists of the (gradually diminishing) heap. Begin by pulling off max of heap and moving value at pos $n-1$ into position 0, and performing "downheap"; place max into pos $n-1$. Then pull off next max of smaller heap, place value at pos $n-2$ into position 0, perform downheap, and place this next max in pos $n-2$. Continue.

Comparison of HeapSort to Other Sorting Algorithms

- Using optimizations, HeapSort is extremely fast – about the same running times as optimized MergeSort implementations

Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searchers is O(log n).

2. In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of $\Omega(n)$. By incorporating balance conditions, the worst case can be improved to O(log n).

3. *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.

4. *Impulses Within The Transcendental Field.* The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.

5. *Wholeness Moving Within Itself.* In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.