Solver: Zhao Jingyi

Email Address: jzhao009@e.ntu.edu.sg

1. (a) 8,6
8,4
20,12
20,4
Explanation:
When i = 0:
Before f1, x = 2, y = 4
Inside f1, y = 4, *x = 2
After y = y + *x, y = 6, *x = 2
After *x = *x + y, *x = 8, y = 6
printf prints 8,6
After f1, in main, x = 8, y = 4
printf prints 8,4
When i = 1:
Before f1, x = 8, y = 4
Inside f1, y = 4, *x = 8
After y = y + *x, y = 12, *x = 8
After *x = *x + y, *x = 20, y = 12
printf prints 20,12
After f1, x = 20, y = 4
printf prints 20,4

(b)
```
/* missing code */
b[k++] = *p++;
if (i % 3 == 2)
    b[k++] = '#';
i++;
if (!*p) // if (*p == '\0')
    b[k] = '\0';
```
Explanation: the first line copies the character in a to b. The second and third line decides whether we should add the '#' to b. The last two lines checks whether we arrive at the end of the character string a.

(c)
```
/* missing code (i) */
*p != ' '
/* missing code (ii) */
max < len
/* missing code (iii) */
max = len;
```
Explanation: The first missing code means the we won't stop the looping while *p is not a space. The second and third missing code are just updating the max value.

(d)
```
/* missing code (i) */
num = 100; num < 1000; num++
/* missing code (ii) */
```

```
y = (num / 10) % 10;
z = num % 10;
/* missing code (iii) */
a[i++] = num;
```

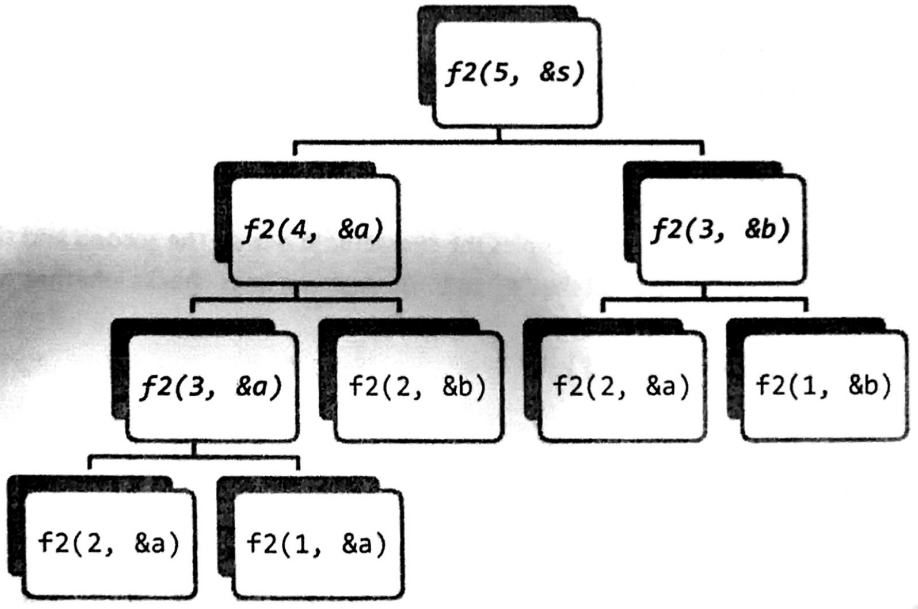
Explanation: The first missing code means we need to loop through all three-digit numbers. The second missing code extracts the second and last digit of num. `num / 10` means we discard the last digits, so `(num / 10) % 10` means we extract the second digit. The third missing code adds this num into the array.

(e)
```
/* missing code*/
max = a[row][0];
for (col = 1; col < 4; col++)
    if (a[row][col] > max)
        max = a[row][col];
if (row == 0)
    min = max;
else
    min = (max < min ? max : min);
```

Explanation: The first line initializes the `max` value to be the first number of this row. Next, the `for` loop finds the `max` of this row. Finally, we update the `min` value.

2. (a)
f2(): 0
f2(): -1
f2(): 0
f2(): -1
Result: -1

Explanation: Since this question involves 2 recursions, it might be clearer to draw a binary tree representing each recursive call.



Notice that we call `f2(n - 1, &a)` before `f2(n - 2, &b)`. Therefore, we "traverse" the left subtree before the right subtree. Notice also that if the first parameter

<= 2, we will not print out anything. (Those function calls that will execute printf is in **bold** and *italic* font)

(b)
```
/* missing code */
for (i = 0; i < 3; i++)
{
    scanf("%s%d", man[i].name, &man[i].age);
    if (i == 0)
        max = min = 0;
    else if (max < man[i].age)
        max = i;
    else if (min > man[i].age)
        min = i;
}
for (i = 0; i < 3; ++i)
    if (i != max && i != min)
    {
        printf("%s %d", man[i].name, man[i].age);
        break;
    }
```

Explanation: The first `for` loop reads in all the information and finds the index of the max age and min age. The second `for` loop looks for the middle (neither the `max` nor the `min`) and prints the corresponding person.

(c)
```
/* missing code (i) */
*s == '\0'
/* missing code (ii) */
0;
/* missing code (iii) */
rStrLen(s + 1) + 1;
```

Explanation: The base case is when the string is empty. For the recursive call we reduce the string length by 1 and reflect this change be advancing the pointer. After we find the length of the rest of the string, we add 1 to indicate that we add back this character.

(d)
```
/* missing code (i) */
if (!p->source)
    *t++ = *s;
else
    *t++ = p->code;
s++;
/* missing code (ii) */
*t = '\0';
```

Explanation: The `for` loop before the missing code (i) will end in two conditions: when `*s == p->source`, which means this character can be encoded, or when `p->source == '\0'`, which means we cannot find this character in the table of rules. Therefore, after the `for` loop we need to differentiate between finding the code or not and write the character in `t` correspondingly. Notice that the `while` loop ends when we encounter the null character in `s`, but we haven't added the null character to `t`. Therefore, we need to do so in the second missing code.

(e)
```
/* missing code (i) */
prtLine(' ', h - i);
prtLine('*', 2 * i - 1);
putchar('\n');
/* missing code (ii) */
if (n != 0)
{
    putchar(c);
    prtLine(c, n - 1);
}
```

Explanation: Observe that we need to print two type of characters `' '`, and `'*'`. And the number of each character to print is easy to find. Lastly we need to print the end of line character as well. Hence the first missing code. For the recursion, if we still need to print more characters (`n != 0`) we print one character and reduce the number by one.

3. (a)

(i) Dynamic memory allocation means we asks for memory to store objects from the system from runtime. `malloc()` accepts the size of the memory we want and asks for that from the system. `free()` deallocates the memory and returns it to the system. If we keep calling `malloc()` but not calling `free()` when we are done with the memory, the memory will be exhausted and we can no longer create new objects.

(ii) When we dynamically allocate memory, we store the data on the heap; when we define statically allocated variables, we store the data on the stack.

(b)

(i) Stack data structure is Last In, First Out (LIFO). We want that once the recursive procedure ends, the program goes back one level and runs the rest of the code in the calling procedure. Therefore, recursive procedures make use of stacks.

(ii) Using circular queues is more space efficient. For normal static queues, once we perform the dequeue operation, we cannot make use of those memory spaces again. For circular queues, after we perform the dequeue operation, if we want to enqueue new items, we can just add to the front of the array, but conceptually it is still appended at the end.

(iii) A linked list allows the user to access all its items, although the user might have to traverse the whole list. A queue does not provide such accessibility. The user can only add items to the end of the queue and delete items from the front of the queue. The user does not have access to the elements in between.

(c)
```
/* missing code (i) */
newNode->next = *headPtr;
*headPtr = newNode;
/* missing code (ii) */
current->next->item < newNode->item
/* missing code (iii) */
current = current->next;
/* missing code (iv) */
newNode->next = current->next;
current->next = newNode;
```

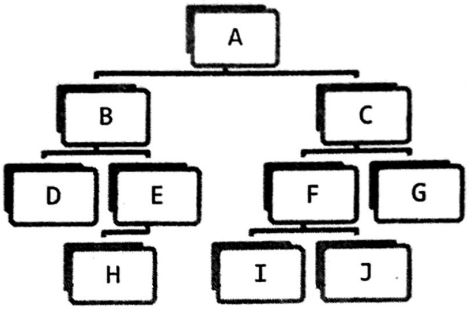
Explanation: missing code (i) is executed when the list is empty (`*headPtr == NULL`) or when the `newNode->item` is less than the first item in the list

((*headPtr)->item >= newNode->item). Therefore, we need to update the headPtr as well. Missing code (ii) and (iii) is executed when we need to insert the newNode after the first node. Therefore, we need to move to the correct position, which means we will insert the newNode at the position after the current node. Missing code (iv) describes the procedure to insert the newNode after the current node.

4. (a)

(i) Notice that the definition of depth is the number of movements from the root to the deepest leave. Therefore, the maximum number of nodes of a binary tree of depth d is $2^{d+1} - 1$

(ii)



Given the inorder traversal and either the preorder traversal or the post order traversal, we can reconstruct the binary tree. To do so, always remember that preorder traversal is: root->left->right; inorder traversal is: left->root->right; post order traversal is: left->right->root.

The procedure to obtain the tree is as follows:

Look at the first node in Preorder, it must be the root of the tree: A

Find A in the Inorder traversal, all the nodes on its left are in the left subtree of A, and all other nodes are in the right subtree: A's left subtree: DBHE, right subtree: IFJCG

Find the first node in A's left subtree in Preorder, it must be the root of A's left subtree: B

Find B in A's left subtree in Inorder, all the nodes on its left are in the left subtree of B, and all other nodes between B and A are in the right subtree of B: B's left subtree: D, B's right subtree: HE

Find the first node in B's right subtree in Preorder, it must be the root of B's right subtree: E. Therefore we can conclude the left subtree of A.

The right subtree of A can also be obtained in the same way.

(b)
```
/* missing code (i) */
node = dequeue(&head, &tail);
printf("%d ", node->data);
/* missing code (ii) */
enqueue(&head, &tail, node->leftPtr);
/* missing code (iii) */
enqueue(&head, &tail, node->rightPtr);
```

Explanation: The code can be referred to in the lecture. Missing code (i) retrieves the front of the queue and prints its value. Missing code (ii) and (iii) try to enqueue node's two children, if any.

(c)
```
/* missing code (i) */
treePtr->data > key
/* missing code (ii) */
binaryTreeSearch(treePtr->leftPtr, key);
/* missing code (iii) */
treePtr->data < key
/* missing code (iv) */
binaryTreeSearch(treePtr->rightPtr, key);
```

Explanation: The code is very simple and self-explanative. Missing code (i) and (ii) represents the key to find is less than the current node's value, and therefore we need to look at its left subtree. Missing code (iii) and (iv) represents the key find is greater than the current node's value, and therefore we need to look at its right subtree.

For reporting of errors and errata, please send me an email
Thank you and all the best for your exams! ☺