ECE 2400 Computer Systems Programming
Spring 2026

Topic 14: Trees

School of Electrical and Computer Engineering
Cornell University

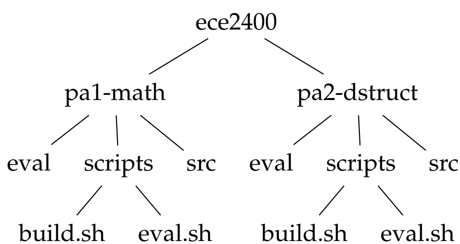
revision: 2026-04-22-11-18

1. Tree Basics

Use Case 1: representing the Linux filesystem with a tree

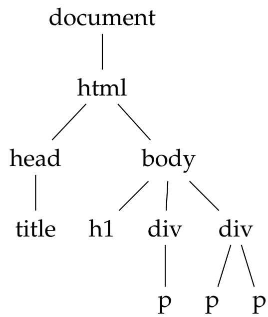
(No coincidence that the command is called `tree`!)

```
% 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

3. Tree Storage

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

```
class BinaryTreeInt
{
 public:

  BinaryTreeInt();
  ~BinaryTreeInt();

  void insert_root( int v );
  void insert_left( Node* node_p, int v );
  void insert_right( Node* node_p, int v );

  void print() const;

  struct Node
  {
    Node( Node* p, int v );
    int   value;
    Node* parent_p;
    Node* left_p;
    Node* right_p;
  };

  Node* m_root_p;
};
```

- Let's defer implementing `print` and destructor for now

```
BinaryTreeInt::Node::Node( Node* p, int v )
{
  parent_p = p; value = v; left_p = nullptr; right_p = nullptr;
}

BinaryTreeInt::BinaryTreeInt()
{
  m_root_p = nullptr;
}

void BinaryTreeInt::insert_root( int v )
{
  m_root_p = new Node(nullptr,v);
}

void BinaryTreeInt::insert_left( Node* node_p, int v )
{
  node_p->left_p = new Node(node_p,v);
}

void BinaryTreeInt::insert_right( Node* node_p, int v )
{
  node_p->right_p = new Node(node_p,v);
}
```

Draw the tree resulting from this code sequence:

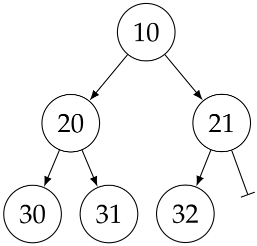
```
BinaryTreeInt bt;
bt.insert_root( 10 );
BinaryTreeInt::Node* r
 = bt.m_root_p;
bt.insert_left ( r, 11 );
bt.insert_right( r, 12 );
bt.insert_left ( r->left_p, 13 );
```

```
int main( void )
{
  BinaryTreeInt bt;
  bt.insert_root( 10 );
  BinaryTreeInt::Node* r
   = bt.m_root_p;
  bt.insert_left ( r, 11 );
  bt.insert_right( r, 12 );
  bt.insert_left ( r->left_p, 13 );
  return 0;
}
```

heap

stack

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

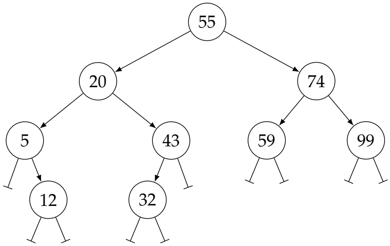
Time Complexity	`add` (*no duplicates: must do contains first!)*	`contains` / `lookup`
list		
list (sorted)		
vector		
vector (sorted)		
binary search tree		

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

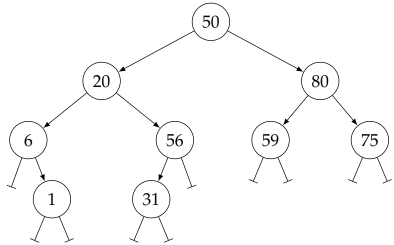
For any node in the tree with value v,
all values in the left subtree of that node are less than v and
all values in the right subtree of that node are greater than v.

- 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

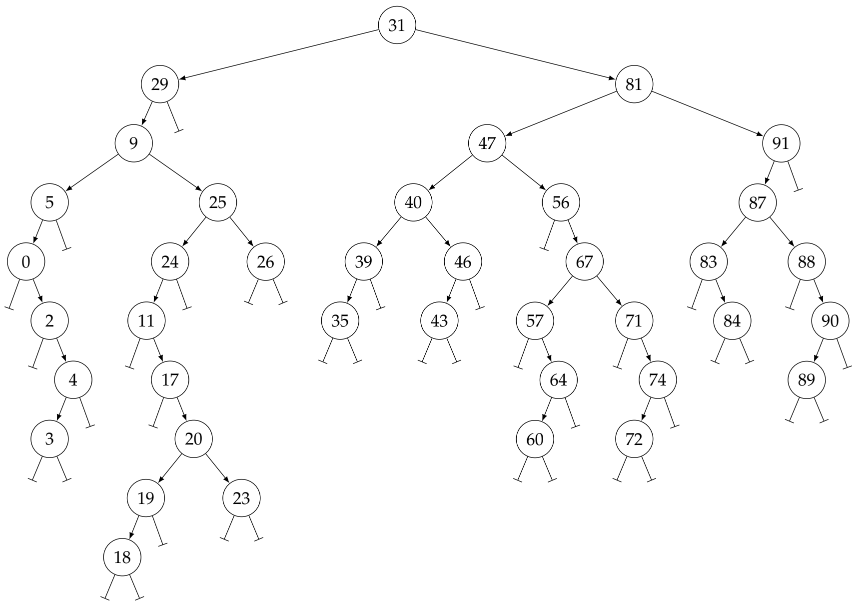
BST invariant is true



BST invariant is not true



Larger BST

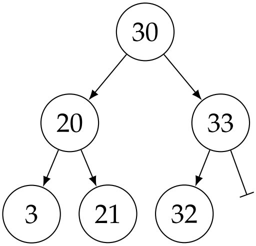


- Focus on object-oriented pointer-based binary search tree storing `ints` to implement a set
  - Could apply same approach to implementing a map
  - Could use object-oriented programming and dynamic polymorphism
  - Could use generic programming and static polymorphism
  - Later: could use concurrent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

```
class BinarySearchTreeInt
{
 public:
  BinarySearchTreeInt();
  ~BinarySearchTreeInt();

  void add( int v );
  bool contains( int v ) const;

 private:

  struct Node
  {
    Node( Node* p, int v );
    int   value;
    Node* parent_p;
    Node* left_p;
    Node* right_p;
  };

  void destruct_h( Node* node_p );
  void add_h( Node* node_p, int v );
  bool contains_h( Node* node_p, int v ) const;

  Node* m_root_p;
};
```

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**.

```
void BinarySearchTreeInt::add( int v ) {
  if ( m_root_p == nullptr ) {
    m_root_p = new Node( nullptr, v );
    return;
  }

  add_h( m_root_p, v );
}

void BinarySearchTreeInt::add_h( Node* node_p, int v )
{
  assert( node_p != nullptr );

  // base case: value is already in the tree
  if ( v == node_p->value )
    return;

  // base case: add new node on right
  if ( (v > node_p->value) && (node_p->right_p == nullptr) ) {
    node_p->right_p = new Node( node_p, v );
    return;
  }

  // base case: add new node on left
  if ( (v < node_p->value) && (node_p->left_p == nullptr) ) {
    node_p->left_p = new Node( node_p, v );
    return;
  }

  // recursive case
  if ( v > node_p->value )
    add_h( node_p->right_p, v );
  else
    add_h( node_p->left_p, v );
}
```

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

Version 2 of recursive helper function `add_h` is **elegant**!

```
void BinarySearchTreeInt::add(int v) {
  m_steps = 0;
  m_root_p = add_h(m_root_p, nullptr, v);
}

BinarySearchTreeInt::Node *
 BinarySearchTreeInt::add_h(Node *node_p, Node *p, int v) {
  m_steps++;

  // base case: found place to insert new node
  if (node_p == nullptr)
    return new Node(p, v);

  // base case: value is already in the tree
  if (v == node_p->value)
    return node_p;

  // recursive case
  if (v > node_p->value)
    node_p->right_p = add_h(node_p->right_p, node_p, v);
  else
    node_p->left_p = add_h(node_p->left_p, node_p, v);

  return node_p;
}
```

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