Discussion Session 6: Trees

CS 225: Data Structures
& Software Principles

Agenda

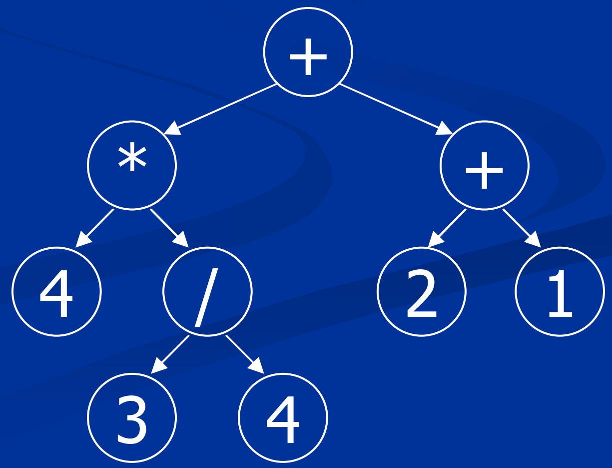
- **Trees**
- Binary Search Trees
- Basic BST operations
- Implementations of Trees

By the end of this class, you

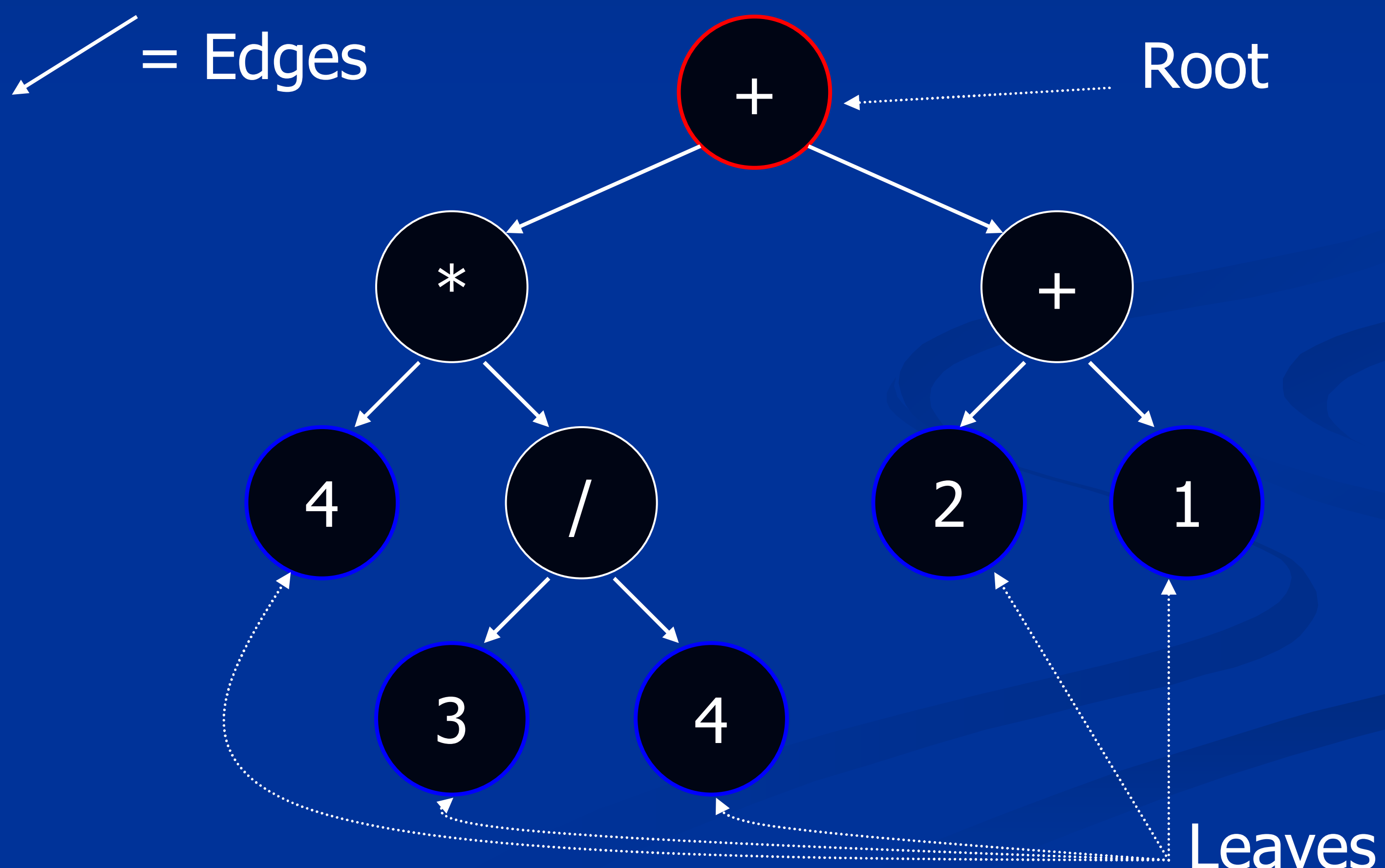
- Need to
  - Understand tree terminology
  - Follow different tree traversals
  - Understand all the BST operations
- Ought to be able to implement a BST
- Might want to think about iterative versions of recursive functions.

Trees

- What is a Tree structure ?
- Some Tree terminology
  - Nodes and Edges
  - Root and Leaf
  - Parent, Child, Descendant and Ancestor
  - Height, Depth
  - Sub-trees and Forests

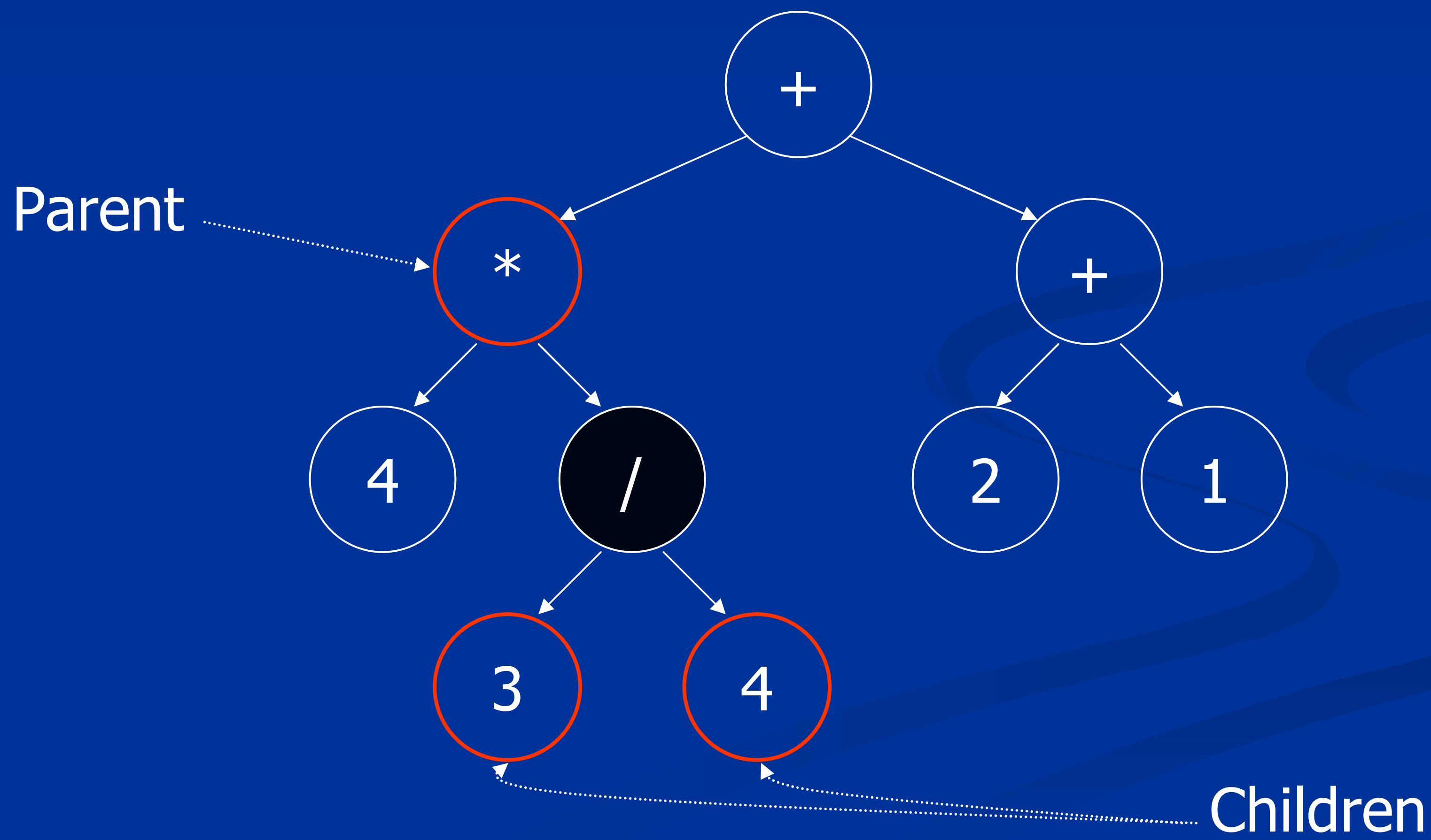


Nodes, Root & Leaves



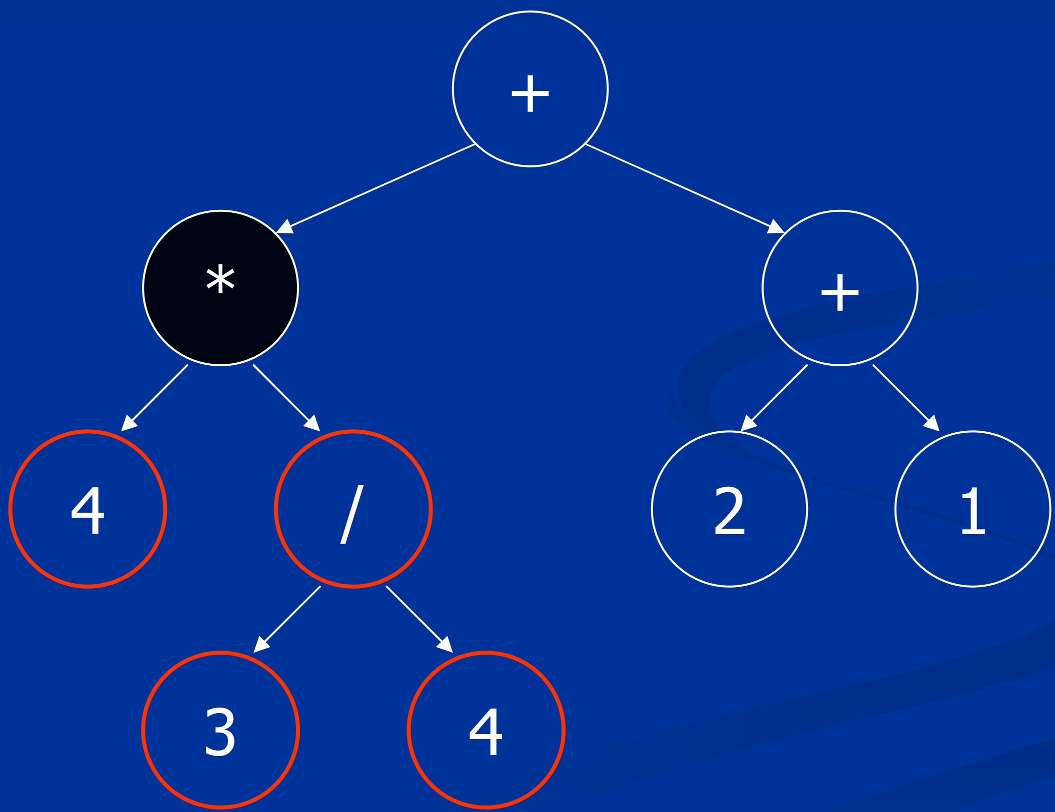
Parent & Children

- Consider the node (/)



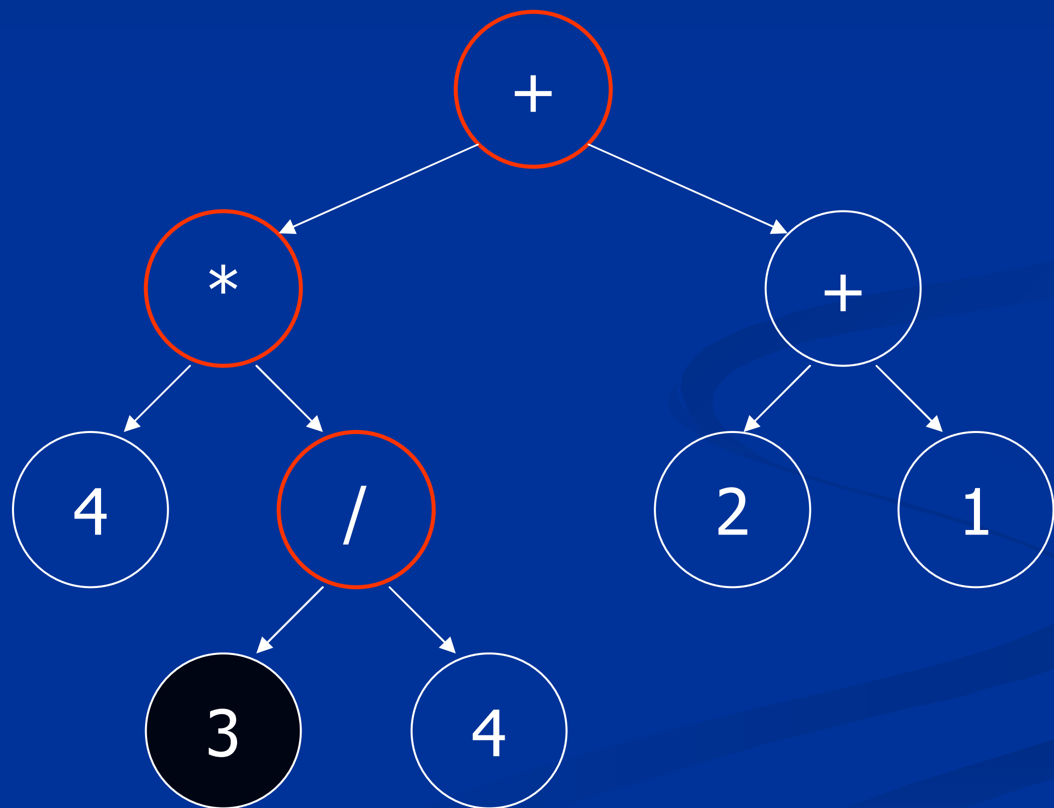
Descendants

- Consider the node (*)



Ancestors

- Consider the node (3)

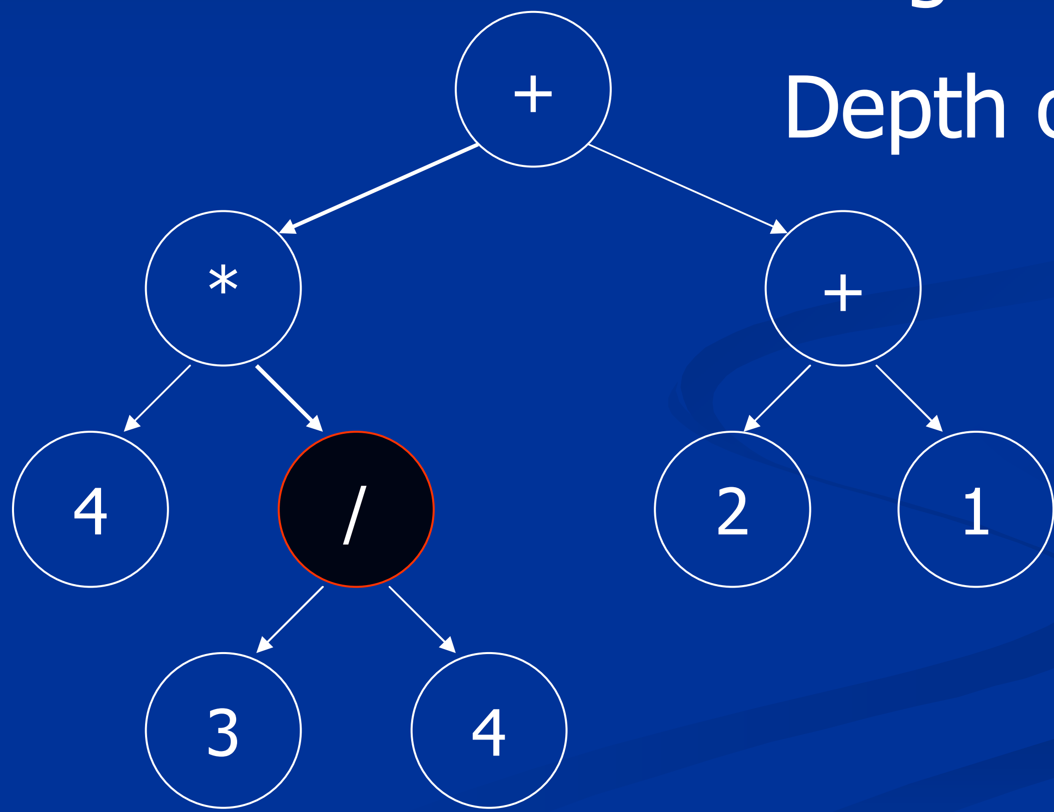


Height & Depth

- Consider the node (/)

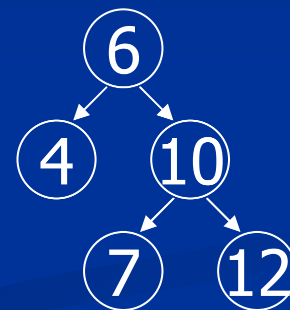
Height (of tree): 3

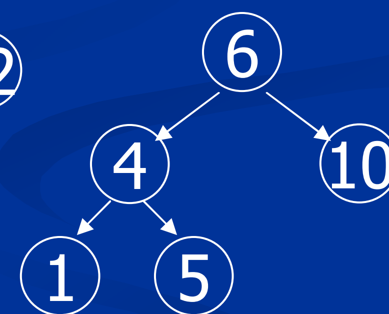
Depth of node: 2

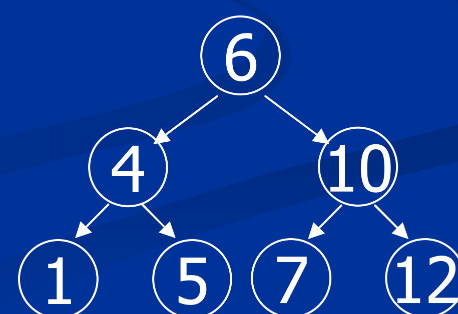


Some Types of Trees

- Ordered trees
- Binary trees
  - Empty binary tree
  - Full binary tree
    - Every node has 0 or 2 children
  - Complete binary tree
    - Every level filled except possibly…
    - …bottom level. Has all nodes as far left as possible
  - Perfect binary tree
    - Leaves have same depth
    - Internal nodes are degree 2







Tree Traversals

- Pre-Order Traversal
- In-Order Traversal
- Post-Order Traversal

- Level-Order Traversal
  - Use a queue

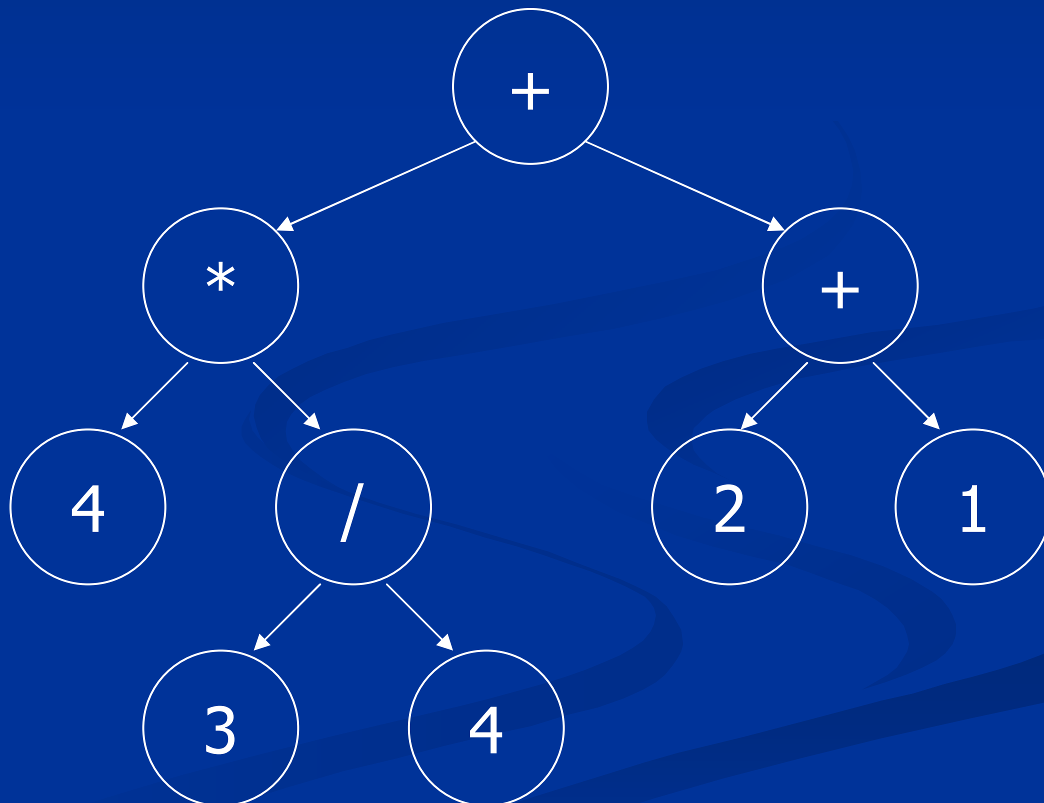
- Future Topic: Graph Traversals

Where does DoStuff(node) go?

```
void Traverse(node)
{
  if (node == null) return;
  // pre
  Traverse(node→left);
  // in
     Traverse(node→right);
  // post
}
```

Traversal Example

- Preorder:
- Postorder:
- Inorder:
- Level-order:



Traversal Example

- Preorder:

  +*4/34+21

- Postorder:

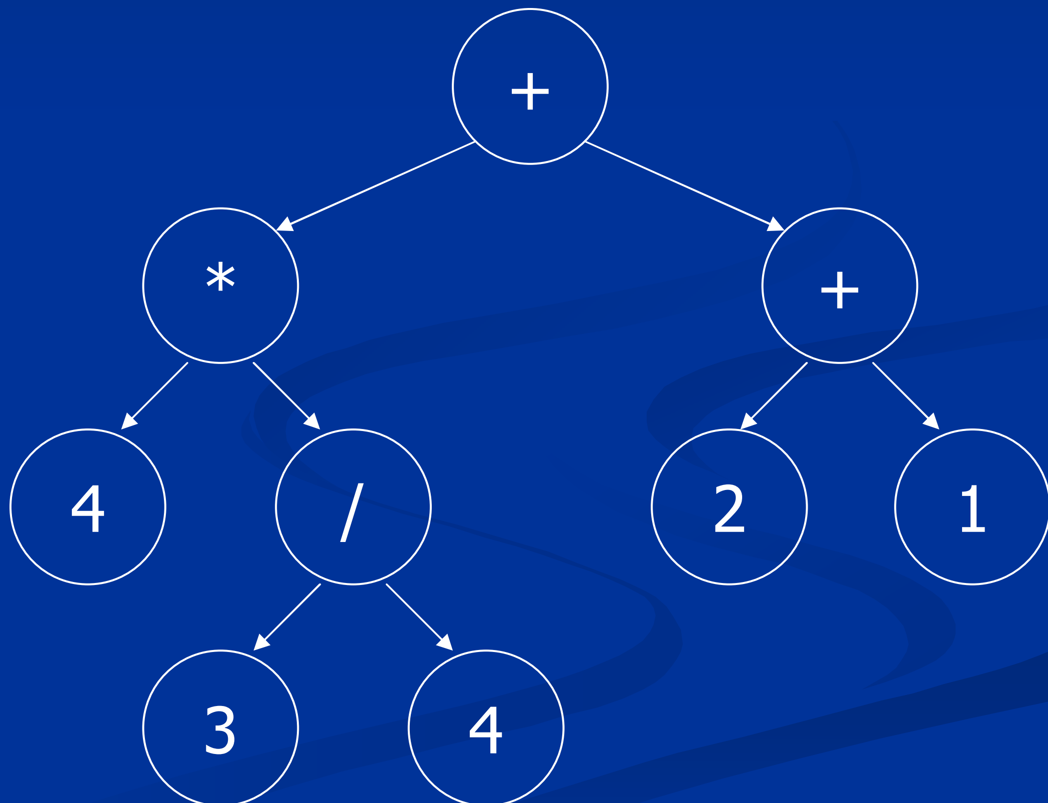
  434/*21++

- Inorder:

  4*3/4+2+1

- Level-order:

  +*+4/2134



Binary Trees

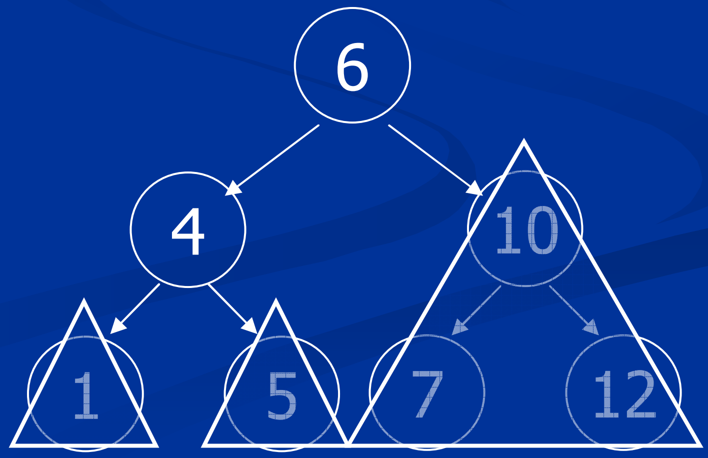
- Height of a complete binary tree with n nodes is exactly $\lfloor \lg n \rfloor$
- The maximum height binary tree with n nodes has a height n-1
- The minimum height binary tree with n nodes has height $\lfloor \lg n \rfloor$

Agenda

- Trees
- **Binary Search Trees**
- Basic BST operations
- Implementations of Trees

Binary Search Trees

- **Definition :**
  - Have a value associated with each node
  - the values have a linear order
  - Every node has a value greater than any value in the left sub-tree and less than any value in the right sub-tree.
- Abbreviated as BST

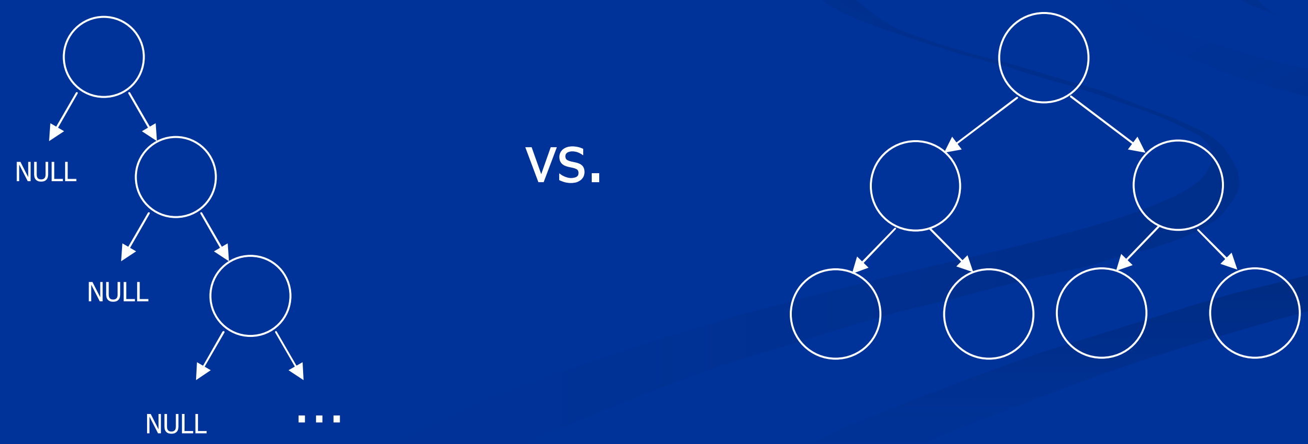


Binary Search Trees

- The worst-case search time
  - for all possible search trees with *n* nodes is

    $\Theta(n)$
  - for the best search tree with *n* nodes is

    $\Theta(\lg n)$



Agenda

- Trees
- Binary Search Trees
- **Basic BST operations**
- **Implementations of Trees**

One Implementation: use Tree Nodes

```
// private, nested in BSTree class
class TreeNode {
  public:
   TreeNode( ) : element( ), left(NULL), right(NULL) {}
   TreeNode( Etype elmt,
         TreeNode* leftPtr = NULL,
         TreeNode* rightPtr = NULL ) :
               element(elmt), left(leftPtr), right(rightPtr) { }

    Etype element;         // element of node
    TreeNode* left;        // pointer to left subtree
    TreeNode* right;       // pointer to right subtree
};
```

Basic BST Operations

- Find
  - Recursive implementation
  - Iterative implementation
- Insert
- Remove

Basic BST Operations: Find

- Recursive Find Algorithm

```
treePtr Find(treePtr P, key K)
{
   if ( P == NULL)
         return NULL
   else if ( K == P→key )
         return P
   else if ( K < P→key )
         return Find(P→LeftChild, K)
   else
         return Find(P→RightChild, K)
}
```

Basic BST Operations: Find

- Iterative Find Algorithm

```
treePtr Find(treePtr P, key K)
{
   while ( P != NULL) {
        if ( K == P→key )
              return P
        else if ( K < P→key )
              P = P→LeftChild
        else
              P = P→RightChild
   }
   return NULL
}
```

Basic BST Operations: Insert

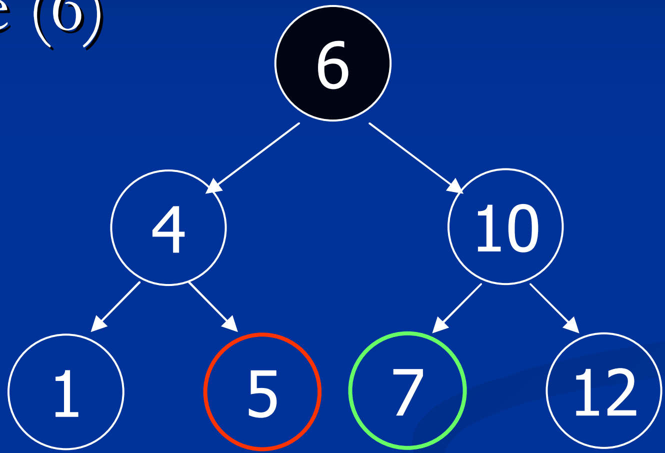
- Insertion
  - Must ensure that tree remains a binary search tree after insertion
  - Determine where the element would have been if it were actually in the BST. Insert there.
  - Compare Insert( ) vs. Find( )

Basic BST Operations: Remove

- Remove
  - More tricky than Insertion
  - First find node with element to remove
  - Split into three cases
    - Node to be deleted is a leaf
    - Node to be deleted has one child
    - Node to be deleted has two children

Terminology for Remove

- Consider root node (6)



- In-order predecessor: greatest (right-most) element in left subtree
- In-order successor: smallest (left-most) element in right subtree

Basic BST Operations: Remove

- Leaf case
  - Simply delete the node
- One-child case
  - Just connect the node's child to it's parent
- Two-child case
  - Replace the node by it's in-order successor and delete the in-order successor
  - Alternatively, we could use also the in-order predecessor

Basic BST Operations: Remove

- Two-child case
  - Replace node with in-order successor (predecessor) and delete the in-order successor (predecessor)

```
typename BSTree<Etype>::TreeNode* tempPtr;
if ((ptr->left != NULL) && (ptr->right != NULL)) {
   // Replace with smallest in right subtree.
   tempPtr = ptr->right;
   while (tempPtr->left != NULL)
         tempPtr = tempPtr->left;
   ptr->element = tempPtr->element;
   Remove(ptr->right, ptr->element);
}
```

Basic BST Operations: Remove

- Leaf case
  - Simply delete the node

```
else if ((ptr->left == NULL) && (ptr->right == NULL))
{
        delete ptr;
        ptr = NULL;
}
```

Basic BST Operations: Remove

- One-child case
  - Just connect the node's child to it's parent

```
else
{
   tempPtr = ptr;
   if (ptr->left == NULL)          // only a right child
        ptr = ptr->right;
   else // ptr->right == NULL  // only a left child
        ptr = ptr->left;
   delete tempPtr;
}
```

BST Operation: Clear

- Used in public interface, destructor, operator=
- How is Jason traversing the tree?

```
// recursively clears the tree
template <class Etype>
void BSTree<Etype>::Clear(
        typename BSTree<Etype>::TreeNode*& ptr) {
   if (ptr != NULL) {
        Clear(ptr->left);
        Clear(ptr->right);
        delete ptr;
        ptr = NULL;
   }
}
```

BST Operation: Copy

- Used in copy constructor & operator=
- How is Jason traversing the tree?

```
// makes a new TreeNode which is a copy of the parameter node, and returns it
template <class Etype>
typename BSTree<Etype>::TreeNode*
BSTree<Etype>::Copy(typename BSTree<Etype>::TreeNode const * ptr) {
    if (ptr != NULL) {
            typename BSTree<Etype>::TreeNode * temp =
                    new typename BSTree<Etype>::TreeNode(ptr->element);
            temp->left = Copy(ptr->left);
            temp->right = Copy(ptr->right);
            return temp;
    }
    else return NULL;
}
```

Implementation Detail

- Recursive operations have 2 versions:
  - Public interface version
  - Private interface version with extra TreeNode parameter
- Examples:

```
void Insert(Etype const& insElem) { Insert(root, insElem); }
void Insert(typename BSTree<Etype>::TreeNode * & ptr,
                                    Etype const & insElem);


void PreOrder( ) const { PreOrder(root); }
void PreOrder(typename BSTree<Etype>::TreeNode const * ptr)
    const;
```

Sample Code for BST

The entire code is available in

~cs225/src/library/06-bst/_latestBST

The BST Applet is available at

www.cs.jhu.edu/~goodrich/dsa/trees/btree.html