ECE430.217 Data Structures

Lists

Weiss Book Chapter 3

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

We will now look at our first abstract data structure: list

- Relation: explicit linear ordering
- Operations
- Implementations of an abstract list with:
  - Linked lists
  - Arrays
- Memory requirements
- The STL `vector` class

Definition

An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

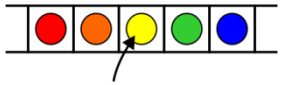
We will look at the most common operations

- The most obvious implementation is to **use either an array or linked list**
- These are, however, not always the most optimal

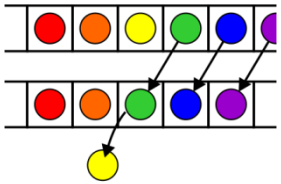
Operations

Operations at the k^{th} entry of the list include:

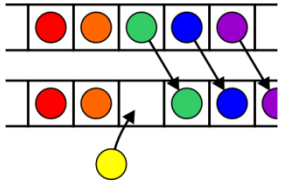
Access to the object



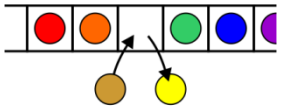
Erasing an object



Insertion of a new object

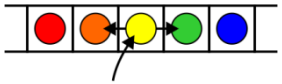


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

Locations and run times

The most obvious data structures for **implementing an abstract list** are **arrays** and **linked lists**

- We will review the run time operations on these structures

We will consider the amount of time required to perform actions such as 1) finding, 2) inserting new entries, or 3) erasing entries at

- the first location (the *front*)
- an arbitrary (k^{th}) location
- the last location (the *back* or n^{th})

The run times will be $\Theta(1)$, $O(n)$ or $\Theta(n)$

Linked lists

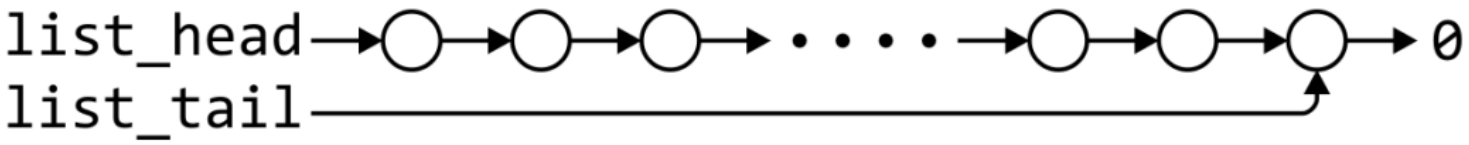
We will consider these for

- Singly linked lists
- Doubly linked lists

Singly linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

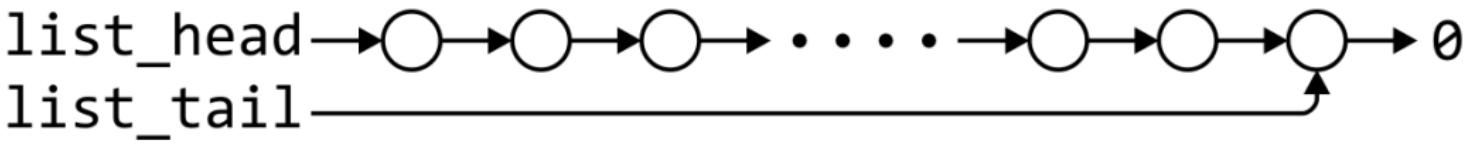
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Singly linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

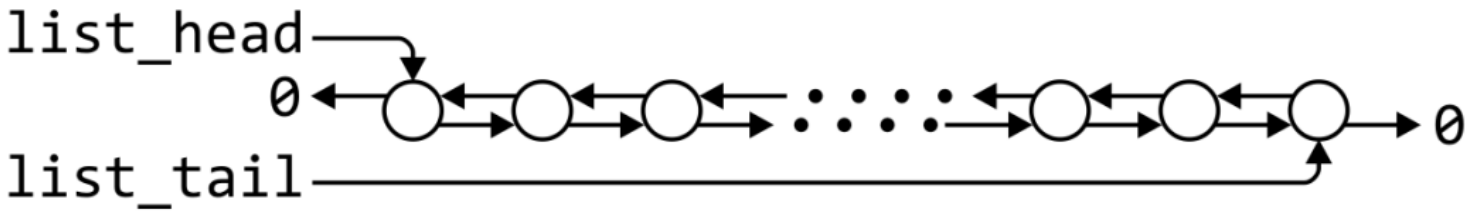
By replacing the value in the node in question, we can speed things up



Doubly linked lists

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Other operations on linked lists

Other operations on linked lists include:

- Allocation and deallocating the memory requires $\Theta(n)$ time
- Concatenating two linked lists can be done in $\Theta(1)$
  - This requires a tail pointer

Arrays

We will consider two array types:

- Standard or one-ended arrays

- Two-ended arrays

Run times

	Accessing the k^{th} entry	Insert or erase at the		
		Front	k^{th} entry	Back
Singly linked lists	$\Theta(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Standard Arrays	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		

* Assume we have a pointer to this node

Desired Run times

In general, we prefer to use a data structure, supporting $\Theta(1)$ execution time for majority operations

Memory usage versus run times

All of these data structures basically require $\Theta(n)$ memory

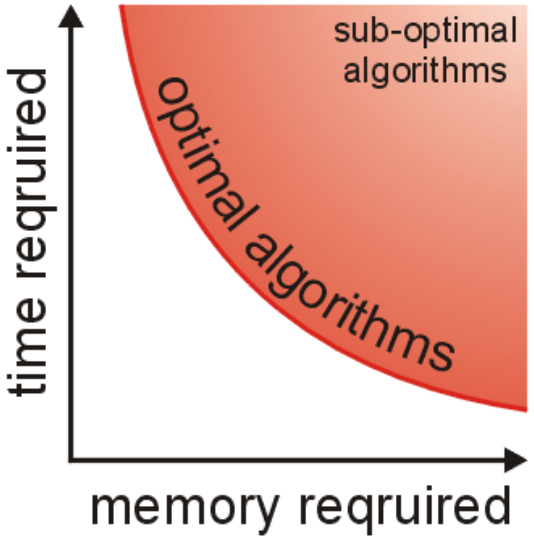
- Using a two-ended array requires one more member variable, $\Theta(1)$, in order to significantly speed up certain operations
- Using a doubly linked list, however, required $\Theta(n)$ additional memory to speed up other operations

Memory usage versus run times

In general, there is an interesting relationship between memory and time efficiency

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



Standard Template Library

In this course, you must understand each data structure and their associated algorithms

- In industry, you will use other implementations of these structures

The C++ Standard Template Library (STL) has an implementation of the `vector` data structure

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

Summary

In this topic, we have introduced Abstract Lists

- Explicit linear orderings
- Implementable with arrays or linked lists
  - Each has their limitations
  - Introduced modifications to reduce run times down to $\Theta(1)$
- Discussed memory usage