Lec 11 Feb 17

Mid-term 1: Feb 24 or 26?

- Topics: (Chapter 4 of text)
  - binary Trees
  - expression trees
  - Binary Search Trees

Trees

- dictionary operations
  - Search, insert and delete
  - Does there exist a simple data structure for which the running time of dictionary operations (search, insert, delete) is O(log N) where N = total number of keys?
  - Arrays, linked lists, (sorted or unsorted), hash tables, heaps – none of them can do it.

- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations

Trees

- A tree is a collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *subtrees* T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge* from r

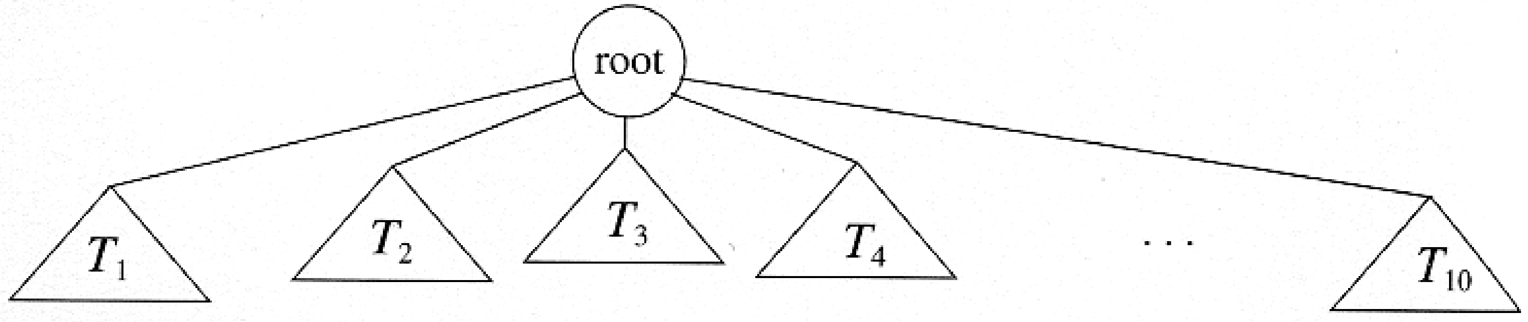


Figure 4.1 Generic tree

Basic terms

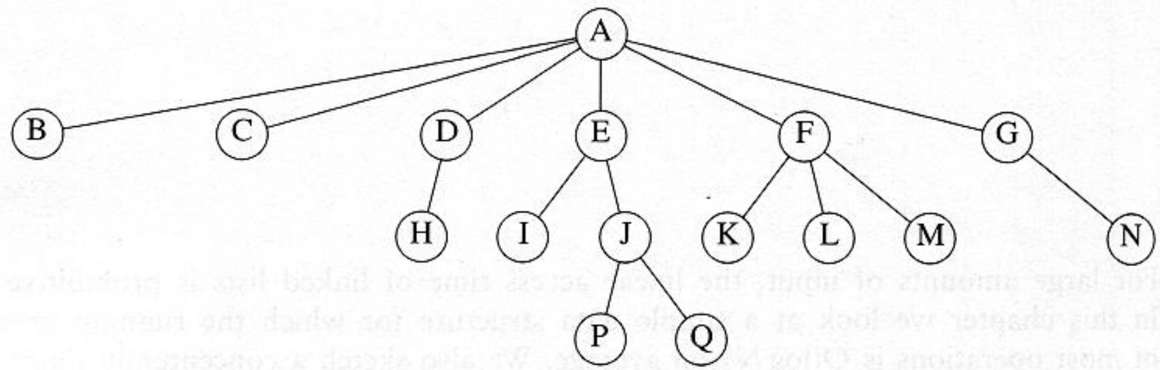


Figure 4.2 A tree

- *Child* and *Parent*
  - Every node except the root has one parent
  - A node can have an zero or more children
- *Leaves*
  - Leaves are nodes with no children
- *Sibling*
  - nodes with same parent

Implementing a tree

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

Figure 4.3 Node declarations for trees

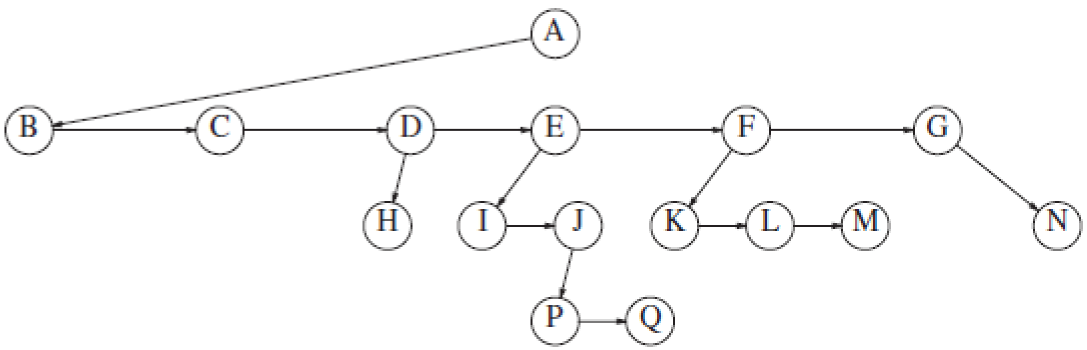


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

More Terms

- *Path*
  - A sequence of edges
- *Length of a path*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node

More Terms

- *Height* of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree = the height of the root

  = the depth of the deepest leaf
- *Ancestor* and *descendant*
  - If there is a path from n_1 to n_2
    - n_1 is an ancestor of n_2, n_2 is a descendant of n_1
    - *Proper ancestor* and *proper descendant*

Example: UNIX Directory

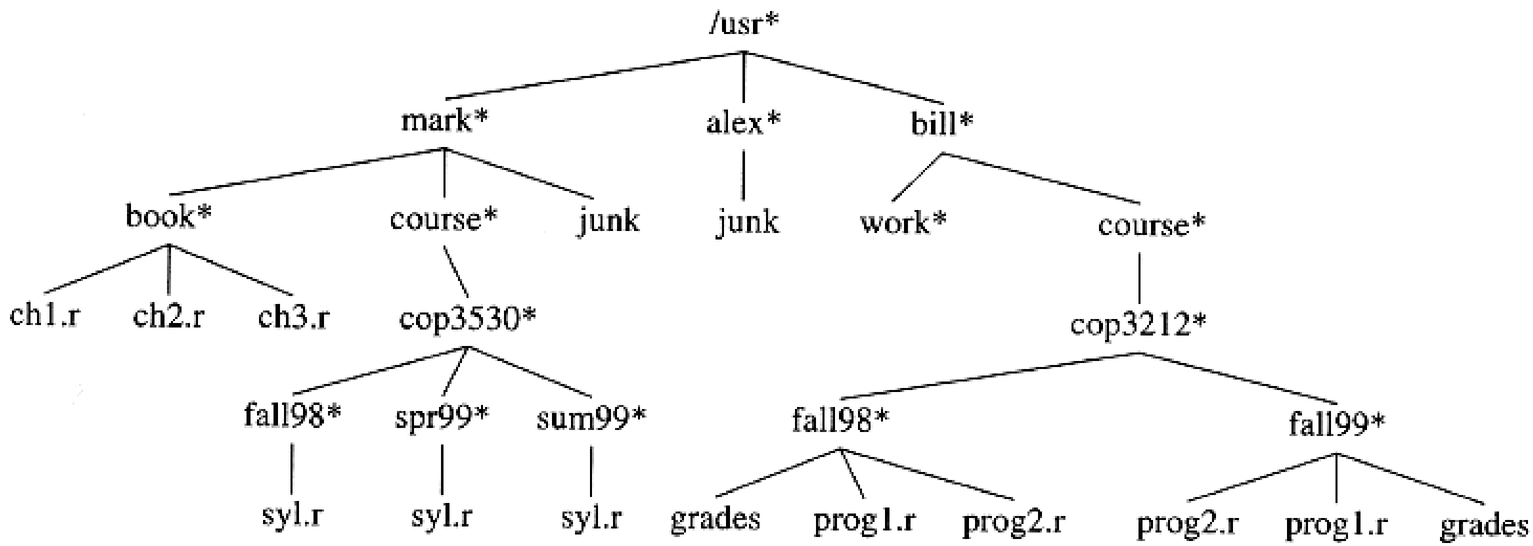


Figure 4.5 UNIX directory

Example: Expression Trees

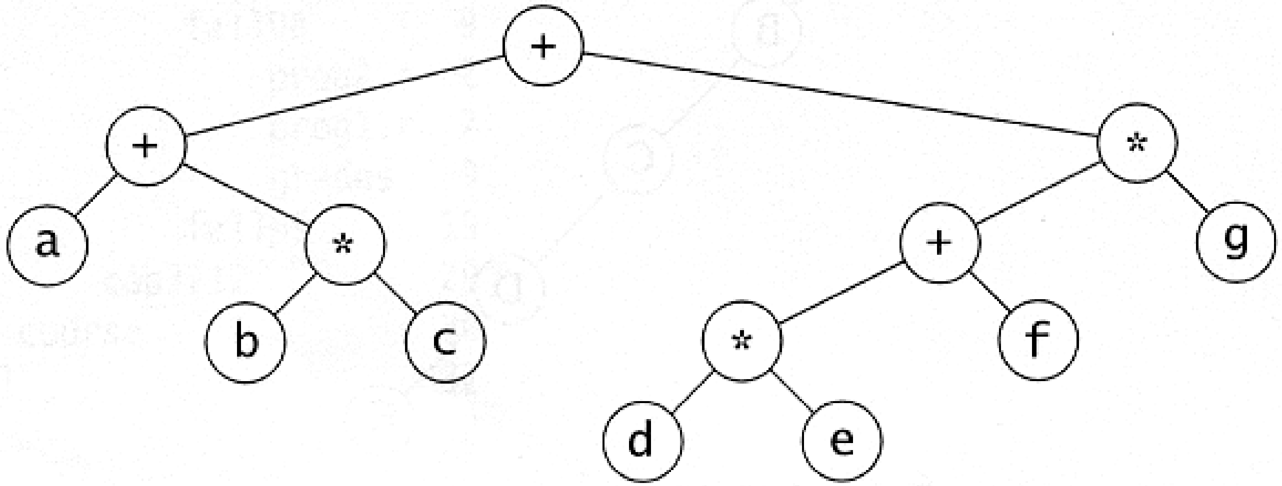


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary (e.g. unary minus)

Expression Tree application

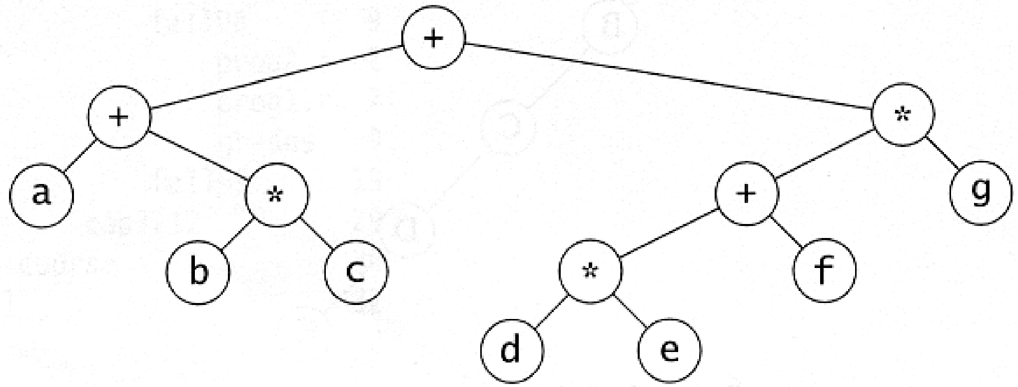


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Given an expression, build the tree
- Compilers build expression trees when parsing an expression that occurs in a program
- Applications:
  - Common subexpression elimination.

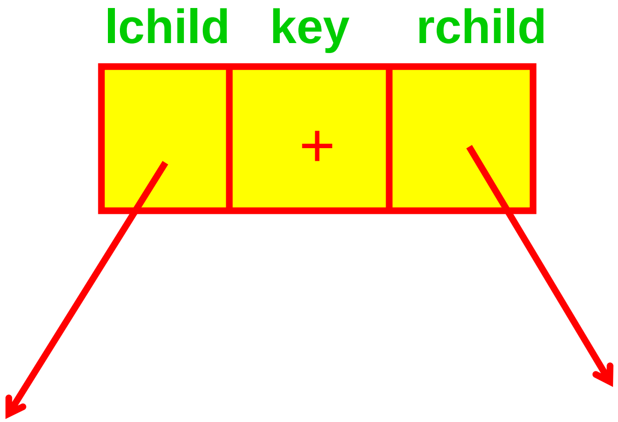
Expression to expression Tree algorithm

Problem: Given an expression, build the tree.

Solution: recall the stack based algorithm for converting infix to postfix expression.

From postfix expression E, we can build an expression tree T.

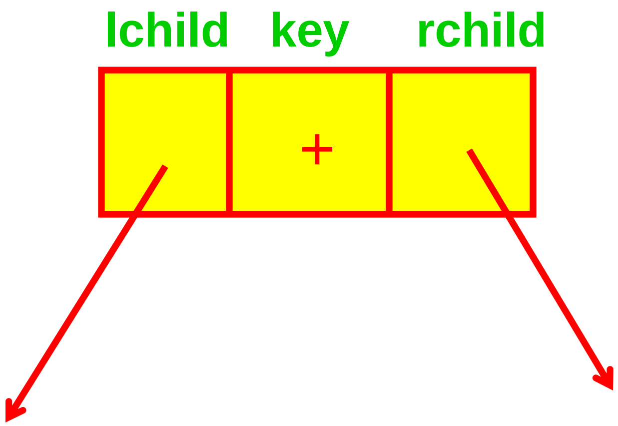
Node structure




```
class Tree {
   char key;
   Tree* lchild, rchild;
    . . .

}
```

Expression to expression Tree algorithm

Node structure



Operand: leaf node

Operator: internal node

Constructor:

```
Tree(char ch, Tree* lft,
   Tree* rgt) {
  key = ch;
  lchild = lft;
  rchild = rgt;
}
```

Expression to expression Tree algorithm

Problem: Given an expression, build the tree.

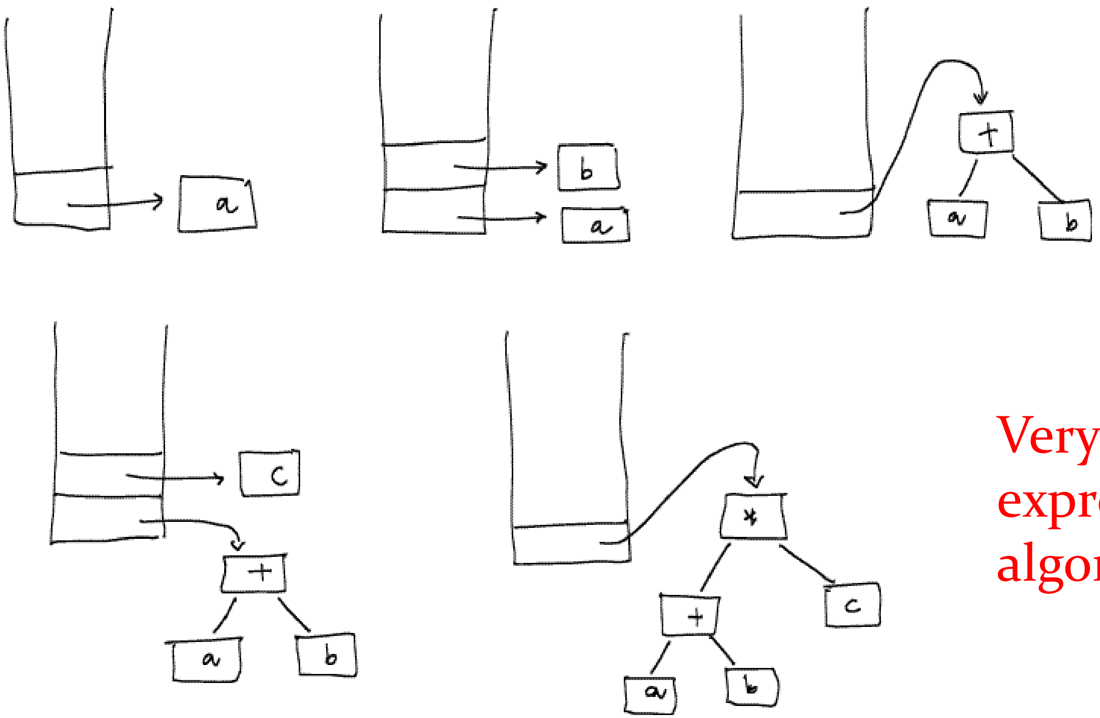
Input: Postfix expression E, output: Expression tree T

```
 initialize stack S;
 for j = 0 to E.size – 1 do
   if (E[j] is an operand) {
    Tree t = new Tree(E[j]);
    S.push(t*);}
   else {
     tree* t1 = S.pop();
     tree* t2 = S.pop();
     Tree t = new(E[j], t1, t2);
     S.push(t*);
   }
```

At the end, stack contains a single tree pointer, which is the pointer to the expression tree.

Expression to expression Tree algorithm

Example: a b + c *



Very similar to prefix expression evaluation algorithm

Tree Traversal

- used to print out the data in a tree in a certain order

- Pre-order traversal
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree

Preorder, Postorder and Inorder

- Preorder traversal
  - node, left, right
  - prefix expression
    - ++a*bc*+*defg

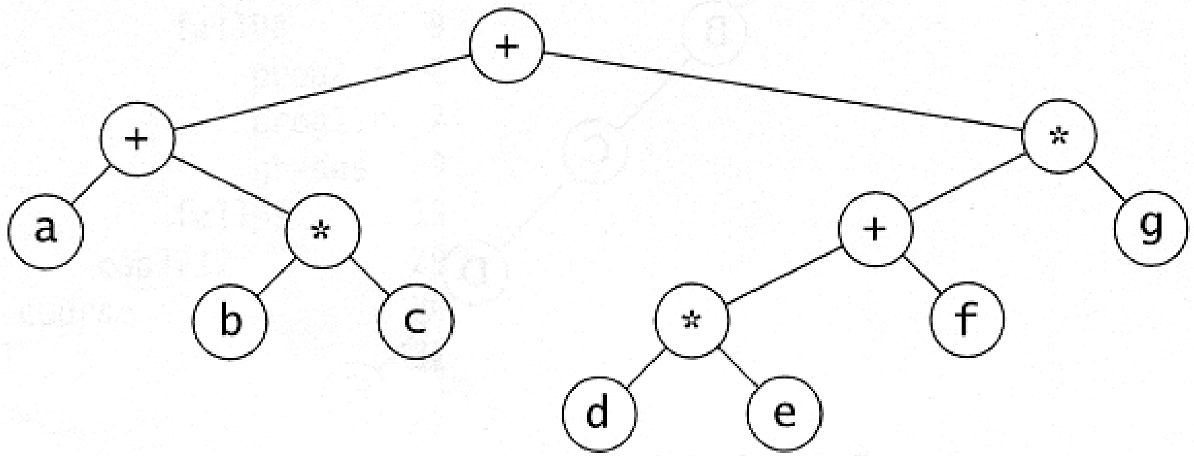


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder, Postorder and Inorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+
- Inorder traversal
  - left, node, right
  - infix expression
    - a+b*c+d*e+f*g

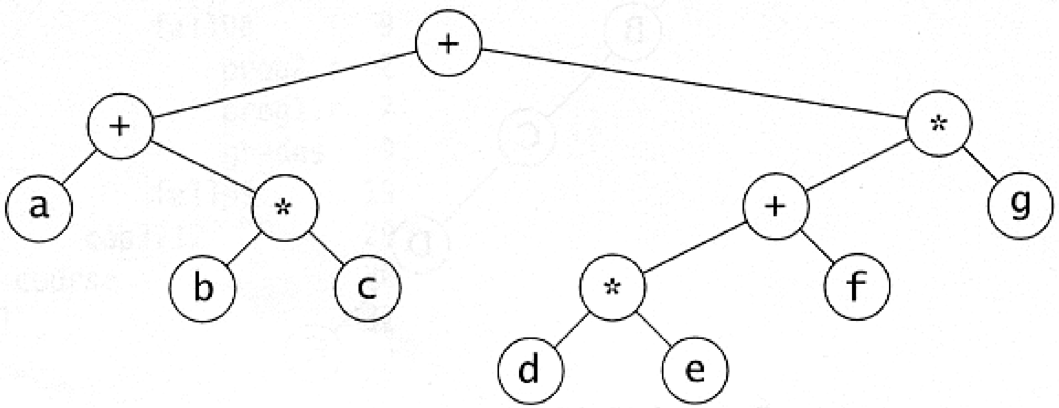


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Example: Unix Directory Traversal

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall98
                    syl.r
                spr99
                    syl.r
                sum99
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall98
                    grades
                    prog1.r
                    prog2.r
                fall99
                    prog2.r
                    prog1.r
                    grades
```

```
            ch1.r            3
            ch2.r            2
            ch3.r            4
        book                10
                    syl.r    1
                fall98       2
                    syl.r    5
                spr99        6
                    syl.r    2
                sum99        3
            cop3530         12
        course              13
        junk                 6
    mark                    30
        junk                 8
    alex                     9
        work                 1
                    grades   3
                    prog1.r  4
                    prog2.r  1
                fall98       9
                    prog2.r  2
                    prog1.r  7
                    grades   9
                fall99      19
            cop3212         29
        course              30
    bill                    32
/usr                        72
```

Recursive algorithm to print all nodes in a tree

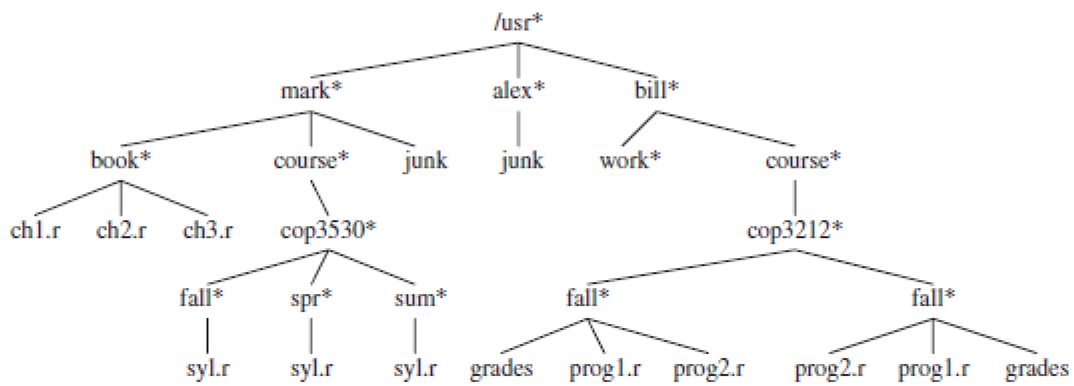


Figure 4.5 UNIX directory

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

Figure 4.6 Pseudocode to list a directory in a hierarchical file system

Preorder, Postorder and Inorder Pseudo Code

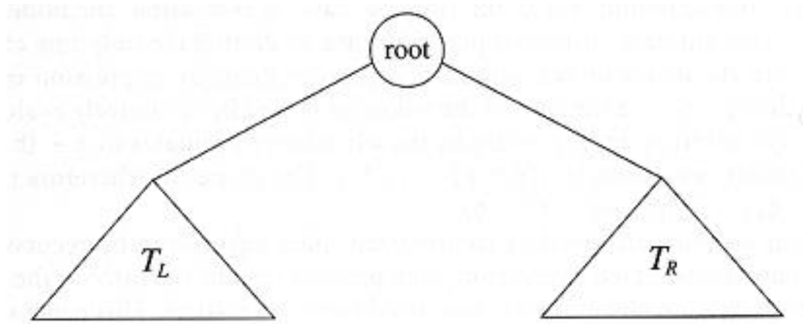
Algorithm *Preorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** output key(x);
3. *Preorder*(left(x));
4. *Preorder*(right(x));

Algorithm *Postorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Algorithm *Inorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Inorder*(left(x));
3. output key(x);
4. *Inorder*(right(x));

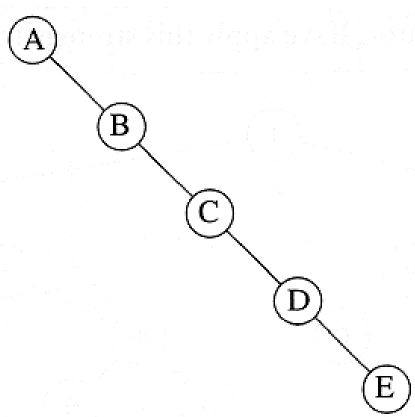
Binary Trees

- A tree in which no node can have more than two children



typical binary tree

- The depth of an “av smaller than N, even though in the worst case, the depth can be as large as N – 1.



Worst-case binary tree

Node Struct of Binary Tree

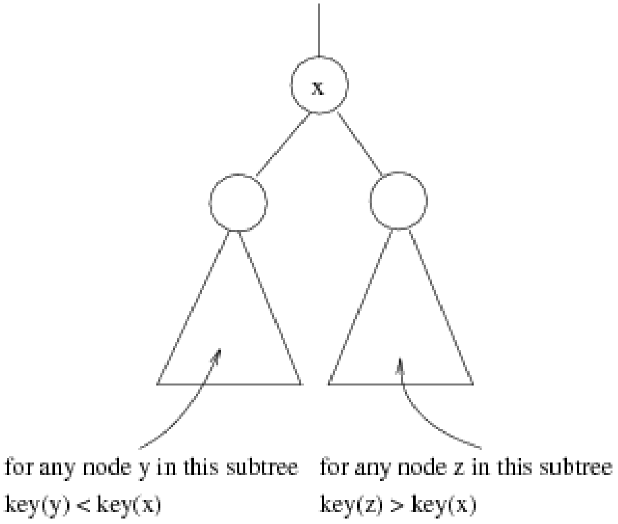
- Possible operations on the Binary Tree ADT
  - Parent, left_child, right_child, sibling, root, etc

- Implementation
  - Because a binary tree has at most two children, we can keep direct pointers to them

```
struct BinaryNode
{
    Object      element;       // The data in the node
    BinaryNode *left;          // Left child
    BinaryNode *right;         // Right child
};
```

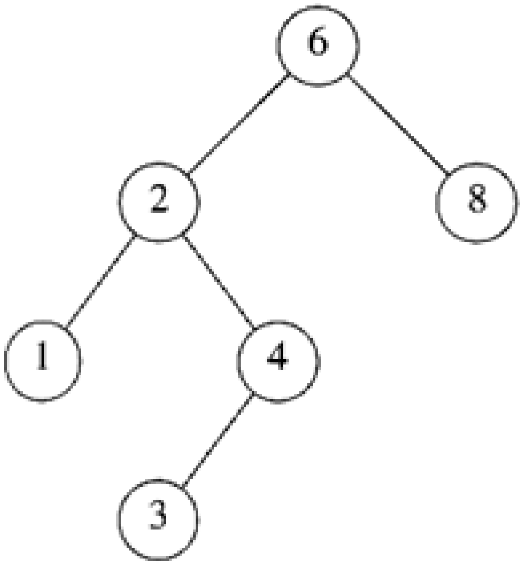
Binary Search Trees (BST)

- A data structure for efficient searching, inser-tion and deletion (dictionary operations)
  - All operations in worst-case O(log n) time
- Binary search tree property
  - For every node x:
  - All the keys in its left subtree are smaller than the key value in x
  - All the keys in its right subtree are larger than the key value in x



Binary Search Trees

Example:

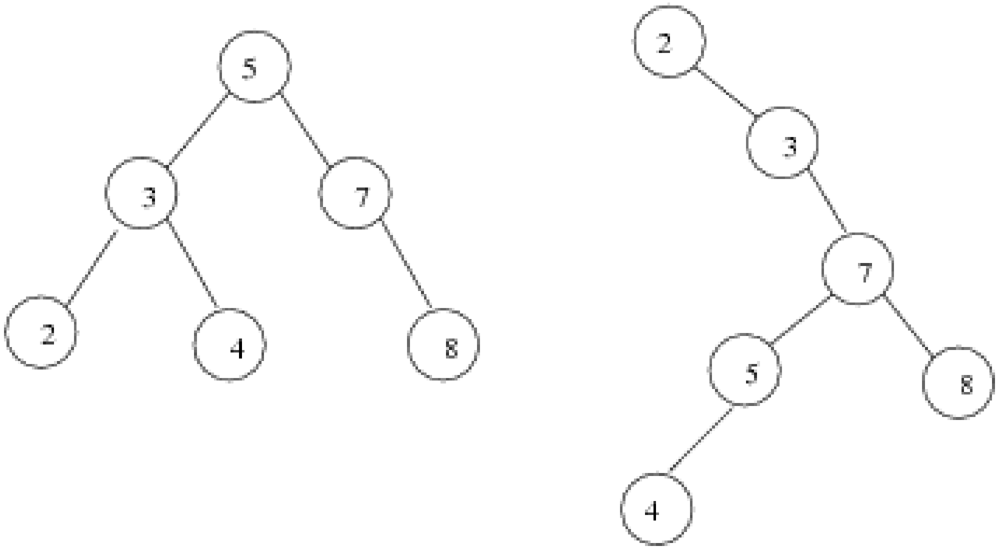


Tree height = 4

Key requirement of a BST: all the keys in a BST are distinct, no duplication

Binary Search Trees

The same set of keys may have different BSTs



- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

(N = the number of nodes in the tree)

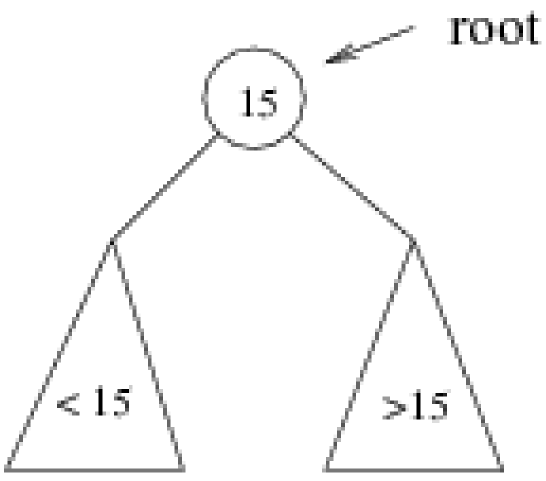
Binary search tree class

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

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ostream & out = cout ) const;

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

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

  private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ theElement }, left{ lt }, right{ rt } { }

        BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
    };

    BinaryNode *root;

    void insert( const Comparable & x, BinaryNode * & t );
    void insert( Comparable && x, BinaryNode * & t );
    void remove( const Comparable & x, BinaryNode * & t );
    BinaryNode * findMin( BinaryNode *t ) const;
    BinaryNode * findMax( BinaryNode *t ) const;
    bool contains( const Comparable & x, BinaryNode *t ) const;
    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode *t, ostream & out ) const;
    BinaryNode * clone( BinaryNode *t ) const;
};
```

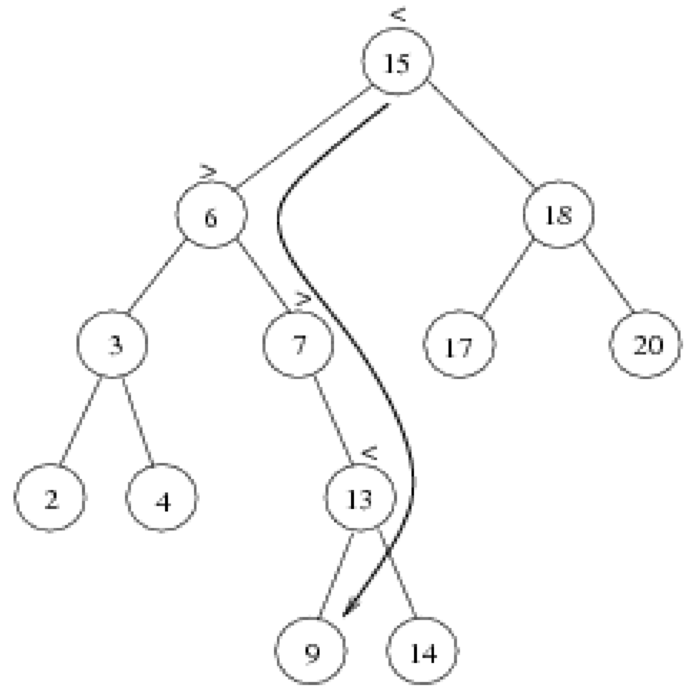
Searching BST

Example: Suppose T is the tree being searched:

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



Example: Search for 9 ...



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!

Search (contains)

- contains (x, t) : return a pointer to the node that has key x in tree rooted at t, or NULL if there is no such node

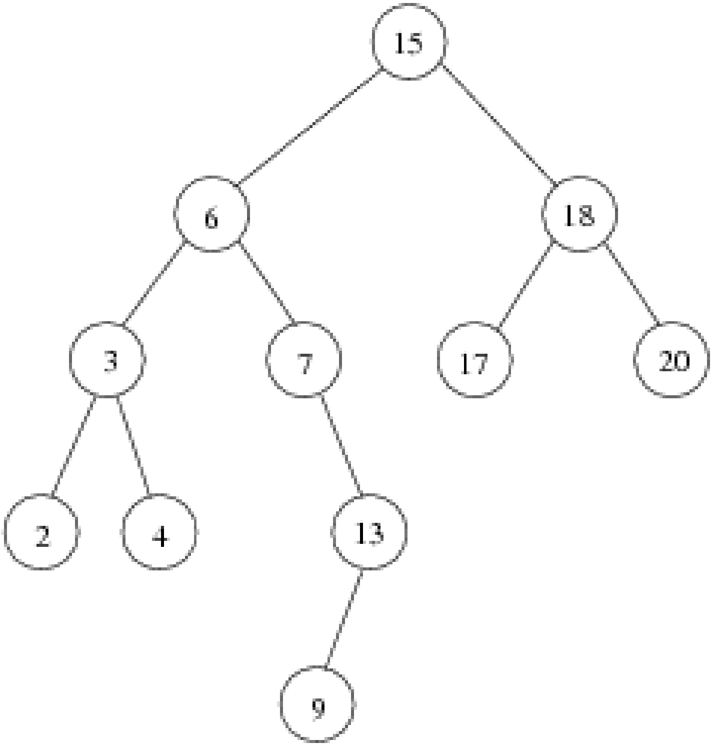
```
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */
bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
```

Figure 4.18 contains operation for binary search trees

- Time complexity: O(height of the tree)

Inorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

findMin/ findMax

- Goal: return the node containing the smallest (largest) key in the tree
- Algorithm: Start at the root and go left (right) as long as there is a left (right) child. The stopping point is the smallest (largest) element

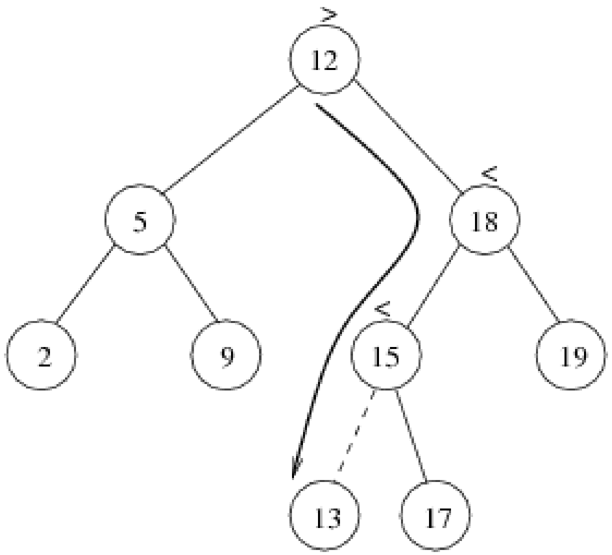
```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

- Time complexity = O(height of the tree)

Insertion

To insert(X):

- Proceed down the tree as you would for search.
- If x is found, do nothing (or update some secondary record)
- Otherwise, insert X at the last spot on the path traversed

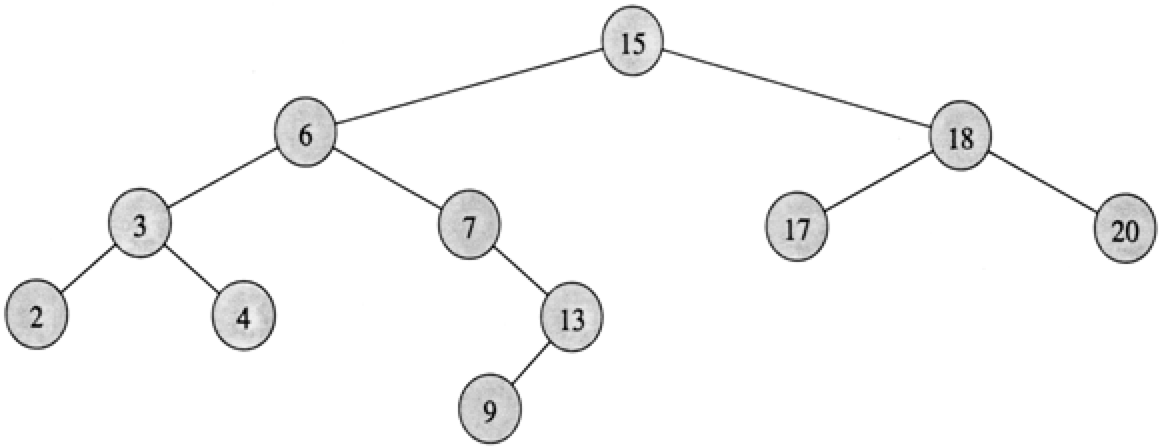


X = 13

- Time complex)

Another example of insertion

Example: insert(11). Show the path taken and the position at which 11 is inserted.



Note: There is a unique place where a new key can be inserted.

Code for insertion (from text)

Insert is a recursive (helper) function that takes a pointer to a node and inserts the key in the subtree rooted at that node.

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        t = new BinaryNode( 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
}
```

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

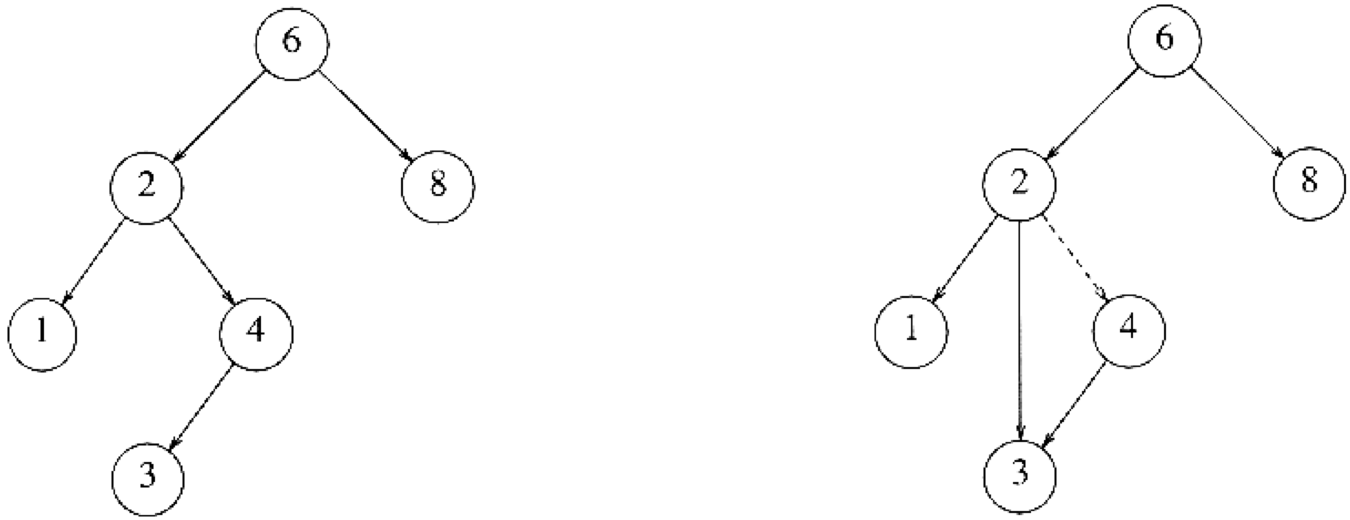


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

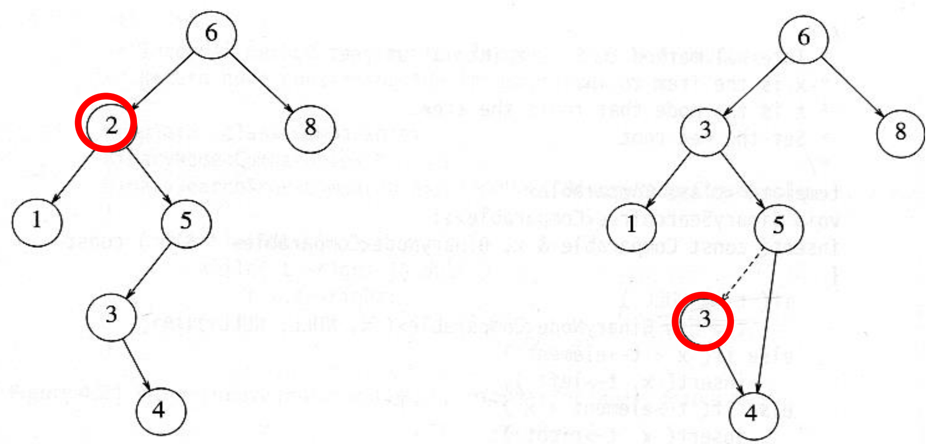


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Code for Deletion

Code for findMin:

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

Code for Deletion

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    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 *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Summary of BST

- all the dictionary operations (search, insert and delete) as well as deleteMin, deleteMax etc. can be performed in O(h) time where h is the height of a binary search tree.

Good news:

- h is on average O(log n) (if the keys are inserted in a random order).
- code for implementing dictionary operations is simple.

Bad news:

- worst-case is O(n).
- some natural order of insertions (sorted in ascending or descending order) lead to O(n) height. (tree keeps growing along one path instead of spreading out.)

Solution:

- **enforce some condition on the structure that keeps the tree from growing unevenly.**