Algorithms and Data Structures

Lec06

Lists

Dr. Mohammad Ahmad

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 arrays and linked lists

What is Data Structures?

– ***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

Memory allocation can be classified as either

- Contiguous
- Linked
- Indexed

Prototypical examples:

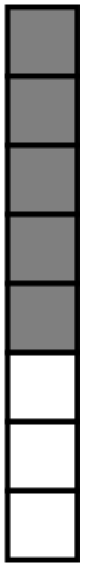
- Contiguous allocation: arrays
- Linked allocation: linked lists

Contiguous Allocation

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

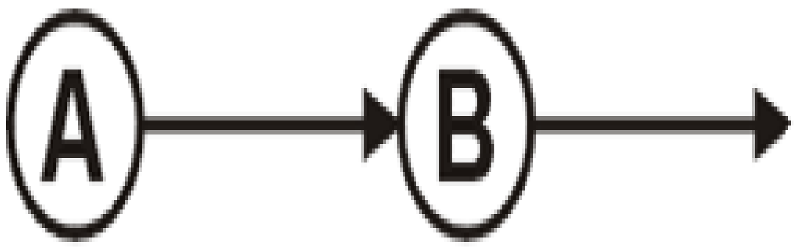
In general, you cannot request for the operating system to allocate to you the next n memory locations



Linked Allocation

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



Linked Allocation

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

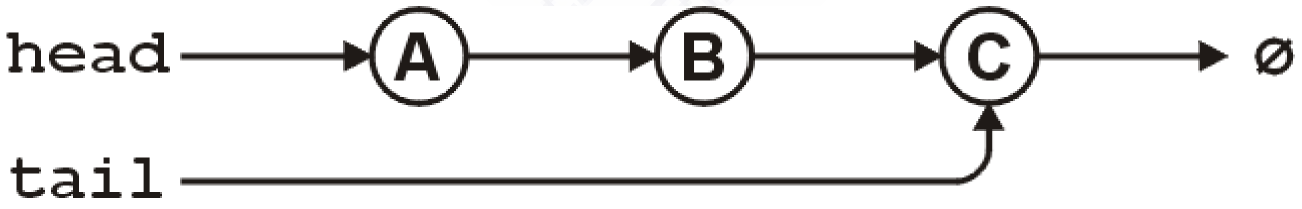
– A head and tail:

```
Node *head;
Node *tail;
```

Optionally, we can also keep a count

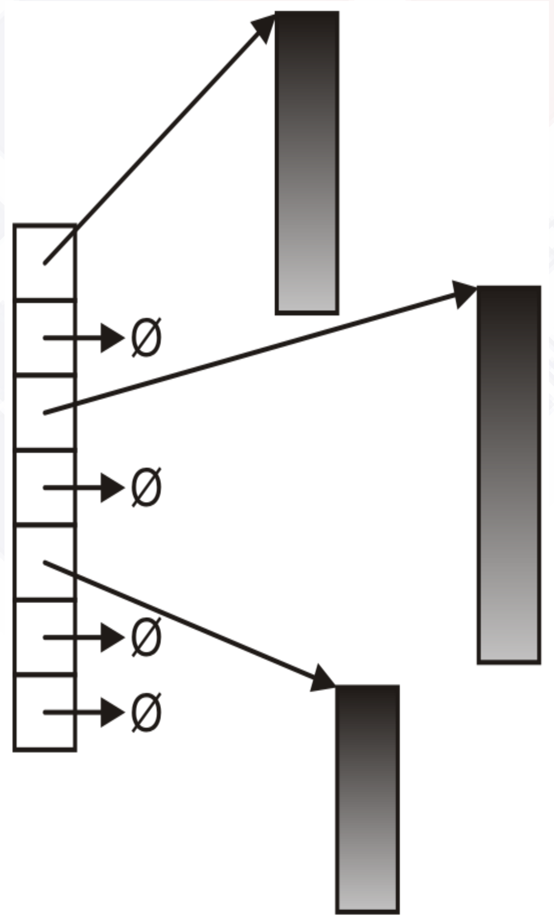
```
int count;
```

The `next_node` of the last node is assigned `nullptr`



Indexed Allocation

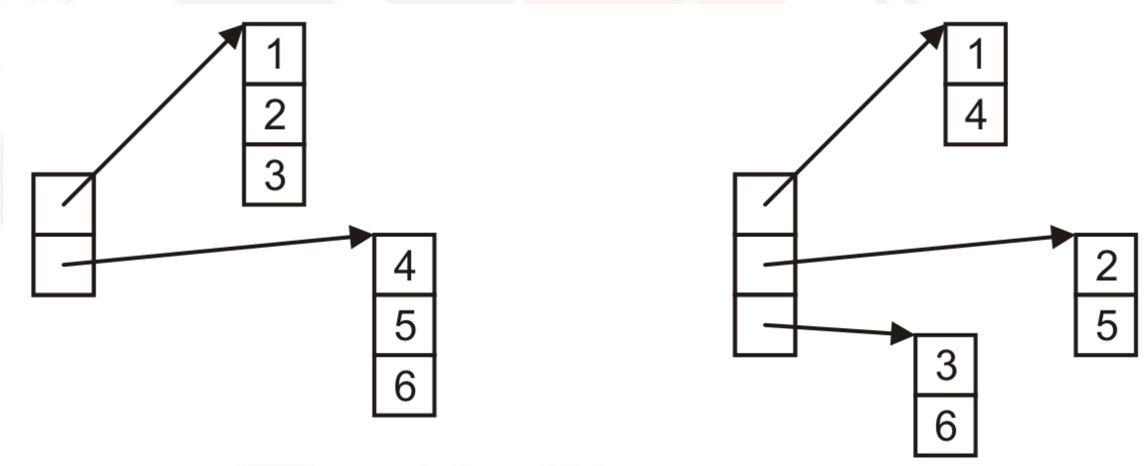
With indexed allocation, an array of pointers (possibly NULL) link to a sequence of allocated memory locations



Indexed Allocation

Matrices can be implemented using indexed allocation:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



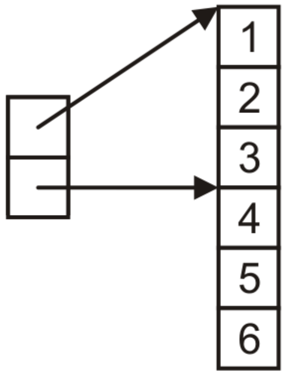
Indexed Allocation

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

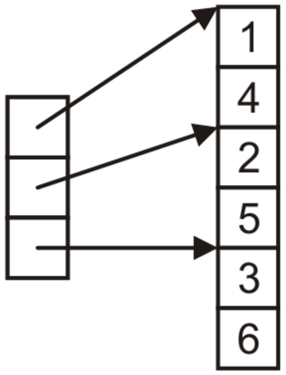
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Row-major order



C, Python

Column-major order



Matlab, Fortran

Algorithm run times

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?

Operations

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

	Front/1st	Arbitrary Location	Back/nth
Find	?	?	?
Insert	?	?	?
Erase	?	?	?

Operations on Sorted Lists

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

	Front/1st	Arbitrary Location	Back/nth
Find	Good	Okay	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

Operations on Lists

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

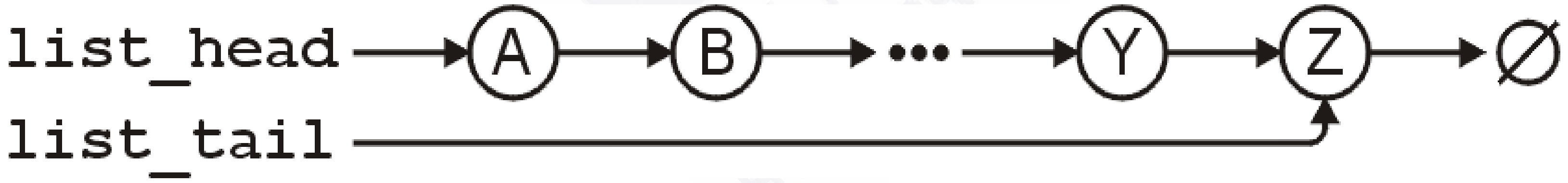
	Front/1st	Arbitrary Location	Back/nth	
Find	Good	Bad	Good	
Insert	Bad	Bad	Good*	Bad
Erase	Bad	Bad	Good	

* only if the array is not full

Operations on Lists

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

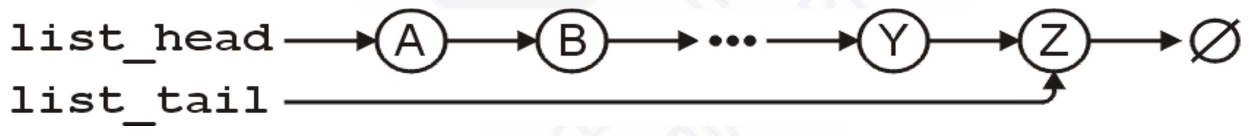
	Front/1st	Arbitrary Location	Back/nth
Find	Good	Bad	Good
Insert	Good	Bad	Good
Erase	Good	Bad	Bad



Operations on Lists

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

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

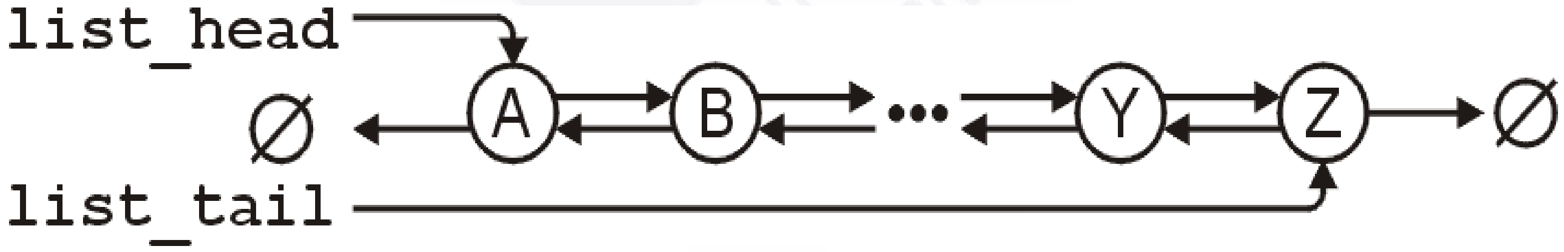


– Note, this requires a little bit of trickery: we must modify the value stored in the *k*th node

Operations on Lists

For a doubly linked list, one operation becomes more efficient:

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



Definition

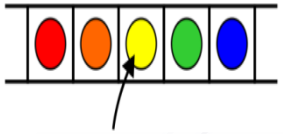
An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

We will look at the most common operations that are usually

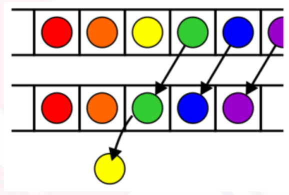
- The most obvious implementation is to use either an array or linked list
- These are, however, not always the most optimal

Operations

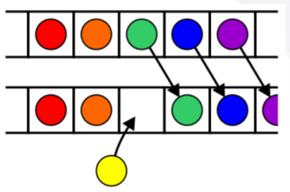
Operations at the k^{th} entry of the list include:



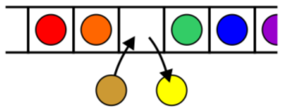
Access to the object



Erasing an object



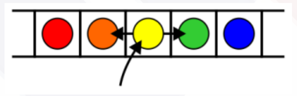
Insertion of a new object



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 finding, inserting new entries before or after, or erasing entries at

- the first location (the *front*)
- an arbitrary (kth) location
- the last location (the *back* or nth)

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

Linked lists

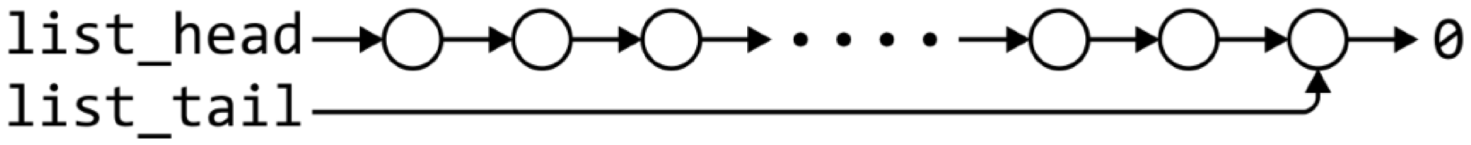
We will consider these for

- Singly linked lists
- Doubly linked lists

Singly linked list

	Front/1st node	kth node	Back/nth 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	$O(n)$	$\Theta(n)$

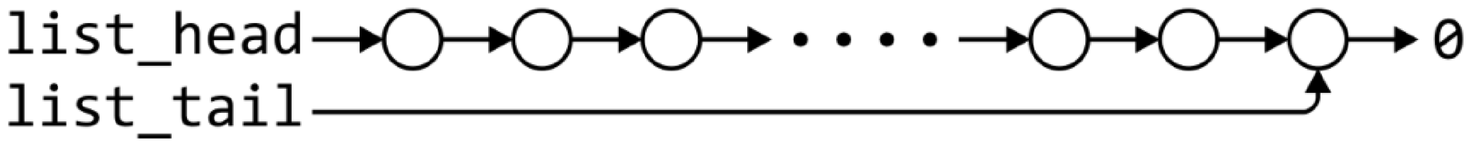
* These assume we have already accessed the kth entry—an $O(n)$ operation



Singly linked list

	Front/1st node	kth node	Back/nth 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	$O(n)$	$\Theta(n)$

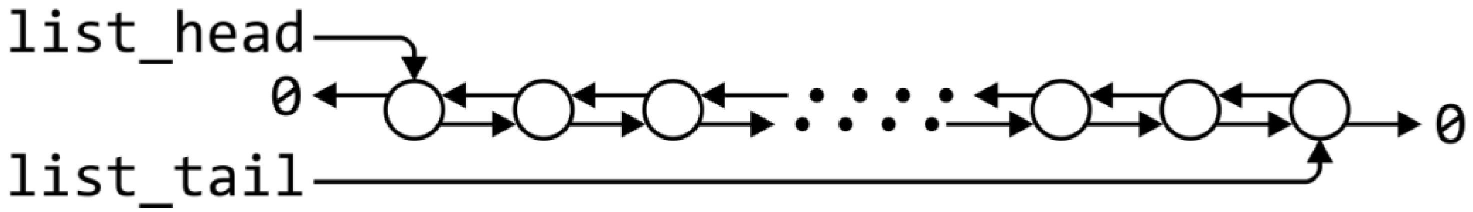
By replacing the value in the node in question, we can speed things up – useful for interviews



Doubly linked lists

	Front/1st node	kth node	Back/nth 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	$\Theta(1)$*	$\Theta(1)$

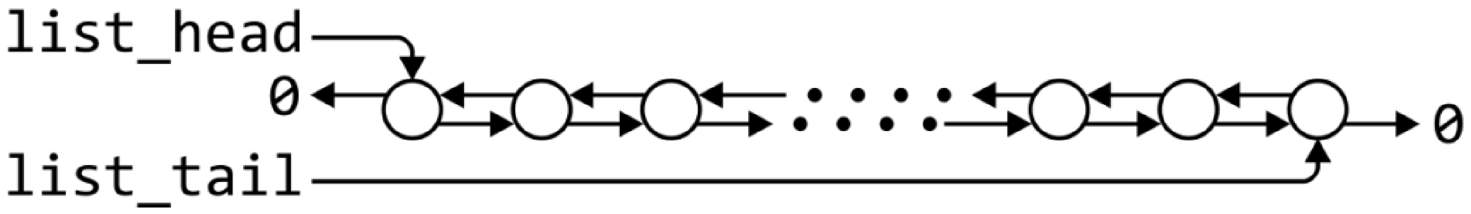
* These assume we have already accessed the kth entry—an $O(n)$ operation



Doubly linked lists

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

	k^{th} node
Insert Before	$\Theta(1)$
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

Arrays

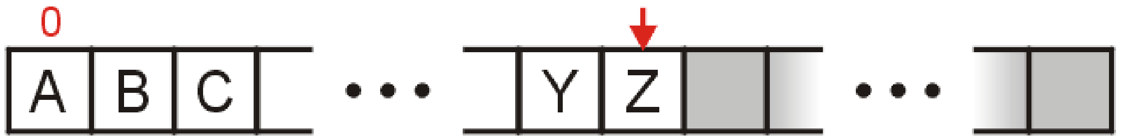
We will consider these operations for arrays, including:

- Standard or one-ended arrays
- Two-ended arrays

Standard arrays

We will consider these operations for arrays, including:

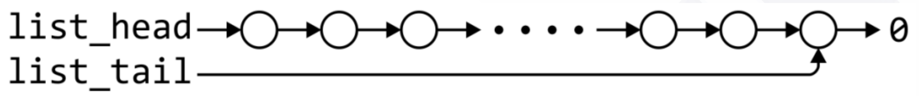
- Standard or one-ended arrays
- Two-ended arrays

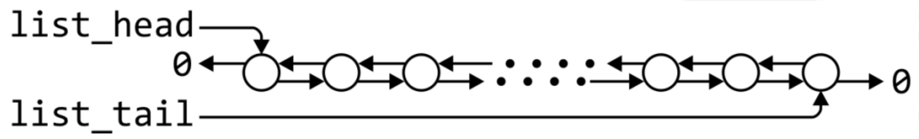


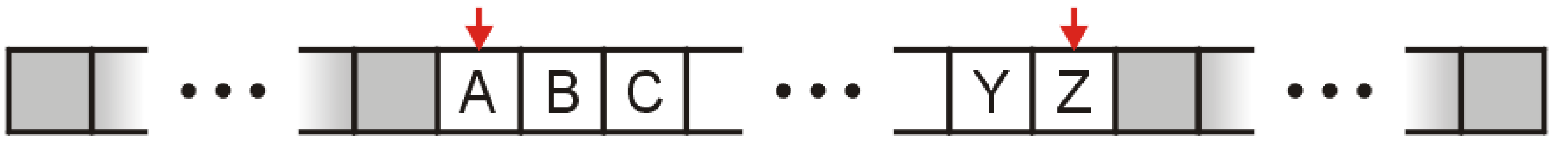
Run times

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

* Assume we have a pointer to this node







Memory usage versus run times

All of these data structures 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

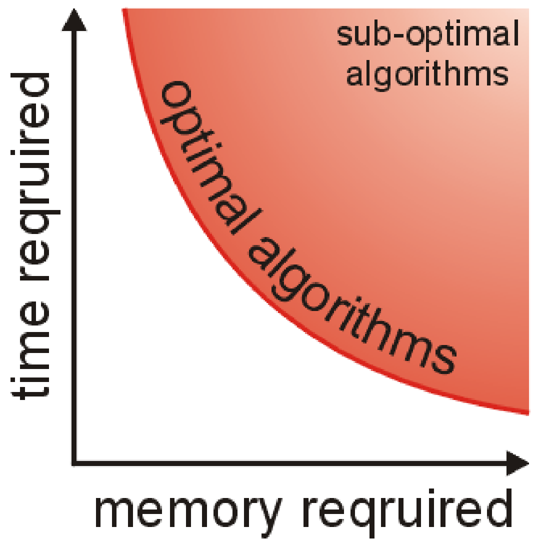
As well as determining run times, we are also interested in memory usage

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 usage versus run times

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

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