Stacks and Queues

A linear data structure is one which is ordered. There are two special types with restricted access: a stack and a queue.

11.1 Stacks Basics

A stack is a data structure of ordered items such that items can be inserted and removed only at one end (called the top). It is also called a LIFO structure: last-in, first-out.

The standard (and usually only) modification operations are:

- push: add the element to the top of the stack
- pop: remove the top element from the stack and return it

If the stack is empty and one tries to remove an element, this is called underflow.

Another common operation is called peek: this returns a reference to the top element on the stack (leaving the stack unchanged).

A simple stack algorithm could be used to reverse a word: push all the characters on the stack, then pop from the stack until it’s empty.

```
   this → s
   i    → s i h t
   h
   t
```

11.2 Implementation

A stack is commonly and easily implemented using either an array or a linked list. In the latter case, the head points to the top of the stack: so addition/removal (push/pop) occurs at the head of the linked list.

11.3 Application: Balanced Brackets

A common application of stacks is the parsing and evaluation of arithmetic expressions. Indeed, compilers use a stack in parsing (checking the syntax of) programs.

Consider just the problem of checking the brackets/parentheses in an expression. Say \([(3+4)*(5–7)]/(8/4)\). The brackets here are okay: for each left bracket there is a matching right bracket. Actually, they match in a specific way: two pairs of matched brackets must either nest or be disjoint. You can have \([()]\) or \([[]]\), but not \([()]\).

We can use a stack to store the unmatched brackets. The algorithm is as follows:
Scan the string from left to right, and for each char:
1. If a left bracket, push onto stack
2. If a right bracket, pop bracket from stack
   (if not match or stack empty then fail)
At end of string, if stack empty and always matched, then accept.

For example, suppose the input is: ([])][() Then the stack goes:

```
( → ] → ( → → ) → (
```

and then a bracket mismatch occurs.

11.4 Application: Evaluating Arithmetic Expressions

Consider the problem of evaluating the expression: (((3+8)-5)*(8/4)). We assume for this that the brackets are compulsory: for each operation there is a surrounding bracket. If we do the evaluation by hand, we could:

repeatedly evaluate the first closing bracket and substitute

(((3+8)-5)*(8/4)) → ((11-5)*(8/4)) → (6*(8/4)) → (6*2) → 12

With two stacks, we can evaluate each subexpression when we reach the closing bracket:

Algorithm (assuming brackets are correct!) is as follows:

Scan the string from left to right and for each char:
1. If a left bracket, do nothing
2. If a number, push onto numberStack
3. If an operator, push onto operatorStack
4. If a right bracket, do an evaluation:
   a) pop from the operatorStack
   b) pop two numbers from the numberStack
   c) perform the operation on these numbers (in the right order)
   d) push the result back on the numberStack
At end of string, the single value on the numberStack is the answer.

The above example (((3+8)-5)*(8/4)): at the right brackets

```

8 3  →  11  
nums ops nums ops
```

TO BE READ: -5)*(8/4))
11.5 Application: Convex Hulls

The convex hull of a set of points in the plane is a polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight elastic band that surrounds all the nails. For example, the highest, lowest, leftmost and rightmost points are on the convex hull. It is a basic building block of several graphics algorithms.

One algorithm to compute the convex hull is Graham’s scan. It is an application of a stack. Let 0 be the leftmost point (which is guaranteed to be in the convex hull). Then number the remaining points by angle from 0 going counterclockwise: 1, 2, ..., n − 1. Let n<sup>th</sup> be 0 again.

**Graham Scan**

1. Sort points by angle from 0
2. Push 0 and 1. Set i=2
3. While i ≤ n do:
   
   If i makes left turn w.r.t. top 2 items on stack
   
   then { push i; i++ }
   
   else { pop and discard }

We do not attempt to prove that the algorithm works. The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is
looked at only once, the loop is executed at most $2n$ times. There is a constant-time method for checking, given three points in order, whether the angle is a left or a right turn. This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$.

For the points given earlier, the labeling is as follows:

The algorithm proceeds:

```plaintext
push(0)
push(1)
push(2)
pop(2), push(3)
pop(3), push(4)
push(5)
pop(5), push(6)
pop(6), push(7)
push(8)
push(0)
```

### 11.6 Queue Basics

A queue is a **linear** data structure that allows items to be added only to the rear of the queue and removed only from the front of the queue. Queues are **FIFO** structures: First-in First-out. They are used in operating systems to schedule access to resources such as a printer.

The two standard modification methods are:

- **void enqueue(QueueType ob)**: insert the item at the **rear** of the queue
- **QueueType dequeue()**: delete and return the item at the **front** of the queue (sometimes called the first item).

A simple task with a queue is **echoing** the input (in the order it came): repeatedly insert into the queue, and then repeatedly dequeue.
11.7 Queue Implementation as Array

The natural approach to implementing a queue is, of course, an array. This suffers from the problem that as items are enqueued and dequeued, we reach the end of the array but are not using much of the start of the array.

The solution is to allow wrap-around: after filling the array, you start filling it from the front again (assuming these positions have been vacated). Of course, if there really are too many items in the queue, then this approach will also fail. This is sometimes called a circular array.

You maintain two markers for the two ends of the queue. The simplest is to maintain instance variables:

- double data[] stores the data
- int count records the number of elements currently in the queue, and int capacity the length of the array
- int front and int rear are such that: if rear <= front, then the queue is in positions data[front] ... data[rear]; otherwise it is in data[front] ... data[capacity-1] data[0] ... data[rear]

For example, the enqueue method is:

```c
void enqueue(double elem)
{
    if (count == capacity)
        return;
    rear = (rear+1) % capacity ;
    data[rear] = elem ;
    count++;
}
```

Practice. As an exercise, provide the dequeue method.

11.8 Queue Implementation as Linked List

A conceptually simpler implementation is a linked list. Since we need to add at one end and remove at the other, we maintain two pointers: one to the front and one to the rear. The front will correspond to the head in a normal linked list (doing it the other way round doesn’t work: why?).
11.9 Application: Discrete Event Simulation

There are two very standard uses of queues in programming. The first is in implementing certain searching algorithms. The second is in doing a simulation of a scenario that changes over time. So we examine the CarWash simulation (taken from Main).

The idea: we want to simulate a CarWash to gain some statistics on how service times etc. are affected by changes in customer numbers, etc. In particular, there is a single Washer and a single Line to wait in. We are interested in how long on average it takes to serve a customer.

We assume the customers arrive at random intervals but at a known rate. We assume the washer takes a fixed time.

So we create an artificial queue of customers. We don’t care about all the details of these simulated customers: just their arrival time is enough.

\[
\text{for } \text{currentTime running from 0 up to end of simulation:}
\]

1. toss coin to see if new customer arrives at currentTime;
   
   if so, enqueue customer

2. if washer timer expired, then set washer to idle

3. if washer idle and queue nonempty, then
   
   dequeue next customer
   
   set washer to busy, and set timer
   
   update statistics

It is important to note a key approach to such simulations, is to look ahead whenever possible. The overall mechanism is an infinite loop:

\[
\text{while(simulation continuing) do }
\]

\[
\text{dequeue nextEvent;}
\]

\[
\text{update status;}
\]

\[
\text{collect statistics}
\]

\[
\text{precompute associated nextEvent(s) and add to queue;}
\]

\[
\}
\]

Thus when we “move” the Customer to the Washer, we immediately calculate what time the Washer will finish, and then update the statistics. In this case, it allows us to discard the Customer: the only pertinent information is that the Washer is busy.

Sample Code

Here is code for an array-based stack, and a balanced brackets tester.

```
ArrayStack.h
ArrayStack.cpp
brackets.cpp
```