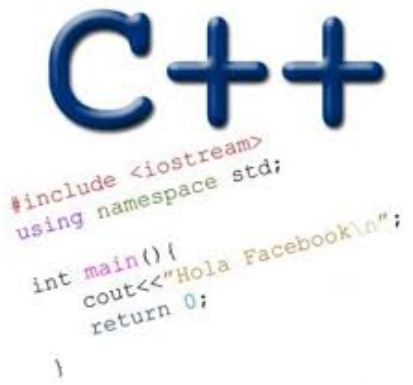
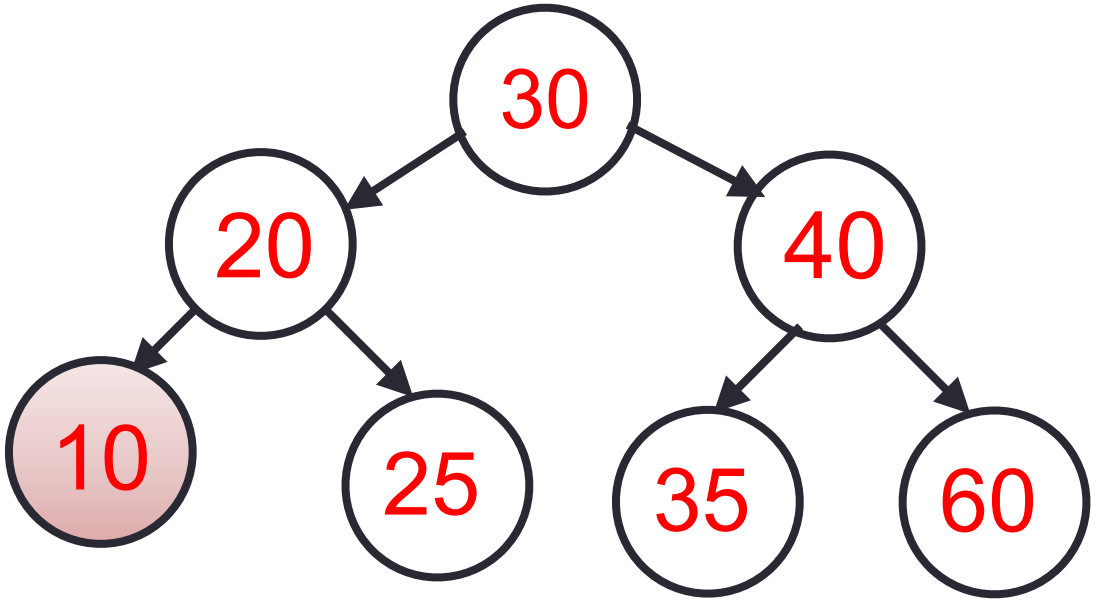
ITERATORS:
AN ADT SPECIALIZED FOR TRAVERSAL

Problem Solving with Computers-II



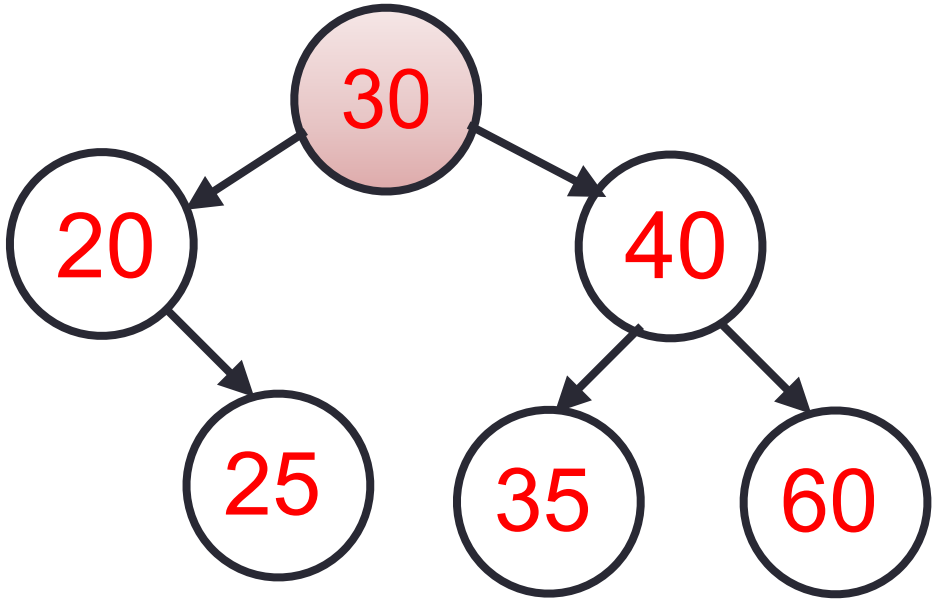
Min: Smallest value node



getMin():

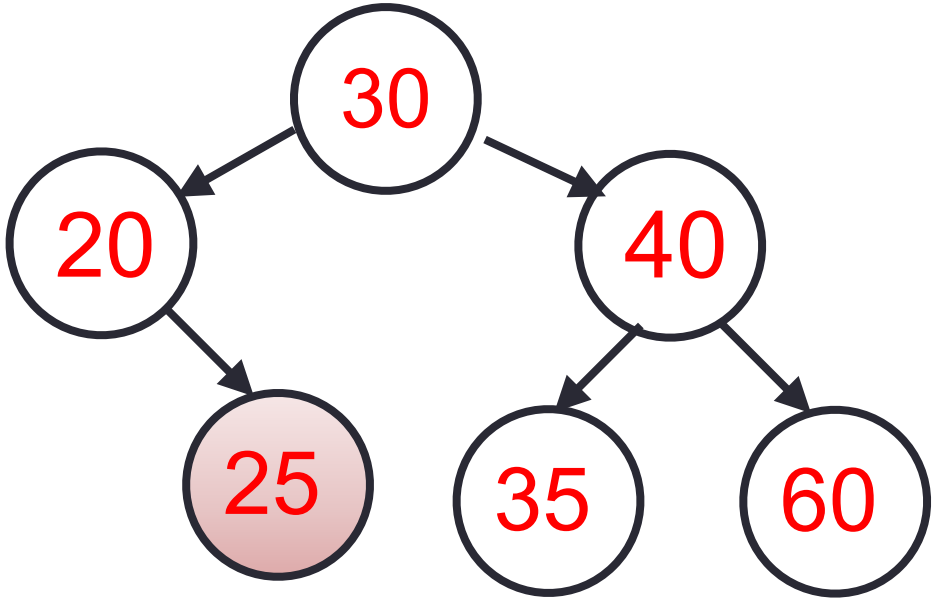
- Starting at the root,
- traverse all the way to the leftmost node in the BST
- return pointer to the leftmost node

Successor: Next largest element



- Case 1: Node (n) has a right child

```
Algo: Go to right child, then
traverse left as far as possible
```

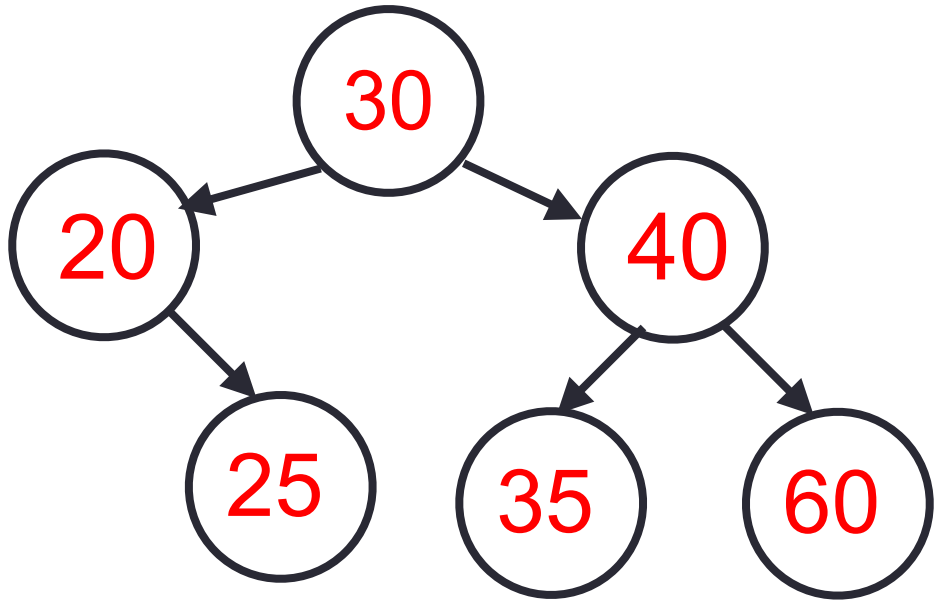


- Case 2: Node (n) has no right child

```
Algo: Go up until you find an
ancestor where current node is in its
left subtree
```

What does this code do?

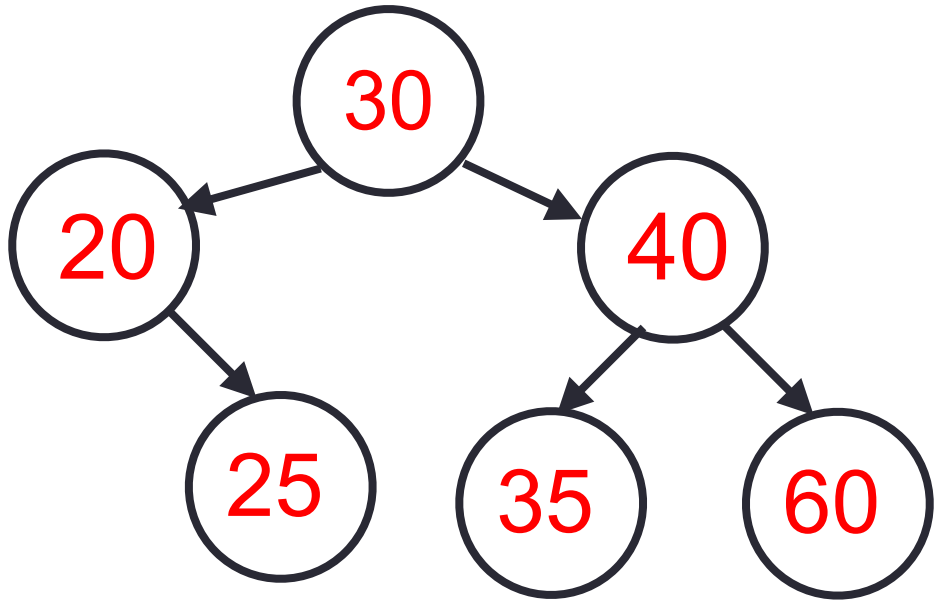
```
Node* r = b.getmin(root);
while(r){
    cout << r->data << " ";
    r = b.successor(r);
}
```



A. Prints the keys in the BST in order (smallest to largest)
B. Prints the keys in preorder
C. Prints the keys in postorder
D. None of the above

What does this code do?

```
std::set<int> s = {30, 20, 60, 35, 40, 25}
auto it = s.begin();
while(it != s.end()) {
    cout << *it << " ";
    it++;
}
```



A. Prints the keys in the BST in order (smallest to largest)
B. Prints the keys in preorder
C. Prints the keys in postorder
D. None of the above

Compare two different ways of iterating through a BST

```
std::set<int> s;
//insert keys into bst

auto it = s.begin();
while(it != s.end()) {
    cout << *it << " ";
    it++;
}
```

Code A

```
bst b;
//insert keys into bst

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

Code B

Why do you think the standard library designers chose code A to iterate through a BST instead of code B?

Why the standard library designers chose code A for iterating through a bst

```
std::set<int> s;
//insert keys into bst

auto it = s.begin();
while(it != s.end()) {
    cout << *it << " ";
    it++;
}
```

Code A

```
bst b;
//insert keys into bst

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

Code B

- Abstraction hides implementation details (no exposed `Node*`)
- Uniform interface across all containers
- Compatibility with STL algorithms
- Encapsulation (users don't need to know about `successor()`)

Iterator: An abstraction for traversal

An iterator is an abstraction for traversing any data structure.

Operation	Meaning
`begin()`	Where do I start?
`end()`	Where do I stop?
`++`	Move to the next element
`*`	Get key of current element
`==`	At same element?

Why are iterators useful?

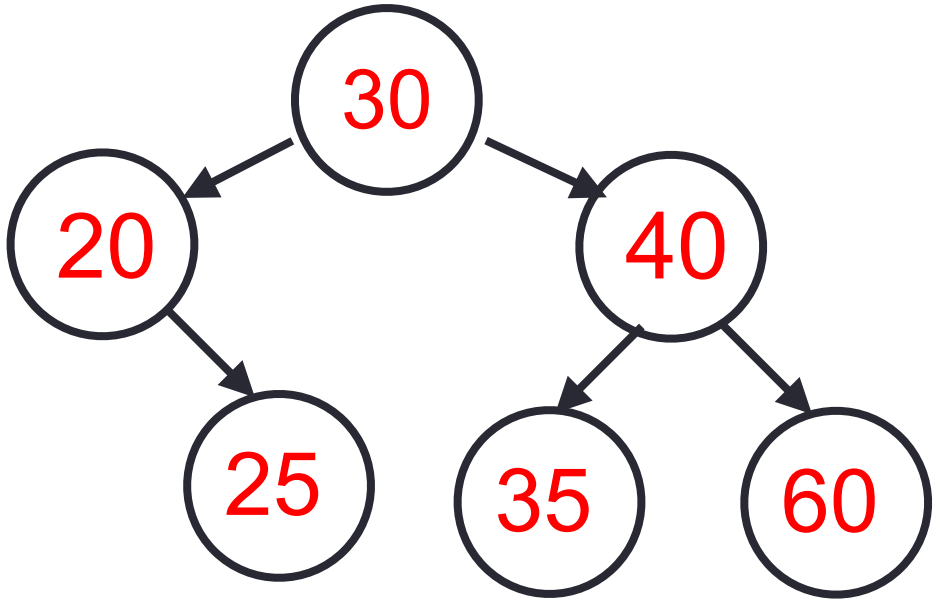
```
vector<int> v = {1, 2, 3, 4, 5};
list<int>   l = {1, 2, 3, 4, 5};
set<int>    s = {1, 2, 3, 4, 5};

// Same function call, same result (15)
accumulate(v.begin(), v.end(), 0);
accumulate(l.begin(), l.end(), 0);
accumulate(s.begin(), s.end(), 0);
```

Iterators separate what you do with elements from how you get to them

Big idea: Design a new `iterator` class for custom bst

Activity 2: Trace the iterator's ++ (move to successor) and * (get data) for node 20:

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



BST Helper functions to initialize iterators

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

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



}
```

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

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



}
```

Implement `operator*`

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



}
```

Implement `operator++`

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



}
```

Implement `operator!=`

```
bool bst::iterator::operator!=(const iterator& rhs) const {
    // 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!