Problem Set 2 Solutions

Problem 1. Consider the following function:

```
/* Makes an array of 10 integers and returns a pointer to it */

int *makeArrayOfInts(void) {
  int arr[10];
  int i;
  for (i=0; i<10; i++) {
    arr[i] = i;
  }
  return arr;
}
```

Explain what is wrong with this function. Rewrite the function so that it correctly achieves the intended result using malloc().

Solution:

The function is erroneous because the array arr will cease to exist after the line return arr, since arr is local to this function and gets destroyed once the function returns. So the caller will get a pointer to something that doesn't exist anymore, and you will start to see garbage, segmentation faults, and other errors.

Arrays created with malloc() are stored in a separate place in memory, the heap, which ensures they live on indefinitely until you free them yourself.

The correctly implemented function is as follows:

```
int *makeArrayOfInts() {
  int *arr = malloc(sizeof(int) * 10);
  int i;
  for (i=0; i<10; i++) {
    arr[i] = i;
  }
  return arr;  // this is fine because the array itself will live on
}
```

Problem 2. Consider the following program:

```
#include <stdio.h>
#include <stdlib.h>

void func(int *a) {
  a = malloc(sizeof(int));
}

int main(void) {
  int *p;
  func(p);
  *p = 6;
  printf("%d\n",*p);
  free(p);
  return 0;
}
```

Explain what is wrong with this program.

Solution:

The program is not valid because func() makes a *copy* of the pointer p. So when malloc() is called, the result is assigned to the copied pointer rather than to p. Pointer p itself is pointing to random memory (e.g., 0x0000) before and after the function call. Hence, when you dereference it, the program will (likely) crash.

If you want to use a function to add a memory address to a pointer, then you need to pass the *address* of the pointer (i.e. a pointer to a pointer, or "double pointer"):

```
void func(int **a) {
  a = malloc(sizeof(int));
  *a = malloc(sizeof(int));
}

int main(void) {
  int *p;

  func(&p);
  *p = 6;
```

```
  printf("%d\n",*p);
  free(p);
  return 0;
}
```

Problem 3. Write a C-program to compute the first n Fibonacci numbers, print them on the standard output and store them in a dynamic array of unsigned long long int numbers (8 bytes, only positive numbers), where *n* is given as command line argument. For example, **./fib 60** should result in 1548008755920.

Hint: The placeholder %lld (instead of %d) can be used to print an unsigned long long int. Remember that the Fibonacci numbers are defined as Fib(1) = 1, Fib(2) = 1 and Fib(*n*) = Fib(*n*-1)+Fib(*n*-2) for *n*≥3.

Solution:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s number\n", argv[0]);
    return 1;
  }

  int n = atoi(argv[1]);
  if (n > 0) {
    unsigned long long int *arr = malloc(n * sizeof(unsigned long long int));
    arr[0] = 1;
    arr[1] = 1;
    int i;
    for (i = 2; i < n; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }
    printf("%lld\n", arr[n-1]);
    free(arr);              // don't forget to free the array
  }
  return 0;
}
```

Problem 4. Describe in words how you would implement a *queue* ADT using a dynamic linked list. Which of the functions for the linked list implementation of a stack from the lecture need to be changed, and how?

Solution:

In the stack ADT, elements are added to ("push") and removed from ("pop") the beginning of the linked list. For a queue, we have two options: either we add ("enqueue") new elements at the end and continue to take elements off ("dequeue") from the beginning. Or we continue to add elements at the beginning and dequeue from the end. Operating on both ends will be more efficient if we use a datastructure with two pointers: one pointing to the first and one pointing to the last element of a list.

Problem 5. Suppose that you have a stack *S* containing *n* elements and a queue *Q* that is initially empty. Describe how you can use *Q* to scan *S* in order to check if it contains a certain element *x*, with the additional constraint that your algorithm must return the elements back to *S* in their original order. You should **not** use an additional array or linked list — only use *S* and *Q*.

Solution:

The solution is to use the queue Q to process the elements in two phases. In the first phase, we iteratively pop all the elements from S and enqueue them in Q, then dequeue the elements from Q and push them back onto S. As a result, all the elements are now in reversed order on S. In the second phase, we again pop all the elements from S, but this time we also look for the element x. By again passing the elements through Q and back onto S, we reverse the reversal, thereby restoring the original order of the elements on S.

Problem 6. Write a C-program called **llbuild.c** that builds a linked list of integers from user input. The program works as follows:

- starts with an empty linked list called *all* (say), initialised to NULL
- prompts the user with the message "Enter a number: "
- makes a linked list node called *new* from user's response
- appends *new* to *all*
- asks for more user input and repeats the cycle
- the cycle is terminated when the user enters any non-numeric character
- on termination, the program generates the message "Finished. List is " followed by the contents of the linked list in the format shown below.

A sample interaction is shown as follows:

prompt$. /llbuild

Enter an integer: **12**

Enter an integer: **34**

Enter an integer: **56**

Enter an integer: **quit**

Finished. List is 12->34->56

Note that any non-numeric data 'finishes' the interaction. If the user provides no data, then no list should be output:

prompt$. /llbuild

Enter an integer:**#**

Finished.

Solution:

```
// llbuild.c: create a linked list from user input, and print
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node {
  int data;
  struct node *next;
} NodeT;

NodeT *joinLL(NodeT *head1, NodeT *head2) {
// either or both head1 and head2 may be NULL
  if (head1 == NULL) {
    head1 = head2;
  } else {
    NodeT *p = head1;
    while (p->next != NULL) {
      p = p->next;
    }
    p->next = head2; // this does nothing if head2 == NULL
  }
  return head1;
}

NodeT *makeNode(int v) {
```

```
  NodeT *new = malloc(sizeof(NodeT));
  assert(new != NULL);
  new->data = v;
  new->next = NULL;
  return new;
}

void showLL(NodeT *list) {
  NodeT *p;
  for (p = list; p != NULL; p = p->next) {
    printf("%d", p->data);
    if (p->next != NULL)
      printf("->");
  }
  putchar('\n');
}

void freeLL(NodeT *list) {
   NodeT *p = list;
   while (p != NULL) {
      NodeT *temp = p->next;
      free(p);
      p = temp;
   }
}

int main(void) {
  NodeT *all = NULL;
  int data;

  printf("Enter an integer: ");
  while (scanf("%d", &data) == 1) {
    NodeT *new = makeNode(data);
    all = joinLL(all, new);
    printf("Enter an integer: ");
  }
  if (all != NULL) {
```

```
    printf("Finished. List is ");
    showLL(all);
    freeLL(all);
  } else {
    printf("Finished.\n");
  }
  return 0;
}
```

Problem 7. Extend the C-program in Q6 to split the linked list in two equally-sized halves and output the result. If the list has an odd number of elements, then the first list should contain one more element than the second.

Note that:

- your algorithm should be 'in-place' (so you are not permitted to create a second linked list or use some other data structure such as an array);
- you should not traverse the list more than once (e.g. to count the number of elements and then restart from the beginning).

An example of the program executing could be

```
prompt$ ./llsplit
Enter an integer: 421
Enter an integer: 456732
Enter an integer: 321
Enter an integer: 4
Enter an integer: 86
Enter an integer: 89342
Enter an integer: 9
Enter an integer: #
Finished. List is 421->456732->321->4->86->89342->9
First half is 421->456732->321->4
Second half is 86->89342->9
```

Solution:

The following solution uses a "slow" and a "fast" pointer to traverse the list. The fast pointer always jumps 2 elements ahead. At any time, if slow points to the i^{th} element, then fast points to

the $2 \cdot i^{th}$ element. Hence, when the fast pointer reaches the end of the list, the slow pointer points to the last element of the first half.

```
NodeT *splitList(NodeT *head) {   // returns pointer to second half
  assert(head != NULL);

  NodeT *slow = head;
  NodeT *fast = head->next;

  while (fast != NULL && fast->next != NULL) {
    slow = slow->next;
    fast = fast->next->next;
  }
  NodeT *head2 = slow->next;    // this becomes head of second half
  slow->next = NULL;            // cut off at end of first half
  return head2;
}
```