Object-Oriented Programming and Data Structures

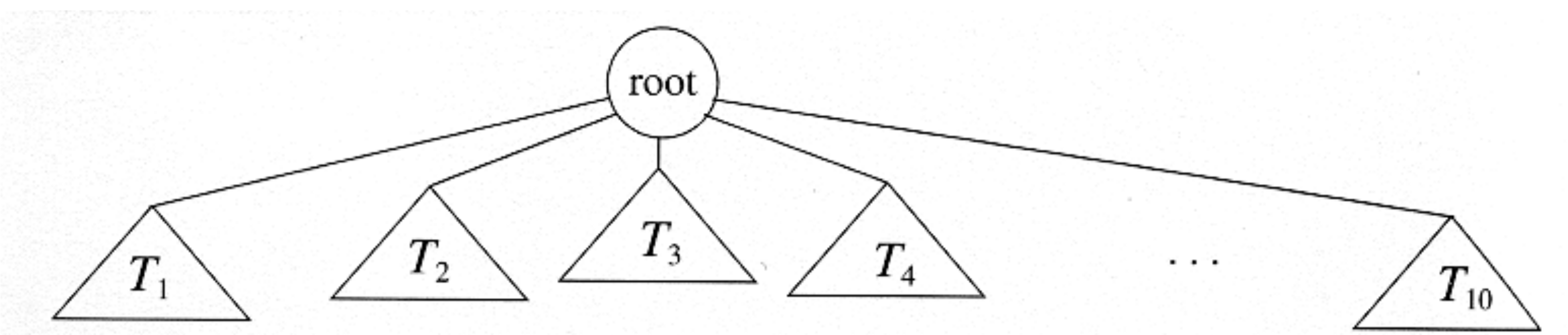
COMP2012: Trees, Binary Trees and Binary Search Trees

Trees

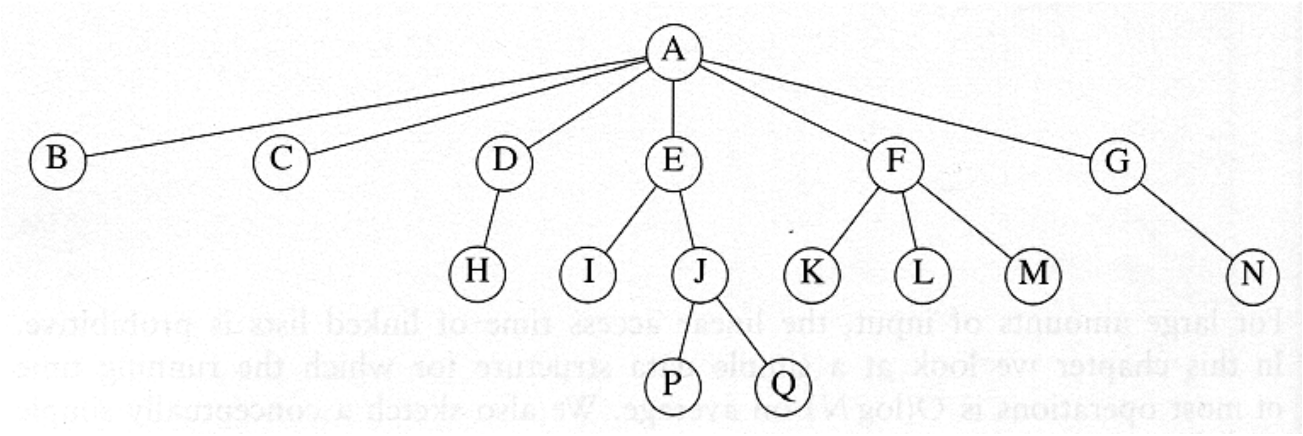
- The linear access time of linked lists is prohibitive for large amount of data.
  - Does there exist any simple data structure for which the average running time of most operations (search, insert, delete) is $O(\log N)$?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations

Recursive Definition of Trees

- A tree T is a collection of nodes
  - (base case) T can be empty
  - (recursive definition) If not empty, a tree T consists of
    - a root node r
    - and zero or more non-empty subtrees T_1, T_2, ..., T_k



Tree Terminologies

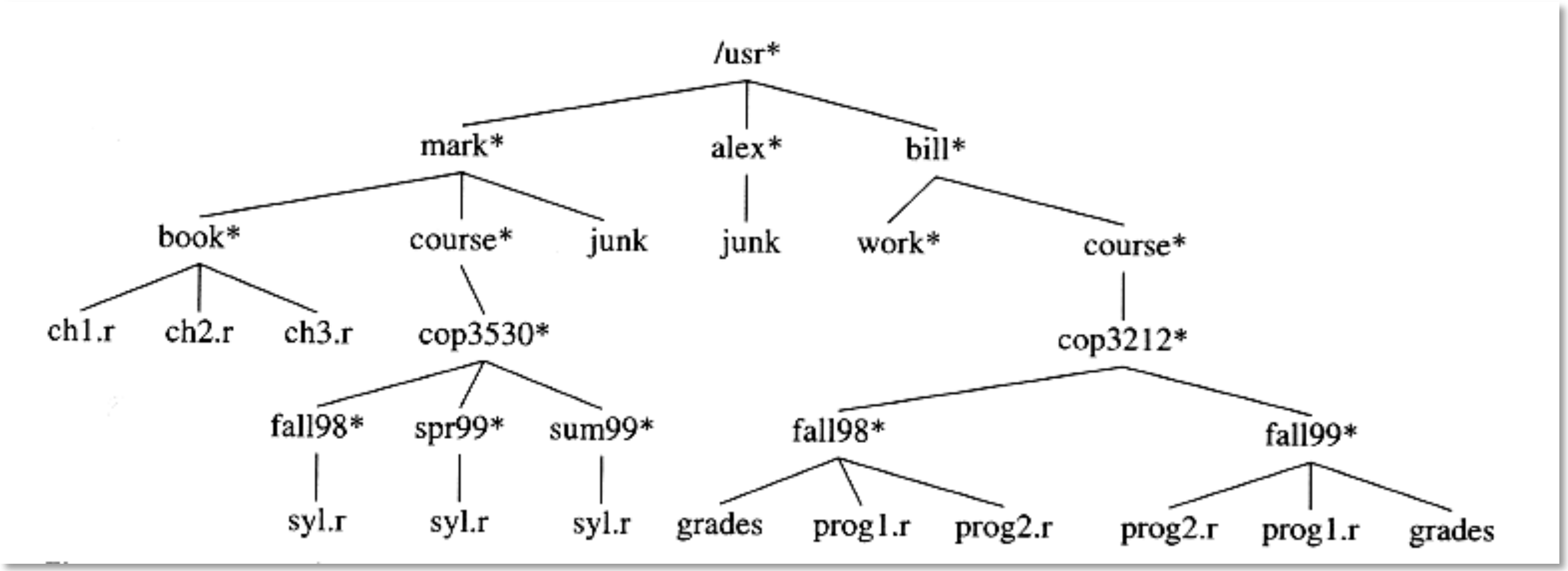


- Child and parent
  - Every node except the root has only one parent
  - A node can have zero or more children
- Leaves: nodes with no children
- Sibling: nodes with same parent
- Path from node n_1 to n_k: a sequence of nodes $n_1, n_2, \ldots, n_k$ such that for $1 \le i \le k$, n_i is the parent of n_{i+1}.

Tree Terminologies ..

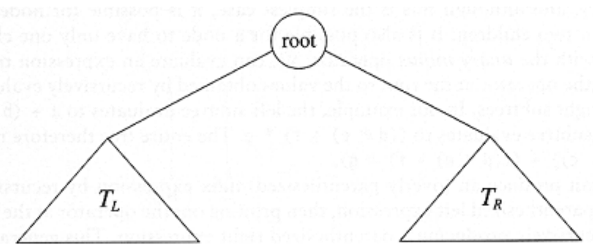
- Length of a path: number of edges on the path
- Depth of a node
  - length of the unique path from the root to that node
- Height of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree
  - = the height of the root
  - = the depth of the 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
  - if $n_1 \neq n_2$, proper ancestor and proper descendant

Example: Unix Directory

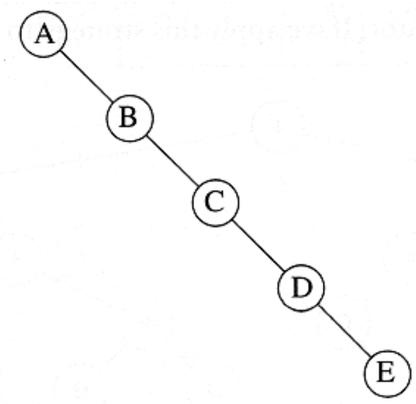


Binary Trees

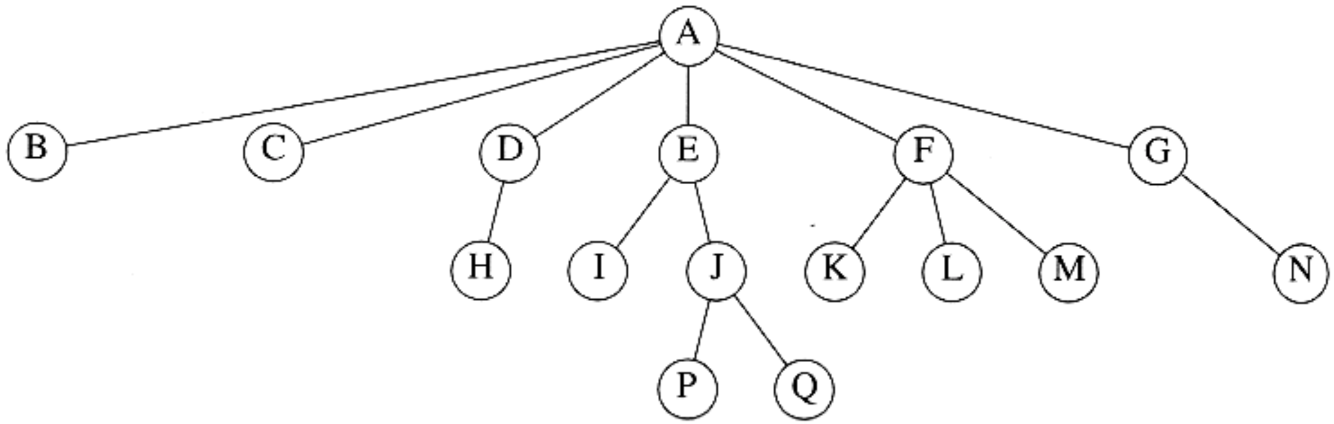
- Generic binary tree: A tree in which no node can have more than two children.



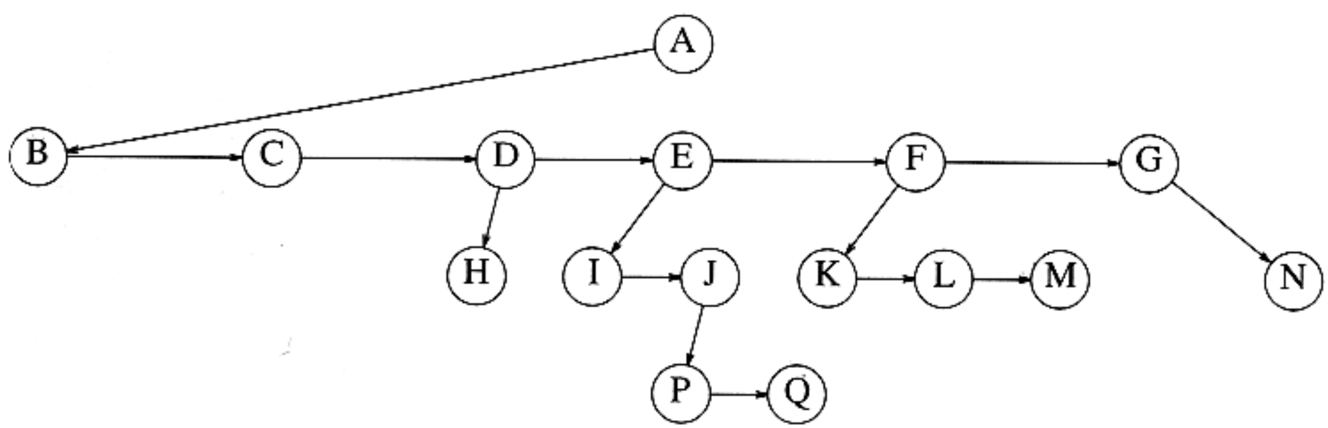
- The depth of an "average" binary tree is considerably < N.
- A well-balanced tree has a depth of O(log N).
- But, in the worst case, the depth can be as large as N – 1.



Convert a Generic Tree to a Binary Tree



- A downward link points to the first child.
- A horizontal link points to the next sibling.



Pre-/In-/Post-order Binary Tree Traversal

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));

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);

- Use #1: To print out the data in a tree in a certain order
- Use #2: To evaluate an expression tree

Binary Tree ADT

```
template <class Comparable>
class Btree_node
{
    Comparable element;    // the data
    Btree_node* left;      // left child
    Btree_node* right;     // right child
  public:
    Btree_node get_left ( ) { return left; }
    Btree_node get_right ( ) { return right; }
};

template<class Comparable>
void preorder (Btree_node<Comparable>* root)
{
    if (root)
    {
        // output the node
        preorder(root->get_left( ));
        preorder(root->get_right( ));
    }
}
```

Example: Unix Directory Traversal

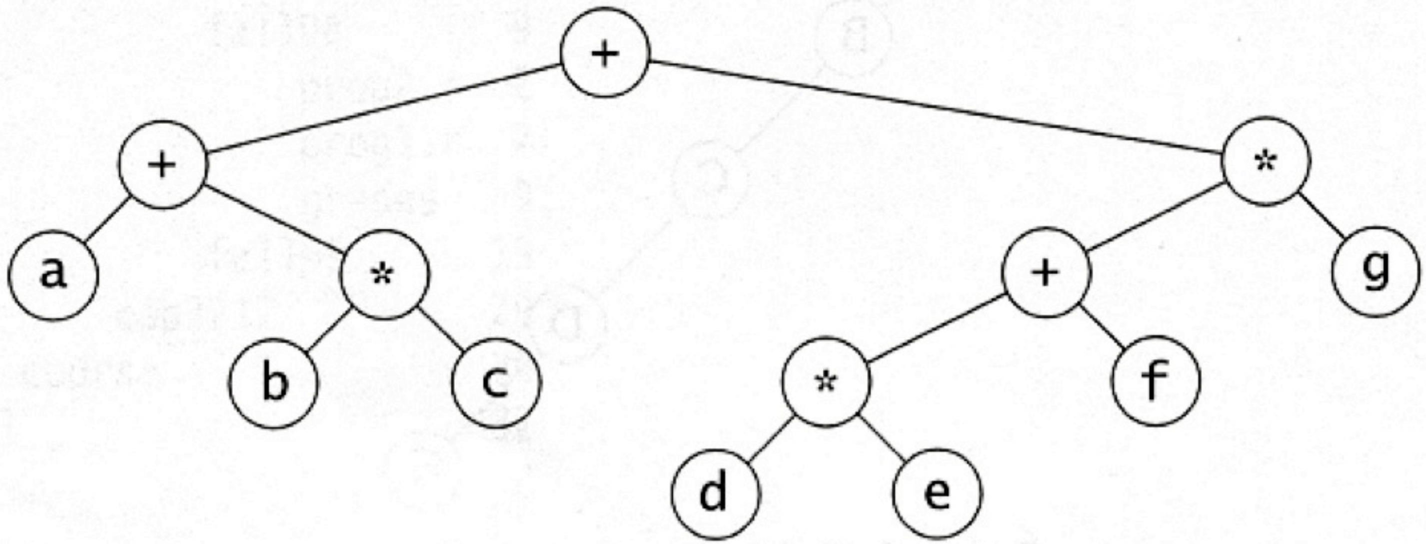
Preorder

```
/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
```

Postorder

```
            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
```

(Compiler) Example: Expression Trees

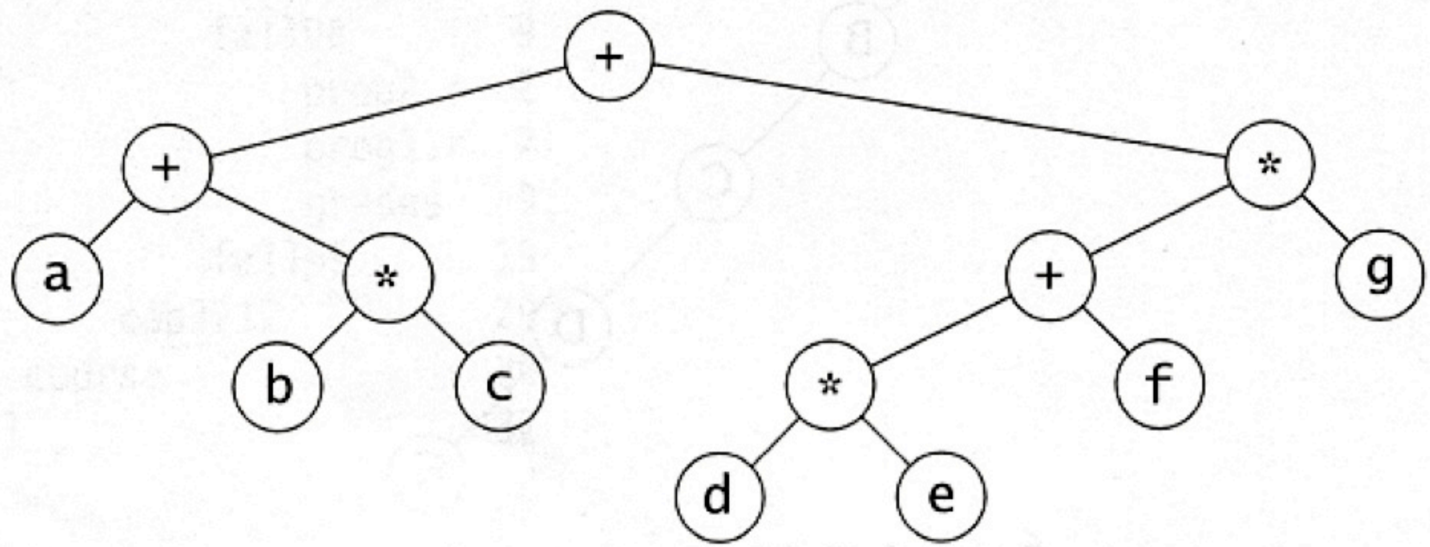


Expression tree for (a + b * c) + ((d * e + f) * g)

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

Preorder Traversal

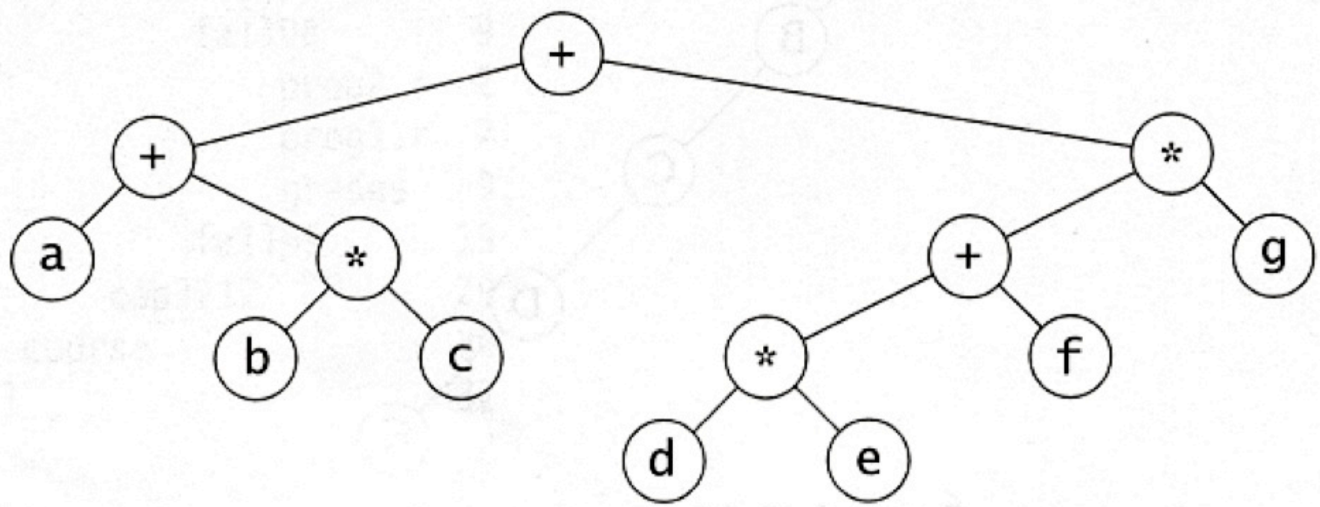
- Order: node, left, right
- Prefix expression: ++a*bc*+*defg



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

Inorder and Postorder Traversal

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



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

RPN or Postfix Notation

- Reverse Polish Notation (RPN) = Postfix notation
- Most compilers convert infix expressions to postfix notation. Advantages:
  - expressions can be written without parentheses
  - efficiently evaluated using a stack

Infix	RPN (Postfix)	Prefix
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D---	-A-B-C D
A - B - C - D	A B-C-D-	---A B C D

Infix to Postfix Conversion

1. Initialize an empty stack of operators
2. While no error and not end of expression
   a) Get next input "token" from infix expression, where a token is a constant/variable/arithmetic operator/parenthesis
   b) switch (token):
      i. "(" : push onto stack
      ii. ")" : pop the stack and display the elements until "(" occurs, do not display the “(“
      iii. operator:
         if the operator has *higher priority* than the top of stack
         or top is ‘(‘ *or* stack is empty
            push token onto the stack
         else // same or lower precedence
            pop the stack and display it
          repeat comparison of the operator with the top of the stack
      iv. operand: display it
3. End of infix reached: pop and display stack items until empty

RPN Expression Evaluation

Underlining principle

1. Scan the expression from left to right to find an operator
2. Locate ("underline") the last two preceding operands and combine them using this operator
3. Repeat until the end of the expression is reached

Example

2 * ((3+4) - (5-6))

2 3 4 + 5 6 - - *

→ 2 $\underline{3\ 4\ +}$ 5 6 - - *

→ 2 7 5 6 - - *

→ 2 7 $\underline{5\ 6\ -}$ - *

→ 2 7 -1 - *

→ 2 $\underline{7\ -1\ -}$ *

→ 2 8 *

→ $\underline{2\ 8\ *}$

→ 16

RPN Expression Evaluation ..

1. Initialize an empty stack
2. Repeat the following until the end of the expression is encountered
   a) Get the next token (const, var, operator)
   b) Operand – push onto stack
      Operator – do the following
      i. Pop 2 values from stack
      ii. Apply operator to the two values
      iii. Push resulting value back onto stack
3. When the end of expression is reached, its value should be the only number left in stack; otherwise it is in error.

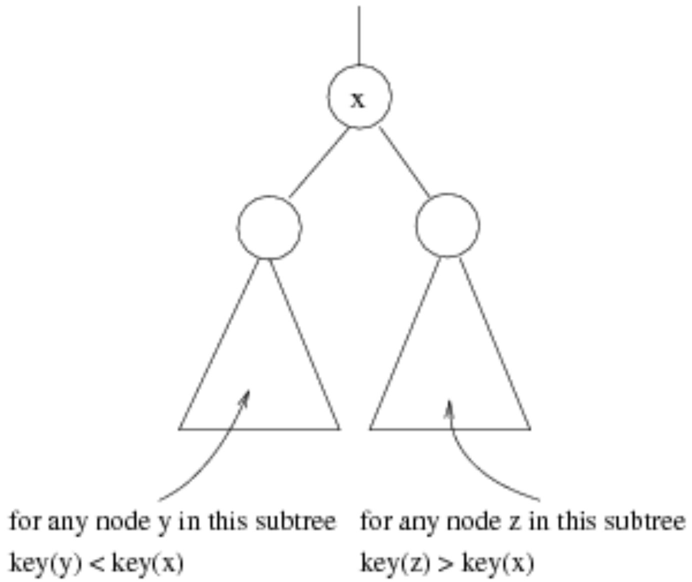
Note: if only 1 value on stack, this is a pop error, i.e., an invalid RPN expression

RPN Expression Evaluation ...

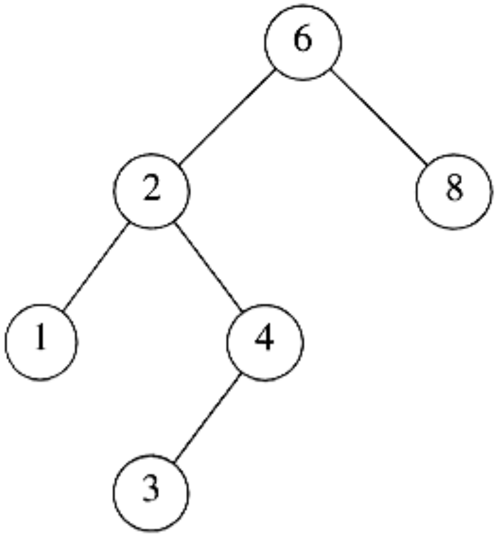
Input	Stack (top first)	Action
2 4 * 9 5 + -	**2**	✧ Push 2 onto the stack
4 * 9 5 + -	**4**, 2	✧ Push 4 onto the stack
* 9 5 + -	**8**	✧ Pop 4 and 2 from the stack , multiply, and push the result 8 back
9 5 + -	**9**, 8	✧ Push 9 onto the stack
5 + -	**5**, 9, 8	✧ Push 5 onto the stack
+ -	**14**, 8	✧ Pop 5 and 9 from the stack, add, and push the result 14 back
-	**-6**	✧ Pop 14 and 8 from the stack, subtract, and push the result -6 back
(end of strings)	**-6**	✧ Value of expression is on top of the stack

Binary Search Trees (BST)

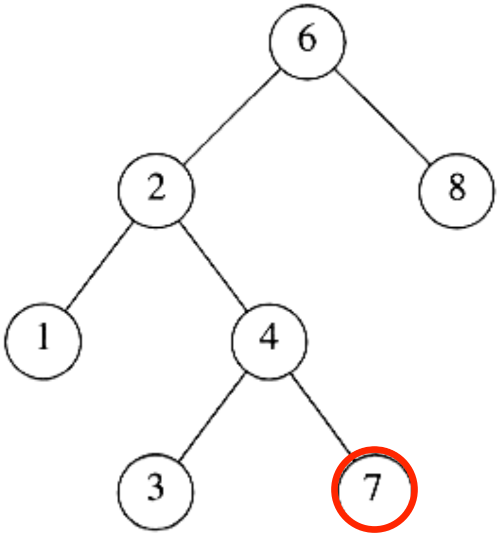
- A data structure for efficient searching, insertion and deletion.
- Binary search tree property: For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X



Binary Search Trees ..



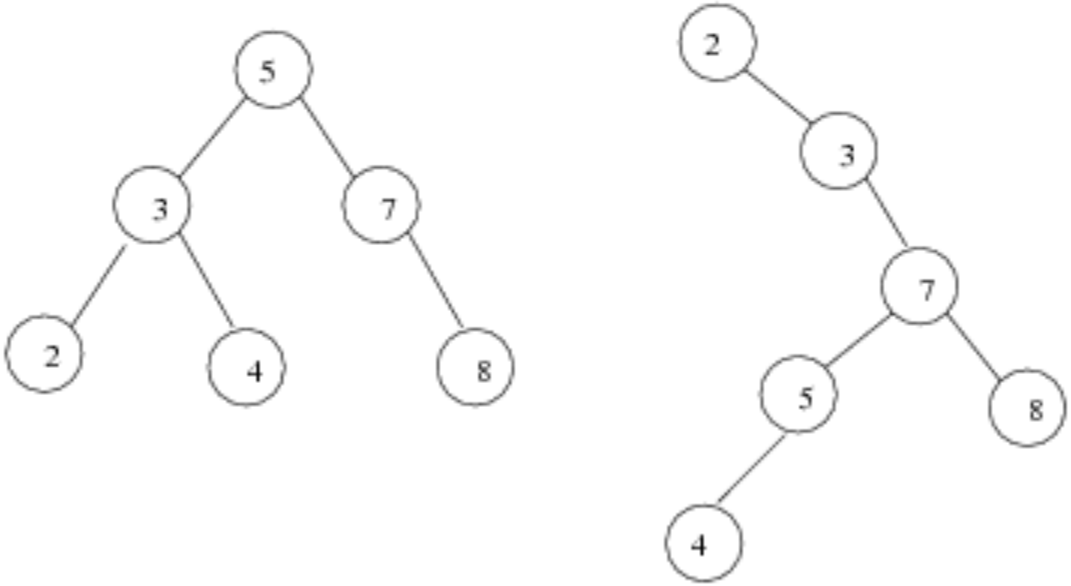
A binary search tree



Not a binary search tree

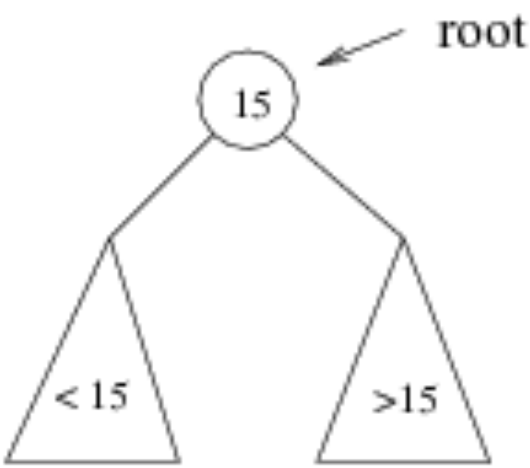
BSTs May Not Be Unique

- The same set of values may have different BSTs.
- Average depth of a node is $O(\log N)$
- Maximum depth of a node is $O(N)$



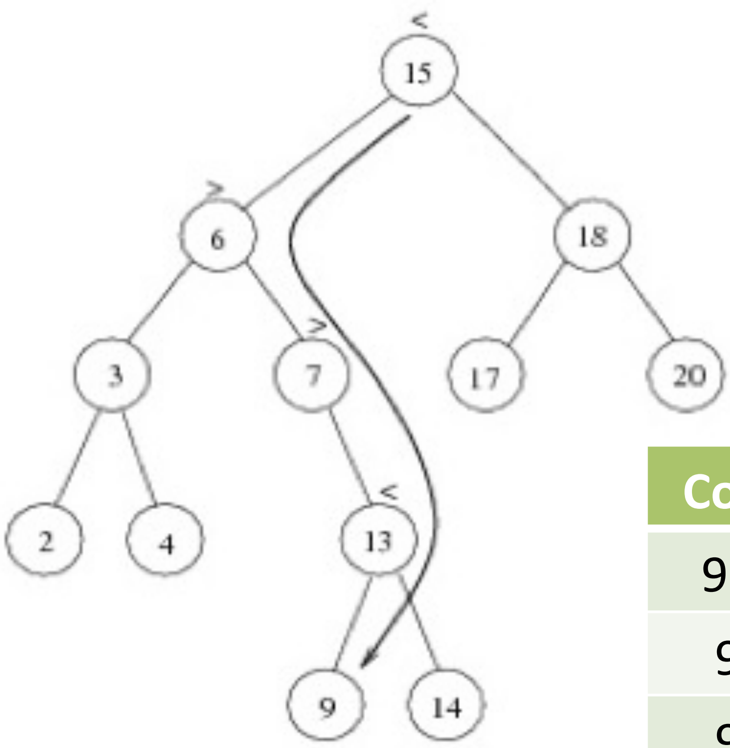
Searching BST

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



Example: Search BST

Search for 9 ...



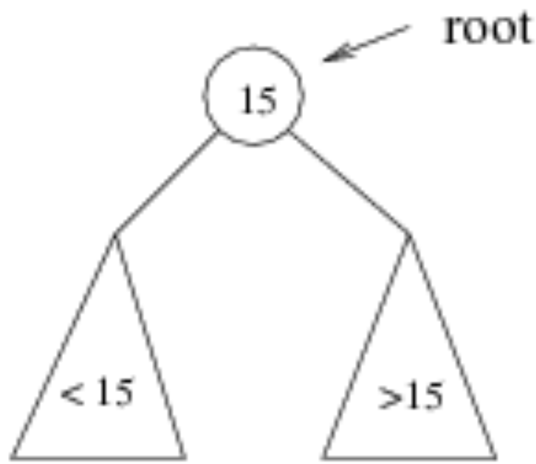
Compare	Action
9 vs. 15	continue with the left subtree
9 vs. 6	continue with the right subtree
9 vs. 7	continue with the right subtree
9 vs. 13	continue with the left subtree
9 vs. 9	eureka!

BST ADT (different from textbook)

```cpp
template <typename T> class BST
{
  private:
    static const T DUMMY; // Returned value of an empty BST
    struct bst_node {     // A node in a binary search tree
        T value;
        BST left;         // Left sub-tree or called left child
        BST right;        // Right sub-tree or called right child
        bst_node(const T& x) : value(x), left( ), right( ) { };
    };

    bst_node* root;

  public:
    BST( ) : root(NULL) { }     // Empty BST when its root is NULL
    BST(const BST& bst) { root = bst.root; } // Shallow BST copy
    ~BST( ) { delete root; }

    bool is_empty( ) const { return root == NULL; }
    bool contains(const T& x) const;
    void print(int depth = 0) const;

    const T& find_max( ) const; // Find the maximum value
    const T& find_min( ) const; // Find the minimum value
    void insert(const T&);      // Insert an item with a policy
    void remove(const T&);      // Remove an item
};
```

Our BST ADT

- Our ADT definition conforms more with the following BST figure:



- That is, a BST object has a root pointing to a BST node which has
  - a value (of any type)
  - a left BST subtree
  - a right BST subtree

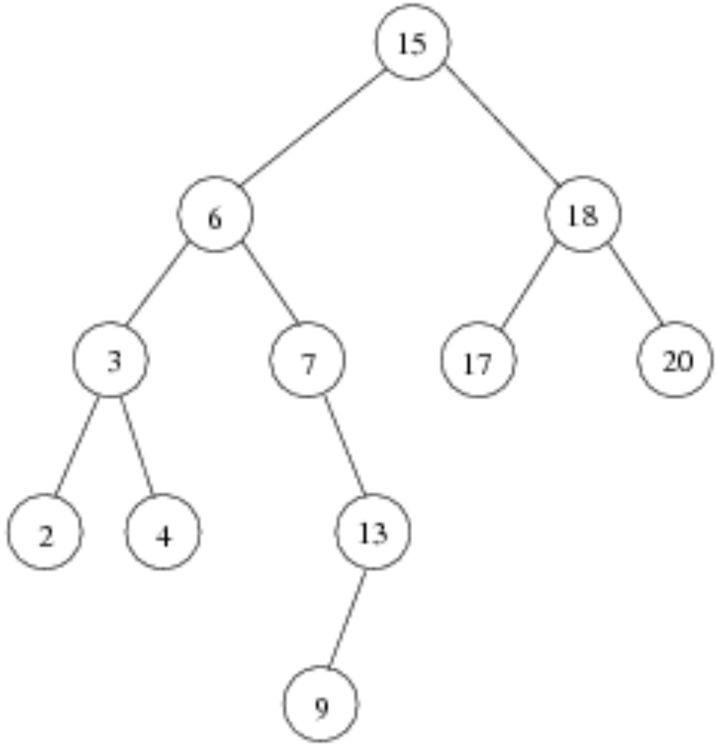
Searching (contains)

- Check if the BST contains the value X: true or false.
- Time complexity: O(height of the tree)

```
template <typename T>
bool BST<T>::contains(const T& x) const
{
    if (is_empty( ))
        return false;

    if (root->value == x)
        return true;
    else if (x < root->value)
        return root->left.contains(x);
    else
        return root->right.contains(x);
}
```

Inorder Traversal of BST

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



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

Print By Rotating BST -90 Degrees

```
template <typename T>
void BST<T>::print(int depth) const
{
    if (is_empty( ))                    // Base case
       return;

   root->right.print(depth+1);          // Recursion: right subtree

    for (int j = 0; j < depth; j++)     // Print the node value
        cout << '\t';
    cout << root->value << endl;

    root->left.print(depth+1);          // Recursion: left subtree
}
```

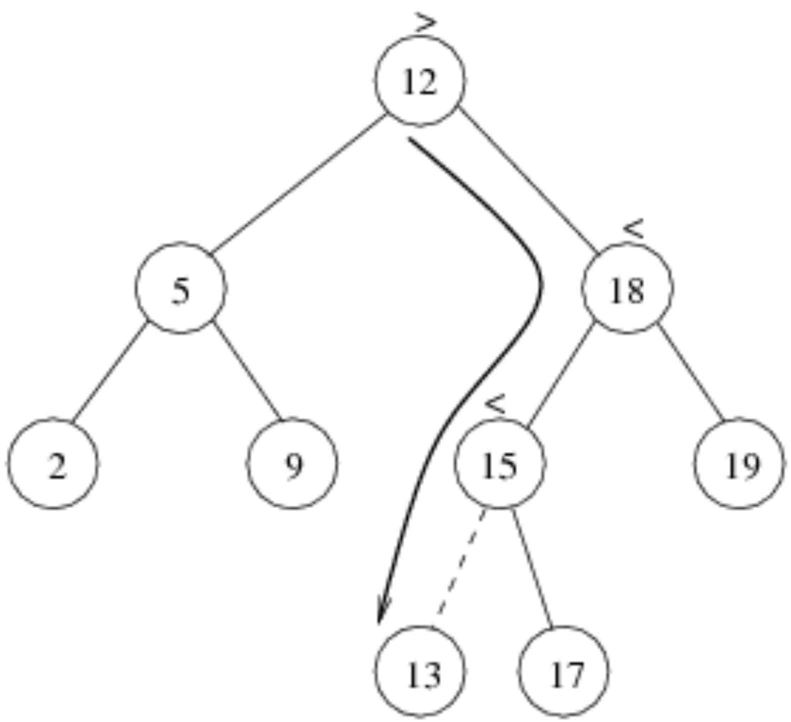
Finding the Min/Max Value in a BST

- Start at the root and go left (right) as long as there is a left (right) child. The stopping point is the min. (max.) element.
- Time complexity = O(height of the tree)

```
template <typename T>   // Find the minimum value
const T& BST<T>::find_min( ) const
{
    if (is_empty( ))    // Should not happen!
    {
        cerr << "Error: find_min( ) called by an empty BST\n";
        return DUMMY;   // Returned value is a dummy
    }

    const bst_node* node = root;
    while (!node->left.is_empty( ))
        node = node->left.root;

    return node->value;
}
```

Insertion

- Proceed down the tree as you would with a search.
- If X is found, do nothing (or update something).
- Otherwise, insert X at the last spot on the path traversed.
- Time complexity = O(height of the tree)
- E.g., search 13:



Insertion ..

```
template <typename T>   // Insert and preserve the BST property
void BST<T>::insert(const T& x)
{
    if (is_empty( ))    // Find the spot
        root = new bst_node(x);
    else if (x < root->value)
        root->left.insert(x);
    else if (x > root->value)
        root->right.insert(x);
    else
        ;               // x == root->value; do nothing
}
```

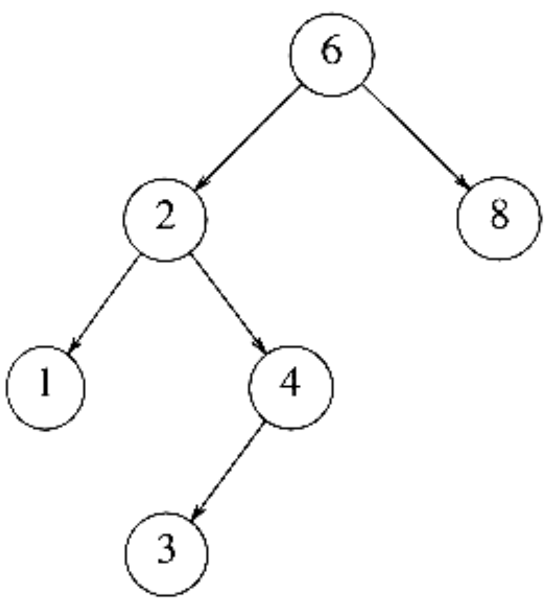
Deletion

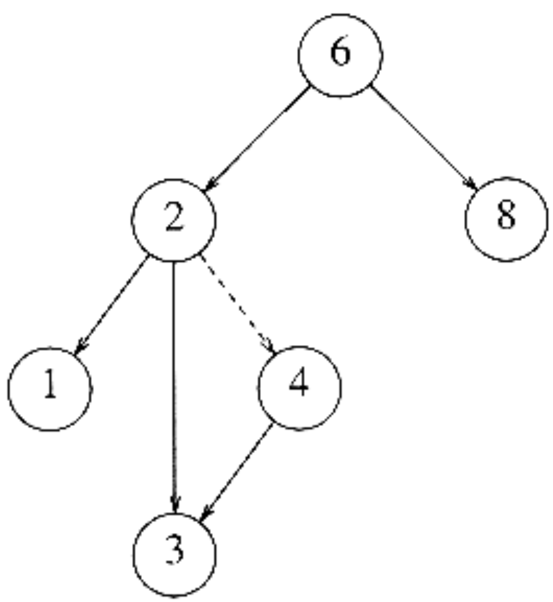
- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.
- Time complexity = O(height of the tree)

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

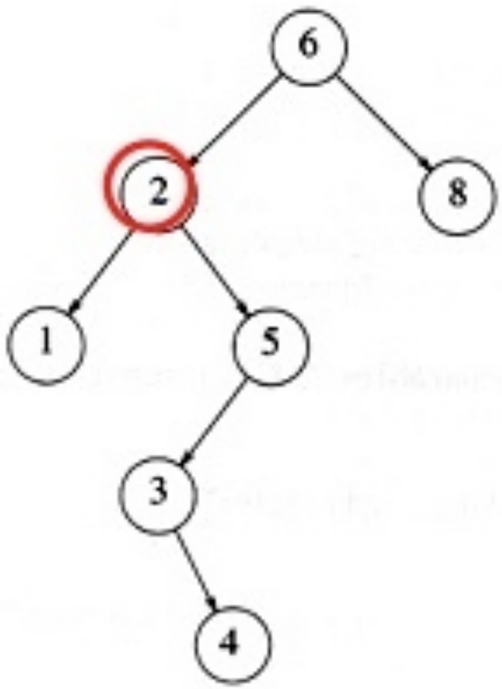
Example: delete 4

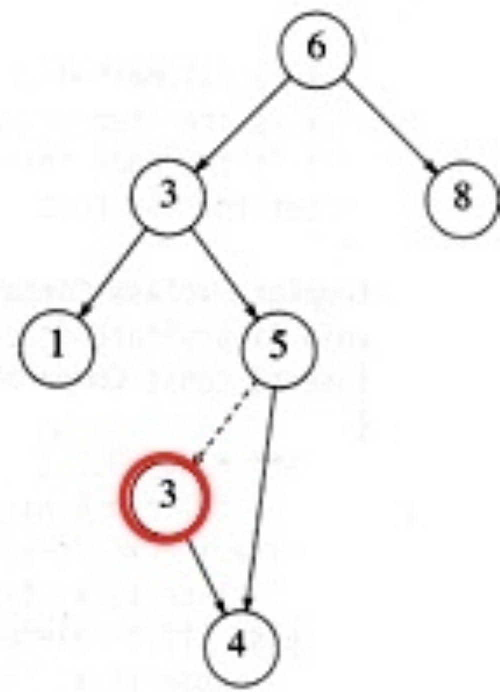




Deletion under Different Cases ..

- Case 3: the node has 2 children
  - Replace its value with the min. value of its right subtree
  - Delete that node with the min. value.
    - Must have either no child or only right child otherwise, that node would not have been the min. in the first place! So, invoke case 1 or 2.





Deletion Code

```
void BST<T>::remove(const T& x) // leftmost item of its right subtree
{
    if (is_empty( ))              // Item is not found; do nothing
        return;

    if (x < root->value)          // Remove from the left subtree
        root->left.remove(x);
    else if (x > root->value)     // Remove from the right subtree
        root->right.remove(x);
    else if (root->left.root && root->right.root) // Found node has 2 children
    {
        root->value = root->right.find_min( );
        root->right.remove(root->value);
    }
    else // Found node has 0 or 1 child
    {
        bst_node* deleting_node = root; // Save the root to delete first
        root = (root->left.is_empty( )) ? root->right.root : root->left.root;

        // Reset its left/right subtree to null first before removal
        deleting_node->left.root = deleting_node->right.root = NULL;
        delete deleting_node;
    }
```

Deletion Code ..

- The code is not efficient when the node to be deleted has 2 children because
  - it makes another recursive call to delete the node in the right subtree with the min. value
  - deletion in this case is done by copying the value of the min. node to the deleting node (what if the data type of value is that of a big object?)
- Try to re-write without the additional recursive call and without copying.