資料結構 Homework#6 F74109016　葉惟欣

```c
//F74109016 葉惟欣 finished in 2021/10/20 Homework#6
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct node{
    char data;
    struct node *right_link;
    struct node *left_link;
    struct node *back;
};
typedef struct node NODE;
NODE* read_prefix(char *arr){
    NODE *first,*new_NODE;
    NODE *current =(NODE *)malloc(sizeof(NODE));
    int i=0;
    while(*(arr+i)!='\0'){
        new_NODE= (NODE *)malloc(sizeof(NODE));
        new_NODE->left_link = NULL;
        new_NODE->right_link = NULL;
        new_NODE->back= NULL;
        if(i==0){
            new_NODE->data = arr[i];
            new_NODE->back = current;
            current->left_link = new_NODE;
            current = new_NODE;
            first = current;
            i++;
        }
        else{
            if(current->left_link== NULL){
                new_NODE->data = arr[i];
                new_NODE->back = current;
                current->left_link = new_NODE;
                current = new_NODE;
                i++;
            }
            else if(current->right_link== NULL){
                new_NODE->data = arr[i];
                new_NODE->back = current;
                current->right_link = new_NODE;
                current = new_NODE;
                i++;
            }
            if(!((arr[i-1]=='+')||(arr[i-1]=='-')||(arr[i-1]=='*')||(arr[i-1]=='/')||(arr[i-1]=='^'))){
                current = current->back;
            }
        }
    }
    return first;
}
NODE* read_postfix(char *arr){
    NODE *first,*new_NODE;
    NODE *current =(NODE *)malloc(sizeof(NODE));
    int i=0;
    while(*(arr+i)!='\0'){
        new_NODE= (NODE *)malloc(sizeof(NODE));
        new_NODE->left_link = NULL;
        new_NODE->right_link = NULL;
        new_NODE->back= NULL;
        if(i==0){
            new_NODE->data = arr[i];
            new_NODE->back = current;
            current->left_link = new_NODE;
            current = new_NODE;
            first = current;
            i++;
        }
        else{
            if(current->right_link== NULL){
                new_NODE->data = arr[i];
                new_NODE->back = current;
                current->right_link = new_NODE;
                current = new_NODE;
                i++;
            }
            else if(current->left_link== NULL){
                new_NODE->data = arr[i];
                new_NODE->back = current;
                current->left_link = new_NODE;
                current = new_NODE;
                i++;
            }
            if(!((arr[i-1]=='+')||(arr[i-1]=='-')||(arr[i-1]=='*')||(arr[i-1]=='/')||(arr[i-1]=='^'))){
                current = current->back;
            }
        }
    }
    return first;
}
```

```
90  int reverse(char *str){
91      int i, j;
92      char temp;
93
94      j=strlen(str) - 1;
95      for(i=0; i<j; i++){
96          temp = str[i];
97          str[i] = str[j];
98          str[j] = temp;
99          j--;
100     }
101     return 0;
102 }
103
104 void postorder(NODE * current){
105     if(current==NULL){
106         return;
107     }
108     postorder(current->left_link);
109     postorder(current->right_link);
110     printf("%c",current->data);
111 }
112 void preorder(NODE * current){
113     if(current==NULL){
114         return;
115     }
116     printf("%c",current->data);
117     preorder(current->left_link);
118     preorder(current->right_link);
119 }
120 int main(){
121     char input [50];
122     scanf("%s",input);
123     NODE * result;
124     if((input[0]=='+')||(input[0]=='-')||(input[0]=='*')||(input[0]=='/')||(input[0]=='^')){
125         result =read_prefix(input);
126         postorder(result);
127         printf("\n");
128     }
129     else{
130         reverse(input);
131         result =read_postfix(input);
132         preorder(result);
133         printf("\n");
134     }
135 }
```

時間複雜度: 以下用 count number of steps 來計算 time complexity

先從 把 tree 生成出來的函數看起

n 為運算元加上運算子的數量，運算元的數量為(n/2)+1 與運算子的數量為 n/2

將原為 prefix 的字串生成一個 tree 的函數叫做 read_prefix()。

將原為 postfix 的字串生成一個 tree 的函數叫做 read_postfix()。

呼叫 read_prefix()與 read_postfix()的時間複雜度相同。

這裡以 read_prefix()做計算。

statement	frequency	Total_steps
NODE *first,*new_NODE;	宣告不算	
NODE *current =(NODE *)malloc(sizeof(NODE));	1(因為有 assign)	1
int i=0;	1(因為有 assign)	1
while(*(arr+i)!='\0'){	運算元的數量加上運算子的數量➔n	n
new_NODE= (NODE *)malloc(sizeof(NODE));	n	n
new_NODE->left_link = NULL;	n	n

new_NODE->right_link = NULL;	n	n
new_NODE->back= NULL;	n	n
if(i==0){	這個情況只會發生一次(第一次建立 node 時)➔1	1
new_NODE->data = arr[i];	1	1
new_NODE->back = current;	1	1
current->left_link = new_NODE;	1	1
current = new_NODE;	1	1
first = current;	1	1
i++;	1	1
}		
else{	第一次建立節點不會，其他次都會發生➔ n-1 運算子數量=L+R=(n/2)+1	n-1
if(current->left_link== NULL){	建立運算子節點時還沒有左小孩(剛建立時)➔L	L
new_NODE->data = arr[i];	L	L
new_NODE->back = current;	L	L
current->left_link = new_NODE;	L	L
current = new_NODE;	L	L
i++;	L	L
}		
else if(current->right_link== NULL){	運算子已經有左小孩(已建立運算回正在回溯)➔R	R
new_NODE->data = arr[i];	R	R
new_NODE->back = current;	R	R
current->right_link = new_NODE;	R	R
current = new_NODE;	R	R
i++;	R	R
}		
if(!((arr[i-1]=='+')||(arr[i-1]=='-')||(arr[i-1]=='*')||(arr[i-1]=='/')||(arr[i-1]=='^'))){	運算元的數量➔ n/2	n/2
current = current->back;	n/2	n/2
}		
}		
}		

此函數的時間複雜度為所有 steps 相加
9+5n+(n-1)+6L+6R+2(n/2) = 9+5n+(n-1)+6(R+L)+2(n/2) =
9+5n+6((n/2)+1)+2(n/2) = 9n+15

因此得知此生成 tree 的函數的時間複雜度 O(n)。

生成 tree 後的後序採訪與前序採訪時間複雜度相同，以下以後序採訪做計算。
n 為運算元加上運算子的數量，運算元的數量為(n/2)+1 與運算子的數量為 n/2

statement	frequency	step
if(current==NULL){	n	n
return;	這個 body 只有葉節點指向的左小孩與右小孩進得來，因為葉節點(n/2 個)沒有左小孩與右小孩，所以葉節點數量乘 2 = 2(n/2)	2(n/2)
}		
postorder(current->left_link);	n(每個節點都會跑)	n
postorder(current->right_link);	n(每個節點都會跑)	n
printf("%c",current->data);	n(每個節點都會跑)	n

n + 2(n/2) +n+n+n = 5n
此函數的時間複雜度為 O(n)

因此全部的時間複雜度為 O(n)

空間複雜度:為程式進行中會占用幾個儲存空間。

1. 主函數:
   只有宣告 char input[50]，不會因輸入的數量而空間複雜度有所不同，因此空間複雜度為 O(50)=O(1)

2. 生成 tree 的函數:

n 為運算元加上運算子的數量，運算元的數量為(n/2)+1 與運算子的數量為 n/2
將原為 prefix 的字串生成一個 tree 的函數叫做 read_prefix()。
將原為 postfix 的字串生成一個 tree 的函數叫做 read_postfix()。
呼叫 read_prefix()與 read_postfix()的空間複雜度相同。
這裡以 read_prefix()做計算。
函數在生成數的過程需要宣告

NODE * first 用來指向 tree 的 root➔O(1)
與 NODE *current 用來當作現在要生成下一個 NODE 的參考➔O(1)
而每遇到新的運算元或運算子就會產生一個新的 NODE➔O(n)
且函數為了避免讀到使用者輸入外的元素用變數 int i 來跑迴圈➔O(1)
每個 NODE 結構需要用到空間包含 int data 與 NODE * left_link 與 reight_link 與 back。因此空間複雜度為 O(4)
因此呼叫整個函數用 (1+1+n)*4+1 = 4n+9。
最後得出生成 tree 的函數空間複雜度為 O(n)

3. 採訪 tree 的函數:
生成 tree 後的後序採訪與前序採訪空間複雜度相同，以下以後序採訪做計算。n 為運算元加上運算子的數量，運算元的數量為(n/2)+1 與運算子的數量為 n/2 空間複雜度一定會呼叫 n 次函數，如此才可以印出採訪的節點，因此空間複雜度可看做 O(n)。而當採訪到 leaf node 的時候左小孩與右小孩為 NULL，再次呼叫函數後會 return，不會再採訪下去，但因為每個葉節點呼叫兩次所以空間複雜度為 O((n/2)+1)*2) = O(n)。而 O(n)+O(n)=O(2n)=O(n)。

採訪 tree 的函數空間複雜度為 O(n)。

因此全部的空間複雜度為 O(n)。