ADT Stacks and Queues

October 12, 2020 4:52 PM

* An ADT is the abstraction of a Data structure.

-> We use ADT to specify data stored, operations on the data and error conditions associated with the operations.

-> It specifies the operations, but not how they are done, because its an interface so methods have empty bodies.

-> Its used to model the concrete Data structure, by a class, that implements all the methods from the interface.

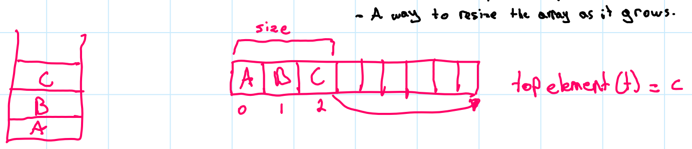
* STACK

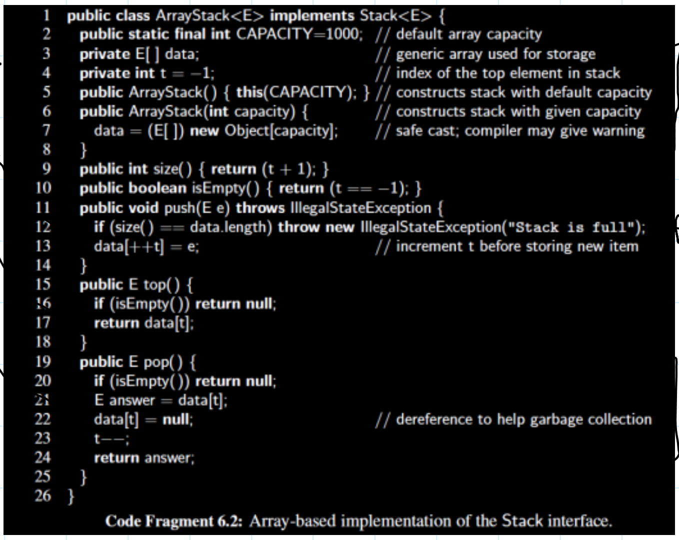
. A Last in First out Structure (LIFO)

. Its a form of a list, linear data structure.

. Should include these methods (operations). push(x), pop(), $\underbrace{\text{top(), size(), isEmpty()}}_{\text{optional}}$.

. If implemented using an Array : - Add elements left to right.
- Variable to keep track of the index of element at top
- A way to resize the array as it grows.





```java
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;  // default array capacity
  private E[ ] data;                      // generic array used for storage
  private int t = −1;                     // index of the top element in stack
  public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
  public ArrayStack(int capacity) {       // constructs stack with given capacity
    data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
  }
  public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == −1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                        // increment t before storing new item
  }
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
  }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;                       // dereference to help garbage collection
    t−−;
    return answer;
  }
}
```

Code Fragment 6.2: Array-based implementation of the Stack interface.

Keep track of top index

Fixed capacity

push if not at max

pop if not empty

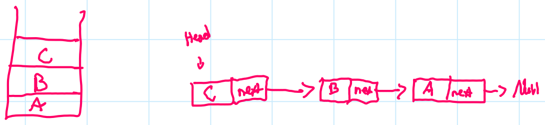
. Each operation runs in O(1), space used is O(n). Pros: Less memory cuz not storing pointers. Cons: Preset size

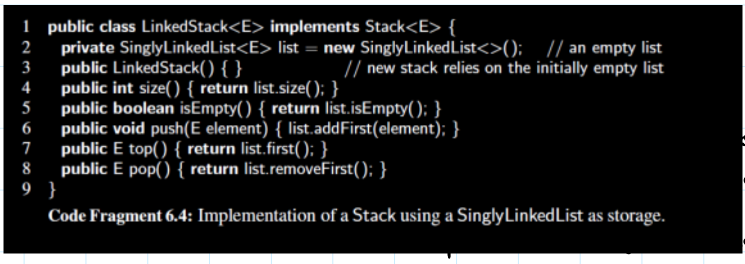
. When the array storing the stack elements reaches max capacity, a push() will throw a FullStackException
↳ Size needs to be predefined and can't change. This exception is only for array implementation. Not intrinsic to Stack ADT

. With Linked list, we don't need to define a maximum size. push and pop add/remove at head so its O(1)

Java Stack interface corresponds to our Stack ADT, which is different than the built in Java.util.Stack class.

It includes EmptyStackException thats thrown if we try pop() or top() on an empty Stack.





```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                  // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

Code Fragment 6.4: Implementation of a Stack using a SinglyLinkedList as storage.

ad, because its in O(1)

would be O(n)

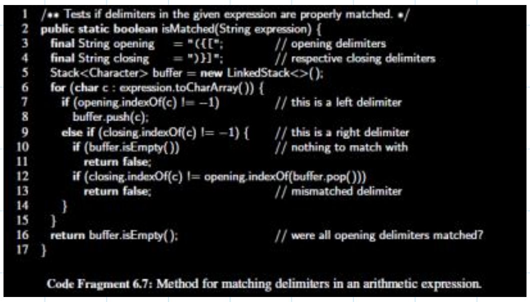
ame complexity

- Use cases of Stack:

. The java call stack - The method call stack in JVM Keeps track of the chain of active methods with a stack.
→ when a method is called JVM pushes a frame to the stack. The frame has the local variables, return value and a program counter, to keep track of statement being executed
→ when a method ends/returns, the frame is popped and control is passed to the method at top of stack. This allows us to use recursion

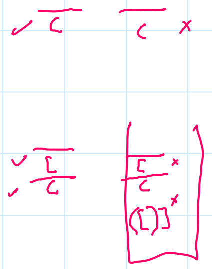
. Depth first search - Uses a stack to keep track of nodes to visit next.

String parsing / brackets matching - We use stacks to write code that tells us if opening and closing tags are properly nested/matched.



```java
/** Tests if delimiters in the given expression are properly matched. */
public static boolean isMatched(String expression) {
  final String opening    = "({[";          // opening delimiters
  final String closing    = ")}]";          // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>();
  for (char c : expression.toCharArray()) {
    if (opening.indexOf(c) != −1)           // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) {    // this is a right delimiter
      if (buffer.isEmpty())                 // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
        return false;                       // mismatched delimiter
    }
  }
  return buffer.isEmpty();                  // were all opening delimiters matched?
}
```

Code Fragment 6.7: Method for matching delimiters in an arithmetic expression.



* QUEUE

. A First in First out (FIFO) Data Structure.

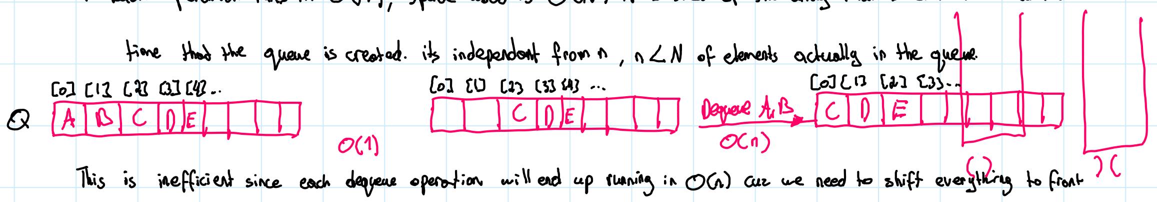
. Just like Stack, its also a Linear Data Structure.

. Insertion happens at the end (tail of the list), removal at the front (head)

. Should include these methods (operations). enqueue(object), dequeue(object), object front(), int size(), bool isEmpty()

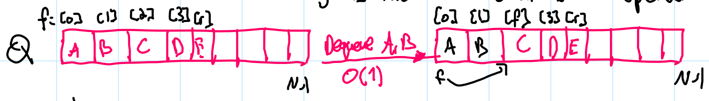
. If implemented using Arrays: - Two index variables to keep track of front and rear (f=r=0) empty queue.
- Can use a circular array from Q[0] to Q[n-1] then wrap around back to Q[0], this holds N elements. Logic for this explained below.

. Each operation runs in O(1), space used is O(N) N is size of the array that is determined at the time that the queue is created. its independent from n, n < N of elements actually in the queue.

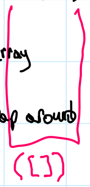


This is inefficient since each dequeue operation will end up running in O(n) cuz we need to shift everything to front

So instead of starting at Q[0] we use 2 variables for front/rear. Q[f] = Q[r] = 0. r is left empty and incremented after something is inserted. This allows our operations to be in O(1), but theres a problem.



. Now with an array of size N we end up storing less than N elements. Because we're letting the array's begining drift away, so if we repeatedly (N times) enqueue or dequeue an element we will have f = r = N. and get an ArrayOutOfBoundsException. while the array actually has many empty slots. So instead of all this, we use a circular array, allowing us to wrap around

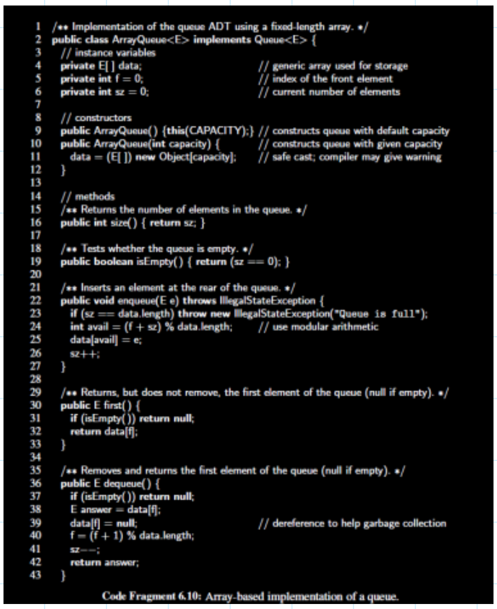


. When the array storing the stack elements reaches max capacity, enqueue(object) throws FullQueueException.

actually has many empty slots. So instead of all this, we use a circular array, allowing us to wrap around

- When the array storing the stack elements reaches max capacity, enqueue(object) throws FullQueueException.
  ↳ This is array implementation dependent. Not intrinsic to the Queue ADT
- dequeue() or front(), on empty queue will throw a EmptyQueueException.
- We use the modulo operation to determine the index in circular array to "advance" it.

$$front = (front + 1) \% N$$

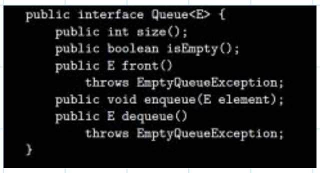


```
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  private E[ ] data;                         // generic array used for storage
  private int f = 0;                         // index of the front element
  private int sz = 0;                        // current number of elements

  // constructors
  public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
  public ArrayQueue(int capacity) {          // constructs queue with given capacity
    data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
  }

  // methods
  /** Returns the number of elements in the queue. */
  public int size() { return sz; }

  /** Tests whether the queue is empty. */
  public boolean isEmpty() { return (sz == 0); }

  /** Inserts an element at the rear of the queue. */
  public void enqueue(E e) throws IllegalStateException {
    if (sz == data.length) throw new IllegalStateException("Queue is full");
    int avail = (f + sz) % data.length;      // use modular arithmetic
    data[avail] = e;
    sz++;
  }

  /** Returns, but does not remove, the first element of the queue (null if empty). */
  public E first() {
    if (isEmpty()) return null;
    return data[f];
  }

  /** Removes and returns the first element of the queue (null if empty). */
  public E dequeue() {
    if (isEmpty()) return null;
    E answer = data[f];
    data[f] = null;                          // dereference to help garbage collection
    f = (f + 1) % data.length;
    sz--;
    return answer;
  }
}
```

Code Fragment 6.10: Array-based implementation of a queue.

So if we have an array of length 10 and our front index is 7, we calculate the new index after enquing by doing $(7+1)\%10 = 8$, for $8 \Rightarrow (8+10)\%10 = 9$ but then for 9, $(9+10)\%10 = 0$, so we go back to the beginning.

* The Java Queue interface corresponding to our Queue ADT:
  ↳ Unlike Stack, there is no Queue class in Java



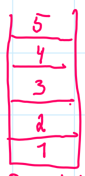
```
public interface Queue<E> {
    public int size();
    public boolean isEmpty();
    public E front()
        throws EmptyQueueException;
    public void enqueue(E element);
    public E dequeue()
        throws EmptyQueueException;
}
```

- If implemented using a LinkedList instead of Array:
  - No capacity issue, list will always be the right size (no empty head/tails)
  - To remove, we only take from head, no need for the whole mod stuff.
  - Bigger Memory usage for the structure, storing pointers, every node is an Object.

- Use cases of Queue:
  - Printers: Printers use queues to manage the print requests. requests get printed in the order they're submitted.
  - CPU scheduling: Processes wait in the CPU scheduler's queue for their turn to run.
  - Breadth-first search: Uses a queue to keep track of nodes to visit next.

Fun Bonus Exercise: How can we implement a Queue (FIFO) using a Stack (LIFO)

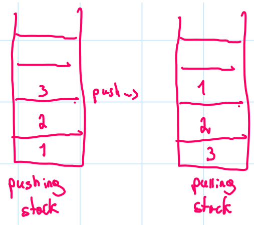
[1,2,3,4,5]



Queue but actually stack.

now if we dequeue it should remove/return 1 not 5. But since its a stack, the only way to do that is popping everything, empty the stack in O(n), then we add everything back to the stack. Again O(n). Except thats useless because we would be ruining the order. So we can try using 2 stacks.

[1,2,3,4,5]

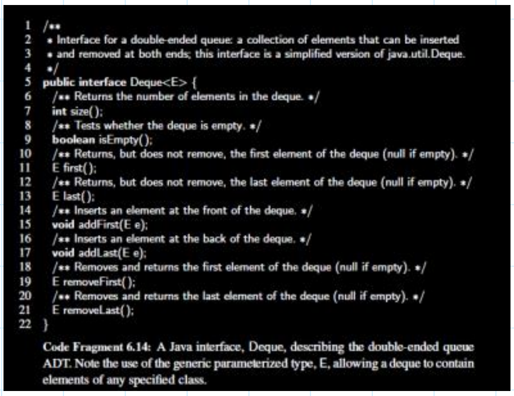


if we enqueue 1, 2, 3. They go to push stack. if we dequeue we look at pulling stack, if its empty we push everything to it from the pushing stack. so 3,2,1. Now the dequeue will return 1 cuz its at top of stack Enqueue 4, 5. dequeue 2, then 3, dequeue again, empty push 5, 4, returns 4. ✓

Now its running in $O(1)$ instead of $O(n)$ (except when its getting an empty stack and moving ever thing over but this doesn't happen often enough so we can say that amortized running time is $O(1)$)

✱ Double-ended Queues. (DQ / DEque)

- We saw Stack –> LIFo
  Queue –> FIFo
- Now Deque –> LIFo / FIFO, its richer than Stack and queue ADT.
- Should include these methods: addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast(), size(), isEmpty()
-



```java
/**
 * Interface for a double-ended queue: a collection of elements that can be inserted
 * and removed at both ends; this interface is a simplified version of java.util.Deque.
 */
public interface Deque<E> {
  /** Returns the number of elements in the deque. */
  int size();
  /** Tests whether the deque is empty. */
  boolean isEmpty();
  /** Returns, but does not remove, the first element of the deque (null if empty). */
  E first();
  /** Returns, but does not remove, the last element of the deque (null if empty). */
  E last();
  /** Inserts an element at the front of the deque. */
  void addFirst(E e);
  /** Inserts an element at the back of the deque. */
  void addLast(E e);
  /** Removes and returns the first element of the deque (null if empty). */
  E removeFirst();
  /** Removes and returns the last element of the deque (null if empty). */
  E removeLast();
}
```

Code Fragment 6.14: A Java interface, Deque, describing the double-ended queue ADT. Note the use of the generic parameterized type, E, allowing a deque to contain elements of any specified class.