作业 3

第 4 章：串

龚舒凯 2022202790 应用经济-数据科学实验班

2023 年 10 月 17 日

1. 假设以结点大小为 1（且附设头结点）的链表结构表示串。试编写实现下列六种串的基本操作 StrAssign, StrCopy, StrCompare, StrLength, Concat 和 SubString 的函数。

解.

串赋值 StrAssign、串复制 Strcopy、串比较 StrCompare、求串长 StrLength、串联接 Concat 以及求子串 SubString 等六种操作构成串类型的最小操作子集。六种基本操作的函数定义如下：

- `LinkString* StrAssign(char *str)`：将字符串`char *str`的值赋给链串。
- `void StrCopy(LinkString *s, LinkString *t)`：将链串`t`的值拷贝给链串`s`。
- `int StrCompare(LinkString *s, LinkString *t)`：比较链串`s`和`t`，如果两串相等则返回 1，否则返回 0。
- `int StrLength(LinkString *s)`：返回链串`s`的长度。
- `void Concat(LinkString *s, LinkString *t)`：把链串`t`接到链串`s`之后。
- `void SubString(LinkString *sub, LinkString *text, int pos, int len)`：将串`text`中第`pos`位置起长度为`len`的子串存到串`sub`中。

时间复杂度分析：StrAssign 时间复杂度 $O(\text{strlen(str)})$，StrCopy 时间复杂度 $O(\text{strlen(t)})$，StrCompare 时间复杂度 $O(\min\{\text{strlen}(s), \text{strlen}(t)\})$，StrLength 时间复杂度 $O(\text{strlen}(s))$，Concat 时间复杂度 $O(\text{strlen}(s))$，SubString 时间复杂度 $O(\max\{\text{pos}, \text{len}\})$

空间复杂度分析：StrAssign 空间复杂度 $O(\text{strlen(str)})$，StrCopy 空间复杂度 $O(\text{strlen(t)})$，StrCompare 空间复杂度 $O(1)$（用了两个指针逐一比较字符），StrLength 空间复杂度 $O(1)$（用一个指针遍历链表，一个变量储存长度），Concat 空间复杂度 $O(1)$（不涉及新结点的开辟），SubString 空间复杂度 $O(\text{len})$

```
typedef struct LinkString{
        char c;
        struct LinkString *next;
}LinkString;

LinkString* StrAssign(char *str){
        LinkString *Mystring = new LinkString;
        Mystring->c = '\0';
        Mystring->next = NULL;
        LinkString *p = Mystring;
        for (int i = 0 ; i < strlen(str) ; i++){
                LinkString *node = new LinkString;
                node->c = str[i];
                node->next = NULL;
```

```
                p->next = node;
                p = node;
        }
        return Mystring;
}

void StrCopy(LinkString *s, LinkString *t){//把串 t 拷贝给串 s
        LinkString *p = t->next;
        LinkString *q = s;
        while (p != NULL){
                LinkString *node = new LinkString;
                node->c = p->c;
                node->next = NULL;
                q->next = node;
                p = p->next;
                q = q->next;
        }
}

int StrCompare(LinkString *s, LinkString *t){//比较串 s 和串 t
        LinkString *p = t->next;
        LinkString *q = s->next;
        while (p != NULL && q != NULL){
                if (p->c != q->c) return 0;//两串不相等
                p = p->next;
                q = q->next;
        }
        return 1;//两串相等
}

int StrLength(LinkString *s){//返回串的长度
        int len = 0;
        LinkString *p = s->next;
        while (p != NULL){
                len++;
                p = p->next;
        }
        return len;
}

void Concat(LinkString *s, LinkString *t){//把串 t 接到串 s 之后
```

```
        LinkString *p = s;
        LinkString *q = t;
        while (p->next != NULL){
                p = p->next;
        }//找到串 s 的末尾位置
        p->next = q->next;//把串 t 接到串 s 之后
        delete q;//删除串 t 的头结点
}

void SubString(LinkString *sub, LinkString *text, int pos, int len){
        //将串 text 中第 pos 位置起长度为 len 的子串存到串 sub 中
        LinkString *p = sub;
        LinkString *q = text;
        for (int i = 0 ; i <= pos ; i++){
                q = q->next;
        }//找到串 text 第 pos 位置
        for (int i = 0 ; i < len ; i++){
                LinkString *node = new LinkString;
                node->c = q->c;
                node->next = NULL;
                p->next = node;
                p = node;
                q = q->next;
        }
}
```

2. 假设以块链结构作为串的存储结构。试编写判别给定串是否具有对称性的算法，并要求算法的时间复杂度为O(StrLength(S))。

解.

将块链串的前一半的元素入栈，后一半的元素每次和栈顶元素比对。如果不相等就直接返回 false，表明不对称，如果相等就弹出栈顶元素，进行下一次对比，直到串的后一半元素比对完，说明该块链串是对称的。入栈块链串元素时唯一要注意的是是否一个块中的字符已经遍历完了，指针需要移到块链串的下一个块。

时间复杂度分析：从长度为 StrLength(S) 的块链串需要完全遍历一遍，一半元素入栈一半元素比对，时间复杂度为 $O(\text{StrLength(S)})$。
空间复杂度分析：使用了一个栈来存放块链串的一半元素，空间复杂度为 $O(\frac{1}{2}\text{StrLength(S)})$。

```
typedef struct Chunk{
        char ch[CHUNKSIZE];//每个结点含有 NodeSize 个字符
        struct StrNode* prior,* next;
}Chunk;

typedef struct{
        Chunk *head, *tail;
        int curlen;
}ChunkString;

bool IsSymmetric_ChunkString(ChunkString *str){
        int idx = 0;
        Chunk *p = str->head->next;
        char top;
        for (int i = 0 ; i < str->curlen ; i++){
                if (idx == CHUNKSIZE){
                        p = p->next;
                        idx = 0;
                }
                if (i < str->curlen / 2){
                        temp.push(p->ch[idx]);
                        idx++;
                }
                else if (i > (str->curlen - 1) / 2){
                        top = temp.top();
                        temp.pop();
                        if (top != p->ch[idx]) return false;
                        idx++;
```

```
                }
            }
            return true;
    }
```

3. 试写一算法，实现堆存储（即动态顺序存储）结构的串的置换操作 Replace(&S, T, V)。

解.

创建一个空串 U 用于存储替换后的结果。遍历堆中的串 S，将当前字符与待替换的串 T 进行比较：

1. 如果当前字符与 T 的第一个字符相同，进行进一步比较：
   - 如果当前字符与 T 中的字符序列完全匹配，将替换串 V 的内容复制到 U 中，并更新 U 的长度。
   - 否则字符与待替换的串 T 不匹配，继续遍历下一个字符。
2. 如果当前字符与 T 的第一个字符不相同，则从 S 的下一个字符开始匹配。

时间复杂度分析：遍历堆中的串 S 的时间复杂度为 $O(n)$，其中 n 是串 S 的长度。替换操作的时间复杂度为 $O(m)$，其中 m 是待替换的串 T 的长度。复制替换串 V 到结果串 U 的时间复杂度为 $O(m)$，复制 U 到 S 的时间复杂度为 $O(m+n)$。故整个算法的时间复杂度为 $O(n+m)$。

空间复杂度分析：堆中串 S 所需的额外空间为 $O(n)$，其中 n 是串 S 的长度。待替换的串 T 所需的额外空间为 $O(m)$，其中 m 是待替换的串 T 的长度。故整个算法的空间复杂度为 $O(n+m)$。

```
void Replace(HString *S, char *T, char *V) {
        int tLen = strlen(T);
        int vLen = strlen(V);
        int sLen = S->length;

        // 遍历堆中的串 S
        for (int i = 0; i < sLen; ) {
                // 检查当前字符与待替换的串 T 是否匹配
                if (strncmp(S->ch + i, T, tLen) == 0) {
                        // 匹配成功，将替换串 V 复制到结果串 U 中
                        char* U = new char[sLen - tLen + vLen + 1];
                        strncpy(U, S->ch, i);
                        strncpy(U + i, V, vLen);
                        strcpy(U + i + vLen, S->ch + i + tLen);
                        // 更新堆中的串 S
                        delete[] S->ch;
                        S->ch = U;
                        S->length = sLen - tLen + vLen;
                        sLen = S->length;
                        i += vLen;
                        delete[] U;
                }
```

```
                else {
                        // 当前字符与待替换的串 T 不匹配，继续遍历下一个字符
                        i++;
                }
        }
}
```

4. 假设以结点大小为 1（带头结点）的链表结构表示串，则在利用 next 函数值进行串匹配时，在每个结点中需要设三个域：数据域 chdata、指针域 succ 和指针域 next。其中 chdata 域存放一个字符；succ 域存放指向同一链表中后继结点的指针；next 域在主串中存放指向同一链表中前驱结点的指针；在模式串中存放指向当该结点的字符与主串中的字符不等时，模式串中下一个应进行比较的字符结点（即与该结点字符的 next 函数值相对应的字符结点）的指针，若该结点字符的 next 函数值为零，则其 next 域的值应指向头结点。试按上述定义的结构改写求模式串的 next 函数值的算法。

解.

顺序储存串和链串求解 next 数组唯一的区别就是主串模式串模式匹配时移动下标的方式改为了`p = p->succ;`，模式串回溯的方式改为了`p = p->next;`。next 数组的求算方式与顺序储存串的方式相同，代码如下所示：

```
typedef struct LNode {
        char chdata;
        struct LNode* succ;
        struct LNode* next;
}LString;

void GetNext(LString *P) {
        LString *p = P->next;
        LString *q = P;
        int k = 0;
        int j = -1;
        p->next = P;

        while (p) {
                if (j == -1 || p->chdata == q->chdata) {
                        p = p->succ;
                        q = q->succ;
                        j++;
                        k++;
                        p->next = q;
                } else {
                        q = P;
                        j = 0;
                }
        }
}
```

5. 假设以定长顺序存储结构表示串，试设计一个算法，求串 s 中出现的第一个最长重复子串及其位置，并分析你的算法的时间复杂度。

解.

为了在串 str 中找最大子串，从串 str 的第 k 个字符开始 ($k = 0, 1, 2..., \text{strlen(str)}$)，后面的部分作为 KMP 的主串，开始和自身进行模式匹配，寻找重复部分。每次找到一个重复部分就把重复部分的起始位置和重复部分长度记到 loc 和 max 中，随着 k 的变化不断更新 loc, max，最终根据 loc 和 max 输出子串 sub 的开始位置和子串 sub。

时间复杂度分析：设串 s 的长度为 n。该算法对从串 s 的每个字符开始，对后面的剩余部分做 KMP 模式匹配。一次 KMP 的时间复杂度为 $O(m+n)$，因此此处算法的时间复杂度为 $O(n^2)$。
空间复杂度分析：用到了一个和串 s 长度相同的 next 数组和比串 s 长度小的 sub 数组，空间复杂度为 $O(n)$。

```
void CommonStr(char *str, char *sub, int &loc){
//求串 str 中出现的第一个最长重复子串 sub 及其位置 loc
//KMP, 时间复杂度为 O(n^2)
        int next[50];
        int i,j;
        int len = 0;
        int max = 0;
        for(int k = 0; k < strlen(str); ++k){
                //这里为了找最长重复字串，必须以 str 每个字符为起点开始模式匹配
                //从串 str 的第 k 个字符开始，后面的部分作为 KMP 的主串，开始模式匹配
                i = k;
                j = k - 1;
                next[k] = k - 1;//赋 next[0] 初值
                while(i < strlen(str)){
                        if(j == k - 1 || str[i] == str[j]){//模式匹配
                                ++i;
                                ++j;
                                next[i] = j;
                                if(str[i] == str[j]) len = j - k + 1;
                                //如果模式匹配成功 len+1
                                else len = j - k;
                                //否则保持 len 不变
                        }
                        else j = next[j];//模式匹配失败，模式串向右移
                        if(len > max){//不断更新重复字串的 max
                                loc = k;    //记录出现第一个最长重复子串的位置
                                max = len;  //记录最长重复子串的长度
```

```
                        }
                }
        }
        //把最长重复字串储存在 sub[] 中
        for(i = loc, j = 0; i < max + loc; ++i, ++j) sub[j] = str[i];
        sub[j] = '\0';
        //打印最长重复字串的起始位置 loc 和最长重复字串 sub
        std::cout << " 最长重复字串起始位置: " << loc << std::endl;
        for (int i = 0 ; i < strlen(sub) ; i++){
                std::cout << sub[i];
        }
        std::cout << std::endl;
}
```