

Syrian Private University

Algorithms & Data Structure I

Instructor: Dr. Mouhib Alnoukari



Lists

NY MASHAR KONNY WINY WALKANA SUMMER AND SUMMER

Data Structure

Outline

This topic will describe:

- The concrete data structures that can be used to store information.
- The basic forms of memory allocation:
 - Contiguous
 - Linked
 - Indexed
- The prototypical examples of these: arrays and linked lists.
- Finally, we will discuss the run-time of queries and operations on linked lists.

– A data structure is defined by

- (1) the logical arrangement of data elements, combined with
- (2) the set of operations we need to access the elements.
- Atomic variables can only store one value at a time. int num;

float s;

A value stored in an atomic variable cannot be subdivided.

Memory allocation can be classified as either:

- Contiguous
- Linked
- Indexed

Prototypical examples:– Contiguous allocation: arrays– Linked allocation: linked lists

An array stores *n* objects in a single contiguous space of memory.

Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory.

In general, you cannot request for the operating system to allocate to you the next *n* memory locations.

Linked storage such as a linked list associates two pieces of data with each item being stored:

- The object itself, and

- A reference to the next item



For a linked list, however, we also require an object which links to the first object.

The actual linked list class must store two pointers:

– A head and tail:

Node *head;

Node *tail;

Optionally, we can also keep a count

int count;

The next_node of the last node is assigned nullptr





Matrices can be implemented using indexed allocation:



Matrices can be implemented using indexed allocation

 Most implementations of matrices (or higher-dimensional arrays) use indices pointing into a single contiguous block of memory.



C, Python

Column-major order



Matlab, Fortran

Double Linked List



A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$.

LIST-SEARCH(L, k) 1 x = L.head2 while $x \neq NIL$ and $x.key \neq k$ $\Theta(n)$ 3 x = x.next4 return x



Double Linked List - Insert



Double Linked List - Delete

LIST-DELETE(L, x)

- 1 **if** x.prev \neq NIL
- 2 x.prev.next = x.next O(1)
- 3 else L.head = x.next
- 4 **if** $x.next \neq NIL$
- 5 x.next.prev = x.prev



Once we have chosen a data structure to store both the objects and the relationships, we must implement the queries or operations as algorithms:

- The data structure will be defined by the member variables.
- The member functions will implement the algorithms.

The question is, how do we determine the efficiency of the algorithms?

We will us the following matrix to describe operations at the locations within the structure.

	Front/1 st	Arbitrary Location	Back/n th
Find	?	?	?
Insert	?	?	?
Erase	?	?	?

Given an sorted array, we have the following run times:

	Front/1 st	Arbitrary Location	Back	n^{th}
Find	Good	Okay	Goo	bd
Insert	Bad	Bad	Good*	Bad
Erase	Bad	Bad	Goo	bd

* only if the array is not full

If the array is not sorted, only one operations changes:

	Front/1 st	Arbitrary Location	Back/n th
Find	Good	Bad	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

ī.

However, for a singly linked list where we a head and tail pointer, we have:

	Front/1 st	Arbitrary Location	Back/n th
Find	Good	Bad	Good
Insert	Good	Bad	Good
Erase	Good	Bad	Bad



If we have a pointer to the k^{th} entry, we can insert or erase at that location quite easily

	Front/1 st	Arbitrary Location	Back/n th
Find	Good	Bad	Good
Insert	Good	Good	Good
Erase	Good	Good	Bad



 Note, this requires a little bit of trickery: we must modify the value stored in the kth node For a doubly linked list, one operation becomes more efficient:

	Front/1 st	Arbitrary Location	Back/n th
Find	Good	Bad	Good
Insert	Good	Good	Good
Erase	Good	Good	Good



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 that are usually

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

Operations at the kth entry of the list include: Access to the object Erasing an object

Insertion of a new object Replacement of the object

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 finding, inserting new entries before or after, or 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

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

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



Doubly linked lists

	Front/1 st node	k th node	Back/n th node
Find	$\Theta(1)$	O (<i>n</i>)	$\Theta(1)$
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 kth entry—an O(n) operation list_head 0 list_tail

Doubly linked lists

Accessing the k^{th} entry is O(n)

	<i>k</i> th node
Insert Before	Θ(1)
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$



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

As well as determining run times, we are also interested in memory usage

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



memory reqruired

Warning: programmers often mistake this to suggest that given any solution to a problem, any solution which may be faster must require more memory.

This guideline not true in general: there may be different data structures and/or algorithms which are both faster and require less memory.

- This requires thought and research.

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