1 Tree Abstract Data Type

2 Tree Concepts

3 Tree Storage

4 Binary Trees

5 Binary Search Trees

6 Binary Heap Trees

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.

Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
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
<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

Implementation

<table>
<thead>
<tr>
<th>ADT</th>
<th>List</th>
<th>Vector</th>
<th>Binary Search Tree</th>
<th>Binary Heap Tree</th>
<th>Lookup Table</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Seq</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterable Seq</td>
<td></td>
<td>★</td>
<td></td>
<td></td>
<td></td>
<td>★</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td>★</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td></td>
<td>★</td>
<td></td>
<td></td>
<td></td>
<td>★</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>✓</td>
<td>✓</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Map</td>
<td>✓</td>
<td>✓</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
</tbody>
</table>

While trees can be used on their own as an ADT, in this course we will focus on using trees to efficiently implement other ADTs.
2. Tree Concepts
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

```cpp
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

```cpp
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:

```cpp
BinaryTreeInt bt;
b. insert_root( 10 );
BinaryTreeInt::Node* r = bt.m_root_p;
b. insert_left ( r, 11 );
b. insert_right( r, 12 );
b. insert_left ( r->left_p, 13 );
```
```cpp
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;
}
```
Recursive member function to print tree

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

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

---

Tree Traversals

```cpp
https://repl.it/@cbatten/ece2400-T17-ex1
```
Recursive function to delete tree

```cpp
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

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>add</th>
<th>contains lookup</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• 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
5. Binary Search Trees

**BST invariant is true**

```
   55
  /   \
20    74
 / \   / \
5   43 59   99
```

**BST invariant is not true**

```
   50
  /   \
20    80
 /     /\
6     59 75
```

**Larger BST with 50 nodes**

```
   31
  /   \
29    81
 /     /\
9     91
 /     /\
5     87
 /     /\
0     83
 /     /\
2     84
 /     /\
4     90
 /     /\
3     89
 /     /\
18    35
```

---

**Topic 17: Trees**
• 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
  – 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

```cpp
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)

```cpp
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)

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

BinarySearchTreeInt::Node* BinarySearchTreeInt::
    add_h( Node* node_p, Node* p, int v ) {
```
6. Binary Heap Trees