Tutorial 2: Lists and Hashing
Example Solutions

(1) (a) Here is some simple pseudocode for an algorithm that reverses a singly linked list. An alternative is to use recursion.

Algorithm revList(L)
Input: A linked list L
Output: L in reverse order

i. **if** L.isEmpty() **then**
ii. **return** L
iii. **else**
iv. $L' \leftarrow ()$
v. $p \leftarrow L$.first()
vi. L'.insertFirst(L.element(p))
vii. **while** not(L.isLast(p)) **do**
viii. $p \leftarrow L$.next(p)
ix. L'.insertFirst(L.element(p))
x. **return** L'

If the list is empty the algorithm just executes the first two lines and this takes non-zero constant time so it is $\Theta(1)$. Otherwise the while loop body is executed n times (once for each element). The body of the loop takes $\Theta(1)$ time as do the other statements before and after the while loop (this includes the last execution of the while loop statement to find that we are at the end of the list). Thus the overall time is $\Theta(1)\Theta(n) + \Theta(1) = \Theta(n)$.

(b) To achieve $\Theta(1)$ time for reversal we maintain backwards as well as forwards pointers for each cell. We keep a pointer to the head of the list and one for the end. We also need to keep a record of which pointers in the list cells are forward and which backward (e.g., if the pointers are kept under names A and B respectively and A currently holds forward pointers we record this). Reversing the list is achieved by swapping the two pointers and swapping the name of the current forward pointer to the alternative in our record (e.g., swap A for B). Of course we pay a price in extra storage.

(2) (a) This is a fairly straight forward use of stacks. Figure 4.1 shows the sequence staring with an empty stack.

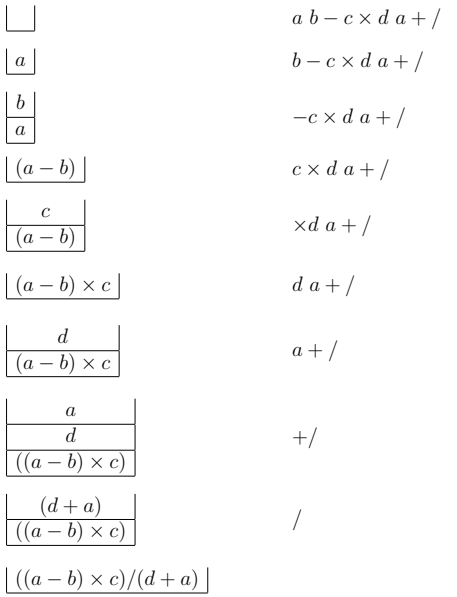

Figure 4.1. Processing an expression in reverse Polish notation by a stack.

(b) The number of stack operations carried out with an input of length n is $O(n)$. The reason is simple: we process the input from left to right. Each operand causes 1 push operation, this is of course $O(1)$. Each operator causes 2 pop operations, if this is not possible then the expression was ill formed and we stop. Otherwise we then push the result back onto the stack, so this is a total of 3 stack operations which again is $O(1)$. Thus in the worst case we do $nO(1)$ stack operations, i.e., $O(n)$ as claimed.

(c) The preceding argument shows that for a properly formed expression we will always do $\Theta(n)$ stack operations, since each step in fact takes time $\Theta(1)$ and we process all the n symbols in the input.

(3) Here is a table showing the hash table after all elements have been hashed in (in an implementation we would hold the contents of each entry in a linked list).

Hashcode	Bucket
0	
1	20
2	
3	
4	16, 5
5	44, 88, 11
6	94, 39
7	12, 23
8	
9	
10	

(4) (a) The idea is to express $h(w)$ using Horner's rule in the form

$$h(w) = \mathbf{ord}(w_{n-1}) + 128\Big(\mathbf{ord}(w_{n-2}) + 128\big(\mathbf{ord}(w_{n-3}) + \cdots + 128(\mathbf{ord}(w_0))\cdots\big)\Big),$$

and then to perform all arithmetic modulo m. This leads to the following procedure for computing h:

```
HASH(w)
  h ← 0
  for i ← 0 to n − 1 do
    h ← (128 ∗ h + ord(w_i)) mod m
  return h
```

Note that HASH has runtime $\Theta(n)$ and so is efficient[1]. Moreover, since $h < m$, HASH never stores an integer larger than $128(m-1) + 127 = 128m - 1$. This integer can clearly be represented in $b' = b + 7$ bits, where b is the number of bits required by m.

[1] If a hash function uses all of the input string then it must look at each symbol so its runtime will be at least $\Omega(n)$. The only way to avoid a linear growth rate is to ignore most of the input string, not just a proportion c since then we still need to look at cn symbols.

The rest of this answer is for illustration and would not normally be required for this course (but make sure you know and understand the basic facts presented here). To check that the procedure is correct, let v_i denote $value(w_0 w_1 \ldots w_i)$. We claim that, after the $(i+1)$st iteration of the loop, the value of h is $v_i \bmod m$. To see this, note from Horner's rule that $v_i = 128v_{i-1} + \mathbf{ord}(w_i)$. This implies

$$v_i \bmod m = \big(128(v_{i-1} \bmod m) + \mathbf{ord}(w_i)\big) \bmod m, \qquad (*)$$

so the assignment in the loop correctly updates h. (Equation $(*)$ relies on the following elementary properties of modular arithmetic. For integers p and q, write $p \equiv q \pmod m$ to denote the fact that $p \bmod m = q \bmod m$. Then if $p \equiv p' \pmod m$ and $q \equiv q' \pmod m$, we have

$$p + q \equiv p' + q' \pmod m \qquad \text{and} \qquad pq \equiv p'q' \pmod m.$$

These two facts suffice to justify $(*)$.)

(b) If $m = 2^{16}$ then $128^i \bmod m = 0$ for all $i \geq 3$. Hence $h(w)$ depends only on the last three characters of w. This is particularly bad in the current application since English words have relatively few distinct three-letter endings: thus huge numbers of words with common endings (such as –ion or –ght) will hash to the same slots in the table, resulting in long search times. This behaviour can be avoided by arranging for m not to be a multiple of the radix 128: in fact, m should preferably be relatively prime to 128, and may as well be a prime. Thus $m = 50021$ would be a reasonable choice.