

C++ ITERATORS

HEAP DATA STRUCTURE

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook!";
    return 0;
}
```



C++STL

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

C++ Iterators behave like pointers

- Let's consider how we generally use pointers to parse an array

10	20	25	30	46	50	55	60
----	----	----	----	----	----	----	----

```

void printElements(int arr[], int size) {
    int* p = arr;
    for(int i=0; i<size; i++) {
        std::cout << *p << std::endl;
        ++p;
    }
}

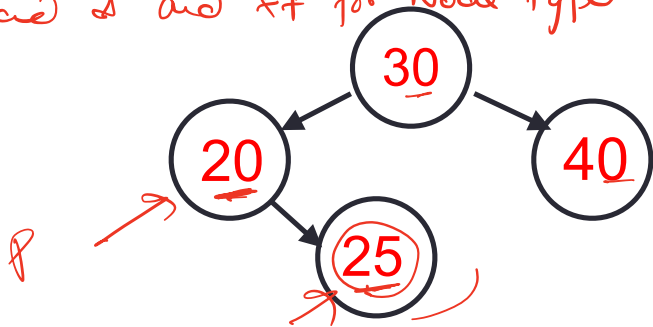
```

- We would like our print “algorithm” to also work with other data structures
- E.g Linked list or BST

Can a similar pattern work with a BST? Why or Why not?

Can we overload `++` and `*` for Node type

A. Yes
B. No



Possible types for p?

- X A: Node*
- X B: Node
- X C: int*
- D: iterator

```

void printElements(set<int>& s) {
    _____ //How should we define p?
    for(int i=0; i<s.size(); i++) {
        std::cout << *p << std::endl;
        ++p;
    }
}
  
```

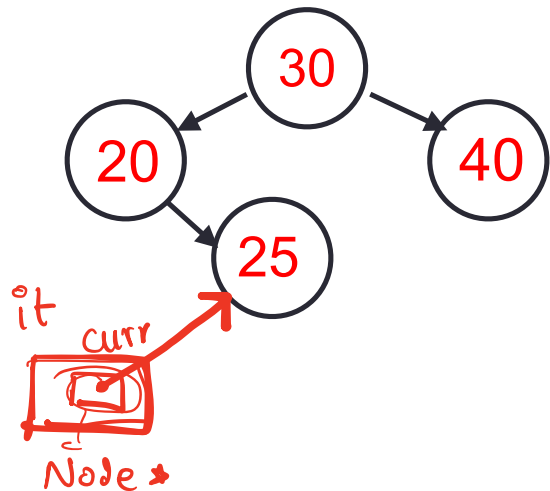
Iterators are objects that behave like pointers

```
set<int> s;
//insert keys 20, 30, 35, 40
set<int>::iterator it = s.begin();
it = s.find(25);
```

it++

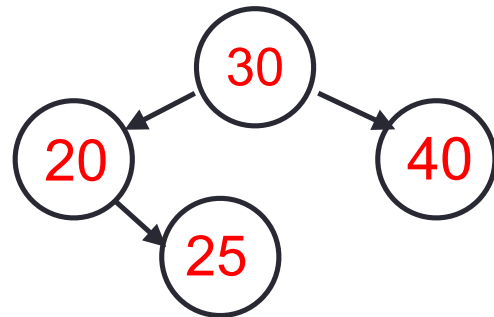
*.iterator::operator++ () {
curr = successor(curr);*

}



- “it” is an iterator object which can be used to access data in the container sequentially, without exposing the underlying details of the class

```
set<int> s;  
//insert keys 20, 30, 35, 40  
set<int>::iterator it;  
it = s.find(25);  
cout<<*it;
```



it

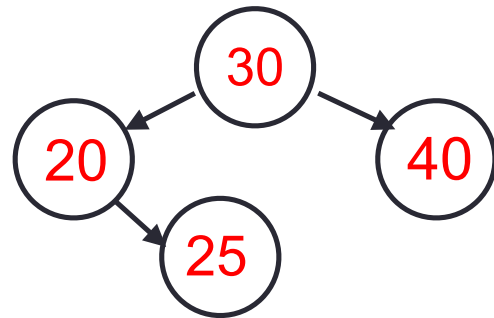


- “it” is an iterator object which can be used to access data in the container sequentially, without exposing the underlying details of the class

```

set<int> s;
//insert keys 20, 30, 35, 40
set<int>::iterator it;
it = s.find(25);
cout<<*it;
it++;

```



Which operators that must be overloaded for the iterator type?

- A. *
- B. ++
- C. <<
- D. All of the above
- E. Only A and B

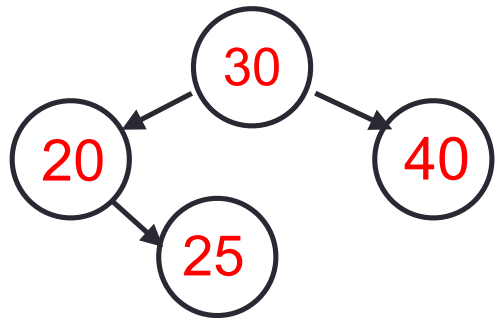
it

curr



C++ Iterators

```
void printElements(set<int>& s) {  
    set<int>::iterator it = s.begin();  
    set<int>::iterator en = s.end();  
    while(it!=en) {  
        std::cout << *it <<" ";  
        it++;  
    }  
    cout<<endl;  
}
```



C++ shorthand: auto

```
void printElements(set<int>& s) {  
    auto it = s.begin();  
    auto en = s.end();  
    while(it!=en){  
        std::cout << *it <<" ";  
        it++;  
    }  
    cout<<endl;  
}
```

Finally: unveiling the range based for-loop

```
void printElements(set<int>& s) {  
    for(auto item:s){  
        std::cout << item <<" ";  
    }  
    cout<<endl;  
}
```

PA02 Learning Goal

- Get familiarized with the STL documentation
- Select among available data structures

Check out the member functions of set and vector

<https://www.cplusplus.com/reference/set/set/set/>

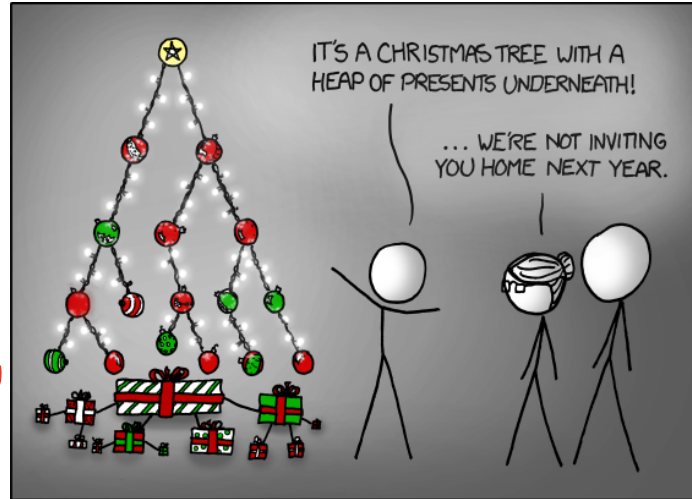
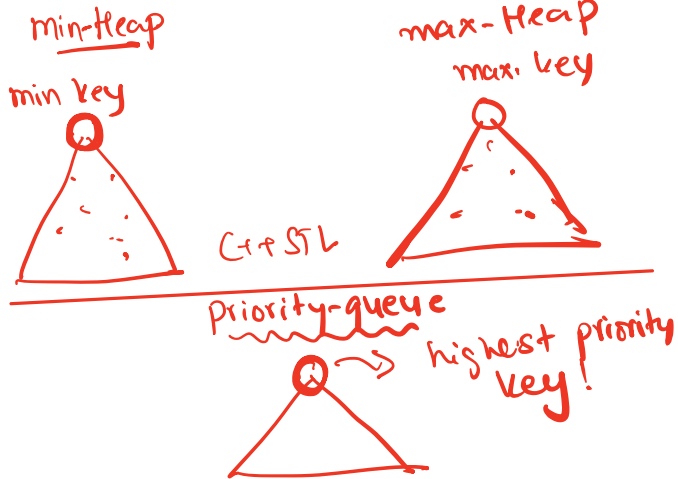
<https://www.cplusplus.com/reference/vector/vector/?kw=vector>

The complexity of each of the member functions is provided:

<https://www.cplusplus.com/reference/set/set/find/>

New data structure: Heap

- Clarification
 - *heap*, the data structure is not related to *heap*, the region of memory
- What are the operations supported?
- What are the running times?



priority-queue

top() $O(1)$

push() $O(\log n)$

pop() $O(\log n)$

empty() $O(1)$

min-heap

min $O(1)$

insert $O(\log n)$

delete min $O(\log n)$

max-heap

max $O(1)$

insert $O(\log n)$

delete max $O(\log n)$

Heap

Min-Heaps

Max-Heap

BST

- Insert :
- Min:
- Delete Min:
- Max
- Delete Max

Applications:

- Efficient sort
- Finding the median of a sequence of numbers
- Compression codes

Choose heap if you are doing repeated insert/delete/(min OR max) operations

std::priority_queue (STL's version of heap)

A C++ `priority_queue` is a generic container, and can store any data type on which an ordering can be defined: for example `ints`, `structs (Card)`, `pointers` etc.

```
#include <queue>
```

```
priority_queue<int> pq;
```

Methods:

- * `push()` //insert
 - * `pop()` //delete max priority item
 - * `top()` //get max priority item
 - * `empty()` //returns true if the priority queue is empty
 - * `size()` //returns the number of elements in the PQ
- You can extract object of highest priority in $O(\log N)$
 - To determine priority: objects in a priority queue must be comparable to each other

STL Heap implementation: Priority Queues in C++

What is the output of this code?

```
priority_queue<int> pq;  
pq.push(10);  
pq.push(2);  
pq.push(80);  
cout<<pq.top();  
pq.pop();  
cout<<pq.top();  
pq.pop();  
cout<<pq.top();  
pq.pop();
```

A. 10 2 80

B. 2 10 80

C. 80 10 2

D. 80 2 10

E. None of the above

max-heap by default

std::priority_queue template arguments

```
template <
    class T,
    class Container= vector<T>,
    class Compare = less <T>
> class priority_queue;
```

The template for priority_queue takes 3 arguments:

1. Type elements contained in the queue.
2. Container class used as the internal store for the priority_queue, the default is **vector<T>**
3. Class that provides priority comparisons, the default is **less**

std::priority_queue template arguments

//Template parameters for a max-heap

```
priority_queue<int, vector<int>, std::less<int>> pq;
```

//Template parameters for a min-heap

```
priority_queue<int, vector<int>, std::greater<int>> pq;
```

Comparison class

- Comparison class: A class that implements a function operator for comparing objects

```
class compareClass{
    bool operator()(int& a, int & b) const {
        return a>b;
    }
};
```

Comparison class

```
class compareClass{
    bool operator()(int& a, int & b) const {
        return a>b;
    }
};
```

```
int main(){
    compareClass c;
    cout<<c(10, 20)<<endl;
}
```

What is the output of this code?

A. 1

B. 0 or false

C. Error

When $c(a, b)$ returns false, priority-queue class interprets that as a has higher priority than b .

STL Heap implementation: Priority Queues in C++

```
class cmp{
    bool operator()(int& a, int & b) const {
        return a>b;
```

When } $c(a, b)$ returns false, priority-queue class interprets that as a has higher priority than b . Here c is an object of cmp

```
};
priority_queue<int, vector<int>, cmp> pq;
pq.push(10);
pq.push(2);
pq.push(80);
cout<< pq.top();
pq.pop();
cout<< pq.top();
pq.pop();
cout<< pq.top();
pq.pop();
```

Output: 2, 10, 80

pq is a min-heap