Abstract Data Types and Data Structures

Often, these terms are used as synonyms. But it’s better to think of them this way:

- An Abstract Data Type (ADT) represents a particular set of behaviours.
  - You can formally define (i.e., using mathematical logic) what an ADT is/does.
  - e.g., a Stack is a list implements a LIFO policy on additions/deletions.
- A data structure is more concrete. Typically, it is a technique or strategy for implementing an ADT.
  - Use a linked list or an array to implement a stack class.
- Going one level lower, we get into particulars of programming languages and libraries
  - Use java.lang.vector or java.util.stack or a C++ library from STL.

ADTs and Data Structures

- Some common ADTs that all trained programmers know about:
  - stack, queue, priority queue, dictionary, sequence, set
- Some common data structures used to implement those ADTS:
  - array, linked list, hash table (open, closed, circular hashing)
  - trees (binary search trees, heaps, AVL trees, 2-3 trees, tries, red/black trees, B-trees)

We’ll discuss these in some detail.

Revisiting the Stack

Recall our friend the stack, as implemented by an array.

```java
class stack {
    private int maxSize, top = -1;
    public static final int DefaultMaxSize = 100;
    private Object[] store;

    public stack () {
        maxSize = DefaultMaxSize;
        store = new Object[maxSize];
    }

    public stack (int desiredMaxSize) {
        maxSize = desiredMaxSize;
        store = new Object[maxSize];
    }

    public void push (Object newVal) {
        if (top < maxSize -1) {
            top++;
            store[top]=newVal;
        } else {
            System.err.println ("Sorry, stack is full.");
        }
    }
}
```

What’s Wrong with this Picture?

There’s something unsatisfying about this implementation.

- We have to state, in advance, exactly how many elements we will need.
  [Or just use the provided default of 100]
  - If we go over that number of elements at any one point in time, the stack stops working.
  - If we use a big desiredMaxSize, then we waste space most of the time.
    [What if we need several stacks?]
- This approach uses a static approach to resource allocation:
  - We have to decide before creating the object what its size will be.
  - We are not allowed to change the size during its lifetime.
- Yes, all resources are ultimately finite. But we would like a little more flexibility and reasonable use of resources.
A Dynamic Approach to Resource Allocation

- What if we could design a stack that used exactly “enough” storage at any given time?
  → Sure we will run out of space eventually if we keep adding elements, but only if we really have to, i.e., only if the stack gets really huge.
  → What are the tradeoffs?
    Are there any disadvantages?
- Dynamically allocated structures can grow and shrink as needed throughout their lifetime.
- A common yin-yang in CS in static vs. dynamic allocation.

A Linked List Approach

- The stack is a set of nodes linked together.
  → Each node has a “value” plus a link variable (a reference to another node).
  → The “value” could be a simple value (e.g., an int), a set of values, or a reference to another object (e.g., an employee record).
- When you want to push a new value:
  1. Create a new node via `new`.
  2. Set the value of the new node.
  3. Set its link field to point to the previous top node.
  4. Reset top to point to the new node.

```java
public interface Stack {
    public abstract void push (Object element);
    public abstract Object pop ();
    public abstract boolean isEmpty ();
    public abstract int size ();
}

class Node {
    // Use package-level visibility.
    Object value;
    Node next;
    public Node (Object value, Node next) {
        this.next = next;
        this.value = value;
    }
}

class LinkedStack implements Stack {
    private int numElements;
    private Node first;
    public LinkedStack () {
        numElements = 0;
        first = null;
    }
    public void push (Object element) {
        Node newNode = new Node (element, first);
        first = newNode;
        numElements++;
    }
    public Object pop () {
        Object returnVal;
        if (numElements > 0) {
            returnVal = first.value;
            first = first.next;
            numElements--;
        } else {
            System.out.println ("Sorry, stack is empty.");
            returnVal = null;
        }
        return returnVal;
    }
    public boolean isEmpty () {
        return numElements == 0;
    }
    public int size () {
        return numElements;
    }
}
```
Still inside LinkedStack ...

public static void main (String [] args) {
    LinkedStack s1 = new LinkedStack();
    s1.push ("hello");
    s1.push ("there");
    s1.push ("world");
    System.out.println ("There are now " + s1.size() + " elements.");
    String s;
    s = (String) s1.pop();
    System.out.println ("Popped: " + s);
    s = (String) s1.pop();
    System.out.println ("Popped: " + s);
    s = (String) s1.pop();
    System.out.println ("Popped: " + s);
    s = (String) s1.pop();
    System.out.println ("Should be empty now.");
}

For completeness, here is the array implementation restated slightly as implementing the Stack interface:

```java
public class ArrayStack implements Stack {
    private int maxSize;
    public static final int DefaultMaxSize = 100;
    private int top = -1;
    private Object [] store;

    public ArrayStack () {
        maxSize = DefaultMaxSize;
        store = new Object [maxSize];
    }

    public ArrayStack (int desiredMaxSize) {
        maxSize = desiredMaxSize;
        store = new Object [maxSize];
    }

    public void push (Object newVal) {
        if (top < maxSize -1) {
            top++;
            store[top]=newVal;
        } else {
            System.err.println ("Sorry, stack is full.");
        }
    }

    public Object pop () {
        Object ans = null;
        if (top >= 0) {
            ans = store[top];
            top--;
        } else {
            System.err.println ("Sorry, stack is empty.");
        }
        return ans;
    }

    public boolean isEmpty () {
        return size() == 0;
    }

    public int size () {
        return top + 1;
    }
}
```

Array vs. Linked List:
Advantages and Disadvantages

- 
- 
- 
- 
-
Another Strategy: Split the Difference

- Keep a linked list of arrays (NodeChunks):
  - Each NodeChunk has a reference to the one before it.
  - Keep a reference to the “top chunk”, plus a top integer that points to the top element in the top chunk.
- Must decide on a good chunk size somehow.
  - Constructor arg and/or use a default size.

“Splitting the Difference”: Advantages and Disadvantages

- 
- 
- 
- 
- 
- 

Another ADT: The Queue

- A queue is a list that implements a FIFO (First In is the First Out) policy on insertions and deletions.
  - Can add elements only to the end of the list.
  - Can remove elements only from the front of the list.
- “Add” is called enter, enqueue, or add
- “Delete” is called leave, dequeue, or remove
- Misc. extra methods: isEmpty, size

public interface Queue {
    public abstract void enter (Object element);
    public abstract Object leave ();
    public abstract boolean isEmpty ();
    public abstract int size ();
}

Implementing a Queue with an Array

An ArrayQueue is trickier than an ArrayStack:
  - There are two pointers (int indexes into array) to keep track of.
  - When you run out of room, start over at the beginning.
  - Of course, this static approach means you are limited to at most N elements in your queue at any given moment.
Implementing a Queue with an Array

Think of the array as a circle.

→ Use modular arithmetic to stay in the valid range of indexes. To increment last on an enter:

\[
\text{last = (last + 1) \mod N;}
\]

[Assumes there is space, i.e., element 0 is empty]

→ Items in the range first ... last are full.

→ Items in the range last+1 ... first-1 are empty.

class ArrayQueue implements Queue {
    private int first=0, last=-1;
    private final int DefaultMaxSize = 100;
    private int maxSize, numElements = 0;
    private Object [] store;

    public ArrayQueue () {
        this.maxSize = DefaultMaxSize;
        store = new Object[maxSize];
    }

    public ArrayQueue (int maxSize) {
        this.maxSize = maxSize;
        store = new Object[maxSize];
    }

    public void enter (Object newElt) {
        if (size() < maxSize) {
            last = (last + 1) % maxSize;
            store[last] = newElt;
            numElements++;
        } else {
            System.err.println ("Sorry, queue is full.");
        }
    }

    public Object leave () {
        if (size() > 0) {
            Object ans = store[first];
            first = (first + 1) % maxSize;
            numElements--;
            return ans;
        } else {
            System.err.println ("Sorry, queue is empty.");
            return null;
        }
    }

    public void print () {
        System.out.println ("Printing the queue: ");
        int j = first;
        for (int i=0; i< size(); i++) {
            System.out.println (store[j]);
            j = (j + 1) % maxSize;
        }
    }

    public int size () {
        return numElements;
    }

    public boolean isEmpty () {
        return size() == 0;
    }
}

class ArrayQueue main (String[] args) {
    ArrayQueue q1 = new ArrayQueue(5);
    q1.enter ("do");
    q1.enter ("re");
    q1.enter ("mi");
    q1.enter ("fa");
    q1.enter ("so");
    q1.print();
    System.out.println ("Queue should be full now.");
    q1.enter ("la");
    System.out.println ("Removed " + q1.leave());
    System.out.println ("Removed " + q1.leave());
    System.out.println ("Removed " + q1.leave());
    System.out.println ("Removed " + q1.leave());
    q1.print();
    q1.enter ("la");
    q1.enter ("ti");
    q1.print();
    q1.enter ("doh");
    q1.enter ("ray");
    q1.enter ("me");
    q1.print();
    q1.enter ("far");
    q1.enter ("sew");
    q1.print();
    System.out.println ("Removed " + q1.leave());
}
Implementing a Queue as a Linked List

- Pretty straightforward to implement.
- Only caveat: watch out on enter
  - If queue empty, adjust first to point to new node.
  - Otherwise reset last.next to new node.
- Can also implement using a NodeChunk
  [Each NodeChunk contains an array of values.]

```java
public class LinkedQueue implements Queue {
    private int numElements = 0;
    private Node first = null, last = null;

    public void enter (Object element) {
        Node newNode = new Node (element, null);
        if (numElements == 0) {
            first = newNode;
        } else {
            last.next = newNode;
        }
        last = newNode;
        numElements++;
    }

    public Object leave () {
        Object returnVal;
        if (numElements > 0) {
            returnVal = first.value;
            first = first.next;
            numElements--;
        } else {
            System.out.println ("Sorry, queue is empty.");
            returnVal = null;
        }
        return returnVal;
    }

    // Obvious definitions for isEmpty and size omitted.
}
```

Misc. Notes

- We have decided to declare the Node fields value and next as package-level visible.
  - This means that other classes in the same package, such as LinkedQueue, can directly manipulate the instance variables of each Node instance.
- Also, we can use more interesting objects than Strings and Integers in our stacks and queues.
  - Could have a list of Figures or EmployeeRecords

This program:

```java
class FigureQueueTest {
    public static void main (String[] args) {
        Queue q = new LinkedQueue();
        Circle c1, c2;
        c1 = new Circle();
        c1.setSize(50);
        c1.setLoc(25,30);
        c2 = new Circle();
        Square s1 = new Square();
        s1.setSize (75);
        q.enter(c1);
        q.enter(c2);
        q.enter(s1);
        while (!q.isEmpty()) {
            Figure f = (Figure) q.leave();
            f.draw();
        }
    }
}
```

Produces this output:

Circle at x = 25 y = 30 with radius = 50
Circle at x = 0 y = 0 with radius = 0
Square at x = 0 y = 0 with width = 75 and height = 75