Prepared by: Joshua Tan
Email Address: jtan253@e.ntu.edu.sg

1 (a)(i)
Output:

```
%
&
*&
%#
```

Explanation:
The `for` loop is executed five times. Note that the statement `c=str[k]` is an **assignment** that takes place in the `for` loop.

1st execution

Variable	Value
k	0
c	'A'

Hence, `switch(c)` causes the statement `putchar('%')` to be printed. `continue;` terminates the current iteration and causes the conditional test and increment part of the loop to run.

2nd execution

Variable	Value
k	1
c	'B'

Hence, `switch(c)` causes the statement `++k;` to be run. Then, `printf("\n");` prints a new line.

3rd execution

Variable	Value
k	3
c	'C'

Hence, `switch(c)` causes the statement `putchar('&')` to be run. Then `printf("\n");` prints a new line.

4th execution

Variable	Value
k	4
c	'D'

Hence, switch(c) causes the default statement `putchar('*')` to be run. However, since there is no break; statement, `putchar('&')` is also run. Then `printf("\n");` prints a new line.

5th execution

Variable	Value
k	5
c	'A'

Hence, `switch(c)` causes the statement `putchar('%')` to be printed. `continue;` terminates the current iteration and causes the conditional test and increment part of the loop to run.

We have reached the end of the string and after exiting the `for` loop, `putchar('#')` is printed.

1 (a)(ii)
Output:

```
Program
PROGRAM
margor
gram
```

The statement `printf("%c%s\n", *a, b+1);` prints the first character of a and the entire string of b (starting at index 1).

The `while(*(a+i))` block prints the entire a string (starting at index 0).

The `while(--i)` block prints the b string backwards. However, 'p' is not printed because of the prefix decrement. If the postfix decrement `while(i--)` was used, the output would be "`margorp`".

The statement `printf("%s\n", &b[3]);` prints the entire b string (starting at index 3).

1 (b)(i)

```
/* missing code (i) */
c[k] = a[i];
i++;
```

If `a[i]` is alphabetically smaller than `b[j]`, append to c.

```
/* missing code (ii) */
c[k] = b[j];
j++;
```

Else, append `b[j]` to c.

```
/* missing code (iii) */
b[j] != '\0'
```

If b is longer than a, then we later concatenate c and what's left of b. Else, we later concatenate c and what's left of a.

1 (b)(ii)

```
/* missing code (i) */
s++, t++
```

We are trying to find the first instance where the letter in s is different from the letter in t. If we are unable to find the instance, it means that s == t and that the strings are exactly the same (hence we return 0).

```
/* missing code (ii) */
*s - *t
```

Once we've found the first letter difference, we return the ASCII value difference of *s - *t, as given in the question.

1 (b)(iii)

```
/* missing code */
      i = 0, j = 0;
      while (str[i] != '\0')
      {
            while (substr[j] != '\0')
            {
                  if (str[i + j] != substr[j])
                  {
                        count--;
                        break;
                  }
                  j++;
            }
            count++;
            j = 0;
            i++;
      }
```

2 (a)(i)

Output:

```
0 0 1 2 2 4
0 0
2 3
4 6
```

The first `for` loop is responsible for printing the first line.

When i=	*pa	*pb	a	b
0	0	0	{0, ...}	{0, ...}
1	1	2	{0, 1, ...}	{0, 2, ...}
2	2	4	{0, 1, 2, ...}	{0, 2, 4, ...}

The second `for` loop is responsible for printing the next three lines.

When i=	*pa	*pb
0	0+0 = 0	0+0 = 0
1	1+1 = 2	2+1 = 3
2	2+2 = 4	4+2 = 6

2 (a)(ii)

Output:

```
2
30
30
51
```

Tip: Refer to the precedence table.

The first `printf` statement makes p point to `b[1]`. The value `b[1].x` is printed.

The second `printf` statement makes p point to `b[2]`. The dereferenced value of &a`[2]` is printed.

The third `printf` statement prints the dereferenced value of &a`[2]` again. After the value is printed, p points to b`[3]`.

The fourth `printf` statement makes p point to b`[4]`. After &a`[4]` is dereferenced, 1 is added to the value. The value is then printed out.

2 (b)

Errors:

```
7     void input(struct Stud * s);
8     void output(struct Stud * s);
13    input(&s[i]);
15    output(&s[i]);
19    void input(struct Stud *s){
21    scanf("%d", &(s->id));
23    scanf("%s", s->name);
25    void output(struct Stud * s){
26    printf("%s(%d) ", s->name, s->id);
```

2 (c)(i)

```
/* missing code (i) */
j = i;
min = *(a+i);
```

Stores the position of the smallest number in j. Stores the value of the smallest number in min.

```
/* missing code (ii) */
k = i;
max = *(a+i);
```

Stores the position of the largest number in k. Stores the value of the largest number in max.

```
/* missing code (iii) */
*(a+k) = min;
*(a+j) = max;
```

Swaps the position of the minimum and maximum numbers.

2 (c)(ii)

```
/* missing code */
if (*(a+i) > *(a+(*index)))
      *index = i;
findmax(a, n, i+1, index);
```

3 (a)(i)

Items can be added to and removed from any position in the linked list. On the other hand, items can only be added and removed from the top of the stack (first-in-last-out).

3 (a)(ii)

Linked lists are easy to shrink, rearrange, and expand. Arrays are difficult to expand and re-arrange. In addition, insert/remove operations performed on linked lists require a fixed number of operations regardless of list size. Arrays might be required to perform some shifting operations when inserting/removing in the front or in the middle.

3 (a)(iii)

Singly-linked list: Each ListNode is linked to at most one other ListNode. List traversal can only be performed in one direction.

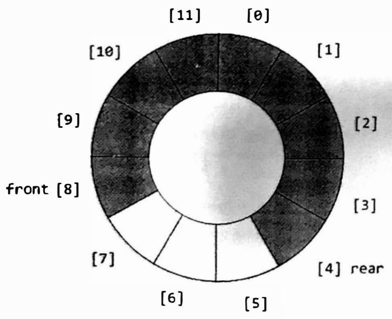
Doubly-linked list: Each ListNode is linked to at most two other ListNodes (previous and next). List traversal can be performed in two directions.

Circular singly-linked list: Similar to singly-linked list, except that the last node's next pointer is not pointing to NULL but pointing to the first node.

Circular doubly-linked list: Similar to doubly-linked list, except that the last node's next pointer is pointing to the first node and the first node's previous pointer is pointing to the last node.

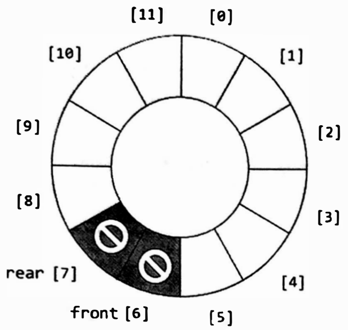
3 (b)(i)

9 elements in the queue



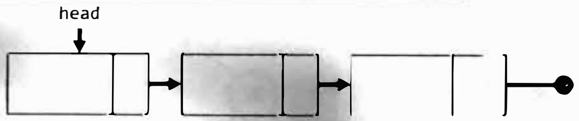
3 (b)(ii)

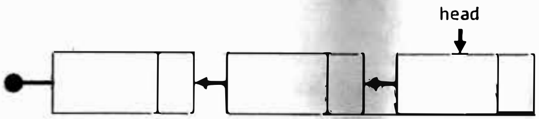
0 elements in the queue



3 (c)

```
/* missing code */
ListNode *traverse = *head;
if (traverse == NULL) return; // list is empty
ListNode *storeNext = NULL;
ListNode *storePrev = NULL;
while (traverse != NULL)
{
	storeNext = traverse->next;
	traverse->next = storePrev;
	storePrev = traverse;
	traverse = storeNext;
}
*head = storePrev;
```





4 (a)

```
/* missing code (i) */
push(stack, root);
```

First, push the root node into the stack.

```
/* missing code (ii) */
pop(stack);
```

Now that we are in the while loop, what we'd like to do is to first pop (and store) the top item in the stack.

```
/* missing code (iii) */
push(stack, node->right);
```

Next, we push the right node into the stack (if it exists). This is because stack operates on a first-in-last-out principle. Since this is a pre-order traversal, we need to process the left node first (hence the right node is pushed first).

```
/* missing code (iv) */
push(stack, node->left);
```

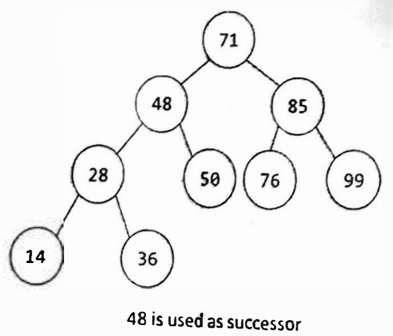
Finally, we push the left node into the stack (if it exists). .

4 (b)(i)

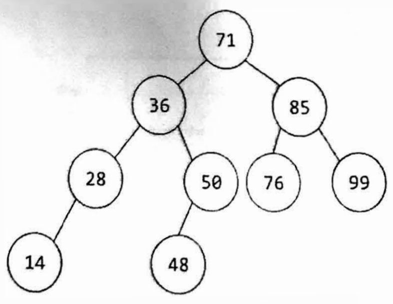
71, 45, 28, 14, 36, 50, 48, 85, 76, 99

4 (b)(ii)

Two possible answers, depending on the node that is chosen as the successor.



48 is used as successor



36 is used as successor

If 48 is chosen as successor, post-order traversal order: 14, 36, 28, 50, 48, 76, 99, 85, 71
If 36 chosen as successor, post-order traversal order: 14, 28, 48, 50, 36, 76, 99, 85, 71

4 (c)

```
int similar(BTNode *tree1, BTNode *tree2)
{
      if (tree1 == tree2) return 1;
      if (tree1 == NULL || tree2 == NULL) return 0;

      return (tree1->item == tree2->item) && similar(tree1->left,
tree2->left) && similar(tree1->right, tree2->right);
}
```