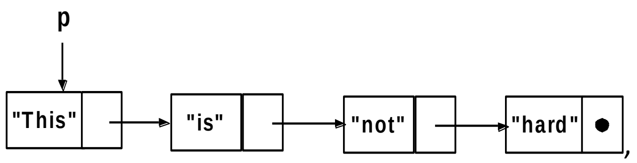
Program Design Exam 4 **Solution**

Do all work on this exam; no scratch paper allowed.
Also, no calculators. Turn off cell phones.

For problems 1 and 2, complete the code for two functions that have as input a linked list of nodes having info field a string. The first function returns the sum of the lengths of the strings in the list. The second function returns a dynamically created string containing the concatenation of the strings in the list, in order, with consecutive node strings separated by a space. The second function will probably need to call the first function.



For example, for the list shown above, the first function would return 13 and the second function would return a dynamic array containing the string "This is not hard".
You may not assume any upper bound on the length of the strings in the nodes.

```
typedef struct node {
	char * info;
	struct node *next;
}  Node;
```

1.

```
int totalStringLength(Node *p)
{
   int len = 0;
   Node * tmp = p;
   while (tmp != NULL) {
	   len += strlen(p->info) +1;
	   tmp = tmp->next;
   }
   return len;
}
```

2.

```
char  *listConcatenate( Node *p)   /* if the list is empty, returns the empty string */
{
    char *bigStr = malloc(totalStringLength(p) +1);
    bigStr[0] = 0;
    tmp = NULL;

    if (p == NULL)
           return bigStr;

    strcat(bigStr,p->info);
    tmp = p->next;
    while(tmp != NULL)
    {
           strcat(bigStr," ");
           strcat(bigStr,tmp->info);
           tmp = tmp->next;
    }
   return bigStr;
}
```

3. Trace the execution of the RPN evaluation algorithm by showing the contents of the stack, the output and the value in the variables left and right after each Pop or Push operation. The stack grows from left to right, so the rightmost position is the bottom of the stack. You may not need all the rows below; if you need more, draw them yourself.

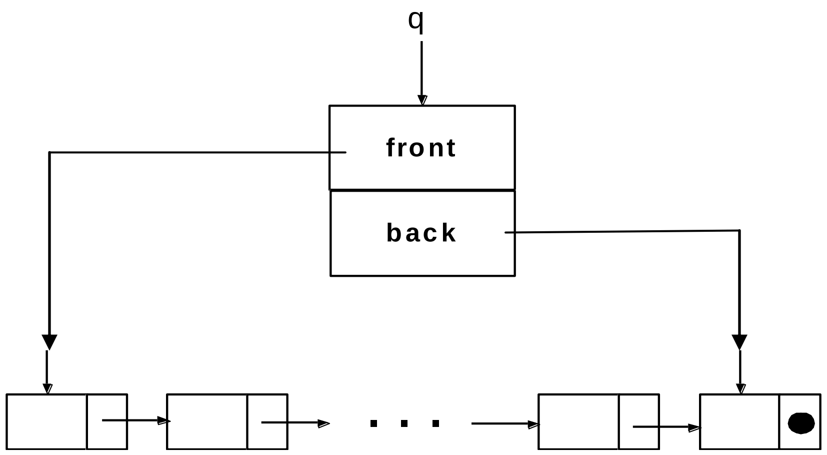
left	right	Stack	Input
			2 4 1 – 3 * +
		2	**4 1 – 3 * +**
		2 4	**1 – 3 * +**
		2 4 1	**– 3 * +**
	1	**2 4**	**– 3 * +**
4	**1**	**2**	**– 3 * +**
		2 3	**3 * +**
		2 3 3	*** +**
	3	**2 3**	*** +**
3	**3**	**2**	*** +**
		2 9	**+**
	9	**2**	**+**
2	**9**		**+**
		11	

A **queue** is a list in which insertions can only be done at one end (the back of the queue) and deletions can only be done at the other end (the front of the queue). The insert function is traditionally called Enqueue and the delete function is traditionally called Dequeue. One implementation uses a linked list with the following typedefs (for a queue of integers):

```
typedef struct node {
        int  info;
        struct node *next;
} Node;
```

```
typdef  struct {
        Node *front;
        Node *back;
} Queue;
```

4. In the diagram below, connect each of the front and back pointer boxes to the appropriate node in the linked list. Your choice must be such that the Enqueue and Dequeue functions will have very efficient implementations



Hint: think about how you would implement the Enqueue and Dequeue functions for each of the two possibilities before you draw your connections.

5. Now complete the code to implement the Dequeue function; remember, this function deletes the node at the front of the queue. It returns the value that was in that front node via the parameter val.

```
void Dequeue(int *val, Queue *q)
{
    Node * tmp = q->front;
    assert(!EmptyQueue(q));
    *val = tmp->info;
    q->front = tmp->next;
     free(tmp);
}
```