Stack Implementations

Chapter 6

Pre-requisite: Chapters 3, 4, 5
Contents

• A Linked Implementation
• An Array-Based Implementation
Objectives

• Implement ADT stack by using either
  ▪ Linked chain
  ▪ Array

• Compare and contrast various implementations and their performance
Generic Class **Stack** (listing 5-1, pp. 116)

```java
public interface StackInterface < T >  {
    public void push ( T newEntry);        // Adds a new entry to the top of this stack.
    public T pop ();                        // Remove & return top entry if non-empty
    public T peek ();                       // Retrieves this stacks top entry if non-empty
    public boolean isEmpty ();             // Return true if this stack is empty
    public void clear ();                   // Removes all entries from this stack
}
```
# Abstract Data Type Stack

**DATE**

- A collection of objects in reverse chronological order and having the same data type

**Operations**

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>UML</th>
<th>Description</th>
</tr>
</thead>
</table>
| `push(newEntry)` | `+push(newEntry: T): void` | Task: Adds a new entry to the top of the stack.  
Input: `newEntry` is the new entry.  
Output: None. |
| `pop()` | `+pop(): T` | Task: Removes and returns the stack's top entry.  
Input: None.  
Output: Returns either the stack’s top entry or, if the stack is empty before the operation, `null`. |
| `peek()` | `+peek(): T` | Task: Retrieves the stack’s top entry without changing the stack in any way.  
Input: None.  
Output: Returns either the stack’s top entry or, if the stack is empty, `null`. |
| `isEmpty()` | `+isEmpty(): boolean` | Task: Detects whether the stack is empty.  
Input: None.  
Output: Returns true if the stack is empty. |
| `clear()` | `+clear(): void` | Task: Removes all entries from the stack.  
Input: None.  
Output: None. |
Exercises: Review of Chapter 5

1. Entries in a stack are "ordered". What is the meaning of this statement?
   (a.) A collection of stacks can be sorted.
   (b.) Stack entries may be compared with the '<' operation.
   (c.) The entries must be stored in a linked list.
   (d.) There is a first entry, a second entry, and so on.

2. Consider listing 5-2 (pp. 122-123) for testing balance in a sequence of parentheses. Determine maximum depth of stack for input of (((()()())(())))?
   (a.) 1    (b.) 2    (c.) 3    (d.) 4    (e.) 5

3. Consider listing 5-2 (pp. 122-123) for testing balance in a sequence of parentheses. Determine maximum depth of stack for input of with 4 left parentheses and 3 right parentheses (in some order):
   (a.) 1    (b.) 2    (c.) 3    (d.) 4    (e.) 5 or more

4. Which of the following applications may use a stack?
   (a.) A parentheses balancing program.  (b.) Syntax analyzer for a compiler.
   (c.) Keeping track of local variables of methods at run time.  (d.) All of the above.

5. Google Interview Question: How many traditional stacks are needed to implement a new data structure to support "push", "pop", and "find minimum" operations? "Find minimum" returns the smallest element in the stack with O(1) computation. Justify your answer by describing maintenance of minimum in push and pop operations.
   (a.) 1    (b.) 2    (c.) 3    (d.) 4
Recall the private class **Node**

- Figure 3-5 Two linked nodes that each reference object data

![Diagram of two linked nodes](image)

- Recursive class definition (**similar to Scheme list in Csci 1901**)

```java
class Node {
    private T data; // entry in bag
    private Node next; // link to next node

    // constructors
    ...

    // Accessor and mutator methods: getData, setData, getNextNode, setNextNode
    ...
}
```

Copyright ©2012 by Pearson Education, Inc. All rights reserved
private class Node {
    private T data;
    private Node next;
    private Node(T data portion) { this(dataPortion, null); }
    private Node(T dPart, Node nNode) { data = dPart; next = nNode; }
    private T getData() { return data; }
    private void setData(T newData) { data = newData; }
    private Node getNextNode() { return next; }
    private void setNextNode(Node nextNode) { next = nextNode; }
}

Exercise: Match methods in Node class with following Scheme functions:
(a.) car (b.) cdr (c.) cons (d.) set! (e.) set-car! (f.) set-cdr! (g.) list
Linked Implementation

• Design Decision 1: Where to place the top entry in a linked list of nodes?
  ▪ Choices: (a) First node, (b) Last node, (c.) some other place
  ▪ Compare choices using time-complexity for push(), pop() and peek()
    • Each involves top of stack
  ▪ Best to put top of stack at head node
    • Fastest, easiest to access

• Design Decision 2: Recycle popped data-item
  ▪ Java will manage reclaiming memory without instruction from programmer

• Note source code for linked implementation
  ▪ **Listing 6-1**
public T peek()
{
    T top = null;
    if (topNode != null)
        top = topNode.getData();

    return top;
} // end peek

**Ex.** Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 3 elements, i.e. Fig. 6-1

Figure 6-1 A chain of linked nodes that implements a stack
public void push(T newEntry)
{
    Node newNode = new Node(newEntry, topNode);
    topNode = newNode;
}
// end push

Ex. Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 3 elements, i.e. Fig. 6-1

FIGURE 6-2: push() operation
(a) A new node that references the node at the top of the stack;
(b) the new node is now at the top of the stack
Figure 6-3 The stack (a) before the first node in the chain is deleted
public T pop()
{
    T top = peek();
    if (topNode != null)
        topNode = topNode.getNextNode();
    return top;
} // end pop

Ex. Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 3 elements, i.e. Fig. 6-1

Figure 6-3 The stack (b) after the first node in the chain is deleted
Stack Using Linked-Nodes (listing 6-1, pp. 142)

```java
public class LinkedStack <T> implements StackInterface <T> {
    private Node topNode;  // first node in a chain
    public LinkedStack ()    { topNode = null; }   // constructor
    public void push( T newEntry)   {
        Node newNode = new Node(newEntry, topNode);  topNode = newNode;
    }
    public T peek() {
        T top = null;   if (topNode != null) top = topNode.getData();
    }
    public T pop() {
        T top = peek();  if (topNode != null) topNode = topNode.getNextNode();
        return top;
    }
    public boolean isEmpty() { return topNode == null; }
    public void clear()       { topNode = null ; }

    private class Node {
        private T data;       Node next;
        /* Constructors, getData, setData, getNextNode, and setNextNode */
    }
}
```
Array Based Implementation

- Decision 1: Were to place the top entry?
  (a) Lowest possible index, i.e., 0, (b) highest occupied index
  Compare option using time-complexity for push() and pop()
- Decision 2: Recycle popped data-item
Array Based Implementation: Decisions

- Decision 1:
  - More efficient operations with bottom of stack at beginning of array
  - Top of stack at last occupied entry
- Decision 2:
  - Consider memory wastage of unused array elements
  - Set array entry to null in pop()
public class ArrayStack <T> implements StackInterface <T> {
    private T[] stack;
    private int topIndex;
    private static final int CAPACITY = 50;
    public ArrayStack() {this (CAPACITY);}
    public ArrayStack(int Capacity)
    {
        // list = new T[initialCapacity]; is not legal in Java as explained on pp. 31
        @SuppressWarnings("unchecked") // Annotation for following statement
        T[]tempStack = (T[]) new Object[Capacity];
        stack = tempStack;
        topIndex = -1;
    }
    /* Implementations of the stack operations goes here */
}
Retrieving the Top

**Ex.** Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 4 elements, i.e. Fig. 6-4
(d) Full Stack

```java
public T peek()
{
    T top = null;
    if (!isEmpty())
        top = stack[topIndex];

    return top;
} // end peek
```

Figure 6-4 An array that implements a stack; its first location references (b) the bottom entry in the stack;
Ex. Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 4 elements, i.e. Fig. 6-4
(d) Full Stack

```java
public void push(T newEntry)
{
    ensureCapacity();
    topIndex++;
    stack[topIndex] = newEntry;
} // end push

private void ensureCapacity()
{
    if (topIndex == stack.length - 1) // if array is full,
        // double size of array
        stack = Arrays.copyOf(stack, 2 * stack.length);
} // end ensureCapacity
```
Ex. Test the code for following cases:
(a) Empty stack
(b) Stack with 1 element
(c) Stack with 4 elements, i.e. Fig. 6-4
(d) Full Stack
Implement Stack using an array (listing 6-2, pp. 146)

```java
public class ArrayStack<T> implements StackInterface<T> {
    private T[] stack;    private int topIndex;
    private static final int CAPACITY = 50;
    public ArrayStack() { this (CAPACITY); }
    public ArrayStack(int Capacity) {
        @ SuppressWarnings("unchecked") T[] tempStack= (T[]) new Object[Capacity];
        stack = tempStack;    topIndex = -1;
    }
    public void push(T newEntry) {
        ensureCapacity();    topIndex++;
        stack[topIndex] = newEntry;
    }
    public T peek() {
        T top = null;    if (!isEmpty()) top = stack[topIndex];
    }
    public T pop() {
        T top = null;    if (!isEmpty()) { top = stack[topIndex];
        stack[topIndex] = null;    topIndex--;    return top;
    }
    public boolean isEmpty() {    return topIndex < 0 ; }
    public void clear() {    topIndex = -1 ; }
    private ensureCapacity() {
        if (topIndex == stack.length – 1) stack = Arrays.copyOf(stack, 2 * stack.length);
    }
}
```
Exercises: Stack implementation

1. An array implementation of stack class (capacity 100) has 10 items stored at data[0] through data[9]. Where does the push place the new entry?
   (a.) data[0] (b.) data[1] (c.) data[9] (d.) data[10]

2. In linked-node implementation of stack, where is new entry added by push()?
   (a.) At the head (b.) At the tail (c.) At a place to maintain sorted order

3. Which operations have linear-time worst case in the linked-node implementation of stack?
   (a.) push (b.) pop (c.) peek (d.) is_empty (e.) No operation requires linear time.

4. Which operations have linear-time worst case in the array-based implementation of stack?
   (a.) push (b.) pop (c.) peek (d.) is_empty (e.) No operation requires linear time.

5. Consider an alternate array-based implementation, where top is index 0 and bottom is at last used index. Which operations will require linear time?
   (a.) push (b.) pop (c.) peek (d.) is_empty (e.) No operation requires linear time

6. Write a "for" loop for array-based clear() to facilitate memory recycling (i.e., garbage collection).
End

Chapter 6