Trees

CS 202 – Fundamental Structures of Computer Science II

Bilkent University

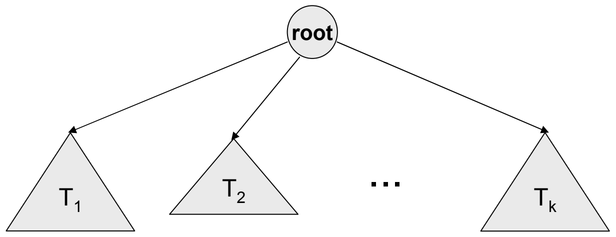
Computer Engineering Department

Outline

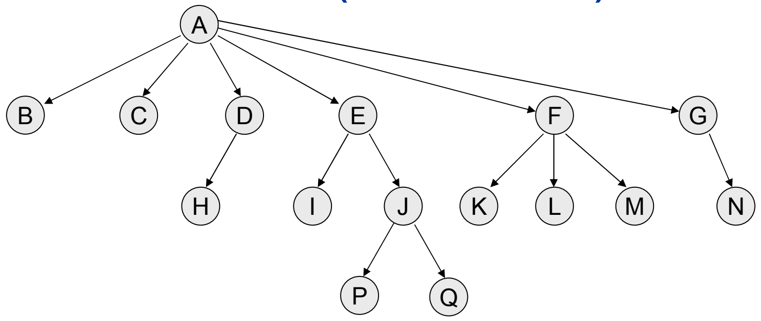
- Preliminaries
  - What is Tree?
  - Implementation of Trees using C++
  - Tree traversals and applications
- Binary Trees
- Binary Search Trees
  - Structure and operations
  - Analysis
- AVL Trees
- Splay Trees
- B-trees

What is a Tree

- A tree is a collection of nodes with the following properties:
  - The collection can be empty.
  - If collection is not empty, it consists of a distinguished node r, called *root*, and zero or more nonempty sub-trees T_1, T_2, … , T_k, each of whose roots are connected by a *directed edge* from r.
- The root of each sub-tree is said to be *child* of r, and r is the *parent* of each sub-tree root.
- If a tree is a collection of N nodes, then it has N-1 edges.

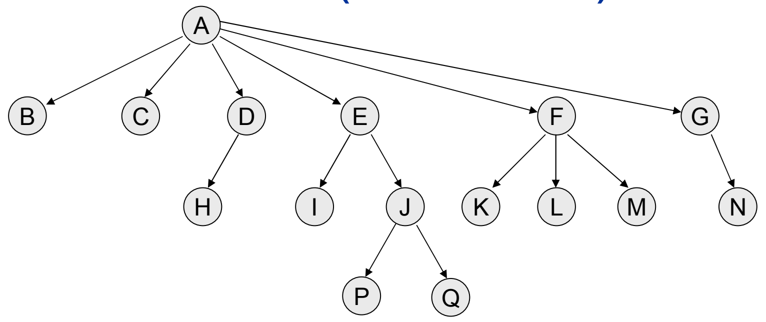


What is a tree (continued)



- A node may have arbitrary number of children (including zero)
  - Node A above has 6 children: B, C, D, E, F, G.
- Nodes with no children are called leaves.
  - B, C, H, I, P, Q, K, L, M, N are leaves in the tree above.
- Nodes with the same parent are called siblings.
  - K, L, M are siblings since F is parent of all of them.

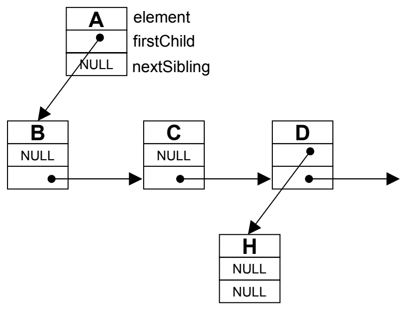
What is a tree (continued)

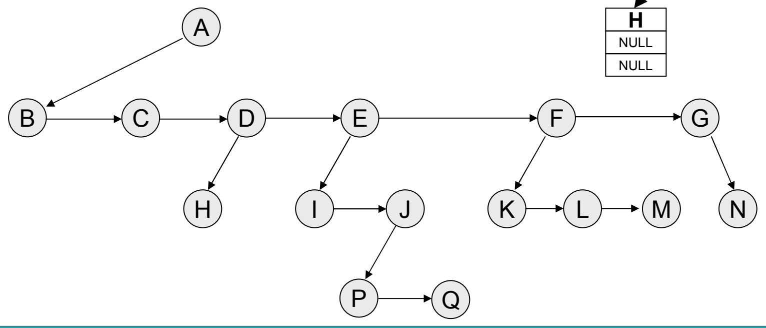


- A ***path*** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is parent of n_{i+1} $(1 \leq i < k)$
  - The ***length*** of a path is the number of edges on that path.
  - There is a path of length zero from every node to itself.
  - There is exactly one path from the root to each node.
- The ***depth*** of node n_i is the length of the path from root to node n_i
- The ***height*** of node n_i is the length of longest path from node n_i to a leaf.
- If there is a path from n_1 to n_2, then n_1 is ***ancestor*** of n_2, and n_2 is ***descendent*** of n_1.
  - If $n_1 \neq n_2$ then n_1 is *proper ancestor* of n_2, and n_2 is *proper descendent* of n_1.

Implementation of Trees

```
struct TreeNode {
        Object       element;
        TreeNode *firstChild;
        TreeNode *nextSibling;
};
```



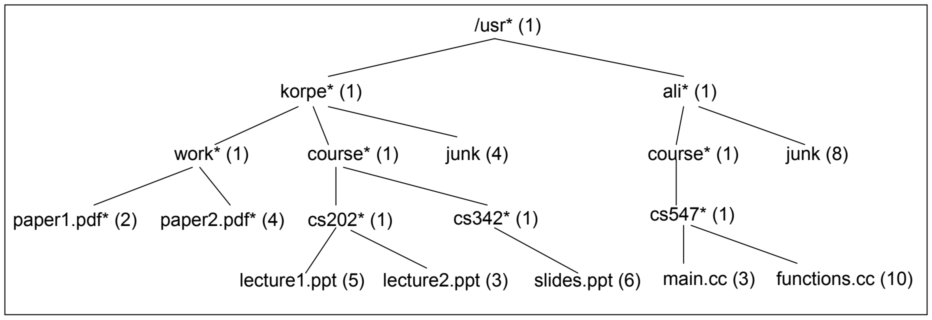


Tree Applications

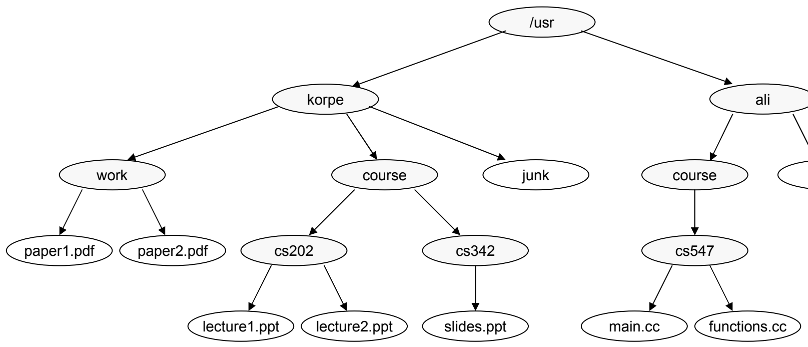
- Tree Applications
  - There many applications of trees in Computer Science and Engineering.
    - Organization of filenames in a Unix File System
    - Indexing large files in Database Management Systems
    - Compiler Design.
    - Routing Table representation for fast lookups of IP addresses
    - Search Trees for fast access to stored items

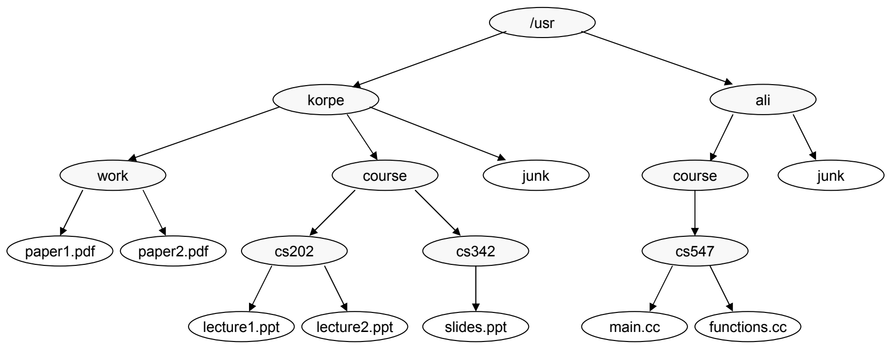
An example Application: Unix Directory Structure

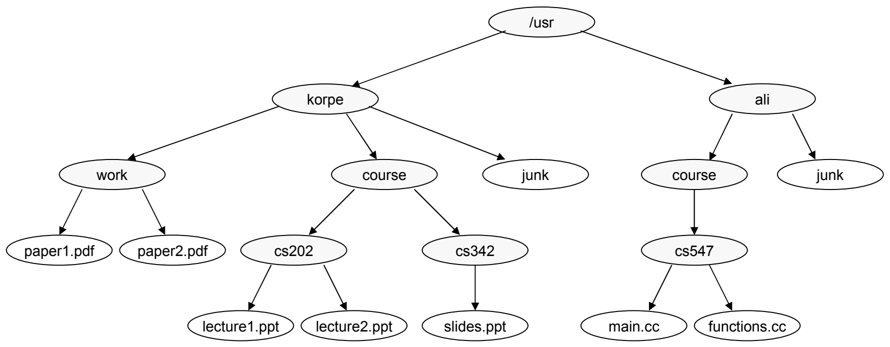
- Unix directory structure is organized as a tree.



- An asterisk next to a filename indicates that it is a directory that contains other files.
- A number next to a filename indicates how many disk blocks that file or directory occupies.


/usr
korpe
ali
work
course
junk
course
junk
paper1.pdf
paper2.pdf
cs202
cs342
cs547
lecture1.ppt
lecture2.ppt
slides.ppt
main.cc
functions.cc


/usr
korpe
ali
work
course
junk
course
junk
paper1.pdf
paper2.pdf
cs202
cs342
cs547
lecture1.ppt
lecture2.ppt
slides.ppt
main.cc
functions.cc



- We want to list files in the directory
- We want to computer the size of all files (in a recursive manner) in the directory.

Listing Files

```
Void FileSystem::listAll ( int depth = 0 ) const
{
        printName ( depth );   /* print name of object */
        if (isDirectory())
                for each file c in  this directory (for each child)
                        c.listAll( depth+1 );
}
```

Work
Is done
here!

Pseudocode to list a directory in a Unix file system

printName() function prints the name of the object after “depth” number of tabs -indentation. In this way, the output is nicely formatted on the screen.

Here, the a directory (which is a tree structured) is traversed: Every node Is visited and a work is done about each node.

The order of visiting the nodes in a tree is important while traversing a tree.

Here, the nodes are visited according to *preorder* traversal strategy.

Traversal strategies

- Preorder traversal
  - Work at a node is performed before its children are processed.
- Postorder traversal
  - Work at a node is performed after its children are processed.
- Inorder traversal (for *binary* trees)
  - For each node:
    - First left child is processed, then the work at the node is performed, and then the right child is processed.

Listing Files - Output

```
/usr
    korpe
        work
            paper1.pdf
            paper2.pdf
        course
            cs202
                lecture1.ppt
                lecture2.ppt
            cs342
                slides.ppt
        junk
    ali
        course
            cs547
                main.cc
                functions.cc
        junk
```

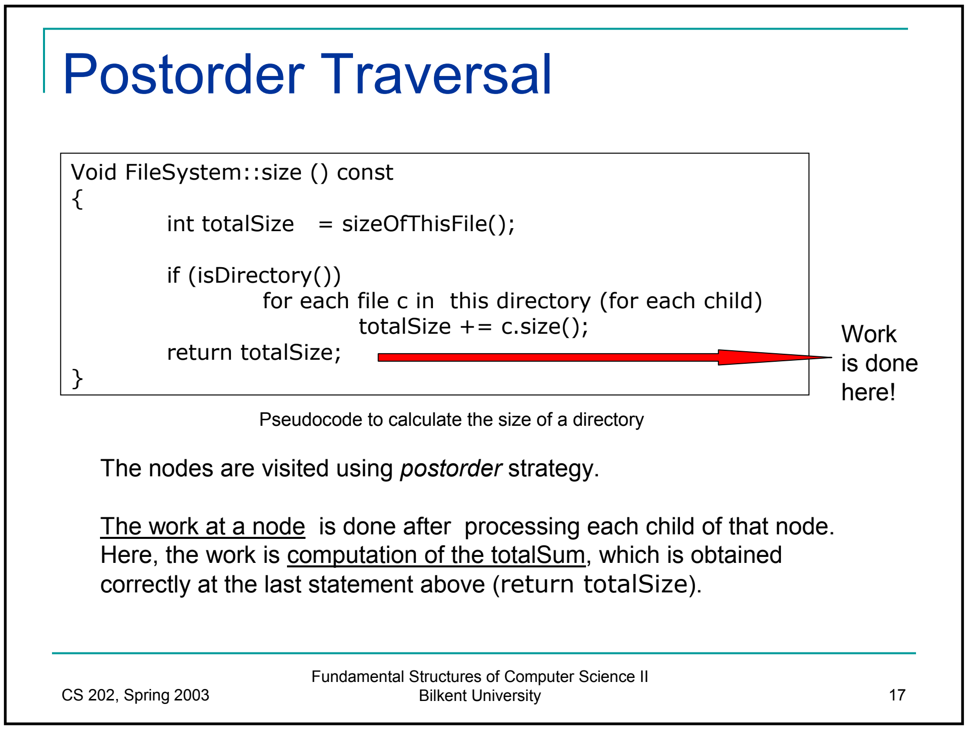
The listing of the files are done using *pre-order traversal*.

A node is processed first:
The filename is printed.

Then, the children of the node is processed starting from the left-most child.

Traversing a Tree

- For some applications, it is more suitable to traverse a tree using post-order strategy.
- As an example we want to computer the size of the directory which is defined the sum of all the sizes of the files and directories inside our directory.
- In this case, we want first computer the size of all children, add them up together with the size of the current directory and return the result.



Postorder Traversal

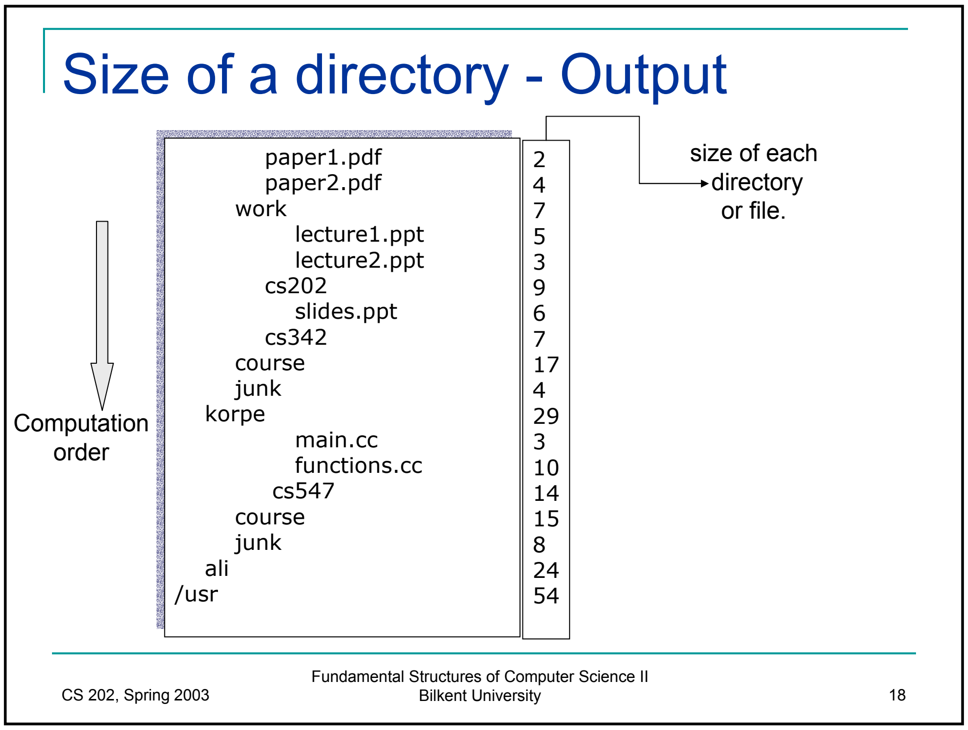
```
Void FileSystem::size () const
{
        int totalSize   = sizeOfThisFile();

        if (isDirectory())
                for each file c in  this directory (for each child)
                        totalSize += c.size();
        return totalSize;
}
```

Work is done here!

Pseudocode to calculate the size of a directory

The nodes are visited using *postorder* strategy.

The work at a node is done after processing each child of that node. Here, the work is computation of the totalSum, which is obtained correctly at the last statement above (return totalSize).



Size of a directory - Output

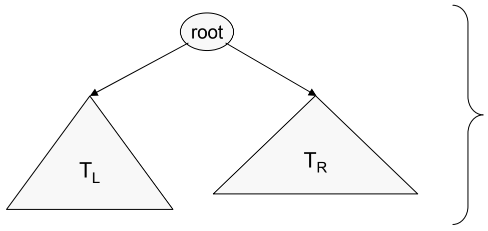
```
            paper1.pdf       2
            paper2.pdf       4
        work                 7
                lecture1.ppt 5
                lecture2.ppt 3
            cs202            9
                slides.ppt   6
            cs342            7
        course               17
        junk                 4
    korpe                    29
                main.cc      3
                functions.cc 10
             cs547           14
        course               15
        junk                 8
    ali                      24
/usr                         54
```

Computation order

size of each directory or file.

Binary Trees

- A *binary tree* is a tree in which no node can have more than two children
- Average depth of a binary tree is $O(\sqrt{N})$
- For a special binary tree, called *binary search tree*, the average depth is $O(\log N)$
- The depth can be as large as *N-1* in the worst case.



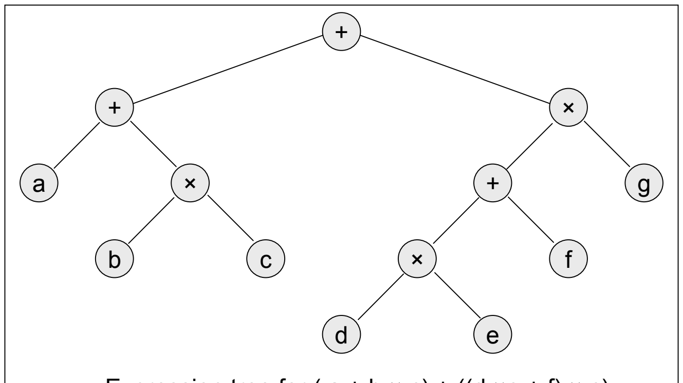
A binary tree consisting of a root and two subtrees T_L and T_R, both of which could possibly be empty.

Binary Trees - Implementation

```
struct BinaryNode {
         Object          element;       // the data in the node
         BinaryNode *firstChild;       // left child
         BinaryNode *nextSibling;   // right child
};
```

- Binary trees have many important uses.
- One of the applications is in compiler design.
- Mathematical expressions may be represented as binary trees in compiler design.
- A expression consist of *operands* and *operators* that operate on these operands.
  - Most operators operate on two operands (+, -, x, …)
  - Some operators may operate on only one operand (*unary minus*)
  - A operand could be a constant or a variable name.

Expression Trees



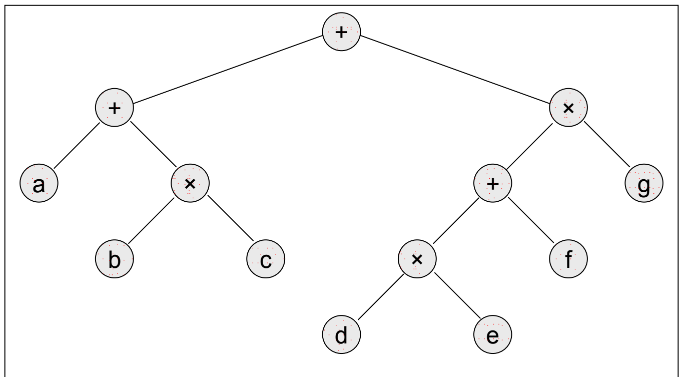
Expression tree for (a + b × c) + ((d ×e + f) × c)

There are three notations for a mathematical expression:
1) *Infix* notation : (a + (b × c)) + (((d ×e) + f) × c)
2) *Postfix* notation: a b c × + d e × f + g * +
3) *Prefix* notation : + + a × b c × + × d e f g

Expression Tree traversals

- Depending on how we traverse the expression tree, we can produce one of these notations for the expression represented by the three.
  - *Inorder* traversal produces *infix* notation.
    - This is a overly parenthesized notation.
    - Print out the operator, then print put the left subtree inside parentheses, and then print out the right subtree inside parentheses.
  - *Postorder* traversal produces *postfix* notation.
    - Print out the left subtree, then print out the right subtree, and then printout the operator.
  - *Preorder* traversal produces *prefix* notation.
    - Print out the operator, then print out the right subtree, and then print out the left subtree.

Postorder traversal



Expression tree for (a + b × c) + ((d ×e + f) × c)

Postfix notation: a b c × + d e × f + g × +

Construction an expression tree

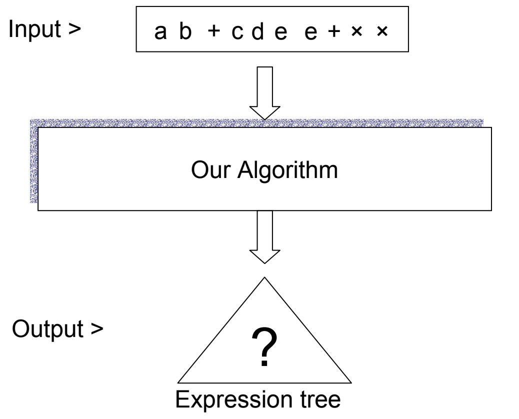
- Given an expression tree, we can obtain the corresponding expression in postfix notation by traversing the expression tree in postorder fashion.
- Now, given an expression in postfix notation, we will see an algorithm to obtain the corresponding expression tree.

Sketch of the algorithm

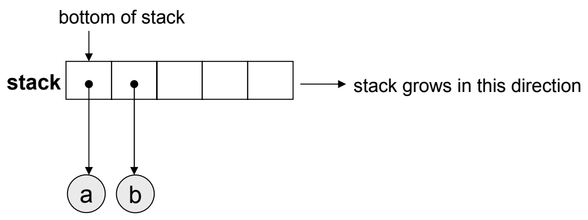
- Read the expression (given in postfix notation) one *symbol* at a time.
- If the symbol is an operand:
  - We create a one-node tree (that keeps the operand) and push a pointer to this tree on top of a stack.
- If the symbol is an operator:
  - We fist pop up two pointers from the stack. The pointers point to trees T_1 and T_2.
  - Then we generate a new tree whose root is the operator (the symbol just read) and the root's left and right children point to trees T_1 and T_2 respectively.
  - A point to the root of this new tree is pushed onto the stack.

Example

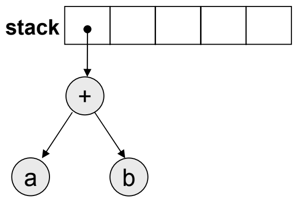
We are given an expression in postfix notation:



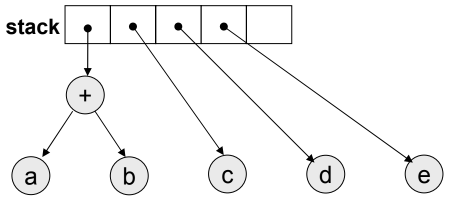
After reading and processing symbols **a** and **b**:



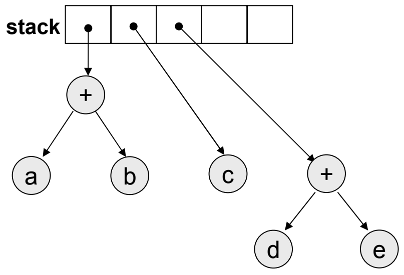
After reading and processing symbol **+**:



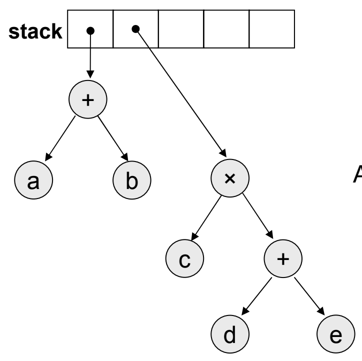
After reading and processing symbols **c**, **d**, and **e**:



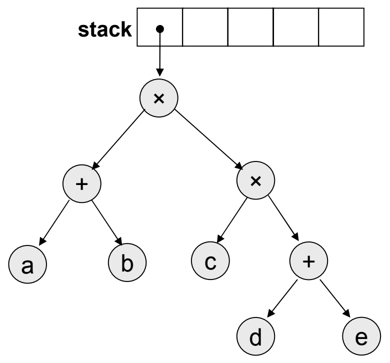
After reading and processing symbol **+**:



After reading and processing symbol ×:



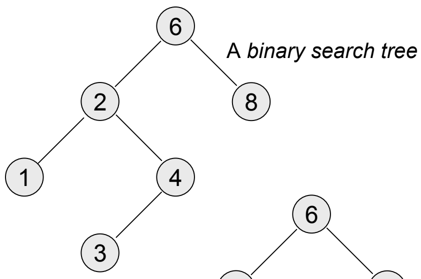
After reading and processing last symbol ×:



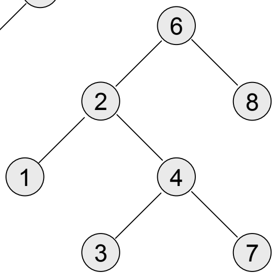
Binary Search Trees

- Assume each node of a binary tree stores a data item
- Assume data items are of some type that be ordered and all items are distinct. No two items have the same value.

- A binary search tree is a binary tree such that
  - for every node X in the tree:
    - the values of all the items in its left subtree are smaller than the value of the item in X
    - the values of all items in its right subtree are greater than the value of the item in X.



A *binary search tree*



Not a *binary search tree*, but a *binary tree*

Definition

```
template <class Comparable>
class BinarySearchTree;

template <class Comparable>
class BinaryNode
{
   Comparable element;   // this is the item stored in the node
   BinaryNode *left;
   BinaryNode *right;

   BinaryNode( const Comparable & theElement, BinaryNode *lt,
                                                    BinaryNode *rt )
     : element( theElement ), left( lt ), right( rt ) { }
   friend class BinarySearchTree<Comparable>;
};
```

A class template is used so that we don't need to define a separate class for each element type.

The type of element here is generic "Comparable".

BinaryNode class

BinarySearchTree class is defined as *friend* so that it can access the private members of BinaryNode class.

Operations on BSTs

- Find
  - Given a value find the item in the tree that has the same value.
  - If the item is not found return a special value.
- Find Minimum
  - Find the item that has the minimum value in the tree
- Find Maximum
  - Find the item that has the maximum value in the tree
- Insert
  - Insert a new item in the tree.
    - Check for duplicates.
- Delete
  - Delete an item from the tree.
    - Check if the item exists in the tree.
- Copy
  - Obtain a new binary search tree from a given binary search tree. Both should have the same structure and values.
- Print
  - Print the values of all items in the tree using a traversal strategy that is appropriate for the application

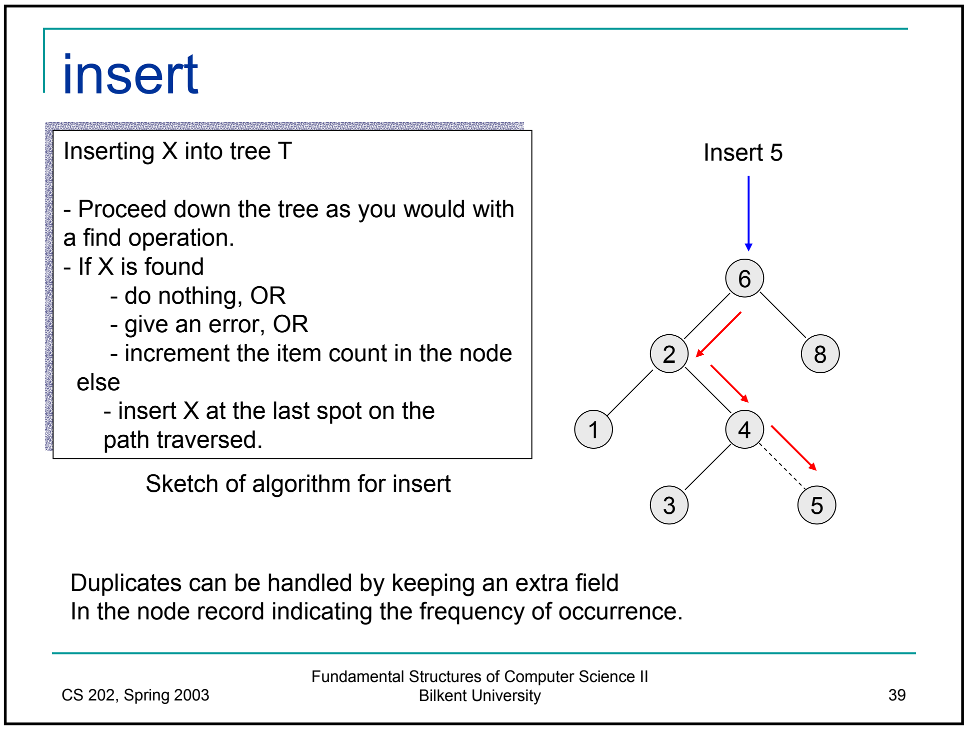
- Most of the operation on binary trees are O(log*N*).
  - This is the main motivation for using binary trees rather than using ordinary lists to store items.
- Most of the operations can be implemented using recursion.
  - Since the average depth of binary search trees is O(log*N*), we usually do not need to worry about running out of stack space while using recursion.

```
// BinarySearchTree class

template <class Comparable>
class BinarySearchTree
{
 public:
     explicit BinarySearchTree( const Comparable & notFound );
     BinarySearchTree( const BinarySearchTree & rhs );
     ~BinarySearchTree( );

     const Comparable & findMin( ) const;
     const Comparable & findMax( ) const;
     const Comparable & find( const Comparable & x ) const;
     bool isEmpty( ) const;
     void printTree( ) const;

     void makeEmpty( );
     void insert( const Comparable & x );
     void remove( const Comparable & x );

     const BinarySearchTree & operator=( const BinarySearchTree & rhs );

     //continued on the next page
private:
    BinaryNode<Comparable> *root;
    const Comparable ITEM_NOT_FOUND;

    const Comparable & elementAt( BinaryNode<Comparable> *t ) const;

    void insert( const Comparable & x, BinaryNode<Comparable> * & t ) const;
    void remove( const Comparable & x, BinaryNode<Comparable> * & t )
const;
    BinaryNode<Comparable> * findMin( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * findMax( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * find( const Comparable & x,
                                             BinaryNode<Comparable> *t ) const;
    void makeEmpty( BinaryNode<Comparable> * & t ) const;
    void printTree( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * clone( BinaryNode<Comparable> *t ) const;
};
```

- There are public members and private members in the class definition.
  - They have the same but different signatures.
- Private member functions are recursive.
- Public member functions make use of the private member functions.
- For example public find() calls private recursive find() function.

```
 /**
  * Find item x in the tree.
  * Return the matching item or ITEM_NOT_FOUND if not found.
  */
template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::
                    find( const Comparable & x ) const
{
   return elementAt( find( x, root ) );
}

template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::
elementAt( BinaryNode<Comparable> *t ) const
{
   if( t == NULL )
      return ITEM_NOT_FOUND;
   else
      return t->element;
}
```

```
 /**
  * Internal method to find an item in a subtree.
  * x is item to search for.
  * t is the node that roots the tree.
  * Return node containing the matched item.
  */
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
   if( t == NULL )
      return NULL;
   else if( x < t->element )
      return find( x, t->left );
   else if( t->element < x )
      return find( x, t->right );
   else
      return t;    // Match
}
```



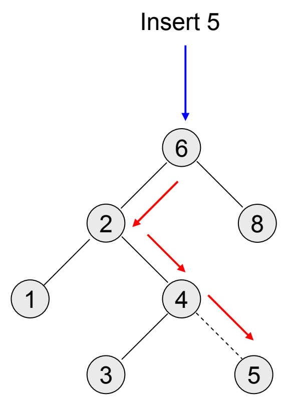
insert

Inserting X into tree T

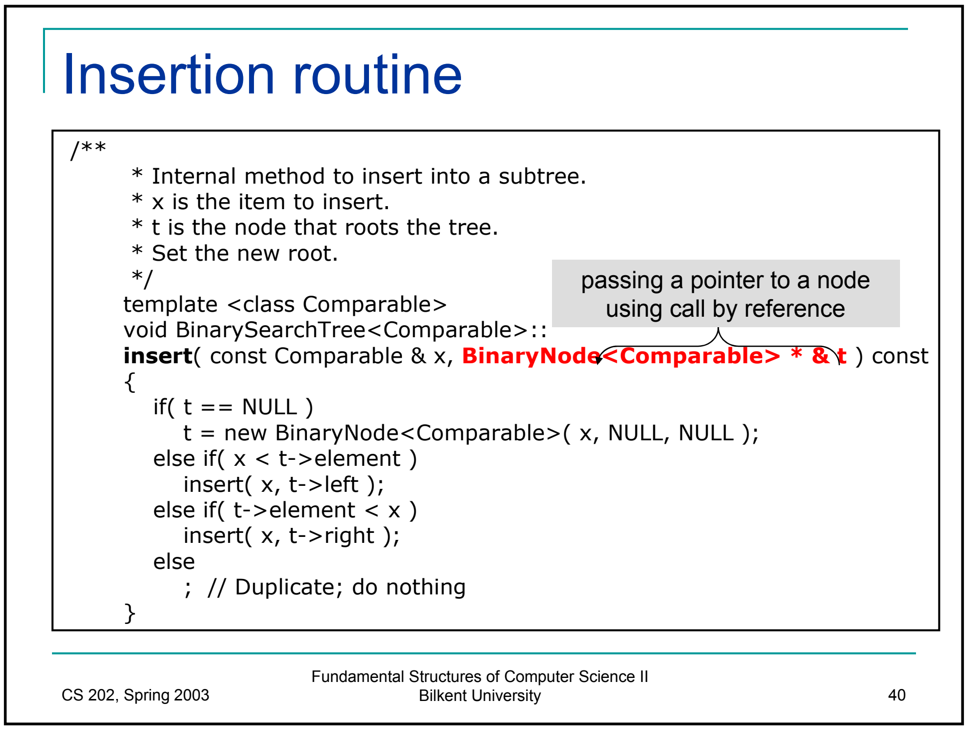
- Proceed down the tree as you would with a find operation.
- If X is found
    - do nothing, OR
    - give an error, OR
    - increment the item count in the node

else
    - insert X at the last spot on the path traversed.

Sketch of algorithm for insert



Duplicates can be handled by keeping an extra field
In the node record indicating the frequency of occurrence.

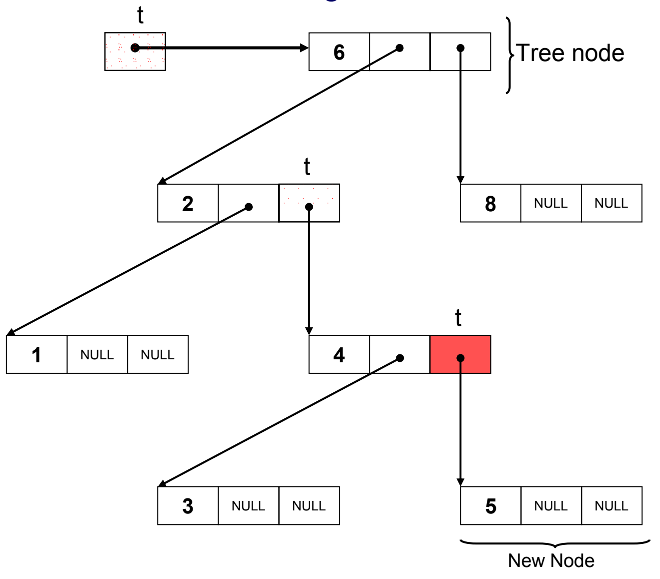


Insertion routine

```
/**
  * Internal method to insert into a subtree.
  * x is the item to insert.
  * t is the node that roots the tree.
  * Set the new root.
  */
 template <class Comparable>
 void BinarySearchTree<Comparable>::
 insert( const Comparable & x, BinaryNode<Comparable> * & t ) const
 {
    if( t == NULL )
       t = new BinaryNode<Comparable>( x, NULL, NULL );
    else if( x < t->element )
       insert( x, t->left );
    else if( t->element < x )
       insert( x, t->right );
    else
       ;  // Duplicate; do nothing
 }
```

passing a pointer to a node using call by reference

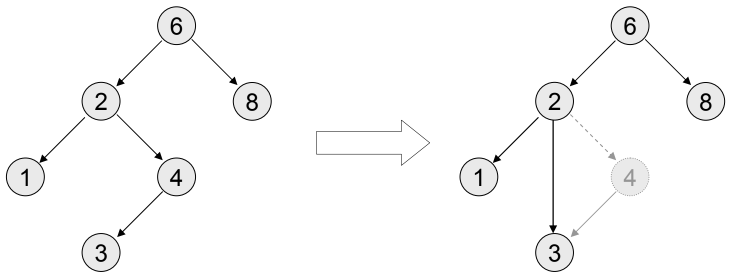
Inserting Item 5 to the Tree



Remove

- Deleting an item is more difficult
- There are several cases to consider:
  - If the node (that contains the item) is leaf:
    - then we can delete it immediately.
  - If the node has one child:
    - then the node can be deleted after its parent adjust a link to bypass the node.
  - If the node has two children:
    - then the general strategy is:
      - Replace the data of this node with the smallest data on the right subtree of this node.
      - Recursively delete that node on the right subtree.

Deleting a node with one child

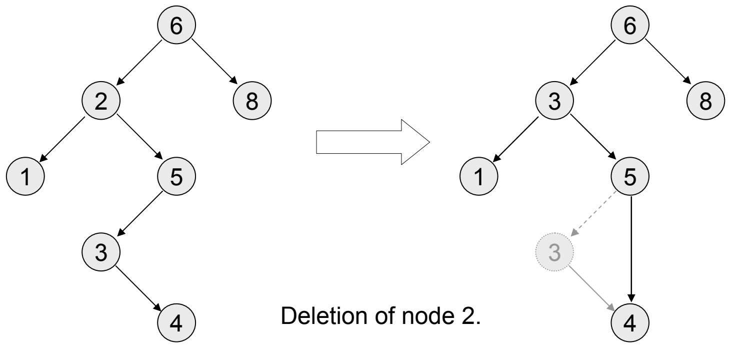


Deletion of node 4.

before

after

Deleting a node with two children



Deletion of node 2.

before

after

```
template <class Comparable>
void BinarySearchTree<Comparable>::
remove( const Comparable & x,
            BinaryNode<Comparable> * & t ) const
{
   if( t == NULL )
      return;   // Item not found; do nothing
   if( x < t->element )
      remove( x, t->left );
   else if( t->element < x )
      remove( x, t->right );
   else if( t->left != NULL && t->right != NULL ) // Two children
   {
      t->element = findMin( t->right )->element;
      remove( t->element, t->right );
   }
   else
   {
      BinaryNode<Comparable> *oldNode = t;
      t = ( t->left != NULL ) ? t->left : t->right;
      delete oldNode;
   }
}
```

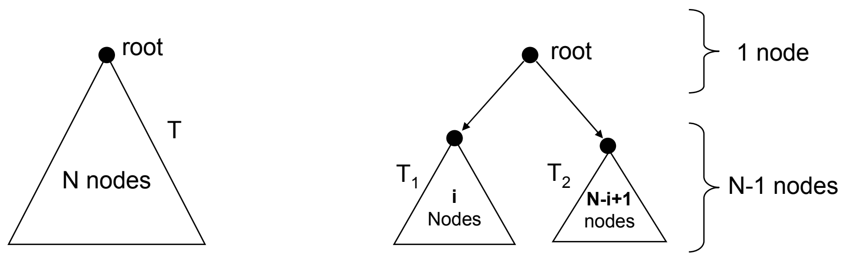
Average case analysis

- The running time of all operations (find, insert, remove, findMin, findMax) are O(d), where d is the depth of the node containing the accessed item
- The average depth over all nodes in a binary search tree is O(log*N*), where *N* is number of nodes in the tree.
  - Assuming that insertion sequences are equally likely.

Average case analysis

- Definition: The sum of depths of all nodes in a tree is called *internal path length.*
- Computing *average internal path length* of a BST will give as average depth.
  - Assuming all insertion sequences are equally likely.
- Let D(N) denote the internal path length for some tree T of N nodes.
  - D(1) = 0

Derivation of average depth



D(N)

D(N) = D(i) + D(N-i+1) + N - 1

$T \equiv T_1$ – root – T_2
The depth of a node in T_1 or T_2 will have one less then the corresponding node in T.
Therefore, we have the N - 1 term in the above equation for D(N).

Derivation of average depth

Assuming all subtree sizes are equally likely, then the average value of both D(i) and D(N-i+1) is equal to:

$$avg(D(i)) = avg(D(N-i+1)) = \frac{1}{N}\sum_{j=0}^{N-1} D(j)$$

This yields:

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

The above formula is a recurrence relation. The solution of this yields:

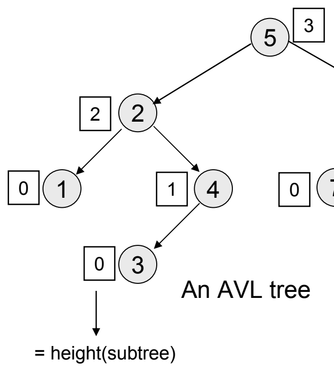
$$D(N) = O(N \log N)$$

$$average\_depth = O\left(\frac{N \log N}{N}\right) = O(\log N)$$

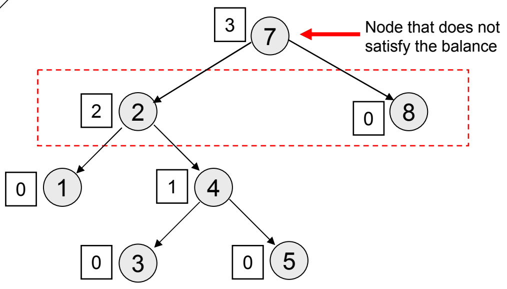
AVL Trees

- A binary search tree with a balance condition
  - Balance condition must be easy to maintain.
  - Balance condition ensures that the depth of the tree is O(logN).
- AVL Tree definition:
  - A tree that is identical to a binary search tree, except that for every node in the tree, the *height* of the left and right subtrees can differ by *at most 1*.
    - (The height of an empty tree is defined to be -1).

Example



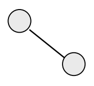
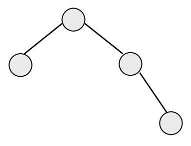
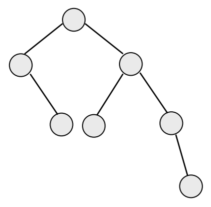
An AVL tree



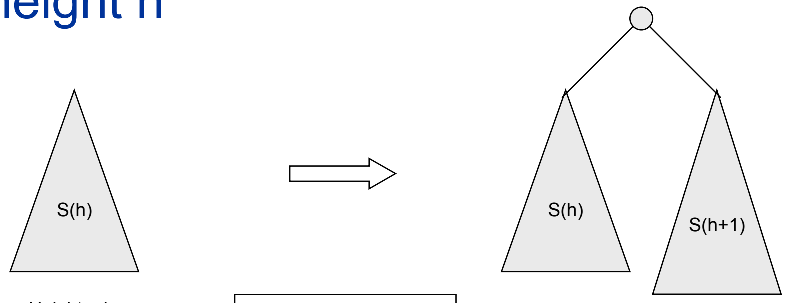
A binary search tree, but not an AVL tree

Minimum number of nodes in an AVL of height h

h: Height of Tree
S(h): Minimum number of nodes in the Tree
S(h) = S(h-1) + S(h-2) +1

	Empty Tree				
Height h	-1	0	1	2	3
Minimum Number of Nodes: S(h)	0	1	2	4	7

Minimum number of nodes in an AVL of height h



Height = h

$\boxed{S(h+2) = S(h)+S(h+1)+1}$

Height = h+2

$\downarrow$

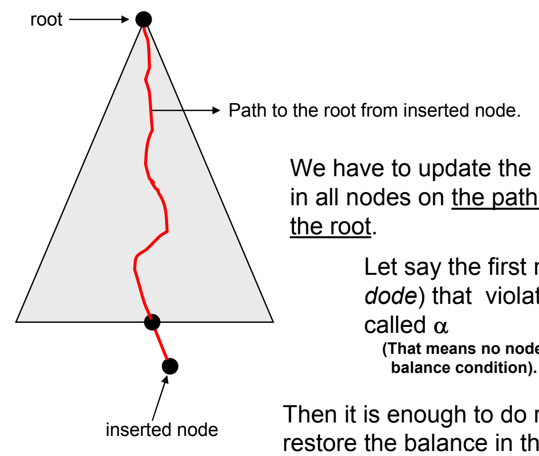
$\boxed{S(h) = S(h-1)+S(h-2)+1}$

$\downarrow$ S(h) is closely related to the Fibonacci Numbers

Height of an AVL tree is O(log*N*)

- Since, height is O(log*N*), most operations can be done in O(logN) time.
- Deletion is simple assuming lazy deletion and it is O(logN): we just have to find the node that contains the value
  - In lazy deletion, we just invalidate the value, but do not remove the node – hence the balance is not affected.
- Insertion is more difficult
  - After inserting a node into proper place in the search tree, the balance of the tree may be violated.
  - Therefore, the balancing information in all nodes on the path from inserted node to the root should be updated.
  - After this updates, we may find some nodes violating the AVL tree balance condition.
  - Therefore the balance should be restored by some operation on the tree.
  - We will show that this can be done always by operations, called *rotations*.

Insertions: sketch



We have to update the balance information in all nodes on the path from inserted node to the root.

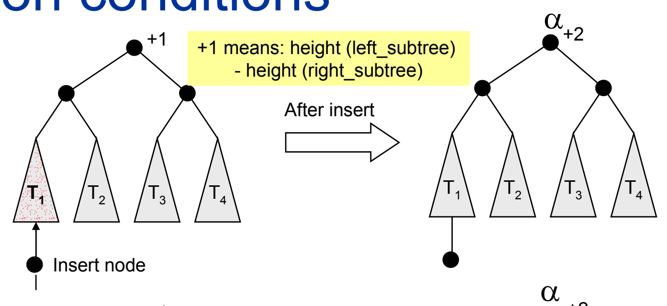
Let say the first node on this path (*deepest dode*) that violates the balance condition is called α

(That means no node below α on the path violated the balance condition).

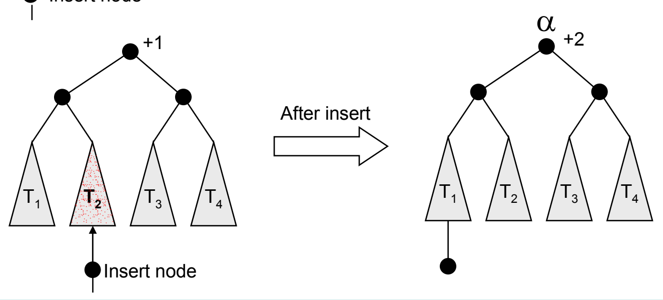
Then it is enough to do rotation around α to restore the balance in the tree. We do not Need to repeat the rotation on other nodes on the path that may have unbalanced condition.

Violation conditions

Case 1: Insert to the **left** subtree of **left** child of α



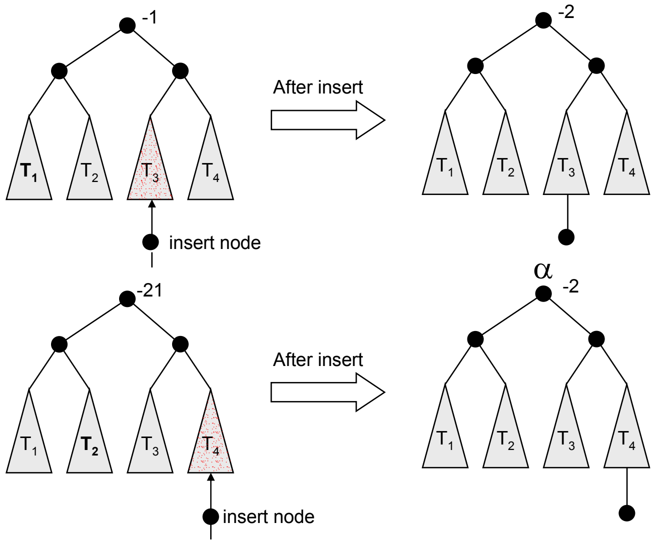
Case 2: Insert to the **right** subtree of **left** child of α



Violation conditions

Case 3:
Insert to the **left** subtree of **right** child of α

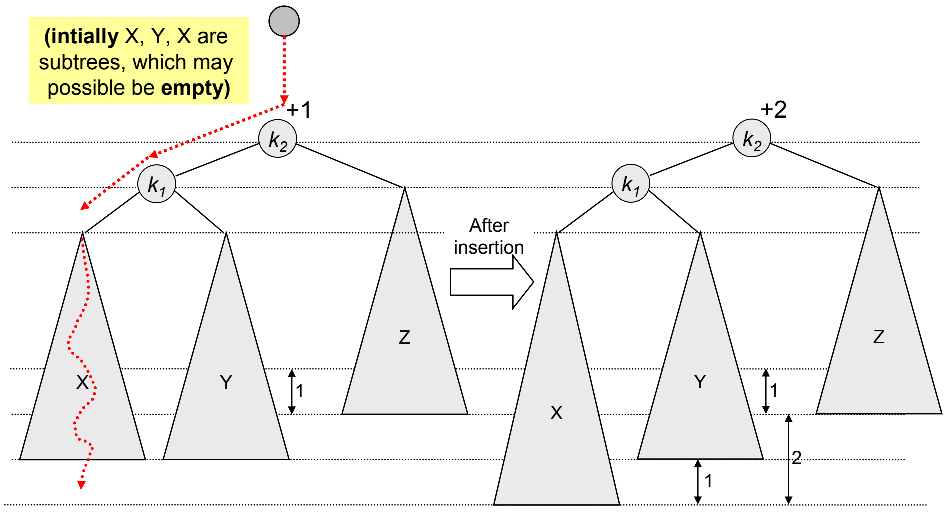
Case 4:
Insert to the **right** subtree of **right** child of α



Balancing Operations: Rotations

- Case 1 and case 4 are symmetric and requires the some operation for balance.
- Case 2 and case 3 are symmetric and requires the some operation for balance.
  - Case 1,4 is handle by an operation called *single rotation.*
  - Case 2,3 are handled by an operation called *double rotation.*

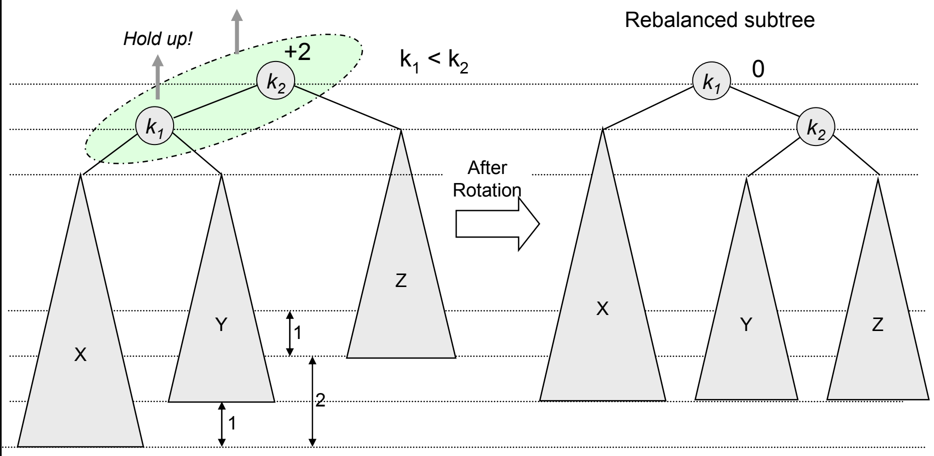
Case 1: Insertion



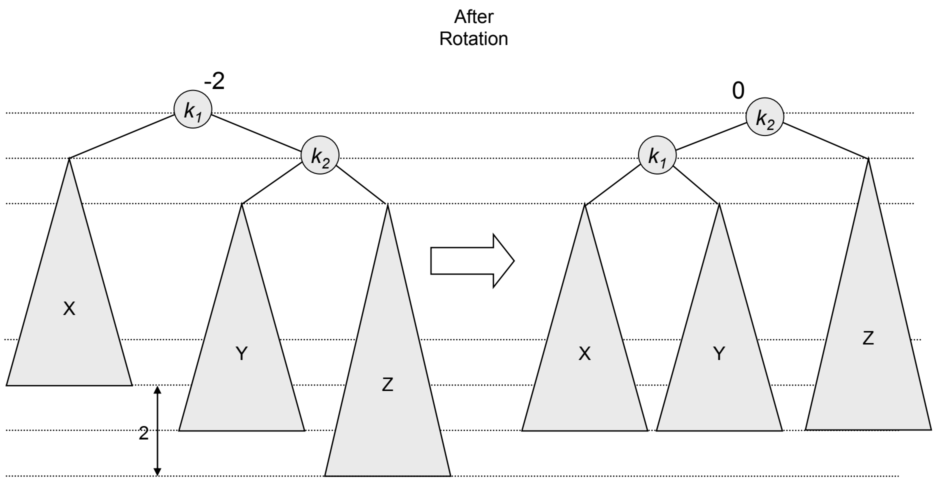
(intially X, Y, X are subtrees, which may possible be empty)
+1
k2
k1
X
Y
Z
1
After insertion
+2
k2
k1
X
Y
Z
1
1
2

Case 1: Singe (right) Rotation

Rotation between parent k2 and child k1: child goes up.

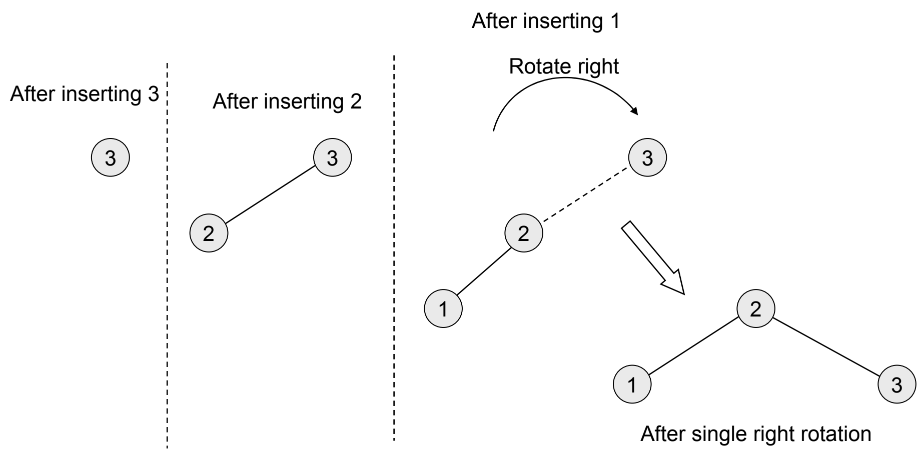


Case 4: Single (left) Rotation

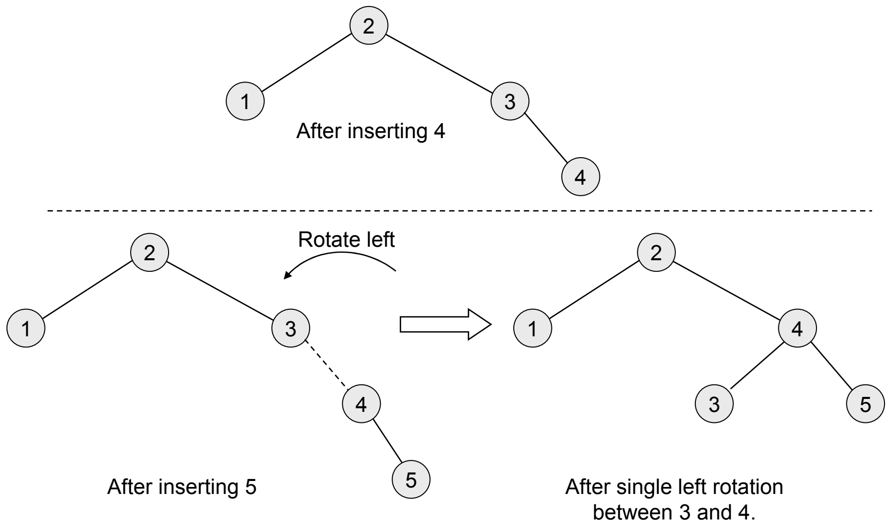


- Single rotation preserves the original height:
  - The height of the subtree where rotation is performed (root at α) is the same before insertion and after insertion+rotation
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Therefore the rotation takes O(1) time.
- Hence insertion is O(logN)

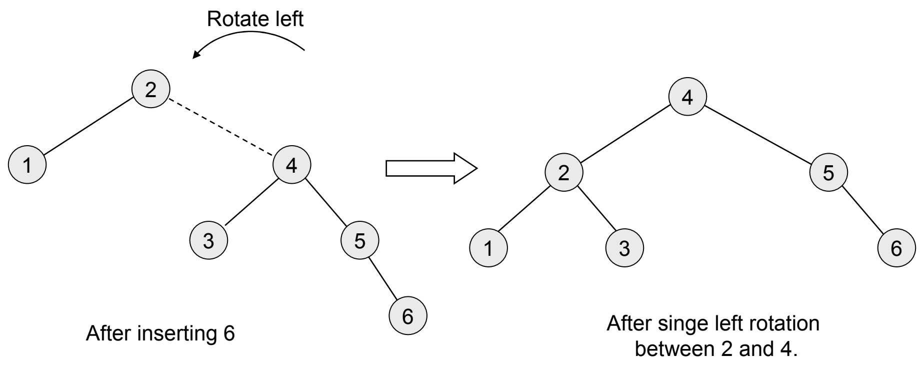
Example: Insertion of items 3,2,1,4,5,6,7 into an empty AVL tree.



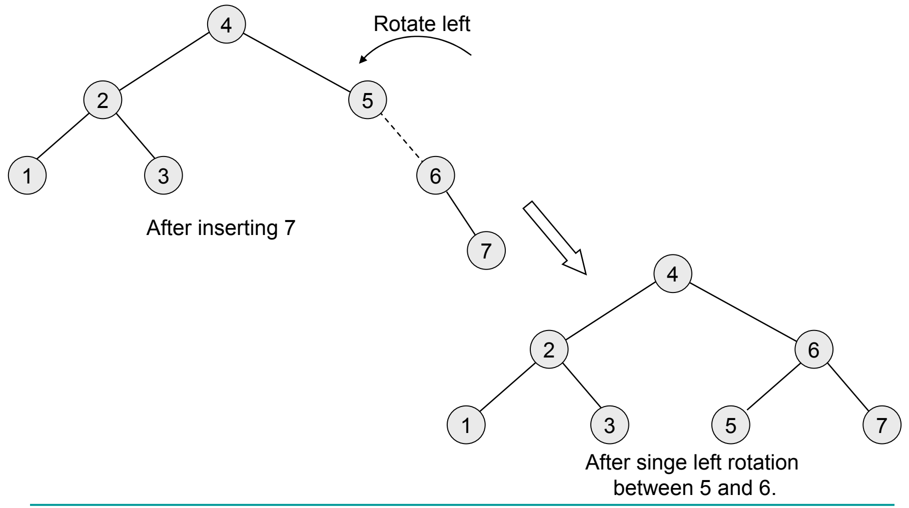
Example continued



Example continued



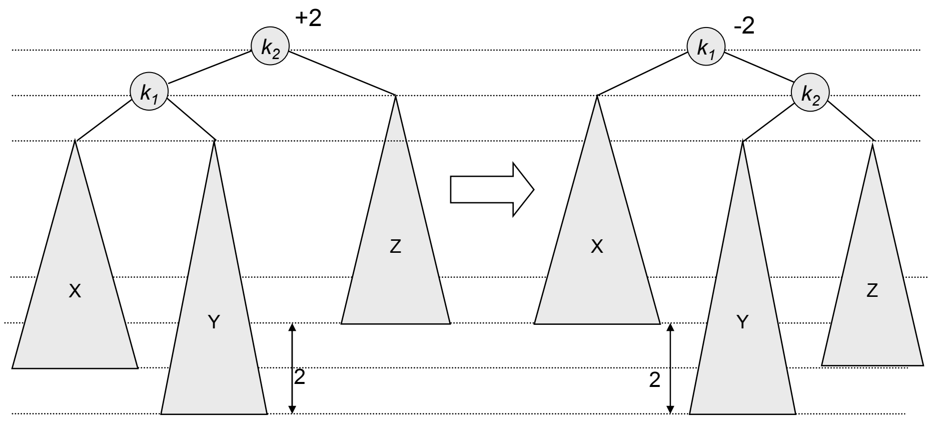
Example continued



Double Rotation

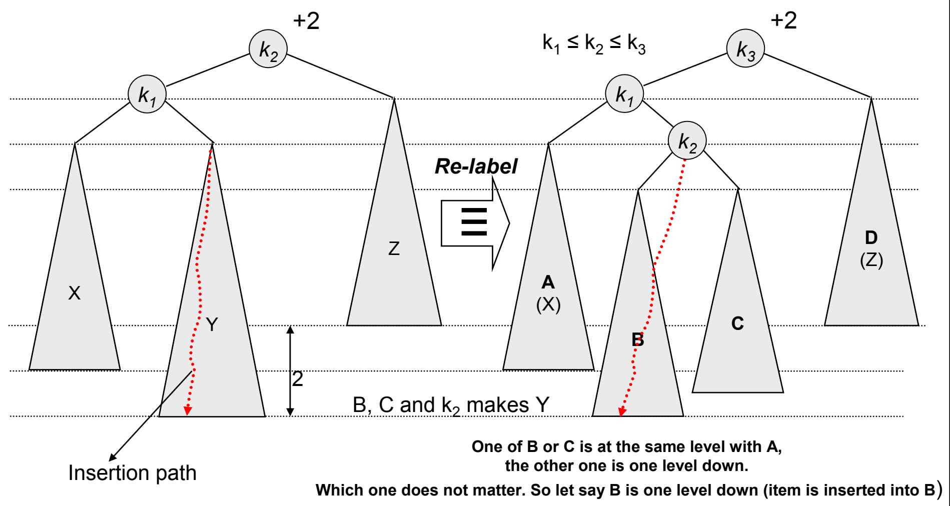
- We have solved cases 1 and 4.
  - An insertion in these cases requires single left or right rotation, depending on whether the case1 or case 4 occurs.
- Single rotation in cases 3 and 4 do not work.
  - It does not rebalance the tree.
- We need a new operation which called double rotation.

Need for Double Rotation



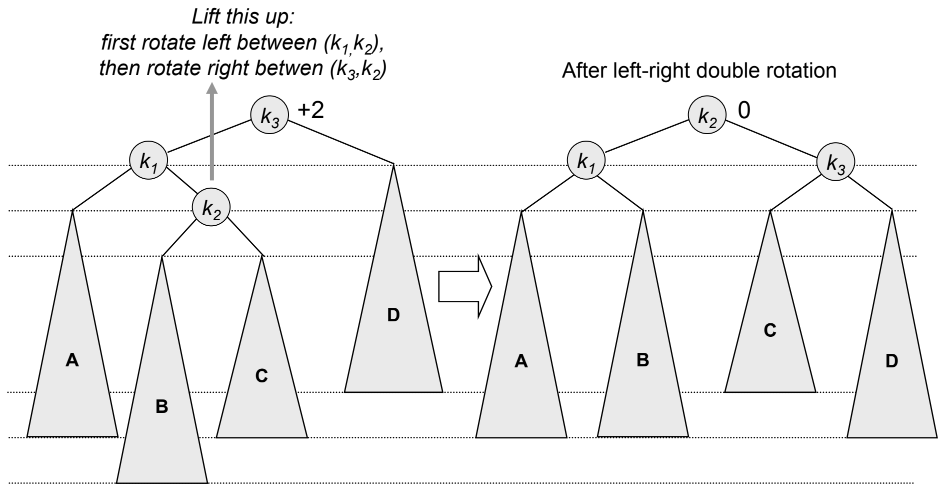
Single rotation does not provide rebalance in cases 2 and 3

Case 2



+2
k2
k1
X
Y
Z
2
Insertion path
Re-label
≡
k1 ≤ k2 ≤ k3
+2
k3
k1
k2
A
(X)
B
C
D
(Z)
B, C and k2 makes Y
One of B or C is at the same level with A,
the other one is one level down.
Which one does not matter. So let say B is one level down (item is inserted into B)

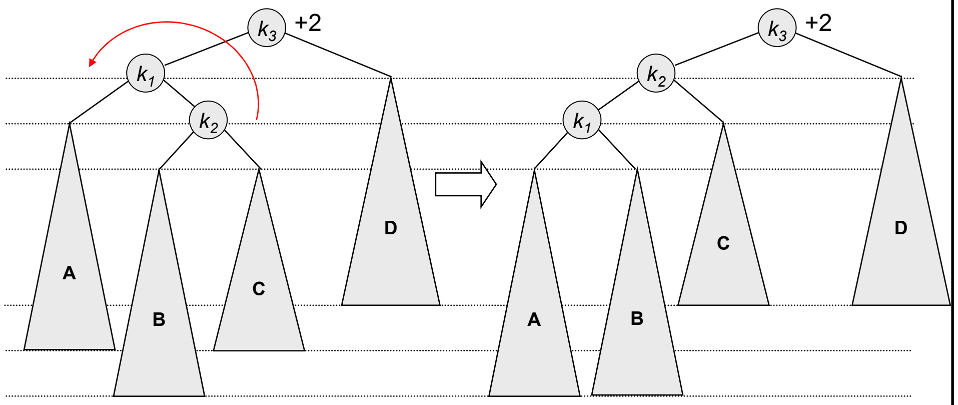
Left-Right Double Rotation



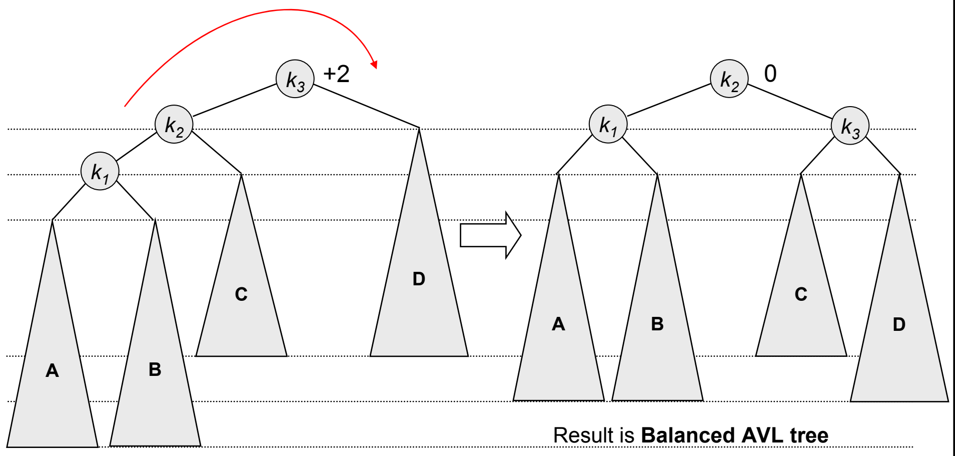
Lift this up:
first rotate left between (k1,k2),
then rotate right betwen (k3,k2)
k3
+2
k1
k2
A
B
C
D
After left-right double rotation
k2
0
k1
k3
A
B
C
D

- A left-right double rotation can be done as a sequence of two single rotations:
  - 1st rotation on the original tree:
    a *left* rotation between left-child and grandchild
  - 2nd rotation on the new tree:
    a right rotation between a node and its left child.

Rotation 1: Single Left rotation



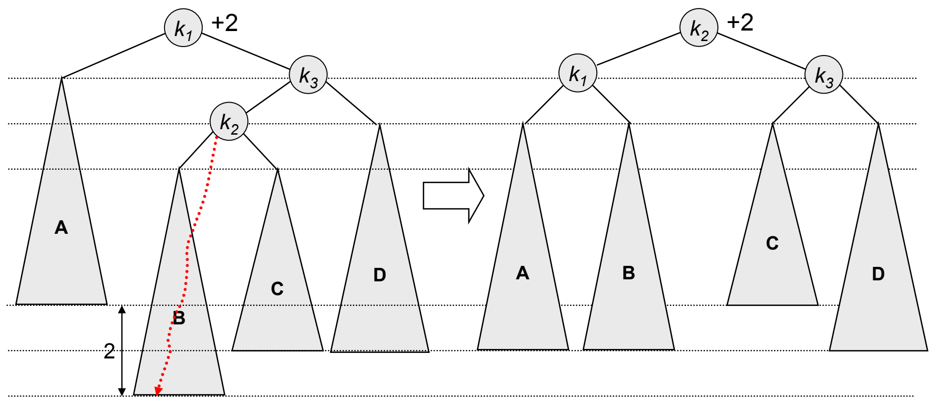
Rotation 2: Single Right rotation



Result is **Balanced AVL tree**

Resulting tree has the same height with the original tree before insertion

Case 3: Right-Left Double Rotation

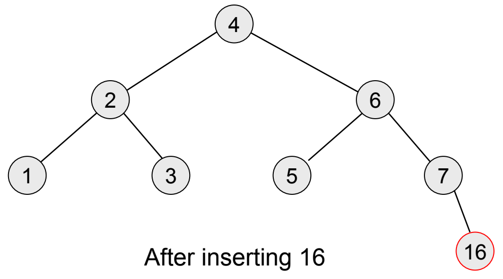


After inserting

After **right-left double** rotation

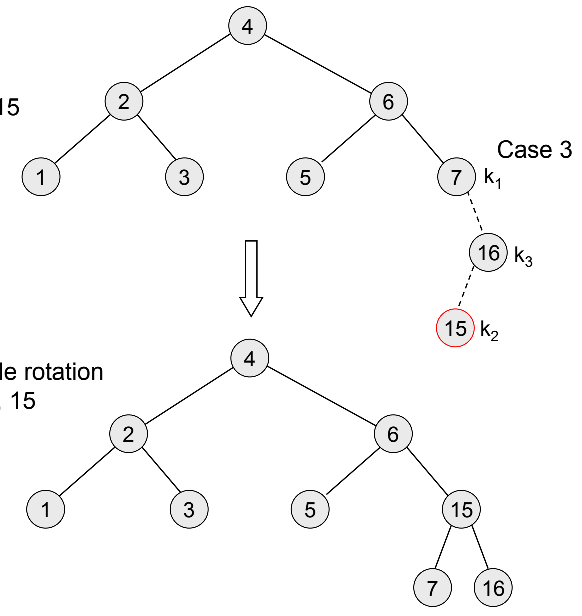
Example

- Insert 16, 15, 14, 13, 12, 10, and 8, and 9 to the previous tree obtained in the previous single rotation example.



After inserting 16

After inserting 15

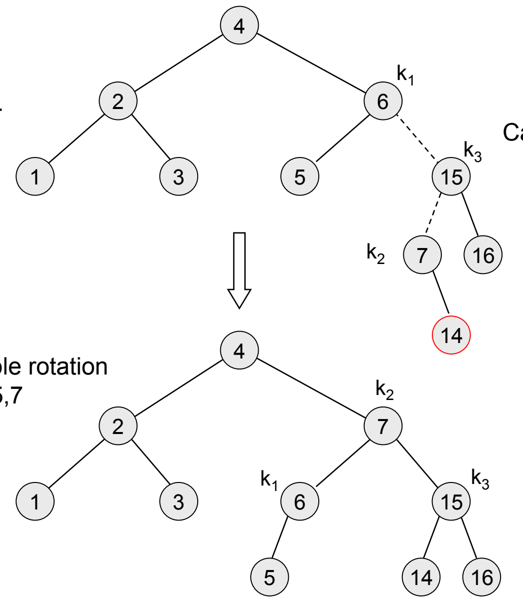


After right-left double rotation among 7, 16, 15

After inserting 14

Case 3

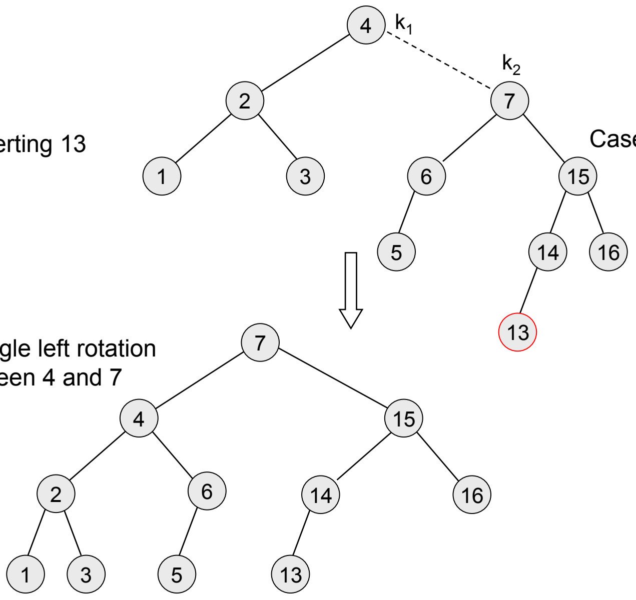
After right-left double rotation among 6,15,7



After inserting 13

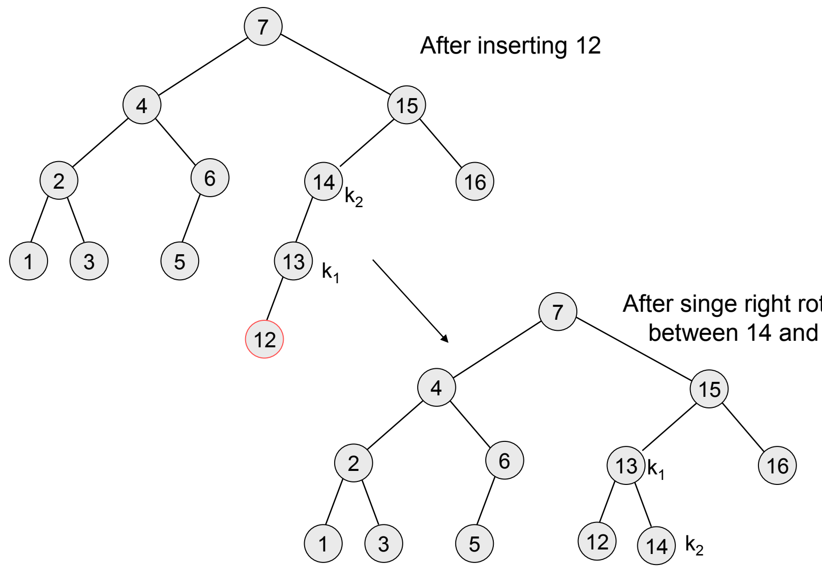
Case 4

After single left rotation between 4 and 7



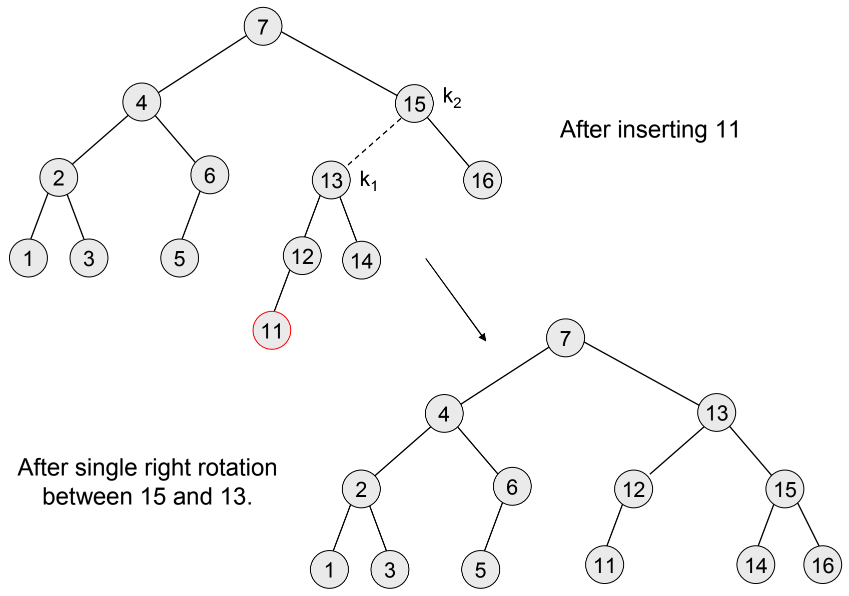
After inserting 12

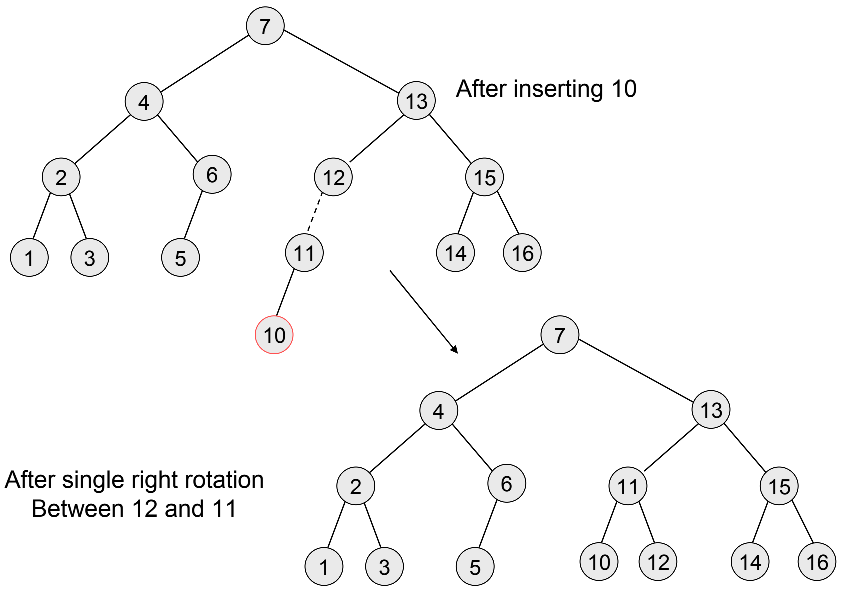
After singe right rotation between 14 and 13



After inserting 11

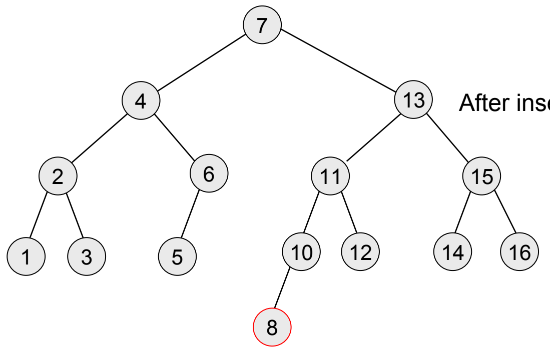
After single right rotation between 15 and 13.



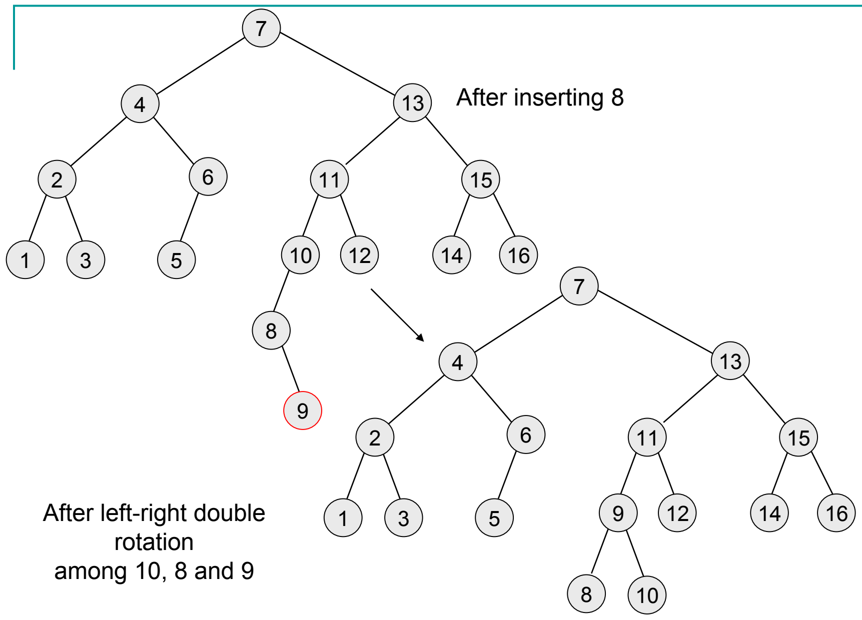


After inserting 10

After single right rotation
Between 12 and 11



After inserting 8



After inserting 8

After left-right double rotation
among 10, 8 and 9