Stacks and Linked Data Structures

The final two topics we will discuss in this module are reviews of stacks and linked data structures.

A. Stacks

Stacks are fundamental components in solving problems using computers. All modern programming languages use a call stack to store the local state of a method or function. Many modern computing systems also use stacks (as opposed to other instruction architectures that use mechanisms such as registers) to implement any calculations performed on the computer. Two examples of stack-based architectures in wide use today are the Java Virtual Machine (JVM) and the .Net Common Language Runtime (CLR).

This section will first present the implementation of a stack. Stacks will then be used to solve two problems: the first will address the parentheses matching problem, and the second will solve an infix equation with precedence rules and parentheses.

i. Implementation of a Stack

A stack, like a queue, can have multiple implementations. Therefore, it is best to define an interface as an ADT for a stack, which defines the operations that can be performed on an abstract stack. The following interface defines the ADT for a stack:

```
public interface Stack<E> {
  public boolean isFull();
  public boolean isEmpty();
  public E peek() throws StackException;
  public void push(E item) throws StackException;
  public E pop() throws StackException;
}
```

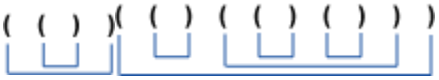
Finally, a stack is defined that implements this interface. In this case, the stack is implemented as a fixed-length array. The following class implements the stack in a fixed-length array:

```
public class ArrayStack<E> implements Stack<E> {
  private static int DEFAULT_SIZE = 10;
  E elements[];
  int numberOfElements = 0;

  public ArrayStack() {
    this(DEFAULT_SIZE);
  }

  @SuppressWarnings({"unchecked", "deprecated"})
  public ArrayStack(int size) {
    elements = (E[])(new Object[size]);
  }

  public boolean isFull() {
```

```
    if (numberOfElements == elements.length)
      return true;
    else
      return false;
  }

  public boolean isEmpty() {
    if (numberOfElements == 0)
      return true;
    else
      return false;
  }

  public E peek() throws StackException {
    if (numberOfElements == 0)
      throw new StackException("Stack is Empty");
    return elements[numberOfElements - 1];
  }

  public void push(E item) throws StackException {
    if (numberOfElements == elements.length)
      throw new StackException("Stack is Full");
    elements[numberOfElements] = item;
    numberOfElements++;
  }

  public E pop() throws StackException {
    if (numberOfElements == 0)
      throw new StackException("Stack is Empty");
    E elementToReturn = elements[numberOfElements - 1];
    numberOfElements--;
    return elementToReturn;
  }
}
```

ii. Parentheses Matching Program

The first problem we'll look at that uses a stack is a parentheses matching program. The purpose of this program is to determine whether or not a string of parentheses is valid. The string is valid if the parentheses are nested to exactly match each other. If the parentheses do not match, the string is invalid. For example, the following strings are valid:



An example of an invalid string is the following:

To solve this problem, it is insufficient to count the number of left and right parentheses to see if they are equal, as that would mean the problem above would be seen as valid. A mechanism has to be developed that allows them to be matched. This can be done easily with a stack. A pseudocode algorithm for this is presented below:

```
while(moreSymbols) {
  if symbol is "("
    stack.push(symbol)
  else  -- We know the symbol is a ")"
    if stack.pop() != "("
      Error in parentheses string
}
if any symbol still in stack
  Error in parentheses string
```

To see how this algorithm would work, the following diagram shows the stack as each symbol of the string (( ) (( ) ( ))) is processed. The top row of the table shows the input symbol that is to be processed in each step, and below that is the stack to be maintained. When a "(" symbol is processed, the "(" is pushed on the stack. When a ")" symbol is processed, the corresponding "(" symbol, which must be on the top of the stack, is popped. If a ")" is ever encountered and the symbol at the top of the stack is not a "(" (or if the stack is empty) the string is invalid. Also, if after the entire string is processed there are any values left on the stack, the string is invalid. Otherwise, the string is valid.

Symbol	(	(	)	(	(	)	)	(	)	)
Stack	((	((	((	((	(((	((	((	((	((	((

Figure 1.VI.1 – Stack for the processing of a parenthesis string

ii. Infix Notation Parsing Program

This problem will calculate a string containing an infix expression. There are a few rules about this string and how to processes it. First, the string must be well formed, meaning that it must not have any errors. This program will not account for errors. Second, the * and / operators have precedence over the + and – operators and are thus evaluated before the + and – operators. The * and / have the same precedence, as do the + and – operators, and operators with the same precedence are evaluated left to right. Finally, the string can have parentheses, but it does not have to be fully parenthesized. The operations inside the parentheses take precedence over operators outside the parentheses.

In order to solve this problem, stacks must be used. The algorithm to solve this problem is given in pseudocode below. Note that this calculator uses two stacks to do the calculations. The first stack is an operator stack, which will contain the +, –, *, and / operators, as well as left and right

parentheses. The second stack is an operand stack, which will contain any numbers that are input to the program, as well as the results of calculations.

Here is the algorithm needed to implement this solution:

The problem as described here could also be solved using recursion, but because it does not require the input string to be fully parenthesized, a more complex grammar, and thus a more complex use of recursion, would be needed. The study of these grammars and how to implement them is a topic for a course in programming languages or compiler theory, and so it will not be covered here.

```
while(moreTokens) {
  if token is not an operator(in other words, it is a number)
    operandStack.push(token)
  else if token is "("
    operatorStack.push(symbol) // This creates a false
                               // bottom to the stack.
                               // It must be matched by
                               // a ")"
  else if token is ")"
    // Process all the operators until a matching "("
    // is found.
    while (op = operatorStack.pop() is not "(") {
      value2 = operandStack.pop()  // Note pop the values off
      value1 = operandStack.pop()  // in reverse order. This
                                   // accounts for the fact
                                   // subtraction and division
                                   // are not commutative.

      newValue = value1 op value2  // Perform the popped
                                   // operator on these two
                                   // values
      operandStack.push(newValue)  // store the value back
                                   // on the stack.
    }

    else                           // token is +, -, *, /
      // Check the precedence of the previous operator.
      // If the stack is empty, or the operator is a "("
      // or the precedence of the previous operator is higher
      // then the current operator is pushed and the inner loop
      // is exited.
      while ( true ) {
        if ( 1 - the stack is empty OR
             2 - the top of the stack is a "(" OR
             3 - the precedence of the operator at the top
                 of the stack is less than this operator's
                 precedence ) {
          operatorStack.push(token)
          leave the inner while loop
        }
        // Process the operator as earlier.
        op = operatorStack.pop()
        value2 = operandStack.pop()
```

```
            value1 = operandStack.pop()

            newValue = value1 op value2
            operandStack.push(newValue)
        }
}
```

The value at the top of the operandStack is the answer

The use of this algorithm to solve the expression (7 * (2 + 5 + 3 * 2 – 1)) is illustrated in the following step diagram, which shows the stack as it is built through each step. This is the way the stack will be represented in the problems at the end of this chapter, and how it should be illustrated if a problem such as this is presented on an exam.

B. Linked Data Structures - Cursors

Linked data structures are often associated with linked lists. A concept called an array-based linked list, or cursor, will be presented.

One major problem with linked lists is that they require memory allocation and deallocation for each member in the list. Allocating and deallocating memory is an expensive operation that can have an impact even on high-level programs such as those written in Java.

A cursor removes the need to allocate and deallocate the memory for each member in the list. The items in the list are stored in an array, and array indices—which allow list elements to point to other array elements—are maintained with each element. This is a very efficient implementation of a linked data structure, since the cursor's array needs to be allocated only once, and there is no need to deallocate it and recover the memory.

The limitation of a cursor is that since the elements are stored in an array, the data must be fixed in size. This limit is severely constraining in Java, but not for general problems. In fact, as will be shown in the module on hash tables, cursors are so efficient that they have been used as part of the implementation of a hardware cache, an operation that must be run billions of times per second. This would not be possible with linked lists, in which each allocation and deallocation is many orders of magnitude greater in execution time. This implementation is possible with cursors because what is stored in the cache is fixed sized pages, and the number of pages in cache must correspond to a hardware memory constraint, so the size of each member of the cursor and the overall size of the cursor are known and can be allocated during the boot phase of the operating system.

To operate, each element in a cursor must contain the data to be stored, as well as a pointer (an array index) to the next item in the list. In addition, a cursor must maintain at least two separate lists, though it could contain many more, as is often the case. If only two lists exist in the cursor, the first is a list of the array members that are not yet used (called the free list), and the second is a list of the elements stored in the cursor. To implement the item list and the free list, all that needs to be stored in the cursor is the head of each list.

The following figure is a graphical representation of a nine-element cursor that contains the list of integers (3, 6, 7, 9). It is assumed that this cursor contains sorted data. This is not a requirement of a cursor, which, like a linked list, can contain sorted or unsorted data.

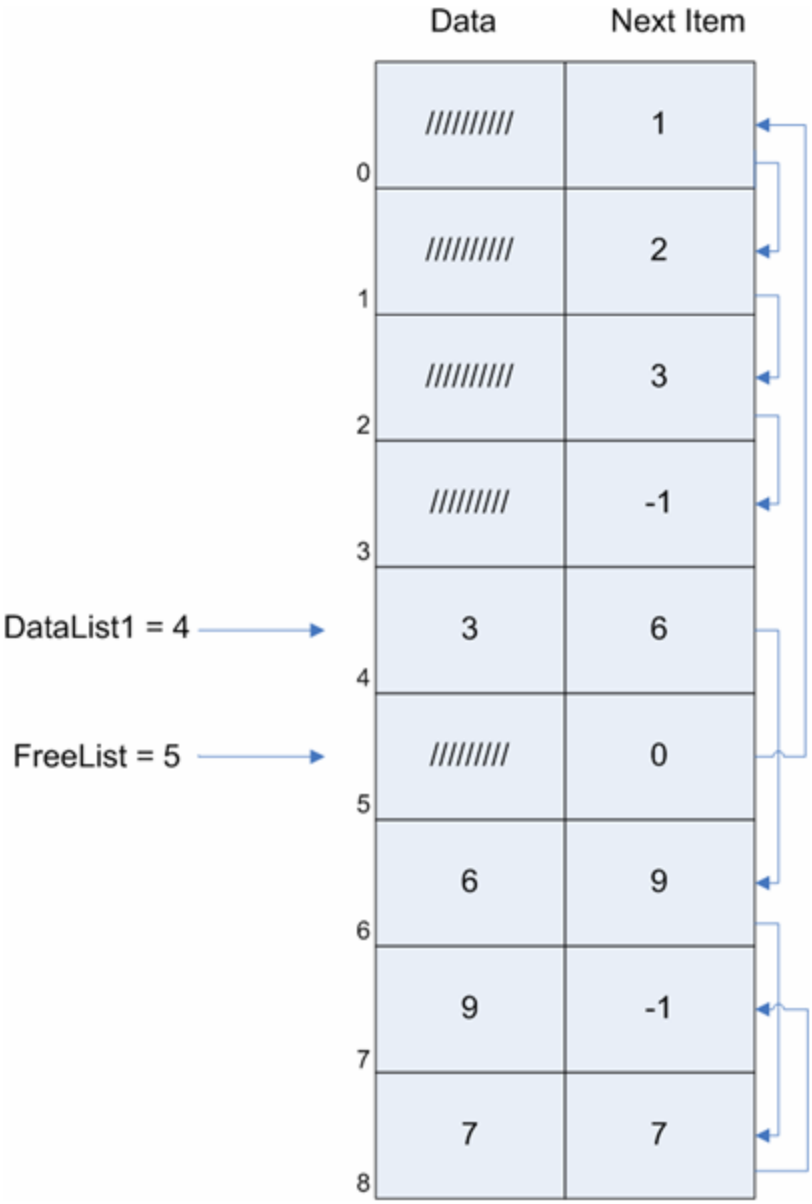


Figure 1.VI.2 – Cursor with integer list

The steps involved in adding the element 4 to this sorted list of integers stored in the cursor are given in the following step diagram.

The steps involved in removing the element 7 from this sorted list of integers stored in the cursor are given in the following step diagram.

These steps are exactly analogous to inserting and deleting an item from a linked list, with the only difference being that an array index, rather than a pointer to an object, is changed.

As was mentioned previously, there is no reason that a cursor should be limited to one list of elements, as long as all elements on both lists are of the same type. All that is needed is an extra variable that contains the array index for the start of the second list. The following graphical representation of a cursor contains two lists of integers, the first containing the list (3, 6, 7, 9), and the second containing the list (4, 8).

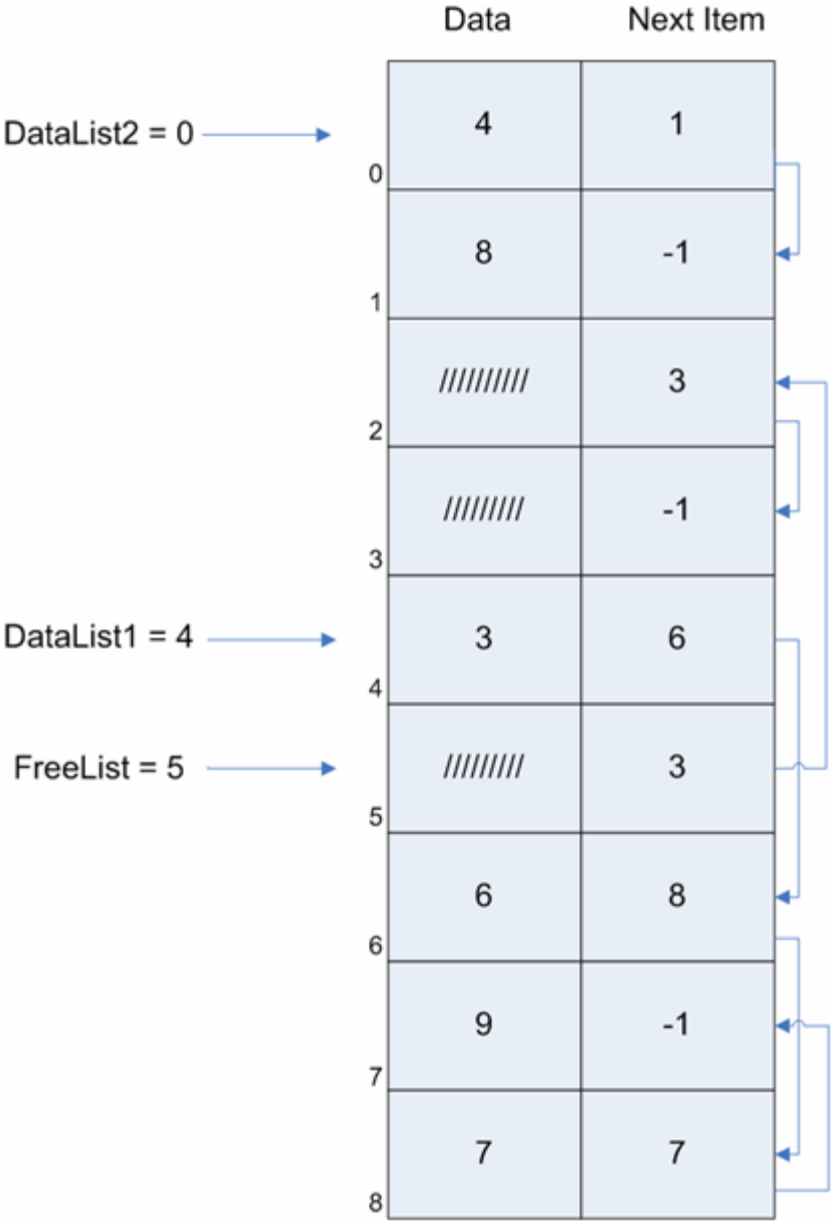


Figure 1.VI.3 – Cursor with two integer lists

This module has presented the basic math background and introductory programming and data structures you will need for the rest of this course. The rest of the modules we'll cover will build on the mathematics and programming presented here to implement new data structures that are increasingly advanced and useful.