```
    if (allStacksAreFull()) {
      throw new FullStackException();
    }

    /* If this stack is full, expand it. */
    StackInfo stack = info[stackNum];
    if (stack.isFull()) {
      expand(stackNum);
    }

    /* Find the index of the top element in the array + 1, and increment the
     * stack pointer */
    stack.size++;
    values[stack.lastElementIndex()] = value;
  }

  /* Remove value from stack. */
  public int pop(int stackNum) throws Exception {
    StackInfo stack = info[stackNum];
    if (stack.isEmpty()) {
      throw new EmptyStackException();
    }

    /* Remove last element. */
    int value = values[stack.lastElementIndex()];
    values[stack.lastElementIndex()] = 0; // Clear item
    stack.size--; // Shrink size
    return value;
  }

  /* Get top element of stack.*/
  public int peek(int stackNum) {
    StackInfo stack = info[stackNum];
    return values[stack.lastElementIndex()];
  }
  /* Shift items in stack over by one element. If we have available capacity, then
   * we'll end up shrinking the stack by one element. If we don't have available
   * capacity, then we'll need to shift the next stack over too. */
  private void shift(int stackNum) {
    System.out.println("/// Shifting " + stackNum);
    StackInfo stack = info[stackNum];

    /* If this stack is at its full capacity, then you need to move the next
     * stack over by one element. This stack can now claim the freed index. */
    if (stack.size >= stack.capacity) {
      int nextStack = (stackNum + 1) % info.length;
      shift(nextStack);
      stack.capacity++; // claim index that next stack lost
    }

    /* Shift all elements in stack over by one. */
    int index = stack.lastCapacityIndex();
    while (stack.isWithinStackCapacity(index)) {
      values[index] = values[previousIndex(index)];
      index = previousIndex(index);
    }
```

```

    /* Adjust stack data. */
    values[stack.start] = 0; // Clear item
    stack.start = nextIndex(stack.start); // move start
    stack.capacity--; // Shrink capacity
  }

  /* Expand stack by shifting over other stacks */
  private void expand(int stackNum) {
    shift((stackNum + 1) % info.length);
    info[stackNum].capacity++;
  }

  /* Returns the number of items actually present in stack. */
  public int numberOfElements() {
    int size = 0;
    for (StackInfo sd : info) {
      size += sd.size;
    }
    return size;
  }

  /* Returns true is all the stacks are full. */
  public boolean allStacksAreFull() {
    return numberOfElements() == values.length;
  }

  /* Adjust index to be within the range of 0 -> length - 1. */
  private int adjustIndex(int index) {
    /* Java's mod operator can return neg values. For example, (-11 % 5) will
     * return -1, not 4. We actually want the value to be 4 (since we're wrapping
     * around the index). */
    int max = values.length;
    return ((index % max) + max) % max;
  }

  /* Get index after this index, adjusted for wrap around. */
  private int nextIndex(int index) {
    return adjustIndex(index + 1);
  }

  /* Get index before this index, adjusted for wrap around. */
  private int previousIndex(int index) {
    return adjustIndex(index - 1);
  }
}
```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with StackInfo, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

3.2 **Stack Min:** How would you design a stack which, in addition to `push` and `pop`, has a function `min` which returns the minimum element? `Push`, `pop` and `min` should all operate in $O(1)$ time.

pg 98

SOLUTION

The thing with minimums is that they don't change very often. They only change when a smaller element is added.

One solution is to have just a single `int` value, `minValue`, that's a member of the `Stack` class. When `minValue` is popped from the stack, we search through the stack to find the new minimum. Unfortunately, this would break the constraint that `push` and `pop` operate in $O(1)$ time.

To further understand this question, let's walk through it with a short example:

```
push(5); // stack is {5}, min is 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5.
```

Observe how once the stack goes back to a prior state (`{6, 5}`), the minimum also goes back to its prior state (5). This leads us to our second solution.

If we kept track of the minimum at each state, we would be able to easily know the minimum. We can do this by having each node record what the minimum beneath itself is. Then, to find the `min`, you just look at what the top element thinks is the min.

When you push an element onto the stack, the element is given the current minimum. It sets its "`local min`" to be the `min`.

```
public class StackWithMin extends Stack<NodeWithMin> {
   public void push(int value) {
      int newMin = Math.min(value, min());
      super.push(new NodeWithMin(value, newMin));
   }

   public int min() {
      if (this.isEmpty()) {
         return Integer.MAX_VALUE; // Error value
      } else {
         return peek().min;
      }
   }
}

class NodeWithMin {
   public int value;
   public int min;
   public NodeWithMin(int v, int min){
      value = v;
      this.min = min;
   }
}
```

There's just one issue with this: if we have a large stack, we waste a lot of space by keeping track of the `min` for every single element. Can we do better?

We can (maybe) do a bit better than this by using an additional stack which keeps track of the `mins`.

```
public class StackWithMin2 extends Stack<Integer> {
  Stack<Integer> s2;
  public StackWithMin2() {
    s2 = new Stack<Integer>();
  }

  public void push(int value){
    if (value <= min()) {
      s2.push(value);
    }
    super.push(value);
  }

  public Integer pop() {
    int value = super.pop();
    if (value == min()) {
      s2.pop();
    }
    return value;
  }

  public int min() {
    if (s2.isEmpty()) {
      return Integer.MAX_VALUE;
    } else {
      return s2.peek();
    }
  }
}
```

Why might this be more space efficient? Suppose we had a very large stack and the first element inserted happened to be the minimum. In the first solution, we would be keeping n integers, where n is the size of the stack. In the second solution though, we store just a few pieces of data: a second stack with one element and the members within this stack.

3.3 **Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

pg 99

SOLUTION

In this problem, we've been told what our data structure should look like:

```
class SetOfStacks {
  ArrayList<Stack> stacks = new ArrayList<Stack>();
  public void push(int v) { ... }
```

```
    public int pop() { ... }
}
```

We know that `push()` should behave identically to a single stack, which means that we need `push()` to call `push()` on the last stack in the array of stacks. We have to be a bit careful here though: if the last stack is at capacity, we need to create a new stack. Our code should look something like this:

```
void push(int v) {
   Stack last = getLastStack();
   if (last != null && !last.isFull()) { // add to last stack
      last.push(v);
   } else { // must create new stack
      Stack stack = new Stack(capacity);
      stack.push(v);
      stacks.add(stack);
   }
}
```

What should pop() do? It should behave similarly to `push()` in that it should operate on the last stack. If the last stack is empty (after popping), then we should remove the stack from the list of stacks.

```
int pop() {
   Stack last = getLastStack();
   if (last == null) throw new EmptyStackException();
   int v = last.pop();
   if (last.size == 0) stacks.remove(stacks.size() - 1);
   return v;
}
```

Follow Up: Implement popAt(int index)

This is a bit trickier to implement, but we can imagine a "rollover" system. If we pop an element from stack 1, we need to remove the *bottom* of stack 2 and push it onto stack 1. We then need to rollover from stack 3 to stack 2, stack 4 to stack 3, etc.

You could make an argument that, rather than "rolling over," we should be okay with some stacks not being at full capacity. This would improve the time complexity (by a fair amount, with a large number of elements), but it might get us into tricky situations later on if someone assumes that all stacks (other than the last) operate at full capacity. There's no "right answer" here; you should discuss this trade-off with your interviewer.

```
public class SetOfStacks {
   ArrayList<Stack> stacks = new ArrayList<Stack>();
   public int capacity;
   public SetOfStacks(int capacity) {
      this.capacity = capacity;
   }

   public Stack getLastStack() {
      if (stacks.size() == 0) return null;
      return stacks.get(stacks.size() - 1);
   }

   public void push(int v) { /* see earlier code */ }
   public int pop() { /* see earlier code */ }
   public boolean isEmpty() {
      Stack last = getLastStack();
      return last == null || last.isEmpty();
   }
```

```

  public int popAt(int index) {
    return leftShift(index, true);
  }

  public int leftShift(int index, boolean removeTop) {
    Stack stack = stacks.get(index);
    int removed_item;
    if (removeTop) removed_item = stack.pop();
    else removed_item = stack.removeBottom();
    if (stack.isEmpty()) {
      stacks.remove(index);
    } else if (stacks.size() > index + 1) {
      int v = leftShift(index + 1, false);
      stack.push(v);
    }
    return removed_item;
  }
}

public class Stack {
  private int capacity;
  public Node top, bottom;
  public int size = 0;

  public Stack(int capacity) { this.capacity = capacity; }
  public boolean isFull() { return capacity == size; }

  public void join(Node above, Node below) {
    if (below ! = null) below.above = above;
    if (above != null) above.below = below;
  }

  public boolean push(int v) {
    if (size >= capacity) return false;
    size++;
    Node n = new Node(v);
    if (size == 1) bottom = n;
    join(n, top);
    top = n;
    return true;
  }

  public int pop() {
    Node t = top;
    top = top.below;
    size--;
    return t.value;
  }

  public boolean isEmpty() {
    return size = =0;
  }

  public int removeBottom() {
    Node b = bottom;
```

```
      bottom = bottom.above;
      if (bottom != null) bottom.below = null;
      size--;
      return b.value;
   }
}
```

This problem is not conceptually that tough, but it requires a lot of code to implement it fully. Your interviewer would not ask you to implement the entire code.

A good strategy on problems like this is to separate code into other methods, like a `leftShift` method that popAt can call. This will make your code cleaner and give you the opportunity to lay down the skeleton of the code before dealing with some of the details.

3.4 Queue via Stacks: Implement a `MyQueue` class which implements a queue using two stacks.

pg 99

SOLUTION

Since the major difference between a queue and a stack is the order (first-in first-out vs. last-in first-out), we know that we need to modify `peek()` and `pop()` to go in reverse order. We can use our second stack to reverse the order of the elements (by popping `s1` and pushing the elements on to `s2`). In such an implementation, on each `peek()` and `pop()` operation, we would pop everything from `s1` onto `s2`, perform the `peek` / `pop` operation, and then `push` everything back.

This will work, but if two `pop` / `peeks` are performed back-to-back, we're needlessly moving elements. We can implement a "lazy" approach where we let the elements sit in `s2` until we absolutely must reverse the elements.

In this approach, `stackNewest` has the newest elements on top and `stackOldest` has the oldest elements on top. When we dequeue an element, we want to remove the oldest element first, and so we dequeue from `stackOldest`. If `stackOldest` is empty, then we want to transfer all elements from `stackNewest` into this stack in reverse order. To insert an element, we push onto `stackNewest`, since it has the newest elements on top.

The code below implements this algorithm.

```
public class MyQueue<T> {
   Stack<T> stackNewest, stackOldest;

   public MyQueue() {
      stackNewest = new Stack<T>();
      stackOldest = new Stack<T>();
   }

   public int size() {
      return stackNewest.size() + stackOldest.size();
   }

   public void add(T value) {
      /* Push onto stackNewest, which always has the newest elements on top */
      stackNewest.push(value);
   }

   /* Move elements from stackNewest into stackOldest. This is usually done so that
    * we can do operations on stackOldest. */
```

```
    private void shiftStacks() {
      if (stackOldest.isEmpty()) {
        while (!stackNewest.isEmpty()) {
          stackOldest.push(stackNewest.pop());
        }
      }
    }

    public T peek() {
      shiftStacks(); // Ensure stackOldest has the current elements
      return stackOldest.peek(); // retrieve the oldest item.
    }

    public T remove() {
      shiftStacks(); // Ensure stackOldest has the current elements
      return stackOldest.pop(); // pop the oldest item.
    }
}
```

During your actual interview, you may find that you forget the exact API calls. Don't stress too much if that happens to you. Most interviewers are okay with your asking for them to refresh your memory on little details. They're much more concerned with your big picture understanding.

3.5 **Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

pg 99

SOLUTION

One approach is to implement a rudimentary sorting algorithm. We search through the entire stack to find the minimum element and then push that onto a new stack. Then, we find the new minimum element and push that. This will actually require a total of three stacks: `s1` is the original stack, `s2` is the final sorted stack, and `s3` acts as a buffer during our searching of `s1`. To search `s1` for each minimum, we need to pop elements from `s1` and push them onto the buffer, `s3`.

Unfortunately, this requires two additional stacks, and we can only use one. Can we do better? Yes.

Rather than searching for the minimum repeatedly, we can sort `s1` by inserting each element from `s1` in order into `s2`. How would this work?

Imagine we have the following stacks, where `s2` is "sorted" and `s1` is not:

s1	s2
	12
5	8
10	3
7	1

When we pop 5 from `s1`, we need to find the right place in `s2` to insert this number. In this case, the correct place is on `s2` just above 3. How do we get it there? We can do this by popping 5 from `s1` and holding it in a temporary variable. Then, we move 12 and 8 over to `s1` (by popping them from `s2` and pushing them onto `s1`) and then push 5 onto `s2`.

Step 1

s1	s2
	12
	8
10	3
7	1

tmp = 5

->

Step 2

s1	s2
8	
12	
10	3
7	1

tmp = 5

->

Step 3

s1	s2
8	
12	5
10	3
7	1

tmp = --

Note that 8 and 12 are still in `s1`—and that's okay! We just repeat the same steps for those two numbers as we did for 5, each time popping off the top of `s1` and putting it into the "right place" on `s2`. (Of course, since 8 and 12 were moved from `s2` to `s1` precisely *because* they were larger than 5, the "right place" for these elements will be right on top of 5. We won't need to muck around with `s2`'s other elements, and the inside of the below `while` loop will not be run when `tmp` is 8 or 12.)

```
void sort(Stack<Integer> s) {
  Stack<Integer> r = new Stack<Integer>();
  while(!s.isEmpty()) {
    /* Insert each element in s in sorted order into r. */
    int tmp = s.pop();
    while(!r.isEmpty() && r.peek() > tmp) {
      s.push(r.pop());
    }
    r.push(tmp);
  }

  /* Copy the elements from r back into s. */
  while (!r.isEmpty()) {
    s.push(r.pop());
  }
}
```

This algorithm is $O(N^2)$ time and $O(N)$ space.

If we were allowed to use unlimited stacks, we could implement a modified quicksort or mergesort.

With the mergesort solution, we would create two extra stacks and divide the stack into two parts. We would recursively sort each stack, and then merge them back together in sorted order into the original stack. Note that this would require the creation of two additional stacks per level of recursion.

With the quicksort solution, we would create two additional stacks and divide the stack into the two stacks based on a pivot element. The two stacks would be recursively sorted, and then merged back together into the original stack. Like the earlier solution, this one involves creating two additional stacks per level of recursion.

3.6 Animal Shelter: An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog`, and `dequeueCat`. You may use the built-in `LinkedList` data structure.

pg 99

SOLUTION

We could explore a variety of solutions to this problem. For instance, we could maintain a single queue. This would make `dequeueAny` easy, but `dequeueDog` and `dequeueCat` would require iteration through the queue to find the first dog or cat. This would increase the complexity of the solution and decrease the efficiency.

An alternative approach that is simple, clean and efficient is to simply use separate queues for dogs and cats, and to place them within a wrapper class called `AnimalQueue`. We then store some sort of timestamp to mark when each animal was enqueued. When we call `dequeueAny`, we peek at the heads of both the dog and `cat` queue and return the oldest.

```
abstract class Animal {
  private int order;
  protected String name;
  public Animal(String n) { name = n; }
  public void setOrder(int ord) { order = ord; }
  public int getOrder() { return order; }

  /* Compare orders of animals to return the older item. */
  public boolean isOlderThan(Animal a) {
    return this.order < a.getOrder();
  }
}

class AnimalQueue {
  LinkedList<Dog> dogs = new LinkedList<Dog>();
  LinkedList<Cat> cats = new LinkedList<Cat>();
  private int order = 0; // acts as timestamp

  public void enqueue(Animal a) {
    /* Order is used as a sort of timestamp, so that we can compare the insertion
     * order of a dog to a cat. */
    a.setOrder(order);
    order++;

    if (a instanceof Dog) dogs.addLast((Dog) a);
    else if (a instanceof Cat) cats.addLast((Cat)a);
  }

  public Animal dequeueAny() {
    /* Look at tops of dog and cat queues, and pop the queue with the oldest
     * value. */
    if (dogs.size() == 0) {
      return dequeueCats();
    } else if (cats.size() == 0) {
      return dequeueDogs();
    }
```