CSE 100: BST OPERATIONS AND C++ ITERATORS
Announcements

• Gradesource and clickers:
  • We’ll be making one more pass for unregistered clickers tonight, but after that you’ll be on your own…
How is Assignment 1 going?

A. I haven’t looked at it.
B. I’ve read it, but I haven’t done anything
C. I’ve gotten the code and possibly started looking at it/playing around with it.
D. I’ve implemented some of the required functions, but I’m not done.
E. I’m done!
Deleting a node from a BST
In-order traversal of BST

Which of the following results from an in-order traversal of a BST?

A. Nodes are visited in the order in which they were inserted into the BST
B. Nodes are visited in order of the number of children that they have
C. Nodes are visited in sorted order of their keys
D. None of the above
The successor method performs an in-order tree traversal, one step at a time. (True or False?)
The successor method performs an in-order tree traversal, one step at a time.

Find several different cases to consider in the tree below; describe how to find the successor in each case.
The successor method performs an in-order tree traversal, one step at a time.

Find several different cases to consider in the tree below; describe how to find the successor in each case.
In Java:

```java
public class BSTNode {
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode(int d) {
        data = d;
    }
}
```

C++, attempt 2:

```cpp
class BSTNode {
    public:
        BSTNode left;
        BSTNode right;
        BSTNode parent;
        int const data;

    BSTNode(const int & d) {
        data = d;
    }
};
```

Which of the following is a problem with the C++ implementation above?
A. Because data is a constant variable, the constructor will cause an error.
B. You cannot pass an integer by reference into a function. Integers must be passed by value.
C. Since d is passed by reference, you cannot assign its value to data, which is an int. You need to dereference it first.
D. The constructor needs a semi-colon at the end of its definition.
In Java:

```java
public class BSTNode {
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode( int d ) {
        data = d;
    }
}
```

C++, attempt 2:

```cpp
class BSTNode {
    public:
        BSTNode left;
        BSTNode right;
        BSTNode parent;
        int const data;

        BSTNode( const int & d ) {
            data = d;
        }
};
```

Which of the following is a problem with the C++ implementation above?
A. Because data is a constant variable, the constructor will cause an error.

Why make data const?
What is the problem with how we have declared left, right and parent above?

A. They should be `BSTNode*` (pointers to `BSTNode` instances) and not `BSTNode` type.
B. They should be declared to be `const`.
C. They should be declared as `BSTNode&` (reference variables).
In Java:

```java
public class BSTNode {
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode( int d ) {
        data = d;
    }
}
```

C++, attempt 4:

```cpp
class BSTNode {
public:
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
    int const data;

    BSTNode( const int & d ) :
        data(d) {  }
};
```

And now, a little practice with pointers…
Which of the following statements is true about this code?

```c
int a = 5;
int b = a;
int* pt1 = a;
```

A. Both ptr and b can be used to change the value of a.
B. Only pts can be used to chance the value of a.
C. This code causes a compile error.
```c
int a = 5;
int b = a;
int* pt1 = &a;
```

<table>
<thead>
<tr>
<th>address</th>
<th>memory cell</th>
<th>identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>512000</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>512004</td>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>512008</td>
<td>512000</td>
<td>pt1</td>
</tr>
</tbody>
</table>
int a = 5;
int b = a;
int* pt1 = &a;
How does the diagram change if we change the code as follows. Which is the correct picture now?

```c
int a = 5;
int & b = a;
int* pt1 = &a;
```

Options:

A.  

```
<table>
<thead>
<tr>
<th>a:</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td></td>
</tr>
<tr>
<td>pt1:</td>
<td></td>
</tr>
</tbody>
</table>
```

B.  

```
<table>
<thead>
<tr>
<th>a:</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td>5</td>
</tr>
<tr>
<td>pt1:</td>
<td></td>
</tr>
</tbody>
</table>
```

C.  

```
<table>
<thead>
<tr>
<th>a:</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td></td>
</tr>
<tr>
<td>pt1:</td>
<td></td>
</tr>
</tbody>
</table>
```
In Java:

```java
public class BSTNode {
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode(int d) {
        data = d;
    }
}
```

C++, attempt 4:

```cpp
class BSTNode {
public:
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
    int const data;

    BSTNode(const int & d) : data(d) {
    }
};
```

Are there any remaining problems with this C++ implementation?
A. Yes
B. No
In Java:

```java
public class BSTNode {
    public BSTNode left;
    public BSTNode right;
    public BSTNode parent;
    public int data;

    public BSTNode( int d ) {
        data = d;
    }
}
```

C++, attempt 5:

```cpp
class BSTNode {
public:
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
    int const data;

    BSTNode( const int & d ) :
        data(d) {
            left = right = parent = 0;
        }
};
```

ALWAYS initialize in C++. C++ won't do it for you. Why not?

What if we don't want to be stuck with ints?
BST, with templates:

```cpp
template<typename Data>

class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
        left = right = parent = 0;
    }

};
```
BST, with templates:

```cpp
template<
  typename Data>

class BSTNode { 
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) : 
    data(d) { 
      left = right = parent = 0;
    }
};
```

A. How would you create a BSTNode object on the runtime stack?
B. How would you create a pointer to BSTNode with integer data?

```cpp
BST, with templates:

```template<typename Data>```

class BSTNode {
    public:
        BSTNode<Data>* left;
        BSTNode<Data>* right;
        BSTNode<Data>* parent;
        Data const data;

        BSTNode( const Data & d ) :
            data(d) {
                left = right = parent = 0;
            }
};
```
BST, with templates:

```cpp
template<
typename Data>

class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
            left = right = parent = 0;
        }
};

C. How would you create an BSTNode object on the heap?
BST, with templates:

```cpp
template<
    typename Data>

class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
            left = right = parent = 0;
        }
};
```

BSTNodes will be used in a BST, and with a BSTIterator…
The C++ Standard Template Library is a very handy set of built-in data structures (containers), including:

- array
- vector
- deque
- forward_list
- list
- stack
- queue
- priority_queue
- set
- multiset (non-unique keys)
- unordered_set
- map
- unordered_map
- multimap
- bitset

Of these, `set` is one that is implemented using a balanced binary search tree (typically a red-black tree).
Imagining ourselves as C++ STL class designers...

- set’s find function has this prototype:

```cpp
template<typename T>

class set {

public:
    iterator find ( T const & x ) const;

What does the final const in the function header above mean?
A. find cannot change its input argument
B. find cannot change where its input argument, which is a pointer, points to
C. find cannot change the underlying set
Imagining ourselves as C++ STL class designers…

- set’s find function has this prototype:

```cpp
template <typename T>

class set {

public:
    iterator find ( T const & x ) const;

};
```

The documentation for set’s find function says:

*Searches the container for an element with a value of x and returns an iterator to it if found, otherwise it returns an iterator to the element past the end of the container.*
C++ STL Iterators

• What is an iterator?
C++ STL Iterators

What is an iterator?

- In the iterator pattern of OO design, a container has a way to supply to a client an iterator object which is to be used by the client to access the data in the container sequentially, without exposing the container’s underlying representation.
C++ STL Iterators

set<int> c;
...

// get an iterator pointing to container’s first element
set<int>::iterator itr = c.begin();

What do you think `begin()` returns?
A. The address of the root in the set container class
B. The address of the node with the smallest data key
C. The address of the smallest data key
D. None of the above
An iterator class template for a BST

• Suppose a BST's nodes are instances of a class template `Node` as shown before.

• At each step in an iteration, an iterator for the BST only needs to keep a pointer to the current `Node` in the BST.

• So, define a `BSTIterator` class template with one member variable that is a pointer to the current node; then a constructor and overloaded operators for the class are easy to define:

```cpp
template <typename T>
class BSTIterator { 

private:
    Node<T>* curr;

public:
    /** Constructor */
    BSTIterator(Node<T>* n) : curr(n) {} 

};
```

Is this definition of the BSTIterator class complete?
A. Yes
B. No
C++ STL Iterators

```cpp
c set<int> c;
...

// get an iterator pointing to container’s first element
set<int>::iterator itr = c.begin();

// get an iterator pointing past container’s last element
set<int>::iterator end = c.end();

// loop while itr is not past the last element
while (itr != end) {
    cout << *itr << endl; // dereference the itr to get data
    ++itr; // increment itr to point to next element
}
```

Circle/list the functions that the iterator has to implement.
template<typename Data>
class BSTIterator : public std::iterator<std::input_iterator_tag, Data> {

private:
    BSTNode<Data>* curr;

public:
    /** Constructor. Use the argument to initialize the current BSTNode * in this BSTIterator. */ // TODO
    BSTIterator(BSTNode<Data>* curr) { // TODO }

    /** Dereference operator. */
    Data operator*() const {
        return curr->data;
    }

    /** Pre-increment operator. */
    BSTIterator<Data>& operator++() {
        curr = curr->successor();
        return *this;
    }
}
C++ STL Iterators

set<int> c;
...
// get an iterator pointing to container's first element
set<int>::iterator itr = c.begin();
// get an iterator pointing past container's last element
set<int>::iterator end = c.end();
// loop while itr is not past the last element
while(itr != end) {
    cout << *itr << endl; // dereference the itr to get data
    ++itr; // increment itr to point to next element
}

What kind of traversal is the above code doing?
A. In order
B. Pre order
C. Post order
D. None of the above
The successor method performs an in-order tree traversal, one step at a time.

Bored? Implement remove in a BST. Discuss the merits of the C++ iterator pattern.
Next class: Average case analysis of search in a BST

- Warning! There will be math 😊