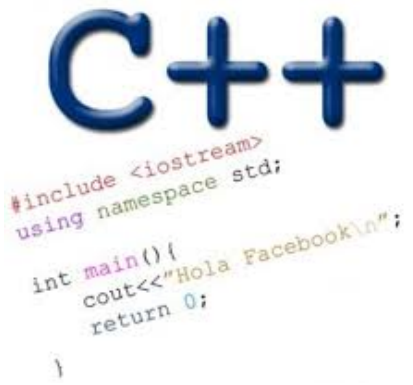
ITERATORS:
AN ADT SPECIALIZED FOR TRAVERSAL

Problem Solving with Computers-II



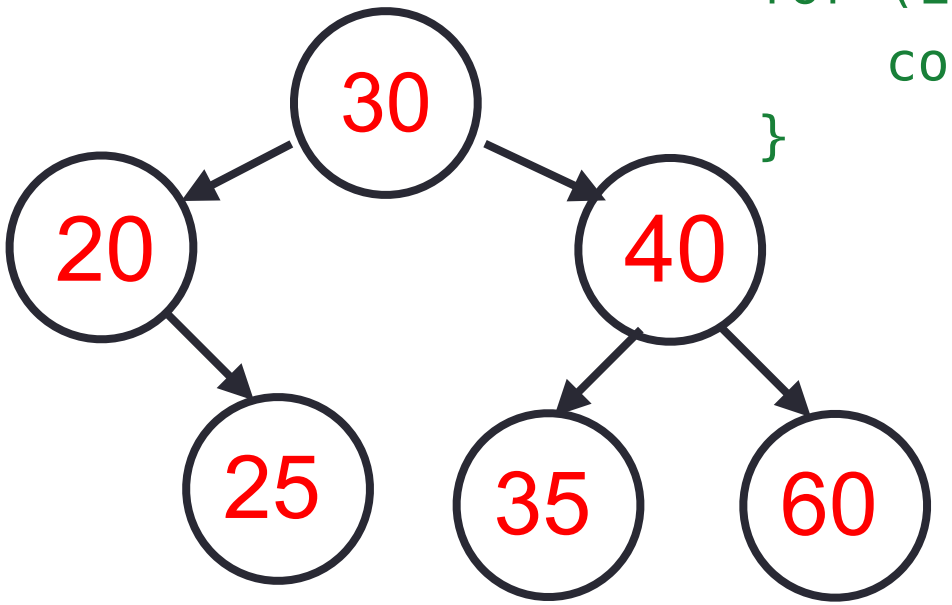
Recursive vs. Iterative traversal of a BST

Recursive in order: `printInorder`

```
void bst::printInorder(Node *r) const{
    if (!r) return;
    printInorder(r->left);
    cout << r->data << " ";
    printInorder(r->right);
}
```

Iterative: offered by `std::set`

```
std::set<int> s =
{30, 20, 25, 40,35, 60};
for (int x : s) {
    cout << x << " ";
}
```

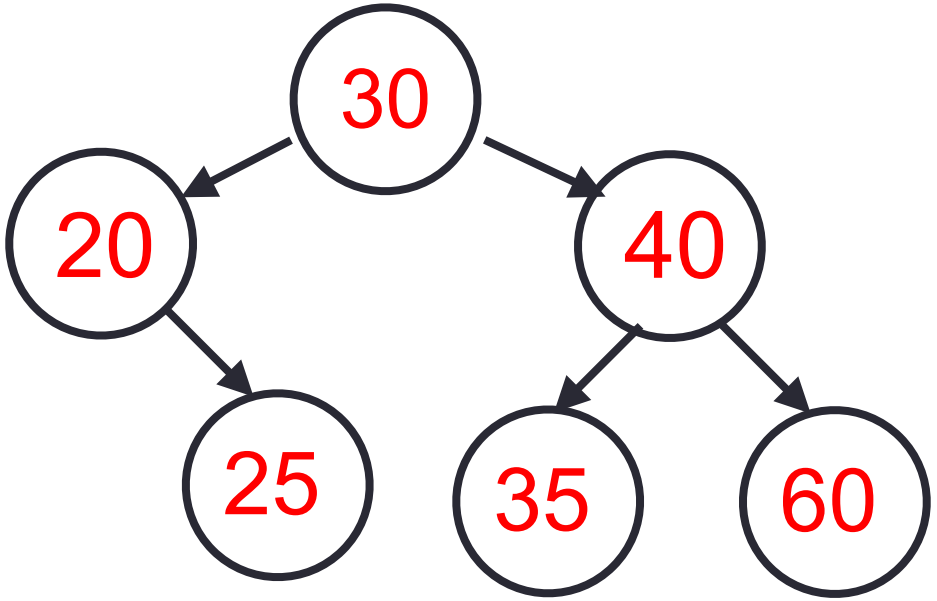


Why doesn't std::set have a printInorder() function?

Our goal: Implement one-at-a-time navigation for custom BST

What you write…

```
std::set<int> s = {. . .}
for (int x : s) {
     cout << x << " ";
}
```



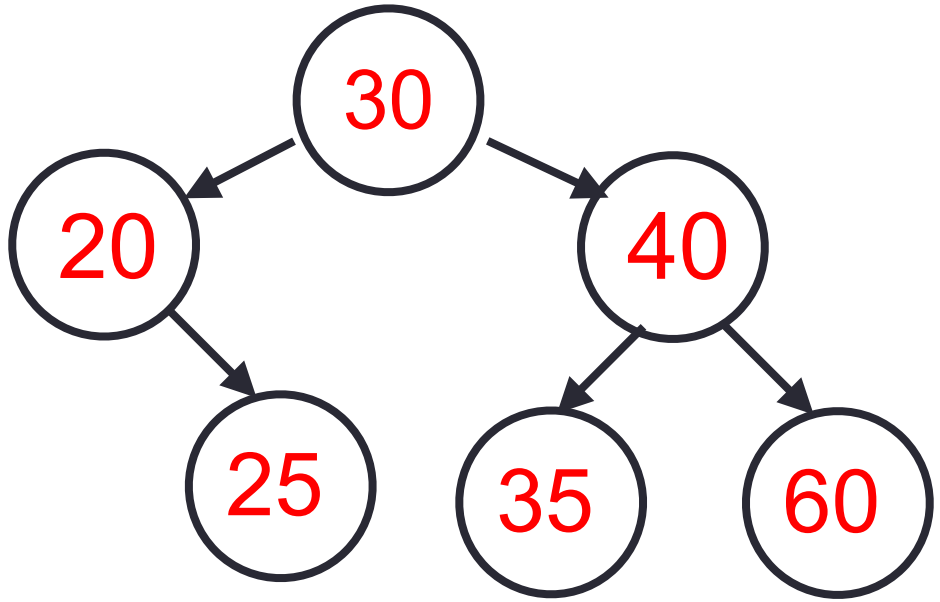
What actually happens:

```
for (std::set<int>::iterator it = s.begin(); it != s.end(); ++it) {
    cout << *it << " ";
}
```

Roadmap to implementing one at-a-time navigation for bst class

(1) Implement helpers: getmin and successor

```
Node* r = b.getmin(root);
while(r){
    cout << r->data << " ";
    r = b.successor(r);
}
```



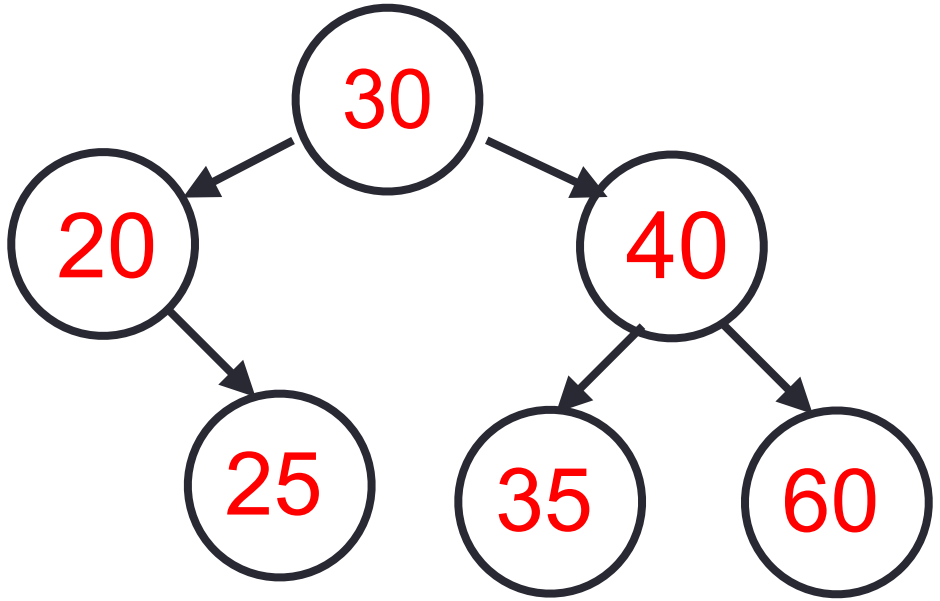
Are we done? Why/why not? - Discuss (2 mins)

Roadmap to implementing one at-a-time navigation for bst class

(1) Implement helpers: getmin and successor

(2) Implement a new ADT called `iterator` that **abstracts a traversal pointer**!!!

```
iterator it;
*it = ____ (data)
++it; // Moves to ____
```



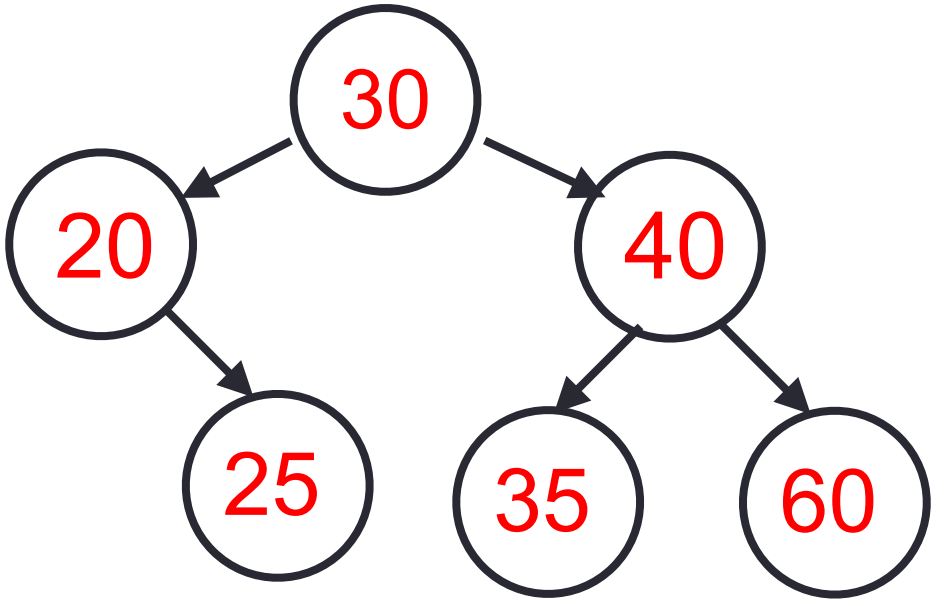
Discuss (2 mins):
What functions does iterator ADT need to allow operations like `*it` and `++it`?

Task 1: Implement two useful functions: getmin and successor

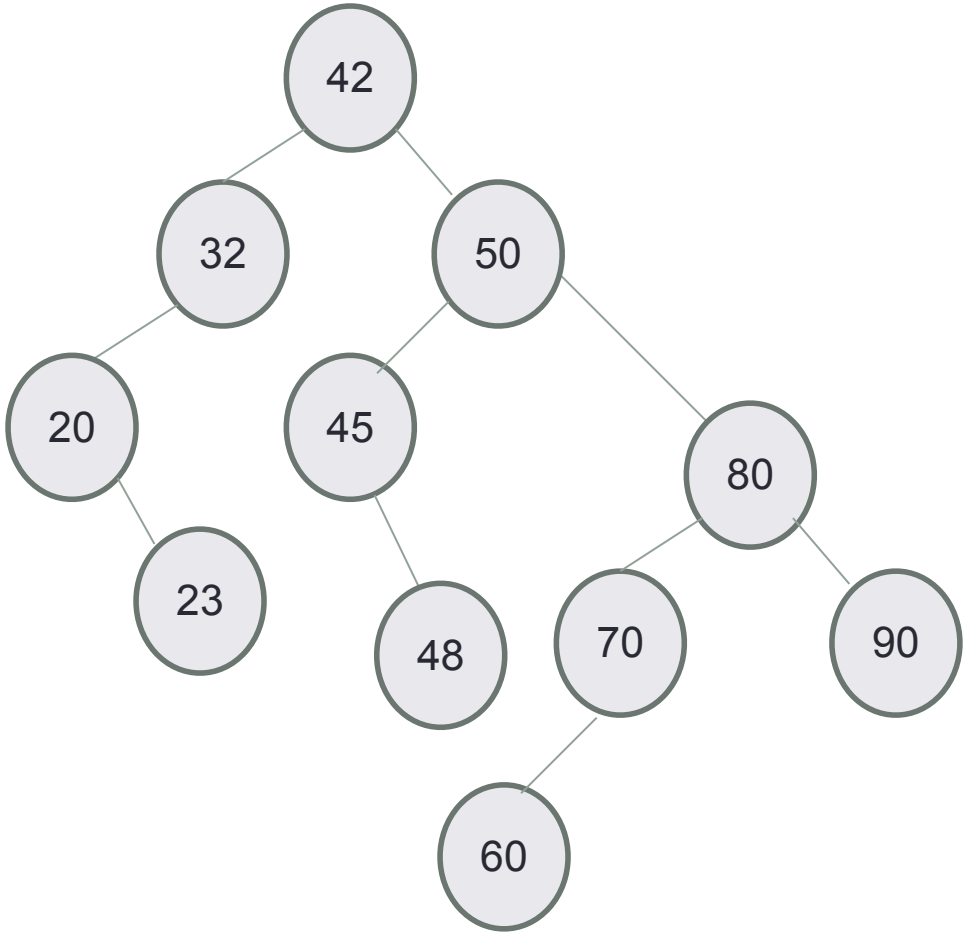
`getmin(root)`: returns pointer to Node with minimum value

`successor(n)`: returns pointer to the next Node (after n) in an in order traversal

Nodes visited in an inorder traversal:
20, 25, 30, 35, 40, 60



Discover the algo for successor



Your turn (10 min): Work through handout 1.1-1.3

Task 1.1: Implement getmin to return the leftmost node in a subtree.

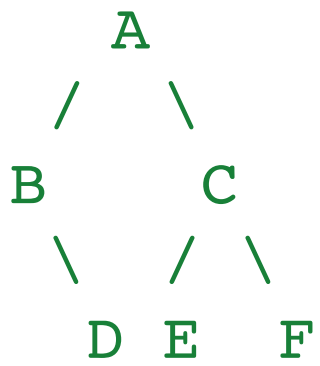
```
Node* bst::getmin(Node* r) const {
    // Fill in the code



}
```

Task 1.2: Discover the Successor Algorithm

Consider BST with keys masked by labels:



Succesor of A? _____

Successor of D? _____

What steps did you take in each case?

Task 1.3: Implement `successor`

```
Node* bst::successor(Node* r) const {
    // Fill in the code



}
```

Brainstorm next steps

We can now write an iterative traversal for bst

```
Node* r = b.getmin(root);
while(r){
    cout << r->data << " ";
    r = b.successor(r);
}
```

Problem we encountered before: `Node` is private, so `Node*` can't be used externally.

Big idea: Create a new ADT `iterator` that behaves like a traversal pointer.

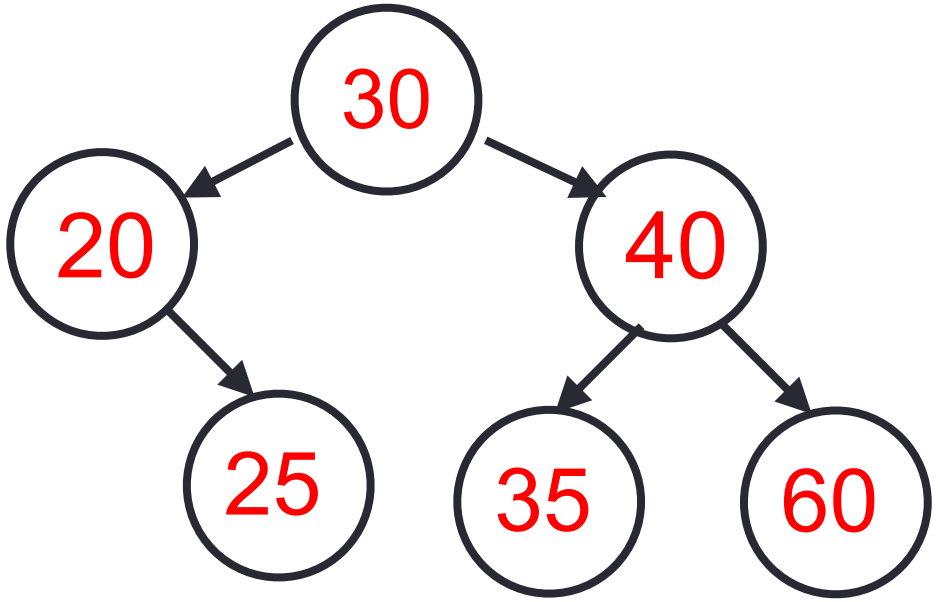
Big idea: Create a new ADT called `iterator` that behaves like a pointer.

```
class bst::iterator {
  public:




  private:


};
```

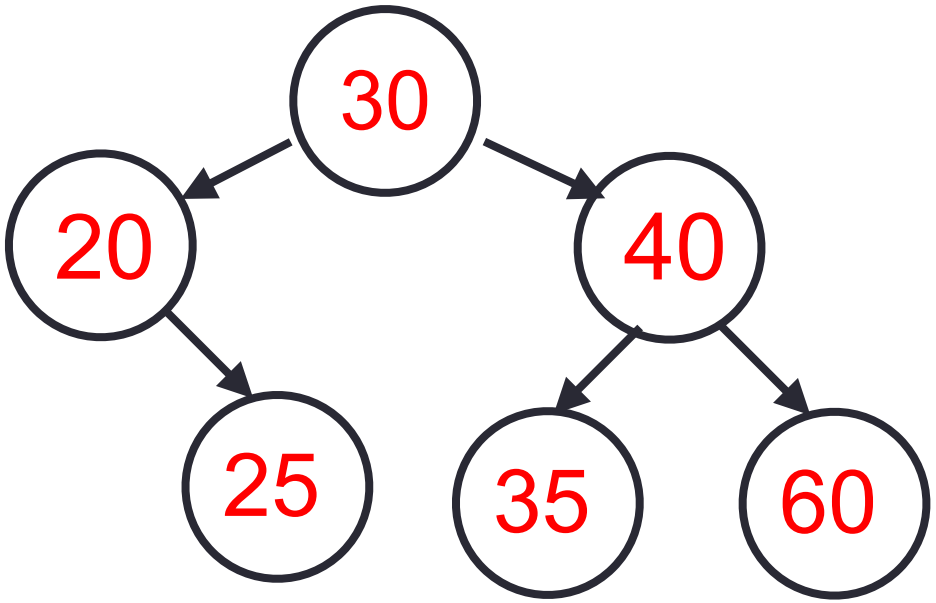



```
bst::iterator it;
*it = ____ (data)
++it; // Moves to ____
```

(5 min) Convert code to use iterator ADT

```
Node* r = b.getmin(root);
while(r){
    cout << r->data << " ";
    r = b.successor(r);
}
```

Discuss: What problem(s) do you encounter?



BST Helper functions to initialize iterators

Task 3.1: Implement `begin`: Returns an iterator to the smallest node.

```
bst::iterator bst::begin() {
    // Fill in the code




}
```

Task 3.2: Implement `end`: Returns an iterator for “past the end.”

```
bst::iterator bst::end() {
 // Fill in the code



}
```

Task 4.1: Implement `operator*`

```
int bst::iterator::operator*() const {
    // Fill in the code



}
```

Task 4.2: Implement `operator++`

```
bst::iterator& bst::iterator::operator++() {
    // Fill in the code



}
```

Task 4.3: Implement `operator!=`

```
bool bst::iterator::operator!=(const iterator& rhs) {
    // Fill in the code


}
```

C++STL

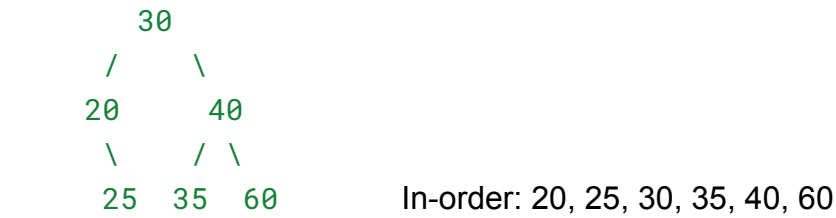
- The C++ Standard Template Library is a handy set of three built-in components:
  - Containers: Data structures
  - Iterators: Standard way to traverse containers
  - Algorithms: These are what we ultimately use to solve problems

In this lecture, you learned how to implement an iterator for any custom ADT. Useful for working with STL classes and writing clean code in the upcoming assignment (PA01) where you have to implement a card game.
The big challenge is to iterate through the cards of two players in a seamless way (no passing around pointers like Node* in the main logic of your game). Use iterators!

Iterators: An ADT specialized for traversal

Objective: Implement an iterator for our `bst` class, enabling flexible, one-at-a-time navigation like `std::set`. We'll build 6–8 functions and a new `iterator` type, culminating in a range-based for loop for `bst`. We'll explore the successor algorithm with a magic trick!

Example BST (use this for all tasks):



In-order: 20, 25, 30, 35, 40, 60

Overview: Recursive vs. Iterative traversal of a BST

Last lecture, we implemented `printInorder` for `bst.`

```
void bst::printInorder(Node *r) const{
    if (!r) return;
    printInorder(r->left);
    cout << r->data << " ";
    printInorder(r->right);
}
```

Compare to how `std::set`'s range-based for loop iterates through keys of the bst:

```
std::set<int> s = {30, 20, 25, 40, 35, 60};
for (int x : s) {
    cout << x << " "; // Prints: 20 25 30 35 40 60
}
```

Discuss (2 mins): Why doesn't `std::set` have a `printInorder` function?

Our goal: Implement one-at-a-time navigation for custom BST

What you write:

```
for (int x : s) {
    cout << x << " "; // Prints: 20 25 30 35 40 60
}
```

What actually happens:

```
for (std::set<int>::iterator it = s.begin(); it != s.end(); ++it) {
    cout << *it << " ";
}
```

Roadmap to achieve our goal:

1. Implement useful helpers: `getmin` (smallest node) and `successor` (next node from a given node in an in-order traversal).

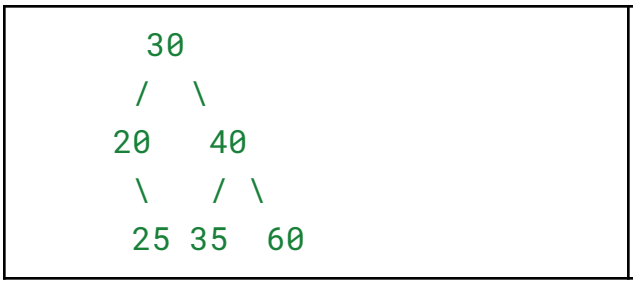
Let's assume we have the helper functions implemented correctly, then an iterative traversal should be possible as follows:

```
Node* r = b.getmin(root); //assume b is an object of bst
while(r){
   cout << r->data << " ";
   r = b.successor(r);
}
```

Discuss (2 min): Are we done? Why or why not?

2. **Big idea**! Implement a new ADT (`iterator`) that behaves like a traversal pointer.

Draw the `iterator`'s `++` (move to successor) and `*` (get data) for node 20:




```
iterator it; // Assume it
             // points to 20
*it = ____ (data)
++it; // Moves to ____
```

Discuss (2 min): What functions does iterator ADT need to implement to allow operations like `*it` and `++it`?

1. Implementing `getmin` and `successor`

Implement `getmin` (smallest node) and `successor` (next node from a given node in an in-order traversal).

Task 1.1: Implement getmin to return the leftmost node in a subtree.

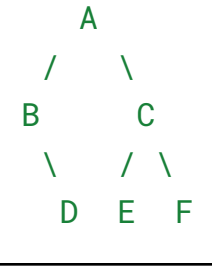
```C/C++
Node* bst::getmin(Node* r) const {
    // Fill in the code




}
```

Task 1.2: Discover the Successor Algorithm

Consider BST with keys masked by labels:

	Write the label in each case: Succesor of A? _____ Successor of D? _____

What steps did you take to find A's successor? Describe the pattern

What steps did you take to find D's successor? Describe the pattern.

Task 1.3: Implement `successor` (next node that appears in an in-order traversal).

```C/C++
Node* bst::successor(Node* r) const {
    // Fill in the code




}
```

Question: What should `successor` return for the maximum node (F)? Why?

Task 2.1 Brainstorm next steps

We can now write an iterative traversal (milestone one achieved)

```C/C++
Node* r = b.getmin(root); //Assume b is an object of bst
while(r){
   cout << r->data << " ";
   r = b.successor(r);
}
```

But there is a problem! `Node` is private, so `Node*` can't be used externally.

Big idea: Create a new ADT `iterator` that behaves like a traversal pointer

Iterator: An iterator is an abstract data type (ADT) that acts like a cursor, providing a simple and controlled way to traverse a data structure, such as a BST, one element at a time. It hides the internal details of the data structure (like nodes and pointers) and supports pointer-like operations (e.g., ++ to move to the next element, * to access the current element) to navigate through a data structure.

2. Iterator Class Definition

Big idea: Create a new ADT `iterator` that behaves like a traversal pointer.
Define the `iterator` class, including its constructor (implemented outside).
It needs to store two pointers:

- `Node* curr`: Tracks the current node.
- `bst* tree`: Allows calling `successor`, a `bst` method.

```C/C++
class bst::iterator {

 public:
    iterator(Node* p = nullptr, bst* ptr_tree = nullptr){




    }
    // Member functions: overloaded *, ++, and != operators









 private:
    // Member variables
    ------------------------------------------
    ------------------------------------------
};
```

(2min) Now, convert this code to use the iterator ADT instead of Node*.
Discuss any problems that you encounter.

`//Assume b is an object of bst`
`Node* r = b.getmin(root);`
`while(r){`
`  cout << r->data << " ";`
`  r = b.successor(r);`
`}`
	Write your code that uses iterator class

3. Implementing `begin` and `end (new bst functions)`

Task 3.1: Implement `begin`: Returns an iterator to the smallest node.

```
C/C++
bst::iterator bst::begin() {
    // Fill in the code



}
```

Task 3.2: Implement `end`: Returns an iterator for “past the end.” This just means iterator that stores a nullptr for the given BST

```
C/C++
bst::iterator bst::end() {
 // Fill in the code



}
```

4. Implementing Iterator Operators

Implement `iterator`'s operators: `*` (get data), `++` (move to successor), and `!=` (compare).

Task 4.1: Implement `operator*`

```C/C++
int bst::iterator::operator*() const {
    // Fill in the code



}
```

Task 4.2: Implement `operator++`

```C/C++
bst::iterator& bst::iterator::operator++() {
    // Fill in the code


}
```

Task 4.3: Implement `operator!=`

```C/C++
bool bst::iterator::operator!=(const iterator& rhs) {
    // Fill in the code


}
```

If everything is implemented correctly, this code should work!

Long form of the range-based for loop

```C/C++
bst::iterator it = mybst.begin();
while (it != mybst.end()) {
    cout << *it << " ";
    ++it;
}
```

Short-hand version of range-based for loop!

```C/C++
bst mybst; // Contains {30, 20, 25, 40, 35, 60}
for (auto e: mybst) {
    cout << e << " ";
}
```

Summary: The C++ Standard Template Library is a handy set of three built-in components:

- Containers: Data structures
- Iterators: Standard way to traverse containers
- Algorithms: These are what we ultimately use to solve problems

In this lecture, you learned how you can implement an iterator for any custom Abstract Data Type. Useful for working with STL classes and writing clean code in the upcoming assignment (PA01) where you have to implement a card game. The big challenge is to iterate through the cards of two players in a seamless way (no passing around pointers like Node* in the main logic of your game). Use iterators!