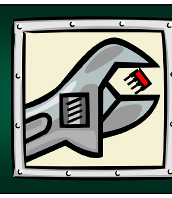
Abstract Data Types

Part 2

1



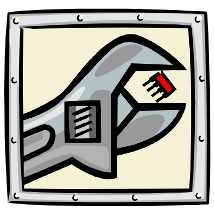
Data Structures

Return to CSC 15 and CSC 20

2

Data Structures

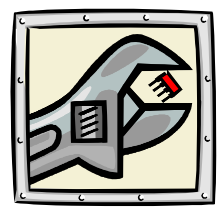
- Arrays and linked-lists are both examples of *data structures*
- These are different techniques of storing and organizing data
- In other words, this is *how* data is stored



3

Data Structures

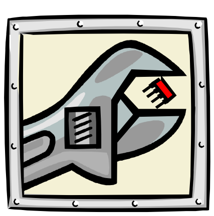
- Depending on *how* data is accessed, some data structures can either excel and falter
- This is true of both arrays and linked lists



4

Data Structures

- We will do a quick review of arrays and linked lists
- There are more data structures than these two
- We will cover them this semester – some which have incredible in features



5

Array Data Structure

Hidden math = easy code

6

Array Data Structure

- The array data structure is found in practically every programming language
- This is also one of the fundamental ways data is stored in memory

7

Behind the Scenes…

- Arrays are just continuous blocks of memory containing multiple instances of the same type
- Since the instances are continuous, values can be accessed randomly in **O(1)**

8

Array Math Example: 32-bit int

- Let's assume the array starts at address 2000
- Each array element will take 8 bytes (for 64-bit integers)
- Array elements are stored continuous

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353433335000000

9

Array Math Example: 32-bit int

- `array[0]` is 2000
- `array[1]` is 2008
- `array[2]` is 2016
- `array[3]` is 2024
- `array[4]` is 2032
- etc…

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353433335000000

10

Behind the Scenes…

- So, when an array element is read, internally, a mathematical equation is used
- It uses the start array, the array index, and the size of each element

```
start + (index × element_size)
```


11

Behind the Scenes…

- *This is why the C Programming Language uses zero as the first array element*
- If zero is used with this formula, it gets the start of the array

```
start + (index × element_size)
```


12

Auxiliary Storage in arrays

- Also, because elements are calculated, there is no extra storage overhead based on the array size
- So, the *auxiliary storage* overhead is O(1)

13

Resizing Arrays

- A *dynamically allocated array* is resized anytime an object is added or removed
- Because arrays require all elements to be stored continuously…

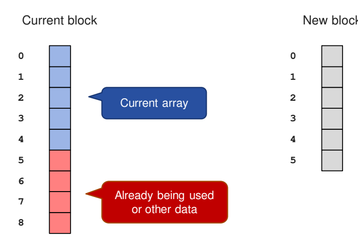
14

Resizing Arrays

- …the old block of memory (old array) needs to be copied to a new one
- This is extremely costly in both time and resources

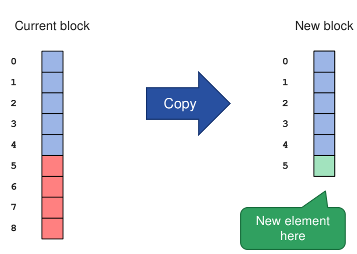
15

Arrays in Memory



16

Copy Values to New Block



17

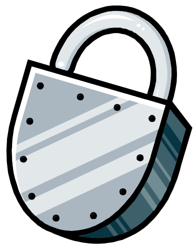
Resizing Arrays is O(n)

- While reading / writing elements takes only O(1)…
- … every time an array is resized, it will require **O(n)** time to copy the old array to the new one

18

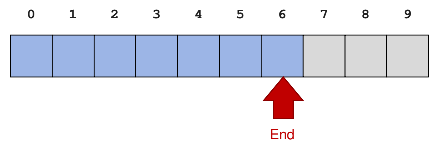
Fixed-Sized Arrays

- Arrays can also have a fixed sized called a *capacity*
- In this case, the array is never resized
- The array is often only partially filled
- An "end" index is maintained



19

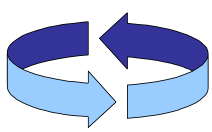
Fixed-Size Array



20

Fixed-Size Wrapping Around

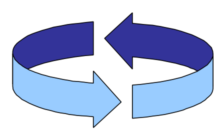
- Sometimes, you might need an array that wraps
- These are useful if both the first and last items can be removed
- … or older items can be discarded if space is needed



21

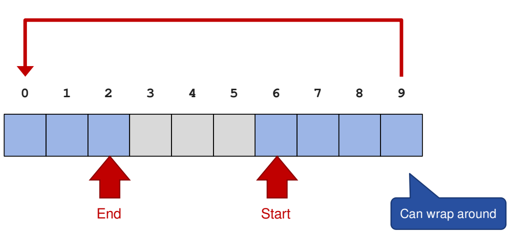
Fixed-Size Wrapping Around

- In addition to a "end" index, a "start" index is maintained
- Once the end of the array is reached, the array "wraps" to index 0
- … and continues until end is reached



22

Fixed-Size Array



23

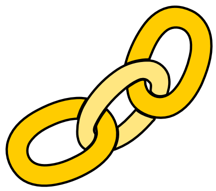
Linked List Data Structure

CSC 20's Revenge

24

Linked List Data Structure

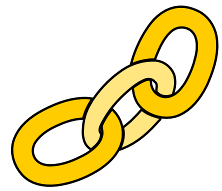
- Linked lists are a fundamental data structure that was covered in CSC 20
- Data is stored in a series of nodes which are connected with links



25

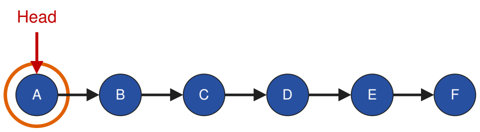
Linked List Data Structure

- Unlike arrays, where the element can be found using a calculation, linked-lists require the list to be traversed
- So, finding an item in a linked list requires **O(n)**



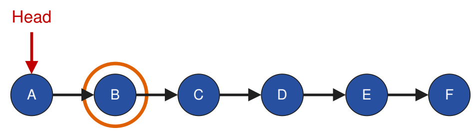
26

Single-Linked List – Find D



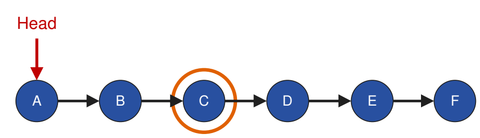
27

Single-Linked List – Find D



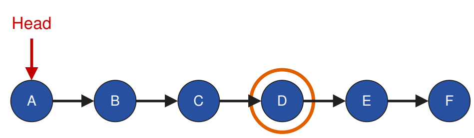
28

Single-Linked List – Find D



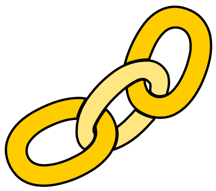
29

Single-Linked List – Find D



30

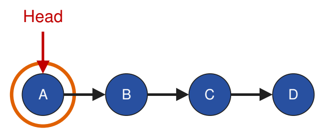
Head and Tail Nodes



- Linked lists maintain a link to the head node
- Often, in well-written linked lists, a link to the tail node is also maintained
- Why? It has a huge impact on time complexity

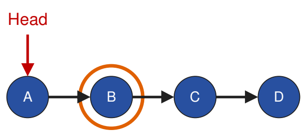
31

Append Value – No Tail Node



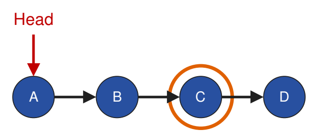
32

Append Value – No Tail Node



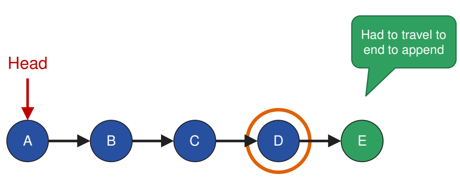
33

Append Value – No Tail Node



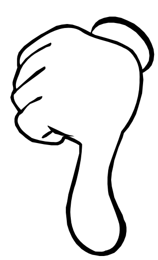
34

Append Value – No Tail Node



35

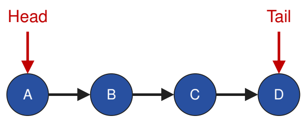
Head and Tail Nodes



- Without a tail node, the entire list must be traversed to find the end
- This will require **O(n)**
- Adding a tail node, will decrease it to **O(1)**

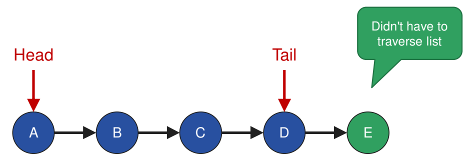
36

Append Value – With Tail Node



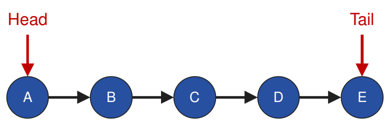
37

Append Value – With Tail Node



38

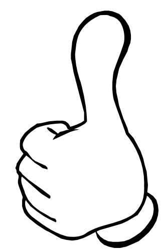
Append Value – With Tail Node



39

Use a Tail Node!

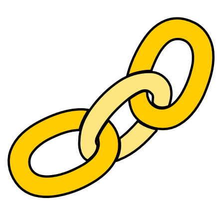
- Unless you are only appending nodes at the head of a linked list, maintain a tail node
- For **all** the examples used in these slides… assume the linked list has a tail node



40

Auxiliary Storage in Linked Lists

- Unlike arrays, linked lists must store the "next" links between nodes
- So, the *auxiliary storage* overhead is O(n)
  - …which is usually the size of an address
  - 64-bit system → 8 bytes



41

Big-O: Test Your Might…

```
LinkedList list;

for(i = 0; i < list.Count; i++)
{
   total += list.Find(i);
}
```

O(n)

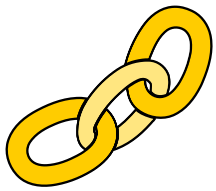
O(n)

$O(n^2)$

42

Iterators

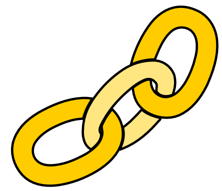
- To avoid accidental $O(n^2)$, major programming languages support *iterator objects*
- They store information about the current state (e.g. a node) when data is being are <u>sequentially</u> read



43

Iterators

- Iterators maintain O(n) for sequentially accessing all the list's elements
- This is the purpose of the For-Each Statement
- Notation varies greatly between languages (when they are supported)

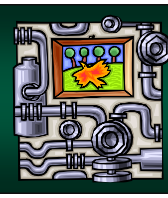


44

Array vs. Linked List

Operation	Array	Linked List
Find (to read or write)	O(1)	**O(n)**
Insert (arbitrary)	**O(n)**	**O(n)**
Add first/last	**O(n)**	O(1)
Remove first/last	**O(n)**	O(1)
Auxiliary storage	O(1)	**O(n)**

45



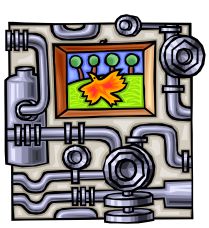
Data Abstraction

Abstraction is power

46

Abstract Data Types

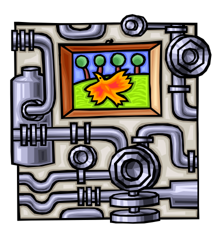
- *Data types* are used in practically all programming languages
- The core data types found in language is known as a *primitive data type*



47

Data Types Specify 2 Things

1. Set of possible values
2. *Operations* on the data
   - these are alternatively called *functions* or *methods*
   - data types often define the errors can occur during each operation



48

Integer Example

- *int* is a type (found in most languages)
- The 32-bit version can contain values from -2^{31} to $2^{31}-1$

```
int n;
```


49

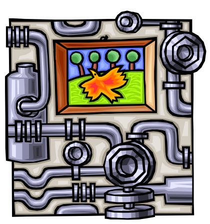
Integer Example

- Operations include: `+`, `*`, `–`, `/`, `%`, and many more (e.g. comparisons)

```
int n;
```


50

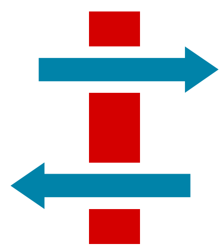
Abstract Data Types



- An *abstract data type (ADT)* hides how it is implemented from the *client* (programmer)
- The client only interacts with the defined operations
- This layer of abstraction separates implementation from behavior

51

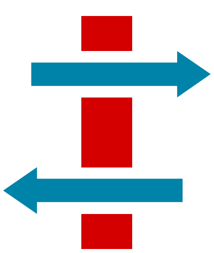
ADTs vs Data Structures



- An ADT is implementation independent
- Can, internally, use any data structure
  - array, linked list, etc…
  - depending how the ADT works, some are better than others

52

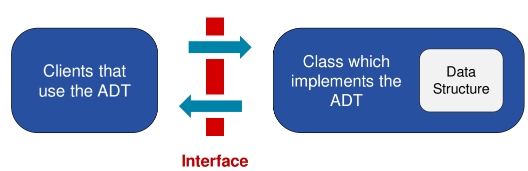
ADTs vs Data Structures



- ADT defines an *interface*
- It defines:
  - operations (public methods)
  - properties (public fields)

53

Data Structures



54

Example ADT: Cheese Trader

- Data stores orders of cheese
- The operations supported are
  - buy (cheese, count)
  - sell (cheese, count)
  - cancel (Order)
  - balance – current funds

55

Example ADT: Cheese Trader

- Error conditions:
  - nonexistent cheese
  - sell a cheese we don't have
  - count is not greater than 0

56

Cheese Trader API

public class CheeseTrader		
int	buy(string name, int count)	*Returns order #*
int	sell(string name, int count)	*Returns order #*
void	cancel(int order)	
double	balance()	

57

Bags

Just toss it in

58

Bags

- A bag is one of the most simplistic ADT that stores multiple objects
- It can only add items
- Order doesn't matter – nor is it expected to be maintained

59

Bags

- At is core, the class only requires one method: add()
- Other attributes, such as size, count, etc… and be inferred from return types (i.e. null)

60

Bag API

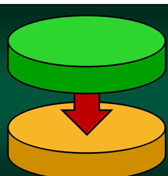
public class Bag		
	Bag()	*Create an empty bag*
void	add(Item item)	
bool	isEmpty()	
int	size()	

61

Bag Summary

Operation	Fixed Array	Resizable Array	Linked List
Add()	O(1)	**O(n)**	O(1)

62



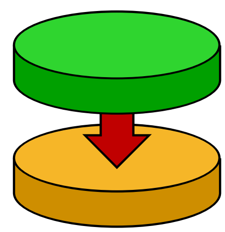
Stacks

Piles of… Data

63

Stack

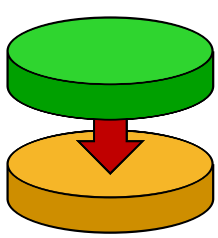
- The *Stack ADT* stores objects based on the concept of a stack of items – like a stack of dishes
- Data can only be added to or removed from the top of the stack



64

Stack

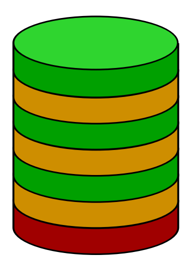
- This gives a first-in-last-out logic (aka FILO)
- Same concept is also called last-in-first-out (LIFO)



65

Stack Operation: Push

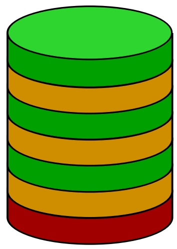
- A value is added to the stack
- It is placed on the top location
- Rest of the items are "covered"



66

Stack Operation: Pop

- Removes an item from the stack
- Last item added is removed
- 2nd item becomes the top



67

Stack API

public class Stack		
	Stack()	*Create empty stack*
void	push(Item item)	
Item	pop()	
bool	isEmpty()	
int	size()	

68

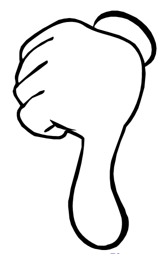
Stacks: Error Conditions

- The execution of an operation may sometimes cause an error condition, called an *exception*
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty

69

Resizing an Array-Based Stack

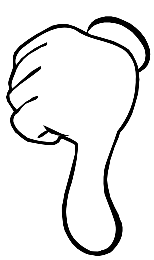
- For stacks, if a dynamically allocated array is used, each pop/push will require the entire array to be resized
- It will require O(n)
- So, a dynamic array is a poor choice



70

One Solution… Not a Great One

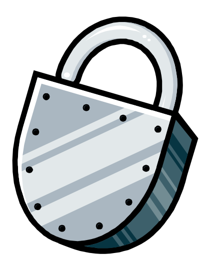
- The array *could* grow/shrink by a specific # of elements
- So, the array will resize only when a new "block" of elements is needed
- Like a fixed-capacity array, we need to keep an end index



71

Fixed-Capacity Stacks

- A fixed-capacity array can be used instead
- For a *fixed-capacity stack*, an array is an excellent choice – in specific situations…



72

Array-Based Fixed-Capacity Stack

- The stack would behave as normal until the capacity is reached
- In this case, one of two things will happen…

73

When the Stack is filled…

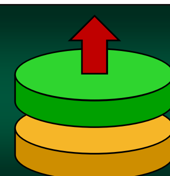
1. Stack throws an *Overflow Error*
2. Stack discards an object
   - the bottom of the stack is typically removed
   - this gives the space needed for the newly pushed object
   - e.g. the history feature of your web browser

74

Stack Summary

Operation	Fixed Array	Resizable Array	Linked List
Pop()	O(1)	**O(n)**	O(1)
Push()	O(1)	**O(n)**	O(1)
Top()	O(1)	O(1)	O(1)

75



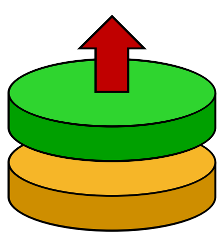
Queues

Conga-line of Data!

76

Queues

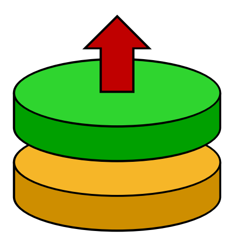
- *Queue ADT* stores list of arbitrary objects
- Based on the concept of a line – e.g. when you buy groceries
- Objects enter the back of the line, and must wait for prior items to leave before they do



77

Queues

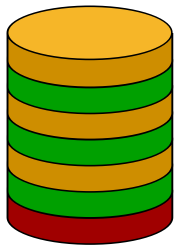
- In most parts of the World, they call a "line" a "queue"
- Main queue operations:
  - enqueue (object): place on item on the queue
  - dequeue: removes and returns the first inserted object



78

Queue Operation: Enqueue

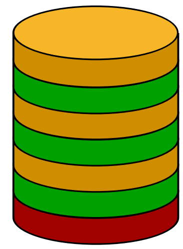
- When an object is "enqueued", it is put on to the **end** of the queue
- The items on the top of the queue are not covered



79

Queue Operation: Dequeue

- Dequeue removes the item from the front of the queue
- Second item becomes the new first item
- This gives a first-in-first-out logic (aka FIFO)



80

Auxiliary Queue Operations

- Queues also tend to have some operations defined
- These are not necessary, but they are useful
- Auxiliary operations:
  - peek: return the next object without removing it. This is also sometimes called "front"
  - size: returns the number of objects on the queue
  - isEmpty: indicates whether the queue contains no objects. This is an alterative to size()

81

Queue API

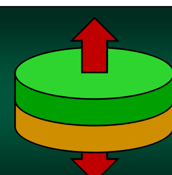
public class Queue		
	Queue()	*Create empty queue*
void	enqueue(Item item)	
Item	dequeue()	
bool	isEmpty()	
int	size()	

82

Queue Summary

Operation	Fixed Array	Resizable Array	Linked List
Enqueue()	O(1)	**O(n)**	O(1)
Dequeue()	O(1)	**O(n)**	O(1)
Peek()	O(1)	O(1)	O(1)

83



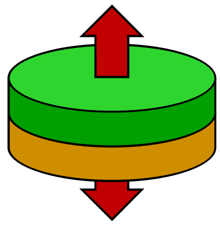
The Deque ADT

Time to shuffle the "deck"

84

Deque ADT

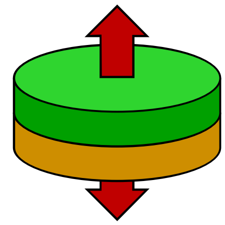
- There is a variant of the queue called a *deque* (pronounced "deck")
- The name is derived from **d**ouble-**e**nded **que**ue (sometimes it is shorted more to DQ)



85

Deque ADT

- As the name implies, it's a queue allows insertions and removals from both ends
- It is a merging of a stack and queue ADT and the operations are union of the two
- Be warned: name of each operation varies greatly between programming languages



86

Deque ADT

- addFront
  - place an object on the front of the deque
  - this is same as stack "push"
  - also called: offerFirst, pushFirst
- addBack
  - place an object on the end of the deque
  - this is the same as queue "enqueue"
  - also called: offerLast, pushLast

87

Deque ADT

- removeFront
  - remove an object from the front of the deque
  - same as: queue "dequeue" or stack "pop"
  - also called: pollFirst, popFront
- removeBack
  - this is unique – and not found in either a stack or queue ADT
  - also called pollLast, popBack

88

Deque API

public class Deque		
	Deque()	*Create empty deque*
void	addFront(Item item)	
void	addBack(Item item)	
Item	removeFront()	
Item	removeBack()	
bool	isEmpty()	
int	size()	

89

Deque Example

1. addFront('N')	N
2. addBack('E')	E
3. addFront('W')	W
4. addBack('D')	D
5. addFront('P')	P

90

Deque Advantages

- A deque can function as either a stack or queue
- "Add Front" operation can be used to "redo" or "undo" a queue removal – remove then put it back in line
- There are some scenarios where this logic is needed

91

Deque Disadvantages

- While, Stacks/Queues can be created with a single-linked-list, *a Deque requires a double-linked-list*
- …otherwise, removing items from the end would require O(n) – *even with a tail node*
- Also, the link overhead (memory requirements) is doubled

92

Deque Summary

Operation	Fixed Array	Resizable Array	Single Linked List	Double Linked List
addFront()	O(1)	**O(n)**	O(1)	O(1)
addBack()	O(1)	**O(n)**	O(1)	O(1)
removeFront()	O(1)	**O(n)**	O(1)	O(1)
removeBack()	O(1)	**O(n)**	**O(n)**	O(1)

93