

COMP 220

Lecture Notes 08

Structural Traversal of Stacks, Queues, Maps, and Sets

October 11, 2016

In these notes we look at structural traversal patterns for our new non-vector typed ADTs.

Loop-Based Traversal

Traversing stacks, queues, maps, and sets is different than traversing a vector or a string. We cannot simply count off index values.

std::stack

You cannot traverse a `std::stack` without mutating the stack. You must pop the top in order to get to the next. In general, do not count based on the stack size as the the size changes everytime you push and pop.

```
1  std::stack< ... > st;
2  ...
3  while( !st.empty() ){
4      ...st.top()... //non-destructive top checking
5
6      ...st.pop()... // the ++ of stack traversal
7  }
```

Figure 1: Top to bottom traversal of a stack

std::queue

Like stack traversal, queue traversal is a destructive process. You must remove the front in order to get to the next front.

```
1  std::queue< ... > qu;
2  ...
3  while( !qu.empty() ){
4      ...qu.front()... //non-destructive front checking
5
6      ...qu.pop()... // the ++ of queue traversal
7  }
```

Figure 2: Front to Back traversal of a queue

std::map

A map only gets a sense of order from its implementation. Abstractly, they are best viewed as a set of key+value pairs. The `std::map` implementation effectively orders data in key order. The simplest way to traverse a `std::map` is with the range-based for loop ¹. This loop presents you with a `std::pair<kt,vt>` where `kt` is the key type and `vt` is the value type for the map².

Declaring the elements of the map as pairs will present you with copies of the map data.

¹ aka for-each loop

² <http://www.cplusplus.com/reference/utility/pair/>

```

1  std::map<kt,vt> mp;
2  ...
3  // for each key value pair kvp in map mp...
4  for( std::pair<kt,vt> kvp : mp ){
5      ...kvp.first... //copy of the key
6      ...kvp.second... //copy of the value
7      ...mp[kvp.first]... //the actual value
8  }
```

Figure 3: Map Traversal with by-value pairs

For efficiency reasons or because we intend to do a mutation while we traverse we might want access to the actual key and value data through the pair. Keys cannot be mutated but values can. Thus, we use `const` type keys and a by-reference pair.

```

1  std::map<kt,vt> mp;
2  ...
3  // for each key value pair kvp in map mp...
4  for( std::pair<const kt,vt>& kvp : mp ){
5      ...kvp.first... //const reference to the key
6      ...kvp.second... //reference to the the value
7      ...mp[kvp.first]... //same as kvp.second
8  }
```

Figure 4: Map Traversal with by-reference pairs

Finally, we can avoid the types of the pair altogether by using the C++11 `auto` type³.

std::set

Set traversal is also best accomplished via range based for loops. This traversal is more straight forward than map traversal.

If the copy cost of a by-value traversal is a concern than you can do a constant reference traversal as well.

You cannot to a set traversal that allows you to mutate the set

³ You should treat this as a way to simplify complex type declarations and not a way to dodge a lack of knowledge of type

```

1  std::map<kt,vt> mp;
2  ...
3  // for each key value pair kvp in map mp...
4  for( const auto& kvp : mp ){
5      ...kvp.first... //const reference to the key
6      ...kvp.second... //reference to the the value
7      ...mp[kvp.first]... //same as kvp.second
8  }

```

Figure 5: Read only Map Traversal with const-reference auto-typed pairs

```

1  std::set<t> s;
2  ...
3  // for each element of type t in set<t> s
4  for( t e : s ){
5      ...e... //copy of current set element e
6  }

```

Figure 6: Traversal of set by-value

```

1  std::set<t> s;
2  ...
3  // for each element of type t in set<t> s
4  for( const t& e : s ){
5      ...e... //copy of current set element e
6  }

```

Figure 7: Traversal of set by const-reference

elements. If you need to modify the set contents you should be using set removal and insert operations to remove an item and add the “new” value.

Recursion

Stacks and Queues

Once again, we have no ability to access the “rest” of a stack or a queue without discarding the first. The structures must be traversed through mutation. This means recursive procedures for this structures must pass the structures by reference.

```

1
2 ... stackfoo(std::stack<t>& st, ...){
3
4     if( st.empty() ){
5         ...
6         return ...;
7     }
8
9     ...st.top()...
10    ...st.pop()...
11    ...stackfoo(st,...)...
12
13    return ... ;
14 }
```

Figure 8: Recursive Traversal of Stack data

```

1
2 ... queuefoo(std::queue<t>& qu, ...){
3
4     if( qu.empty() ){
5         ...
6         return ...;
7     }
8
9     ...qu.front()...
10    ...qu.pop()...
11    ...queuefoo(qu,...)...
12
13    return ... ;
14 }
```

Figure 9: Recursive Traversal of Queue data

Maps and Sets

At this point you do not want to do recursive procedures for maps and sets. It requires utilizing C++ iterators and opens a can of worms we'd rather not open at this point.