CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 4 – ADT, Containers, Sequence Containers

(Week 6, starting 12 September 2016)

1. Container ADT vs List ADT

A **Container** is a collection of objects. The C++ standard library has defined some containers.

Reference: http://www.cplusplus.com/reference/stl/

Some of these containers are **Lists** (not just the STL list class). In C++ terminology, they are termed sequence containers.

(a) What is the relationship between a Container and a List (sequence container)?

(b) What do both ADTs (Container vs sequence container) have in common?

(c) What extra functionality do List ADTs (sequence containers) have?

Answer

(a) A List inherits Container. Every List is a Container but not the other way round. Therefore, we expect a List to have additional functionality.

(b) Containers typically have `size()` and `empty()` functions, describing the **number of elements** within. They also have **iterators** to move through all elements from `begin()` to `end()`, insert, and delete data (`insert()`, `erase()`).

(c) Lists have a notion of sequence/ordering for each element. Therefore, they typically have member functions to allow **first** and **last** element access (`front()`, `back()`), and iterators in **reverse** order (`rbegin()`, `rend()`).

The big picture - Besides Lists, there are other Containers[1] such as:

- Queue - First element added will be first to be removed, not bothered about the middle
- Stack - Latest element to be added will be first to be removed, not bothered about the middle
- Map - A "lookup table" that gives you an associated value when you provide a key
- Set - A Container of unique elements
- Hash map/set (unordered) - Same functionality as (ordered) map and set, but different underlying implementation
- Priority queue - Element with highest priority first to be removed, not bothered about the middle

When we work with these other Containers, unlike a List, we are not primarily concerned with the ordering of every single element in the Container.

[1] CS2010 will allow you to understand how (an ordered) Map and Set works, and how it looks like in memory.

2. *List ADT Implementations*

In lectures, we have learned two List implementations – **array-based** and **reference-based** (i.e. in C++, using pointers). STL vector<T> is an array-based list implementation, while list<T> is a reference-based implementation. Let us **compare and contrast** both implementations.

For a list containing N elements, **how many elements would be accessed/modified** when:

- **(a)** Adding to end of the List (new index == N / tail)
- **(b)** Adding to front of the List (index == 0 / head)
- **(c)** Removing from front of the List (index == 0 / head)
- **(d)** Getting element at any index, on average (from index == 0 to index == N-1)

Answer

(a)
For **array-based** list, **most of the time** only **1** element is accessed/modified as no shifting is required. However, some lists allow their capacity (not just size) to grow dynamically. STL vector<T> is one such list.

For such lists, when the array is already at capacity, the entire underlying array has to be **reallocated**. All **N** elements from the original array have to be copied over to a new array of larger length. If efficiently implemented, when adding a very large number of elements, the **average** number of elements modified per add operation is **still a constant**.[2]

For a linked list, only 1 node is accessed, or 2 nodes if you also count the new node.

(b), (c)
Adding to front: In a typical **array-based** implementation, **N** elements from the insertion point onward have to be copied away from the front (to the right) before insertion. For a **linked list**, only **1 or 2** nodes are accessed/modified.

Removing from front: Efficiency the same as (b), N-1 elements left shifted instead for array-based impl.

(d)
In an **array-based** implementation, only **1** access is needed to move to any index, because arrays have random access. For a doubly linked list with head and tail pointers, the average number of accesses is **N/4** if your algorithm intelligently chooses to start from the nearer end.

Therefore, insertion/removal from an array-based list, except at one end (typically the back), is inefficient. On the other hand, array-based list is good for random access, unlike a linked list

- ☻ -

Have you been revising
and ***practicing*** daily?

[2] Such analysis is out of the scope of this course, and will only be learnt much later (in CS3230). For now, you just need to appreciate that adding to the back of an array-based list is often efficient, and generally efficient, but not every single time