CPTS 223 Advanced Data Structure C/C++

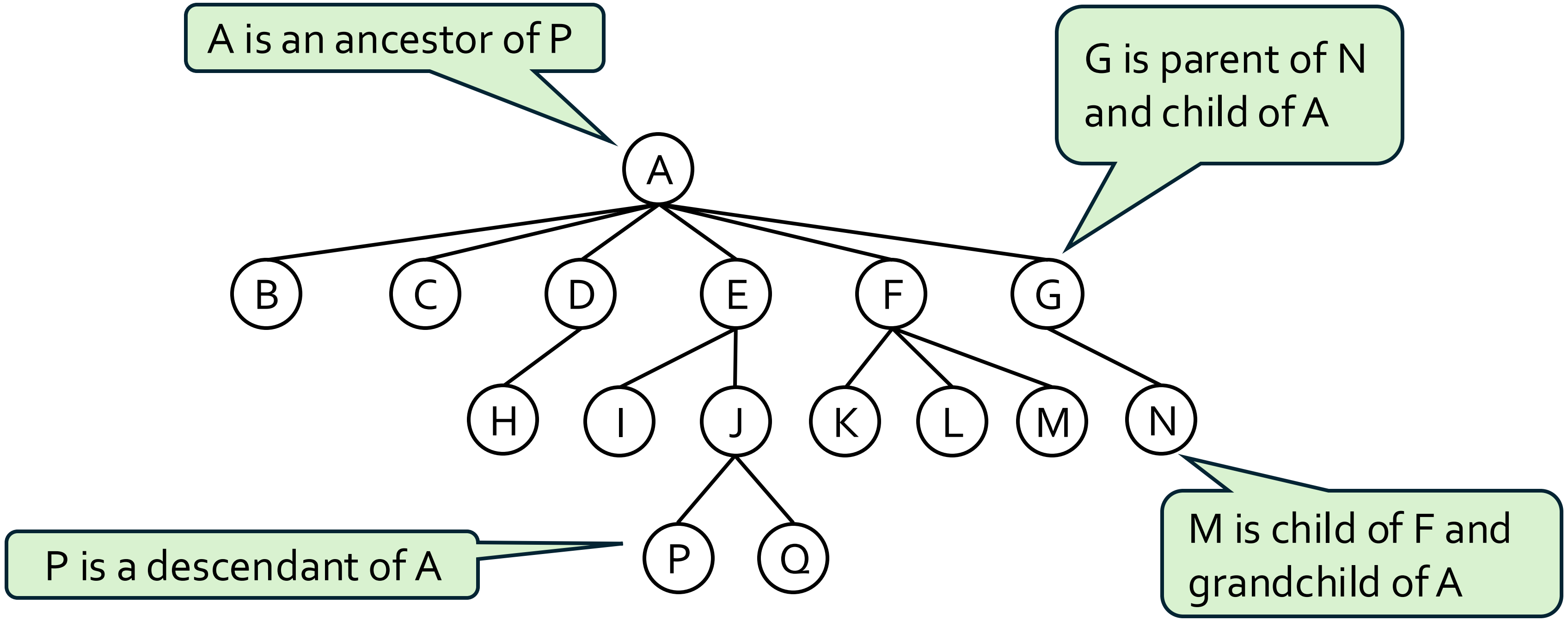
Tree: Binary Search Tree

Overview

- Tree data structure
- Binary search trees
    - Support O(lg(n)) operations
    - Balanced trees
- STL set and map classes
- B-trees for accessing secondary storage
- Applications of Tree

WSU

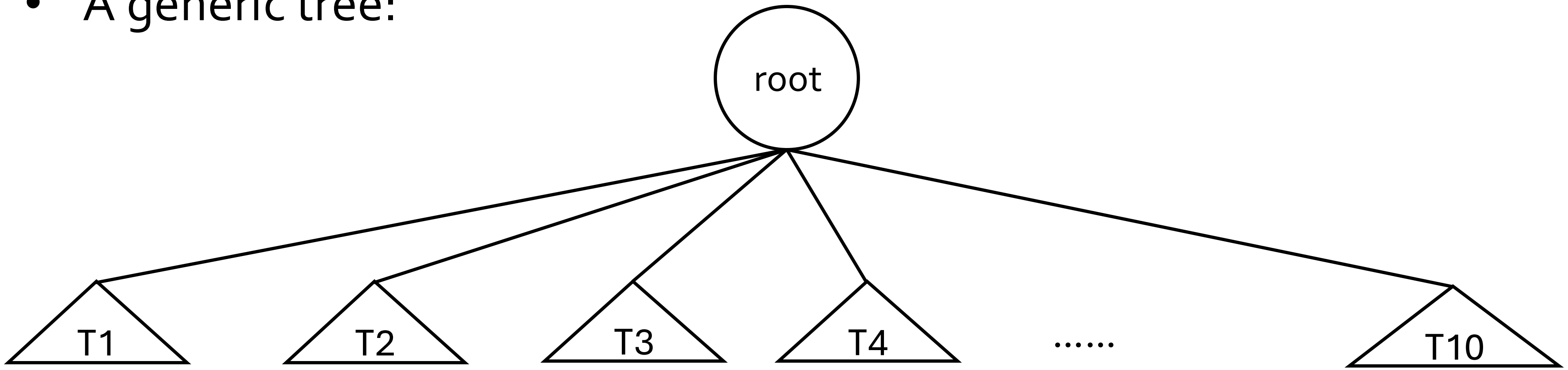
Trees


A is an ancestor of P
G is parent of N and child of A
A
B
C
D
E
F
G
H
I
J
K
L
M
N
P
Q
P is a descendant of A
M is child of F and grandchild of A

WSU

Trees

- A generic tree:



Basic definitions

- A tree T is a set of nodes that form a directed acyclic graph (DAG) such that:
  - Each non-empty tree has a root node and zero or more sub-trees $T_1, \ldots, T_k$
  - Each sub-tree is a tree
    Recursive definition
  - An internal node is connected to its children by a directed edge
- Each node in a tree has only one parent
  - Except the root, which has no parent

Basic definitions

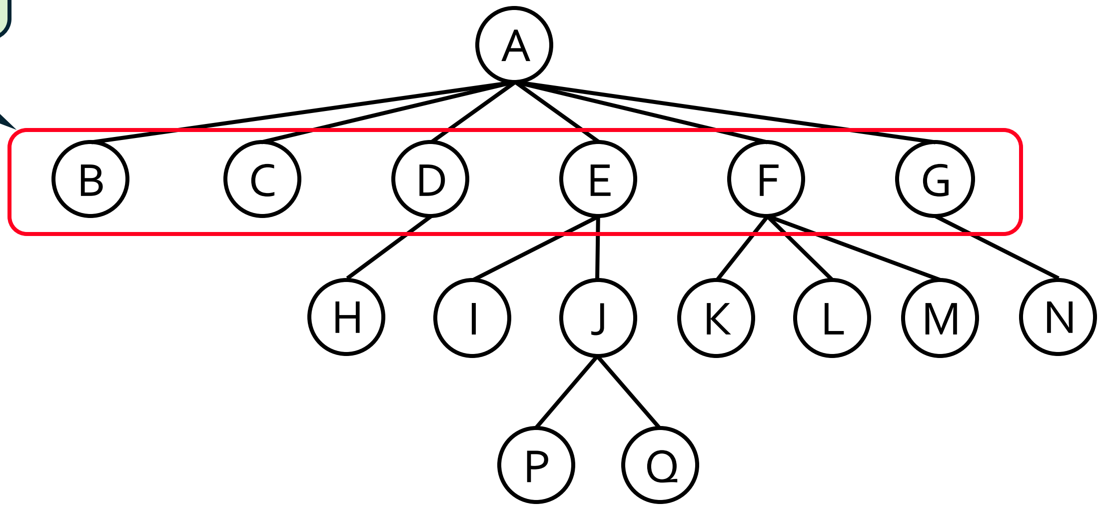
- Internal node: nodes with at least one child
- Leaf node: nodes with no children

Nodes → either a leaf or an internal node

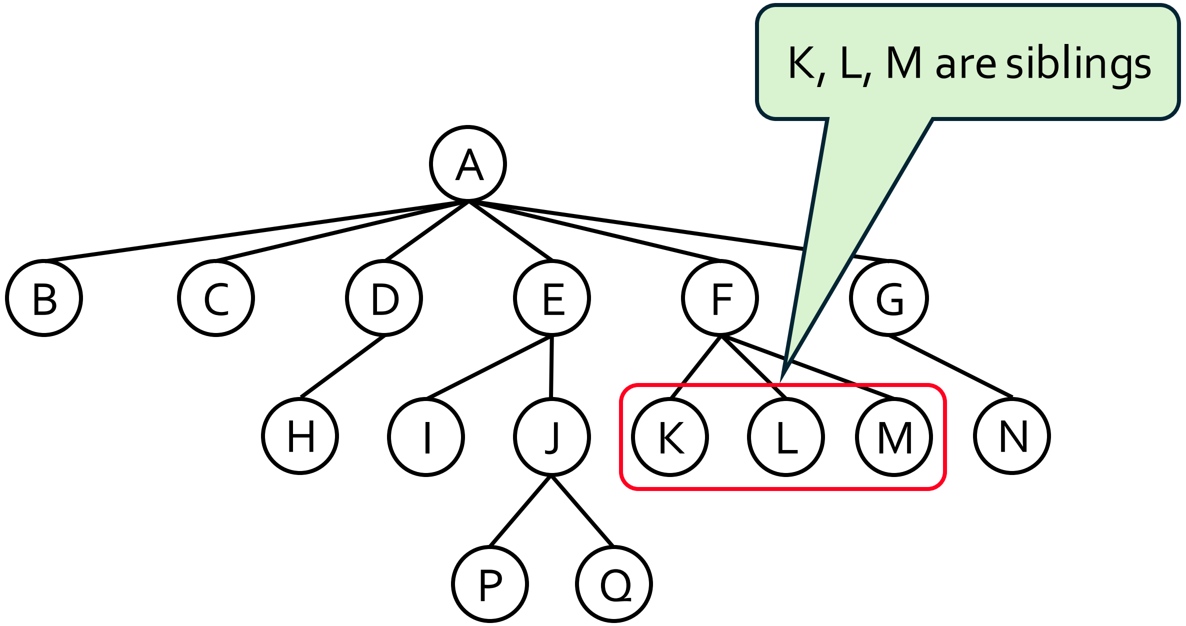
- Siblings: nodes with the same parent
- A path from node n_1 to n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$
  - The length of a path is the number of edges on the path (i.e., $k-1$)
  - Each node has a path of length 0 to itself
  - There is exactly one path from the root to each node in a tree
  - Nodes $n_i, \ldots, n_k$ are descendants of n_i and ancestors of n_k
  - Nodes $n_{i+1}, \ldots, n_k$ are proper descendants of n_i
  - Nodes $n_i, \ldots, n_{k-1}$ are proper ancestors of n_k

Node relationships

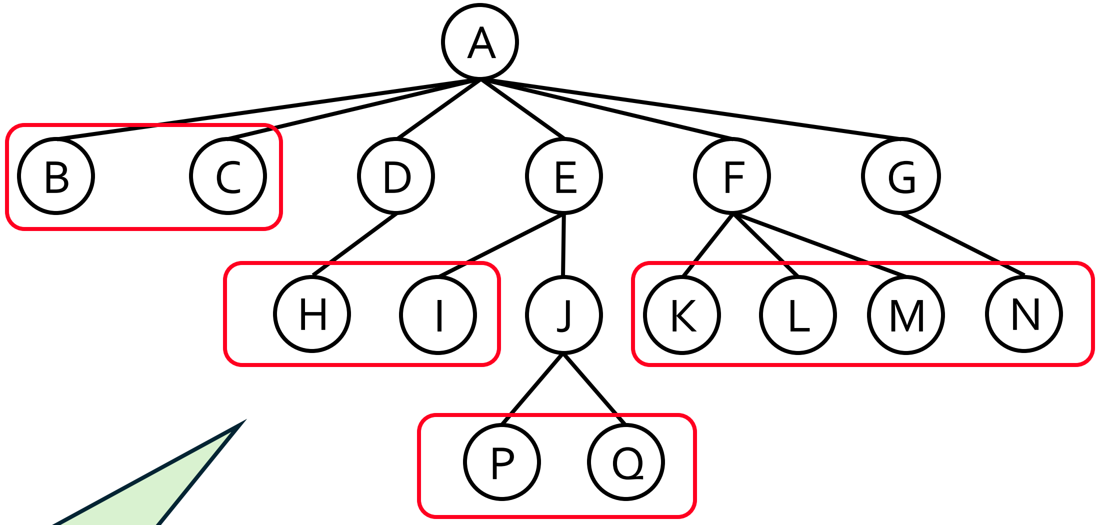
B, C, D, E, F, G are siblings



Node relationships



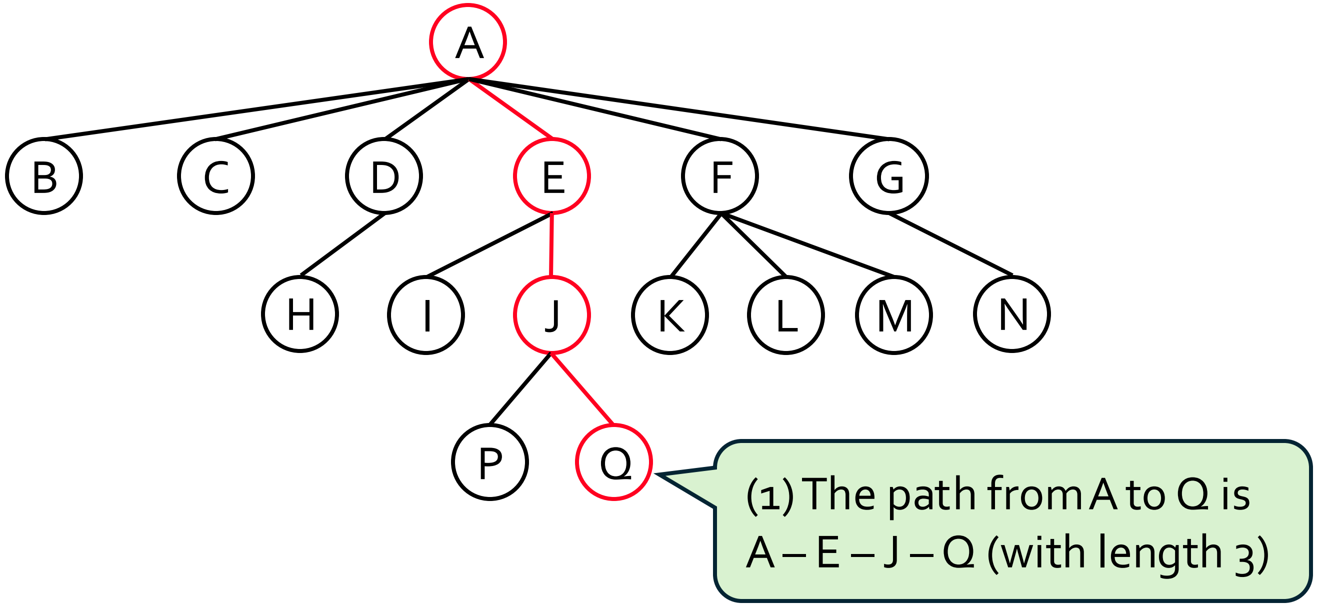
Node relationships



B, C, H, I, P, Q, K, L, M, N are leaves

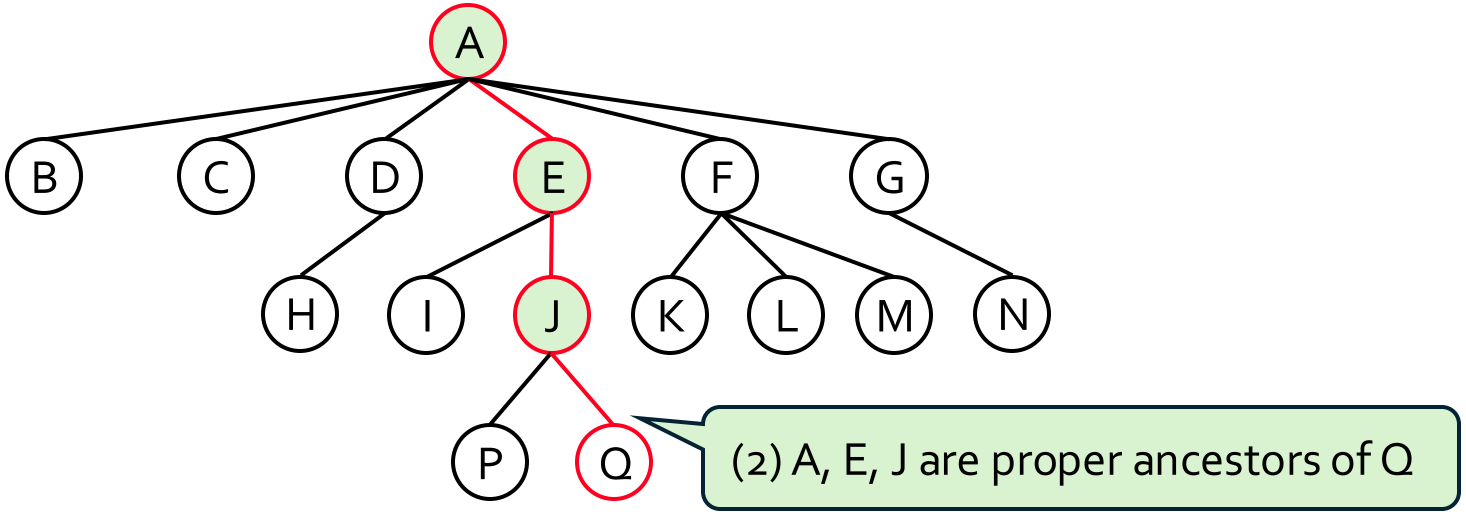
WSU

Node relationships

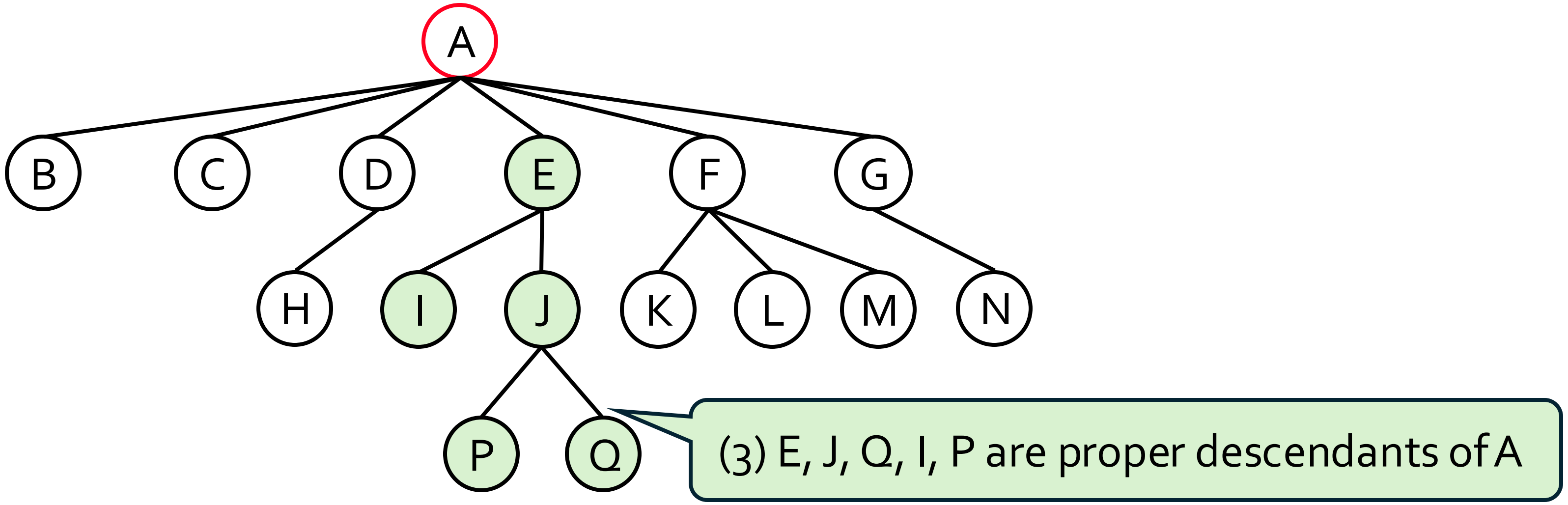


WSU

Node relationships



Node relationships

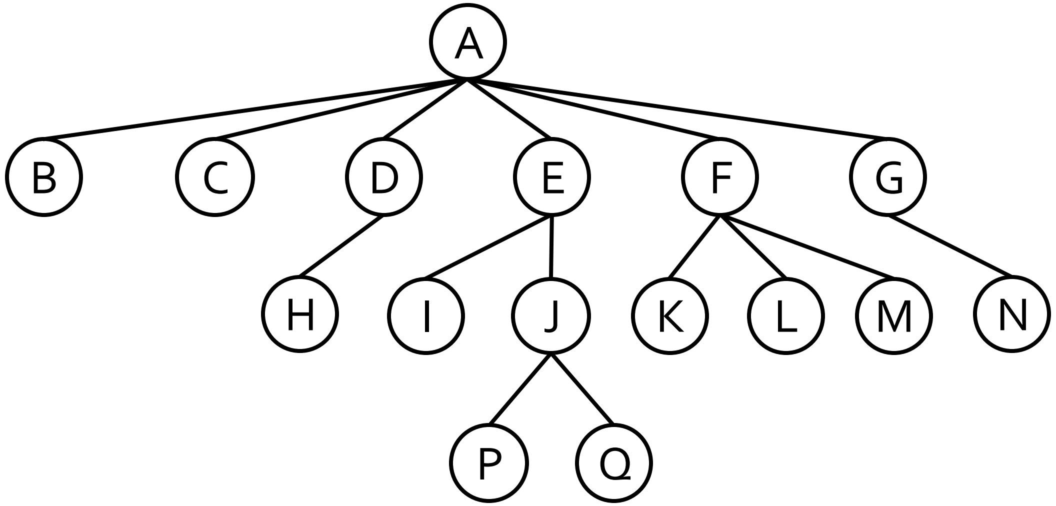


WSU

Basic definitions

- The depth of a node n_i is the length of the path from the root to n_i
  - The root node has a depth of 0
  - The depth of a tree == the depth of its deepest leaf
- The height of a node n_i is the length of the longest path under n_i's subtree
  - All leaves have a height of 0
- height of tree = height of root = depth of tree

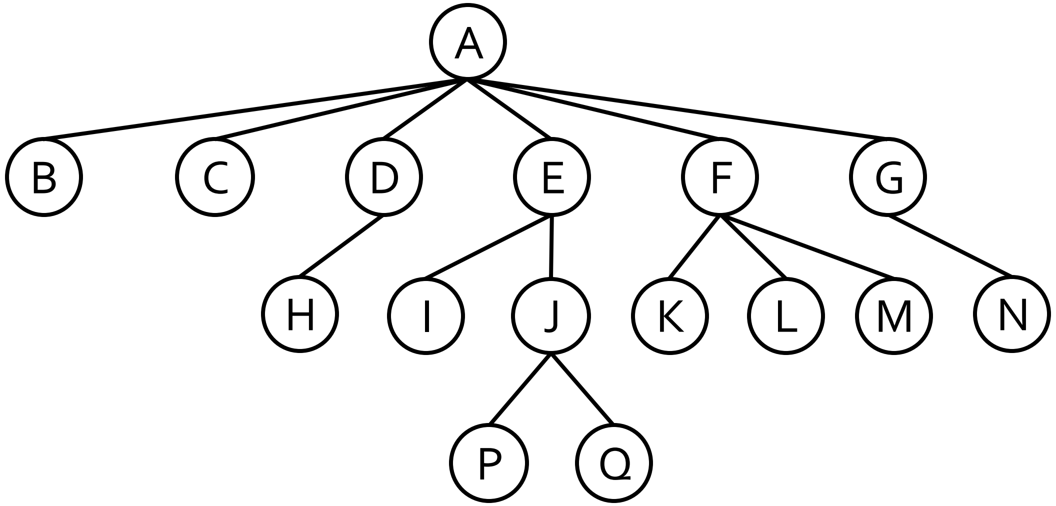
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

e.g., height(E)=2, height(L)=0

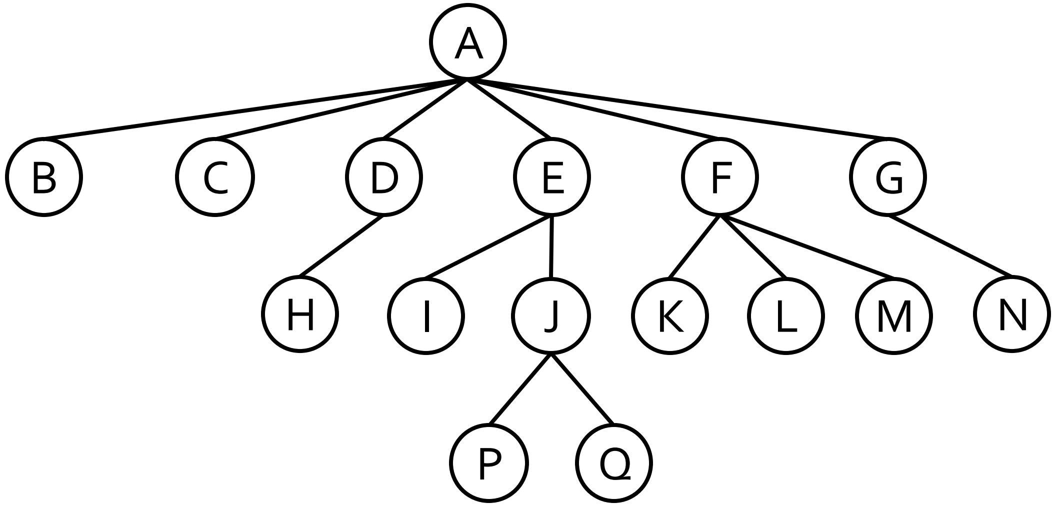
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

= 3 (height of longest path from root)

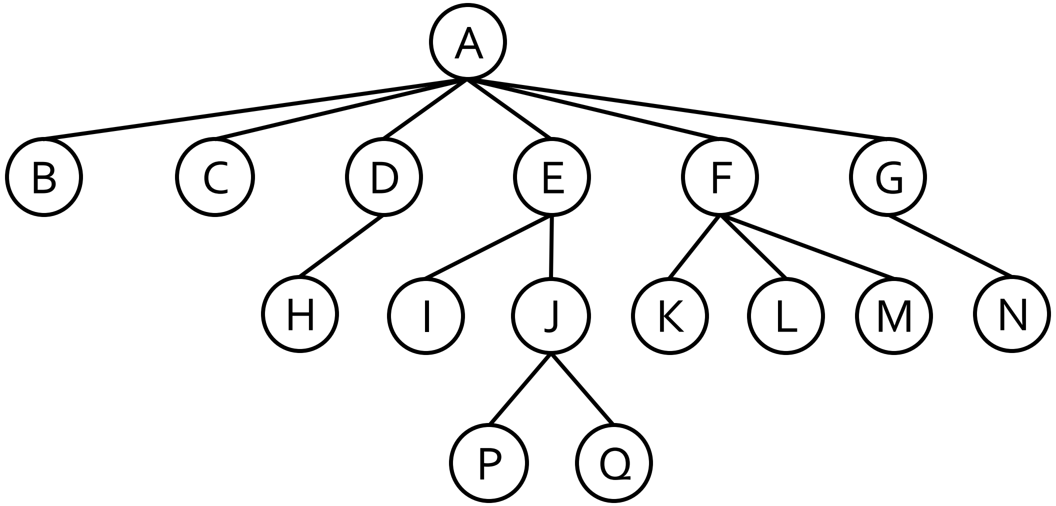
Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

e.g., depth(E)=1, depth(L)=2

Height and depth



1. Height of each node?
2. Height of the tree?
3. Depth of each node?
4. Depth of the tree?

= 3 (length of the path to the deepest node)

Implementation of trees

- Solution 1: vector of children

```
Struct TreeNode {
  Object element;
  vector<TreeNode> children;
 }
```

Direct access to children[i]
However:
 need to know max allowed
 children in advance & more space

- Solution 2: list of children

```
Struct TreeNode {
  Object element;
  list<TreeNode> children;
 }
```

Number of children can be
dynamically determined
However:
 more time to access children

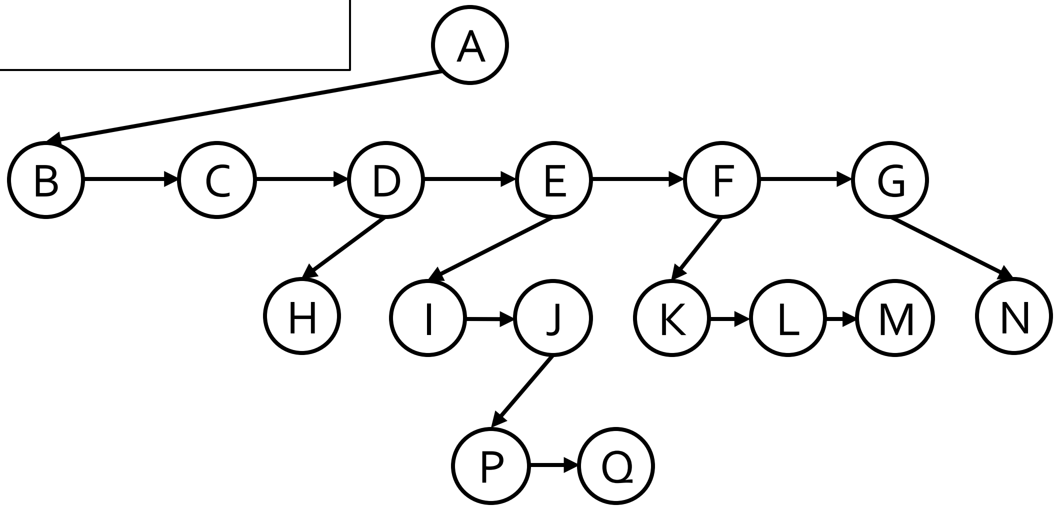
WSU

Implementation of trees

- Solution 3: left-child, right-sibling

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

Guarantees 2 pointers per node (independent of #children)
However:
Access time proportional to #children



Implementation of trees

- Solution 3: left-child, right-sibling

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

Guarantees 2 pointers per node
(independent of #children)
However:
Access time proportional to #children

- Which is the best solution?

Implementation of trees

- Solution 3: left-child, right-sibling

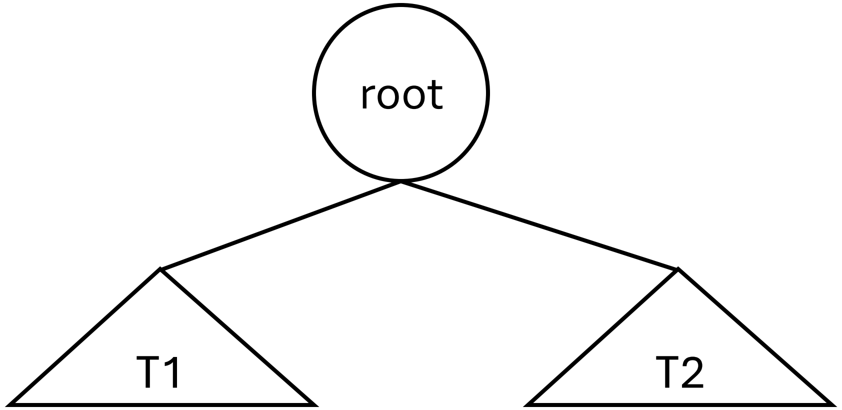
```
Struct TreeNode {
  Object element;
  TreeNode *firstChild;
  TreeNode *nextSibling;
}
```

Guarantees 2 pointers per node
(independent of #children)
However:
Access time proportional to #children

- Which is the best solution?
- Which is the best solution for searching tasks?

Binary trees (two-way trees)

- A binary tree is a tree where each node has no more than two children

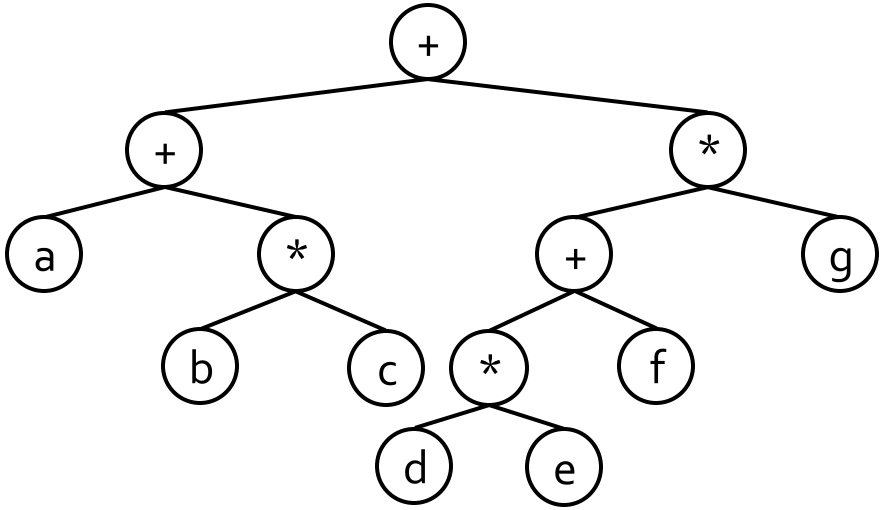



```
struct BinaryTreeNode
{
  Object element;
  BinaryTreeNode *leftChild;
  BinaryTreeNode *rightChild;
}
```

- If a node is missing one or both children, then that child pointer is NULL

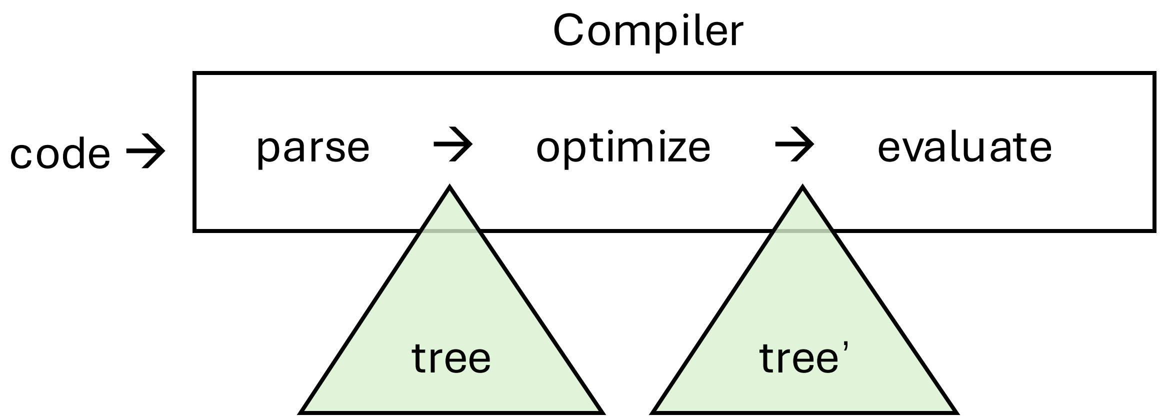
Example: expression trees

- Store expressions in a binary tree
  - Leaves of tree are operands (e.g., constants, variables)
  - Other internal nodes are unary or binary operators
- Used by compilers to parse and evaluate expressions
  - Arithmetic, logic, etc.
  - e.g., (a + b * c)+((d * e + f) * g):



WSU

Example: expression trees

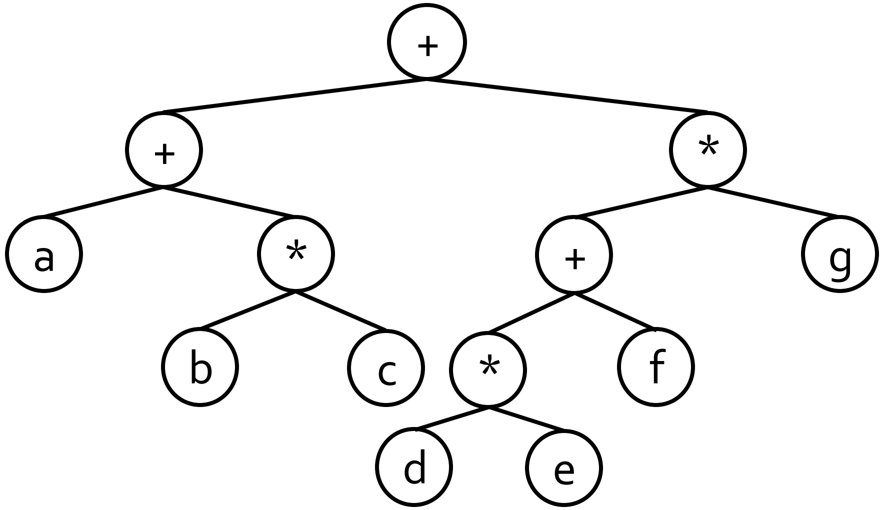


Example: expression trees

- Evaluate expression
  - Recursively evaluate left and right subtrees
  - Apply operator at root node to results from subtrees
- DFS (depth-first search) traversals (recursive definitions)
  - Same evaluation result, different expression notations
  - Post-order: left, right, root
  - Pre-order: root, left, right
  - In-order: left, root, right

WSU

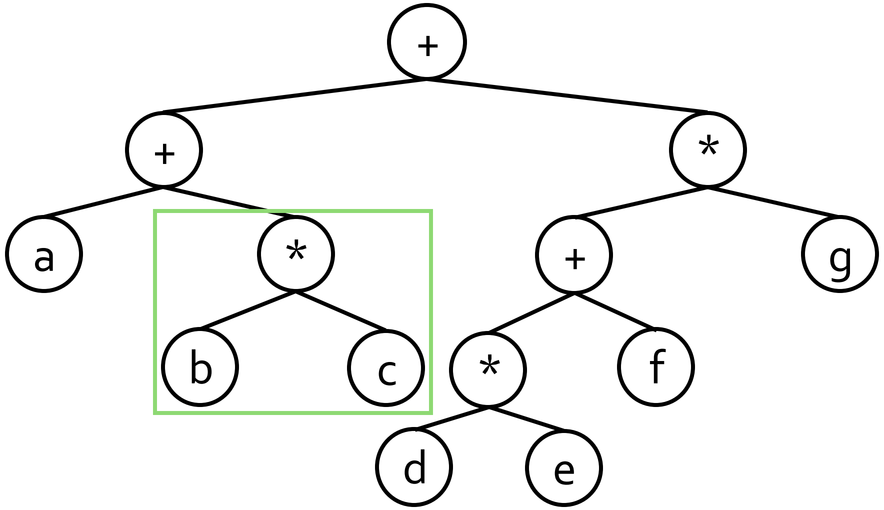
Traversals



- Pre-order: root - left - right
- Post-order: left - right - root
- In-order: left - root - right

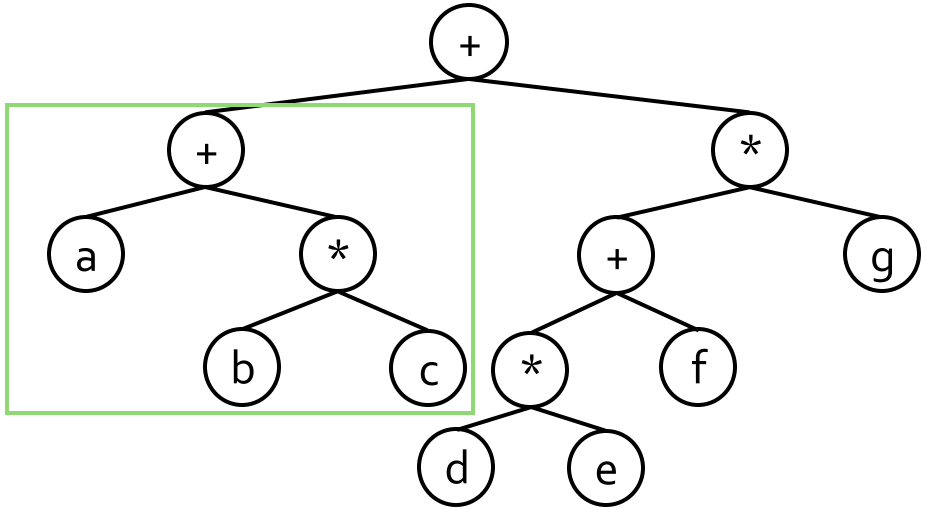
WSU

Traversals



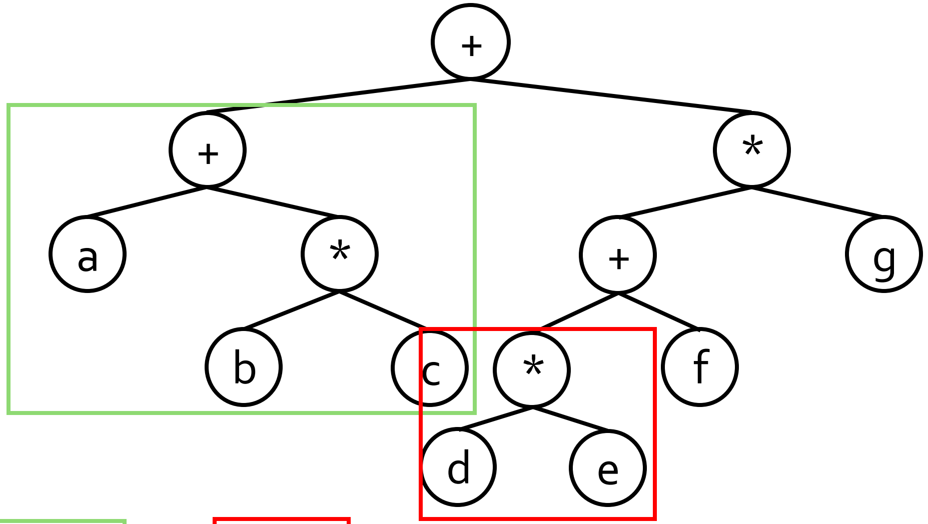
- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

Traversals



- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

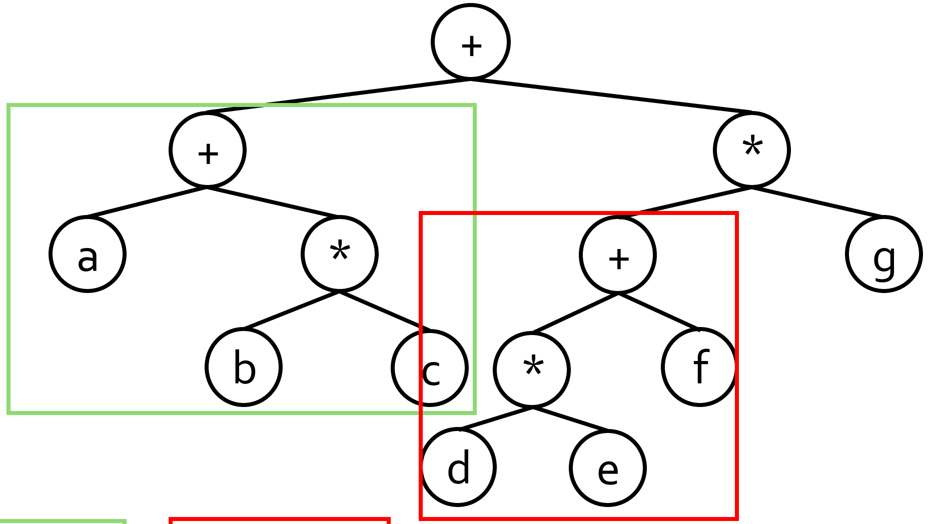
Traversals



- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

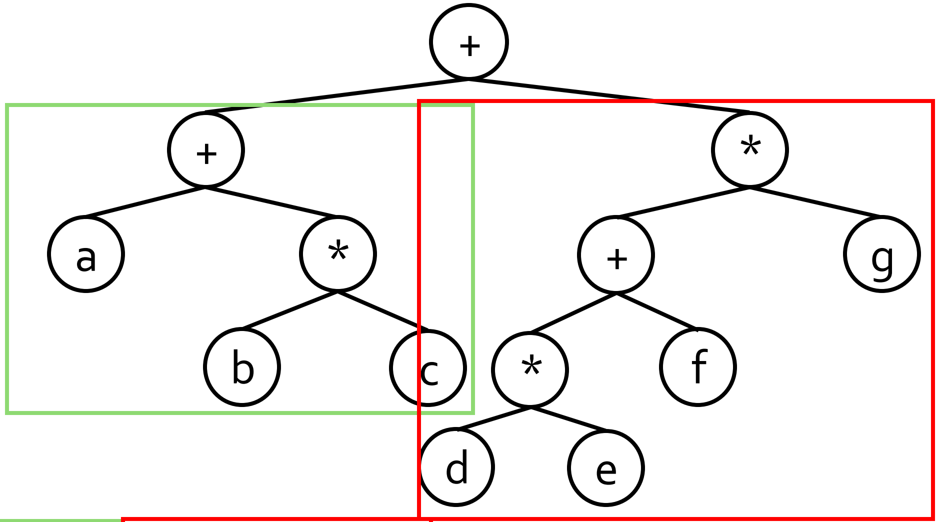
WSU

Traversals



- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

Traversals



- Pre-order: + + a * b c * + * d e f g
- Post-order: a b c * + d e * f + g * +
- In-order: a + b * c + d * e + f * g

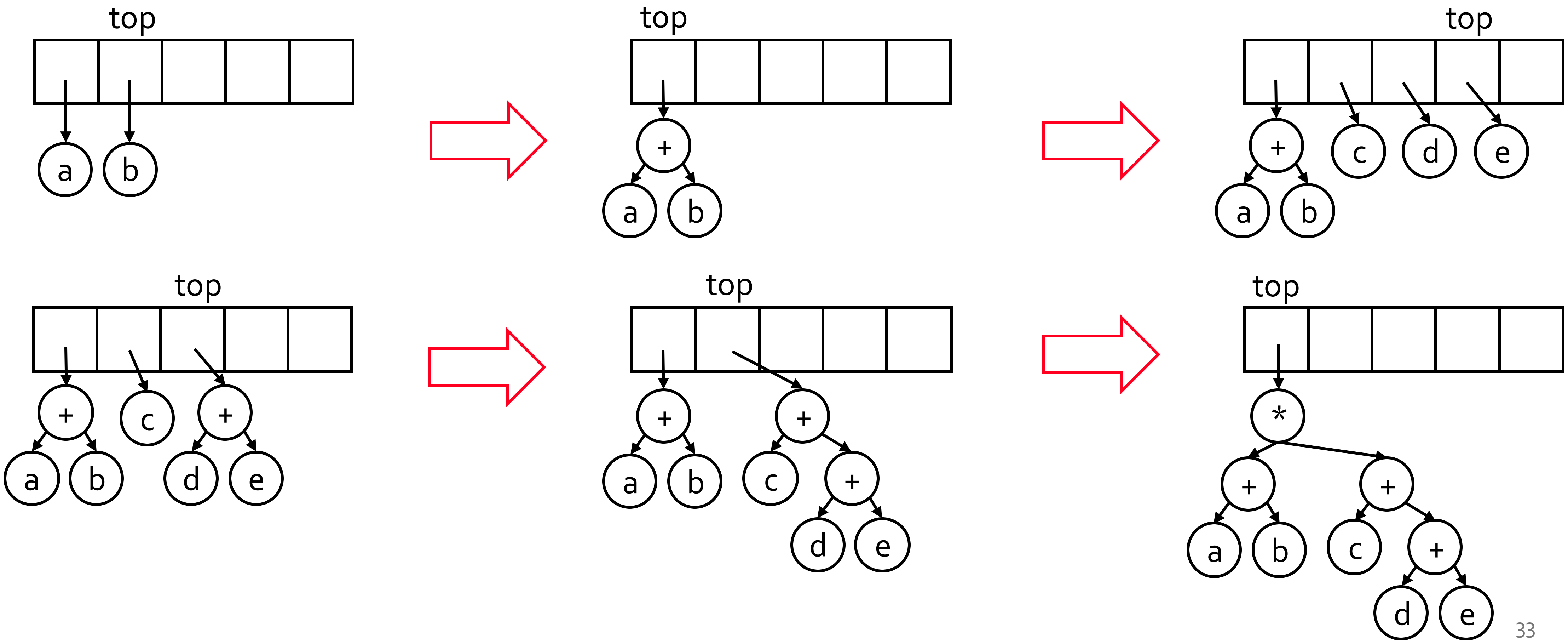
Example: expression trees

- Constructing an expression tree from postfix notation
    - Use a stack of pointers to trees
    - Read postfix expression left to right
    - If operand, then push on stack
    - If operator, then:
        - Create a BinaryTreeNode with operator as the element
        - Pop top two items off stack
        - Insert these items as left and right child of new node
        - Push pointer to node on the stack
    - End of the expression? 1 pointer on stack, -> root

WSU

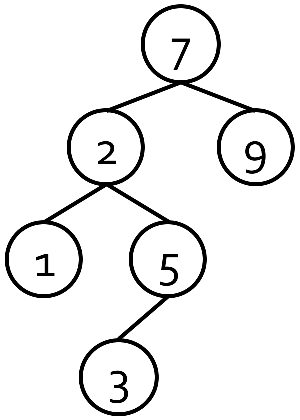
Example: expression trees

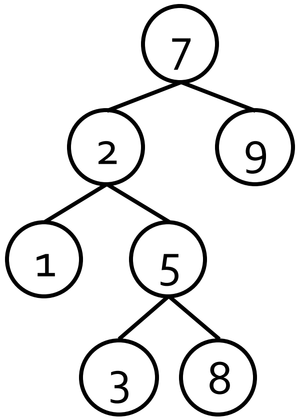
Target: a b + c d e + * * with stack

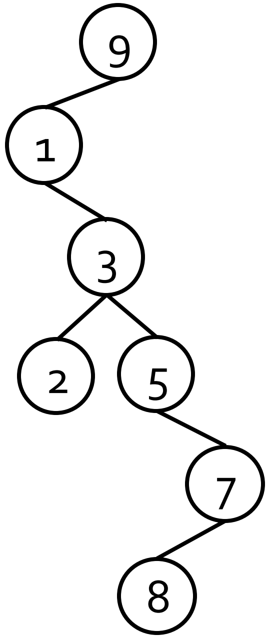


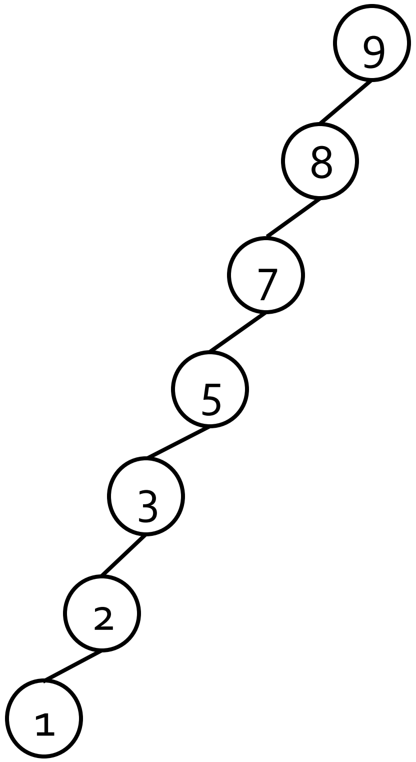
Binary search trees (BSTs)

- For any node n, items in left subtree of n
  - ≤ item in node n
  - ≤ items in right subtree of n
- Which one is not a BST?









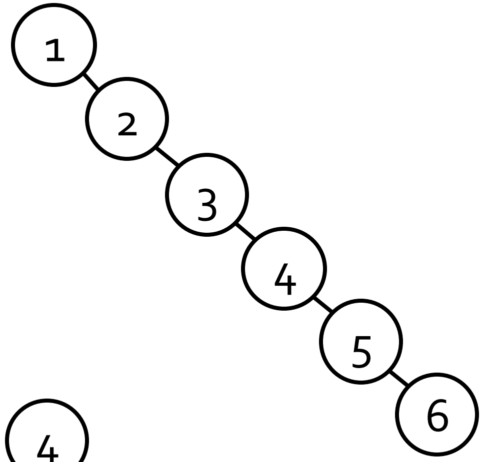
Searching in BSTs

```
Contains(T, x) {
  if (T == NULL)
    then return NULL
  if (T->element == x)
    then return T
  if (x < T->element)
    then return Contains(T->leftChild, x)
  else return Contains(T->rightChild, x)
}
```

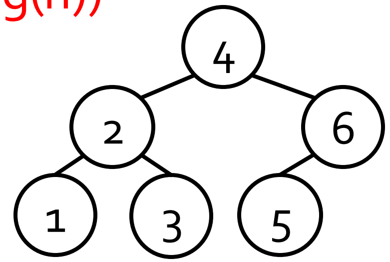
- Typically assume no duplicate elements
- If duplicates:
  - then store counts in nodes, or
  - each node has a list of objects

Searching in BSTs

- Time to search using a BST with n nodes is O(?)
  - For a BST of height h, it is O(h)
  - What is the value of h?
  - Worst-case: h=O(n)



  - Balanced tree: h=O(lg(n))

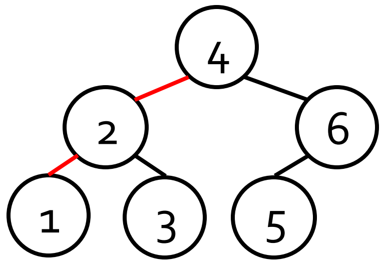


WSU

Searching in BSTs

- Finding the minimum element
    - Smallest element in left subtree

```
findMin(T) {
  if (T == NULL)
    then return NULL
  if (T->leftChild == NULL)
    then return T
  else return findMin(T->leftChild)
}
```

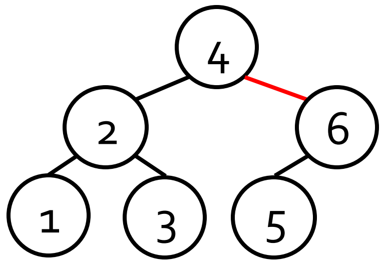


- Complexity?
    - O(h)

Searching in BSTs

- Finding the minimum element
  - Smallest element in left subtree

```
findMax(T) {
  if (T == NULL)
    then return NULL
  if (T->rightChild == NULL)
    then return T
  else return findMax(T->rightChild)
}
```



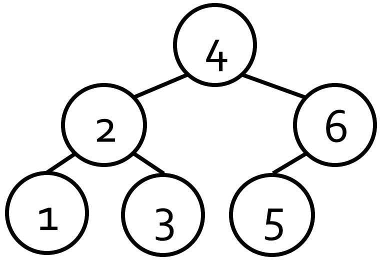
- Complexity?
  - O(h)

WSU

Printing BSTs

- In-order traversal ==> sorted

```
PrintTree(T) {
  if (T == NULL)
    then return
  PrintTree(T->leftChild)
  cout << T->element
  PrintTree(T->rightChild)
}
```

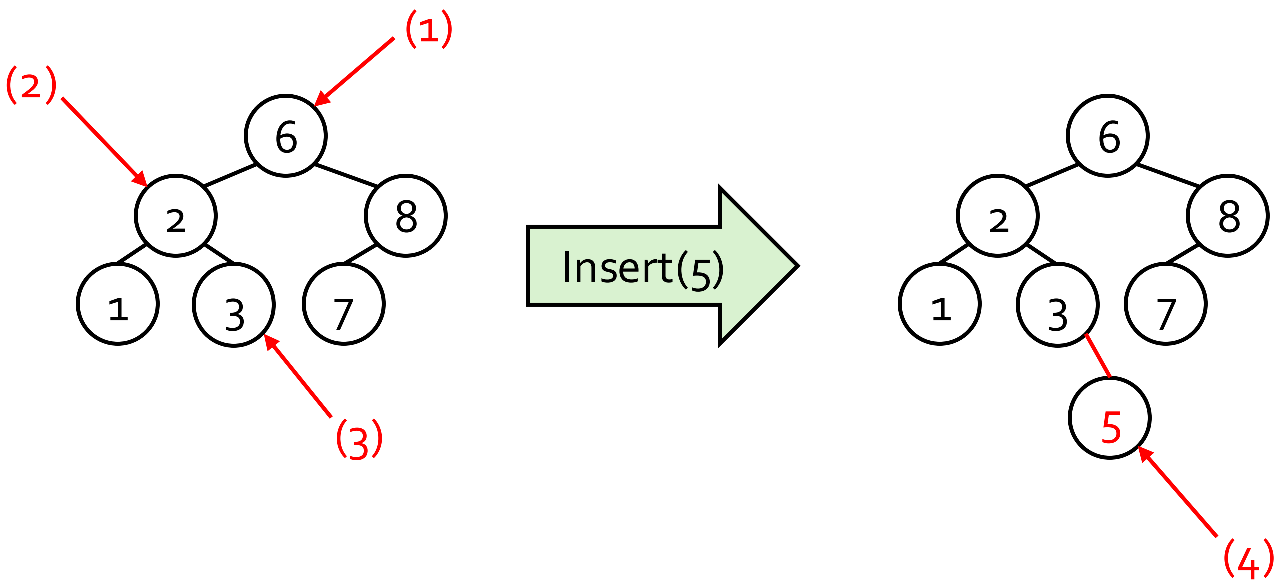


1 2 3 4 5 6

- Complexity?
  - Θ(n)

Inserting into BSTs

- e.g., insert 5



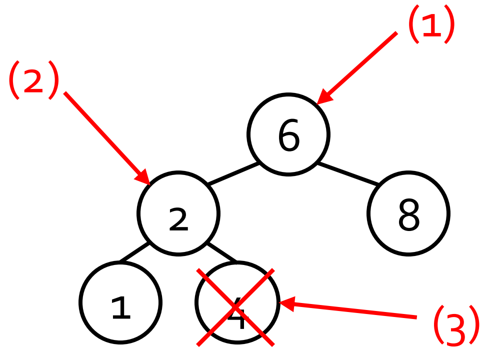
Inserting into BSTs

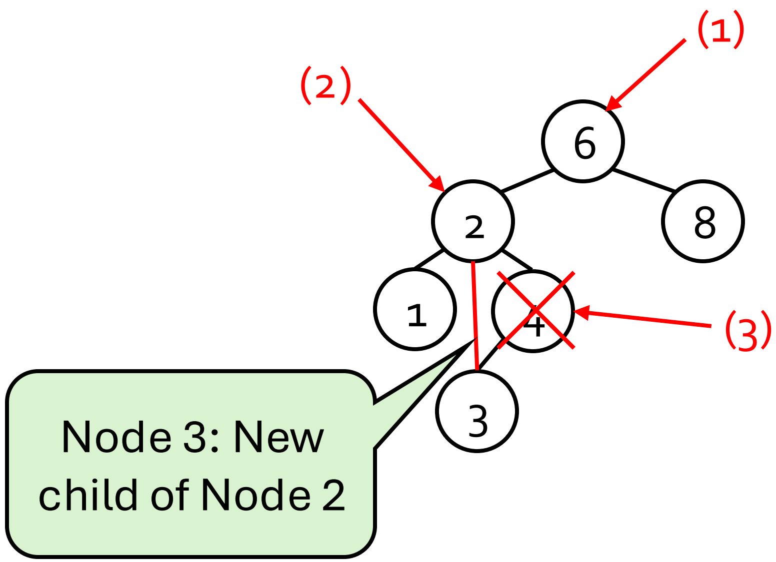
- Search for element until reach end of tree
- Insert new element there

```
Insert(x, T) {
  if (T == NULL)
    then T = new Node(x)
  else
    if (x < T->element)
      then if (T->leftChild == NULL)
             then T->leftChild = new Node(x)
           else Insert(x, T->leftChild)
    else if (T->rightChild == NULL)
           then (T->rightChild = new Node(x)
         else Insert(x, T->rightChild)
}
```

Removing from BSTs

- There are two cases for removal
- Case 1: Node to remove has 0 or 1 child
  - Action: remove it and make appropriate adjustments to retain BST structure
  - e.g., remove(4)

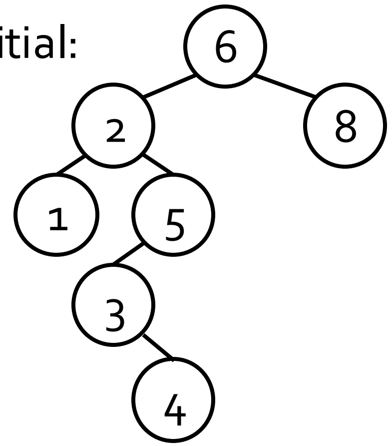


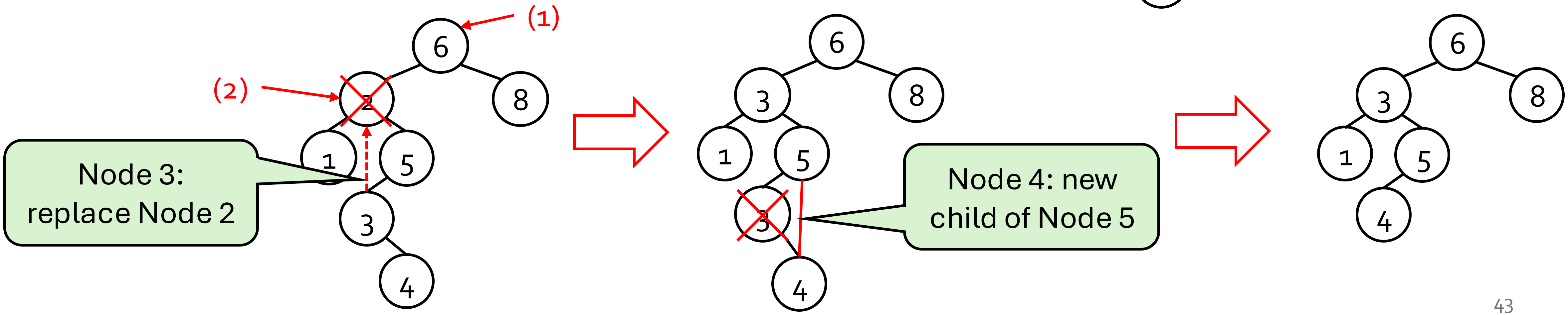


Removing from BSTs

- Case 2: Node to remove has 2 children
  - Action:
    - Replace node element with successor
    - Remove the successor (case 1)
  - e.g., remove(2)

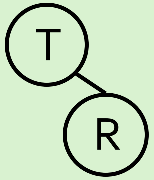
Initial:





Removing from BSTs

```
Remove(x, T) {
  if (T == NULL)
    then return
  if (x == T->element)
    then if ((T->left == NULL) && (T->right != NULL))
           then T = T->right
         else if ((T->right == NULL) && (T->left != NULL))
                then T = T->left
         else if ((T->right == NULL) && (T->left == NULL))
                then T = NULL
         else {
           successor = findMin(T->right)
           T->element = successor->element
           Remove(T->element, T->right)
         }
  else if (x < T->element)
         then Remove(x, T->left) // recursively search
       else Remove(x, T->right) // recursively search
}
```

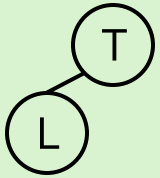


T: only right child

WSU

Removing from BSTs

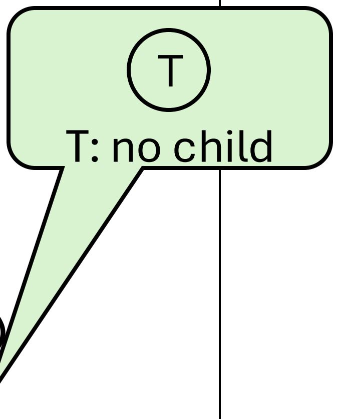
```
Remove(x, T) {
  if (T == NULL)
    then return
  if (x == T->element)
    then if ((T->left == NULL) && (T->right != NULL))
           then T = T->right
         else if ((T->right == NULL) && (T->left != NULL))
               then T = T->left
         else if ((T->right == NULL) && (T->left == NULL))
               then T = NULL
         else {
           successor = findMin(T->right)
           T->element = successor->element
           Remove(T->element, T->right)
         }
  else if (x < T->element)
         then Remove(x, T->left) // recursively search
       else Remove(x, T->right) // recursively search
}
```



T: only left child

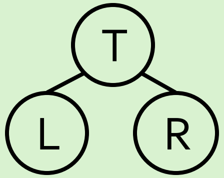
Removing from BSTs

```
Remove(x, T) {
  if (T == NULL)
    then return
  if (x == T->element)
    then if ((T->left == NULL) && (T->right != NULL))
           then T = T->right
         else if ((T->right == NULL) && (T->left != NULL))
                then T = T->left
         else if ((T->right == NULL) && (T->left == NULL))
                then T = NULL
         else {
           successor = findMin(T->right)
           T->element = successor->element
           Remove(T->element, T->right)
         }
  else if (x < T->element)
         then Remove(x, T->left) // recursively search
       else Remove(x, T->right) // recursively search
}
```



Removing from BSTs

```
Remove(x, T) {
  if (T == NULL)
    then return
  if (x == T->element)
    then if ((T->left == NULL) && (T->right != NULL))
           then T = T->right
         else if ((T->right == NULL) && (T->left != NULL))
                then T = T->left
         else if ((T->right == NULL) && (T->left == NULL))
                then T = NULL
         else {
           successor = findMin(T->right)
           T->element = successor->element
           Remove(T->element, T->right)
         }
  else if (x < T->element)
         then Remove(x, T->left) // recursively search
       else Remove(x, T->right) // recursively search
}
```



T: left and right children

(1) Replace node element with successor
(2) Remove the successor (case 1)

WSU

Implementation of BSTs

```
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 );
```

Figure 4.16 in textbook

Implementation of BSTs

```
  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;
};
```

Figure 4.16 in textbook

WSU

Implementation of BSTs

```
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```

Figure 4.17 in textbook

Tree: Binary Search Tree

Implementation of BSTs

```
/**
 * 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 in textbook

WSU

Implementation of BSTs

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

Figure 4.20 in textbook

WSU

Implementation of BSTs

```
/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```

Figure 4.21 in textbook

WSU

Implementation of BSTs

```
/**
 * 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 == nullptr )
        t = new BinaryNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ;  // Duplicate; do nothing
}
```

Figure 4.23 in textbook

Implementation of BSTs

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert by moving.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( Comparable && x, BinaryNode * & t )
{
    if( t == nullptr )
        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
    else if( x < t->element )
        insert( std::move( x ), t->left );
    else if( t->element < x )
        insert( std::move( x ), t->right );
    else
        ;  // Duplicate; do nothing
}
```

Figure 4.23 in textbook

Imp


```
/**
 * 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 == nullptr )
        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 != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Figure 4.26 in textbook

WSU

Implem...s

```
/**
 * Destructor for the tree
 */
~BinarySearchTree( )
{
    makeEmpty( );
}
/**
 * Internal method to make subtree empty.
 */
void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}
```

Figure 4.27 in textbook

WSU

BST analysis

- printTree, makeEmpty and operator=
  - Always $\Theta(n)$
- insert, remove, contains, findMin, findMax
  - O(h), where h = height of tree
- Worst case: h = ?
- Best case: h = ?
- Average case: h = ?

O(n)

Ω(lg(n))

Θ(lg(n))

WSU

BST average-case analysis

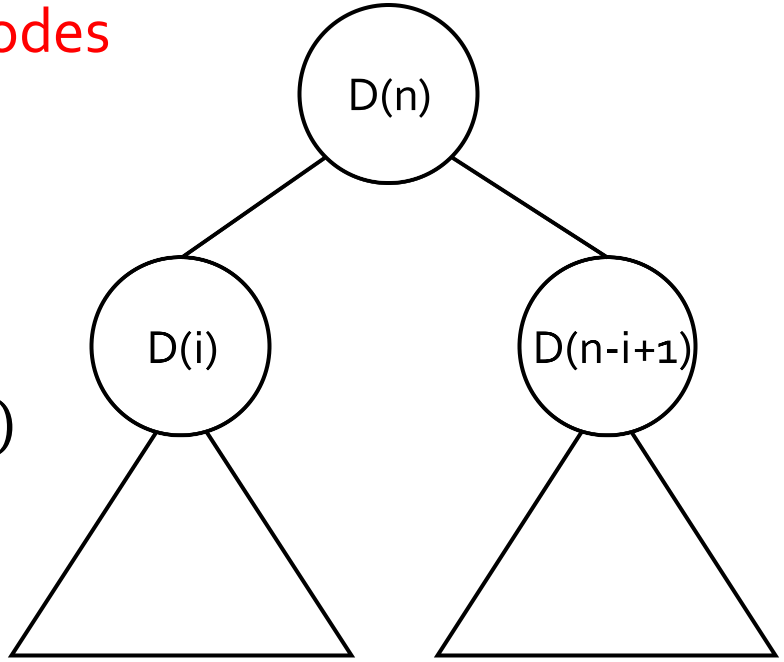
Average case: h

- Define "internal path length" of a tree:
  - = Sum of the depths of all nodes in the tree
  - Implies: average depth of a tree = (internal path length)/n
- But there are lots of trees possible (one for every unique insertion sequence)
  - Compute average internal path length over all possible insertion sequences
  - It has been proven that when a binary search tree is constructed through a random sequence of insertions, avg depth of a node lg(n)
  - $n*lg(n)/n = \Theta(lg(n))$

see textbook Section 4.3.6 and 7.7.5 (Analysis of Quicksort, average-case analysis)

Average internal path length

- Let D(n) = internal path length for a tree with n nodes
- = D(left) + D(right) + D(root)
- = D(i) + D(n-i-1) + n-1
- If all tree sizes are equally likely,
- → average D(i) = average D(n-i-1) = 1/n $\sum_{j=0}^{n-1} D(j)$
- → average D(n) = 2/n $\sum_{j=0}^{n-1} D(j)$ + n-1
- → O(n lg(n))



WSU

Randomly Generated BST

Randomly inserting 500 nodes

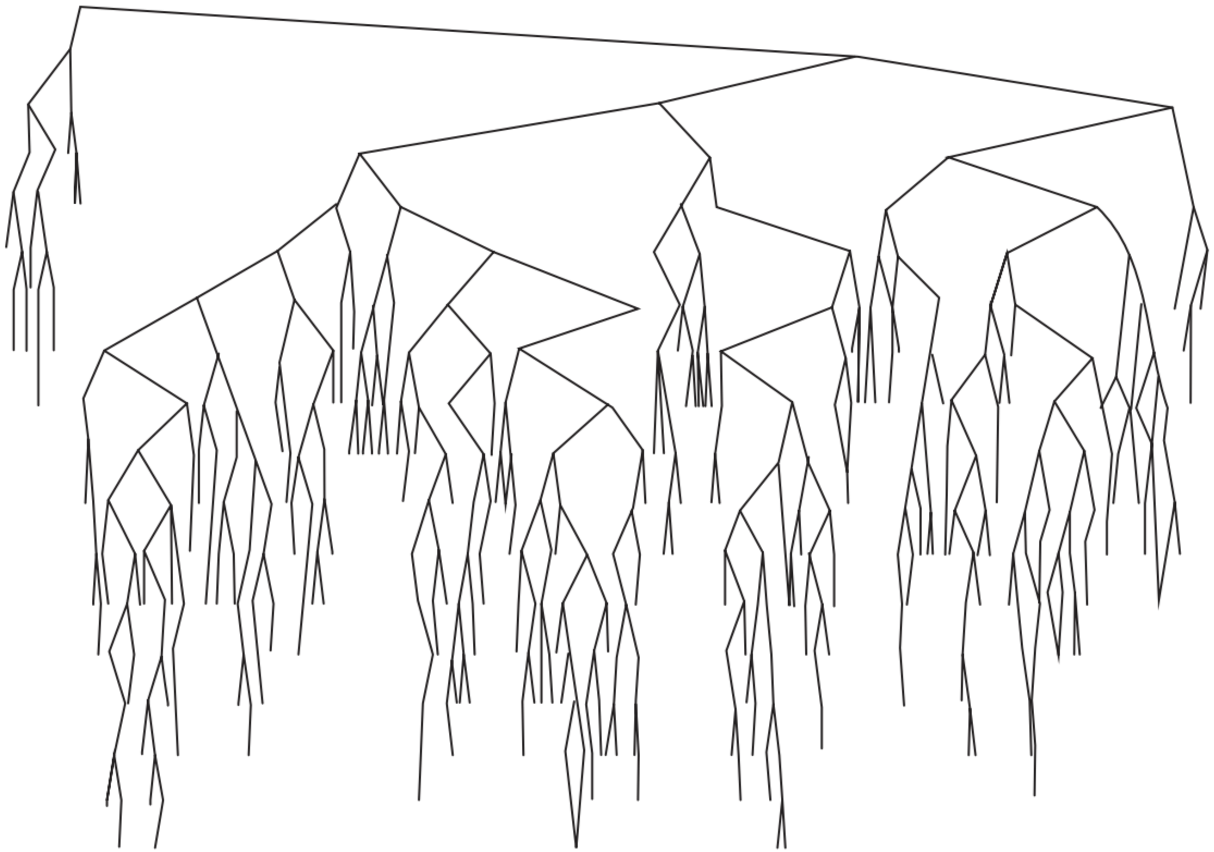


Figure 4.29 A randomly generated binary search tree

Average node depth = 9.98
log2 500 = 8.97

Randomly Generated BST

Randomly inserting/removing 500^2 nodes

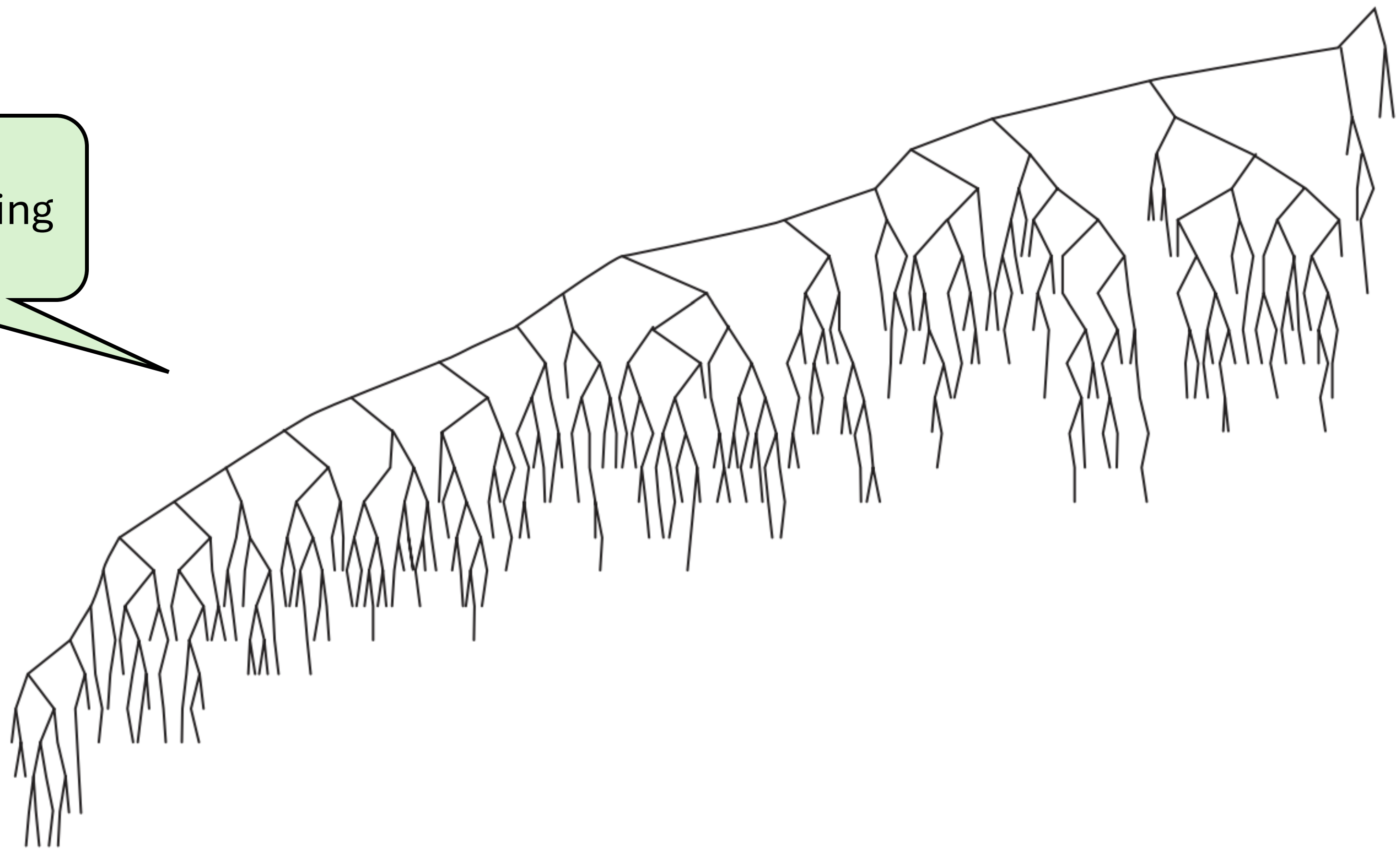


Figure 4.30 Binary search tree after $\Theta(N^2)$ `insert/remove` pairs

Average node depth = 12.51