Solver: Kong Zhong Han

Email Address: KONG0110@e.ntu.edu.sg

1)

a)
```
i = 0;
while( str[i] != '\0')
{
      if (str[i] >= '0' && str[i] <= '9')
      {
            j = i; // j = index pointer to current pointer
            while (str[j+1] != '\0') // iterate all char till end
            {
                  str[j] = str[j+1]; // replace curr with next
                  j++; // increment pointer to next char
            }
            str[j] = '\0'; // once at last char, replace with \0
            continue;
      }
      i++; // move to next index if (char != digit)
}
```

Note: This is not exactly a smart algorithm to remove all digits from the string, to begin with, so any other way of implementing this would still be fine as long as it works.

The idea of this algorithm would be to find a digit at any index, and move every character after this character to the left. For example, ab1cef, once I identify that 1 is a digit, i shift every character 1 by 1, so abccef, abceef, abceff, abcef(\0). Since you will be dealing with a string, the null terminator is very important so you must be mindful of it whenever you deal with strings. 'continue' is important as I must not increment my pointer after shifting everything left, as the new char will not be checked if you increment after shifting.

Here is another solution with better complexity:

```
i = j = 0;
while (str[i])
{
    if (str[i] < '0' || str[i] > '9') {
        str[j++] = str[i];
    }
    ++i;
}
str[j] = '\0';
```

b)
```
for (i = 0; i < 4; i++)
        for (j = i + 1; j < 4; j++)
            if (M[i][j] != M[j][i])
                return 0;
    return 1;
```

Note: The basic idea of this function is to compare the symmetric positions and see if they are the same. For a position (i, j), its new position after transposition is (j, i). So to find whether a matrix is symmetric we just loop through all elements and compare each of them. However, we actually only need to loop through half of them, since each comparison takes two elements already.

c)
```
if (i == 0 || b[i] != b[i - 1])
    p = 1;
else if (b[i] == b[i - 1])
    p++;
max = p > max ? p : max;
++i;
```

Note: The basic idea is to check whether the current element we are looking at is the same as the previous element. p is used to count the number of consecutive appearances of the current element so far. If the current element is the same as the previous one, we increment p. Otherwise we reset p to 1 since we see a new number.

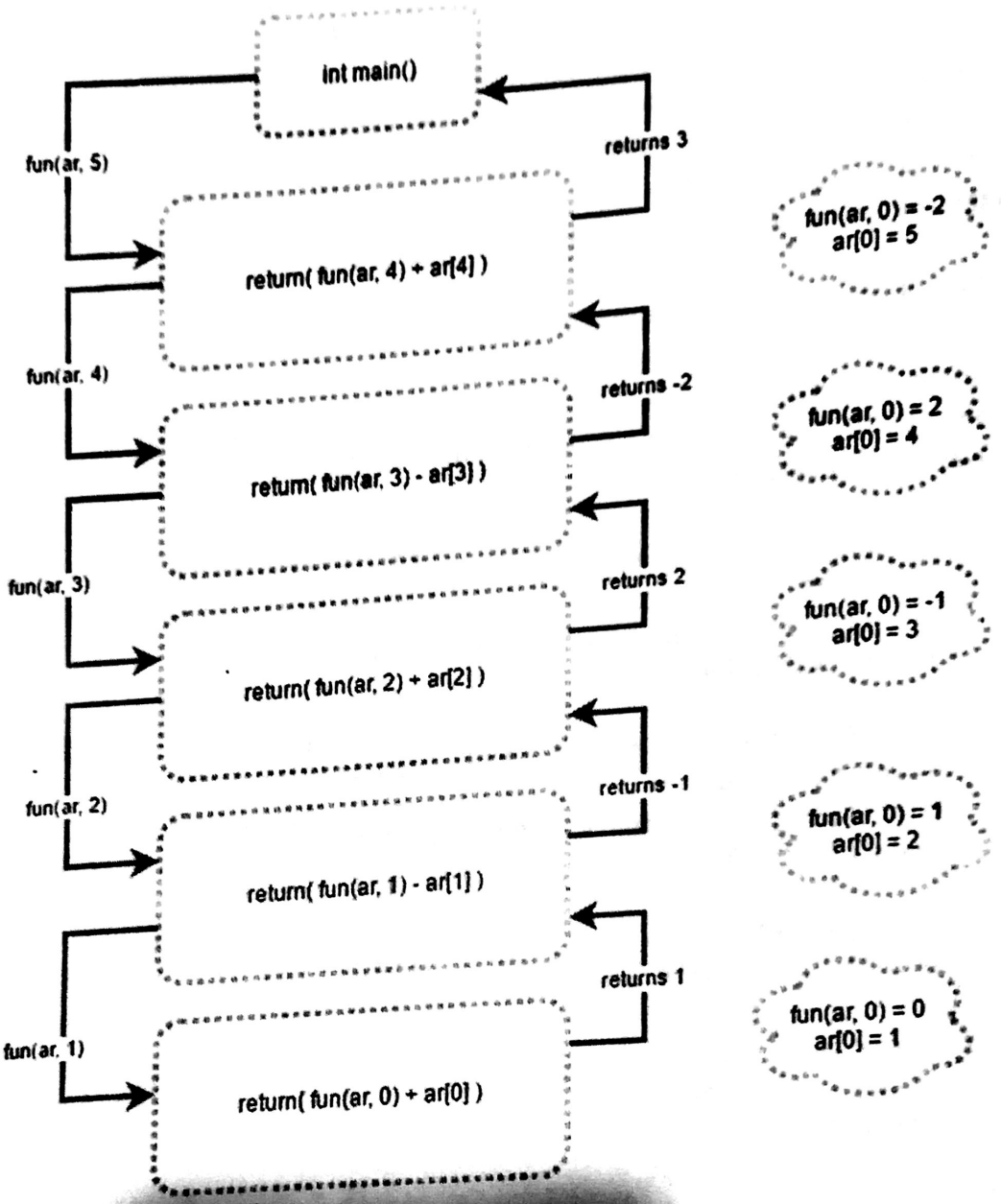
d) Note: Again concept on String and Nested for loops, always remember '\0' at the end.

i)
```
count2 = 0;
while (str2[count2] != '\0'){
   if (str1[count1] == str2[count2]){
         str3[count3] = str1[count1];
         count3++;
         break;
   }
   count2++;
}
```

ii)
```
str3[count3] = '\0';
```

2)

a) Note: Simple tracing exercise, included a diagram of my thinking process if I were to do it.



b)
```
if (studPos == 1){ //base condition
        return 10;
}else{
        return age(studPos-1) + 2;
}
```

Note: Remember that in recursion you will always need a terminating/base condition, so based on the question, assuming that youngest student is 10, hence you will return 10. studPos = 4, age = 2; studPos = 3, age = 4; studPos = 2, age = 6; studPos = 1, age = 16; Therefore, studPos should be 1 when we return 10. This 'thinking' only works if you have a tangible/small set to work with, and might not work for every question though.

c) Note: Concept on structure, and tricky one would be that double type requires '%lf'. Remember that accessing members for structs, when to use a.re and when to use a->re. We must not directly use z->im or z->re before setting where z points to, since in the function z does not point to anywhere. Dereferencing z is very likely to lead to segmentation fault.

i)
```
printf("Enter Real Number(a): ");
scanf("%lf", &a.re);
printf("Enter Imaginary Number(a): ");
scanf("%lf", &a.im);

printf("Enter Real Number(b): ");
scanf("%lf", &b.re);
printf("Enter Imaginary Number(b): ");
scanf("%lf", &b.im);

t = d(&a, &b);
printf("%lf, %lf\n", t->re, t->im);
```

ii)
```
x->re -= y->re;
 x->im -= y->im;
 return x;
```

d) Note: Question might seem daunting but it is really easy, for the mark given. As stated in the #include <string.h> line, you can use strcmp and strlen to make your life easier. Good to read up and remember the important ones, as it will definitely help you in such questions in finals.

However, we cannot direct assign other pointers to first and last inside the firstLast() function. Doing so will simple change the local copy of the two pointers rather than the actual content of first and last array inside main() function. Hence, we need to use another library function strncpy() (or strcpy() but highly discouraged since there it does not specify the max number of characters copied and might lead to buffer overflow.) Using strncpy() will copy the characters over which is not very efficient, but we have no other choice since we can't declare more variables.

In part ii, they have provided extra variables which I did not use as I did not find it necessary. As this questions should be quite similar to the many questions which probably would have done in tutorial/labs and other PYPs, I won't elaborate more on this.

i)
```
strncpy(first, str[0], 40);
strncpy(last, str[0], 40);
for (i = 1; i < size; i++)
{
    if (strcmp(first, str[i]) > 0)
        strncpy(first, str[i], 40);
    else if (strcmp(last, str[i]) < 0)
        strncpy(last, str[i], 40);
}
```

ii)
```
*maxIndex = 0;
*length = strlen(str[0]);
for (i = 1; i < size; i++)
{
    tmp = strlen(str[i]);
    if (tmp > *length)
    {
        maxIndex = i;
        *length = tmp;
    }
}
return str[maxIndex];
```

3)

a) Note: The pointer variable 'tail' will be so called the temp pointer which you should work with, in order to further add copies of the nodes to the new list.

i)
```
tail->next = malloc(sizeof(ListNode));
tail = tail->next;
tail->data = current->data;
tail->next = NULL;
```

ii)
```
current = current->next;
```

b)

i) A buffer overflow occurs when data that is written into the memory is written outside of the buffer, i.e. the area of the memory which does not belong to the variable it is supposed to be written in originally. One example would be as follows:
```
char buff[10];
buff[10] = 'a';
```

ii) An example of a non-linear data structure would be Graphs/Trees, compared to linear ones such as arrays.
Advantage:
- Uses memory efficiently as 'free contiguous memory' is not a requirement for allocation.
- Length/size of data not require to be known, i.e. Dynamic allocating during runtime is possible.

Disadvantages:
- Overhead of the link to the next data item, i.e. Takes more memory required to store the pointer from one node to next.

iii) Static data structure requires you to declare on the size of the variable you will be working with before runtime, hence the size of the variables cannot be changed once the program is running.
As such, the use of Dynamic data structure allows the programmer to dynamically increase and add new data into the program during runtime.

Linked list is called a dynamic data structure as you manually allocate memory space whenever you are adding a node into the list, and there would not be any memory which are not utilized.

c) Note: This question is actually pretty tough at first, especially if you don't have any pre-existing idea of how this thing works. I actually took quite a while to solve this question also, but it's actually do-able with just 3 lines of code.

I would suggest that you **understand the concept behind this**, so that you will be able to perhaps apply the concept of how this works to any similar question you would encounter during your exams.

Assuming we have a list of 4 nodes, i.e. [1, ptr2] -> [2, ptr3] -> [3, ptr4] -> [4, NULL]; at every step of the recursion, take head of node = first, everything else = rest, i.e. first = [1, ptr2], rest = [2, ptr3] ->....-> [4, NULL].
After every 'recurse-step', the list you pass in will be shorter by a node. Hence, the answer in (i). (Also can be inferred logically from the 'if' statement after (i), so the answer in (i) must definitely assign 'rest' to something.

Next, we look at the statement 'first->next->next = first;'. Doesn't really make much sense at first, but let's look at the state when headPtr points to: [3, ptr4] -> [4, NULL], if we apply the statement here, 'first->next->next' refers to the pointer of [4, NULL], and after executing that statement, we get [3, ptr4] -> [4, ptr3] -> [3, ptr4] ... So we can see some reversing action in place now.

So, if the reversing starts here, it would make sense that we make [4, ptr3] be the head of this pointer, hence *headPtr = rest.
[At this point, in an exam setting, If it was me I would have done this question last, if not, I will probably just stop here and just believe that this is the answer that they will have wanted, by intuition. ]

We can always go ahead up one step to check if indeed this would work for another level up, so at this stage, first = [2, ptr3], rest = headPtr(previously) = [4, ptr3] -> [3, ptr4]. So, by doing 'first->next->next = first', we get [4, ptr3] -> [3, ptr2] -> [2, ptr3].

Once I can confirm this I will just **assume** that my thinking is sound, and move on to make sure the answers are correct, as there around double pointers here, so make sure that your recursive calls, you pass the address of the pointer in.

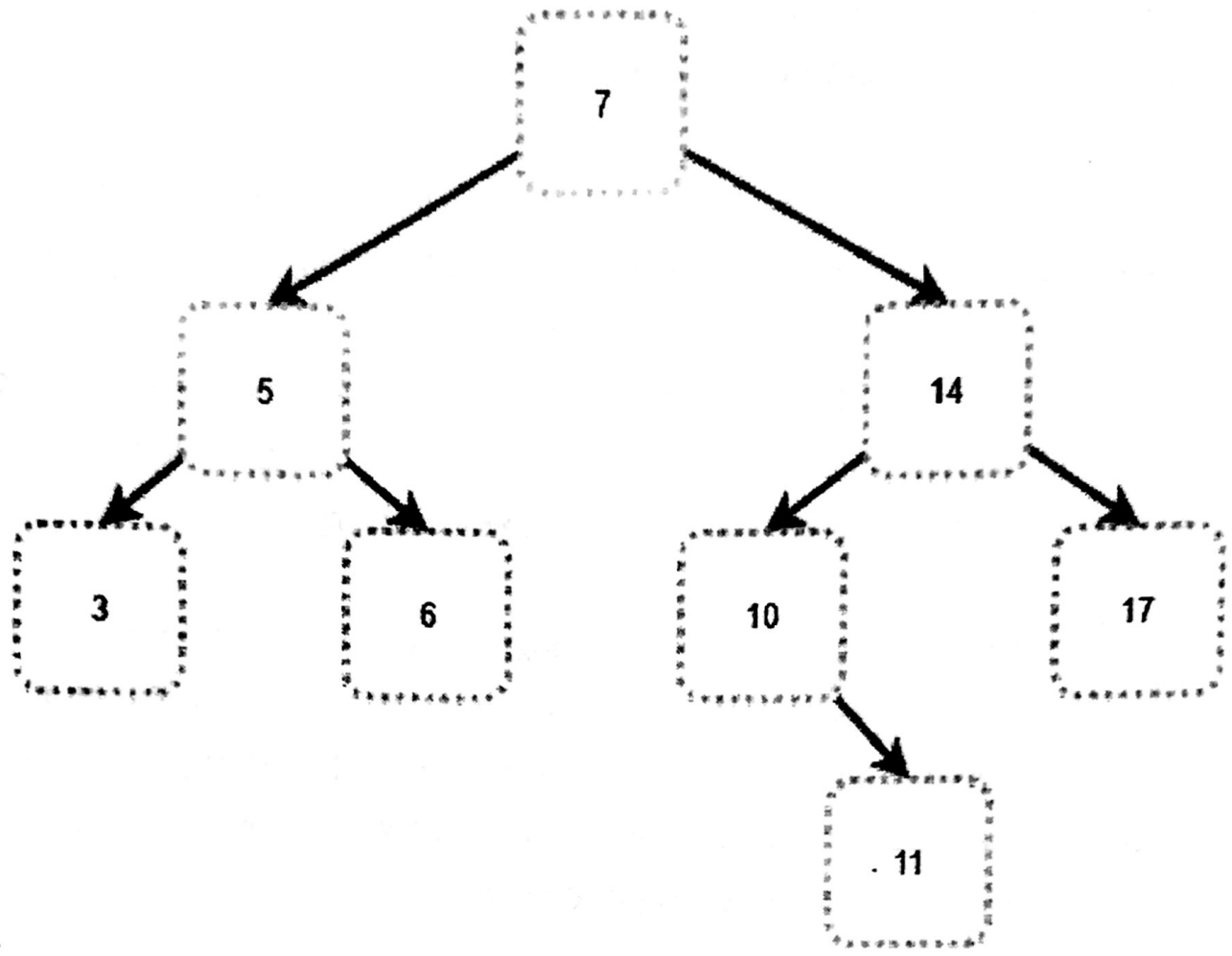
i) `rest = first->next;`
ii) `recursiveReverse(&rest);`
`*headPtr = rest;`
iii) `first->next = NULL;`

4)

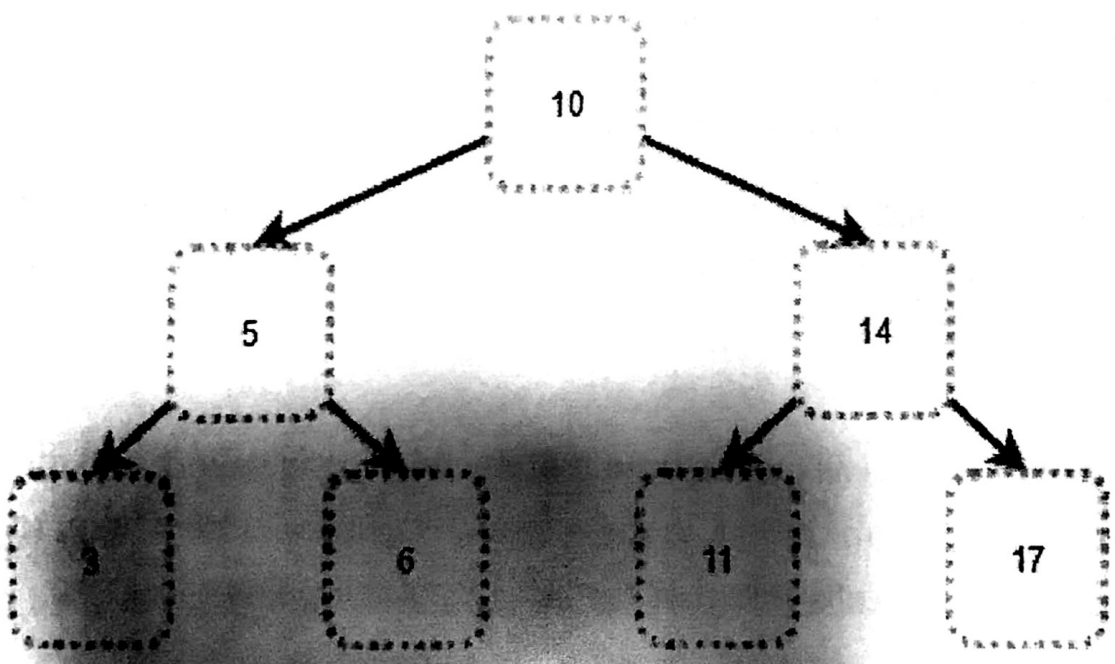
a)

i) The main property of BST states that the values in the left subtree will always be smaller than the value of the current node and all nodes to the right subtree, and the same for the right subtree inversely.

ii)




iii) Note: Following the rule in (i), find the smallest node in the right subtree, and make it root node, hence following the rule of root node bigger than all nodes on left, and smaller than all nodes on right.



b) Note: This question is pretty tricky. By the given code tail = L1 we can see that we actually don't need to copy nodes. We only need to manipulate the pointers so that the two lists are merged together. Personally the hardest part is what the return value is. It is definitely wrong that we return &temp at the end. This is because temp is a local variable. After the function returns the memory for this variable will be destroyed. Returning the address of a destroyed variable is very dangerous. With this in mind, it might be cleared that temp is just a dummy head, and what we need to return is actually temp.next. Another evidence for returning temp.next is that we always assign to tail->next. At the start we have tail = &temp, and for the first time tail->next = <something> is essentially temp.next = <something>. This <something> must be either L1 or L2, which is from outside of this function. Hence, returning temp.next is correct.
The idea for this function is to check whether we have taken all elements from L1 and L2. If one list is empty already, we directly concatenate the merged list with the remaining list. If both lists still have elements left, we append the element from L1 first followed by the element from L2.

i) `tail->next = L2;`
ii) `tail->next = L1;`
iii) `tail->next = L1;`
iv)
```
tail->next = L2;
tail = L2;
L2 = L2->next;
```
v) `return temp.next;`

c) Solving BST traversal using iteration and stack is quite troublesome as it requires a lot of trial-and-errors. It's even harder when we are limited to the given code. For in-order traversal, we need to traverse the left subtree first, followed by the root itself, and finally the right subtree. How to come back to the root once we have finished traversing the left subtree? The answer is to use stack to store the current node before moving.
The given code basically describes what the conditions we need to consider are. The outer if (current != NULL) separates the whole process into two cases: the current node is NULL or not. It should be intuitive that we need to go back to the last level (to the parent) if we encounter a NULL pointer, which is why the inner if (isEmpty(&stack)) occurs in the else case. We should also know that we need to pop the stack after checking the emptiness of the stack. Since we will be printing the data, we can also infer that the popped item will be assigned to current (after all, we cannot print the data of a NULL pointer). Now, if we treat the data we just printed as the root of a certain subtree, then the printf() functions actually visited the root node, and we should visited the right subtree next. Hence, we move to the right subtree (current = current->right).

Now, what if the current pointer is not NULL? Based on the above analysis we should push the current node to the stack. Why? Firstly we cannot print the current data in this case because this is handled in the else case. Secondly, we managed to traverse the current node and move to the right subtree in the else case as well, which implies that we still need to traverse the left subtree. With this, it should be (kind of) clear that we should save the current node and move to the left subtree. This makes sense, as in in-order traversal we will print the left-most element first. Whenever possible we should try to move to the left subtree and see if there is an element. If not, then we come back to the last level and print the parent, and continue to traverse the sibling (parent's right subtree);

i)
```
push(&stack, current);
current = current->left;
```
ii)
```
current = pop(&stack);
```
iii)
```
current = current->right;
```