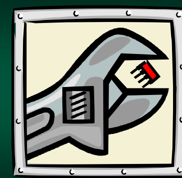
Abstract Data Types

Section 1.2, 1.3

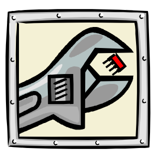


Data Structures

Return to CSC 15 and CSC 20

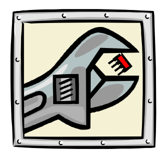
Abstract Data Types

- Arrays and linked-lists are both *data structures*
- They are methods of storing and organizing data



Abstract Data Types

- Depending on how data is accessed, arrays and linked lists have areas where they excel and falter
- We will cover more later in the semester – some which have incredible in features



Array Data Structure

Hidden math = easy code

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

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 read/written randomly in O(1)

Array Math Example: 32-bit int

- Let's assume the array starts at address 2000
- Each array element will take 4 bytes (for 32-bit integers)
- Array elements are stored continuous

2000	31302070
2004	74732074
2008	6F205261
2012	76656E63
2016	6C617721

Array Math Example: 32-bit int

- **array[0]** is at 2000
- **array[1]** is at 2004
- **array[2]** is at 2008
- **array[3]** is at 2012
- **array[4]** is at 2016
- etc…

2000	31302070
2004	74732074
2008	6F205261
2012	76656E63
2016	6C617721

Behind the Scenes…

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

```
start + (index × size of type)
```


Behind the Scenes…

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

```
start + (index × size of type)
```


Behind the Scenes…

- Java uses zero-indexing because C does
- … and C does so it can create efficient assembly!

```
start + (index × size of type)
```

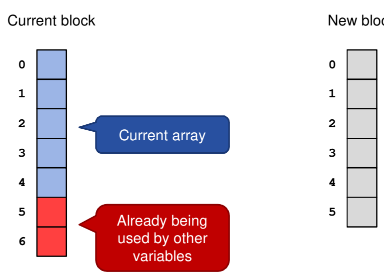
Auxiliary Storage in arrays

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

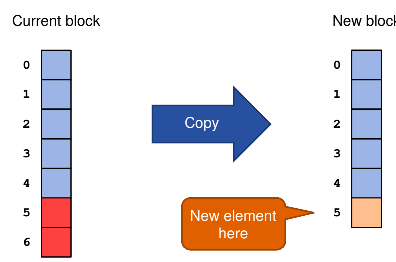
Resizing Arrays

- A *dynamically allocated array* is resized anytime an object is added or removed
- Because arrays require all elements to be stored continuously…
- …the old block of memory (old array) needs to be copied to a new one

New Block Must Be Created



Copy Values to New Block

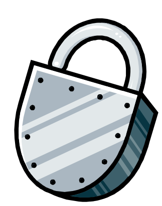


Resizing Arrays is O(n)

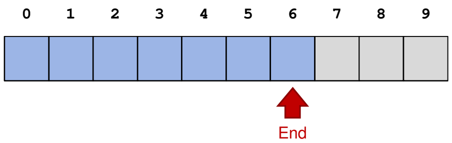
- While reading / writing elements takes only O(1)…
- … every time an array is resized, it will require O(n) time to copy the old array to the new one

Fixed-Sized Arrays

- Arrays can 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

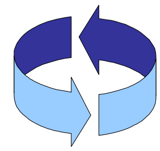


Fixed-Size Array



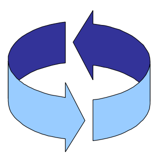
Fixed-Size Wrapping Around

- 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

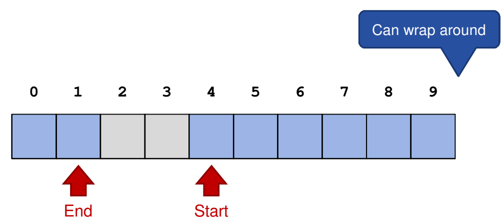


Fixed-Size Wrapping Around

- 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



Fixed-Size Array

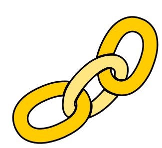


Linked List Data Structure

CSC 20's Revenge

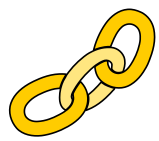
Linked List Data Structure

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

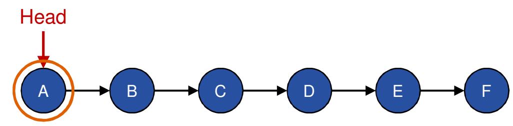


Linked List Data Structure

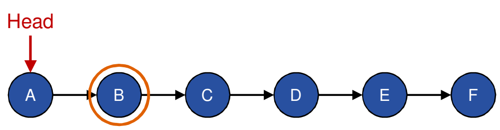
- Unlike arrays, where the element can be found using a calculation, linked-lists require the list to be traversed
- So, finding an item in a linked list requires O(n)



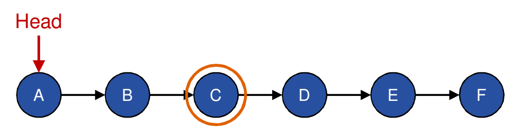
Single-Linked List – Find D



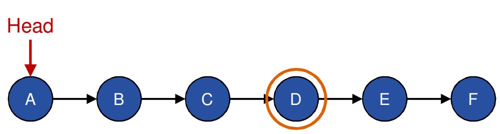
Single-Linked List – Find D



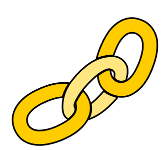
Single-Linked List – Find D



Single-Linked List – Find D

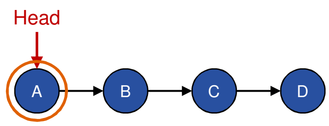


Head and Tail Nodes

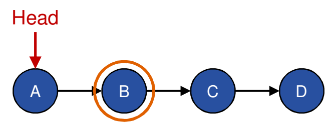


- Linked lists maintain a link to the head node
- Often, in a well-written linked lists, a link to the tail node is also maintained
- Why? It has a huge impact on time complexity

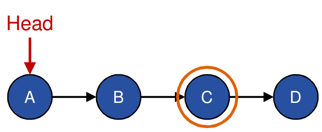
Append Value – No Tail Node



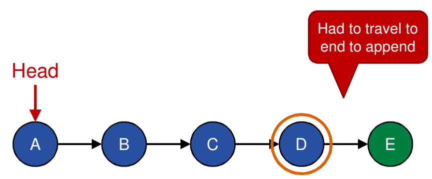
Append Value – No Tail Node



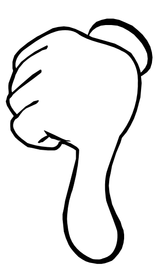
Append Value – No Tail Node



Append Value – No Tail Node

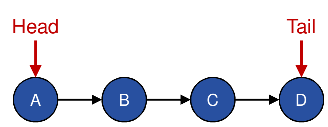


Head and Tail Nodes

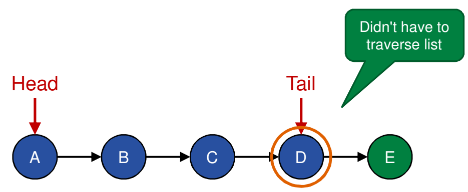


- Without a tail node, the entire list must be traversed to find the end
- This will require O(n)
- Adding a tail node, will decrease it to O(1)

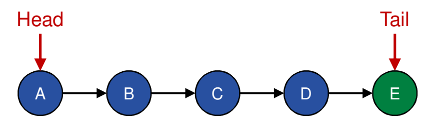
Append Value – With Tail Node



Append Value – With Tail Node

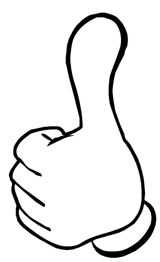


Append Value – With Tail Node



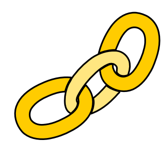
Use a Tail Node!

- 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



Auxiliary Storage in Linked Lists

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



Test Your Might…

```
LinkedList list;

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

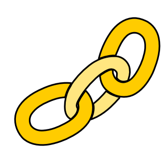
O(n)

O(n)

$O(n^2)$

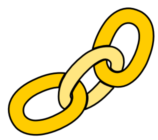
Iterators

- To avoid accidental $O(n^2)$, mainly programming languages support iterator objects
- They store information about the current state when all items in a data structure are sequentially read



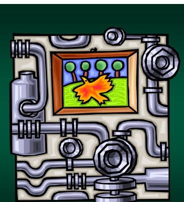
Iterators

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



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)**

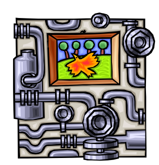


Data Abstraction

Section 1.2

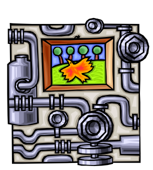
Abstract Data Types

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



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



Integer Example

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

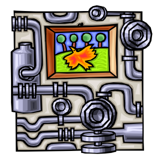
```
int n;
```

Integer Example

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

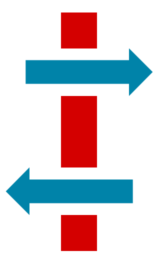
```
int n;
```


Abstract Data Types



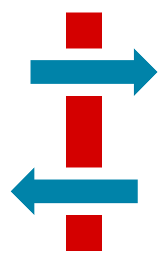
- An *abstract data type (ADT)* hides how it is implemented from the *client* (programmer)
- The client only interacts with the defined operations
- This layer of abstraction separates implementation from behavior

ADTs vs Data Structures



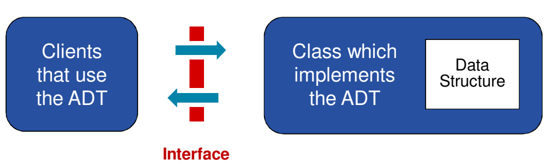
- An ADT is implementation <u>independent</u>
- Can internally use any data structure
  - array, linked list, etc…
  - depending how the ADT works, some are better than others

ADTs vs Data Structures



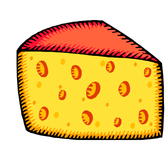
- ADT *defines application program interface (API)*
- ... or just *interface* (for short)
- It defines:
  - operations (methods)
  - properties (public fields)

Data Structures



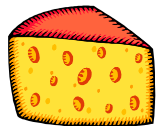
Example ADT: Cheese Trader

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

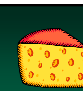


Example ADT: Cheese Trader

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

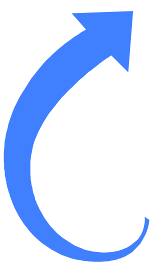


Cheese Trader API



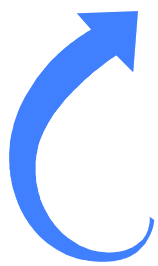
public class CheeseTrader		
int	buy(string name, int count)	*Returns order #*
int	sell(string name, int count)	*Returns order #*
void	cancel(int order)	
double	balance()	

Reference Types



- Most languages are based on largely based on building abstract data types called *reference types*
- They are links to nebulous *objects* – whose contents & implementation are unknown

Reference Types



- This is known as *object-oriented programming*
- … and is the basis of all modern programming languages

Bags

Just toss it in

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

Bags

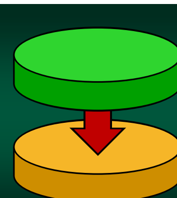
- At is core, the class only requires one method (add)
- Other attributes, such as size, count, etc… and be inferred from return types (i.e. null)

Bag API

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

Bag Summary

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

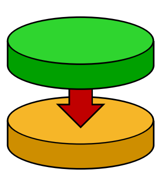


Stacks

Piles of… Data

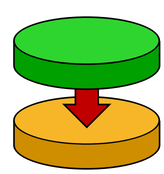
Stack

- A stack is an abstract data type that 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



Stack

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

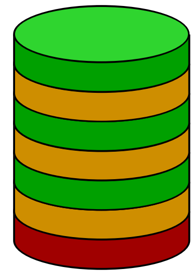


Examples of Stacks

- Page-visited "back button" history in a web browser
- Undo sequence in a text editor
- Deck of cards in Windows Solitaire

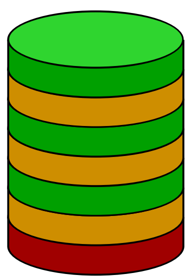
Stack Operation: Push

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

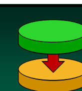


Stack Operation: Pop

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



Stack API



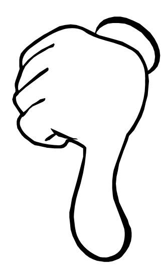
public class Stack		
	`Stack()`	*Create an empty stack*
`void`	`push(Item item)`	
`Item`	`pop()`	
`bool`	`isEmpty()`	
`int`	`size()`	

Stacks: Error Conditions

- Attempting the execution of an operation of ADT 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

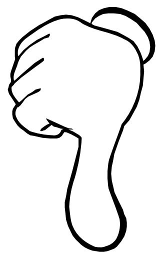
Resizing an Array-Based Stack

- Resizing the stack is expensive
- If a dynamically allocated array is used, each push/pop will require O(n)



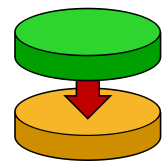
Some Stack-based solutions

- For a dynamic allocated array - grow/shrink by a specific # of elements each time in an attempt to minimize resizes
- … or used a fixed-capacity array – but your stack will have a fixed capacity



Fixed-Capacity Stacks

- It can be implemented by keep track of a capacity value (usually an int)
- The stack would behave as normal until the capacity is reached
- In this case, one of two things will happen…



Full Fixed-Capacity Stack…

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

Array-Based Fixed-Capacity Stack

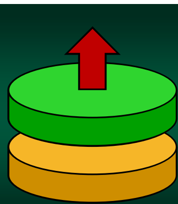
- While using an array for a normal stack (no fixed capacity) has a number of drawbacks
- … for fixed-capacity, an array is an excellent choice – in specific situations…

Array-Based Fixed-Capacity Stack

- When the capacity is reached, either an error occurs or the bottom of the stack is simply discarded
- … this is the case for the "undo" feature found in most applications

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)

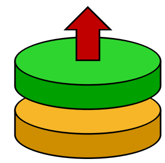


Queues

Conga-line of Data!

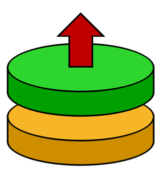
Queues

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



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



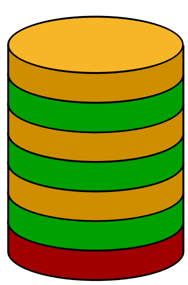
Examples of Queues

- Data being read from the keyboard or a file
- People waiting in line to go on Space Mountain
- Instructions on how to perform a task

82

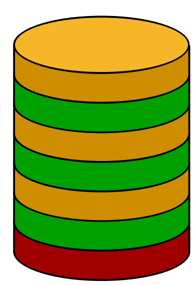
Queue Operation: Enqueue

- 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



Queue Operation: Dequeue

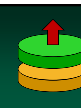
- Dequeue removes the item from the front of the queue
- Second item becomes the new first item
- This gives a first-in-first-out logic (aka FIFO)



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()

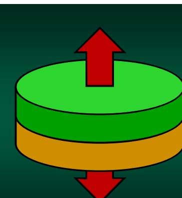
Queue API



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

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)

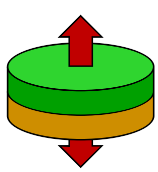


The Deque ADT

Time to shuffle the "deck"

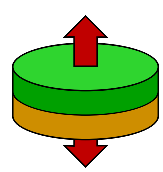
Deque ADT

- There is a variant of the queue called a deque (pronounced "deck")
- The name is derived from double-ended queue (sometimes it is shorted more to DQ)



Deque ADT

- As the name implies, its queue allows insertions and removals from both ends
- It is a merging of a stack and queue data ADT and the operations are union of the two
- Be warned: the names of the operations vary greatly between languages



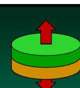
Deque ADT

- 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

Deque ADT

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

Deque API



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

Deque Example

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

N

E

W

D

P

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

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 of the list would require O(n) – even with an end pointer
- Also, the link overhead (memory requirements) is doubled

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)