CSE 100: C++
TEMPLATES AND ITERATORS
Announcements

• iClickers:
  • Please register at ted.ucsd.edu.

• Start ASAP!! For PA1 (Due next week).
  • 10/6 grading and 10/8 regrading
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!
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?
Pointers to const, and const pointers

• Creating a pointer to a const...

  ```
  const int c = 3; // create a const int variable
  // equivalent: int const c = 3;
  const int * p;  // create a pointer to a const int
  // equivalent: int const * p;
  p = &c;        // make p point to c
  // ERROR: p is pointer to const
  *p = 4;        // ERROR: p is pointer to const
  ```

• Creating a const pointer...

  ```
  int d = 5;    // create an int variable
  int * const q = &d; // create and initialize a const ptr to int
  // equivalent: int * const q = &d;
  *q = 9;       // okay... changing what q points to
  q = &d;       // ERROR: q itself is const!
  ```
Memory management in Java and C++

- In Java, all objects are created using **new**, and memory an object takes is automatically reclaimed by the garbage collector when it can no longer be accessed in your program:

  ```java
  C x = new C(); // create an object
  x = null;      // object is inaccessible; no problem, gc will free
  ```

- In C++, objects can be created using **new**, but memory for such objects is not automatically reclaimed:

  ```c++
  C* x = new C(); // create an object using new
  x = nullptr;    // object is inaccessible; problem!! it will not be freed
  ```

- That is a **memory leak**! In C++, an object created using new must be explicitly freed using **delete** for its memory to be reclaimed for later use in your program:

  ```c++
  C* x = new C(); // create an object using new
  ...             // do things with the object
  delete x;      // free the object using delete
  x = nullptr;   // make pointer null to avoid dangling pointer
  ```

- Note: some objects created in other ways in a C++ program are reclaimed automatically. **Understanding the details of memory management** is an important part of C++ programming.
Automatic, static, and dynamic data in Java and C++

- A running program typically has 3 kinds of data in it:
  - automatic
  - static
  - dynamic

- These kinds of data differ in:
  - what region of machine memory the data is stored in
  - when and how memory for the data is allocated ("created")
  - when and how memory for the data is deallocated ("destroyed")

- Both Java and C++ have basically these same 3 kinds of data, with some differences
  - (Note: these are kinds of data, i.e. variables. Executable instructions are stored somewhat differently.)
Automatic data

- Automatic data is called "automatic" because it is automatically created and destroyed when your program runs.
  - Examples of automatic data in both Java and C++: formal parameters to functions, local variables in functions (if the variables are not declared static).
- Memory for automatic data is allocated on the runtime stack.
- Memory for automatic data is allocated when execution reaches the scope of the variable’s declaration.
- Memory for automatic data is deallocated when execution leaves that scope.
Static data

- Static data is called "static" because it essentially exists during the entire execution of your program
  - Examples of static data in Java and C++: variables declared static (or, in C++, variables declared outside any function and class body)
- Memory for static data is allocated in the static data segment
- Memory for static data is allocated when your program starts, or when execution reaches the static data declaration for the first time
- Memory for static data is deallocated when your program exits
Dynamic data

- Dynamic data is called "dynamic" because it is allocated (and, in C/C++, destroyed) under explicit programmer control
  - Example of dynamic data in Java: *all* objects (instances of a class), including arrays. Dynamic data is created using the `new` operator
  - Example of dynamic data in C++: any variables created using the `new` operator. In C++, any type variable can be created using `new`
- Memory for dynamic data is allocated from the "heap"
- In Java, dynamic data exists from the time `new` is called until the object is reclaimed by the garbage collector
- In C++, dynamic data exists from the time `new` is called until it is deallocated with the `delete` operator
In Java:

```java
public class BSTNode {
    public BSTNode left; // left child
    public BSTNode right; // right child
    public BSTNode parent; // parent node
    public int data; // data value

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

C++, attempt 5:

```cpp
class BSTNode {
public:
    BSTNode* left; // left child
    BSTNode* right; // right child
    BSTNode* parent; // parent node
    int const data; // data value

    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?
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;
        }
};
```

B. How would you create a pointer to BSTNode with integer data?
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:

```
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

A. find cannot change its input argument

A. find cannot change its input argument
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
Iterator class template for BST

template <typename T>
class BSTIterator {

private:
    Node<T>* curr;

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

```cpp
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 */
    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
Average case analysis

- Warning! There will be math 😊
- Why is it important that we do this?
  - So you have a hope of doing it yourself on a new data structure (perhaps one you invent?)
  - Mathematical analysis can be insightful!
Given a BST having:

- N nodes $x_1, \ldots, x_N$ such that $\text{key}(x_i) = k_i$
- Probability of searching for key $k_i$ is $p_i$

What is the expected number of comparisons to find a key?

A. $\sum_{i=1}^{N} p_i \cdot (\text{No. of comparisons to find } k_i)$

B. $\sum_{i=1}^{N} p_i \cdot x_i$

C. $\left( \sum_{i=1}^{N} \text{No. of comparisons to find } k_i \right) / N$
Number of compares to find key $k_i$ is related to the **Depth** of $x_i$ in the BST

- **Depth** of node $x_i$: No. of nodes on the path from the root to $x_i$ inclusive
- Notation for depth of $x_i$: 
Probabilistic Assumption #1

- **Probabilistic Assumption #1:**
  All keys are equally likely to be searched (how realistic is this)?

- Thus $p_1 = \ldots = p_N = \frac{1}{N}$ and the average number of comparisons in a successful find is:

\[
D_{\text{avg}}(N) = \sum_{i=1}^{N} p_i d(x_i) = \sum_{i=1}^{N} \frac{1}{N} d(x_i) = \frac{1}{N} \left( \sum_{i=1}^{N} d(x_i) \right)
\]

\[
\sum_{i=1}^{N} d(x_i) = \text{total node depth}
\]
Calculating total node depth

What is the total node depth of this tree?
A. 3
B. 5
C. 6
D. 9
E. None of these
Calculating total node depth

• In a complete analysis of the average cases, we need to look at all possible BSTs that can be constructed with same set of N keys
• What does the structure of the tree relate to?
How many possible ways can we insert three elements into a BST?

- Suppose N=3 and the keys are (1, 3, 5)
How many possible ways can we insert three elements into a BST?

• Suppose N=3:
  
  (1,3,5); (1,5,3); (3,1,5); (3,5,1); (5,1,3); (5,3,1)

  6 possible trees

What is the total number of possibilities for an N-node BSTs?

A. \(N^N\)
B. \(N!\)
C. \(e^N\)
D. \(N*N\)
E. None of these
Relationship between order of insertion and structure of the BST

• Given a set of N keys: The structure of a BST constructed from those keys is determined by the order the keys are inserted.

• Example: N=3. There are N! = 3! = 6 different orders of insertion of the 3 keys. Here are resulting trees:
Probabilistic assumption #2

- We may assume that each key is equally likely to be the first key inserted; each remaining key is equally likely to be the next one inserted; etc.
- This leads to **Probabilistic Assumption #2**
  
  *Any insertion order (i.e. any permutation) of the keys is equally likely when building the BST*

- This means with 3 keys, each of the following trees can occur with probability 1/6
Average Case for successful Find: Brute Force Method

3, 1, 5
1, 3, 5
1, 5, 3
5, 1, 3
5, 3, 1
3, 5, 1
Let $D(N)$ be the expected total depth of BSTs with $N$ nodes, over all the $N!$ possible BSTs, assuming that Probabilistic Assumption #2 holds.

\[
D(N) = \sum_{\text{all BSTs } T_j \text{ with } N \text{ nodes}} \left( \text{probability of } T_j \right) \left( \text{Total Depth}(T_j) \right)
\]

\[
= \sum_{\text{all BSTs with } N \text{ nodes}} \left( \frac{1}{N!} \right) \left( \sum_{i=1}^{N} d(x_i) \right)
\]

If Assumption #1 also holds, the average # comparisons in a successful find is

\[
D_{\text{avg}}(N) = D(N) / N
\]

The computationally intensive part is constructing $N!$ trees to compute $N!$ total depth values: This is a brute force method!
How do we compute $D(N)$?

$$D(N) = \sum_{\text{all BSTs with } N \text{ nodes}} \left( \frac{1}{N!} \right) \left( \sum_{i=1}^{N} d(x_i) \right)$$

We need an equation for that does not involve computing $N!$ total depth values (in a brute force fashion).

Key Idea: We will build a recurrence relation for $D(N)$ in terms of $D(N-1)$ and then solve that recurrence relation to give us a sum over $N$ (instead of $N!$).
Towards a recurrence relation for average BST total depth

- Define $D(N|i)$ as expected total depth of a BST with $N$ nodes, assuming that $T_L$ has $i$ nodes (and $T_R$ has $N-i-1$)
Average case analysis of find in BST

- Given N nodes, how many such subsets of trees are possible as i is varied?

A. $N$
B. $N!$
C. $\log_2 N$
D. $(N-1)!$
Probability of subtree sizes

• Let $P_N(i) =$ the probability that $T_L$ has $i$ nodes
• It follows that $D(N)$ is given by the following equation

$$D(N) = \sum_{i=0}^{N-1} P_N(i)D(N \mid i)$$
Average total depth of a BST with $N$ nodes

$$D(N) = \sum_{i=0}^{N-1} P_N(i)D(N \mid i)$$

$$D(N) = \sum_{i=0}^{N-1} \frac{1}{N} [D(i) + D(N - i - 1) + N]$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} D(i) + \frac{1}{N} \sum_{i=0}^{N-1} D(N - i - 1) + N$$

True or false: The term in the blue box is equal to the term in the red box
A. True
B. False
• Note that those two summations just add the same terms in different order; so

\[ D(N) = \sum_{i=0}^{N-1} \frac{2}{N} D(i) + N \]

• ... and multiplying by \( N \),

\[ ND(N) = 2 \sum_{i=0}^{N-1} D(i) + N^2 \]

• Now substituting \( N-1 \) for \( N \),

\[ (N-1)D(N-1) = 2 \sum_{i=0}^{N-2} D(i) + (N-1)^2 \]

• Subtracting that equation from the one before it gives

\[ ND(N) - (N-1)D(N-1) = 2D(N-1) + N^2 - (N-1)^2 \]

• ... and collecting terms finally gives this recurrence relation on \( D(N) \):

\[ ND(N) = (N+1)D(N-1) + 2N - 1 \]
N*D(N) = (N+1)*D(N-1) + 2N – 1

How does this help us, again?
A. We can solve it to yield a formula for D(N) that does not involve N!
B. We can use it to compute D(N) directly
C. I have no idea, I’m totally lost
Through unwinding and some not-so-complicated algebra (which you can find in your reading, a.k.a. Paul’s slides) we arrive at:

\[ ND(N) = (N + 1)D(N - 1) + 2N - 1 \]

we arrive at:

\[ D(N) = 2(N + 1) \sum_{i=1}^{N} \frac{1}{i} - 3N \]

No N! to be seen! Yay!
And with a little more algebra, we can even show an approximation:

\[ D_{avg}(N) \approx 1.386 \log_2 N \]

Conclusion: The average time to find an element in a BST with no restrictions on shape is \( \Theta(\log N) \).
The importance of being balanced

- A binary search tree has average-case time cost for Find = \( \Theta (\log N) \):

What does this analysis tell us:
- On an average things are not so bad provided assumptions 1 and 2 hold
- But the probabilistic assumptions we made often don’t hold in practice
  - Assumption #1 may not hold: we may search some keys many more times than others
  - Assumption #2 may not hold: approximately sorted input is actually quite likely, leading to unbalanced trees with worst-case cost closer to \( O(N) \) when \( N \) is large
- We would like our search trees to be balanced
The importance of being balanced

• We would like our search trees to be balanced
• Two kinds of approaches
  • Deterministic methods guarantee balance, but operations are somewhat complicated to implement (AVL trees, red black trees)
  • Randomized methods (treaps, skip lists) (insight from our result) – deliberate randomness in constructing the tree helps!!
    • Operations are simpler to implement
    • Balance not absolutely guaranteed, but achieved with high probability
• We will return to this topic later in the course