SCC 120 Introduction to Data Structures

Workshop Three: Linked Lists

1. Consider the C function `malloc(K)`, where K is a positive integer. What does `malloc` do?

a. Allocates *K* bytes of memory in the current stack frame.
b. Releases *K* bytes of memory to the free space pool.
c. Attempts to allocate *K* bytes of memory in the heap and, if successful, returns the memory address of the first byte.
d. Allocates space for a variable at the memory address *K*.

2. Assuming an `int` type occupies 4 bytes and `a char` type occupies 1 byte, what will be printed out by `printf`?

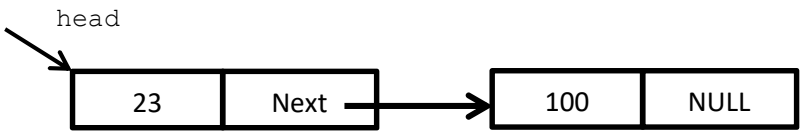
```
int x, y[10];
char z;
printf("%d %d %d\n", sizeof(x), sizeof(y), sizeof(z));
```

4 40 1
`sizeof` returns the number of bytes used to store a variable.
int variable x occupies 4 bytes; integer y[10] has 10 elements, so the array occupies 40 bytes (4 * 10); and z is a char variable, it occupies 1 byte.

3. Consider a linked list that is made up of nodes of type

```
typedef struct{
   int val;
   struct node* next;
}node;
```

The list has two nodes. You are given a pointer "`head`" of type `node`, which points to the first node of the list; and that the `next` field of the last node is NULL. The list is described in the following diagram:



3.1 Which of the following code statements will remove the second node from the list?

```
a) head->next->val=0;
b) head=NULL;
c) head->next = head->next->next;
d) head->val=0;
```

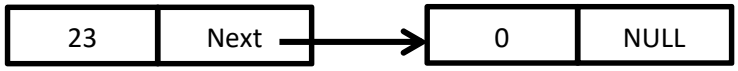
c. `head->next = head->next->next;`

head->next is a pointer points to the second node.
(head->next)->next refers to the next filed of the second node, which has a value of NULL

Therefore, statement `head->next = head->next->next;` stores NULL to head->next (i.e. the next field of the first node).

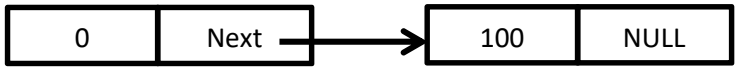
This allows us to remove the second node from the list as the second node can no longer be accessed (nobody points to it).

a. head->next->val=0;
This changes the value stored in the val field of the second node to 0.



b. head=NULL
This empties the list

d. head->val = 0;
This changes the value stored in the val field of the first node to 0.



3.2 Write a few lines of code to add a new node with a value of 17 at the end of the list (depicted in the diagram).

```
void insert(node* head) {
     node *t = malloc(sizeof(node));
     t->val = 17;
     t->next = NULL;
     head->next->next = t;
}
```

3.3 Complete the following function `count()` that takes a linked list as input, and prints out the number of nodes in the linked list, assuming the list has at least two nodes

```
int count(node* head) {
//head points to the first node of the chain

    unsigned count=0;

    node* pt = head;
    while(pt != NULL)
    {
         count++;
         pt = pt->next;
    }

    return count;
}
```

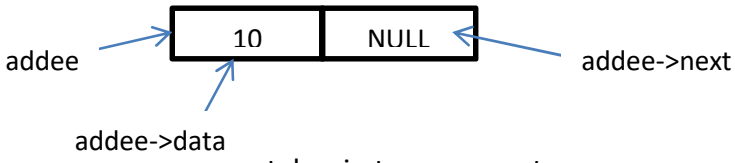
4. Consider the following data structure:

```
typedef  struct _chainCell{
        int data;
        struct _chainCell* next;
} chainCell;
```

The following piece of code creates a chainCell item, addee

```
chainCell *addee = malloc(sizeof(chainCell));
addee -> data = 10;
addee -> next = NULL;
```

This will create a chainCell item looks like:



The following functi[illegible], takes in two paramete[illegible]ameter, `header` points to the first node of the list. The second parameter, `data`, is an integer value. This function creates a new `chainCell` item to store `data`, and inserts the new `chainCell` item, `addee`, to the list. **Pay attention to where of the list the new item is inserted to.**

```
chainCell* addInOrder(chainCell *header, int data) {
        chainCell *addee= malloc(sizeof(chainCell));
        addee -> data = data;
        addee -> next = NULL;

        if (header == NULL) { //   special case one : adding to an empty list
            header = addee;
            return header;
        }
        if (data  <  header->data){ // special case two : data to be added is added at start of chain
            addee->next = header;
            header = addee;
            return header;
        }
        chainCell *pt = header, *previous = NULL;
        // general case – data added in somewhere of the chain
      while ((pt != NULL) && (data > pt->data)){
            previous = pt;
            pt = pt->next;
        }
```

```
            previous->next = addee;
            addee->next = pt;
            return header;
}
```

```
chainCell* header = NULL; // null at this point

// incrementally assemble a chain

header = addInOrder(header, 10);

header = addInOrder(header, 5);

header = addInOrder(header, 8);

header = addInOrder(header, 20);

header = addInOrder(header, 6);
```

The above code creates a list with 5 nodes using the `addInOrder`() function. Here `header` points to the first node of the chain. Write down the value of the `data` field for each node of the chain:

`The` addInOrder() function adds numbers into the list in ascending order. You can draw on a paper to see how the list grows over time. `header` always points to the first node of the list. After executing the above code, the chain has the following values in each node. Playing the computer to understand why!