CSE 100:
C++ TEMPLATES AND ITERATORS (CONTD), AVERAGE CASE ANALYSIS OF FIND
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;
        }
};
```

1. How would you create a `BSTNode` object on the runtime stack?
2. How would you create a pointer to BSTNode with integer data?
BST, with templates:

\[\text{template<\text{typename Data}>}\]

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

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

3. How would you create an \text{BSTNode} \text{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…
CHANGING GEARS: C++ STL and BSTs

• 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 do we do with pointers on basic data structures like arrays?
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

```c
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
}
```
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
What are the different ways that we can implement an in order traversal of a BST?

Bored? Implement remove in a BST. Discuss the merits of the C++ iterator pattern.
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!
Average case analysis of a “successful” find

Given a BST having:

1. **Worst case:**
2. **Best case:**
3. **Average case:**

How many compares to locate a key in the BST?

- $N$ nodes $x_1, x_2, \ldots, x_N$, such that $\text{key}(x_i) = k_i$
Given a BST having:
- $N$ nodes $x_1, .. x_N$ such that 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. $(\sum_{i=1}^{N} \text{No. of comparisons to find } k_i) / N$
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 $x_i$: 

![Diagram of a binary search tree](image)
Given a BST having:

- N nodes \( x_1, \ldots, x_N \) such that 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. \( \sum_{i=1}^{N} p_i \cdot \text{(No. of comparisons to find } k_i \text{)} \)

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

C. \( \frac{\sum_{i=1}^{N} \text{No. of comparisons to find } k_i}{N} \)
Probabilistic Assumption #1

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

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

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

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

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

So far:

Given a particular BST with $N$ nodes, we found a way to calculate average #comparisons to find a node in the BST

But that’s not enough.
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 N keys
• We need to make an assumption about the probabilities that each of these different trees will occur.
How many possible ways can we insert three elements into a BST?

- Suppose \( N=3 \) and the keys are \((1, 2, 3)\)
How many possible ways can we insert three elements into a BST?

• Suppose N=3:
  (1,2,3); (1,3,2); (2,1,3); (2,3,1); (3,1,2); (3,2,1)
  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
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.

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

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

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

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 $D(N)$ 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

- We are interested in finding:
- Assume we have solved the smaller versions of the problem
  So, we know $D(i) \ \forall i < N$
Towards a recurrence relation for average BST total depth

Define the following sub-problem:
Find the expected depth of all trees where the root node is the $i^{th}$ largest key and the rest of the keys can be organized in any fashion in the left and right subtrees of the root

Which of the following best describes the consequence of fixing the root to be the $i^{th}$ largest key:
A. We have described our original problem in terms of a smaller version of the problem
B. We are restricted to trees with a fixed number of nodes in the left and right subtrees of the root
C. We can describe our original problem as $(N-1)!$ such sub problems
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 $N-i-1$ nodes).
Average case analysis of find in BST

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

![Diagram of tree with $i$ nodes and $N-i-1$ nodes]

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

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

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

What is the value of $P_N(i)$?

Hint: use assumption #2, any of the N keys are equally likely to be inserted first

A. $N$
B. It depends on $i$ and $N$
C. It depends only on $N$
Towards a recurrence relation for average BST total depth

- We defined $D(N|i)$ as the expected total depth of a BST with $N$ nodes, assuming that $T_L$ has $i$ nodes (and $T_R$ has $N-i-1$)

What is $D(N|i)$ in terms of $D(i)$ & $D(N-i-1)$?

Hint: all nodes in each subtree are 1 deeper in tree $T$

A. $D(i) + D(N-i-1)$
B. $D(i) + D(N-i-1) + 1$
C. $D(i) + D(N-i-1) + N$
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

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

• ... 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 \( D(N) \):

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

$$N \cdot D(N) = (N+1) \cdot D(N-1) + 2N - 1$$
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