H.O.#12
Fall 2015
Gary Chan

Binary Tree

(N:12)

Outline

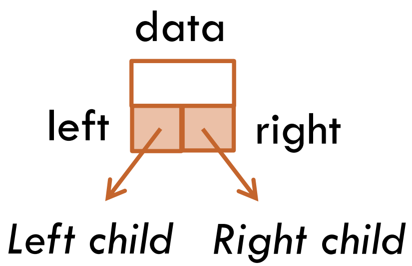
- Binary tree terminology
- Tree traversals: preorder, inorder and postorder
- Dictionary and binary search tree
- Binary search tree operations
  - Search
  - min and max
  - Successor
  - Insertion
  - Deletion
- AVL tree for tree balancing

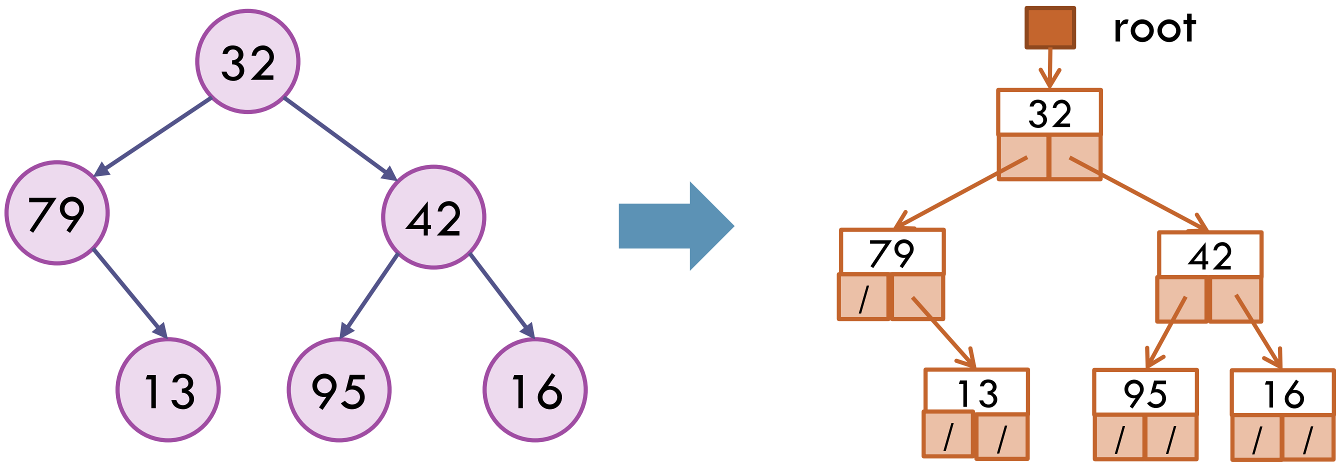
Binary Tree Terminology

- Go to the supplementary notes

Linked Representation of Binary Trees

- The degree of a node is the number of children it has. The degree of a tree is the maximum of its element degree.
  - In a binary tree, the tree degree is two
- Each node has two links
  - one to the left child of the node
  - one to the right child of the node
  - if no child node exists for a node, the link is set to NULL





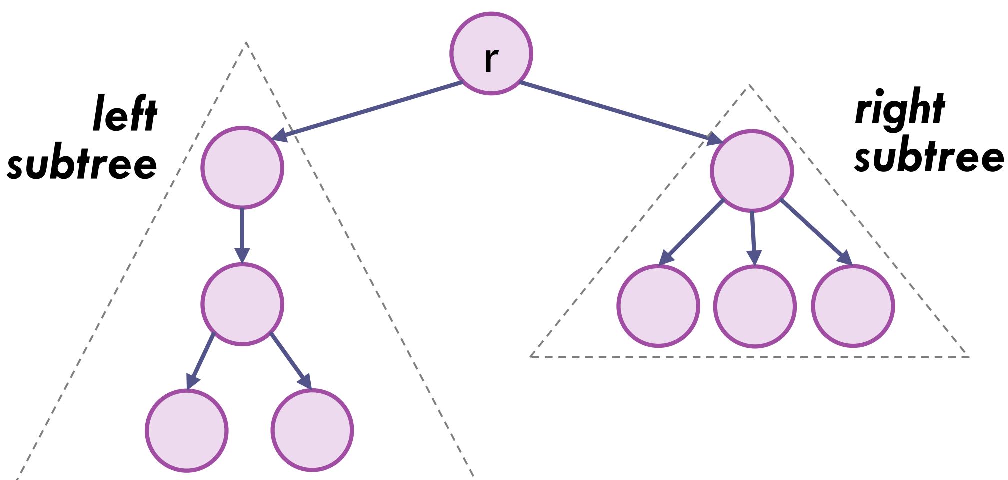
Binary Trees as Recursive Data Structures

- A binary tree is either empty … ← Anchor

  or

- Consists of a node called the root
- Root points to two disjoint binary (sub)trees left and right (sub)tree

} Inductive step



Tree Traversal is Also Recursive (Preorder example)

If the binary tree is empty then
  do nothing

Anchor

Else
  N: Visit the root, process data
  L: Traverse the left subtree
  R: Traverse the right subtree

Inductive/Recursive step

3 Types of Tree Traversal

- If the pointer to the node is not NULL:
  - *Preorder:* Node, Left subtree, Right subtree
  - *Inorder*: Left subtree, Node, Right subtree
  - *Postorder:* Left subtree, Right subtree, Node

$\left.\begin{matrix} \\ \\ \\ \end{matrix}\right\}$ Inductive/Recursive step

```
template<class T>
void BinaryTree<T>::PreOrder(
          void(*Visit)(BinaryTreeNode<T> *u),
                      BinaryTreeNode<T> *t)
{// Preorder traversal.
   if (t) {Visit(t);
          PreOrder(Visit, t->LeftChild);
          PreOrder(Visit, t->RightChild);
          }
}
```

```
template <class T>
void BinaryTree<T>::InOrder(
          void(*Visit)(BinaryTreeNode<T> *u),
                      BinaryTreeNode<T> *t)
{// Inorder traversal.
   if (t) {InOrder(Visit, t->LeftChild);
          Visit(t);
          InOrder(Visit, t->RightChild);
          }
}

template <class T>
void BinaryTree<T>::PostOrder(
          void(*Visit)(BinaryTreeNode<T> *u),
                      BinaryTreeNode<T> *t)
{// Postorder traversal.
   if (t) {PostOrder(Visit, t->LeftChild);
          PostOrder(Visit, t->RightChild);
          Visit(t);
          }
}
```

Traversal Order

- Given expression

  `A – B * C + D`

- Child node: operand
- Parent node: corresponding operator
- Inorder traversal: infix expression

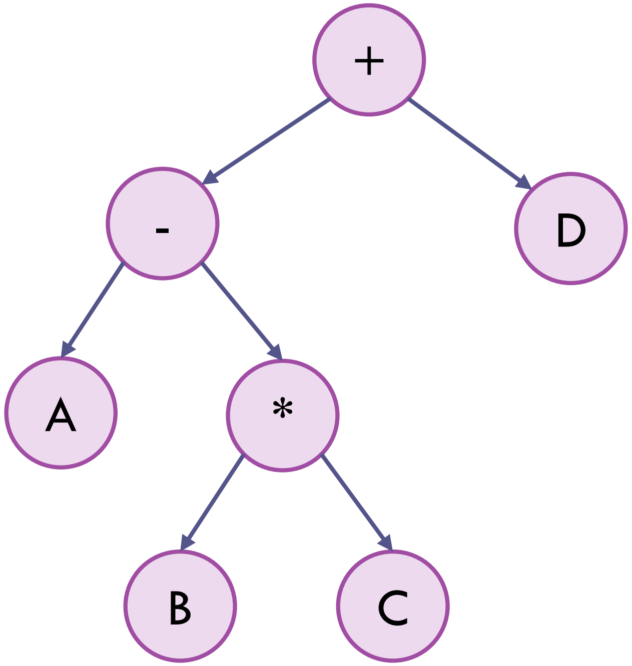
  `A – B * C + D`

- Preorder traversal: prefix expression

  `+ – A * B C D`

- Postorder traversal: postfix or RPN expression

  `A B C * – D +`



Preorder, Inorder and Postorder Traversals

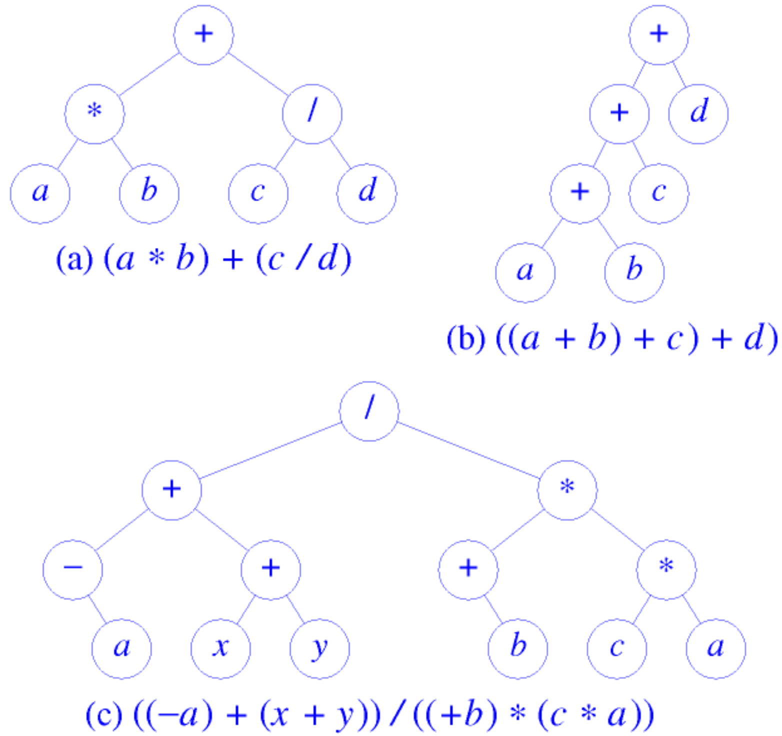


Figure 8.5 Expression trees

Preorder	$+*ab/cd$	$+++abcd$	$/+\text{-}a+xy*+b*ca$
Inorder	$a*b+c/d$	$a+b+c+d$	$-a+x+y/+b*c*a$
Postorder	$ab*cd/+$	$ab+c+d+$	$a-xy++b+ca**/$
	(a)	(b)	(c)

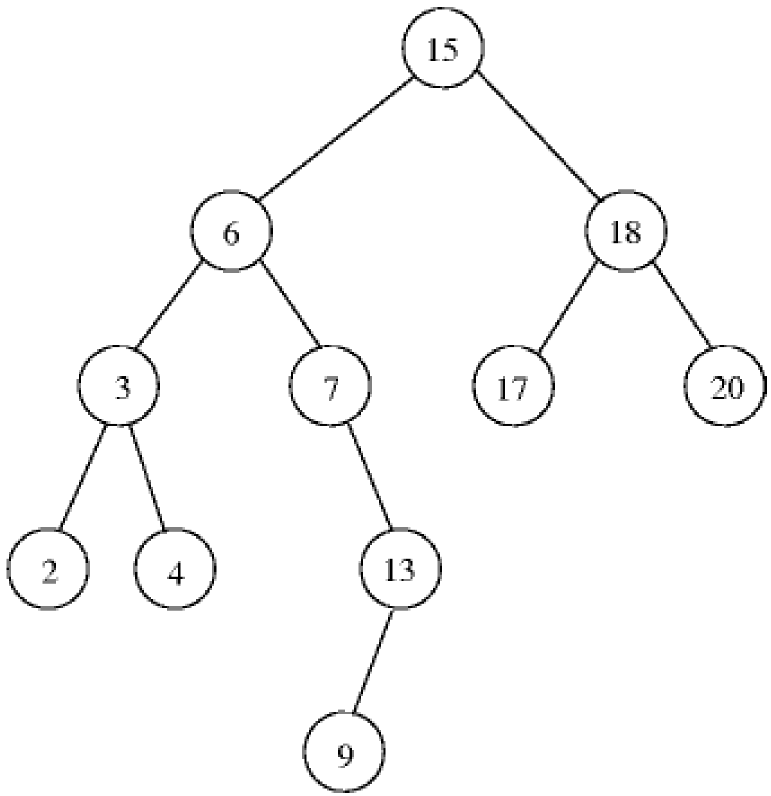
Figure 8.11 Elements of a binary tree listed in pre-, in-, and postorder

A Faster Way for Tree Traversal

- You may eye-ball the solution without using recursion.
- First emanating from each node a “hook.” Trace from left to right an outer envelop of the tree starting from the root. Whenever you touch a hook, you print out the node.
- Preorder:
  - put the hook to the left of the node
- Inorder:
  - put the hook vertically down at the node
- Postorder:
  - put the hook to the right of the node

Another Example (This is a Search Tree)

- Inorder (Left, Visit, Right): 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
- Preorder (Visit, Left, Right): 15, 6, 3, 2, 4, 7, 13, 9, 18,17, 20
- Postorder (Left, Right, Visit): 2, 4, 3, 9, 13, 7, 6, 17, 20,18, 15



Output Fully Parenthesized Infix Form

```
template <class T>
void Infix(BinaryTreeNode<T> *t)
{// Output infix form of expression.
  if (t) {
    cout << '(';
    Infix(t->LeftChild);  // left operand
    cout << t->data;      // operator
    Infix(t->RightChild); // right operand
    cout << ')';
  }
}

 +
/ \   returns ((a)+(b))
a  b
```

Infix to Prefix (Pre-order Expressions)

- Infix = In-order expression

1. Infix to postfix
2. postfix to build an expression tree
   1. Push operands into a stack
   2. If an operator is encountered, create a binary node with the operator as the root, push once as right child, push the 2^{nd} time as left child, and push the complete tree into the stack
3. With the expression tree, traverse in preorder manner
   - Parent-left-right

Binary Search Tree

Search on a Sorted Sequence

- Collection of *ordered* data items to be searched is organized in a list

  $x_1, x_2, \dots x_n$

- Assume == and < operators defined for the type
- Input: An array A of elements in sorted order, and an element x.
- Output: Find x if it exists; otherwise output "no".

4	7	10	15	19	20	42	54	87	90

- Linear search begins with the first item
  - continue through the list until target found
  - or reach end of list

Linear Search: Vector Based

```
template <typename t>
void LinearSearch (const vector<t> &v, const t &item,
                   boolean &found, int &loc)
{
      found = false;  loc = 0;
      for ( ; ; )
      {
            if (found || loc == v.size())
                  return;
            if (item == v[loc])
                  found = true;
            else
                  loc++;
      }
}
```

Binary Search: Vector Based

```
template <typename t>
void LinearSearch (const vector<t> &v, const t &item,
                   boolean &found, int &loc)
{
        found = false;
        int first = 0;
        last = v.size() - 1;
        for ( ; ; )
        {
                if (found || first > last) return;
                loc = (first + last) / 2;
                if (item < v[loc])
                        last = loc - 1;
                else if (item > v[loc])
                        first = loc + 1;
                else
                        /* item == v[loc] */
                        found = true;
        }
}
```

May be replaced by recursive codes with additional function parameters first and last

Binary Search

- Running time: $O(\log n)$
- Outperforms a linear search (infinitely faster asymptotically)
- Q: How to insert/delete an element x?

  A: Have to shift all elements after x, which requires $O(n)$ time.
- A: Or, we can use a linked list instead of an array
  - Insertion/deletion takes $O(1)$ time.
  - But, how to do a binary search on a list?

Dictionary

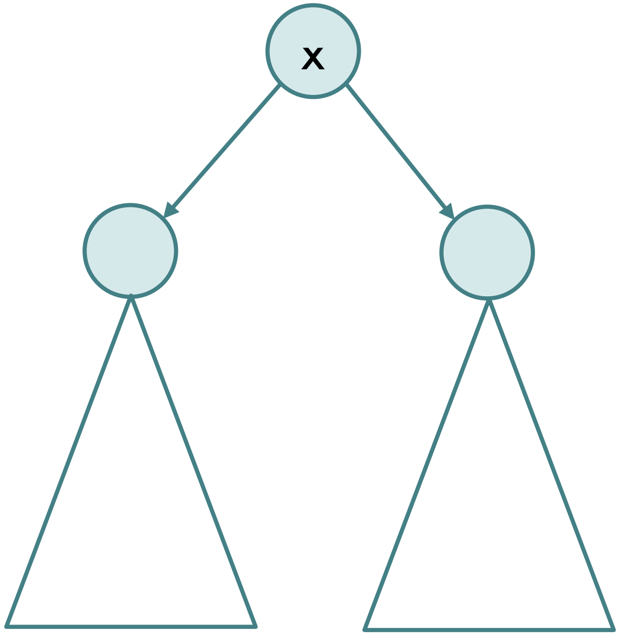
- A dictionary is a collection of elements
- Each element has a field called key
- No two elements have the same key value

```
AbstractDataType Dictionary {
instances
             collection of elements with distinct keys
Operations
   Create (): create an empty dictionary
   Search (k,x): return element with key k in x;
                 return false if the operation
                 fails, true if it succeeds
   Insert (x): insert x into the dictionary
   Delete (k,x): delete element with key k and
                 return it in x
}
```

Binary Search Tree (BST)

- Collection of data elements in a binary tree structure
- Stores keys in the nodes of the binary tree in a way so that searching, insertion and deletion can be done efficiently
- Every element has a key (or value) and no two elements have the same key (all keys are distinct)
- The keys (if any) in the left subtree of the root are smaller than the key in the root
- The keys (if any) in the right subtree of the root are larger than the key in the root
- The left and right subtrees of the root are also binary search trees

Binary Search Tree

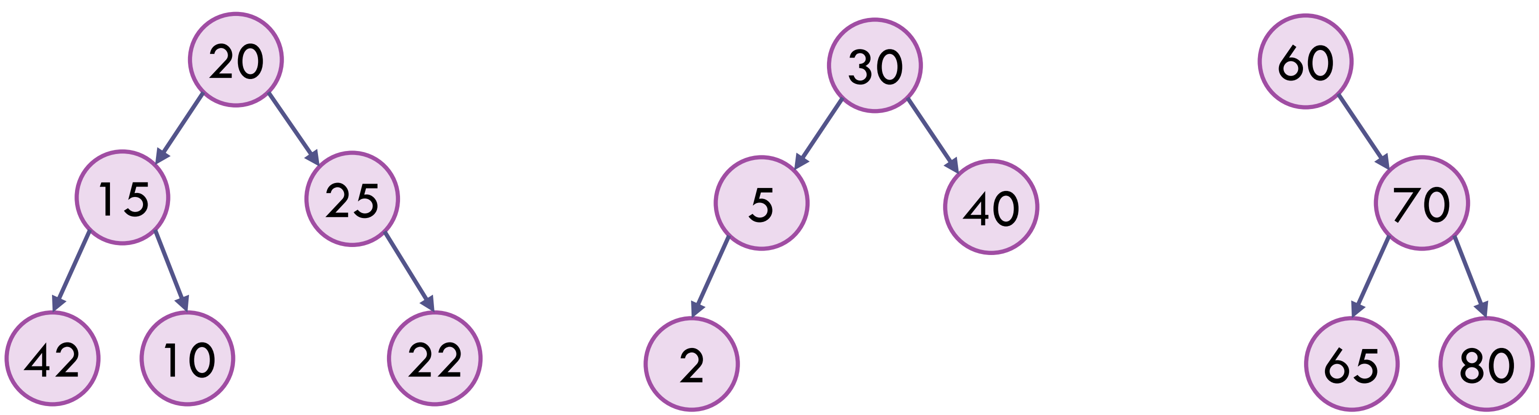


for any node y in this subtree
key(y) < key(x)

for any node z in this subtree
key(z) > key(x)

Examples of BST

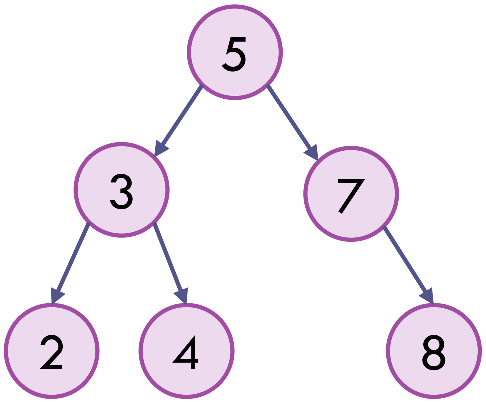
- For each node *x*,
  values in left subtree ≤ value in *x* ≤ value in right subtree
- a) is NOT a search tree, b) and c) are search trees

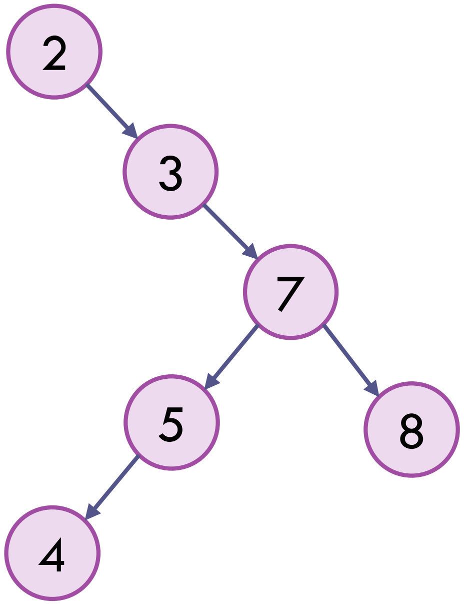


(a) **(b)** **(c)**

Binary Search Tree Property

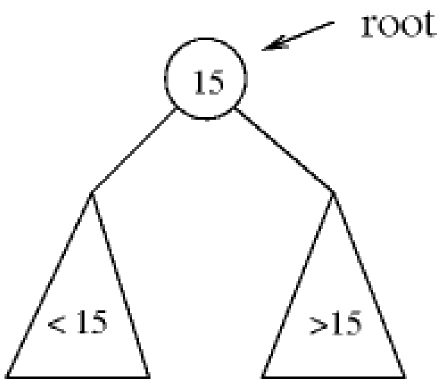
- Two binary search trees representing the same set





Sorting: Inorder Traversal for a Search Tree

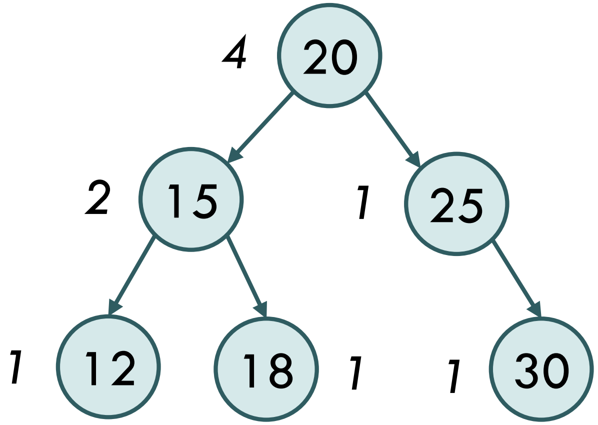
- Print out the keys in sorted order

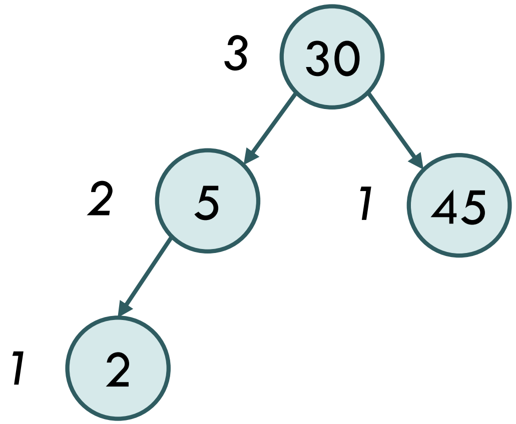


- A simple strategy is to
  1. print out all keys in left subtree in sorted order;
  2. print 15;
  3. print out all keys in right subtree in sorted order;

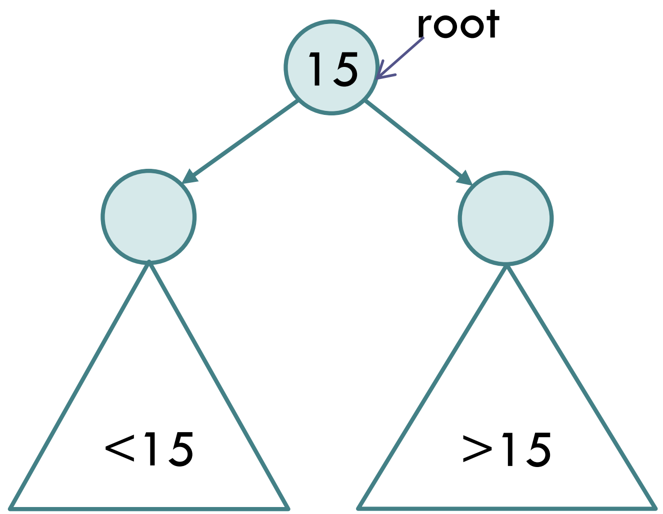
Indexed Binary Search Tree

- Derived from binary search tree by adding another field *LeftSize* to each tree node
- *LeftSize* gives the number of elements in the node's left subtree plus one
- An example (the number inside a node is the element key, while that outside is the value of *LeftSize*)
- It is the rank of the node for the search tree rooted at that node (rank is the position in the sorted order)
  - Can be used to figure out the rank of the node in the tree



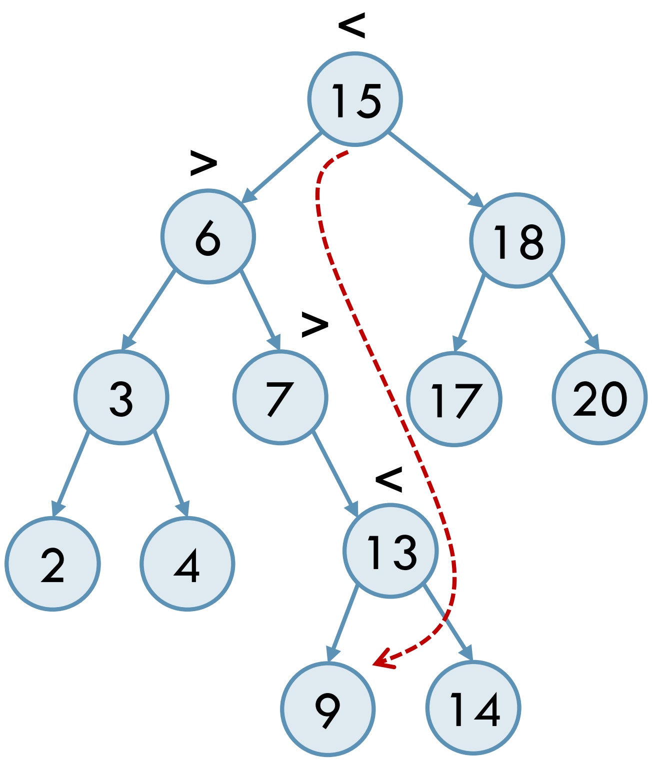


Tree Search



- If we are searching for 15, then we are done
- If we are searching for a key < 15, then we should search for it in the left subtree
- If we are searching for a key > 15, then we should search for it in the right subtree

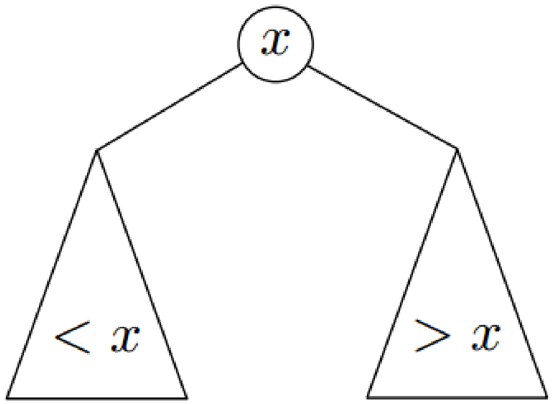
An Example



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Searching in BST



Tree-Search(T, k):

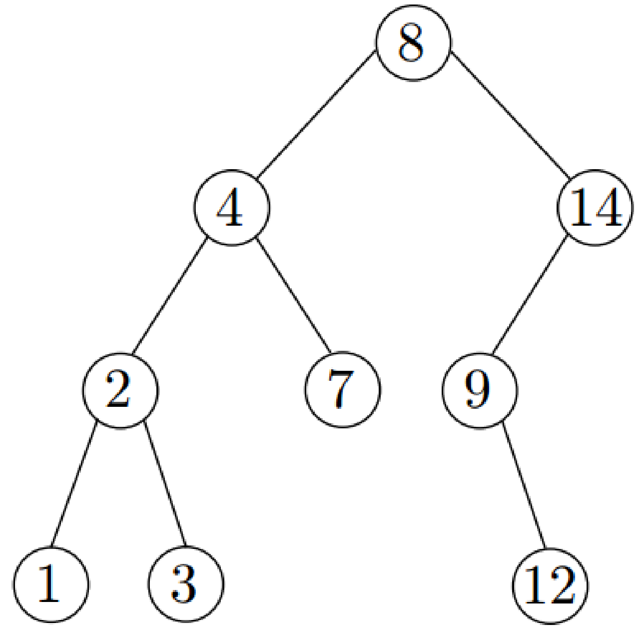
$x \leftarrow T.root$

while $x \neq nil$ **and** $k \neq x.key$ **do**

    if $k < x.key$ **then** $x \leftarrow x.left$

    else $x \leftarrow x.right$

return x



Assumption: All keys are distinct

Note 1: We can also find the neighbors of x (the closest numbers sandwiching x) if x is not present in T.

Note 2: The (worst-case) search time in a BST is O(height of the BST)

Building a BST from a sorted array

<u>**`BuildBST`**(A,p,r)**`:`**</u>
`if` $p > r$ **`then return`** nil
`create a node` x
$q \leftarrow \lfloor (p+r)/2 \rfloor$
$x.key \leftarrow A[q]$
$x.left \leftarrow$ **`BuildBST`**$(A, p, q-1)$
$x.right \leftarrow$ **`BuildBST`**$(A, q+1, r)$
`return` x

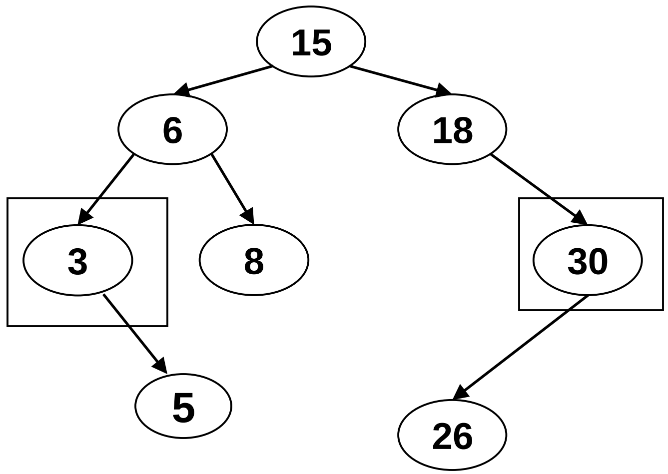
<u>**`First call:`**</u> $root \leftarrow$ **`BuildBST`**$(A, 1, n)$

- Running time
  - $T(n) = 2T(n/2) + O(1)$
  - $T(n) = O(n)$
- The resulting BST
  - Any search on the tree is exactly the same as doing a binary search on A
  - The height is $O(\log n)$

Find Min and Max

<u>Time Complexity</u>
Worse case?
Height of tree, which can be the total number of nodes if tree is not balanced!

Minimum element is always the left-most node.



Maximum element is always the right-most node.

Algorithm *Minimum*(x)
Input: x is the root.
Output: the node containing the minimum key.

```
while left(x) ≠ NULL
    do x := left(x);
return x;
```

Algorithm *Maximum*(x)
Input: x is the root.
Output: the node containing the maximum key.

```
while right(x) ≠ NULL
    do x := right(x);
return x;
```

Successor

The successor of a node ***x*** *is*

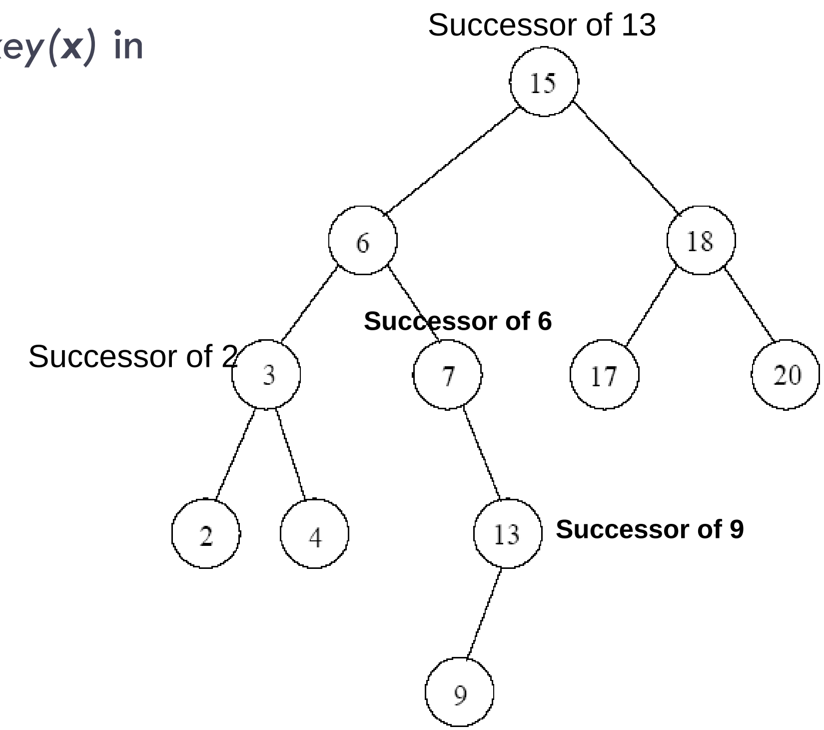
defined as:

- The node **y**, whose key(**y**) is the successor of *key(**x**)* in sorted order

  sorted order of this tree. (2,3,4,6,7,9,13,15,17,18,20)

Some examples:

Which node is the successor of 2?
Which node is the successor of 9?
Which node is the successor of 13?
Which node is the successor of 20? Null



Finding Successor:

Three Scenarios to Determine Successor

Successor(x)

x has right descendants
=> minimum(right(x))

Scenario I

x has no right descendants

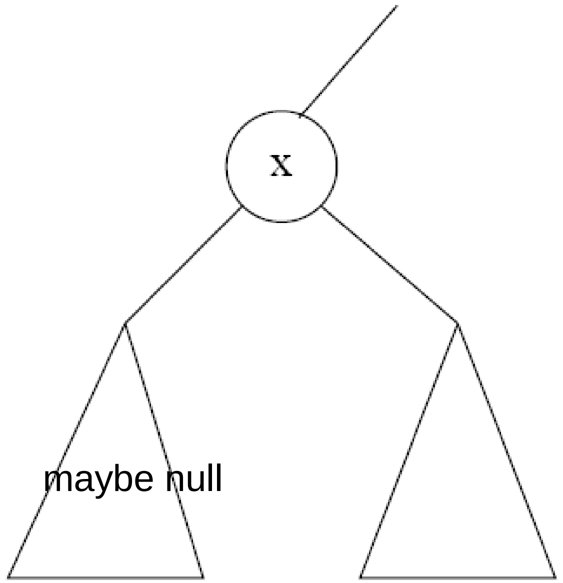
x is the *left* child of some node
=> parent(x)

Scenario II

x is the *right* child of some node

Scenario III

Scenario I: Node x Has a Right Subtree

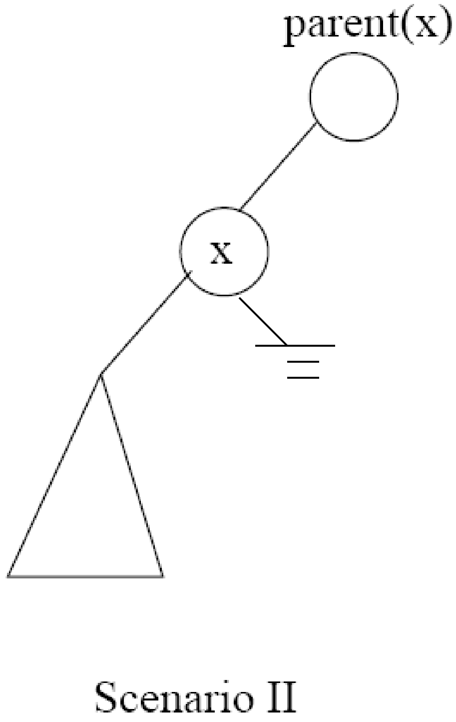


Scenario I

By definition of BST, all items greater than **x** are in this right sub-tree.

Successor is the minimum(right(**x**))

Scenario II: Node x Has No Right Subtree and x is the Left Child of Parent (x)



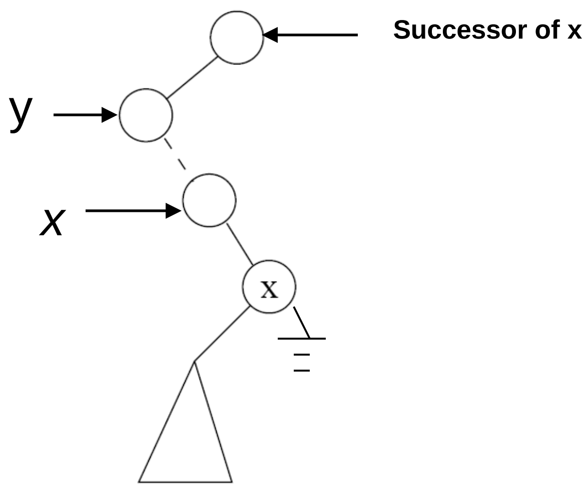
Successor is parent(**x**)

Why? The successor is the node whose key would appear in the next sorted order.

Think about traversal in-order. Who would be the successor of **x**?

The parent of x!

Scenario III: Node x Has No Right Subtree and Is Not a Left-Child of an Immediate Parent



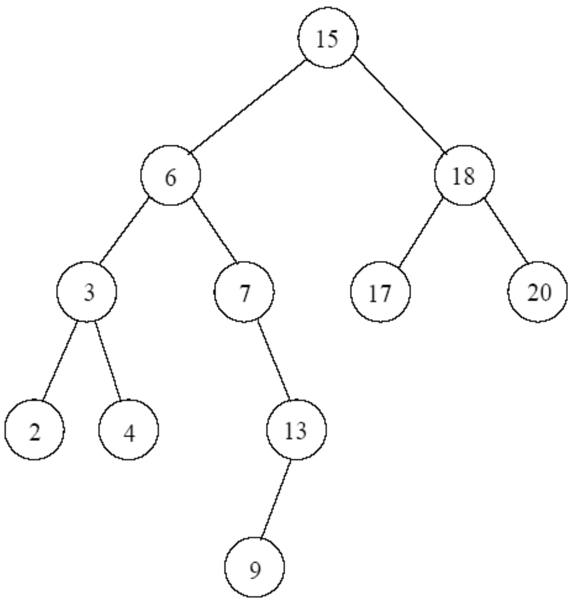
Scenario III

Keep moving up the tree until you find a parent which branches from the left().

Stated in Pseudo code.

$y :=$ parent(x);
while $y \neq$ NULL and $x =$ right(y)
  do $x := y$;
    $y :=$ parent(y);

Successor Pseudo-Codes



Verify this code with this tree.

Find successor of
3 → 4
9 → 13
13 → 15
18 → 20

Note that parent(root) = NULL

Algorithm *Successor*(x)
Input: x is the input node.

if right$(x) \neq$ NULL
  then return *Minimum*(right(x)); Scenario I
  else
    $y :=$ parent(x); Scenario II
    while $y \neq$ NULL and $x =$ right(y)
      do $x := y$; Scenario III
        $y :=$ parent(y);
return y;

Problem

- If we use a “doubly linked” tree, finding parent is easy.

```
class Node
{
        int data;
        Node *left;
        Node *right;
        Node *parent;
};
```

- But usually, we implement the tree using only pointers to the left and right node. ☹ So, finding the parent is tricky.

```
class Node
{
        int data;
        Node *left;
        Node *right;
};
```

For this implementation we need to use a Stack.

Use a Stack to Find Successor

```
Algorithm Successor(r, x)
Input: r is the root of the tree and x is the node.
initialize an empty stack S;
while key(r) ≠ key(x)
    do push(S, r);
        if key(x) < key(r)
            then r := left(r);
            else  r := right(r);
---------------------------------------------------
if right(x) ≠ NULL
    then return Minimum(right(x));
    else
        if S is empty
            then return NULL;
            else
                y := pop(S);
                while y ≠ NULL and x = right(y)
                    do x := y ;
                        if S is empty
                            then y := NULL;
                            else  y := pop(S);
                return y;
```

PART I

Initialize an empty Stack s.

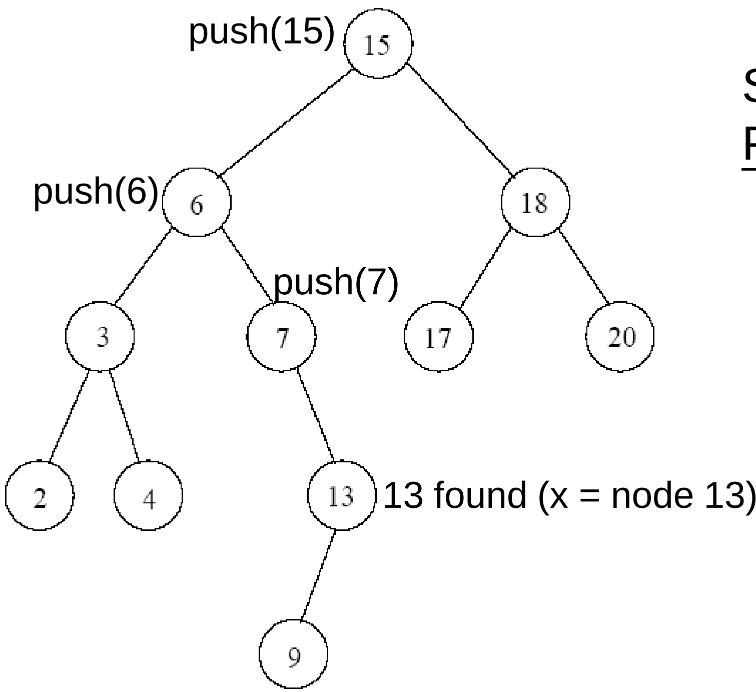
Start at the root node, and traverse the tree until we find the node **x**. Push all visited nodes onto the stack.

PART II

Once node **x** is found, find successor using 3 scenarios mentioned before.

Parent nodes are found by popping the stack!

An Example

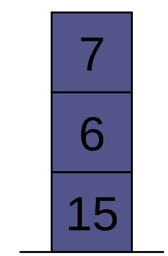


Successor(root, 13)

<u>Part I</u>

Traverse tree from root to find 13

order -> 15, 6, 7, 13



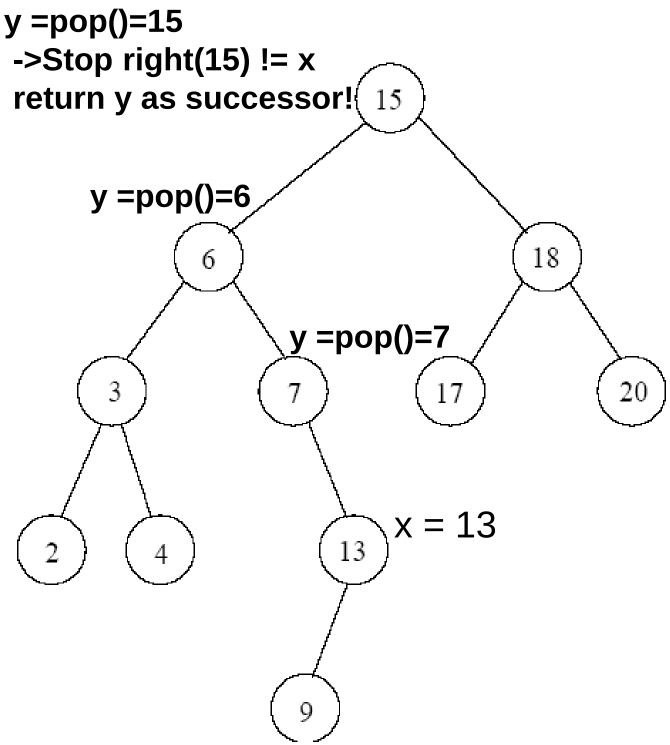
Stack s

Algorithm *Successor*(r, x)

Input: r is the root of the tree and x is the node.

```
initialize an empty stack S;
while key(r) ≠ key(x)
    do push(S, r);
       if key(x) < key(r)
          then r := left(r);
          else r := right(r);
```

Example

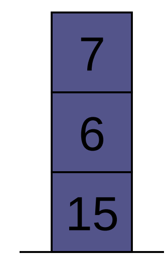


Successor(root, 13)

Part II

Find Parent (Scenario III)

```
y=s.pop()
while y!=NULL and x=right(y)
  x = y;
  if s.isempty()
     y=NULL
  else
     y=s.pop()
  loop
return y
```



Stack s

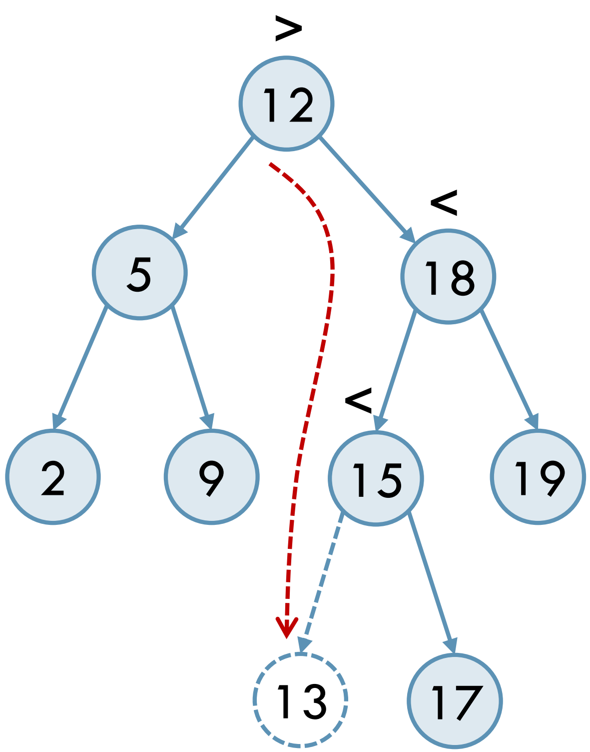
```
if right(x) ≠ NULL
   then return Minimum(right(x));
   else
        if S is empty
           then return NULL;
           else
              y := pop(S);
              while y ≠ NULL and x = right(y)
                 do x := y ;
                    if S is empty
                       then y := NULL;
                       else y := pop(S);
              return y;
```

A Leaner Approach for Case III

- Observe that:
  - x must be in the left branch of its successor y, because it is smaller in value
  - To get to x from left(y), we always traverse right, i.e., the value is increasing beyond left(y). The value never exceeds y, as x is on the left of y.
  - If we plot the values from y to x against the nodes visited, it is hence of a "V" shape, starting from y, dropping to some low value, and then increasing gradually to x, a value below y
- Using stack storing the path from the root to x, we hence can detect the right turn in the reverse path simply as follows:
  - Keep popping the stack until the key is higher than the value x. This must be its successor.

```
while (!s.empty()){
  y = s.pop();
  if( y > x)
    return y; // the successor
}
return NULL;  // empty stack; successor not found
```

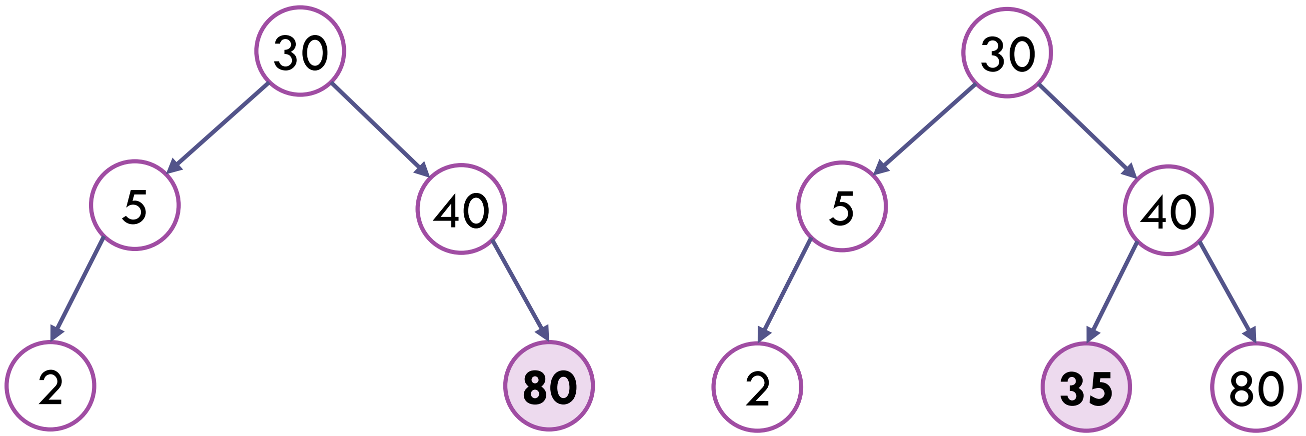
Insertion



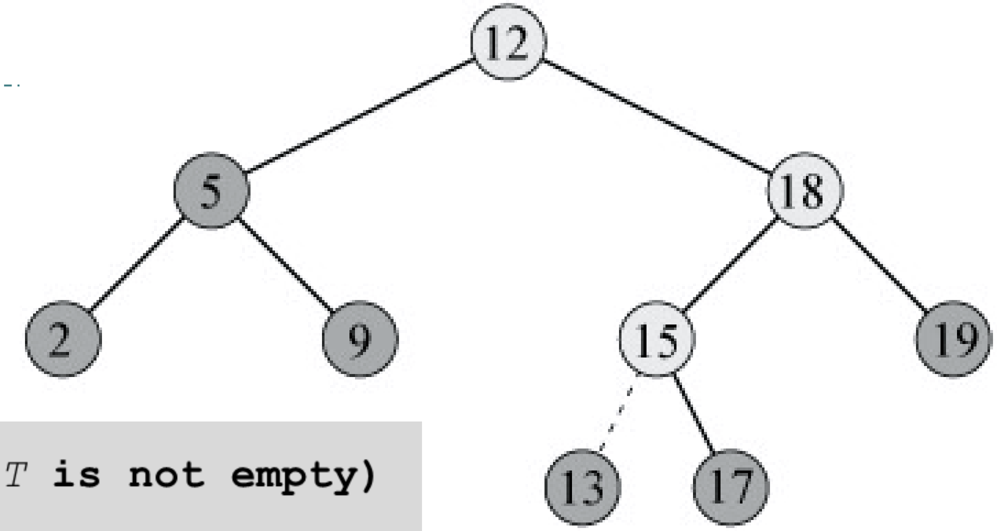
- Insert a new key into the binary search tree
- The new key is always inserted as a new leaf
- Example: Insert 13 ...

Insertion: Another Example

- First add 80 into an existing tree
- Then add 35 into it



Insert into a BST



Tree-Insert(T,k)**:** (assuming T is not empty)

$x \leftarrow T.root$

while $x \neq nil$ **do**

$\quad y \leftarrow x$

$\quad$**if** $k < x.key$ **then** $x \leftarrow x.left$

$\quad$**else** $x \leftarrow x.right$

create a new node z

$z.key \leftarrow k, z.left \leftarrow nil, z.right \leftarrow nil$

if $k < y.key$ **then** $y.left \leftarrow z$

else $y.right \leftarrow z$

The insertion time is O(height of the BST)

Inserting into a BST (1/2)

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Insert(const E& e)
{
    // Insert e if not duplicate.
    BinaryTreeNode<E> *p = this->root, // search pointer
                      *pp = 0; // parent of p
    // find place to insert
    while (p) {
        // examine p->data
        pp = p;
        // move p to a child
        if (e < p->data) p = p->LeftChild;
        else if (e > p->data) p = p->RightChild;
        else throw BadInput(); // duplicate
    }
```

May be replaced by recursive codes with an additional function parameter of binary tree node pointer

Inserting into a BST (2/2)

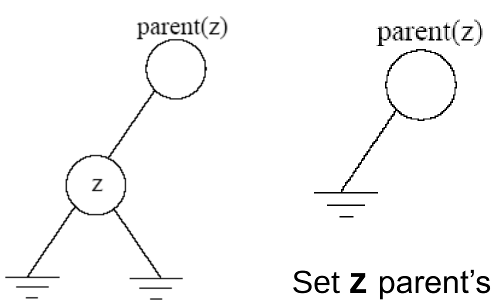
```
    // get a node for e and attach to pp
    BinaryTreeNode<E> *r = new BinaryTreeNode<E> (e);

    if (root) {
        // tree not empty
        if (e < pp->data) pp->LeftChild = r;
        else pp->RightChild = r;
    }
    else // insertion into empty tree
        root = r;

    return *this;
}
```

BST Deletion: Delete Node z from Tree

Three cases for deletion

Case I

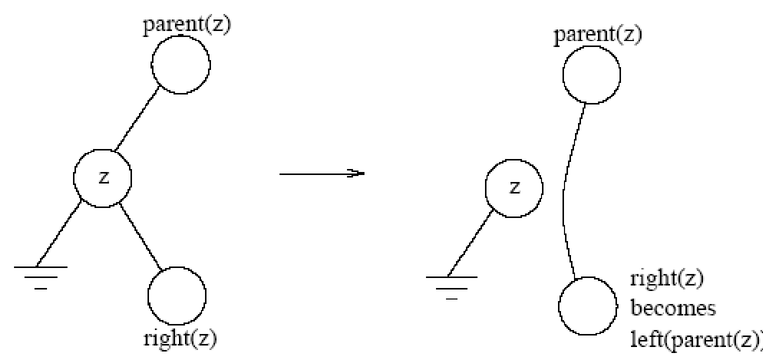
Node **z** is a leaf



Set **z** parent's pointer to **z** to NULL

Case II

Node **z** has exactly 1 (left or right) child

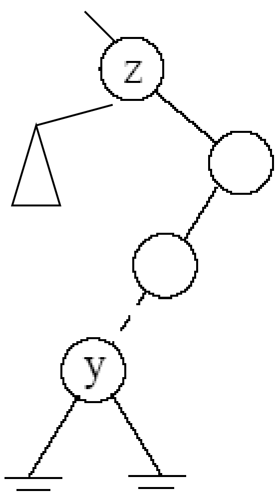


Modify appropriate parent(**z**) to point to **z**'s child (Parent adoption)

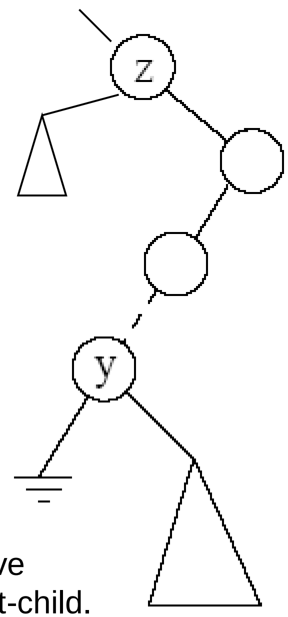
Case III: Node z Has 2 Children

Step 1.
Find successor y of 'z' (i.e. y = successor(z))

Since z has 2 children, successor is **y=minimum(right(z))**



or

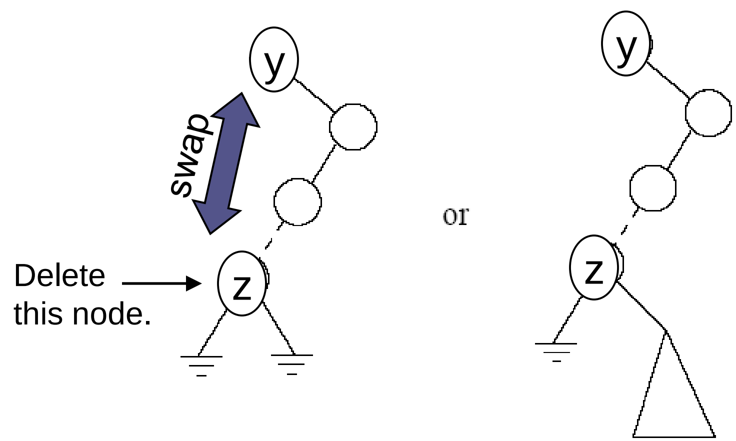


Successor **y** of **z** will have
no children or only a right-child.

Why? Look at the definition of
minimum(..)

Step 2.
Swap keys of z and y.

Now delete node y (which now has value z)!
This *deletion* is either case I or II.



Case I Case II

(deletion of node "z" is
always going to be Case I or II)

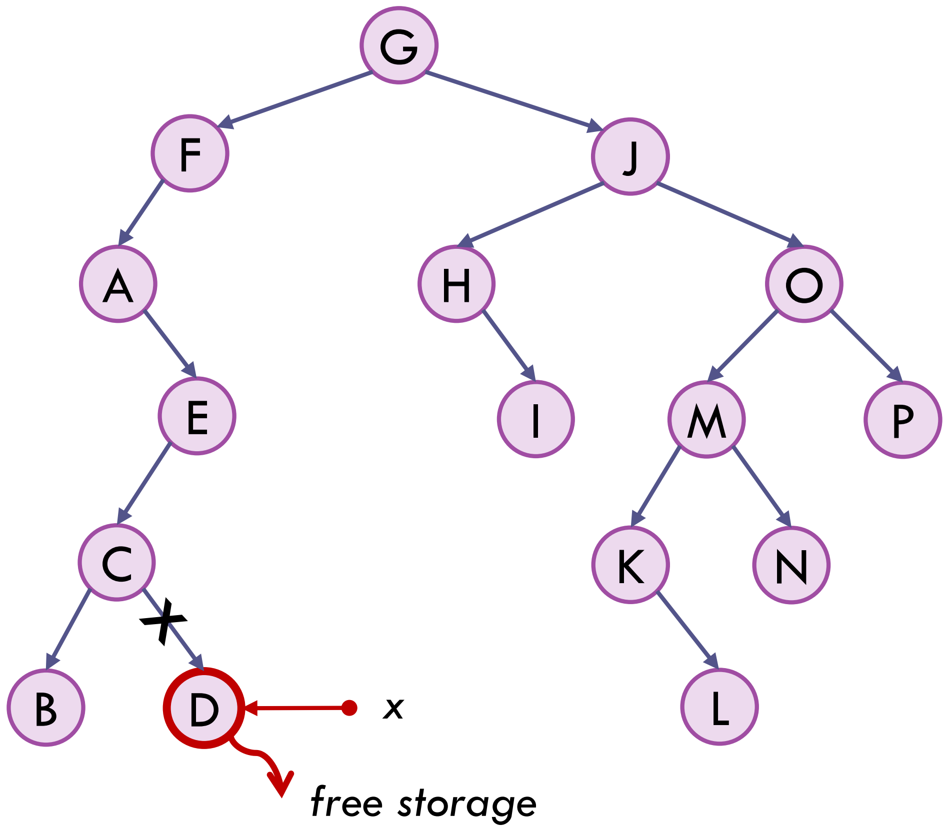
Special Case:
Deleting the Root with 1 Child Descendant

- Move the root to the child

A Deletion Example

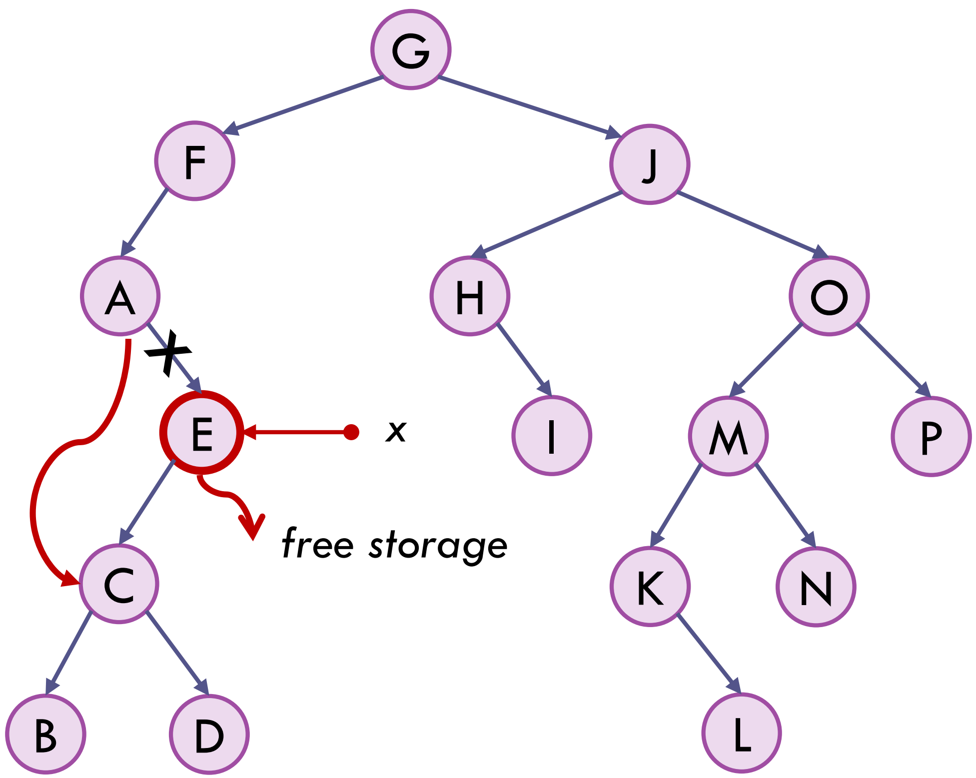
Three possible cases to delete a node x from a BST

1. The node x is a leaf



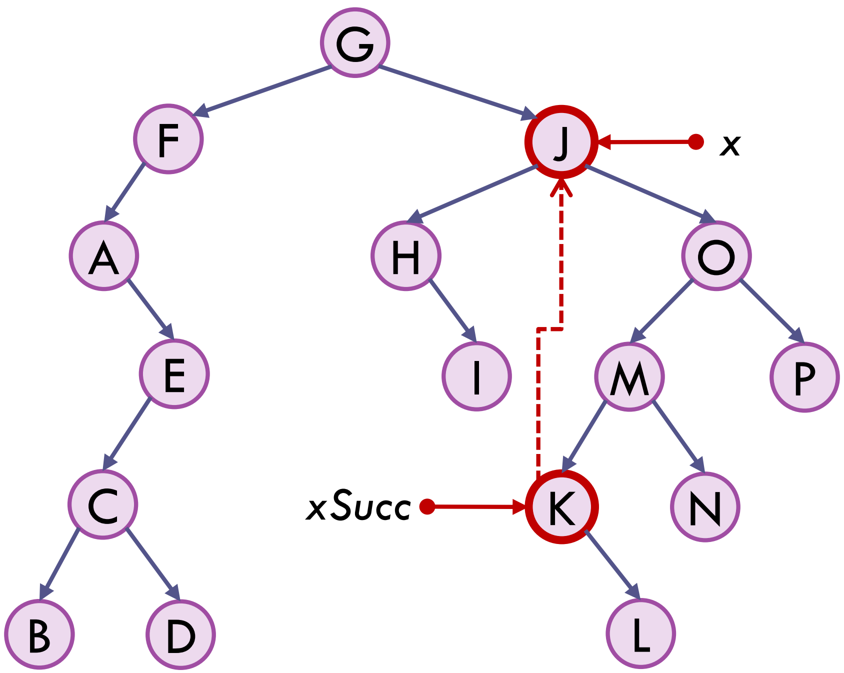
A Deletion Example (Cont.)

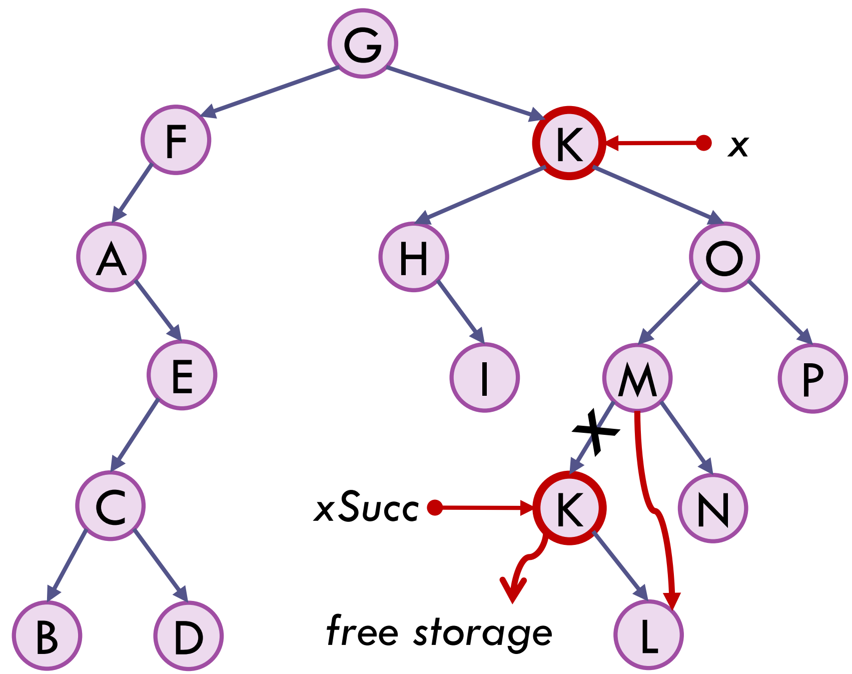
2. The node x has one child



A Deletion Example (Cont.)

3. *x* has two children



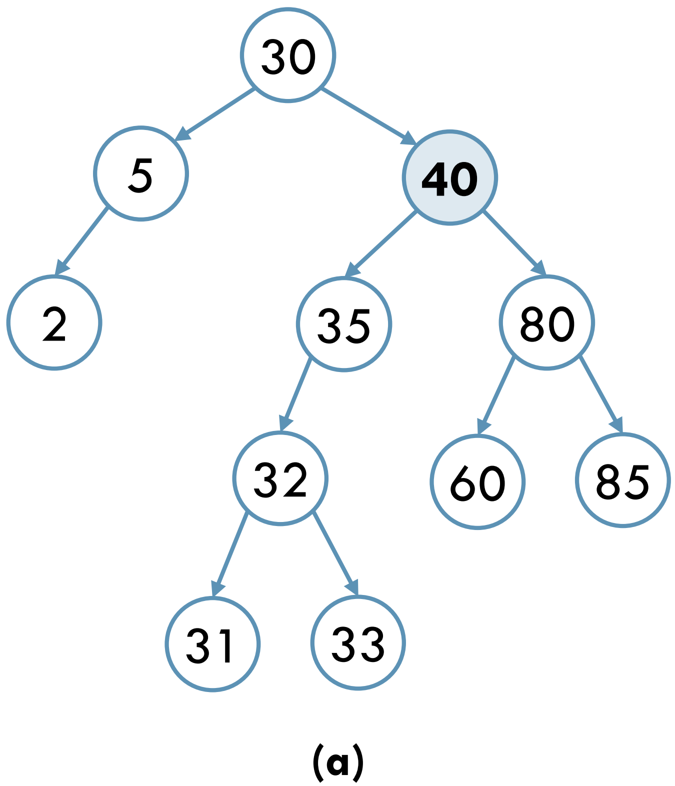


i) Replace contents of *x* with inorder successor (smallest value in the right subtree)

ii) Delete node pointed to by *xSucc* as described for cases 1 and 2

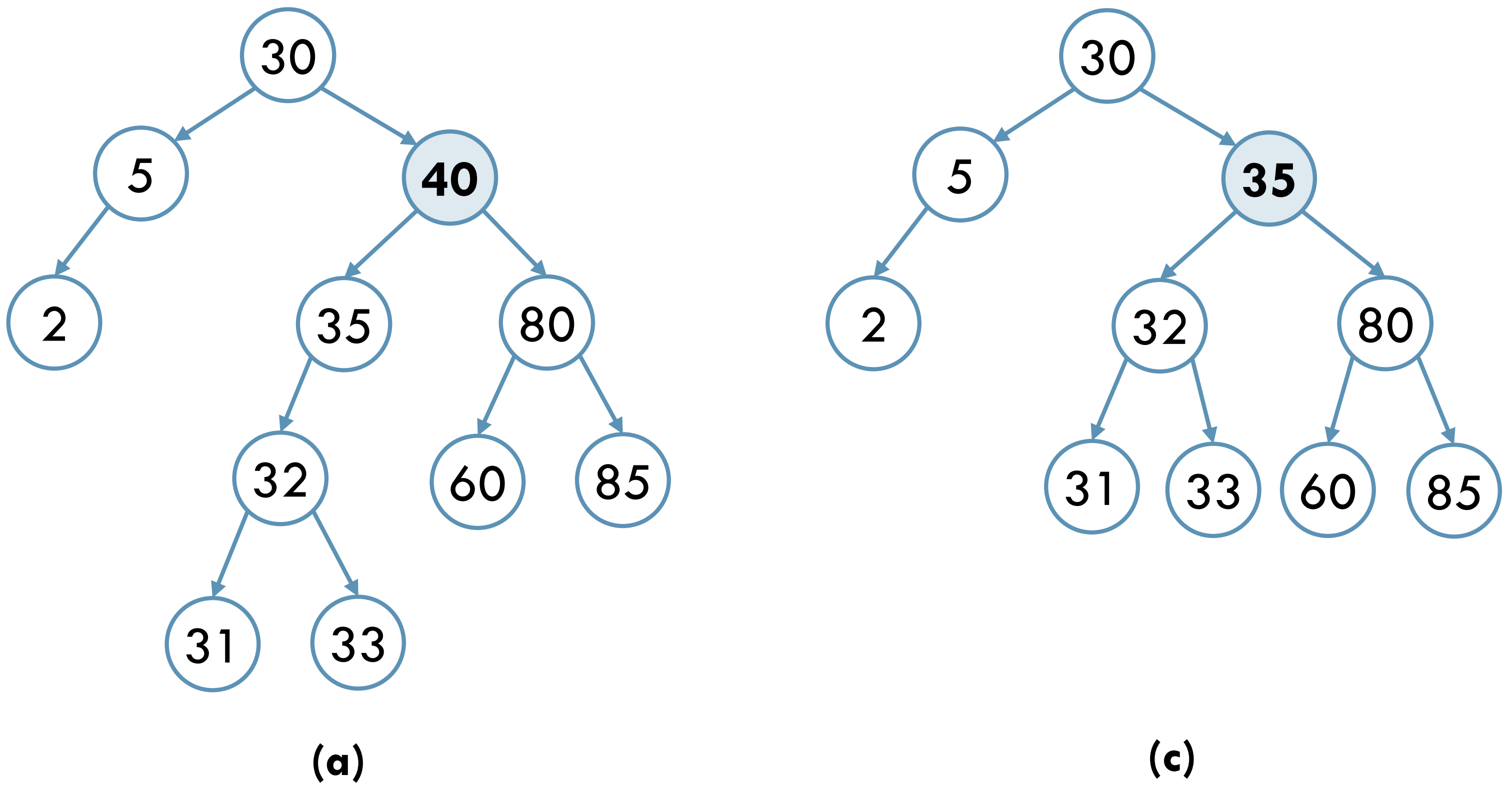
Another Deletion Example

- Removing 40 from (a) results in (b) using the smallest element in the right subtree (i.e., the successor)



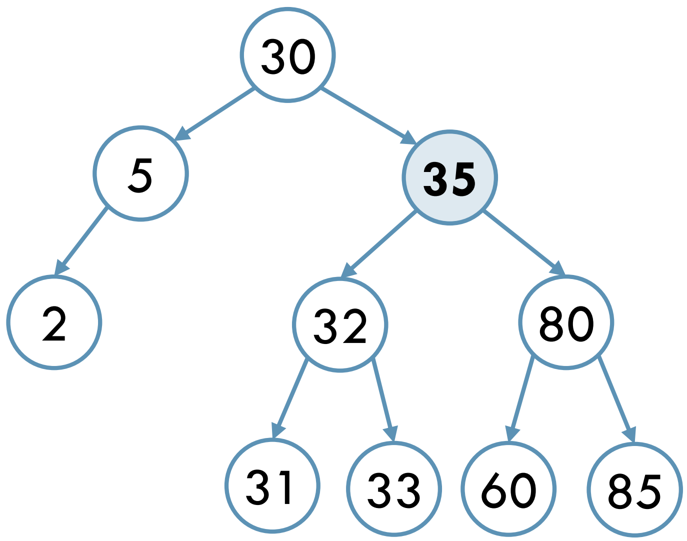
Another Deletion Example (Cont.)

- Removing 40 from (a) results in (c) using the largest element in the left subtree (i.e., the predecessor)

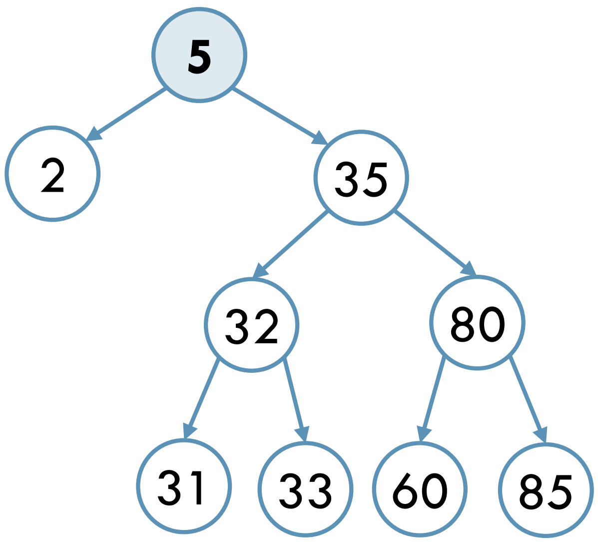


Another Deletion Example (Cont.)

- Removing 30 from (c), we may replace the element with either 5 (predecessor) or 31 (successor). If we choose 5, then (d) results.



(c)



(d)

Deletion Code (1/4)

- First Element Search, and then Convert Case III, if any, to Case I or II

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Delete(const K& k, E& e)
{
   // Delete element with key k and put it in e.
   // set p to point to node with key k (to be deleted)
   BinaryTreeNode<E> *p = root, // search pointer
                     *pp = 0; // parent of p
   while (p && p->data != k){
      // move to a child of p
      pp = p;
      if (k < p->data) p = p->LeftChild;
      else p = p->RightChild;
   }
```

Deletion Code (2/4)

```
if (!p) throw BadInput(); // no element with key k

e = p->data; // save element to delete

// restructure tree
// handle case when p has two children
if (p->LeftChild && p->RightChild) {
    // two children convert to zero or one child case
    // find predecessor, i.e., the largest element in
    // left subtree of p
    BinaryTreeNode<E> *s = p->LeftChild,
    *ps = p; // parent of s
    while (s->RightChild) {
        // move to larger element
        ps = s;
        s = s->RightChild;
    }
```

Deletion Code (3/4)

```
        // move from s to p
        p->data = s->data;
        p = s;    // move/reposition pointers for deletion
        pp = ps;
    }

    // p now has at most one child
    // save child pointer to c for adoption
    BinaryTreeNode<E> *c;
    if (p->LeftChild) c = p->LeftChild;
    else c = p->RightChild; // may be NULL

    // deleting p
    if (p == root) root = c;  // a special case: delete root
    else {
        // is p left or right child of pp?
        if (p == pp->LeftChild) pp->LeftChild = c;//adoption
        else pp->RightChild = c;
    }
```

Deletion Code (4/4)

```
    delete p;

    return *this;
}
```

Implementation: ADT of Binary Search Tree (BST)

- Construct an empty BST
- Determine if BST is empty
- Search BST for given item
- Insert a new item in the BST
  - Need to maintain the BST property
- Delete an item from the BST
  - Need to maintain the BST property
- Traverse the BST
  - Visit each node exactly once
  - The inorder traversal visits the nodes in ascending order

ADT of a BST

```
AbstractDataType BSTree {
instances
      binary trees, each node has an element with a
      key field; all keys are distinct; keys in the left
      subtree of any node are smaller than the key in
      the node; those in the right subtree are larger.
operations
      Create(): create an empty binary search tree
      Search(k, e): return in e the element/record with key k
                    return false if the operation fails,
                    return true if it succeeds
      Insert(e): insert element e into the search tree
      Delete(k, e): delete the element with key k and
                    return it in e
      Ascend(): output all elements in ascending order of
          key
}
```

A Simple Implementation without Inheritance

- tree_codes (BST.h and treetester.cpp)

```
template <typename DataType>
class BST
{
 public:
  // … member functions supporting BST operations
 private:
  /***** Binary node class *****/
  class BinNode
  {
  public:
    DataType data;
    BinNode * left;
    BinNode * right;

    // … BinNode constructors

  };// end of class BinNode declaration

  typedef BinNode *BinNodePointer;

  // … Auxiliary/Utility functions supporting member functions

 /***** Data Members *****/
  BinNodePointer myRoot;   // the root of the binary search tree

}; // end of class template declaration
```

Another Implementation with Inheritance, function pointers, and exception handling

- tree2_codes
  - Binary search tree is derived from binary tree
  - `E` is the record, and `K` is the key
  - bst.h:

```
template<class E, class K>
class BSTree : public BinaryTree<E> {
public:
    bool Search(const K& k, E& e) const;
    BSTree<E,K>& Insert(const E& e);
    BSTree<E,K>& InsertVisit
                 (const E& e, void(*visit)(E& u));
    BSTree<E,K>& Delete(const K& k, E& e);
    void Ascend() {InOutput();}
};
```

Skeleton of tree2_codes

- btnode.h: the node structure to be used in a binary tree

```
template <class T>
class BinaryTreeNode {
   //… friend functions
   public:
   // … constructors
   private:
      T data;     // data is a record
      BinaryTreeNode<T> *LeftChild,  // left subtree
                        *RightChild; // right subtree
};
```

- binary.h: binary tree

```
template<class T>
class BinaryTree {
   //… some friend functions
   public:
      //… member functions and note the use of
      // function pointers
   private:
      BinaryTreeNode<T> *root;  // pointer to root
      //helper/utility functions and static functions
};
```

Code Implementation (tree2_codes)

- bst.h

```
template<class E, class K>
bool BSTree<E,K>::Search(const K& k, E &e) const
{// Search for element that matches k.
   // pointer p starts at the root and moves through
   // the tree looking for an element with key k
   BinaryTreeNode<E> *p = this->root;
   while (p) // examine p->data
      if (k < p->data) p = p->LeftChild;  //implicit cast
      else if (k > p->data) p = p->RightChild;
      else {// found element
          e = p->data;   // copy the record to e
          return true;}
   return false;
}
```

May be replaced by recursive codes

- datatype.h: DataType is to be used in the binary node with field `data`

```
#ifndef DataType_
#define DataType_

class DataType {
   friend ostream& operator<<(ostream&, DataType);
   public:
      operator int() const {return key;} // implicit cast to obtain key
      int key;  // element key, maybe hashed from ID
      char ID;  // element identifier
};

ostream& operator<<(ostream& out, DataType x)
   {out << x.key << ' ' << x.ID << ' '; return out;}
#endif
```

Time Complexity of Binary Search Trees

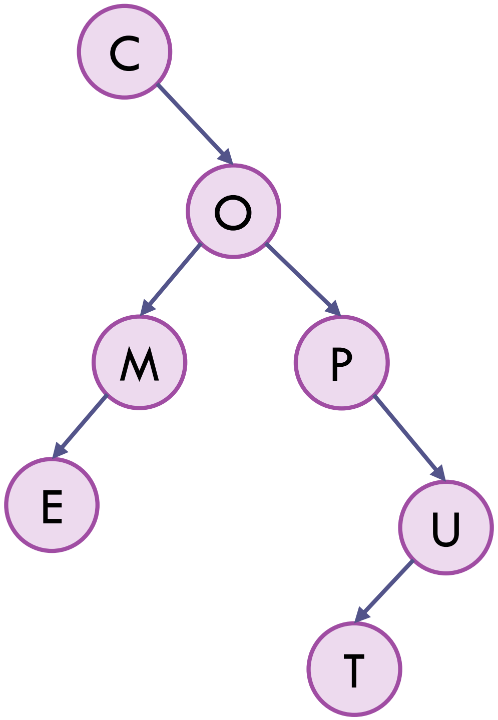
Find(x)	O(height of tree)
Min(x)	O(height of tree)
Max(x)	O(height of tree)
Insert(x)	O(height of tree)
Delete(x)	O(height of tree)
Traverse	O(N)

Binary Search Trees

- Problem
  - How can we predict the height of the tree?
- Many trees of different shapes can be composed of the same data
- How to control the tree shape?

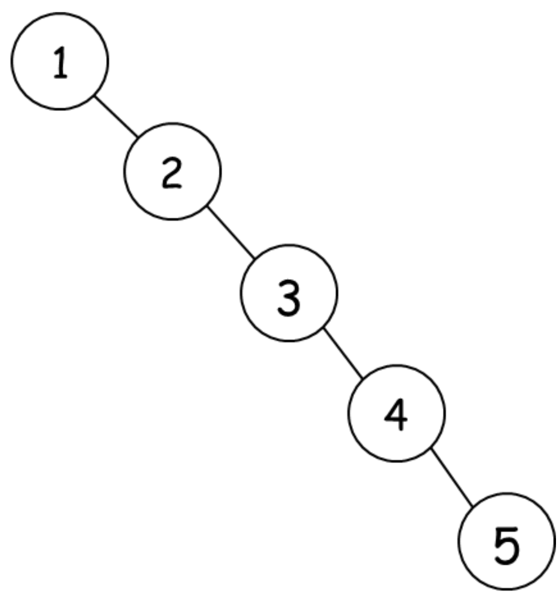
Problem of Lopsidedness

- Trees can be unbalanced
- Not all nodes have exactly 2 child nodes



Problem of Lopsidedness

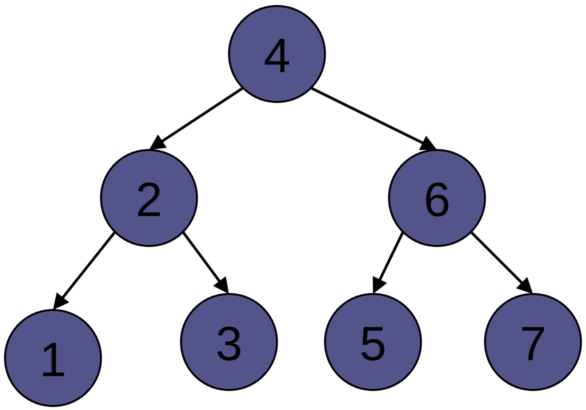
- Trees can be totally lopsided
- If we insert all elements in sorted order… each node would have a right child only
  - Degenerates into a linked list
- Need a way to restore "balance" so that the height is always $O(\log n)$.

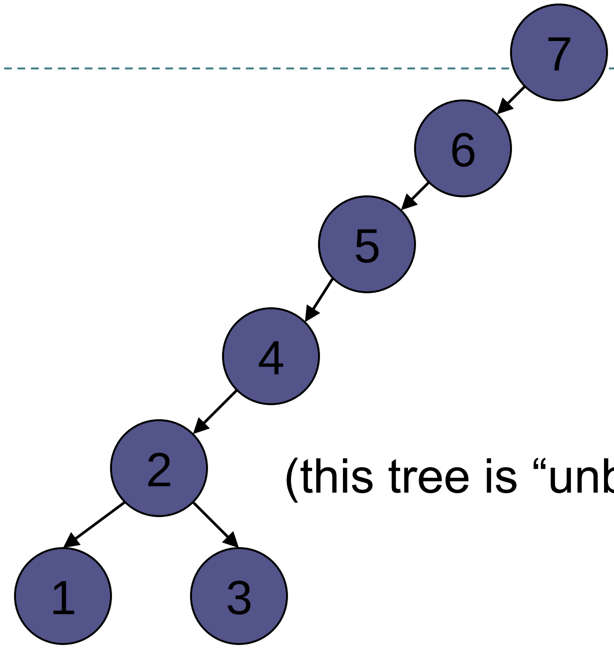


Processing time affected by "shape" of tree

Binary Search Tree

(we say this tree is “balanced”)





(this tree is “unbalanced”)

Tree 1
Same data as Tree 2

Tree 2
Same data as Tree 1

Which tree would you prefer to use?

Tree Examples

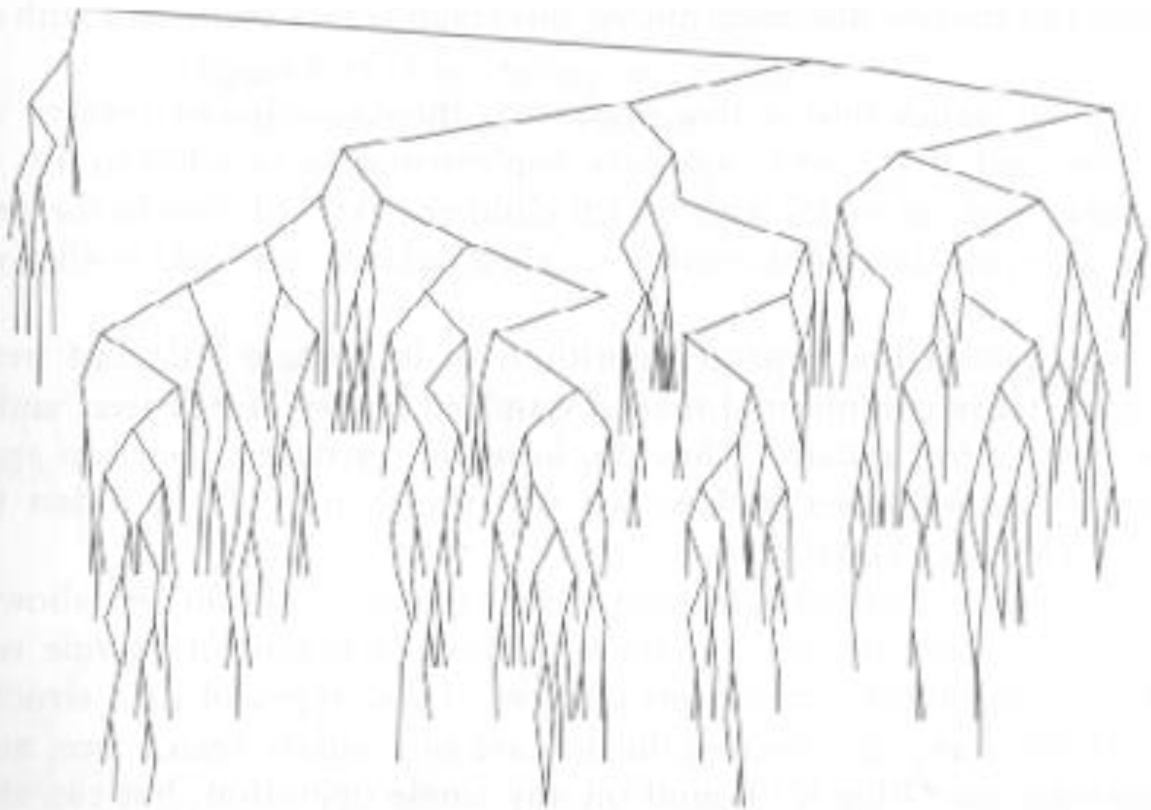
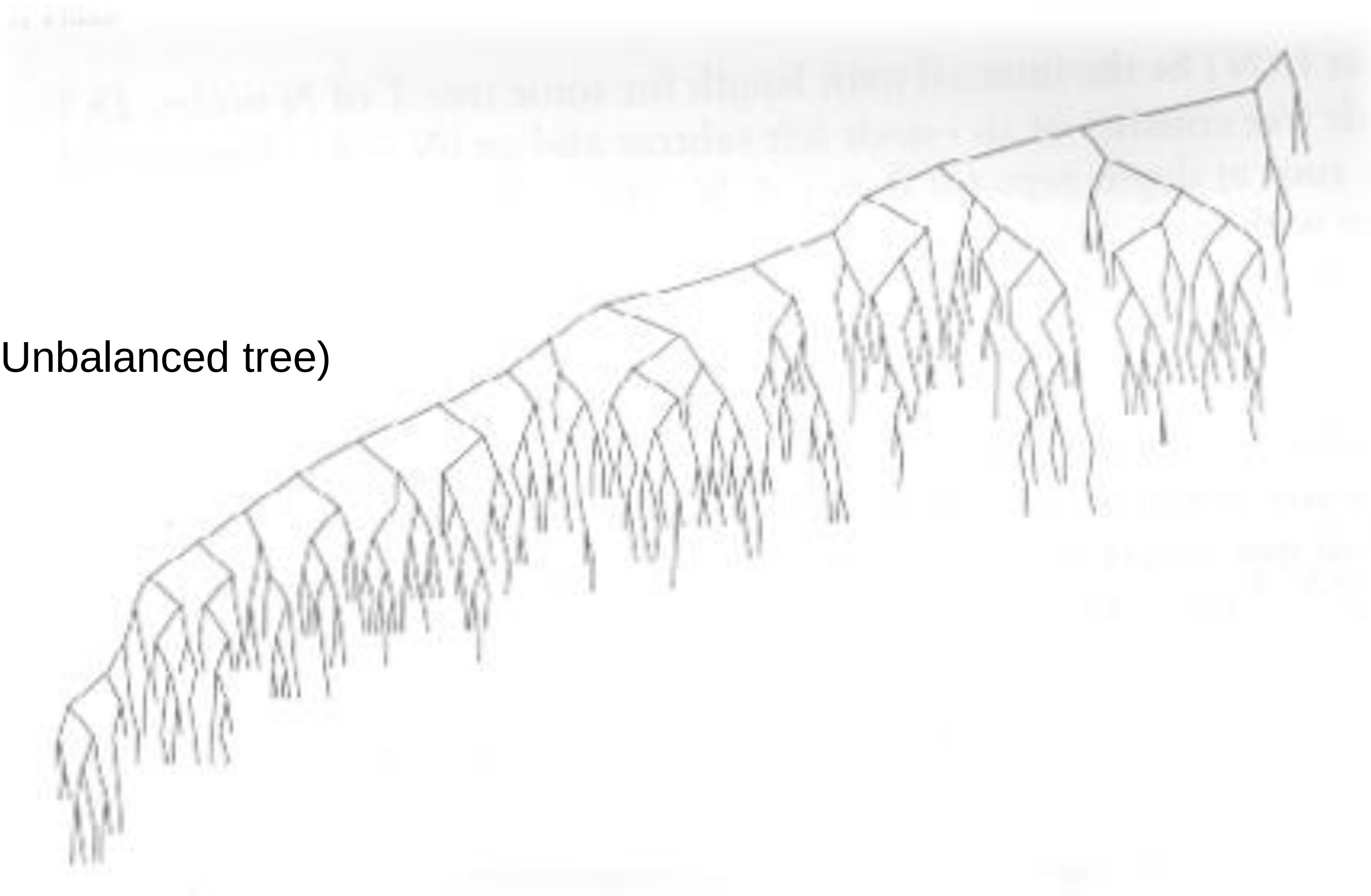


Figure 4.29 A randomly generated binary search tree

(Tree resulting from randomly generated input)

Tree Examples

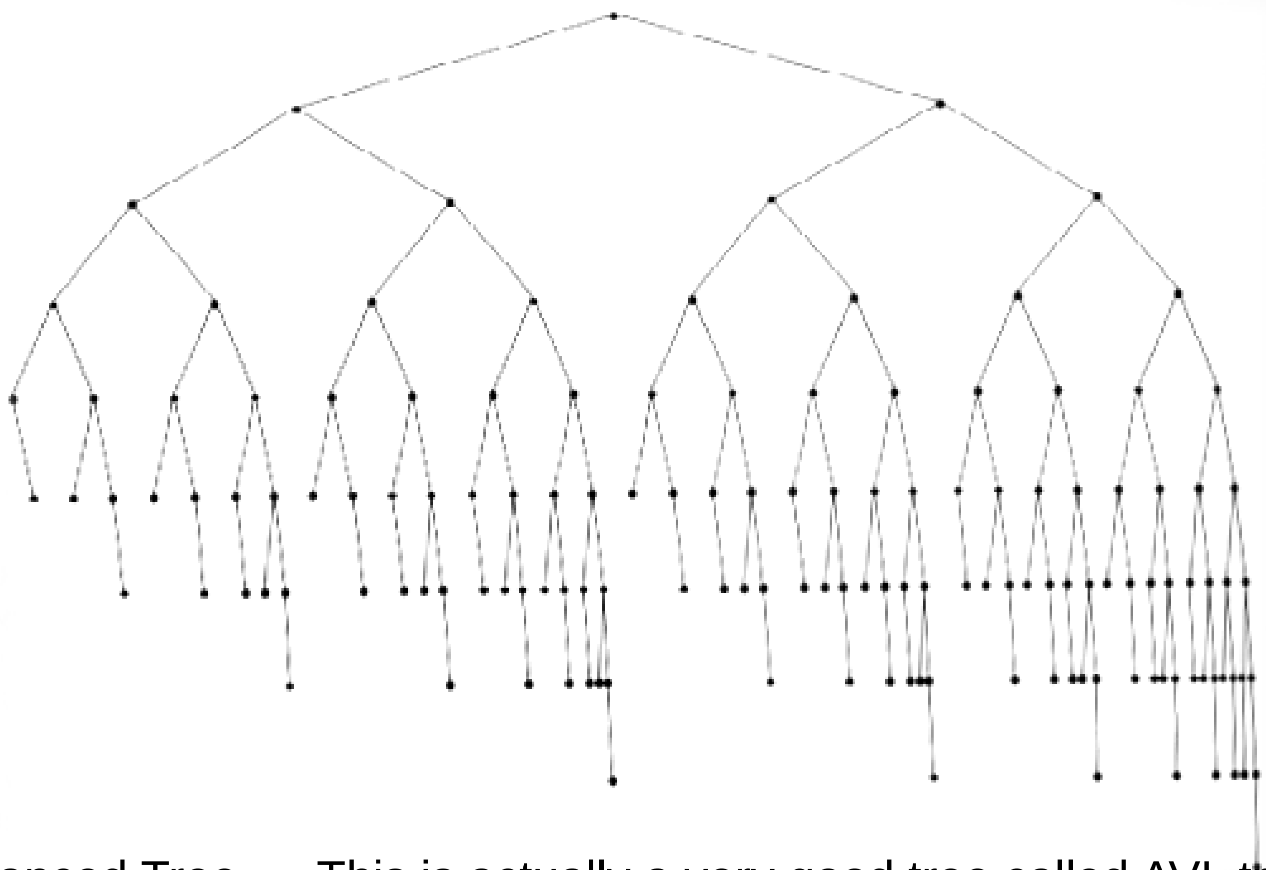
(Unbalanced tree)



How Fast is Sorting Using BST?

- n numbers (n large) are to be sorted by first constructing a binary tree and then read them in inorder manner
- Bad case: the input is more or less sorted
  - A rather "linear" tree is constructed
  - Total steps in constructing a binary tree: $1 + 2 + \ldots + n = n(n+1)/2 \sim n^2$
  - Total steps in traversing the tree: n
  - Total $\sim n^2$
- Best case: the binary tree is constructed in a balanced manner
  - Depth after adding i numbers: log(i)
  - Total steps in constructing a binary tree: $\log 1 + \log 2 + \log 3 + \log 4 + \ldots + \log n < \log n + \log n + \ldots + \log n = n \log n$
  - Total steps in traversing the tree: n
  - Total $\sim n \log n$, much faster
- It turns out that one cannot sort n numbers faster than nlog n
- For any arbitrary input, one can indeed construct a rather balanced binary tree with some extra steps in insertion and deletion
  - E.g., An AVL tree (two Soviet inventors, G. M. Adelson-Velskii and E. M. Landis, 1962)

An AVL Tree → A Rather Balanced Tree for Efficient BST Operations (See Animation)



(Balanced Tree . . This is actually a very good tree called AVL tree)

AVL Trees

(Balanced Trees)

The name comes from the inventors:
Adelson-Velskii and Landis Trees

AVL Tree Performance

- The height with *n* nodes is O(*log n*)
- For every value of *n*, there exists an AVL tree
- An n-element AVL search can be done in O(*log n*) time
- Insertion takes O(*log n*) time
- Deletion takes O(*log n*) time

Idea of AVL

- Make the binary tree a balanced tree
- What affects the tree's shape?
  - Insertion()
  - Deletion()
- So, let's modify the way we do insertions and deletions
- We need some definitions first . .

Height of a node

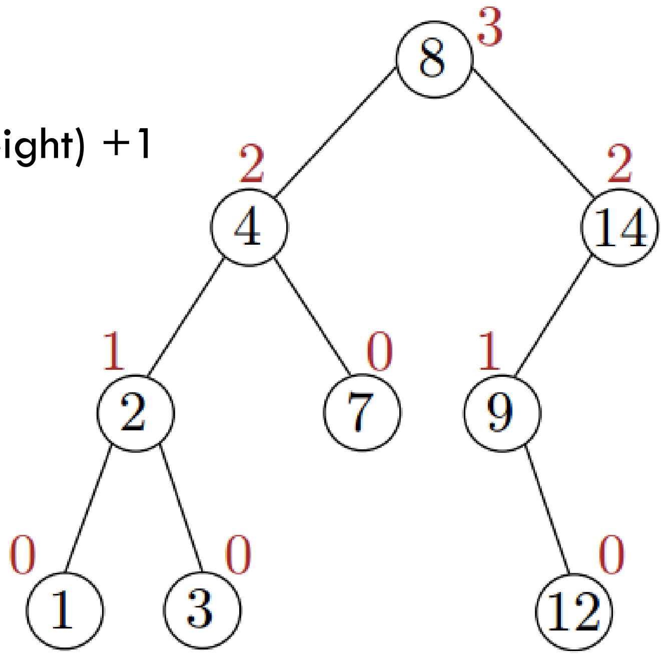
The height of a node in a tree is the number of edges on the longest downward path from the node to a leaf

Node height = max(left child height, right child height) +1

Leaves: height = 0

Tree height = root height

Empty tree: height = −1

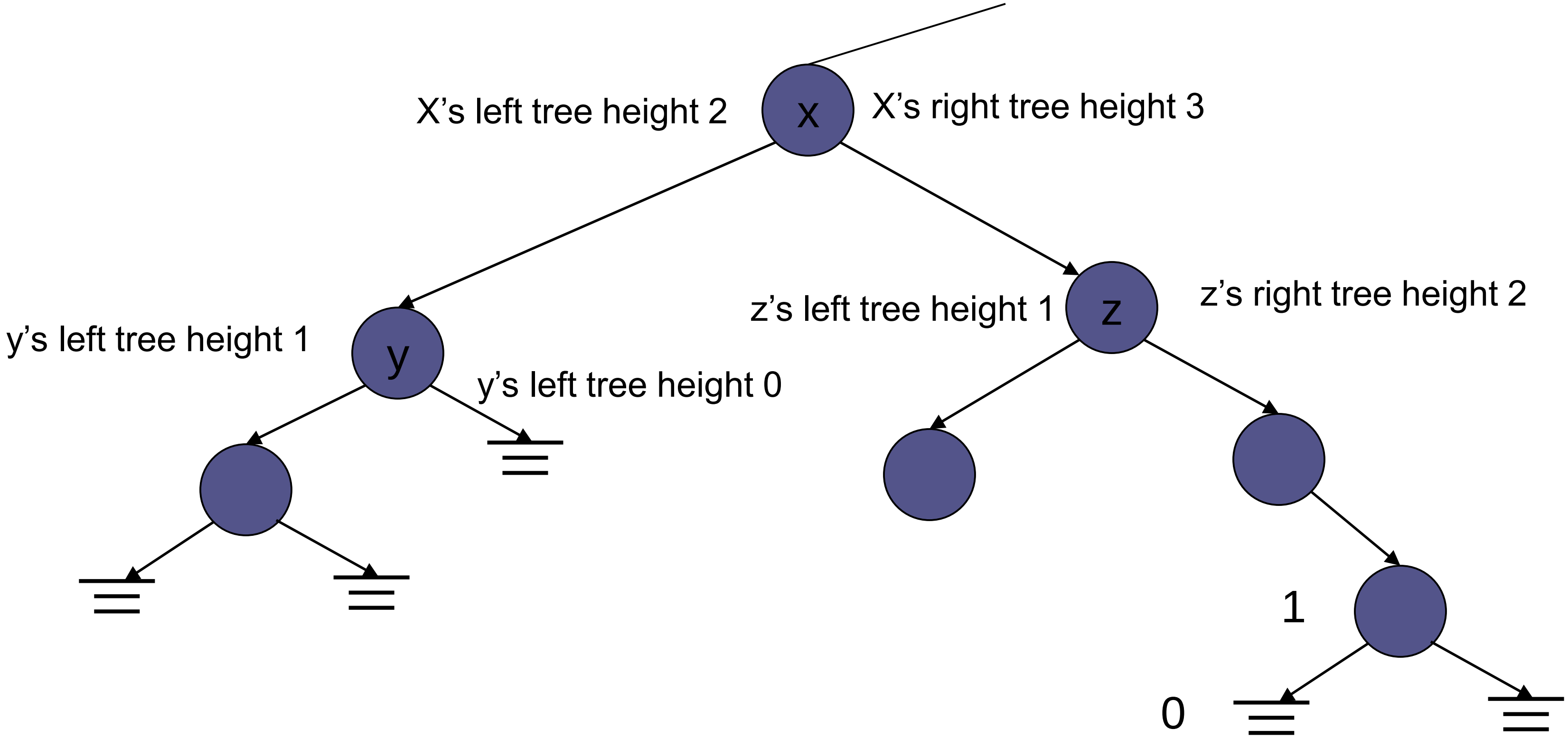


An Alternative Definition on the Height of a Node

- Height of a leaf is 1
- Height of a null pointer is 0
- The height of an internal node is the maximum height of its children plus 1

- Slightly different than our previous definition of height, which counted the number of edges

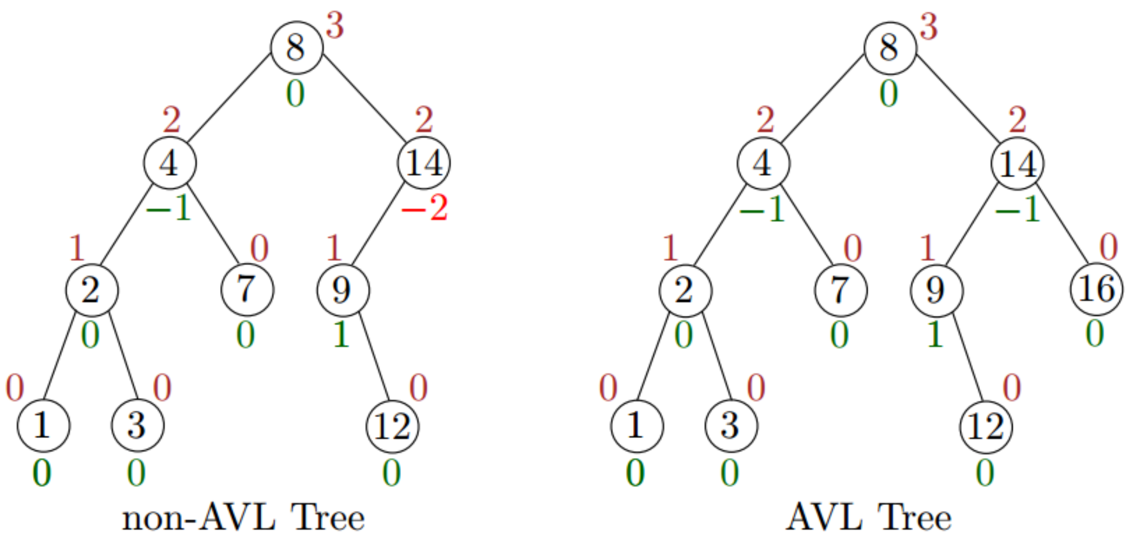
Height of Node example



What is the height of x? Maximum child height + 1, so height(x) = 4.

Balanced Binary Search Tree: AVL Tree

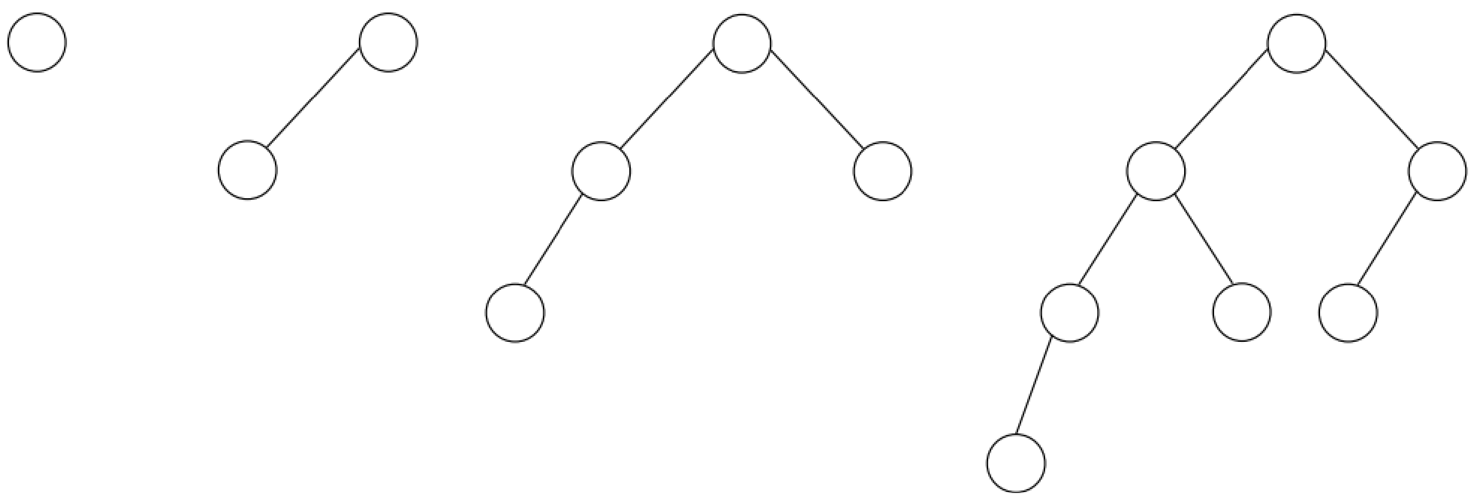
- An **AVL-tree** is a binary search tree in which for every node in the tree, (right height – left height) differs by at most 1.



- Q: Why is the height of an AVLtree $O(\log n)$?
- Q: How to maintain the AVL property after an insertion/deletion?

Height of an AVL-tree

- Observation: Equivalent to show that # nodes $\geq 2^{ch}$ for an AVL-tree of height h for some constant c.
  - Let n_h be the minimum number of nodes in an AVL tree of height h.
  - $n \geq n_h \geq 2^{ch} \Leftrightarrow h \leq \frac{1}{c}\log n = O(\log n)$
- We will prove the statement by mathematical induction



$n_0 = 1$ $\quad$ $n_1 = 2$ $\quad$ $n_2 = n_1 + n_0 + 1 = 4$ $\quad$ $n_3 = n_2 + n_1 + 1 = 7$

Height of an AVL-tree

- Claim: $n_h \geq 2^{h/2}$

Pf: (by induction on h)

$n_0 = 1, n_1 = 2$ (base case)

Recurrence: $n_h \geq \ n_{h-1} + n_{h-2} + 1$

$$
\begin{aligned}
n_h &= n_{h-1} + n_{h-2} + \ 1 \\
&\geq 2n_{h-2} \quad (n_{h-1} \geq n_{h-2} + 1) \\
&\geq 2 \cdot 2^{(h-2)/2} \qquad \text{(induction hypothesis)} \\
&= 2^{h/2}
\end{aligned}
$$

Theorem: The height of an AVL-tree of n nodes is $O(\log n)$.

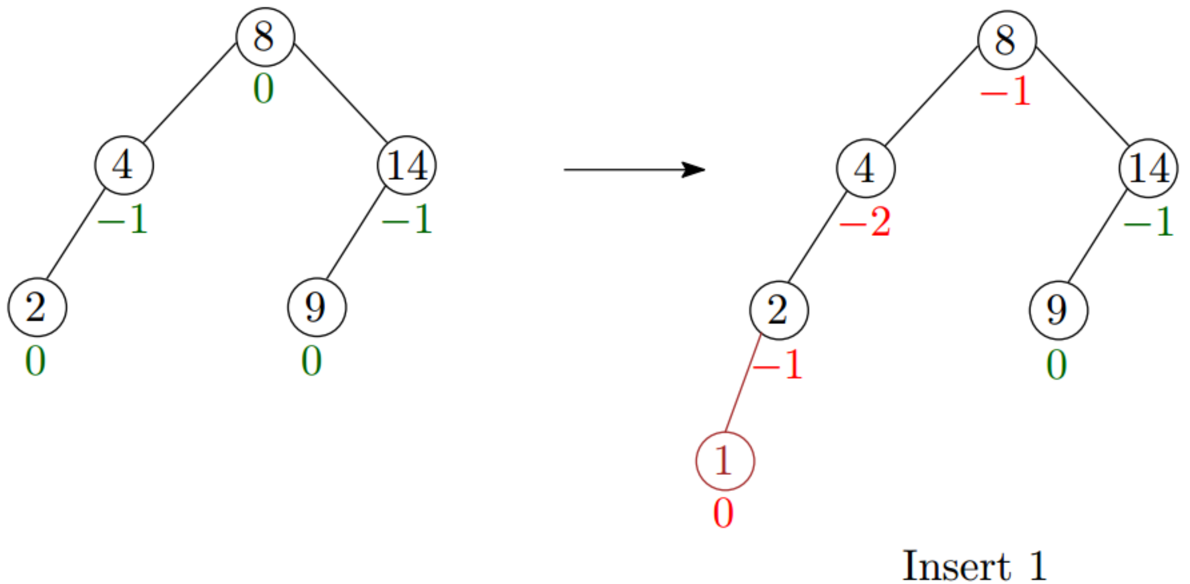
How does the AVL tree work?

- After insertion and deletion we will examine the tree structure and see if any nodes violates the AVL tree property
  - If the AVL property is violated, it means the heights of left(x) and right(x) differ by exactly 2

- If it does violate the property we can modify the tree structure using "rotations" to restore the AVL tree property

Restoring balance after an insertion

- After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered
  - Because only those nodes have their subtrees altered



- Idea:
  - Update the heights of these nodes from the insertion node
  - Stop when we find the lowest node A violating AVL tree property
  - We will fix the tree at A.

Rotations

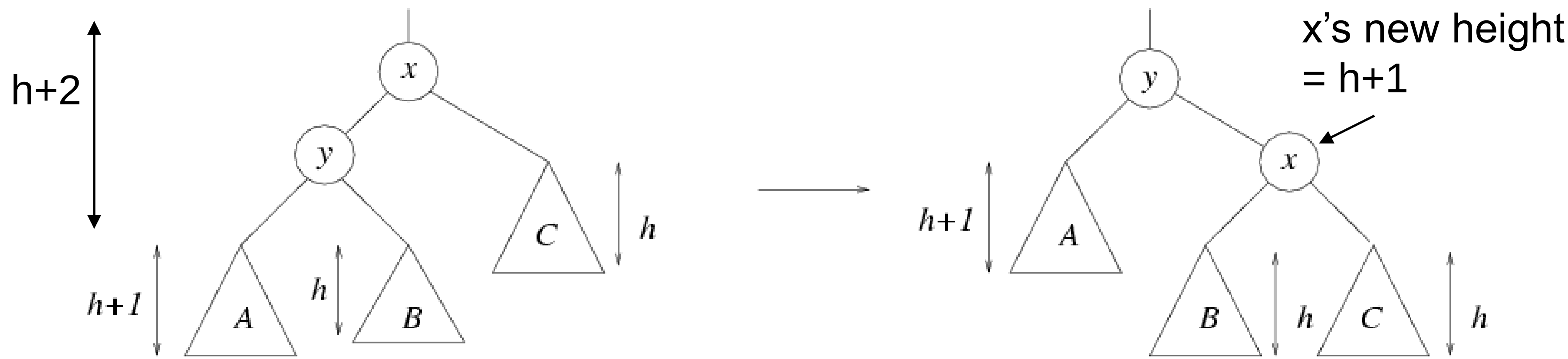
- Two types of rotations
  - Single rotations at the imbalance point
    - two nodes are "rotated"
  - Double rotations at the imbalance point
    - three nodes are "rotated"
- We'll see them first and see when to use them later

The Left-left Single Rotation (Case 1): Height(left(left(x))) = h+1

P: **parent of** x

$P.left/right \leftarrow y$
$x.left \leftarrow y.right$
$y.right \leftarrow x$

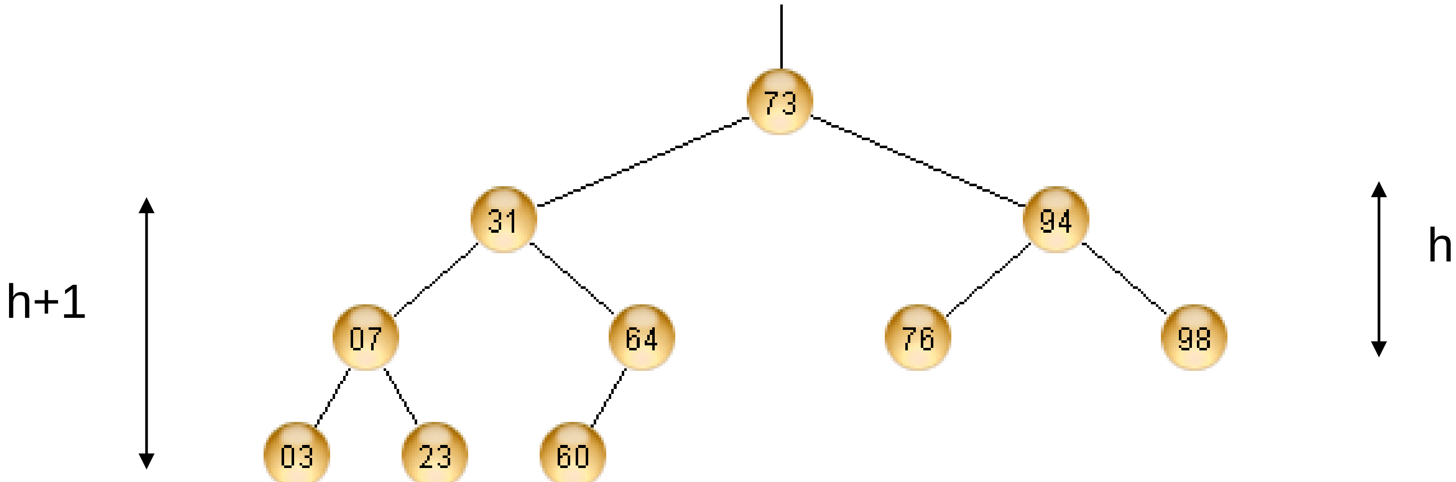
First imbalance point from the leave:
left(x) – right(x) = 2



Rotate **x** with the left child of **y**
(pay attention to the resulting sub-trees positions)

(Note that the height of B can be h+1, in which case x's new height would be h+2)

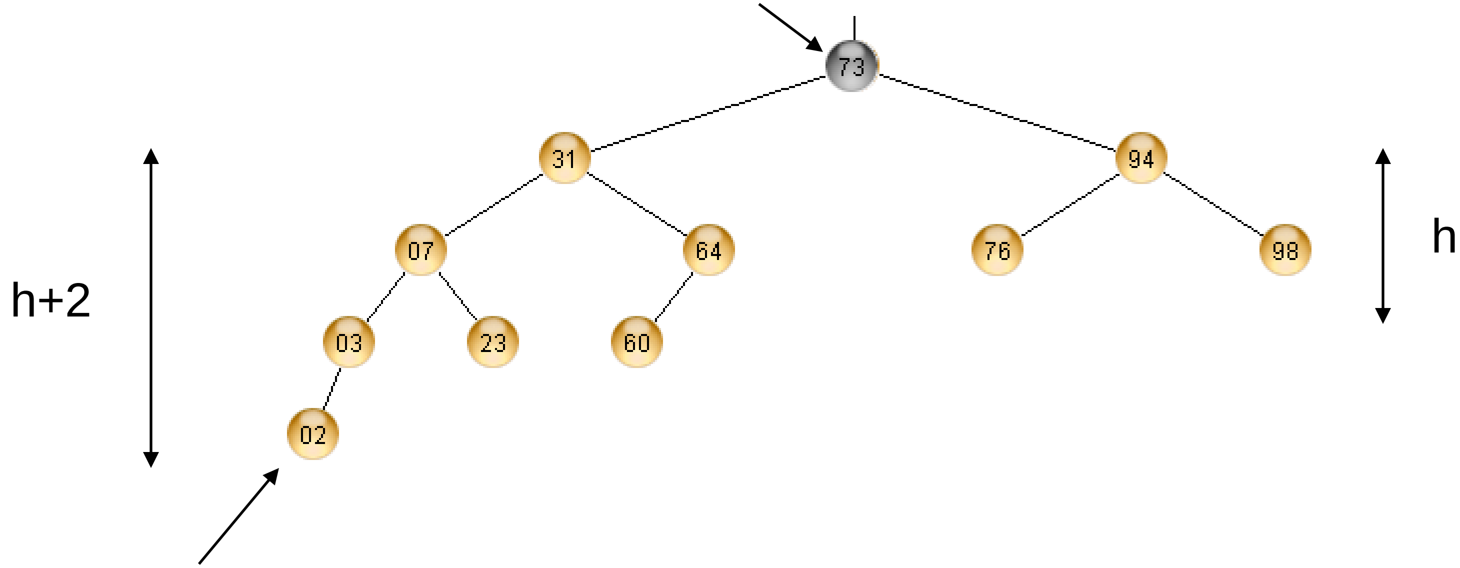
Single Rotation - Example



Tree is an AVL tree by definition.

Add a node 02

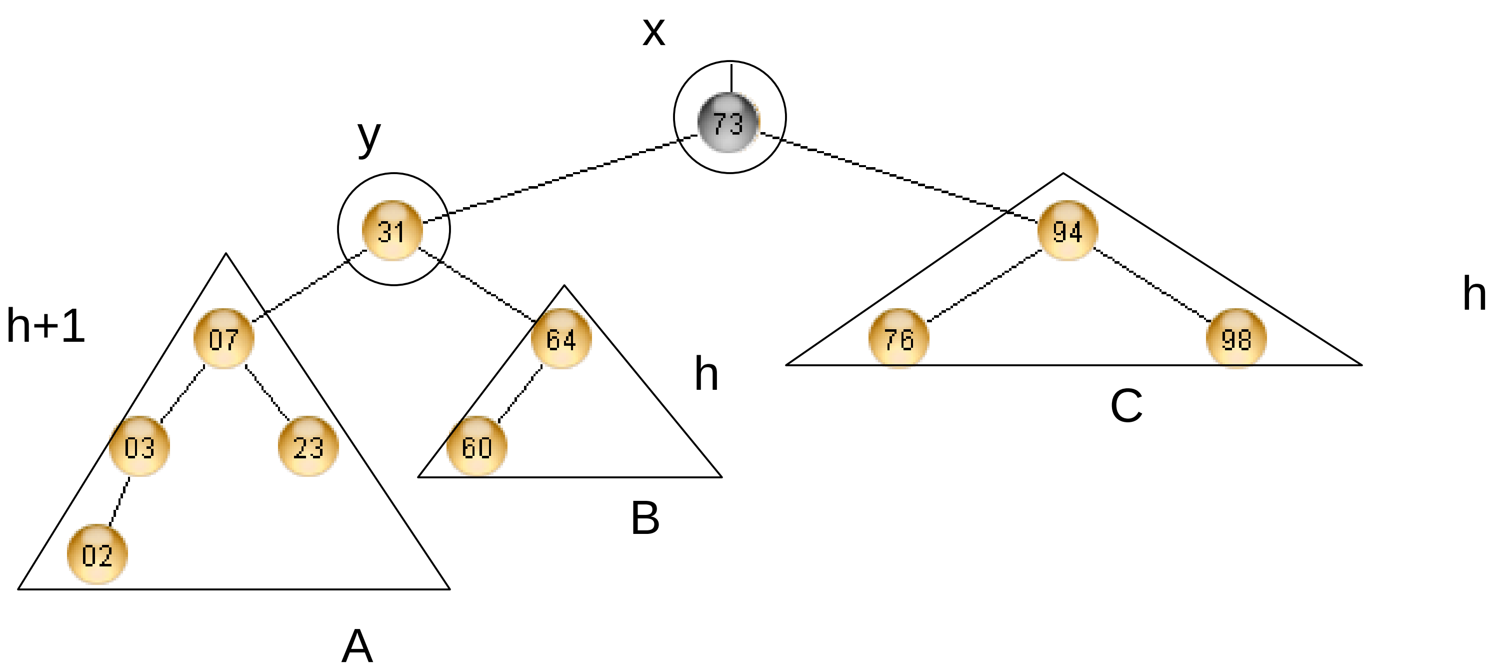
First imbalance point from the leave



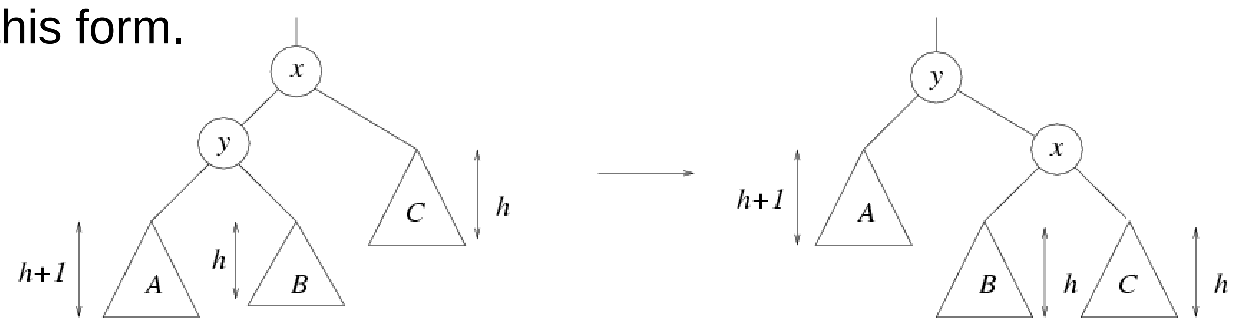
Node 02 added

Tree violates the AVL definition!
Perform rotation.

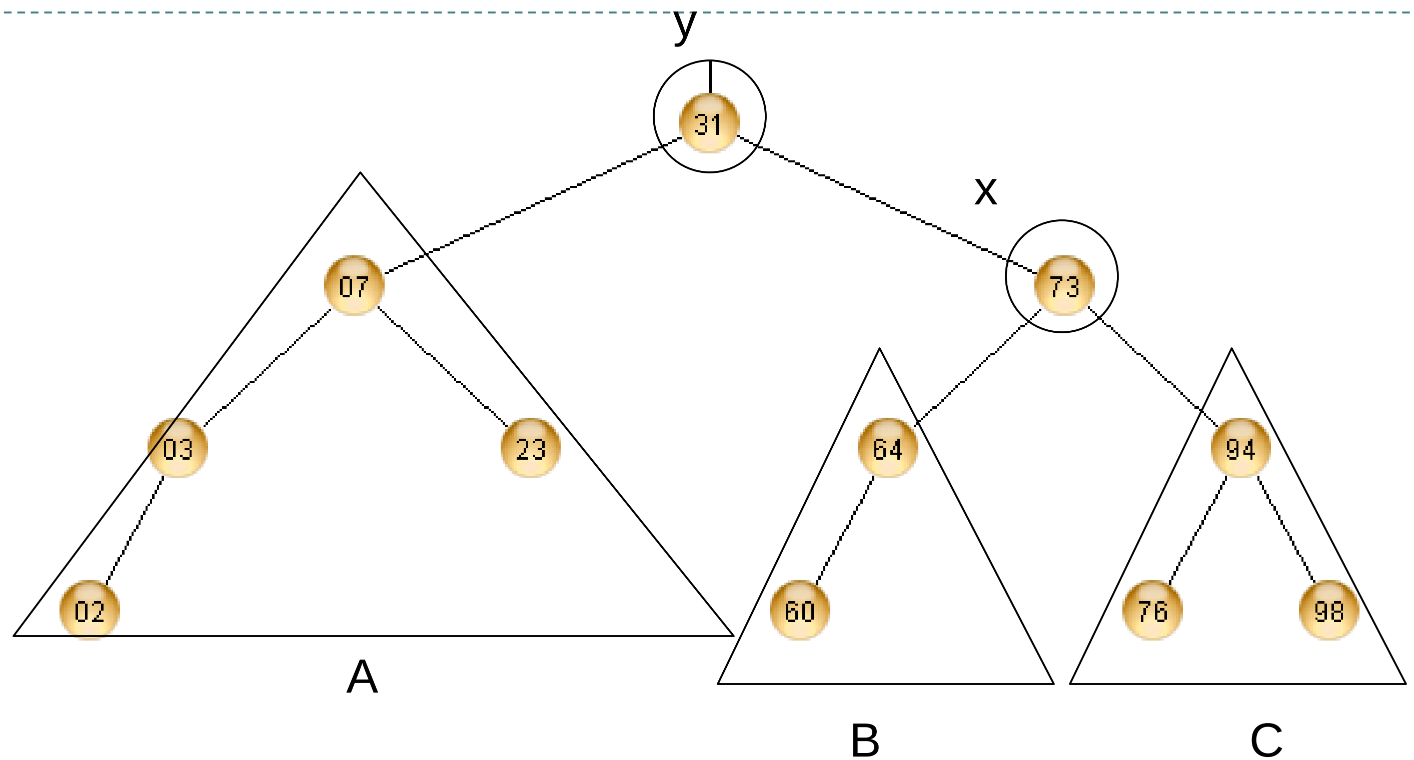
Example



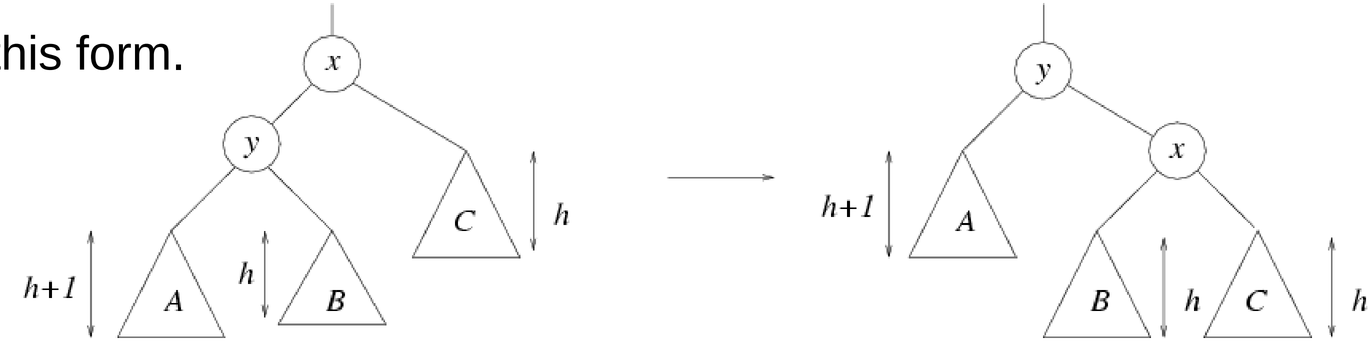
Tree has this form.



Example – After Rotation

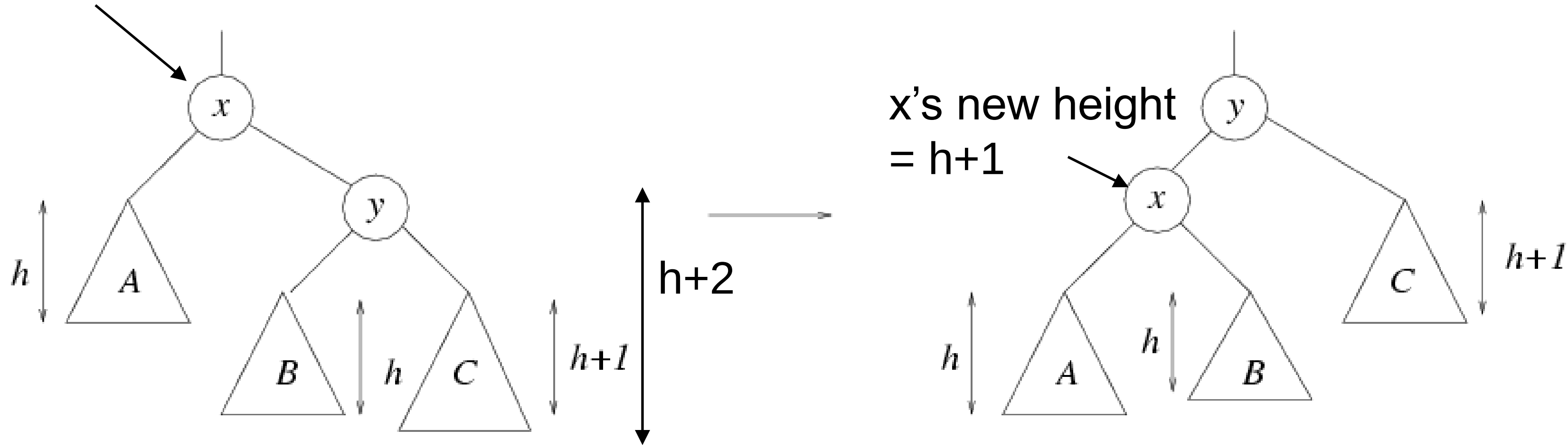


Tree has this form.



Right-Right Single Rotation (Case 2): Height(right(right(x))) = h+1

AVL property is first violated at x from the leave:
left(x) – right(x) = -2

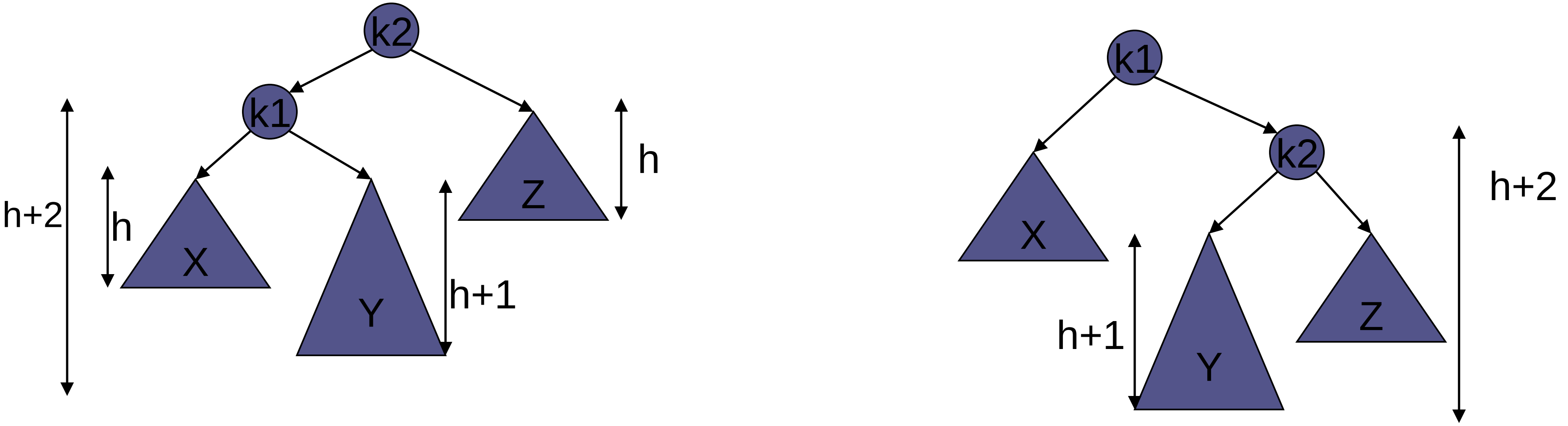


Rotate **x** with the right child of **y**
(pay attention to the resulting sub-trees positions)

(Note that the height of B can be h+1, in which case x's new height would be h+2)

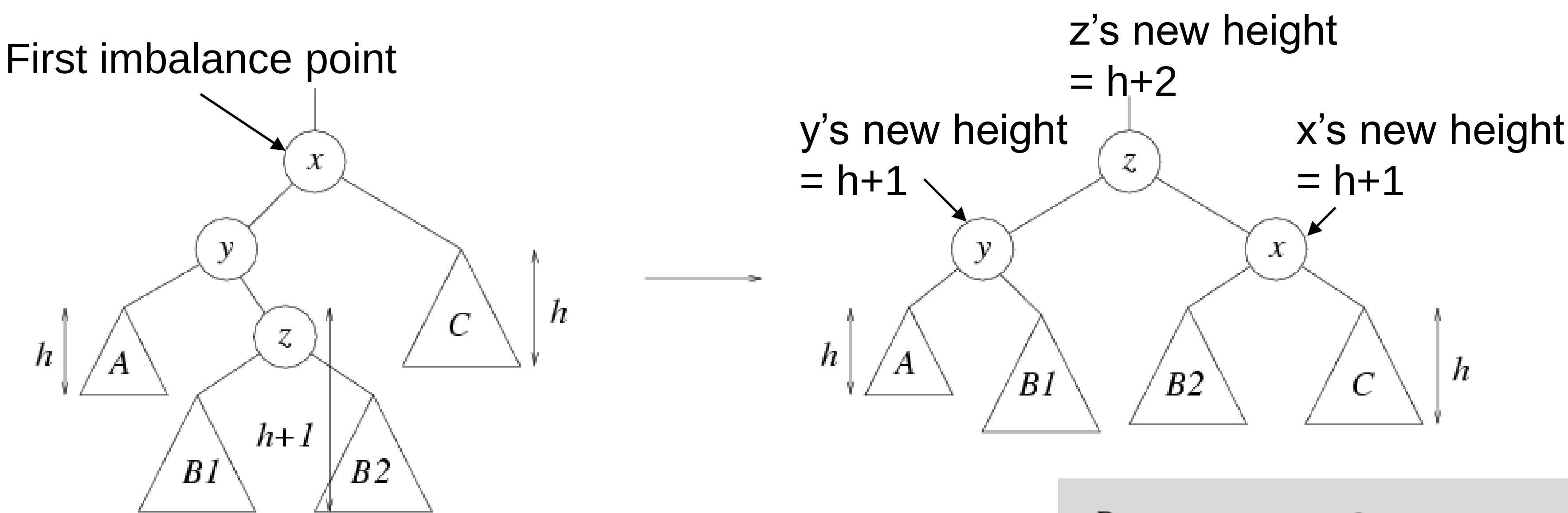
Single Rotation

- Sometimes a single rotation fails to solve the problem



- In such cases, we need to use a double-rotation

Right-Left Double Rotations (Case 3): Height(right(left(x))) = h+1



Double-rotate **x** with its left child **y** **<u>and</u> y's** right child **z**
(pay attention to the resulting sub-trees positions)

Involves 2 single rotations on z:

1. Single rotate z upwards with y, pushing y down
2. Single rotate z upwards again with x, pushing x down in another branch

P: **parent of** x

$P.left/right \leftarrow z$

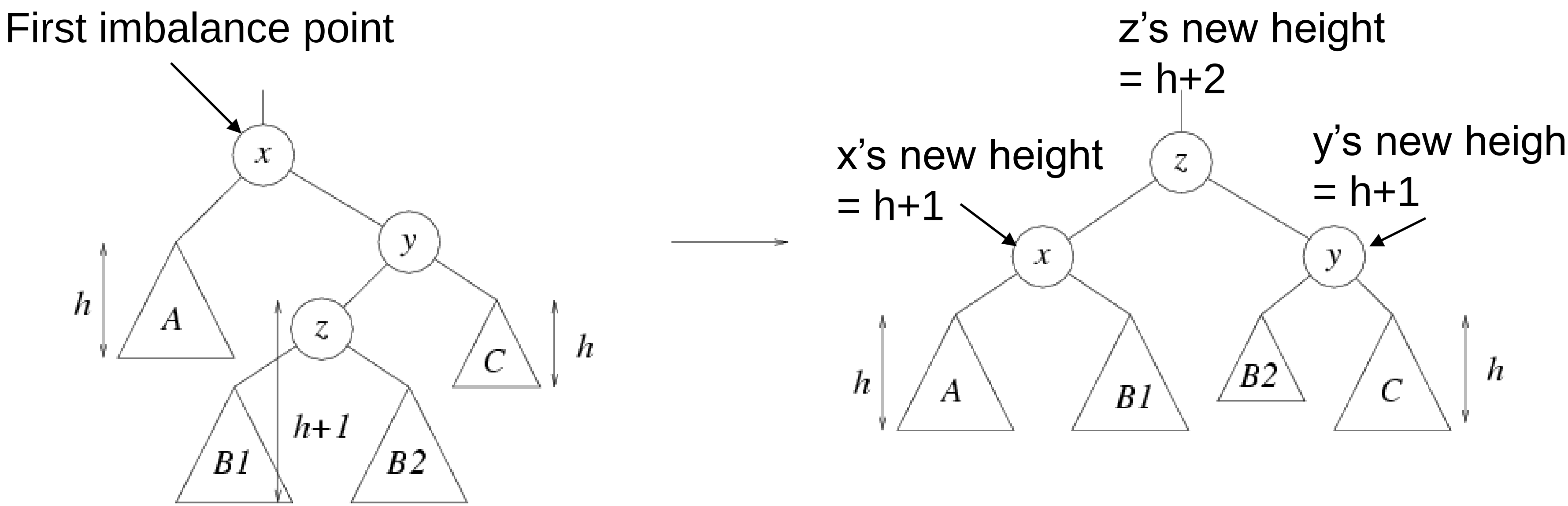
$x.left \leftarrow y.right$

$y.right \leftarrow x.left$

$z.left \leftarrow y$

$z.right \leftarrow x$

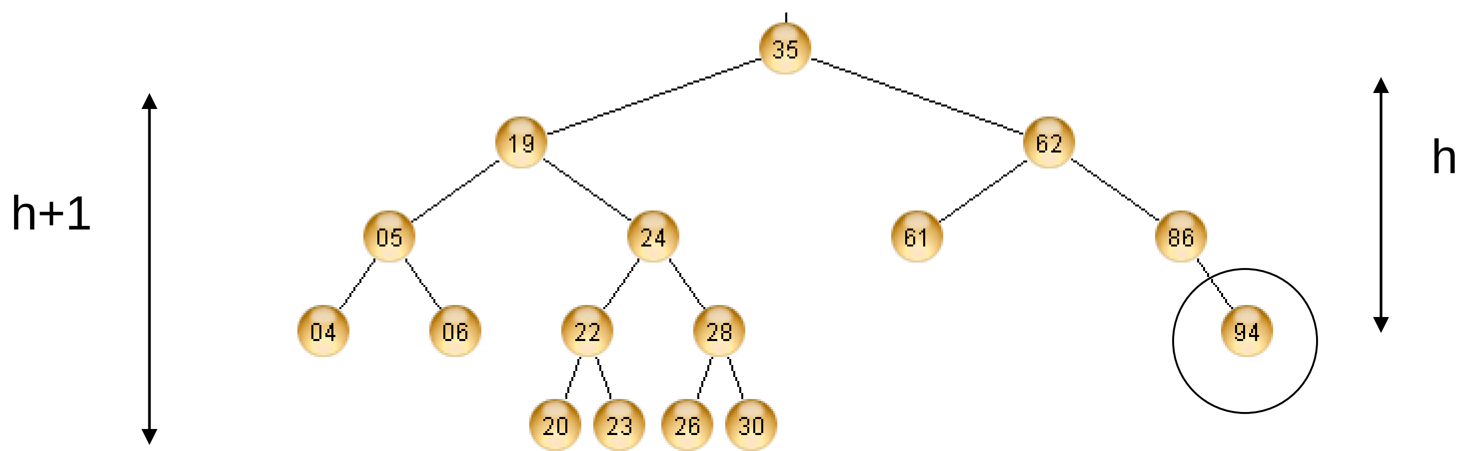
Double Rotations: Case 4
Height(left(right(x))) = h+1



Double-rotate **x** with its right child **y <u>and</u> y's** left child **z**
(pay attention to the resulting sub-trees positions)

Involves 2 single rotations on z (similar as before):
Single rotate z upwards with y, pushing y down
Single rotate z upwards again with x, pushing x down in another branch

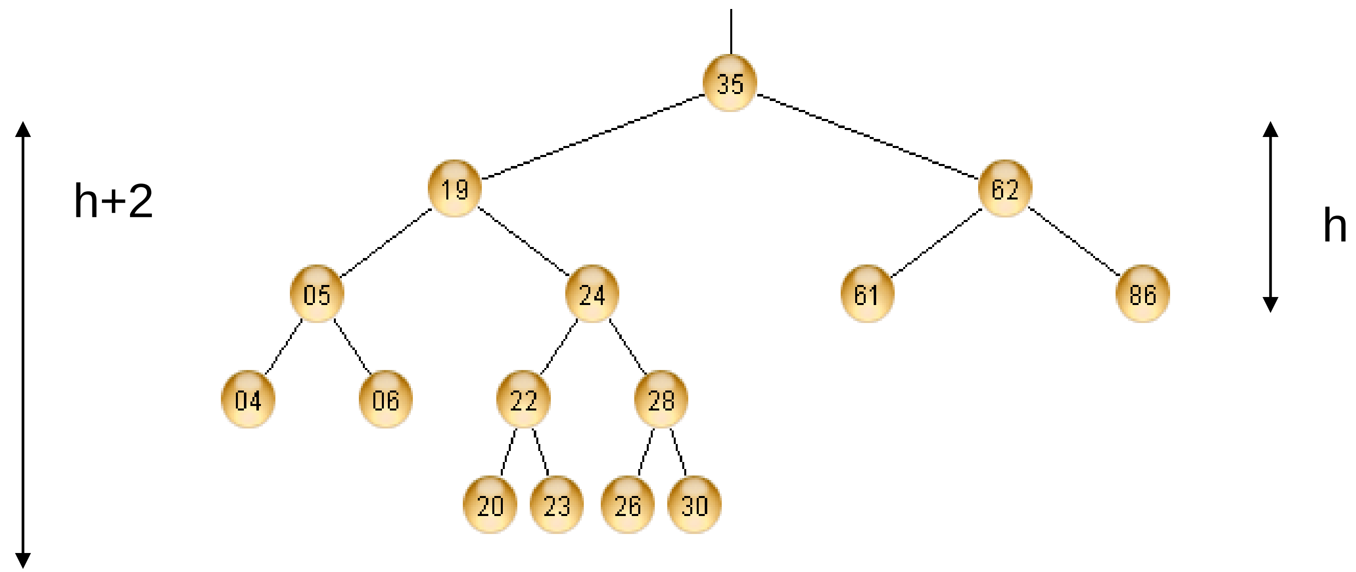
Double Rotation - Example



Delete node 94

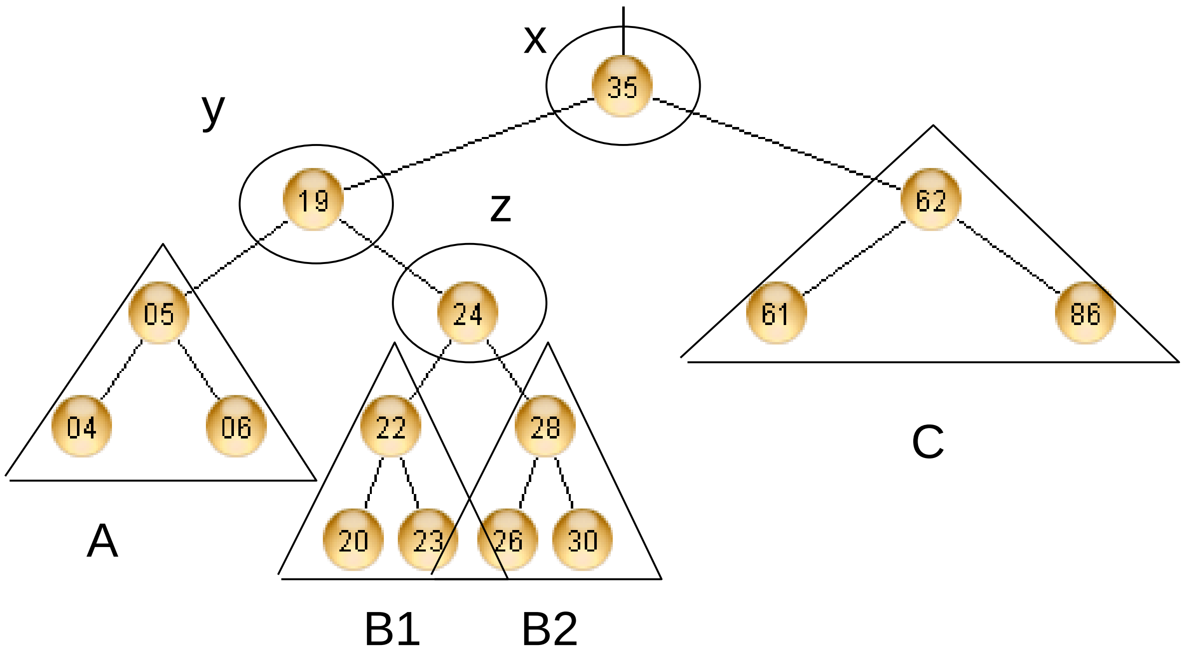
Tree is an AVL tree by definition.

Example

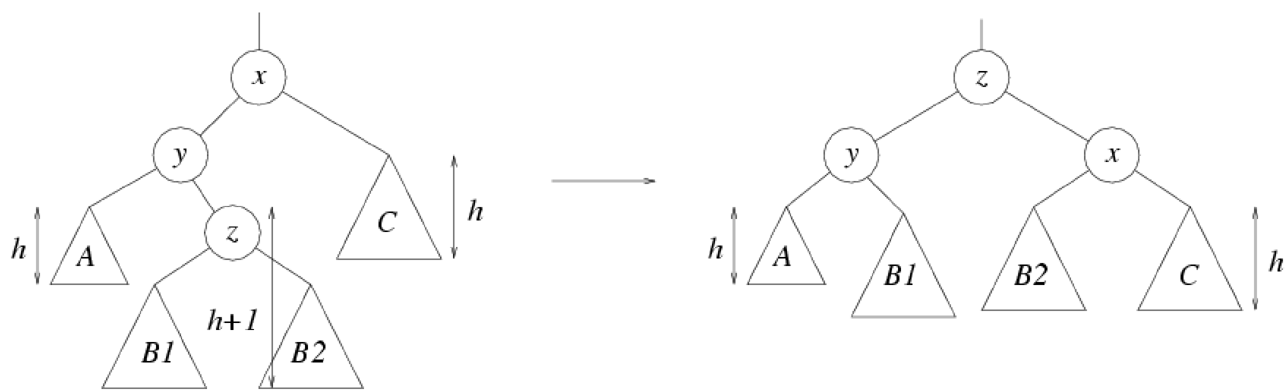


AVL tree property is violated.

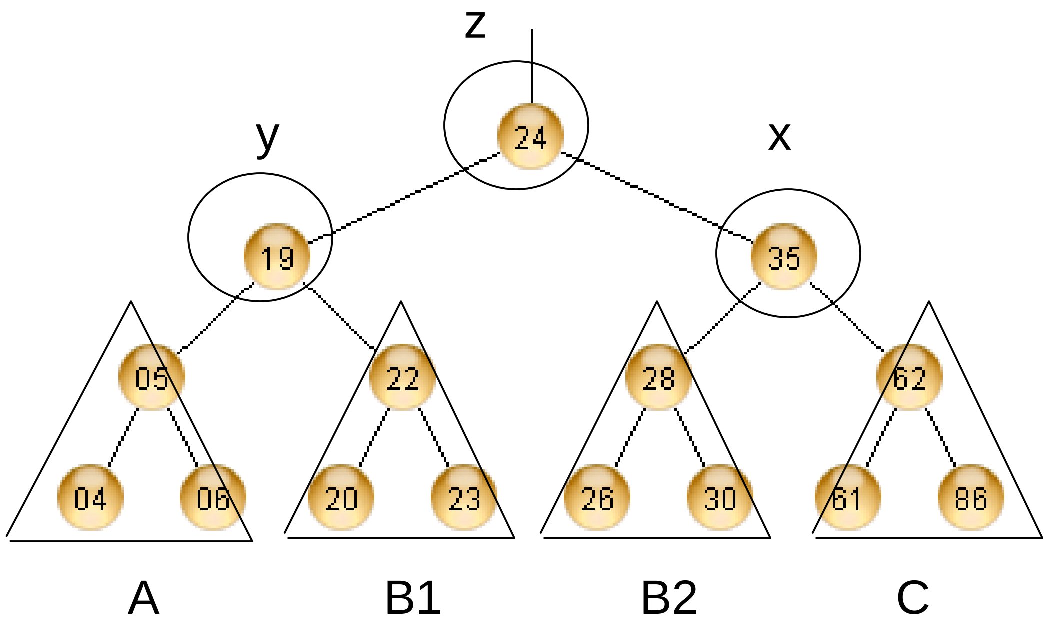
Example



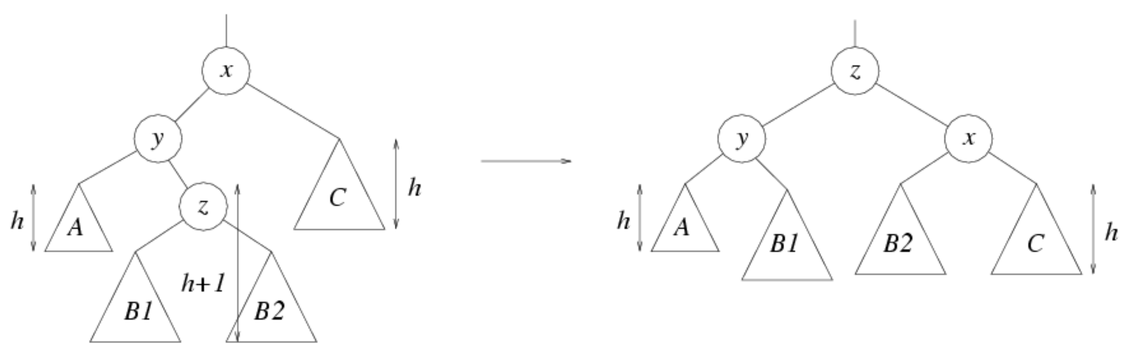
Tree has this form.



After Double Rotation



Tree has this form



Insertion

Part 1. Perform normal BST insertion

Part 2. Check and correct AVL properties

Trace back on the path from the inserted leaf all the way towards the root:

- Check to see if heights of left(x) and right(x) differ at most by 1
- If not, we know x is the imbalance point (the height of x is h+3)
  - If left(x) is higher (h+2), then
    - If left(left(x)) is of height h+1, we single rotate with x's left child, i.e., left(x) (case 1)
    - Otherwise [right(left(x)) is higher (h+1)] we double rotate with x's left child, i.e., left(x) (case 3)
  - Otherwise, height of right(x) is longer (h+2)
    - If right(right(x)) is of height h+1, then we rotate with x's right child, i.e., right(x) (case 2)
    - Otherwise [left(right(x)) is higher (h+1)] we double rotate with x's right child, i.e., right(x) (case 4)

* Rotations may stop somewhere leading to the root. Remember to make the rotated node the new child of parent(x)

Insertion

- The time complexity to perform a rotation is O(1)
- The time complexity to find a node that violates the AVL property is dependent on the height of the tree (which is log(N))
- The height of a node can be found in O(N) time.
- The height of a node can also be more efficiently stored in a node, and dynamically updated locally each time insertion or deletion occurs. In this way, the height can be accessed in O(1) time.
  - In this case, the insertion takes O(log n) time

Deletion

- Perform normal BST deletion

- Perform exactly the same checking as for insertion to restore the tree property

Note

- There are other variations in the way AVL trees are implemented. These notes present a nice way that treats insertion and deletion the same.

- All implementations have the same idea, detect an "imbalance" in height for a node and perform corrections via single or double rotations.

- Red-black tree (more complicated, but more efficient in terms of space; see textbook)

Summary AVL Trees

- Maintains a balanced tree
- Modifies the insertion and deletion routine
  - Performs single or double rotations to restore structure
  - Requires a little more work for insertion and deletion
  - But, since trees are mostly used for searching
    - More work for insert and delete is worth the performance gain for searching
- Guarantees that the height of the tree is O(logn)
  - The guarantee directly implies that functions find(), min(), and max() will be performed in O(logn)