ADTs and Collections

Abstraction

- *abstraction* refers to hiding implementation details and exposing only the essential features of a system
  - a core concept in computer science
- `if` statements and `while` and `for` loops provide *control abstraction*
  - we can focus on defining alternatives and what is repeated without knowing the details of how that is implemented in terms of jump instructions in machine code
- subroutines provide *procedural abstraction*
  - you can use a subroutine based on its header and contract without having to know the details of its body
- *abstract data types* (ADTs) provide *data abstraction*
  - you can define a data type in terms of its concept and operations without regards to how those operations are actually implemented

ADTs vs Data Structures

- an *abstract data type* is defined by its concept and operations
  - e.g. ordered list of strings, Stack, Queue
- *concrete data structures* are used to realize the implementation of an ADT
  - e.g. arrays, linked lists
  - generally have choices about how to implement an ADT, with different time/space tradeoffs
  - changing the data structure used to implement a given ADT does not change the correctness of code using that ADT, but may have a big influence on time/space requirements

	Answer	Respondents	Percentage
×	Linked lists and arrays are examples of ADTs.	8	22%
✓	An ADT allows different implementations, such as using arrays or linked lists.	13	36%
×	Changing the implementation of an ADT affects the program using it.	1	3%
✓	An ADT specifies operations on values without defining implementation details.	12	33%
×	An ADT is a technique for improving the efficiency of algorithms.	0	0%
×	An ADT defines a specific way of storing and managing data in memory.	2	6%

linked lists and arrays are concrete data structures

changing the implementation of an ADT changes only the private internals of a class – instance variables and method bodies

“a specific way of storing and managing day” refers to a specific implementation

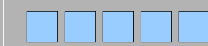
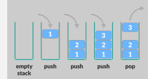
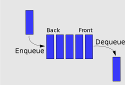
Fundamental ADTs – Collections

Many common ADTs store collections of values –

- *containers* provide storage and retrieval of elements independent of value
  - the ordering of elements depends on the structure of the container rather than the elements themselves
  - elements can be of any type
- *dictionaries* (or *maps*) and *sets* provide access to elements by value
  - lookup according to an element's key
  - elements can be of any type but the key type must support equality comparison (so you can tell if two keys are the same)
- *priority queues* provide access to elements in order by content
  - ordered by priority associated with elements
  - elements can be of any type but the priority type must be comparable (so there is an ordering)

ADTs – Common Containers

typical operations

List (also known as Vector, Sequence)	linear order, access by rank (index) or position (first/last, after/before) 	rank-based (array-like) operations • add(x), add(r,x) – add x at the end or with rank r • get(r) – get element with rank r • remove(r) – remove (and return) elt with rank r position-based (linked-list-like) operations • first, last() – get first/last position • before(p), after(p) – get position before/after p • addBefore(p,x), addAfter(p,x) – insert x after/before position p • get(p) – get element at position p • remove(p) – remove (and return) elt at pos p • replace(p,x) – replace elt at pos p with x
Stack 	linear order, access only at one end • LIFO – insert and remove at the same end	• **push(x)** – insert x at the top of the stack • **top()** – return top item (without removal) • **pop()** – remove and return the top item on the stack
Queue 	linear order, access only at both ends • FIFO – insert at one end, remove from the other	• **enqueue(x)** – insert x at the back of the queue • **peek()** – return front item (without removal) • **dequeue()** – remove and return the front item in the queue

Question	dequeue	enqueue	pop	push	add	append	delete	insert	remove
remove an element from a queue	✓ 13	0	0	0	0	0	0	0	0
insert an element into a queue	0	✓ 13	0	0	0	0	0	0	0
remove an element from a stack	0	0	✓ 11	0	0	0	0	0	2
insert an element into a stack	0	0	0	✓ 11	0	0	0	2	0

it's always possible that you may find implementations that use different names such as `add` or `remove`

Queue operation names are less standardized than Stack

	Answer	Respondents	Percentage
×	A stack allows insertion and removal from both ends, while a queue only allows insertion at one end and removal from the other.	1	2%
✓	A stack follows Last In, First Out (LIFO), while a queue follows First In, First Out (FIFO).	12	27%
×	A queue always requires more memory than a stack.	1	2%
×	A stack only allows insertion and removal at one end, while a queue allows insertion and removal at both ends.	2	5%

the statement about queues is true, but stacks only allow insertion and removal at one end

the memory required depends on the implementation, and both stacks and queues can be implemented efficiently using both arrays and linked lists

the statement about stacks is true, but queues only allow insertion and removal at opposite ends
(there is a variation of a queue that allows insertion and removal at both ends called a *deque*, for *double-ended queue*)

✓	A stack only allows insertion and removal at the same end, while a queue only allows insertion and removal at opposite ends.	11	25%
✕	A stack can be implemented efficiently using an array or a linked list, while a queue can only be implemented efficiently using a linked list.	6	14%
✓	If you insert a bunch of elements into a stack and then remove them, you'll get them in reverse order, whereas if you insert a bunch of elements into a queue and then remove then, you'll get them in the same order.	11	25%

the book notes that an efficient array implementation of a queue is "trickier" but it is possible

Applications of ADTs

The kind of access to elements imposed by different types of containers can be exploited to achieve algorithmic goals.

ADT	some applications of the ADT
List	general-purpose container round-robin scheduling, taking turns
Stack	match most recent thing, proper nesting, reversing program call stack – keeping track of subroutine calls evaluating postfix expressions (e.g. `2 15 12 - 17 * +`) depth-first search (DFS) – go deep before backing up
Queue	FIFO order minimizes waiting time round-robin scheduling, taking turns breadth-first search (BFS) – spread out in levels

Choosing Between Container ADTs

Use a queue when –
- you want things out in the same order you put them in

Use a stack when –
- you want things out in the reverse of the order you put them in
- you want to access the most recent thing added

Use a list when –
- stacks and queues don't serve your needs
- you want to iterate through things repeatedly
- you need to insert/remove/access at any position
  - stacks and queues don't allow direct access to anything but the top/front

Implementing (Collections) ADTs

- defining a type → write a class
- ADT operations → public methods
- need to decide on the instance variables
  - a concrete data structure (array, linked list, ...) to hold the elements
  - supporting things e.g. size for partially-full array
  - additional things to help with efficiency e.g. tail pointer for linked list
- also need to decide on how to use the instance variables
  - e.g. does the top of the stack go at the beginning or the end of the array? at the head or tail of the linked list?
    → consider running time

```
public class StackOfInts {

   /**
    * An object of type Node hold
    * that represents the stack.
    */
   private static class Node {
      int item;
      Node next;
   }

   private Node top;

   /**
    * Add N to the top of the stack.
    */
   public void push( int N ) {
      Node newTop;          // A Node to hold the new item.
      newTop = new Node();
      newTop.item = N;      // Store N in the new Node.
      newTop.next = top;    // The new Node points to the old top
      top = newTop;         // The new item is now on top.
   }

   /**
    * Remove the top item from the stack, and return it.
    * Throws an IllegalStateException if the stack is empty when
    * this method is called.
    */
   public int pop() {
      if ( top == null )
         throw new IllegalStateException("Can't pop from an emp
      int topItem = top.item;  // The item that is being popped
      top = top.next;          // The previous second item is no
      return topItem;
   }

   /**
    * Returns true if the stack is empty.  Returns false
    * if there are one or more items on the stack.
    */
   public boolean isEmpty() {
      return (top == null);
   }

} // end class StackOfInts
```

a private helper class – we need a Node class for the linked list, but it is only used internally to help with the implementation

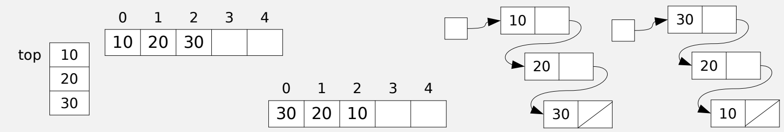
head pointer for the linked list – top is a more descriptive name than head as it reminds us that the head of the list corresponds to the top of the stack

```
import java.util.Arrays;  // For the Arrays.copyOf() method.

public class StackOfInts {  // (alternate version, using an array)

   private int[] items = new int[10];

   private int top = 0;  // The number

   /**
    * Add N to the top of the stack.
    */
   public void push( int N ) {
       if (top == items.length) {
             // The array is full, so make a new, larger array and
             // copy the current stack items into it.
          items = Arrays.copyOf( items, 2*items.length );
       }
       items[top] = N;  // Put N in next available spot.
       top++;           // Number of items goes up by one.
   }

   /**
    * Remove the top item from the stack, and return it.
    * Throws an IllegalStateException if the stack is empty when
    * this method is called.
    */
   public int pop() {
       if ( top == 0 )
          throw new IllegalStateException("Can't pop from an empty stack.");
       int topItem = items[top - 1];  // Top item in the stack.
       top--;    // Number of items on the stack goes down by one.
       return topItem;
   }

   /**
    * Returns true if the stack is empty.  Returns false
    * if there are one or more items on the stack.
    */
   public boolean isEmpty() {
      return (top == 0);
   }

} // end class StackOfInts
```

array and size for a partially-full array
size would be a better name since we are thinking of this as the number of elements

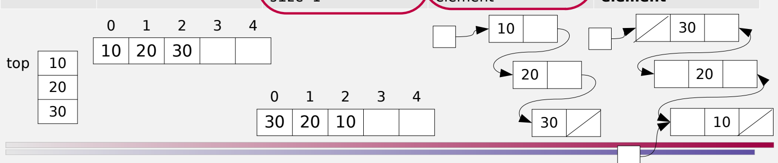
Implementing Stack

operation	array – top at beginning	array – top at end	linked list – top at head	linked list – top at tail
instance variables	• partially full dynamic array – array, size	• partially full dynamic array – array, size	• linked list – head • size	• linked list – head • size
size()	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size
isEmpty()	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0
push(elt)	Θ(n) – shift elements out of the way	O(n) – Θ(1) put element in slot size; Θ(n) if we have to grow	Θ(1) – insert at head	Θ(n) – Θ(n) to find the tail, then Θ(1) to add new node
pop()	Θ(n) – shift elements to fill gap	Θ(1) – top is in slot size-1, decrement size	Θ(1) – remove head	Θ(n) – Θ(n) to find node before the tail, then Θ(1) to remove the tail
top()	Θ(1) – top is in slot 0	Θ(1) – top is in slot size-1	Θ(1) – head's element	Θ(n) – to find the tail



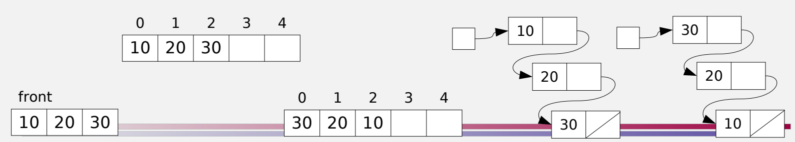
Implementing Stack

operation	array – top at beginning	array – top at end	linked list – top at head	linked list – top at tail
instance variables	• partially full dynamic array – array, size	• partially full dynamic array – array, size	• linked list – head • size	• **doubly** linked list – head, **tail** • size
size()	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size
isEmpty()	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0
push(elt)	Θ(n) – shift elements out of the way	O(n) – Θ(1) put element in slot size; Θ(n) if we have to grow	Θ(1) – insert at head	**Θ(1) – add new node, update tail**
pop()	Θ(n) – shift elements to fill gap	Θ(1) – top is in slot size-1, decrement size	Θ(1) – remove head	**Θ(1) – get node before the tail, then remove the tail**
top()	Θ(1) – top is in slot 0	Θ(1) – top is in slot size-1	Θ(1) – head's element	**Θ(1) – tail's element**



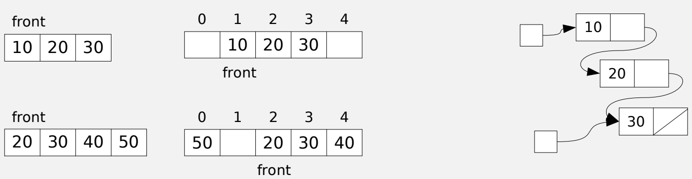
Implementing Queue

operation	array – front at beginning	array – front at end	linked list – front at head	linked list – front at tail
instance variables	• partially full dynamic array – array, size	• partially full dynamic array – array, size	• linked list – head • size	• linked list – head • size
size()	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size	Θ(1) – return size
isEmpty()	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0	Θ(1) – return size == 0
enqueue(elt)	O(n) – Θ(1) put element in slot size; Θ(n) if we have to grow	Θ(n) – shift elements to make room; also Θ(n) if we have to grow	Θ(n) – find tail, add new node	Θ(1) – insert at head
dequeue()	Θ(n) – shift elements to fill gap	Θ(1) – front is in slot size-1, decrement size	Θ(1) – remove head	Θ(n) – Θ(n) to find node before the tail, then Θ(1) to remove the tail
front()	Θ(1) – front is in slot 0	Θ(1) – front is in slot size-1	Θ(1) – head's element	Θ(n) – to find the tail



Implementing Queue

operation	array – front at beginning (circular array)	linked list – front at head
instance variables	• partially full dynamic array – array, size • **index of first element – front**	• linked list – head, **tail** • size
size()	Θ(1) – return `size`	Θ(1) – return `size`
isEmpty()	Θ(1) – return `size == 0`	Θ(1) – return `size == 0`
enqueue(elt)	O(n) – Θ(1) put element in slot `size`; Θ(n) if we have to grow	**Θ(1) – add new node, update tail**
dequeue()	**Θ(1) – increment `front`**	Θ(1) – remove head
front()	Θ(1) – front is in slot `front`	Θ(1) – head's element



Array-Based Implementations

Observations –

doesn't involve a loop – same number of steps regardless of the size of the array

- things arrays are good for – Θ(1)
  - accessing a particular slot (random access)
  - inserting or removing elements at the end
  - inserting or removing elements in the middle when the order doesn't need to be preserved (can swap with the last thing)

involve a loop – number of steps depends on the size of the array

- things arrays are less good for – Θ(n)
  - inserting or removing elements in the middle when the order needs to be preserved
  - varying-size collections when you have to grow or shrink
    - doubling the size mitigates the expense of copy over a series of insertions

Linked List-Based Implementations

Observations –

doesn't involve a loop – same number of steps regardless of the length of the list

- things linked lists are good for – Θ(1)
  - accessing the head
  - inserting or removing elements at the head
    - inserting at the tail with a tail pointer
    - removing the tail if doubly-linked
  - inserting or removing after a node
    - inserting or removing before a node if doubly-linked

involve a loop – number of steps depends on the length of the list

- things linked lists are less good for – Θ(n)
  - accessing a particular position (no random access)
  - inserting or removing at a particular position
  - inserting or removing before a node (if singly-linked)

Arrays vs. Linked Lists

Advantages of linked lists –

- no need to grow when full because nodes are allocated/deallocated as needed
- no empty slots
  - though arrays still have an advantage in space usage as long as they are at least half full
- insert/remove don't require shifting
  - much faster than array if insertion point is known (otherwise requires time to find node)

Advantages of arrays –

- random access
  - linked lists support sequential access only – must scan forward from head
- simpler if the number of elements doesn't change