Data Structure

Recitation VII
Topic

- Recursion: Stack trace
- Queue
Trace Recursive factorial

Executes factorial(4)

factorial(4)

Stack

Main method

Step 0: executes factorial(4)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24
Trace Recursive factorial

Step 0: executes factorial(4)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Executes factorial(3)
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Executes factorial(2)

Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)
Main method

Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6
Step 9: return 24
Trace Recursive factorial

factorial(4) -> return 4 * factorial(3) -> return 3 * factorial(2) -> return 2 * factorial(1) -> return 1 * factorial(0)

Step 0: executes factorial(4)
Step 1: executes factorial(3)
Step 2: executes factorial(2)
Step 3: executes factorial(1)
Step 4: executes factorial(0)
Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6

Executes factorial(1)

Space Required for factorial(4)
Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)
Main method

Stack
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Executes factorial(0)

Main method

Stack

Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method
Trace Recursive factorial

Step 0: executes factorial(4)
- return 4 * factorial(3)
  - Step 1: executes factorial(3)
    - return 3 * factorial(2)
      - Step 2: executes factorial(2)
        - return 2 * factorial(1)
          - Step 3: executes factorial(1)
            - return 1 * factorial(0)
              - Step 4: executes factorial(0)
                - return 1

Main method

Stack
- Space Required for factorial(0)
- Space Required for factorial(1)
- Space Required for factorial(2)
- Space Required for factorial(3)
- Space Required for factorial(4)
- Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 \times \text{factorial}(3)

Step 1: executes factorial(3)

return 3 \times \text{factorial}(2)

Step 2: executes factorial(2)

return 2 \times \text{factorial}(1)

Step 3: executes factorial(1)

return 1 \times \text{factorial}(0)

Step 4: executes factorial(0)

Step 5: return 1

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Main method

returns factorial(0)
Trace Recursive factorial

Main method

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Stack

return 1

Step 5: return 1

return 1

Step 4: executes factorial(0)

Step 3: executes factorial(1)

Step 2: executes factorial(2)

Step 1: executes factorial(3)

Step 0: executes factorial(4)

return 2 * factorial(1)

return 3 * factorial(2)

return 4 * factorial(3)

factorial(4)
Trace Recursive factorial
Trace Recursive factorial

Step 0: executes factorial(4)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

returns factorial(3)
Trace Recursive factorial

Step 0: executes factorial(4)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24

returns factorial(4)
factorial(4) Stack Trace

1. Space Required for factorial(4)
2. Space Required for factorial(3)
3. Space Required for factorial(2)
4. Space Required for factorial(1)
5. Space Required for factorial(0)
6. Space Required for factorial(1)
7. Space Required for factorial(2)
8. Space Required for factorial(3)
9. Space Required for factorial(4)
Typical call sequence

(1) Evaluate arguments left-to-right. If an argument is a simple variable or a literal value, there is no need to evaluate it. When an expression is used, the expression must be evaluated before the call can be made.

(2) Push a new stack frame on the call stack. When a method is called, memory is required to store the following information.

- Parameter and local variable storage. The storage that is needed for each of the parameters and local variables is reserved in the stack frame.
- Where to continue execution when the called method returns. It's automatically saved.
Typical call sequence

(3) Initialize the parameters. When the arguments are evaluated, they are assigned to the local parameters in the called method.

(4) Execute the method. After the stack frame for this method has been initialized, execution starts with the first statement and continues as normal. Execution may call on other methods, which will push and pop their own stack frames on the call stack. (Recursion)
(5) Return from the method. When a return statement is encountered, or the end of a void method is reached, the method returns. For non-void methods, the return value is passed back to the calling method. The stack frame storage for the called method is popped off the call stack.

Popping something off the stack is really efficient - a pointer is simply moved to previous stack frame. This means that the current stack frame can be reused by other methods. Execution is continued in the called method immediately after where the call took place.
Queue

- The basic concept of queues is one people normally find easy to understand, because we encounter them on a daily basis.

- When you go to the bank, unless you're rich enough to avoid having to wait in line, you will wait in a queue for the teller. The person at the front of the queue gets served as soon as a teller becomes free. When a new person enters the bank, they go to the back of the line (queue).

- This means that queues work on a FIFO (first in first out) basis.
Queue Operations

Enqueue

Back

Front

Dequeue
Queue Implementation

- Bounded Queue: Array based implementation
  - Fixed-Front Design Approach (Front Always Zero)
  - Floating-Front Design Approach (Circular Arrays)

- Un-bounded Queue: Linked based implementation
Fixed-Front Design Approach

- Similar solution like stack: keeping the front of the queue fixed in the first array slot and letting the rear move as we add new elements.

To keep the front of the queue fixed at the top of the array, we need to move every element in the queue up one slot.
Fixed-Front Design Approach

- What is the weakness of the design? The need to move all of the elements up every time we remove an element from the queue increases the amount of work needed to dequeue items.

- The technical terminology is that enqueue() is O(1) and dequeue() is O(N).
Floating-Front Design Approach

- If we keep track of the index of the front as well as the rear, we can let both ends of the queue float in the array.
Instead of moving elements up to the beginning of the array, it merely increments the front indicator to the next slot.
Letting the queue elements float in the array creates a new problem when the rear indicator reaches the end of the array. In our first design, this situation told us that the queue was full. Now, however, the rear of the queue might potentially reach the end of the (physical) array when the (logical) queue is not yet full.
Because space may still be available at the beginning of the array, the obvious solution is to let the queue elements "wrap around" the end of the array. In other words, we can treat the array as a circular structure. If not full, enqueue at: \[ \text{rear} = (\text{rear} + 1) \mod \text{maxQue} \];
An Interesting Question

- How to implement a queue by using two stacks?
- Stack: FILO
- Queue: FIFO
An Interesting Question

- We'll implement a FIFO queue using two stacks. Let's call the stacks InBox and OutBox. An element is inserted in the queue by pushing it into the InBox. An element is extracted from the queue by popping it from the OutBox. If the OutBox is empty then all elements currently in InBox are transferred to OutBox but in the reverse order.
public class Queue<E> {
    private Stack<E> inbox = new Stack<E>();
    private Stack<E> outbox = new Stack<E>();

    public void queue(E item) {
        inbox.push(item);
    }

    public E dequeue() {
        if (!outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
                outbox.push(inbox.pop());
            }
        }
        return outbox.pop();
    }
}
So...

- Build a stack using two queues.