1 The STACK ADT

A Stack is an example of a linear data structure. Linear data structures are used to store collection of elements. In particular, the elements are arranged linearly—i.e., the data structures is really employed to maintain a sequences of elements. In this course we will study different types of linear data structures; they will mostly differ on the type of operations that are allowed on the data structure (e.g., they impose different restrictions on which elements of the sequence can be accessed).

Please, keep in mind that, since stacks are employed to store a sequence of elements, the order in which elements are indicated is part of the definition (thus sequences with the same elements in different order represent distinct values). Observe also that we will concentrate here on homogenous stacks, which means that we assume that a stack can store only elements that have all the same type. Thus, we will talk about stacks of integers (where the stack contains a sequence of integer elements), stacks of double (where the stack contains a sequence of doubles), etc.

The main characteristic of stacks is that they allow insertion and deletion to occur only at one extreme of the sequence. Each new element can be added only at the beginning of the sequence, and each element can be only removed from the beginning of the sequence.

We commonly assign names to the two extremes of the sequence of elements contained in a stack. The extreme which is affected by insertion and deletion operations is called top of the stack, while the opposite extreme is called bottom of the stack.

The behavior of a stack is akin to a LIFO structure (Last In First Out). This means that the first deletion of an element is always going to remove from the sequence the last element which has been inserted.

2 Specification

The specification of the Stack ADT includes, as usual, description of the domain of the data type and of the operations which are allowed. The domain is the same as in the case of linear lists, with the only addition of the explicit identification of one extreme of the sequence as the top of the stack.

The complete specification of the Stack ADT is presented in Figure 1. Let us review the basic ideas:

**Set of values:** the set of possible values is represented by the set of all possible sequences of elements; i.e., each value is of the form \((a_1, \ldots, a_n)\), where \(a_i\) identifies the \(i^{th}\) element in the sequence. The element occupying the rightmost position in the sequence is assumed to be the top of the stack (e.g., \(a_n\)).

**Operations:**

- **Constructors:** the only constructor needed is the one to create a new object of type stack.
  
  CreateStack
  
  - preconditions: none
  - postconditions: returns a new object of type stack containing, as value, the empty stack (i.e., the sequence containing zero elements);
AbstractDataType Stack
{
   instances: ordered finite collections of zero or more elements from a
given base type; the right extreme of the sequence is the top
operations:
   1. Constructors:
      CreateStack
      input: nothing
      output: a new stack data object containing the empty sequence

   2. Destructors:
      DeleteStack
      input: none
      output: nothing
      effect: destroys the object given in input

   3. Inspectors:
      EqualStacks
      input: one stack
      output: a boolean value; TRUE if the input stack is equal to the current one
      (they contain the same sequences of elements); FALSE otherwise

      EmptyStack
      input: none
      output: boolean value; TRUE if the stack contains the empty sequence, FALSE otherwise

      StackSize
      input: none
      output: an integer value identifying the length of the sequence currently
      stored in the input data object

      StackTop
      input: none
      output: the element in position in the top position of the sequence stored in the stack

      toString
      input: none
      output: a String representing the description of the content of the stack
      effect: creates a string which represents, in a nice format, the content
      of the stack (so that it can be printed on the screen)

   4. Modifiers
      Push
      input: an element of the base type
      output: a boolean indicating whether the operation was successful
      effect: the element is added to the sequence in the stack, in the top position

      Pop
      input: none
      output: a boolean indicating whether the operation was successful
      effect: the element in the top position in the sequence in the stack is removed

      copyStack
      input: none
      output: a stack
      effect: creates a new stack which has the same content as the current one
}

Figure 1: Abstract Data Type specification for the Stack ADT
• **Destructors**: the only destructor needed is used to trash a stack which is not needed any longer.

  **DeleteList**
  
  – preconditions: receives an existing stack as input
  – postconditions: none

• **Inspectors**: the following inspectors are included in the type specification:

  1. **StackSize** used to compute the size of a stack (i.e., the number of elements present in a given stack)
     
     – preconditions: operates on the current stack
     – postconditions: produces a number representing the count of the elements in the stack
  2. **EmptyStack** used to verify whether a given stack is empty or not
     
     – preconditions: operates on the current stack
     – postconditions: produces true (i.e., 1 in C) if the stack is empty, false (i.e., 0 in C) otherwise
  3. **StackTop** returns the information which is currently stored as top element in the stack (i.e., the element in the rightmost position in the sequence stored in the stack). This is illustrated in the following figure

```
< 10  20  30  40 >
```

  – preconditions: operates on the current stack
  – postconditions: if the stack is not empty, then it returns the first element in the stack, otherwise it gives an error

• **Modifiers**: the following modifiers are included

  1. **Push** used to insert a new element as the top of the stack. This is illustrated in the following figure:

```
<10,20,30>  <10,20,30,40>
```

  – preconditions: operates on the current stack and receives as input an element $x$
  – postconditions: if $x$ is of the correct type (i.e., it has the same type as all the other elements in the stack), and then the stack is modified introducing the element $x$ in the first position of the stack; otherwise it returns an error

  2. **Pop** removes the top element from the stack. This is illustrated in the following figure

```
<10,20,30,40>  <10,20,30>
```

  – preconditions: operates on the current stack
– postconditions: if the stack is not empty, then the stack is modified by removing the element in first position in the stack; otherwise it returns an error

3. `copyStack` creates a new stack with the same content as the current one
   – preconditions: operates on the current stack
   – postconditions: creates a brand new stack which has the same content as the current one

3 Public Part of the Implementation

The public part can be obtained as direct translation of the ADT specification. Following a practice that we will employ throughout the course, we make the first step in the implementation the development of an `interface` for the stack data type. We will later proceed in the development of different implementations of this interface, using different techniques to implement stacks.

```java
/**
 * Interface for a stack ADT; each stack will be used to
 * store a sequence of integers; the operations allows to
 * operate on the sequence by adding and removing elements
 * at one of the two extremes of the sequence.
 * @author Enrico Pontelli
 **/

public interface intStackInterface {
    // note that here we do not say anything about how to represent
    // a stack, as these are details that belong to the implementation
    // of the interface

    /*------------------------------------------------------------------*/
    /* Operations */
    /*------------------------------------------------------------------*/

    /**
     * the constructors and destructors are left out, as they depend
     * on the specific implementation of the interface
     **/

    /*------------------------------------------------------------------*/
    /* Inspectors */
    /*------------------------------------------------------------------*/

    /**
     * checks whether the stack is empty or not
     * @return true if stack is empty, false otherwise
     **/
    public boolean emptyStack ();

    /**
     * determines the number of elements in the stack (the
     * length of the sequence in the stack
     * @return an integer representing the number of elements in the stack
     **/
    public int StackSize ();

```
/**
 * Determines the current top of stack
 * @return the element on the top of the stack
 **/
public int stackTop();

/**
 * Creates a description of the content of the stack
 * @return a string that represents the content of the stack
 **/
public String toString();

/*-------------------*/
/* Modifiers */
/*-------------------*/

/**
 * Push new element in the stack
 * @param element the element to be pushed on the stack
 * @return true if the operation is successful, false otherwise
 **/
public boolean Push(int element);

/**
 * pop an element from the stack if not empty
 * @return true if the operation is successful, false otherwise
 **/
public boolean Pop();
}

Let us make a few observations regarding this interface.

- In this example, the stack is composed of integers. We will tackle later how to relax this type restriction.
- some operations that were described in the specification of the ADT do not appear in the interface. These are operations that are tied to the specific implementation of the interface (equality between stacks, constructors, destructors, and copy of stacks). These operations will have to be provided in each specific implementation of the interface. The interface, remember, is used only to provide a common specification to be used by different implementations of the data type.

4 Implementation Using Arrays

A simple implementation of stacks can be achieved by using arrays. The idea is that the array contains the element of the sequence stored in the stack, and a single index has to be maintained in order to indicate where is currently the top of the stack. This is briefly illustrated in Figure 2. The stack in the figure is assumed to contain the sequence ⟨10, 20, 30, 40⟩ where 40 is the element in the top of the stack.

We are immediately designing a structure for the stack representation which allows for extensions in case of overflow—i.e., if we run out of space in the array, the array is replaced by a dynamically created new array larger then the previous one. This allows us to remove restrictions on the maximum number of elements that can be stored in the stack. Observe that in the structure representing a stack we have two fields, one called count which keeps track of the number of elements currently in the stack, and one called top which indicates...
the top element in the stack. For simplicity, we assume that actually \texttt{top} indicates what is the position of the array which comes after the top element—this allows us to use \texttt{top} directly for inserting new elements and allows us to avoid designing special values for \texttt{top} in the case of an empty stack. Under these assumptions, one could easily observe that at any point in time \texttt{top} and \texttt{count} have always the same value; thus, one could optimize the implementation by removing one of the two fields. For the sake of simplicity we have not adopted such optimization in the implementation described here.

Regarding the various operations:

- in the creation of a stack we need to allocate the initial array which will contain the elements in the stack. Initially the stack is set to empty (i.e., the \texttt{count} field is set to zero).

- the operation which reads the top of the stack is obvious, since the structure representing the stack contains the field \texttt{top} which indicates the array location which is immediately after the top of the stack.

- the operation which pushes a new element on the stack is also obvious, since the \texttt{top} field of the structure is indicating in which position in the array the new element has to be placed;

- the operation which pops an element from the array is simply achieved by decrementing the \texttt{top} field of the structure.

The following implementation is for stack of elements of type integer.

```c
/**
 * STACK ADT - IMPLEMENTATION
 * this implements the intStack ADT using arrays
 */
```
import java.io.*;

public class intStackArray implements intStackInterface
{

/* data members describing a stack */

final int MAXSIZE = 50; // max size of array
private int space[]; // used to store the elements of the stack
private int count; // count of elements in stack
private int top; // identify the current top

/***
* constructor; create a new stack, initially empty
* **/
public intStackArray()
{

    space = new int[MAXSIZE]; // create the array for the elements of the stack
    count = 0; // no elements in the stack
    top = 0; // no top element
}

/***
* destructor; do not need to do anything
***/
```java
/**
 * StackSize
 * determines how many elements are currently in the stack
 * @return the number of elements in the stack
 ***/
public int stackSize ()
{
    return (count);
}

/***
 * StackTop
 * determines the element currently on the top of the stack
 * @return the integer currently on the top of the stack
 ***/
public int stackTop ()
{
    if (emptyStack())
    {
        System.out.println("Attempting to access empty stack");
        System.exit(0);
    }
    else
    return (space[top - 1]);
}

/***
 * equalStack
 * determines if a given stack has the same content as the current one
 * @param s stack to be compared with
 * @return true if the two stacks have same content
 ***/
public boolean equalStack (intStackArray s)
{
    if (count != s.stackSize())
    return false;
    else
    {
        intStackArray temp = s.copyStack();       // copy the input stack

        for (int i=count-1; i >= 0; i--)
        if (space[i] != temp.stackTop())       // compare the tops
            return false;
        else
            temp.Pop();
        return true;
    }
}
```
/**
 * toString: creates a representation of the content of the stack
 * @return a string describing the content of the stack
 ***/
public String toString()
{
    String s;
    s = "Content: ";
    for (int i=0; i < count; i++)
        s = s + " " + space[i];
    return s;
}

/**
 * Push
 * takes an element as input; it modifies the stack by pushing
 * the element as new top
 * @param element the element to be pushed on the stack
 * @return true if the operation is successful, false otherwise
 ***/
public boolean Push (int element)
{
    if (count == space.length)
        return false; // array is full; fail
    else
    {
        space[top] = element;
        top++;  
        count++;
        return true;
    }
}

/**
 * Pop
 * modifies the stack by removing the current top
 * @return true if the operation is successful, false otherwise
 ***/
public boolean Pop ()
{
    if (emptyStack())
        return false;
    else
    {
        top--;
count--;  
    return true;
}
}

/**  
 * copyStack  
 * create a copy of the current stack (same content)  
 *  
 */
public intStackArray copyStack()
{
    intStackArray n = new intStackArray();

    for (int i=0; i < count; i++)
        n.Push(space[i]);

    return n;
}

Observe:

• each stack is created as an instance of the intStackArray class; each object will contain three pieces of information: the count of elements in the stack, the index of the next available position in the array, and an array which is going to store the elements of the stack.

• a constant (MAXSIZE) is used to denote the size of the array, and thus, the size of the stack.

• creation leads to the allocation of the memory area for the descriptor of the stack;

• the top operation simply accesses the array element in position top-1 (since the index top points to the next available location—i.e., the location after the actual top element of the stack);

• push operation fills the location of index top (and moves this index forward); pop operation simply moves the top index backward;

The following is a simple test example that makes use of stacks. It creates a stack and perform some simple operations on it.

import java.io.*;

class teststack
{
    public static void main(String args[])
    {
        intStackArray s = new intStackArray();

        s.Push(10);
        s.Push(20);
        s.Push(30);

        System.out.println("Currently there are "+s.stackSize()+" elements");
4.1 Making the Push Operation more Flexible

In the implementation described above, we are setting a strict limit on how many elements we can have on the stack at any given time. The array space is created of a given size, and once it is filled completely, we will not be able to perform any further Push operations—indeed, we have the provision for the Push operation to fail and return the boolean value false.

This restriction on the size can be relaxed. If the current array fills up, then we can simply throw it away and replace it with a bigger one; this extension operation can be repeated as many times as necessary.

The question is: how do we perform such extension? The process is quite simple. We will need to do the following steps:

- create a new, bigger array
- copy all the elements from the old array to the new one
- replace the old array with the new one

The modified version of the Push operation is as follows:

```java
/**
 * Push
 * takes an element as input; it modifies the stack by pushing
 * the element as new top
 *
 * @param element the element to be pushed on the stack
 * @return true if the operation is successful, false otherwise
 ***/
public boolean Push (int element) {
    if (count == space.length) {
        // array is full - perform extension

        // create an array twice the size of the current one
        int[] newarray = new int[space.length*2];

        // copy elements from old array to new one
        for (int i=0; i<count; i++)
            newarray[i] = space[i];

        // replace the old array with the new one
        space = newarray;
    }
    space[top] = element;
    top++;
    count++;
}
```
4.2 Complexity Considerations

Measuring the complexity of these operations is relatively simple:

- the creation and deletion of a stack are clearly constant time operations;
- all the inspectors, including the stackTop operation, are constant time operations, as they require at most accessing one location of the array using an existing index; the only exception is the equal operation, which is linear (as it requires traversing the whole array).
- the pop operation is also constant time, since it simply requires decrementing two variables;
- the push operation is constant time as long as there is space for the new element in the existing array; if the array is full, then we need to create a new, larger array before continuing with the push operation. In this last case the operation has a complexity which is linearly proportional to the number of elements in the stack (since all the elements in the stack have to be copies from one array to the other).
- the print and assign operations have clearly linear time complexity as they require going through all the elements in the stack.

5 Some Applications of Stacks

5.1 Parenthesis Matching

When a compiler perform analysis of the source program in order to verify the syntactic correctness, one of the tasks performed is to verify the syntactic correctness of all the expressions used. In particular, verifying the correctness of an expression requires checking that all the parenthesis in the expression are properly balanced—i.e., each open parenthesis should be followed by a closed parenthesis. Thus, an expression like \( (a * (b + c) + d) \) is using parenthesis correctly while an expression like \( (a + b)) \) is not.

We would like to write a function which receives in input a string representing an expression and returns TRUE/FALSE depending on whether parenthesis in the expressions are used correctly or not. The function will make use of a stack to support its activities.

The strategy is simple: each open parenthesis '(' encountered is pushed on the stack (we assume we are using a stack with base type char). Each time we encounter a closed parenthesis ')' we pop one element from the stack (if there is any). If all the pop operations are successful and at the end of the expression the stack is empty, then it means that the use of parenthesis was correct, otherwise there was an error.

```java
public static boolean checkParenthesis (String expr)
{
    intStackArray s = new intStackArray();
    int i;
    for (i=0; i < expr.length(); i++)
    {
        if (expr.charAt(i) == '(')
            s.Push('(');  // push open parenthesis on the stack
        if (expr.charAt(i) == ')
            if (s.emptyStack())
                return false;  // there are too many closed parenthesis
            else
                s.Pop();
    }
    return true;
}
```
s.Pop(); // match '(' and ')
}

if (s.emptyStack())
    return true;
else
    return false; // there are too many open parenthesis

5.2 Evaluating Postfix

Another problem related to the manipulation of expressions is the transformation of expressions from infix notation to postfix notation. In infix notation the operations of the expressions are placed in between the operands (e.g., $a + b$); this is the “standard” way of writing expressions:

$$a + b * c - d$$

In postfix notation instead the operands are written first and the operation is written after (e.g., $a b +$).

Computers don’t like infix notations: infix notation requires the use of parenthesis in order to control the order of execution of the different operations. This is not the case in postfix notation—each expression can be unambiguously written without the use of parenthesis. For example, the expression

$$a * b - (c + d) + e$$

is represented in postfix notation as

$$a b * c d + - e +$$

Given an expression in postfix notation, it is easy to develop an algorithm which evaluates the expression, making use of a stack. The idea is that the expression is read from left to right; each operand is pushed on the stack; each time an operation is encountered, this is applied to the two top elements of the stack (which are popped), and the result is pushed back on the stack.

For example, the postfix expression

$$2 4 6 + *$$

(which represents the expression $2 * (4 + 6)$) is evaluated as in the following figure:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 6 + *</td>
<td></td>
</tr>
<tr>
<td>4 6 + *</td>
<td>2</td>
</tr>
<tr>
<td>6 + *</td>
<td>2 4</td>
</tr>
<tr>
<td>+ *</td>
<td>2 4 6</td>
</tr>
<tr>
<td>*</td>
<td>2 10</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

A function which accepts a postfix expression (as a string) and computes the value can be written as follows:

```java
public static String skip_spaces(String expr)
{
    return expr.trim();
}
```
public static String skip_number(String expr)
{
    int i=0;
    while (Character.isDigit(expr.charAt(i)))
        i++;
    return expr.substring(i);
}

public static int read_number(String expr)
{
    int val = 0;
    int index = 0;
    while (Character.isDigit(expr.charAt(index)))
    {
        val = val * 10 + (expr.charAt(index) - '0');
        index++;
    }
    return val;
}

public static int evaluatePostfix (String expr)
{
    intStackArray s = new intStackArray();
    int val;
    int op1,op2;
    int position = 0;
    expr = skip_spaces(expr);
    while (expr.length() > 0)
    {
        if (Character.isDigit(expr.charAt(0))) // read operand and push it
        {
            val = read_number(expr);
            expr = skip_number(expr);
            s.Push(val);
        }
        else // pop two top operands and apply
        {
            op1 = s.stackTop();
            s.Pop();
            op2 = s.stackTop();
            s.Pop();
            if (expr.charAt(0) == '+')
                s.Push(op1+op2);
            if (expr.charAt(0) == '-')
                s.Push(op2-op1);
            if (expr.charAt(0) == '*')
                s.Push(op1*op2);
        }
    }
}
```java
if (expr.charAt(0) == '/')
    s.Push(op2/op1);
expr = expr.substring(1);
}
expr = skip_spaces(expr);
}
val = s.stackTop();
return (val);
}
```

5.3 Rearranging Railroad Cars

A freight train has \( n \) railroad cars. Each is to be left at a different station. Let us assume that the different stations are numbered 1 through \( n \) and let us assume that the train is going to visit the stations in the order \( n, n-1, n-2, \ldots, 2, 1 \). Each railroad car is labeled with its station of destination.

The railroad cars are downloaded from a ship and they will arrive in any arbitrary order. The objective is to rearrange the cars so that they are in the order 1 through \( n \) from front to back (so that at each station we simply need to detach the last car).

The rearrangement process is performed as follows: the cars downloaded from the ship are on an input track. The train will leave from an output track. Between the input and output tracks there are a number of holding tracks which can be used to temporary park cars. The initial and final configuration are shown in the following figure, which assumes the presence of three holding tracks.

<table>
<thead>
<tr>
<th></th>
<th>Input Track</th>
<th>Output Track</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,4,1,3,5]</td>
<td>Holding tracks</td>
<td></td>
</tr>
<tr>
<td>[5,4,3,2,1]</td>
<td>Input track</td>
<td>Output track</td>
</tr>
</tbody>
</table>

Initially all the cars are on the input track. At the end they should be all on the output track in the correct order.

The strategy is to examine the cars on the input track from front to back. If the car examined is the next one to be placed on the output track, then it is immediately moved there. If not, then the car should be temporarily moved on a holding track. Each holding track behaves as a LIFO structure. When rearranging cars, the following operations are permitted:

- the front car in the input track can be moved to the output track or to the top of one holding track;
- a car can be moved from the top of a holding track to the output track

Consider the arrangement which has the cars in the input track in the order [5,8,1,7,4,2,9,6,3]. The first car, 3, cannot be moved to the output track, so it will be placed on the first holding track. Similarly, car 6 should be placed on hold. Since 3 will be moved to the output track earlier than 6, we cannot place 6 in the same holding track as 3 (otherwise 3 will be blocked). So 6 will be placed on the second holding track. Similarly, 9 should be placed in the third holding track. When we reach 2, since 2 is going to be moved to the output track before 3, we can place it in the same holding track as 3. In general, if we need to place a car \( k \) on a holding track, we will place it in the holding track whose top element is the smallest element greater than \( k \). If no holding track has a top element greater than \( k \), then a new holding track has to be used (if any is available—if not, then the rearrangement is impossible).

Each time a car is moved to the output track, we need to verify whether any car in the holding track can now be moved to the output track as well.

The solution makes use of
• the input is represented by an array which represents the input track;
• the output is a printout of the moves needed to transfer the cars to the output track in the correct order;
• each holding track is represented as a stack; since we have a fixed number of holding track, we will encode them as an array of stacks;

Let us just show the method employed to solve the problem (not the entire class).

```java
public static boolean arrangecars(int input[], int len, int holding)
// holding is the number of holding tracks
// len is the number of elements in the input track
{
    Stack[] hold; // array of stacks for holding tracks
    int next = 1; // keep track of next car to move to output
    int i;

    // create holding tracks
    hold = new Stack[i];
    for (i=0; i < holding; i++)
        hold[i] = new Stack();

    // rearrange cars
    for (i=0; i < len; i++)
    {
        if (next == input[i]) // can move car to output
        {
            System.out.println(''Move car ''+next+'' from input to output'');
            next++;

            // can we move cars from holding tracks?
            do
            {
                sel = selectholding(next,holding,hold);
                System.out.println(''Move car ''+next+'' from holding track ''+sel);
                if (sel != -1)
                    next++;
            } while (sel != -1);
        }
        else
            if (!putinholding(input[i],holding,hold,len))
                return false;
    }
    return true;
}
```

```java
public static int selectholding (int nextcar, int holding, Stack hold[])
{
    int i;

    for (i=0; i < holding; i++)
        if (hold[i].stackTop() == next) //is top element the next output car?
        {
```
hold[i].Pop();
return i;
}
return -1;
}

public static boolean putinholding(int car, int holding, Stack hold[], int len)
{
    int i;
    int min = len + 1;
    int minhold = -1;
    int free = -1;

    // find best holding track for car
    for (i = 0; i < holding; i++)
    {
        if (hold[i].emptyStack())
            free = i;
        else if (hold[i].stackTop() > car && hold[i].stackTop() < min)
        {
            min = hold[i].stackTop();
            minhold = i;
        }
    }
    if (free == -1 && minhold == -1)
        return false;
    else
    {
        hold[minhold].Push(car);
        System.out.println('Move car ' + car + ' to holding track ' + minhold);
        return true;
    }
}