Solver: XXXX

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

1.(a)

Output:  
ar[1]=0   
ar[2]=6   
ar[3]=12   
ar[4]=13   
ar[5]=18

1.(b)

#include <stdio.h>  
int rStrCompare(char\* s1, char\* s2);  
int main(int argc, char const \*argv[])  
{  
    char s1[40], s2[40];  
    gets(s1);gets(s2);  
    printf("%d", rStrCompare(s1, s2));  
    return 0;  
}  
int rStrCompare(char\* s1, char\* s2){  
    if (\*s1=='\0' && \*s2=='\0')  
    /\* missing code \*/  
        return 0;  
    // \*s1=='\0' and \*s2=='\0' are added for clearity, as  
    //  '\0' if always less that any characters  
    else if (\*s1=='\0')  
        return -1;  
    else if (\*s2=='\0')  
        return 1;  
      
    else if (\*s1>\*s2)  
        return 1;  
    else if (\*s1<\*s2)  
        return -1;  
    else  
        return rStrCompare(s1+1, s2+1);  
}

1. (c)

#include <stdio.h>  
// recommend to import <stdlib.h>  
char\* connect(char\* s1, char\* s2);  
int main(int argc, char const \*argv[]) {  
    char \*s,s1[80], s2[80];  
    gets(s1);gets(s2);  
    s = connect(s1,s2);  
    printf("%s %s %s\n", s1, s2, s);  
    malloc(1);  
    return 0;  
}  
char\* connect(char\* s1, char\* s2) {  
    char \*s,\*p,\*q;  
    int len1=0, len2=0;  
    /\* missing code \*/  
    p=s1; q=s2;  
    // get length of s1 and s2  
    while(\*p!='\0'){  
        p++;  
        len1++;  
    }  
    while(\*q!='\0'){  
        q++;  
        len2++;  
    }  
    // malloc block of memory for new string   
    s = malloc((len1+len2+1)\*sizeof(char));  
    // cpoying s1 and s2  
    while(\*s1!='\0'){  
        \*(s++)=\*(s1++);  
    }  
    while(\*s2!='\0'){  
        \*(s++)=\*(s2++);  
    }  
    // append a '\0' at the end of s  
    \*s='\0';  
    // return the pointer to the first character of the string  
    return s-(len1+len2);  
}

2.(a)

Output:  
ar[1]: 4  
ar[2]: 6  
ar[3]: 8  
ar[4]: 10  
Result: 4

2.(b)

#include <stdio.h>  
void input(char\* p);   
float avg(char\* p);  
int sentence(char\* p);  
int word(char\* p);  
int main(){  
    char str[200];  
    printf("Enter a paragraph of text: \n");  
    input(str);  
    printf("Average word/sentence: %.2f\n", avg(str));      
    return 0;  
}  
  
void input(char\* p) {  
    /\* missing code (i) \*/  
    gets(p);  
    printf("%s\n", p);  
}   
float avg(char \*p) {  
    float average;  
    /\* missing code (ii) \*/  
    average=(float)word(p)/(float)sentence(p);  
    return average;  
}  
int word(char\* p) {  
    int num=0, temp;  
    /\* missing code (iii) \*/  
    // only need to count spaces, as each word must also seperated by a space.  
    while(\*p!='\0'){  
        if(\*p==' ')  
            num++;  
        p++;  
    }  
    num++; // the last word does not have a space behind  
    printf("Word count: %d\n", num);  
    return num;  
}  
int sentence(char\* p) {  
    int num=0;  
    /\* missing code (iv) \*/  
    // only need to count '.', as each sentence must be seperated by a '.'.  
    while(\*p!='\0'){  
        if(\*p=='.')  
            num++;  
        p++;  
    }  
    printf("Sentence count: %d\n", num);  
    return num;  
}

2.(c)

#include <stdio.h>  
void rConvert(char\* str, int num);  
int main(){  
    int num;  
    char str[50];  
    scanf("%d", &num);  
    rConvert(str, num);  
    puts(str);  
    return 0;  
}  
void rConvert(char\* str, int num){  
    if ( (num/10)<=0 ) {  
        /\* missing code (i) \*/  
        \*str = num + '0';  
        \*(++str) = '\0';  
    }  
    else {  
        /\* missing code (ii) \*/  
        \*str = num%10 + '0';  
        rConvert(str+1, num/10);  
    }  
}

3.(a)

/\* missing code (i) \*/  
insertNode(ll, s, value);  
/\* missing code (ii) \*/  
current = current->next;  
index++;  
/\* missing code (iii) \*/  
current = current->next;

3.(b)

/\* missing code (i) \*/  
temp->data = currentNode->data;  
temp->next = prevNode;  
prevNode = temp;  
/\* missing code (ii) \*/  
tail->next = prevNode;

3.(c)(i) Queues follow FIFO(First in first out) principal. A single resource should follow first-come-first-serve principal, which is same as FIFO;

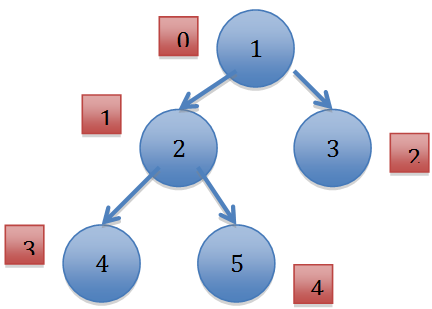
3.(c)(ii)

1. Manage function calls: Stacks follows FILO principal. Functions in programs are called one inside another, and when one function finished, I should return to the function where is was called. The above procedure can be implemented using stack operations, meaning when a function is called an item is pushed to the stack, when a function finishes a item is popped from the stack.

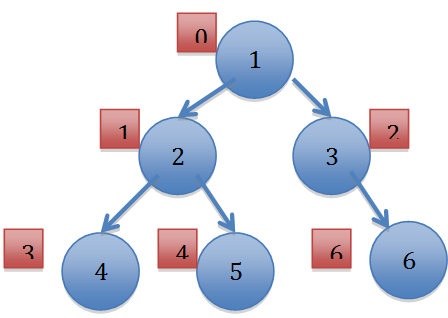
2. Used in Depth-first search algorithm: Stacks follows FILO principal. in Depth-first search algorithm is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. Using stack operations, we initially push the root node to the stack. Then, we should repeat the following procedures until the stack is empty: pop a node from the stack, push its unvisited connected nodes to the stack.

4.(a)

In a binary tree, if the parent node is assigned an index of ‘i’ and left child gets assigned an index of ‘2\*i + 1’ while the right child is assigned an index of ‘2\*i + 2’. If we represent the binary tree below with the respective indices assigned to the different nodes of the tree below are shown below:



As can be seen from the above figure, the assigned indices in case of a complete binary tree will strictly less be than the number of nodes in the complete binary tree. Below is the example of non-complete binary tree with the assigned array indices. As can be seen the assigned indices are equal to the number of nodes in the binary tree. Hence this tree is not a complete binary tree.



Reference: http://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/

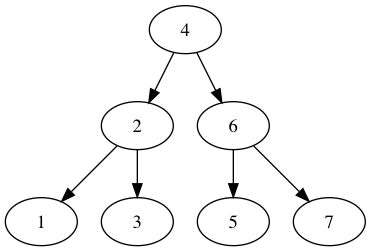
/\* missing code (i) \*/  
root->left, 2\*index + 1, number\_nodes  
/\* missing code (ii) \*/  
root->left, 2\*index + 2, number\_nodes

4.(b)

/\* missing code (i) \*/  
enqueue(q, pop(s));  
  
/\* missing code (ii) \*/  
!isEmptyQueue(q)  
  
/\* missing code (iii) \*/  
push(s, dequeue(q));

4.(c)(i) The shape like a linked list, in which each node only has zero or one child; Normally, the time complexity of searching would be O(log n), but linked lists’ searching time complexity would be O(n).

4.(c)(ii)



4.(c).(iii)

preorder: 4213657

postorder: 1325764

For reporting of errors and errata, please visit pypdiscuss.appspot.com

Thank you and all the best for your exams! ☺