

ECE 2400 Computer Systems Programming

Spring 2026

Topic 14: Trees

School of Electrical and Computer Engineering
Cornell University

revision: 2026-04-22-10-49

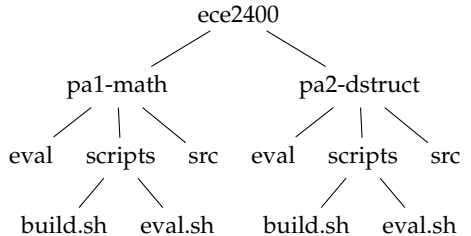
1	Tree Basics	2
2	Tree Concepts	4
3	Tree Storage	5
4	Binary Trees	6
5	Binary Search Trees	10

1. Tree Basics

Use Case 1: representing the Linux filesystem with a tree

(No coincidence that the command is called *tree*!)

```
% tree ece2400
./ece2400
|-- pa1-math
|   |-- eval
|   |-- scripts
|       |-- build.sh
|       |-- eval.sh
|   |-- src
|-- pa2-dstruct
    |-- eval
    |-- scripts
        |-- build.sh
        |-- eval.sh
    |-- src
```

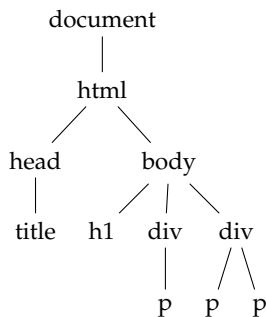


Use Case 2: representing HTML/XML Document Object Model

```

<html>
  <head>
    <title>Simple Website</title>
  </head>
  <body>
    <h1>Simple Website</h1>
    <div>
      <p>some content</p>
    </div>
    <div>
      <p>more content</p>
      <p>even more content</p>
    </div>
  </body>
</html>

```



Trees can be ADTs with operations *insert*, *delete*, *find*, etc. However, in this class, we use **trees as efficient implementations of other ADTs**:

Recall from Topic 10:

Implementation

ADT	List	Vector	Binary Search Tree	Binary Heap Tree	Lookup Table	Hash Table
Indexed Seq	✓	★				
Iterable Seq	★	★				
Stack	★	★				
Queue	★	★				
Priority Queue	✓	✓		★		
Set	✓	✓	★		★	★
Map	✓	✓	★		★	★

2. Tree Concepts

Unrooted Tree

Binary Tree

Balanced Trees

Unbalanced Tree

Ternary Tree

3. Tree Storage

Pointer Based

Array Based

4. Binary Trees

- Focus on object-oriented pointer-based binary tree storing ints
 - Could add iterators to improve data encapsulation
 - Could use object-oriented programming and dynamic polymorphism
 - Could use generic programming and static polymorphism
 - Later: could use concurrent programming to analyze tree in parallel

```
1  class BinaryTreeInt
2  {
3  public:
4
5      BinaryTreeInt();
6      ~BinaryTreeInt();
7
8      void insert_root( int v );
9      void insert_left( Node* node_p, int v );
10     void insert_right( Node* node_p, int v );
11
12     void print() const;
13
14     struct Node
15     {
16         Node( Node* p, int v );
17         int value;
18         Node* parent_p;
19         Node* left_p;
20         Node* right_p;
21     };
22
23     Node* m_root_p;
24 };
```

- Let's defer implementing print and destructor for now

```
1  BinaryTreeInt::Node::Node( Node* p, int v )
2  {
3      parent_p = p; value = v; left_p = nullptr; right_p = nullptr;
4  }
5
6  BinaryTreeInt::BinaryTreeInt()
7  {
8      m_root_p = nullptr;
9  }
10
11 void BinaryTreeInt::insert_root( int v )
12 {
13     m_root_p = new Node(nullptr,v);
14 }
15
16 void BinaryTreeInt::insert_left( Node* node_p, int v )
17 {
18     node_p->left_p = new Node(node_p,v);
19 }
20
21 void BinaryTreeInt::insert_right( Node* node_p, int v )
22 {
23     node_p->right_p = new Node(node_p,v);
24 }
```

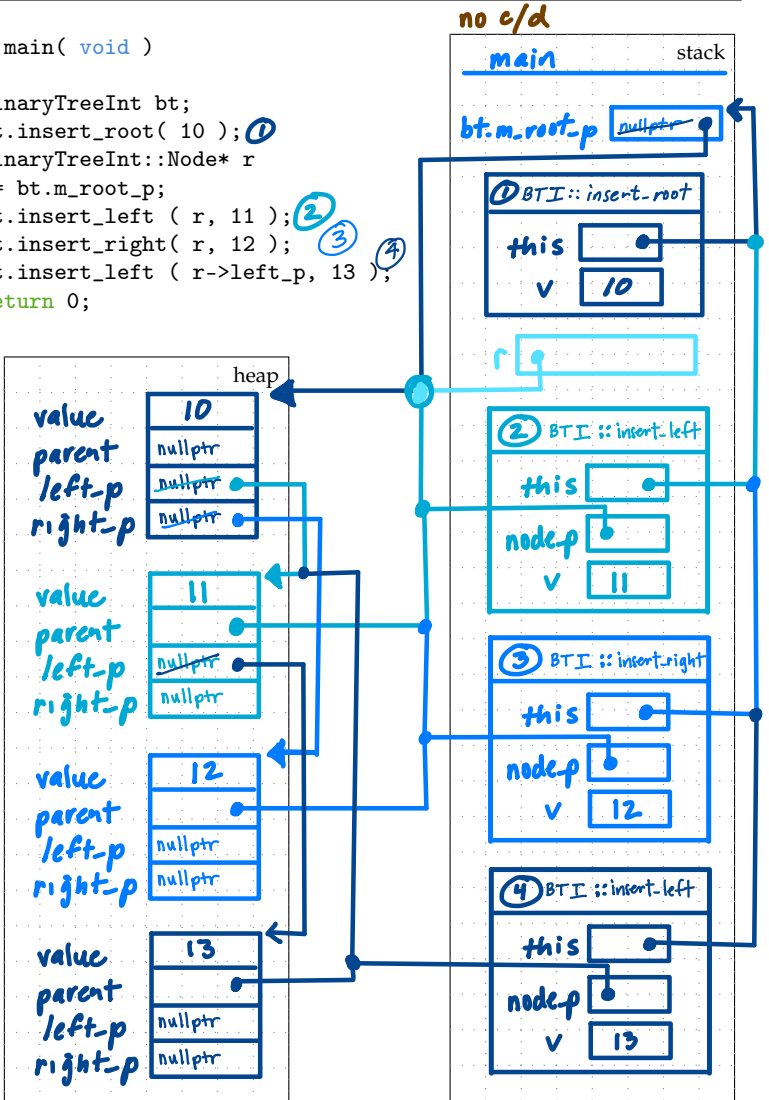
Draw the tree resulting
from this code sequence:

```
1  BinaryTreeInt bt;
2  bt.insert_root( 10 );
3  BinaryTreeInt::Node* r
4  = bt.m_root_p;
5  bt.insert_left ( r, 11 );
6  bt.insert_right( r, 12 );
7  bt.insert_left ( r->left_p, 13 );
```

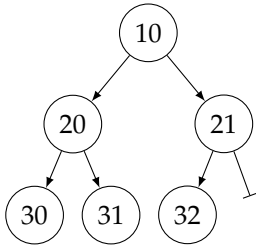
4. Binary Trees

```

01 int main( void )
02 {
03     BinaryTreeInt bt;
04     bt.insert_root( 10 ); ①
05     BinaryTreeInt::Node* r
06     = bt.m_root_p;
07     bt.insert_left ( r, 11 ); ②
08     bt.insert_right( r, 12 ); ③
09     bt.insert_left ( r->left_p, 13 ); ④
10     return 0;
11 }
    
```



3 orderings for Tree Traversals



Traversal Task 1: Printing a tree (member function print)

When writing recursive helper function `print_h`, which traversals work?

```
void BinaryTreeInt::print() const {  
    print_h( m_head_p );  
}  
void BinaryTreeInt::print_h( Node* node_p ) const {
```

[See zybook section 14.1]

Traversal Task 2: Deleting a tree (destructor)

When writing recursive helper function `destruct_h`, which traversals work?

```
BinaryTreeInt::~BinaryTreeInt() {  
    destruct_h( m_head_p );  
}  
void BinaryTreeInt::destruct_h( Node* node_p ) {
```

5. Binary Search Trees

- Recall that sets provide add and contains member functions
- Recall that maps provide add and lookup member functions
- Consider implementing a set/map with a list or vector

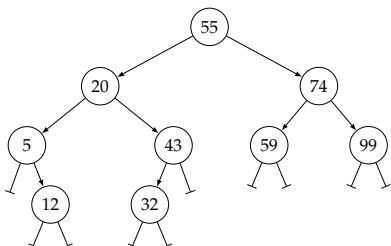
Time Complexity	add (<i>no duplicates: must do contains first!</i>)	contains / lookup
list		
list (sorted)		
vector		
vector (sorted)		
binary search tree		

- A **binary search tree** is a binary tree with the following invariant:

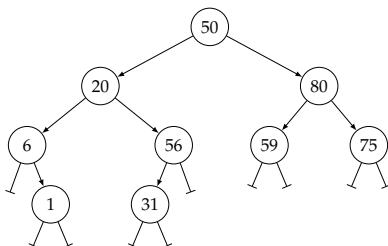
For any node in the tree with value v ,
all values in the left subtree of that node are less than v and
all values in the right subtree of that node are greater than v .

- We can use a binary search tree to achieve $O(\log(N))$ time complexity for both add and contains/lookup
- This time complexity bound assumes binary tree is balanced which may or may not be a reasonable assumption

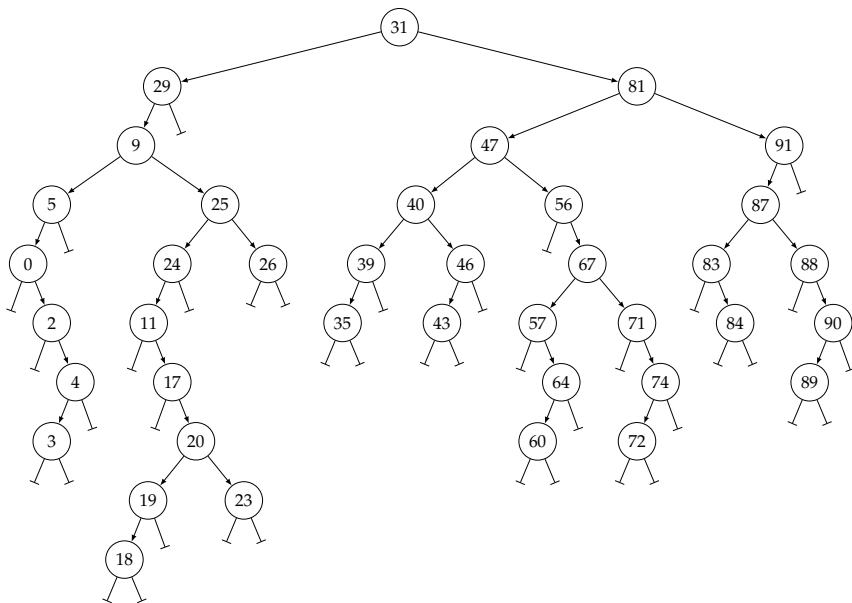
BST invariant is true



BST invariant is not true



Larger BST

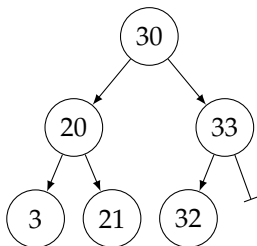


- Focus on object-oriented pointer-based binary search tree storing ints to implement a set
 - Could apply same approach to implementing a map
 - Could use object-oriented programming and dynamic polymorphism
 - Could use generic programming and static polymorphism
 - Later: could use concurrent programming to analyze tree in parallel

```
1  class BinarySearchTreeInt
2  {
3  public:
4      BinarySearchTreeInt();
5      ~BinarySearchTreeInt();
6
7      void add( int v );
8      bool contains( int v ) const;
9
10     private:
11
12     struct Node
13     {
14         Node( Node* p, int v );
15         int value;
16         Node* parent_p;
17         Node* left_p;
18         Node* right_p;
19     };
20
21     void destruct_h( Node* node_p );
22     void add_h( Node* node_p, int v );
23     bool contains_h( Node* node_p, int v ) const;
24
25     Node* m_root_p;
26 };
```

Traversal Task 3: finding a value in a tree (member function contains)

When writing recursive helper function `contains_h`, which traversals work?



```
bool BinarySearchTreeInt::contains( int v ) const {  
    return contains_h( m_root_p, v );  
}
```

```
bool BinarySearchTreeInt::  
    contains_h( Node* node_p, int v ) const {
```

```
}
```

Traversal Task 4: adding a value to a tree (member function add)

Version 1 of recursive helper function add_h is clunky.

```
1 void BinarySearchTreeInt::add( int v ) {
2     if ( m_root_p == nullptr ) {
3         m_root_p = new Node( nullptr, v );
4         return;
5     }
6
7     add_h( m_root_p, v );
8 }
9
10 void BinarySearchTreeInt::add_h( Node* node_p, int v )
11 {
12     assert( node_p != nullptr );
13
14     // base case: value is already in the tree
15     if ( v == node_p->value )
16         return;
17
18     // base case: add new node on right
19     if ( (v > node_p->value) && (node_p->right_p == nullptr) ) {
20         node_p->right_p = new Node( node_p, v );
21         return;
22     }
23
24     // base case: add new node on left
25     if ( (v < node_p->value) && (node_p->left_p == nullptr) ) {
26         node_p->left_p = new Node( node_p, v );
27         return;
28     }
29
30     // recursive case
31     if ( v > node_p->value )
32         add_h( node_p->right_p, v );
33     else
34         add_h( node_p->left_p, v );
35 }
```

Traversal Task 4: adding a value to a tree (member function add)

Version 2 of recursive helper function add_h is elegant!

```
1 void BinarySearchTreeInt::add(int v) {
2     m_steps = 0;
3     m_root_p = add_h(m_root_p, nullptr, v);
4 }
5
6 BinarySearchTreeInt::Node *
7 BinarySearchTreeInt::add_h(Node *node_p, Node *p, int v) {
8     m_steps++;
9
10    // base case: found place to insert new node
11    if (node_p == nullptr)
12        return new Node(p, v);
13
14    // base case: value is already in the tree
15    if (v == node_p->value)
16        return node_p;
17
18    // recursive case
19    if (v > node_p->value)
20        node_p->right_p = add_h(node_p->right_p, node_p, v);
21    else
22        node_p->left_p = add_h(node_p->left_p, node_p, v);
23
24    return node_p;
25 }
```

See zybook section 14.1 for complete runnable Versions 1 and 2!