Lecture 28 — Collections

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 28, 2016

Acknowledgments: D.W. Harder, W.D. Bishop

Collections

Collections are groupings of some variable number of data elements.

Each collection type implements a data structure and the algorithms associated with the data structure.

We're going to examine the following collections:

- List
- Stack
- Queue

Each of these collections may contain an arbitrary number of items.
    Although in practice, computer memory is not infinite.

List

We are all certainly familiar with the concept of a list.
The “to-do list” of tasks to complete is a very common example:

- Wash dishes
- Pick up dry cleaning
- Study for the final exam

The list may contain an arbitrary number of items and there is some concept of ordering: “wash dishes” is the first item in this list.

We can look at or access any item in the list.
There’s no rule that mandates washing the dishes before studying.

We can insert items anywhere in the list.
Though the usual case is to add them at the end of the list.

List

The list is a finite, ordered collection of values, in which repeated values are permitted. A list of integers, for example: {4, 19, 756, 4, 18}

The first object in the list is at the *front* of the list; the last element is at the *back* of the list.

A list supports several conceptual operations:

- Access the k^{th} element of the list
- Given a reference to an element of the list:
  - Access the next or previous element
  - Modify the current element
  - Remove the current element
  - Insert an element immediately before or after the current element

Working with the List

On a list of integers {4, 19, 756, 4, 18}, let's perform some operations.

Add 42 to the end of the list:
{4, 19, 756, 4, 18, 42}

Remove 19 from the list:
{4, 756, 4, 18, 42}

Insert 99 before 756:
{4, 99, 756, 4, 18, 42}

The stack and queue are also collections like a list, but with restrictions on how elements may be accessed, added, or removed.

A stack has a last-in, first-out policy: the only element that may be removed from the collection is the one most recently added.

A queue has first-in, first-out behaviour: the only element that may be removed is the one that has been in the queue the longest.

The Stack and Queue

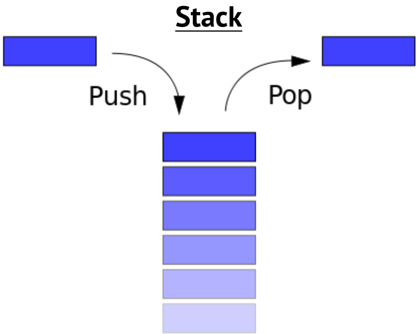

Image source: Shrimp/Wikipedia

`http://upload.wikimedia.org/wikipedia/commons/2/29/Data_stack.svg`

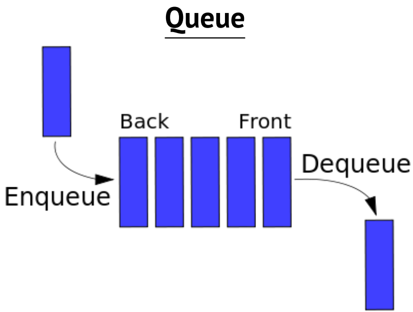

Image source: Vegpuff/Wikipedia

`http://upload.wikimedia.org/wikipedia/commons/5/52/Data_Queue.svg`

We have already introduced the stack when discussing memory.
Recall: the stack as a series of boxes piled one on top of another.

Let's continue that stacked boxes example:

- The box at the top is the only one you can use.
- When finished with a box, you take it off the top of the pile.
- Any new box to be added goes on top of the pile.
- It's not possible to take a box out of the middle.

The stack conceptually supports 3 operations: push, peek, and pop.

Push adds an element to the top of the stack.
Peek examines the element at the top of the stack.
Pop removes and returns the element at the top of the stack.

Note that peek does not remove the element at the top of the stack.

If we attempt to pop when there are no items on the stack, the result is an error called *stack underflow*.

If we attempt to push an item when there is no more space in the underlying representation, the result is a *stack overflow* error.

Working with the Stack

On the board, let's visualize how the stack works conceptually for the following sequence of operations. We start with an empty stack.

1. Push 74
2. Peek
3. Push 867
4. Push 97
5. Peek
6. Pop
7. Push 44
8. Peek
9. Pop
10. Pop
11. Pop
12. Pop

You've surely spent plenty of time in queues over the years.

- In common speech, we typically use the word "line".
- When you have to wait in a line for your turn, you are in a queue.

Example: waiting to order at a fast food restaurant.

- The customer at the front of the line is the next one served.
- Then the line moves up and someone else is at the front of the line.
- When a new customer arrives, s/he goes to the back of the line.
- Cutting in line (going out of turn) is not permitted.

Queue

The queue also supports 3 operations: enqueue, peek, and dequeue.

Enqueue adds an element to the end of the queue.
Peek examines the element at the front of the queue.
Dequeue removes and returns the element at the front of the queue.

Peek does not remove the element at the front of the queue.

If we attempt to dequeue when there are no items on the stack, the result is an error called *queue underflow*.

If we attempt to enqueue an item when there is no more space in the underlying representation, the result is a *queue overflow* error.

Working with the Queue

On the board, let's visualize the same sequence but for a queue. Again, starting with an empty queue.

1. Enqueue 74
2. Peek
3. Enqueue 867
4. Enqueue 97
5. Peek
6. Dequeue
7. Enqueue 44
8. Peek
9. Dequeue
10. Dequeue
11. Dequeue
12. Dequeue

List and Array

An array is very much like a list, although it is of fixed length. It has an ordering (the index), allows random access, insertion etc.

Arrays can be used to implement a list (although we will have to do resizing if the array gets full).

An array can also be used to implement a stack or queue.
You will get a chance to implement this in ECE 250.

For now, we won't write our own implementations of the collections.

Now that you are familiar with how they work conceptually, let us examine one of the built-in collections.

The C++ collection we will examine is the: `vector`.

The `vector` models a list and behaves like an array:
The index operators may be used to access array elements; but
Elements are added using an `push_back( )` function.

It is important to note that the type of objects added to the vector is not specified directly. Any type can be put in the collection.

Using vector

```
vector<int> v = {1, 5, 7};

v.push_back( 55 );
v.push_back( 99 );


for( int n : v ) {
    cout << n << endl;
}
```

Using `vector`

Every time we call `push_back( )`, we put the new element in the list, after the last already-present element.

Even though we initialized the `vector` with a capacity of 3, we were able to add 5 more elements to the list.

The collection increases its own capacity dynamically when items are added beyond its current capacity.

This is Strange...

We just saw a function `push_back( )` be inside the vector type.

This is weird!

This is because the victory is another kind of programmer-defined type... a `class`.