A binary tree is a tree with the following properties:
- Each internal node has at most two children (exactly two for proper binary trees).
- The children of a node are an ordered pair.

We call the children of an internal node left child and right child.

Alternative recursive definition: a binary tree is either
- a tree consisting of a single node, or
- a tree whose root has an ordered pair of children, each of which is a binary tree.

Applications:
- arithmetic expressions
- decision processes
- searching
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands

- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

```
Want a fast meal?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>How about coffee?</td>
<td>On expense account?</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Starbucks</td>
<td>Spike's</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Al Forno</td>
<td>Cafe Paragon</td>
</tr>
</tbody>
</table>
```

© 2010 Goodrich, Tamassia  Trees
Properties of Proper Binary Trees

- **Notation**
  - $n$: number of nodes
  - $e$: number of external nodes
  - $i$: number of internal nodes
  - $h$: height

- **Properties:**
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

© 2010 Goodrich, Tamassia Trees
Inorder Traversal

Each node is visited after its left subtree and before its right subtree.

Algorithm \text{inOrder}(v)

\begin{align*}
\text{if } & \neg v.\text{isExternal}() \\
\text{inOrder}(v.\text{left}()) & \\
\text{visit}(v) & \\
\text{if } & \neg v.\text{isExternal}() \\
\text{inOrder}(v.\text{right}()) & 
\end{align*}
Printing Arithmetic Expressions

- Perform inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

Algorithm `printExpression(v)`

```
if ¬v.isExternal()
  print("(")
  inOrder(v.left())
  print(v.element())
  if ¬v.isExternal()
    inOrder(v.right())
  print (")")
```

(((2 × (a − 1)) + (3 × b))

© 2010 Goodrich, Tamassia

Trees
A node is represented by an object storing:
- Element
- Parent node
- List of children nodes

© 2010 Goodrich, Tamassia

Trees
A node is represented by an object storing:
- Element
- Parent node
- Left child node
- Right child node
$ git fetch origin

$ git checkout -b week12 origin/week12
```cpp
#include <iostream>
#include "LinkedBinaryTree.h"

int main()
{
    /* create the following binary tree:
       9
       / \
      7   11
     / \ / \
    3  8 10 14
     / \
    2   5
    */

    LinkedBinaryTree<int> tree;
    tree.addRoot(9);
    tree.addLeftLeaf(tree.root(), 7);
    tree.addRightLeaf(tree.root(), 11);
    tree.addLeftLeaf(tree.root().left(), 3);
    tree.addRightLeaf(tree.root().left(), 8);
    tree.addLeftLeaf(tree.root().right(), 10);
    tree.addRightLeaf(tree.root().right(), 14);
    tree.addLeftLeaf(tree.root().left().left(), 2);
    tree.addRightLeaf(tree.root().left().left(), 5);

    cout << "Preorder:" << endl;
    tree.preorderPrint();

    cout << endl << "Inorder:" << endl;
    tree.inorderPrint();

    cout << endl << "Postorder:" << endl;
    tree.postorderPrint();

    return 0;
}
```
We begin by defining the basic constituents that make up the LinkedBinaryTree class. The most basic entity is the structure Node, shown in Code Fragment 7.17, that represents a node of the tree.

```c
struct Node {
    Elem   elt;   // element value
    Node*  par;  // parent
    Node*  left; // left child
    Node*  right; // right child
    Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
};
```

**Code Fragment 7.17:** Structure Node implementing a node of a binary tree. It is nested in the protected section of class BinaryTree.
```cpp
#ifndef LINKEDBINARYTREE_H
#define LINKEDBINARYTREE_H

#include <iostream>
#include <list>

using namespace std;

template <typename Object>
class LinkedBinaryTree {

protected:
    // Node declaration
    struct Node {
        Object elem;   // element value
        Node* parent;  // parent
        Node* left;    // left child
        Node* right;   // right child
        Node() : elem(NULL), parent(NULL), left(NULL), right(NULL) {}    // constructor
        Node(Object& e) : elem(e), parent(NULL), left(NULL), right(NULL) {} };
    
};
```
Next, we define the public class Position in Code Fragment 7.18. Its data member consists of a pointer v to a node of the tree. Access to the node’s element is provided by overloading the dereferencing operator (“*”). We declare LinkedBinaryTree to be a friend, providing it access to the private data.

```cpp
class Position {
private:
    Node* v; // pointer to the node
public:
    Position(Node* _v = NULL) : v(_v) { } // constructor
    Elem& operator*() // get element
    {
        return v->elt;
    }
    Position left() const // get left child
    {
        return Position(v->left);
    }
    Position right() const // get right child
    {
        return Position(v->right);
    }
    Position parent() const // get parent
    {
        return Position(v->par);
    }
    bool isRoot() const // root of the tree?
    {
        return v->par == NULL;
    }
    bool isExternal() const // an external node?
    {
        return v->left == NULL && v->right == NULL;
    }
friend class LinkedBinaryTree; // give tree access
};

typedef std::list<Position> PositionList; // list of positions
```

**Code Fragment 7.18:** Class Position implementing a position in a binary tree. It is nested in the public section of class LinkedBinaryTree.
Binary Tree Implementation

```cpp
public:
    // Position declaration
    class Position { // position in the tree
        private:
            Node* v; // pointer to the Node
    public:
        Position(Node* _v = NULL) : v(_v) { } // constructor
        Object& operator*() { return v->elem; } // get element
        Position left() const { return Position(v->left); } // get left child
        Position right() const { return Position(v->right); } // get right child
        Position parent() const { return Position(v->parent); } // get parent
        bool isRoot() const { return v->parent == NULL; } // root of the tree?
        bool isExternal() const { return v->left == NULL && v->right == NULL; } // an external Node?
        bool isNULL() { return (v == NULL); } // check if position is null
        friend class LinkedBinaryTree;
    }
```
We present the major part of the class `LinkedBinaryTree` in Code Fragment 7.19. The class declaration begins by inserting the above declarations of `Node` and `Position`. This is followed by a declaration of the public members, local utility functions, and the private member data. We have omitted housekeeping functions, such as a destructor, assignment operator, and copy constructor.

```cpp
typedef int Elem; // base element type

class LinkedBinaryTree {

protected:
    // insert Node declaration here...

public:
    // insert Position declaration here...

public:
    LinkedBinaryTree(); // constructor
    int size() const; // number of nodes
    bool empty() const; // is tree empty?
    Position root() const; // get the root
    PositionList positions() const; // list of nodes
    void addRoot(); // add root to empty tree
    void expandExternal(const Position& p); // expand external node
    Position removeAboveExternal(const Position& p); // remove p and parent
    // housekeeping functions omitted...

protected:
    void preorder(Node* v, PositionList& pl) const; // preorder utility

private:
    Node* _root; // pointer to the root
    int n; // number of nodes
};
```

**Code Fragment 7.19:** Implementation of a `LinkedBinaryTree` class.
public:

    LinkedBinaryTree(); // constructor
    int size() const; // number of Nodes
    bool empty() const; // is tree empty?
    Position root() const; // get the root
    void addRoot(const Object& value=Object()); // add root to empty tree
    void addLeftLeaf(const Position& p, const Object& value); // add left leaf to the tree
    void addRightLeaf(const Position& p, const Object& value); // add left leaf to the tree
    void preorderPrint() const; // print nodes in preorder
    void inorderPrint() const; // print nodes in inorder
    void postorderPrint() const; // print nodes in postorder
    void expandExternal(const Position& p); // expand external Node
    Position removeAboveExternal(const Position& p); // remove p and parent

protected: // local utilities

private:

    void preorder(const Node* v) const; // preorder utility
    void inorder(const Node* v) const; // inorder utility
    void postorder(const Node* v) const; // postorder utility
    Node* _root; // pointer to the root
    int n; // number of Nodes

#include "LinkedBinaryTree.cpp"
#endif
Implement Functions

```
LinkedListBinaryTree::LinkedListBinaryTree()
    : _root(NULL), n(0) {} // constructor
int LinkedListBinaryTree::size() const
    { return n; } // number of nodes
bool LinkedListBinaryTree::empty() const
    { return size() == 0; } // is tree empty?
LinkedListBinaryTree::Position LinkedListBinaryTree::root() const // get the root
    { return Position(_root); }
void LinkedListBinaryTree::addRoot()
    { _root = new Node; n = 1; } // add root to empty tree
```

**Code Fragment 7.20:** Simple member functions for class LinkedListBinaryTree.
template<typename Object>
LinkedBinaryTree<Object>::LinkedBinaryTree()  // constructor
  : _root(NULL), n(0) {}

template<typename Object>
int LinkedBinaryTree<Object>::size() const  // number of nodes
{
  return n;
}

template<typename Object>
bool LinkedBinaryTree<Object>::empty() const  // is tree empty?
{
  return size() == 0;
}

template<typename Object>
typename LinkedBinaryTree<Object>::Position LinkedBinaryTree<Object>::root() const  // get the root
{
  return Position(_root);
}

template<typename Object>
void LinkedBinaryTree<Object>::addRoot(const Object& value)  // add root to empty tree
{
  _root = new Node;
  _root->elem = value;
  n = 1;
}

template<typename Object>
void LinkedBinaryTree<Object>::addLeftLeaf(const Position& p, const Object& value) {
  Node* v = p.v;  // p's node
  v->left = new Node;  // add a new left child
  v->left->elem = value;  // v is its parent
  v->left->parent = v;  // v is its parent
  n++;
  // one more node
}
Preorder Traversal of a Binary Tree

**Algorithm** preorder \((T, p)\):

- perform the “visit” action for node \(p\)
  - for each child \(q\) of \(p\) do
    - recursively traverse the subtree rooted at \(q\) by calling preorder \((T, q)\)

**Code Fragment 7.9:** Algorithm preorder for performing the preorder traversal of the subtree of a tree \(T\) rooted at a node \(p\).

**Preorder Traversal of a Binary Tree**

Since any binary tree can also be viewed as a general tree, the preorder traversal for general trees (Code Fragment 7.9) can be applied to any binary tree. We can simplify the algorithm in the case of a binary-tree traversal, however, as we show in Code Fragment 7.24. (Also see Code Fragment 7.23.)

**Algorithm** binaryPreorder \((T, p)\):

- perform the “visit” action for node \(p\)
  - if \(p\) is an internal node then
    - binaryPreorder \((T, p.left())\) \{recursively traverse left subtree\}
    - binaryPreorder \((T, p.right())\) \{recursively traverse right subtree\}

**Code Fragment 7.24:** Algorithm binaryPreorder, which performs the preorder traversal of the subtree of a binary tree \(T\) rooted at node \(p\).
Algorithm binaryPostorder(T, p):
    if p is an internal node then
        binaryPostorder(T, p.left()) \{recursively traverse left subtree\}
        binaryPostorder(T, p.right()) \{recursively traverse right subtree\}
    perform the “visit” action for the node p

Code Fragment 7.25: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node p.
Inorder Traversal of a Binary Tree

An additional traversal method for a binary tree is the **inorder** traversal. In this traversal, we visit a node between the recursive traversals of its left and right subtrees. The inorder traversal of the subtree rooted at a node $p$ in a binary tree $T$ is given in Code Fragment 7.27.

**Algorithm** inorder($T, p$):
- if $p$ is an internal node then
  - inorder($T, p\text{.left()}$) \hspace{10mm} \text{\{recursively traverse left subtree\}}
  - perform the “visit” action for node $p$
- if $p$ is an internal node then
  - inorder($T, p\text{.right()}$) \hspace{10mm} \text{\{recursively traverse right subtree\}}

**Code Fragment 7.27:** Algorithm inorder for performing the inorder traversal of the subtree of a binary tree $T$ rooted at a node $p$.

For example, an inorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order ⟨ATL, BWI, JFK, LAX, PVD⟩. The inorder traversal of a binary tree $T$ can be informally viewed as visiting the nodes of $T$ “from left to right.” Indeed, for every node $p$, the inorder traversal visits $p$ after all the nodes in the left subtree of $p$ and before all the nodes in the right subtree of $p$. (See Figure 7.18.)

![Figure 7.18: Inorder traversal of a binary tree.](image)


```cpp
30 template <typename Object>
31 void LinkedBinaryTree<Object>::addLeftLeaf(const Position& p, const Object& value) {
32     Node* v = p.v; // p's node
33     v->left = new Node; // add a new left child
34     v->left->elem = value; // v is its parent
35     v->left->parent = v; // v is its parent
36     n++; // one more node
37 }

40 template <typename Object>
41 void LinkedBinaryTree<Object>::addRightLeaf(const Position& p, const Object& value) {
42     Node* v = p.v; // p's node
43     v->right = new Node; // add a new right child
44     v->right->elem = value; // v is its parent
45     v->right->parent = v; // v is its parent
46     n++; // one more node
47 }
```
template <typename Object>
void LinkedBinaryTree<Object>::preorderPrint() const {
    preorder(_root);
}

template <typename Object>
void LinkedBinaryTree<Object>::preorder(const Node* v) const {
    if (v == NULL) return;
    cout << v->elem << endl;
    preorder(v->left);
    preorder(v->right);
}

template <typename Object>
void LinkedBinaryTree<Object>::inorderPrint() const {
    inorder(_root);
}

template <typename Object>
inorder(const Node* v) const {
    if (v == NULL) return;
    inorder(v->left);
    cout << v->elem << endl;
    inorder(v->right);
}

template <typename Object>
void LinkedBinaryTree<Object>::postorderPrint() const {
    postorder(_root);
}

template <typename Object>
void LinkedBinaryTree<Object>::postorder(const Node* v) const {
    if (v == NULL) return;
    postorder(v->left);
    postorder(v->right);
    cout << v->elem << endl;
}
Implement Functions

In addition to the BinaryTree interface functions and addRoot, the class LinkedBinaryTree also includes the following update functions given a position $p$. The first is used for adding nodes to the tree and the second is used for removing nodes.

$\text{expandExternal}(p)$: Transform $p$ from an external node into an internal node by creating two new external nodes and making them the left and right children of $p$, respectively; an error condition occurs if $p$ is an internal node.

$\text{removeAboveExternal}(p)$: Remove the external node $p$ together with its parent $q$, replacing $q$ with the sibling of $p$ (see Figure 7.15, where $p$’s node is $w$ and $q$’s node is $v$); an error condition occurs if $p$ is an internal node or $p$ is the root.

Figure 7.15: Operation $\text{removeAboveExternal}(p)$, which removes the external node $w$ to which $p$ refers and its parent node $v$. 
Implement Functions

The function expandExternal(p) is shown in Code Fragment 7.21. Letting v be p’s associated node, it creates two new nodes. One becomes v’s left child and the other becomes v’s right child. The constructor for Node initializes the node’s pointers to NULL, so we need only update the new node’s parent links.

```cpp
void LinkedBinaryTree::expandExternal(const Position& p) {
    Node* v = p.v;
    v->left = new Node;
    v->left->par = v;
    v->right = new Node;
    v->right->par = v;
    n += 2;
}
```

// expand external node
// p's node
// add a new left child
// v is its parent
// and a new right child
// v is its parent
// two more nodes

Code Fragment 7.21: The function expandExternal(p) of class LinkedBinaryTree.
The function `removeAboveExternal(p)` is shown in Code Fragment 7.22. Let \( w \) be \( p \)'s associated node and let \( v \) be its parent. We assume that \( w \) is external and is not the root. There are two cases. If \( w \) is a child of the root, removing \( w \) and its parent (the root) causes \( w \)'s sibling to become the tree’s new root. If not, we replace \( w \)'s parent with \( w \)'s sibling. This involves finding \( w \)'s grandparent and determining whether \( v \) is the grandparent’s left or right child. Depending on which, we set the link for the appropriate child of the grandparent. After unlinking \( w \) and \( v \), we delete these nodes. Finally, we update the number of nodes in the tree.

```cpp
LinkedBinaryTree::Position
LinkedBinaryTree::removeAboveExternal(const Position& p) {
    Node* w = p.v; Node* v = w->par;  // get p's node and parent
    Node* sib = (w == v->left ? v->right : v->left);
    if (v == _root) {  // child of root?
        _root = sib;  // ... make sibling root
        sib->par = NULL;
    }
    else {
        Node* gpar = v->par;  // w's grandparent
        if (v == gpar->left) gpar->left = sib;  // replace parent by sib
        else gpar->right = sib;
        sib->par = gpar;
    }
    delete w; delete v;  // delete removed nodes
    n -= 2;  // two fewer nodes
    return Position(sib);
}
```

**Code Fragment 7.22:** An implementation of the function `removeAboveExternal(p)`. 
template <typename Object>
void LinkedBinaryTree<Object>::expandExternal(const Position& p) // expand external node
{
    Node* v = p.v; // p's node
    v->left = new Node; // add a new left child
    v->left->parent = v; // v is its parent
    v->right = new Node; // and a new right child
    v->right->parent = v; // v is its parent
    n -= 2; // two more nodes
}

template <typename Object>
type Name LinkedBinaryTree<Object>::removeAboveExternal(const Position& p) // remove p and parent
{
    Node* w = p.v; Node* v = w->parent; // get p's node and parent
    Node* sib = (w == v->left ? v->right : v->left);
    if (v == _root) { // child or root
        _root = sib; // ...make sibling root
    } else {
        Node* gpar = v->parent;
        if (v == gpar->left)
            gpar->left = sib; // replace parent by sib
        else
            gpar->right = sib;
        sib->par = gpar;
    }
    delete w; delete v; // delete removed nodes
    n -= 2; // two fewer nodes
    return Position(sib);
Tree Traversal

- **Breadth-First**
  - Level-order

- **Depth-First**
  - Pre-order: `<root><left><right>`
  - In-order: `<left><root><right>`
  - Post-order: `<left><right><root>`
Tree Traversal

● Breadth-First
  - Level-order

● Depth-First
  - Pre-order: <root><left><right>
  - In-order: <left><root><right>
  - Post-order: <left><right><root>

- Breadth-First
  - Level-order: A, B, C, D, E

- Depth-First
  - Pre-order: A, B, D, E, C
  - In-order: D, B, E, A, C
  - Post-order: D, E, B, C, A
Void LevelOrder(Node* root) {
    if (root == NULL) return;
    queue<Node*> Q;
    Q.push(root);
    // while there is at least one discover node
    while (!Q.empty()) {
        Node* current = Q.front;
        cout << current->data << " ";
        if (current->left != NULL) Q.push(current->left);
        if (current->right != NULL) Q.push(current->right);
        Q.pop();
    }
}

Level Order:
- Time-complexity: O(n)
- Space-complexity: Worst? Best? Average?
In, Pre, Post order:
- Time-complexity: O(n)
- Space-complexity: O(h) Worst? Best? Average?