作业4 字符串

姓名：叶栩言

学号：2023200033

1. 基于 KMP 算法求 `next` 数组

模式串 `p1 = "abaabaa"`

i（位置）	字符	子串 p[0..i]	最长相等前后缀	next[i]
0	a	a	无	0
1	b	ab	无	0
2	a	aba	a	1
3	a	abaa	无	1 → 现在和前面最长相等前后缀不吻合，只回退
4	b	abaab	ab	2
5	a	abaaba	aba	3
6	a	abaabaa	a	1

故`p1` = `abaabaa`, `next` = `[0, 0, 1, 1, 2, 3, 1]`

模式串 `p2 = "aabbaab"`

i	字符	子串	最长相等前后缀	next[i]
0	a	a	无	0
1	a	aa	a	1
2	b	aab	无	0
3	b	aabb	b	1
4	b	aabbb	bb	2
5	a	aabbba	无	0
6	a	aabbbaa	a	1
7	b	aabbbaab	无	0

故`p2` = `aabbbaab`, `next` = `[0, 1, 0, 1, 2, 0, 1, 0]`

2. 块链结构串的对称性判别

遍历块链，将字符依次读入一个辅助数组 A[0..n-1]，然后对数组使用双指针判断回文。

伪代码

```
bool isSymmetric(LString S) {
    char *A = malloc(S.length);
    int k = 0;
    Chunk *p = S.head;
    while (p != NULL) {
        for (int i = 0; i < CHUNK_SIZE && k < S.length; i++) {
            A[k++] = p->ch[i];
        }
        p = p->next;
    }

    int left = 0, right = S.length - 1;
    while (left < right) {
        if (A[left] != A[right]) {
            free(A);
            return false;
        }
        left++;
        right--;
    }

    free(A);
    return true;
}
```

复杂度分析

操作	复杂度
遍历块链复制字符串	O(n)
回文双指针扫描	O(n)

满足题目要求 O(StrLength(S))

3. 堆存储串的置换 Replace(&S, T, V)

目标是把 S 中出现的子串 T 都替换成 V。任务包含两件事：先用 KMP 定位所有匹配，再根据长度差异做动态扩容或收缩。扫描 S 时用两个指针 i 和 j，一旦 j 达到 T.length，匹配区间就是 pos = i - T.length。这时算好长度差 diff，必要时扩容，把尾部字符手动搬移，再把 V 写入该位置，更新串长，并把 i 拉回替换段结尾以便继续向后。失配时直接按 next[j] 回退。

伪代码

```
int* BuildNext(const HString *T) {
    int *next = (int*)malloc(sizeof(int) * T->length);
```

```
    int i = 0, j = -1;
    next[0] = -1;
    while (i < T->length - 1) {
        if (j == -1 || T->ch[i] == T->ch[j]) {
            ++i;
            ++j;
            if (T->ch[i] != T->ch[j]) {
                next[i] = j;
            } else {
                next[i] = next[j];
            }
        } else {
            j = next[j];
        }
    }
    return next;
}

void EnsureCapacity(HString *S, int need) {
    if (need <= S->capacity) return;
    int newCap = S->capacity ? S->capacity : 1;
    while (newCap < need) newCap <<= 1;
    char *newBuf = (char*)malloc(newCap + 1);
    int i;
    for (i = 0; i < S->length; ++i) newBuf[i] = S->ch[i];
    newBuf[S->length] = '\0';
    free(S->ch);
    S->ch = newBuf;
    S->capacity = newCap;
}

void Replace(HString *S, const HString *T, const HString *V) {
    if (!T->length) return;
    int *next = BuildNext(T);
    int i = 0, j = 0;
    while (i < S->length) {
        if (j == -1 || S->ch[i] == T->ch[j]) {
            ++i;
            ++j;
        } else {
            j = next[j];
        }
        if (j == T->length) {
            int pos = i - T->length;
            int diff = V->length - T->length;
            int tailLen = S->length - pos - T->length;
            if (diff > 0) EnsureCapacity(S, S->length + diff);
            if (diff != 0) {
                int src = pos + T->length + tailLen - 1;
                int dst = pos + V->length + tailLen - 1;
                while (tailLen-- > 0) {
                    S->ch[dst--] = S->ch[src--];
                }
            }
```

```
            int k;
            for (k = 0; k < V->length; ++k) {
                S->ch[pos + k] = V->ch[k];
            }
            S->length += diff;
            S->ch[S->length] = '\0';
            i = pos + V->length;
            j = 0;
        }
    }
    free(next);
}
```

时间复杂度 `O(StrLength(S) + StrLength(T))`；空间复杂度 `O(StrLength(T))`

4. 链式串的 next 函数构造

先把模式串的结点按顺序收集到数组 `nodes[]`，然后直接在这个数组上跑KMP，得到整数型失配表 `fail[]`。再把 `fail[i]` 转成真正的结点指针，`-1` 的情况统一改指向头结点。这样既保持线性时间，也避免在链表里来回跳。

伪代码

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

void BuildNext(LinkStr head) {
    int n = 0;
    LNode *p = head->succ;
    while (p) {
        ++n;
        p = p->succ;
    }
    if (n == 0) return;

    LNode **nodes = (LNode**)malloc(sizeof(LNode*) * n);
    int idx = 0;
    for (p = head->succ; p; p = p->succ) {
        nodes[idx++] = p;
    }

    int *fail = (int*)malloc(sizeof(int) * n);
    fail[0] = -1;
    int i = 0, j = -1;
    while (i < n - 1) {
        if (j == -1 || nodes[i]->chdata == nodes[j]->chdata) {
            ++i;
            ++j;
```

```
            if (nodes[i]->chdata != nodes[j]->chdata) {
                fail[i] = j;
            } else {
                fail[i] = fail[j];
            }
        } else {
            j = fail[j];
        }
    }

    for (idx = 0; idx < n; ++idx) {
        nodes[idx]->next = (fail[idx] == -1) ? head : nodes[fail[idx]];
    }
    free(nodes);
    free(fail);
}
```

复杂度

`nodes` 中的顺序即原串顺序，因此对 `fail` 的维护与数组版 KMP 等价。将 `fail[idx]` 映射为指针即可满足提议。时间复杂度 `O(n)`。

5. 固定顺序存储串中第一个最长重复子串

由于字符串采用定长顺序存储，可以枚举起点 + 最长公共前缀匹配的方法求解最长重复子串。设字符串为 `S[0..n-1]`，枚举两个起始位置 `i` 和`j`，将它们作为两个可能的重复子串起点，并从这两个位置开始向后逐字符比较，统计它们的最长公共前缀长度 `k`。在所有 `(i, j)` 所对应的比较结果中，最长的公共前缀就是最长重复子串。

伪代码

```
void LongestRepeatSubstring(char S[], int n, int *pos, int *length) {
    int i, j, k;
    *length = 0;
    *pos = -1;

    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            k = 0;
            while (i + k < n && j + k < n && S[i + k] == S[j + k]) {
                k++;
            }
            if (k > *length) {
                *length = k;
                *pos = i;
            }
        }
    }
}
```

时间复杂度分析

算法通过两层循环枚举所有 `(i, j)` 组合，共 `O(n^2)` 次，每次最长公共前缀比较在最坏情况下为 `O(n)`，因此算法时间复杂度为：`O(n^2)` × `O(n)` = `O(n^3)`