nodes in the order in which they are viewed.

Interview Questions

3.1 **Three in One:** Describe how you could use a single array to implement three stacks.

Hints: #2, #12, #38, #58

pg 227

3.1 **Three in One:** Describe how you could use a single array to implement three stacks.

pg 98

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're okay with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: We will use the notation "[" to mean inclusive of an end point and "(" to mean exclusive of an end point.

- For stack 1, we will use [0, $^{n}/_{3}$).
- For stack 2, we will use [$^{n}/_{3}$, $^{2n}/_{3}$).
- For stack 3, we will use [$^{2n}/_{3}$, n).

The code for this solution is below.

```
class FixedMultiStack {
   private int numberOfStacks = 3;
   private int stackCapacity;
   private int[] values;
   private int[] sizes;

   public FixedMultiStack(int stackSize) {
      stackCapacity = stackSize;
      values = new int[stackSize * numberOfStacks];
      sizes = new int[numberOfStacks];
   }

   /* Push value onto stack. */
   public void push(int stackNum, int value) throws FullStackException {
      /* Check that we have space for the next element */
      if (isFull(stackNum)) {
         throw new FullStackException();
```

```
        }

        /* Increment stack pointer and then update top value. */
        sizes[stackNum]++;
        values[indexOfTop(stackNum)] = value;
    }

    /* Pop item from top stack. */
    public int pop(int stackNum) {
        if (isEmpty(stackNum)) {
            throw new EmptyStackException();
        }

        int topIndex = indexOfTop(stackNum);
        int value = values[topIndex]; // Get top
        values[topIndex] = 0; // Clear
        sizes[stackNum]--; // Shrink
        return value;
    }

    /* Return top element. */
    public int peek(int stackNum) {
        if (isEmpty(stackNum)) {
            throw new EmptyStackException();
        }
        return values[indexOfTop(stackNum)];
    }

    /* Return if stack is empty. */
    public boolean isEmpty(int stackNum) {
        return sizes[stackNum] == 0;
    }

    /* Return if stack is full. */
    public boolean isFull(int stackNum) {
        return sizes[stackNum] == stackCapacity;
    }

    /* Returns index of the top of the stack. */
    private int indexOfTop(int stackNum) {
        int offset = stackNum * stackCapacity;
        int size = sizes[stackNum];
        return offset + size - 1;
    }
}
```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

Approach 2: Flexible Divisions

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too much work.

```java
public class MultiStack {
  /* StackInfo is a simple class that holds a set of data about each stack. It
   * does not hold the actual items in the stack. We could have done this with
   * just a bunch of individual variables, but that's messy and doesn't gain us
   * much. */
  private class StackInfo {
    public int start, size, capacity;
    public StackInfo(int start, int capacity) {
      this.start = start;
      this.capacity = capacity;
    }

    /* Check if an index on the full array is within the stack boundaries. The
     * stack can wrap around to the start of the array. */
    public boolean isWithinStackCapacity(int index) {
      /* If outside of bounds of array, return false. */
      if (index < 0 || index >= values.length) {
        return false;
      }

      /* If index wraps around, adjust it. */
      int contiguousIndex = index < start ? index + values.length : index;
      int end = start + capacity;
      return start <= contiguousIndex && contiguousIndex < end;
    }

    public int lastCapacityIndex() {
      return adjustIndex(start + capacity - 1);
    }

    public int lastElementIndex() {
      return adjustIndex(start + size - 1);
    }

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

  private StackInfo[] info;
  private int[] values;

  public MultiStack(int numberOfStacks, int defaultSize) {
    /* Create metadata for all the stacks. */
    info = new StackInfo[numberOfStacks];
    for (int i = 0; i < numberOfStacks; i++) {
      info[i] = new StackInfo(defaultSize * i, defaultSize);
    }
    values = new int[numberOfStacks * defaultSize];
  }

  /* Push value onto stack num, shifting/expanding stacks as necessary. Throws
   * exception if all stacks are full. */
  public void push(int stackNum, int value) throws FullStackException {
```

```
    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.

Hints: #27, #59, #78

pg 232

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 `intvalue`, `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
PPP(); // 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.

(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.

Hints: #64, #81

pg 233

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

Hints: #98, #114

pg 236

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 every-thing 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. */
```

```java
    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.