Solver: Zhou Xinzi

Email Address: zhou0199@e.ntu.edu.sg

1. (a)

The outputs are as follows

```
a = 2, b = 2
a = 4, b = 2
a = 2, b = 2
a = 6, b = 4
a = 2, b = 2
a = 4, b = 2
a = 10, b = 6
Result = 16 (with newline in the end)
```

Explanation:

It is easy for us to find the recurrence relation of f(n), which is f(n) = f(n-1) + f(n-2) for n > 2. However, we need to determine what would be printed out during the program execution. When we are calling a function, we are actually pushing it to the system stack and waiting for its return. Let us trace how this program executes.

1) We call f(6). Before we get the return, we need the value of f(5) and f(4), f(5) is called first.

2) To get the value of f(5), we call f(4). And than f(3).

3) When we call f(3), we only need the values of f(2) and f(1), which would return immediately without printing anything. Thus, "a = 2, b = 2" would be the first line printed out. And f(3) returns 4.

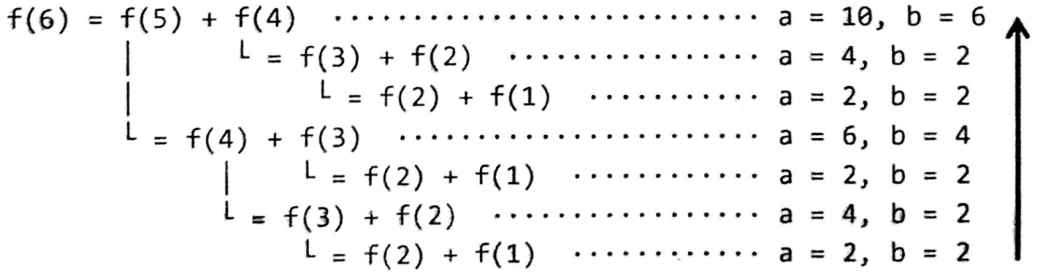
4) Now we come back to calculate f(4). In this situation, f(3) and f(2) are known. So "a = 4, b = 2" is printed. And 6 is returned.

5) Now we come back to calculate f(5), where we need f(4) and f(3). Although we calculate f(3) before getting f(4), the computer have to recalculate f(3), which means another line of "a – 2, b = 2" will be printed before "a = 6, b = 4".

6) Now we are back to f(6). Just like what happens in 5), with the return value of f(5), we need to calculate f(4) again. So what is printed in 3) and 4) will be printed again and then "a = 10, b = 6" is printed.

7) Finally, "Result = 16" is printed.

This whole procedure is illustrated in the diagram below.



(b)

(i) `19 (int)`
(ii) `"Peter" (string) or address of string "Peter"`
(iii) `'O' (char)`
(iv) `'e' (char)`
(v) `'e' (char)`
(vi) `'J' (char)`

Explanation:

One of the difficult parts of this question is the precedence relationship of C operators. For operators appeared in this question, their orders of precedence from high to low are:

Operator	Description	Association
() . ->` ++	Parentheses Member selection via object name Member selection via pointer Postfix increment	Left-to-right
++ *	Prefix increment Dereference	Right-to-left
-	Subtraction	Left-to-right

Another thing we should pay attention to is that the prefix increment return the value after increment while the postfix increment return its original value.

For (i), it actually returns the age of peter plus one. The age of Peter in memory was changed.

For (ii), it is the address of Peter's name, where stores a string.

For (iii), it returns the previous character of the first letter of Peter's name, which is 'P'.

For (iv), it can be rewrote to `*(++(p->name))`. `p->name` is the address of (or a pointer pointing to) Peter's name. `++(p->name)` move the pointer to the next character. And the * in front makes it return the character itself. What needs to be pointed out is that the increment of `p->name` will make 'real' change in the memory.

For (v), its return value is the same as `*p->name`, which is 'e' (remember the increment in (iv)). The only difference is that the pointer p is now pointing to the second object in array b.

For (vi), `++p` makes the pointer point to the last object. So its value would be 'J'.

(c)

One possible version:

```
void reverseAr(char ar[], int n) {
    char t = ar[0];
    ar[0] = ar[n-1];
    ar[n-1] = t;
    if (n > 1) {
        reverseAr(++ar, n-2);
    }
}
```

2. (a)

One possible version:

```
void squeeze(char str[], char c) {
    int read = 0, write = 0;
    while (str[read] != '\0') {
        if (str[read] != c) {
            str[write] = str[read];
            write++;
        }
        read ++;
    }
    str[write] = '\0';
}
```

(b)

One possible version:

```
int strcmp(char *s1, char *s2) {
    if (*s1 == *s2) {
        if (*s1 == '\0') {
            return 0;
        } else {
            return strcmp(++s1, ++s2);
        }
    } else {
        return *s1 > *s2 ? 1 : -1;
    }
}
```

(c)

(i) 'm'
(ii) 'a'
(iii) 'm' (or 'w' if there is a typo in the question)
(iv) 'z'
(v) "mnopqr" (or its address)

Explanation:

In this question, a is a two dimensional array of pointers. Each pointer is pointing to a string.

For sub-questions (i) to (iv), we might convert the expression into more readable ones:

(i) *a[1][0]
(ii) *a[0][0]
(iii) a[1][1][4]
(iv) a[1][2][2]

For sub-question (v), it can be rewrote as a[1][0], which is the address of string "mnopqr". However, we have no way to determine this address with only the address of first element of the array, which is the address of the pointer pointing to string "abc".

3. (a)

Memory leak stands for the situation when a piece of memory spaces allocated to a program becomes inaccessible.

In C programming language, we use 'malloc' or 'calloc' functions to request for allocating memory and, usually, we would use pointers to store the addresses of the allocated memory, so that they can be accessed when needed. When those memory spaces are no longer used, we would use 'free' function to free them manually.

However, sometime we may forget to free unused memory spaces in our programs. This will incur a memory leak.

Memory leak is undesirable because if a program asks for too many memory spaces without properly using them may finally make the computer out of memory or at least waste system resources.

(b)

One possible version:

```
void exchange(linkedlist *ll) {
    listnode *max = ll->head, *min = ll->head, *current = ll->head;
    listnode *beforeMax = NULL, *beforeMin = NULL, *temp = NULL;
    while (current->next != NULL) {
        if (current->next->item > max->item) {
            max = current->next;
            beforeMax = current;
        }
        if (current->next->item < min->item) {
            min = current->next;
            beforeMin = current;
        }

        current = current->next;
    }

    if (max == min) {
        return;
    }

    if (beforeMax == NULL) {
        ll->head = min;
    } else {
        beforeMax->next = min;
    }
    if (beforeMin == NULL) {
        ll->head = max;
    } else {
        beforeMin->next = max;
    }
```

```
        if (ll->tail == max) {
            ll->tail = min;
        }
        if (ll->tail == min) {
            ll->tail = max;
        }

        temp = max->next;
        max->next = min->next;
        min->next = temp;
    }
```

Note: 'linkedlist' and 'listnode' types are defined as:

```
    typedef struct _listnode {
        void *item;
        struct _listnode *next;
    } listnode;

    typedef struct _linkedlist {
        int size;
        listnode *head;
        listnode *tail;
    } linkedlist;
```

This code could be much simpler if we only swap the value of the two nodes or if you are using a bi-directional linked list.

(c)

Suppose there are two stacks stackA and stackB. We are going to use stackA to store those items in the queue and stackB to help implement enqueue function as a temporary stack. Pseudo codes are shown below.

```
    enqueue(newItem):
        while stackA is not empty:
            item = pop(stackA)
            push(stackB, item)

        push(newItem)

        while stackB is not empty:
            item = pop(stackB)
            push(stackA, item)

    dequeue:
        return pop(stackA)
```

4. (a)

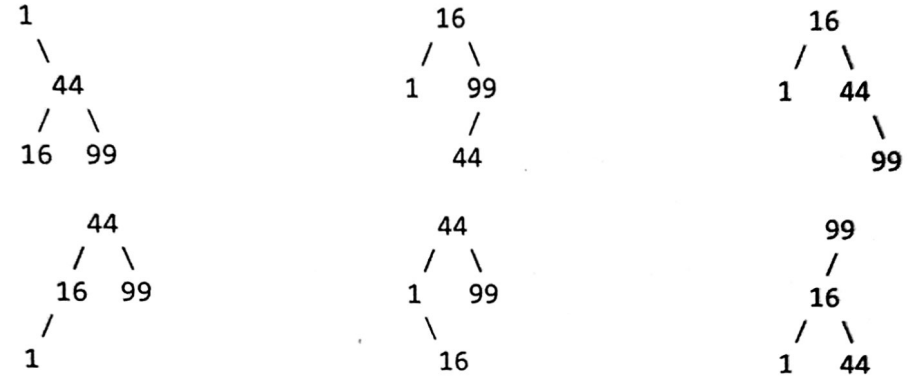
Possible answers are:

1) Nodes in linked lists usually only link to one other node while nodes in a binary tree may link to no more than two nodes.

2) There is a unique order to traverse a linked list while we may traverse a binary tree by many ways.

(b)

The minimal height is 3 (including the root node).



(c)

One possible version:

```
int countonechild(btnode *node) {
    if (node->left == NULL && node->right == NULL) {
        return 0;
    }
    if (node->left == NULL) {
        return 1 + countonechild(node->right);
    }
    if (node->right == NULL) {
        return 1 + countonechild(node->left);
    }
    return countonechild(node->left) + countonechild(node->right);
}
```

Note: the btnode type is defined as:

```
typedef struct _btnode {
    int item;
    struct _btnode *left;
    struct _btnode *right;
} btnode;
```