Chapter 13 Data Structures

13.1 A Data Structure is…

A *data structure* is an object that keeps data in a readily usable way. Generally, the object has methods that allow convenient additions to the data, deletions from the data and inspections of the data. The basic idea is to hide details of how the data is managed from other objects in the program while allowing convenient access to the data. In this chapter, we will discuss a few very common data structures.

If the amount of data to be stored in a data structure is large, the speed of the addition, deletion and inspection methods are as important as the access itself. The speeds of the access methods are closely related to the internal structure of the data. It is very common to hear people discuss data structures by naming the internal structure of the data. Usually, what they care about is the speed of response when requests are made. Thus it might be appropriate to request a list in which individual items can be inspected very rapidly, but for which printing out a sorted list of all the data need not be particularly fast or efficient. This is usually better than requesting a hash table, which is just one example of an internal structure that has these properties.

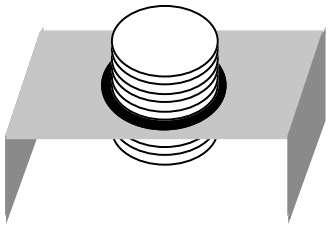
13.2 Stacks

The *stack* is usually the first data structure mentioned in any discussion of data structures. There is good reason for this. A *stack* is one of the simplest of data structures. It is also very widely used.

A *stack* is defined by its *access* methods. These three methods are listed below with their standard names.

1. *push* Put an item into the stack, returns no value.
2. *pop* Remove the last item put into the stack, returns the item removed.
3. *empty* Inquire as to whether or not any data remains in the stack, returns a *boolean* answer.

The names *push* and *pop* more or less come from the image of a stack as a pile of cafeteria plates partially hidden in a round hole in the counter. (Beneath the pile is a spring pushing the pile up as plates are removed).



A stack is often called a *last in first out* or *LIFO* structure.

Using a Stack: a bracket checking method

The following method checks to see if the brackets (braces, parentheses etc.) in a string are nested and matched correctly. For example the string "[[ ] (< >) { }]" has correctly matched and nested pairs, whereas the string "[{] }" has matched pairs but they are not nested correctly.

The idea of the method is to save each left-hand bracket and then when a right-hand bracket is encountered remove the last left-hand bracket saved and make sure that it matches. To make sure that there are no unmatched brackets, the stack must be empty at the end.

```
public boolean bracketsMatch(String s)
    {
    StackOfCharacters stack = new StackOfCharacters();
    for (int i=0; i<s.length(); i++)
        {
        char ch = s.charAt(i);
        switch (ch)
            {
            case '(': case '[': case '{': case '<': stack.push(ch); break;
            case ')': if (stack.empty() || stack.pop() != '(') return false;
                      break;
            case ']': if (stack.empty() || stack.pop() != '[') return false;
                      break;
            case '}': if (stack.empty() || stack.pop() != '{') return false;
                      break;
            case '>': if (stack.empty() || stack.pop() != '<') return false;
                      break;
            default: ;
            }
        }
    return stack.empty();
    }
```

Implementing a Stack - an Array

To use the *bracketsMatch* method above, the class *StackOfCharacters* must also exist. There is a stack class provided with Java, but we will write a couple of our own before using it. This first stack uses an array. The advantages of an array are that it is easy to write and fast running. The disadvantage of an array is that we must decide beforehand how many items it may have to hold.

```
class StackOfCharacters
    {
    public static final int SIZE = 1000;

    private char data[];
    private int howMany;

    public StackOfCharacters()
        {
        data = new char[SIZE];
        howMany = 0;
        }

    public boolean empty()
        {
        return howMany == 0;
        }

    public void push(char ch)
        {
        if (howMany < SIZE)
            {
            data[howMany] = ch;
            howMany++;
            }
        }

    public char pop()
        {
        if (howMany == 0)
            return -9999;
        howMany--;
        return data[howMany];
        }
    }
```

Implementing a Stack - Linked Nodes

This stack uses linked nodes. The advantage is that we do not have to know beforehand the maximum number of elements that might be put onto the stack. Compared to an array implementation the disadvantages are that each element takes up more space in the machine and the methods are not as fast.

```
class StackOfCharacters
   {
   private Node headOfStack;
   public StackOfCharacters()
      {
      headOfStack = null;
      }

   public boolean empty()
      {
      return headOfStack == null;
      }

   public void push(char ch)
      {
      headOfStack = new Node(ch, headOfStack);
      }

   public char pop()
      {
      if (headOfStack == null)
         return 0;
      char ch = headOfStack.getData();
      headOfStack = headOfStack.getLink();
      return ch;
      }
   }
```

```
class Node
   {
   private char data;
   private Node link;

   public Node(char ch, Node n)
      {
      data = ch;
      link = n;
      }

   public char getData()
      {
      return data;
      }

   public Node getLink()
      {
      return link;
      }
   }
```

Using the class *java.util.Stack*

The package of utilities, *java.util* contains a stack class. This class is designed to allow placing any sort of object references onto the stack. Beginning with Java version 1.5 this stack class can be used in a very straightforward way.

In general, a stack containing object references of any type can be declared by 'importing' *java.util* and placing the type desired between '*pointy brackets*' after the class name *Stack*:

```
import java.util.*;

   Stack<Color> cStack = new Stack<Color>();          // a stack of Colors
   Stack<String> sStack = new Stack<String>();        // a stack of Strings
```

The scalar types each have an associated 'wrapper' class. These include *int* → *Integer*, *double* → *Double*, *char* → *Character*, *boolean* → *Boolean*. Only the types *int* and *char*, which are obvious abbreviations, have associated wrapper class names that are not simply the scalar type with the first letter capitalized.

To create a version of the method *bracketsMatch* using the class *java.util.Stack,* place the *import* line above at the beginning of the java file and declare the stack as shown:

```
public boolean bracketsMatch(String s)
   {
   Stack<Character> stack = new Stack<Character>();
   for (int i=0; i<s.length(); i++)
   ...
   }
```

13.3 Queues

The word *queue* is little used in the United States. It is the common term in Great Britain for a waiting line (grocery store, movie theater etc.). Queue is pronounced like the letter 'q'. A queue is very much like a stack. The difference is that the method that deletes an item, deletes (and returns) the item that has been in the queue longest (rather than the item that has been stored for the shortest time). A queue is sometimes known as a *first in first out* or *FIFO* structure.

Implementing a Queue — Linked Nodes

A linked implementation of a queue of *Objects* is shown first, as it is the most straight forward. An efficient implementation requires that the *Queue* class be able to quickly locate either end, hence the two object references.

```
class Queue// Linked nodes implementation
   {
   private Node headOfQueue, tailOfQueue;

   public Queue()
      {
      headOfQueue = null;
      tailOfQueue = null;
      }

   public boolean empty()
      {
      return headOfQueue == null;
      }

   public void putAtEnd(Object obj)
      {
      if (headOfQueue == null)
         {
         headOfQueue = new Node(obj);
         tailOfQueue = headOfQueue;
         }
      else
         {
         Node n = new Node(obj);
         tailOfQueue.append(n);
         tailOfQueue = n;
         }
      }

   public Object takeFromFront()
      {
      if (headOfQueue == null)
         return null;
      Object obj = headOfQueue.getData();
      if (headOfQueue != tailOfQueue)
         headOfQueue = headOfQueue.getLink();
      else
         {
         headOfQueue = null;
         tailOfQueue = null;
         }
      return obj;
      }
   }
```

```
class Node
   {
   private Object data;
   private Node link;

   public Node(Object obj)
      {
      data = obj;
      link = null;
      }

   public void append(Node n)
      {
      link = n;
      }

   public Object getData()
      {
      return data;
      }

   public Node getLink()
      {
      return link;
      }
   }
```

Implementing a queue — an Array

The *takeFromFront* operation could be implemented by moving every element in the list up one space, but if the list is very long, this is terribly inefficient. An efficient implementation requires that the items not be moved. This is usually done with an implementation called a *circular queue*. The idea is to keep track of both the first and the last elements in the array. When the position of either of these passes the end of the array, it 'wraps around' to the beginning.

```
class Queue                 // Array implementation – Circular Queue
   {
   private static final int MAX = 1000;
   private int head, afterTail;
   private Object data[];

   public Queue()
      {
      head = afterTail = 0;
      data = new Object[MAX];
      }

   private int next(int previous)
      {
      return (previous+1) % MAX;
      }

   public boolean empty()
      {
      return head == afterTail;
      }

   public boolean full()
      {
      return head == next(afterTail);
      }

   public void putAtEnd(Object obj)
      {
      if (full())
         return;
      data[afterTail] = obj;
      afterTail = next(afterTail);
      }

   public Object takeFromFront()
      {
      if (empty())
         return null;
      Object obj = data[head];
      head = next(head);
      return obj;
      }
}
```

13.4 Fancier queues

There are several additional types of queue-like data structures, two of which are *double-ended queues* and *priority queues*.

Double ended queues

A *double ended queue* has two methods that add data, *putAtEnd* and *putAtFront*. It also has two methods that delete data, *takeFromEnd* and *takeFromFront*. A double ended queue can be used as a stack (use one pair: *putAtEnd* and *takeFromEnd* or *putAtFront* and *takeFromFront*). It can also be used as a queue (use *putAtEnd* and *takeFromFront* or *putAtFront* and *takeFromEnd*). Double ended queues are sometimes used as stacks, sometimes as queues, but only occasionally as double ended queues.

Priority queues

A *priority queue* keeps track of data objects with a priority (usually an integer) attached to each object. The *takeFromFront* method removes the item with the highest priority. If there are very many items in a priority queue, keeping track of the item with the highest priority with the least expenditure of computer time is an interesting problem. The most common implementation of a priority queue uses a data structure called a *heap*. Note: there are two, unrelated data structures called heaps. The appropriate one here is a binary tree, the other is not.