Lesson 13-14

Objectives

- Stack Introduction
- Stack
  - The STL Stack Implementation
  - What is LIFO?
  - Use of Stack
- Stack via Array
  - Stack Representation in Array
  - Stack Implementation via Array
- Stack via Link List
  - Stack Representation in Link List
  - Stack Implementation via Link List
- Using stack to implement Recursion
  - Using two stacks to reverse an integer
  - Using one stack to reverse an integer
- Application of Stack
  - Evaluate a postfix expression using stack
  - Convert infix expression to postfix

Stack:

A ***stack*** is a version of a list that is particularly useful in applications involving reversing such as the problem will be given later on. In a stack data structure, all insertions and deletions of entries are made at one end, called the ***top*** of the stack. A helpful analogy is to think of a stack of trays or of plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack and employees place returned trays back on top of the stack. The tray most recently push on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.

When we add an item to a stack, we say that we ***push*** it onto the stack, and when we remove an item, we say that we ***pop*** it from the stack. Note that the last item pushed onto a stack is always the first that will be popped from the stack. This property is called ***last in, first out,*** or ***LIFO*** for short. A ***stack*** is particularly useful in applications involving *reversing* such as the problem at the end of this chapter.

A stack allows access to only one data item: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on. This capability is useful in many programming situations. In this section we'll see how a stack can be used to check whether parentheses, braces, and brackets are balanced in a computer program

source file. At the end of this chapter, we'll see a stack playing a vital role in parsing (analyzing) arithmetic expressions such as 3*(4+5).

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The Postal Analogy

To understand the idea of a stack, consider an analogy provided by the U.S. Postal Service. Many people, when they get their mail, toss it onto a stack on the hall table or into an "in" basket at work. Then, when they have a spare moment, they process the accumulated mail from the top down. First, they open the letter on the top of the stack and take appropriate action—paying the bill, throwing it away, or whatever. After the first letter has been disposed of, they examine the next letter down, which is now the top of the stack, and deal with that. Eventually, they work their Way down to the letter on the bottom of the stack (which is now the top). Figure shows a stack of mail. This letter

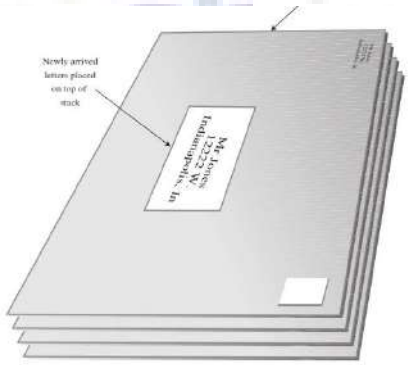
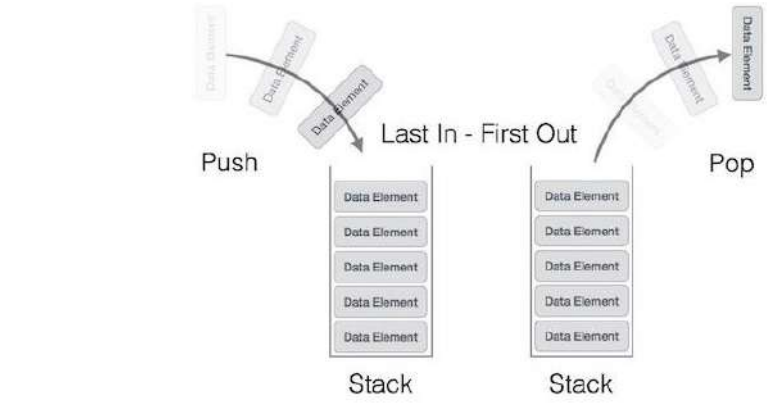
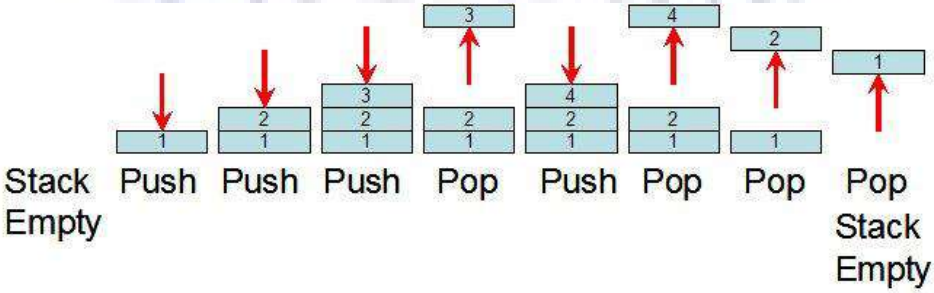


FIGURE A stack of letters.

This "do the top one first" approach works all right as long as you can easily process all the mail in a reasonable time. If you can't, there's the danger that letters on the bottom of the stack won't be examined for months, and the bills they contain will become overdue. Of course, many people don't rigorously follow this top-to-bottom approach. They may, for example, take the mail off the bottom of the stack, so as to process the oldest letter first. Or they might shuffle through the mail before they begin processing it and put higher-priority letters on top. In these cases, their mail system is no longer a stack in the computer-science sense of the word. If they take letters off the bottom, it's a queue; and if hey prioritize it, it's a priority queue. We'll look at these possibilities later. Shown in the following figure are the effects of push and pop operations on the stack.

Figure: Stack operations.





The STL Stack Implementation

```
/* Program stack.java
//demonstrates stacks
// to run this program:*/
public static void main(String[] args) {

        Scanner scan=new Scanner(System.in);
        Stack theStack=new Stack();
        int n;
        System.out.println("Push "+ theStack.maxSize +" Input:   ");
        for(int i=0;i<theStack.maxSize;i++)
        {
        n=scan.nextInt();
        theStack.push(n);                // push items onto stack
        }
        while( !theStack.isEmpty() )     // until it's empty,
        {                                // delete item from stack
        long value = theStack.pop();
        System.out.print(value);         // display it
        System.out.print("  ");
        }  // end while
        System.out.println("  ");
}
```

Output:

1 0 30 20 10

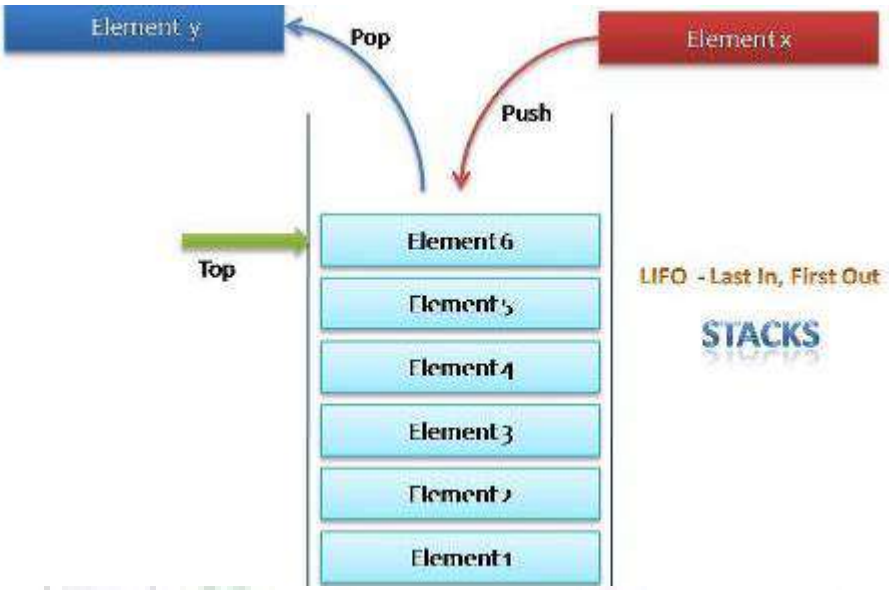
In the program *stack.java* the ***empty()*** member function returns a true if the stack is empty. Otherwise it returns a false

What is LIFO?

This means that the thing we added last is the first thing that gets pulled off.

Use of Stack:

1. Stack typically used for temporary storage of data.
2. Only one way of coming and outgoing.
3. If we want to get centered value we can't get that.



Examples:

1. Stack of books
2. Stack of plates

Plates are pushed onto top and popped off from top.

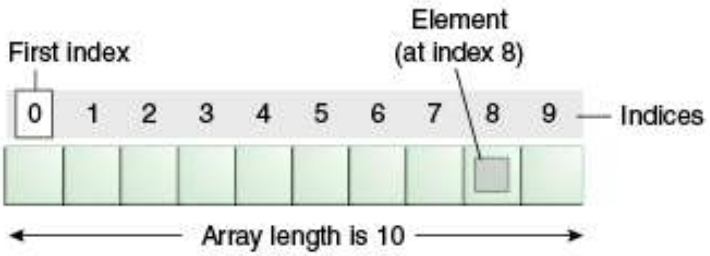
Stack Functions:

- Push**:** This function is used to add an item to the stack.
- Empty: This function is used to check whether stack is empty or not; often returns in boolean.
- Pop: This function is used to extract the most recent pushed value of the stack.

Stack via Array:

- **Stack Representation in Array:**

- To implement a stack items are inserted and removed at the same end (called the top).
- To use an array to implement stack you need both the array itself and an integer.
- The integer tells that which location is currently being pointed of the stack.
- How many elements are in the stack?



Stack Implementation via Array:

```
import java.util.Scanner;

public class Stack {

private int maxSize;        // size of stack array
private long[] stackArray;
private int top;            // top of stack

//--------------------------------------------------------------public
Stack()          // constructor
{
       Scanner scan=new Scanner(System.in);
       int s;
       System.out.println("Enter size of Stack:\t");
       s=scan.nextInt();
       maxSize = s;             // set array size
       stackArray = new long[maxSize];  // create array
       top = -1;                // no items yet
}
void push(long j)    // put item on top of stack
{
stackArray[++top] = j;     // increment top, insert item
}

public long pop()           // take item from top of stack
{
return stackArray[top--];  // access item, decrement top
}
//--------------------------------------------------------------public
long peek()          // peek at top of stack
{
return stackArray[top];
}
//--------------------------------------------------------------public
boolean isEmpty()    // true if stack is empty
{
```

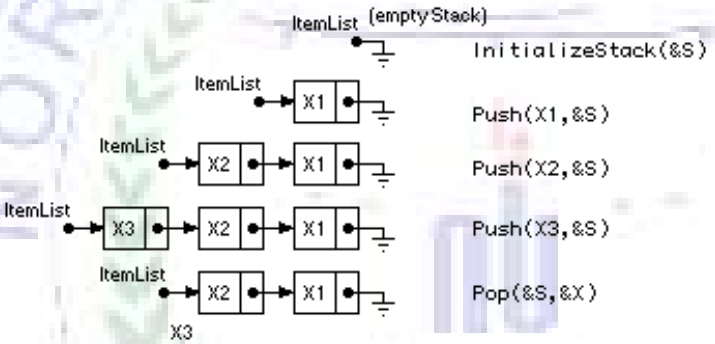
```
return (top == -1);
}
//--------------------------------------------------------------public
boolean isFull()     // true if stack is full
{
return (top == maxSize-1);
}
//--------------------------------------------------------------}
 // end class StackX
```

Stack Representation in Link List:

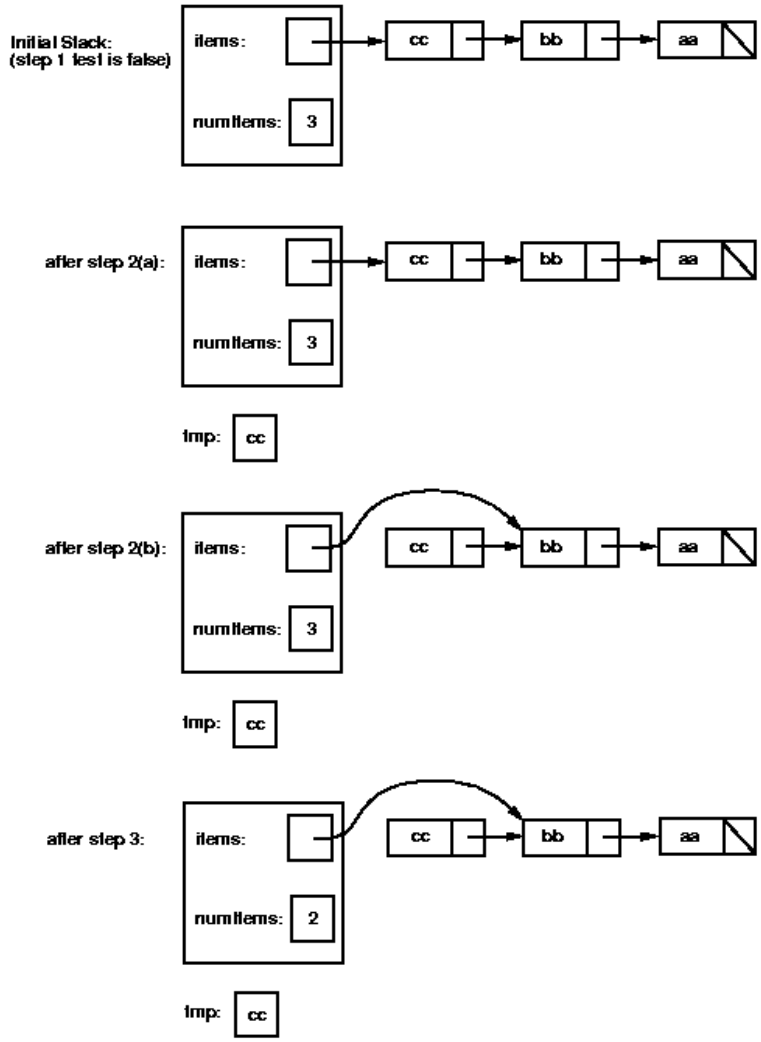
- Since all the action happens at the top of a stack, a singly link list is a fine way to implement it.
- The header of the list point to the top of the stack.

Push:

- Pushing is inserting an element at the front of the list.



POP.: Poping is to delete node from the front of list



- -With a link list representation overflow will not happen.
- -Underflow can happen. When a node is popped from list and the node references an object the reference (the pointer in node) does not need to be set to null.
- -Unlike an array implementation it really is removed.We can avoid size limitation of stack implemented with an array by using link list to hold the stack elements.
- -As with array however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.For a link list insert at start or end takes constant time using the head and current pointers respectively.
- -No memory consumption as compare to array.

Stack Implementation via Link List:

```
import static java.lang.System.exit;
 class StackUsingLinkedlist {

     private class Node {
```

```
    int data;
    Node link;    }
   Node top;
    StackUsingLinkedlist()
{
    this.top = null;
}

    public void push(int x) // insert at the beginning
{
        Node temp = new Node();


            if (temp == null) {
        System.out.print("\nHeap Overflow");
        return;
    }

  temp.data = x;

    temp.link = top;
            top = temp;
}
 public boolean isEmpty()
 {
    return top == null;
 }
 public int peek()
 {
  if (!isEmpty()) {
        return top.data;
    }
    else {
        System.out.println("Stack is empty");
        return -1;
    }
 }
public void pop() // remove at the beginning
 {
    if (top == null) {
        System.out.print("\nStack Underflow");
        return;
    }

   top = (top).link;
 }

 public void display()
 {
   if (top == null) {
        System.out.printf("\nStack Underflow");
        exit(1);
    }
    else {
        Node temp = top;
```

```
            while (temp != null) {

                System.out.printf("%d->", temp.data);

                    temp = temp.link;
                }
            }
        }
    }
    // main class
    public class GFG {
        public static void main(String[] args)
        {
            StackUsingLinkedlist obj = new StackUsingLinkedlist();
            obj.push(11);
            obj.push(22);
            obj.push(33);
            obj.push(44);
              obj.display();
            System.out.printf("\nTop element is %d\n", obj.peek());
            obj.pop();
            obj.pop();
            obj.display();
            System.out.printf("\nTop element is %d\n", obj.peek());
        }
    }
```

Using a Stack to Implement Recursion

With the ideas illustrated so far, we can implement any iterative process by specifying a register machine that has a register corresponding to each state variable of the process. The machine repeatedly executes a controller loop, changing the contents of the registers, until some termination condition is satisfied. At each point in the controller sequence, the state of the machine (representing the state of the iterative process) is completely determined by the contents of the registers (the values of the state variables).

Implementing recursive processes, however, requires an additional mechanism. Consider the following recursive method for computing factorials, which we first examined in section:

```
(define (factorial n)

  (if ( n == 1)

      Return 1

      (else return (factorial (n - 1))* n)))
```

As we see from the procedure, computing *n*! requires computing (*n*-1)!. Our GCD machine, modeled on the procedure

```
(define (gcd a b)

  (if (b == 0)
```

```
return      a

else


      return(gcd b (remainder a b))))
```

Similarly had to compute another GCD. But there is an important difference between the `gcd` procedure, which reduces the original computation to a new GCD computation, and `factorial`, which requires computing another factorial as a sub-problem. In GCD, the answer to the new GCD computation is the answer to the original problem. To compute the next GCD, we simply place the new arguments in the input registers of the GCD machine and reuse the machine's data paths by executing the same controller sequence. When the machine is finished solving the final GCD problem, it has completed the entire computation.

In the case of factorial (or any recursive process) the answer to the new factorial sub-problem is not the answer to the original problem. The value obtained for (*n*-1)!, must be multiplied by *n* to get the final answer. If we try to imitate the GCD design, and solve the factorial sub-problem by decrementing the `n` register and rerunning the factorial machine, we will no longer have available the old value of `n` by which to multiply the result. We thus need a second factorial machine to work on the sub-problem. This second factorial computation itself has a factorial sub-problem, which requires a third factorial machine, and so on. Since each factorial machine contains another factorial machine within it, the total machine contains an infinite nest of similar machines and hence cannot be constructed from a fixed, finite number of parts.

Nevertheless, we can implement the factorial process as a register machine if we can arrange to use the same components for each nested instance of the machine. Specifically, the machine that computes *n*! should use the same components to work on the sub-problem of computing (*n*-1)!, on the sub-problem for (*n*-2)!, and so on. This is plausible because, although the factorial process dictates that an unbounded number of copies of the same machine are needed to perform a computation, only one of these copies needs to be active at any given time. When the machine encounters a recursive sub-problem, it can suspend work on the main problem, reuse the same physical parts to work on the sub-problem, then continue the suspended computation.

In the sub-problem, the contents of the registers will be different than they were in the main problem. (In this case the `n` register is decremented.) In order to be able to continue the suspended computation, the machine must save the contents of any registers that will be needed after the sub-problem is solved so that these can be restored to continue the suspended computation. In the case of factorial, we will save the old value of `n`, to be restored when we are finished computing the factorial of the decremented `n` register.

Since there is no *a priori* limit on the depth of nested recursive calls, we may need to save an arbitrary number of register values. These values must be restored in the reverse of the order in which they were saved, since in a nest of recursions the last sub-problem to be entered is

the first to be finished. This dictates the use of a *stack*, or ``last in, first out'' data structure, to save register values. We can extend the register-machine language to include a stack by adding two kinds of instructions: Values are placed on the stack using a `save` instruction and restored from the stack using a `restore` instruction. After a sequence of values has been `save`d on the stack, a sequence of `restore`s will retrieve these values in reverse order.

With the aid of the stack, we can reuse a single copy of the factorial machine's data paths for each factorial sub-problem. There is a similar design issue in reusing the controller sequence that operates the data paths. To re-execute the factorial computation, the controller cannot simply loop back to the beginning, as with an iterative process, because after solving the (*n*-1)! Sub-problem the machine must still multiply the result by *n*. The controller must suspend its computation of *n*!, solve the (*n*-1)! Sub-problem, then continue its computation of *n*!. This view of the factorial computation suggests the use of the subroutine mechanism described in section, which has the controller use a `continue` register to transfer to the part of the sequence that solves a sub-problem and then continue where it left off on the main problem. We can thus make a factorial subroutine that returns to the entry point stored in the `continue` register. Around each subroutine call, we save and restore `continue` just as we do the `n` register, since each ``level'' of the factorial computation will use the same `continue` register. That is, the factorial subroutine must put a new value in `continue` when it calls itself for a sub-problem, but it will need the old value in order to return to the place that called it to solve a sub-problem.

Figure shows the data paths and controller for a machine that implements the recursive `factorial` procedure. The machine has a stack and three registers, called `n`, `val`, and `continue`. To simplify the data-path diagram, we have not named the register-assignment buttons, only the stack-operation buttons (`sc` and `sn` to save registers, `rc` and `rn` to restore registers). To operate the machine, we put in register `n` the number whose factorial we wish to compute and start the machine. When the machine reaches `fact-done`, the computation is finished and the answer will be found in the `val` register. In the controller sequence, `n` and `continue` are saved before each recursive call and restored upon return from the call. Returning from a call is accomplished by branching to the location stored in `continue`. `Continue` is initialized when the machine starts so that the last return will go to `fact-done`. The `val` register, which holds the result of the factorial computation, is not saved before the recursive call, because the old contents of `val` is not useful after the subroutine returns. Only the new value, which is the value produced by the sub-computation, is needed.

Although in principle the factorial computation requires an infinite machine, the machine in figure is actually finite except for the stack, which is potentially unbounded. Any particular physical implementation of a stack, however, will be of finite size, and this will limit the depth of recursive calls that can be handled by the machine. This implementation of factorial illustrates the general strategy for realizing recursive algorithms as ordinary register machines augmented by stacks. When a recursive sub-problem is encountered, we save on the stack the registers whose current values will be required after the sub-problem is solved, solve the recursive sub-problem, then restore the saved registers and continue execution on the main problem. The `continue` register must always be saved. Whether there are other registers that

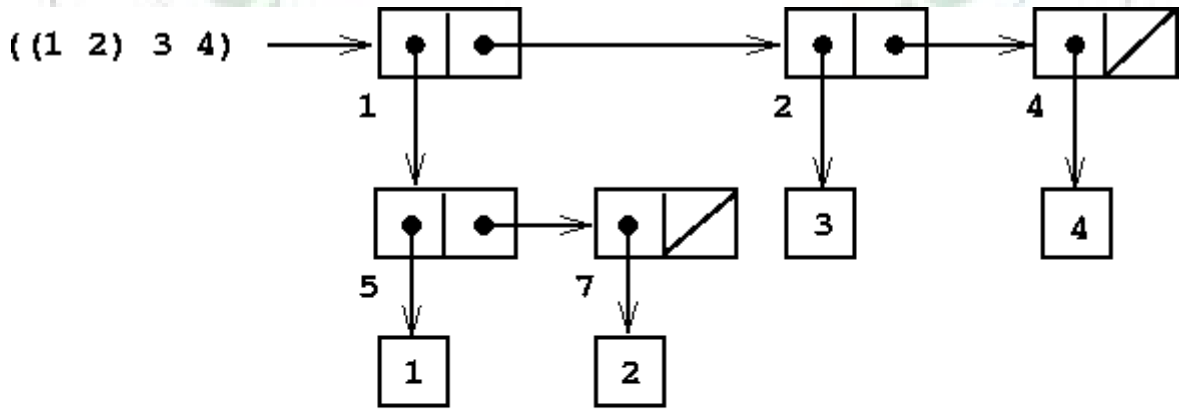
need to be saved depends on the particular machine, since not all recursive computations need the original values of registers that are modified during solution of the sub-problem (see exercise).

A double recursion

Let us examine a more complex recursive process, the tree-recursive computation of the Fibonacci numbers, which we introduced in section :

```
(define (fib n)
  (if (n < 2)
      Return n
      (return(fib (n - 1))+ (fib (n - 2)))))
```

Just as with factorial, we can implement the recursive Fibonacci computation as a register machine with registers `n`, `val`, and `continue`. The machine is more complex than the one for factorial, because there are two places in the controller sequence where we need to perform recursive calls--once to compute Fib(*n*-1) and once to compute Fib(*n*-2). To set up for each of these calls, we save the registers whose values will be needed later, set the `n` register to the number whose Fib we need to compute recursively (*n*-1 or *n*-2), and assign to `continue` the entry point in the main sequence to which to return (`afterfib-n-1` or `afterfib-n-2`, respectively). We then go to `fib-loop`. When we return from the recursive call, the answer is in `val`. Figure shows the controller sequence for this machine.



Index	0	1	2	3	4	5	6	7	8	...
the-cars		p5	n3		n4	n1		n2		...
the-cdrs		p2	p4		e0	p7		e0		...

Exercise. Specify register machines that implement each of the following procedures. For each machine, write a controller instruction sequence and draw a diagram showing the data paths.

a. Recursive exponentiation:

```
(define (expt b n)
  (if (n = 0)
      Return 1
      (return b * (expt b ( n - 1)))))
```

b. Iterative exponentiation:

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (counter == 0)
        Return product
        Return (expt(counter - 1) - iter (b * product))))
```

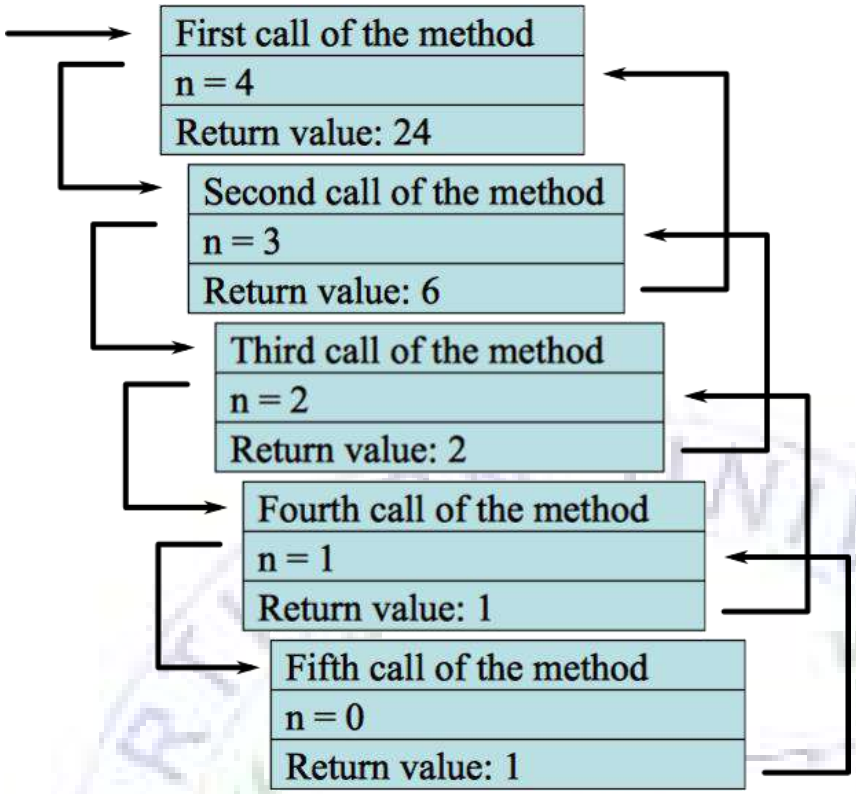
Exercise. Hand-simulate the factorial and Fibonacci machines, using some nontrivial input (requiring execution of at least one recursive call). Show the contents of the stack at each significant point in the execution.

Exercise. Ben Bitdiddle observes that the Fibonacci machine's controller sequence has an extra `save` and an extra `restore`, which can be removed to make a faster machine. Where are these instructions?

```
int factorial(int number) {
        int temp;

        if(number <= 1) return 1;

        temp = number * factorial(number - 1);
        return temp;
```



Reversing the Integer value using Stack with Array Implementation

- **Reverse an integer by two stacks:**

```
import java.util.Scanner;

public class Stack {

	private int maxSize;        // size of stack array
	private long[] stackArray;
	private int top;            // top of stack

	//--------------------------------------------------------------public
	Stack()          // constructor
	{
		Scanner scan=new Scanner(System.in);
		int s;
		System.out.println("Enter size of Stack:\t");
		s=scan.nextInt();
		maxSize = s;             // set array size
		stackArray = new long[maxSize];  // create array
		top = -1;                // no items yet
	}
	//--------------------------------------------------------------public
	void push(long j)    // put item on top of stack
	{
	stackArray[++top] = j;     // increment top, insert item
	}

	public long pop()           // take item from top of stack
	{
	return stackArray[top--];  // access item, decrement top
```

```
        }
        //--------------------------------------------------------------public
        long peek()          // peek at top of stack
        {
        return stackArray[top];
        }
        //--------------------------------------------------------------public
        boolean isEmpty()    // true if stack is empty
        {
        return (top == -1);
        }
        //--------------------------------------------------------------public
        boolean isFull()     // true if stack is full
        {
        return (top == maxSize-1);
        }
        //--------------------------------------------------------------}
         // end class StackX
        ////////////////////////////////////////////////////////////////

        public static void main(String[] args) {

              Scanner scan=new Scanner(System.in);
              Stack theStack=new Stack();
              Stack S=new Stack();

              int n;
              System.out.println("Push "+ theStack.maxSize +" Input:    ");
              for(int i=0;i<theStack.maxSize;i++)
              {
              n=scan.nextInt();
              theStack.push(n);              // push items onto stack
              }
              while( !theStack.isEmpty() )      // until it's empty,
              {                                 // delete item from stack
              long value = theStack.pop();
              System.out.print(value);      // display it
              S.push(value);
              System.out.print("  ");
              }  // end while
              System.out.println("  ");


}

        }
```

Stack Application:

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2). Example : *+AB-CD (Infix : (A+B) * (C-D))

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator). Example : AB+CD-* (Infix : (A+B * (C-D))

The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation e.g.:

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

Algorithm for Prefix to Postfix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack Create a string by concatenating the two operands and the operator after them.
- string = operand1 + operand2 + operator
- And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

Code

```
import java.util.*;

class GFG
{
static boolean isOperator(char x)
{
    switch (x)
    {
        case '+':
        case '-':
        case '/':
        case '*':
        return true;
    }
    return false;
}

static String preToPost(String pre_exp)
{

    Stack<String> s= new Stack<String>();
    int length = pre_exp.length();
    for (int i = length - 1; i >= 0; i--)
    {
        if (isOperator(pre_exp.charAt(i)))
        {
            String op1 = s.peek(); s.pop();
            String op2 = s.peek(); s.pop();
          String temp = op1 + op2 + pre_exp.charAt(i);
           s.push(temp);
        }
        else
        {
           s.push( pre_exp.charAt(i)+"");
        }
```

```
        }
        return s.peek();
    }
    public static void main(String args[])
    {
        String pre_exp = "*-A/BC-/AKL";
        System.out.println("Postfix : " + preToPost(pre_exp));
    }
    }
```

Algorithm for Postfix to Prefix:

- Read the Postfix expression from left to right
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack Create a string by concatenating the two operands and the operator before them. string = operator + operand2 + operand1
- And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

// Java Program to convert postfix to prefix

```
import java.util.*;
class GFG {
    static boolean isOperator(char x)
    {

        switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
        }
        return false;
    }
    static String postToPre(String post_exp)
    {
        Stack<String> s = new Stack<String>();
        int length = post_exp.length();
        for (int i = 0; i < length; i++) {
            if (isOperator(post_exp.charAt(i))) {
                String op1 = s.peek();
                s.pop();
                String op2 = s.peek();
                s.pop();
                String temp = post_exp.charAt(i) + op2 + op1;
                s.push(temp);
            }
            else {
                s.push(post_exp.charAt(i) + "");
            }
```

```
        }
        return s.peek();
}

        public static void main(String args[])
{
        String post_exp = "ABC/-AK/L-*";
        System.out.println("Prefix : " + postToPre(post_exp));
} }
```