ECE 2400 Computer Systems Programming
Fall 2021

Topic 17: Trees

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-29-22-33

Handout for Section 6 will be released later in the semester!

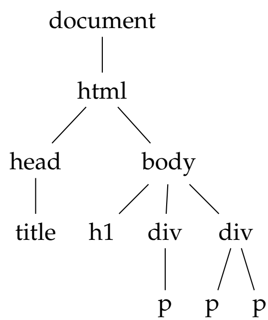
zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

1. Tree Abstract Data Type

- insert new root node for tree
- insert new child for a node in tree
- remove node in tree
- traverse tree

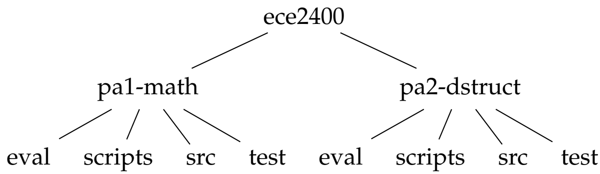
HTML/XML Document Object Model

```
<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>
```



Linux Filesystem

```
% tree ece2400
./ece2400
`- netid
   |- pa1-math
   |  |- eval
   |  |- scripts
   |  |- src
   |  `- test
   `- pa2-dstruct
      |- eval
      |- scripts
      |- src
      `- test
```



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

While trees can be used on their own as an ADT, in this course we will focus on using trees to effeciently implement other ADTs

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
  - Could use functional programming to analyze tree
  - 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

Recursive member function to print tree

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

void BinaryTreeInt::print_h( Node* node_p ) const {
```

Tree Traversals

https://repl.it/@cbatten/ece2400-T17-ex1

Recursive function to delete tree

```
BinaryTreeInt::~BinaryTreeInt()
{
  clear_h( m_head_p );
}

void BinaryTreeInt::clear_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

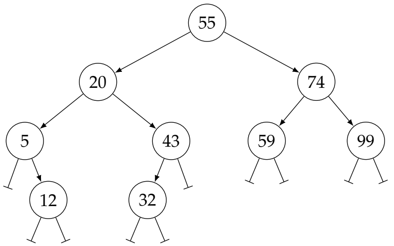
	Time Complexity		
	`add`	`contains` `lookup`	**Space Complexity**
list			
list (sorted)			
vector			
vector (sorted)			
binary search tree			

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

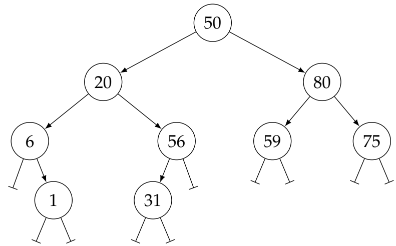
For any node in the tree with value v,
all values in the left subtree of that node are less than v and
all values in the right subtree of that node are greater than v.

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

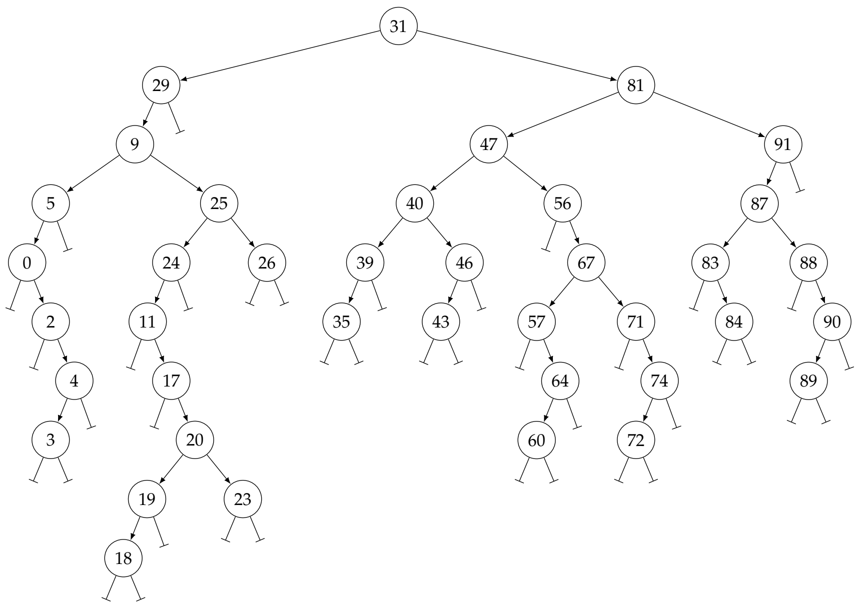
BST invariant is true



BST invariant is not true



Larger BST with 50 nodes



- Focus on object-oriented pointer-based binary search tree storing `int`s 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
  - Could use functional programming to analyze tree
  - 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 clear_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;
};
```

Recursive member function to search for value in tree

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

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

Recursive member function to add value in tree (Version 1)

```
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 );
}
```

Recursive member function to add value in tree (Version 2)

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

BinarySearchTreeInt::Node* BinarySearchTreeInt::
  add_h( Node* node_p, Node* p, int v ) {
```

https://repl.it/@cbatten/ece2400-T17-ex2
https://repl.it/@cbatten/ece2400-T17-ex3

6. Binary Heap Trees