第3章 串与数组

3.1 基础实践

3.1.1 串的堆分配存储表示上的基本操作

1. 实践目的

(1) 能够正确描述串的堆分配存储在计算机中的表示。

(2) 能够正确编写堆分配存储表示上基本操作的实现算法。

(3) 能够编写程序验证在堆分配存储表示上实现的基本操作算法的正确性。

2. 实践内容

(1) 创建串的堆分配存储表示。

(2) 在串的堆分配存储表示上实现串初始化、串赋值、求串长、串比较、串连接、求子串、串删除和串插入等操作。

3. 实践要求

1) 数据结构

串的堆分配存储表示的描述如下：

```
typedef struct
{     char *ch;            //若是非空串,则按串长分配存储区,否则 ch 为 NULL
      int length;          //串长
}HString;
```

2) 函数接口说明

```
void StrInit(HString& S);
//创建一个空串 S
Status StrAssign(HString& S, const char chars[]);
//生成一个其值等于 chars 的串 S
int StrLength(HString S);
//返回串 S 的长度
int StrCompare(HString S, HString T);
//比较串 S 和 T 的大小,如果 S = T 则返回 0; 如果串 S > T 则返回正值; 如果 S < T 则返回负值
Status Concat(HString& T, HString S1, HString S2);
//用 T 返回由 S1 和 S2 连接而成的新串
Status SubString(HString &Sub, HString S, int pos, int len);
//用 Sub 返回串 S 第 pos 个字符起长度为 len 的子串。其中: 1≤pos≤S.length 且 0≤len≤S.
//length - pos + 1
Status StrDelete(HString &S, int pos, int len);
```

```
//从串 S 中删除第 pos 个字符起长度为 len 的子串。其中: 1≤pos≤S.length - len + 1
Status StrInsert(HString &S, int pos, HString T);
//在串 S 中第 pos 个字符之前插入串 T。其中: 1≤pos≤S.length + 1
```


3) 输入输出说明

输入说明:输入信息分 7 行,第 1 行输入串 S1;第 2 行输入串 S2;第 3 行输入求子串的起始位置 pos(1≤pos≤S.length);第 4 行输入求子串的长度 len(0≤len≤S.length−pos+1);第 5 行输入待删除子串的起始位置 pos(1≤pos≤S.length−len+1);第 6 行输入待删除子串的长度 len;第 7 行输入待插入子串的位置 pos(1≤pos≤S.length+1)。

输出说明:输出信息有 6 行。在输入完第 1、2 行信息并回车后,在输出的第 1 行输出 S1 和 S2 两个串的比较结果;第 2 行输出 S1 和 S2 连接而成的新串,第 3 行输出新串的长度;在输入完第 3、4 行信息并回车后,在输出的第 4 行输出求得的子串,如果求子串失败则显示“SubString failed!”。在输入完第 5、6 行信息并回车后,在输出的第 5 行输出删除后剩余的串,如果删除失败则显示“Deletion failed!”。在输入完第 7 行后,在输出的第 6 行输出完成插入后的串,如果插入失败则显示“Insertion failed!”。

4) 测试用例

测试用例如表 3-1 所示。

表 3-1 测试用例

序号	输　入	输　出	说　明
1	data structure 5 6 4 9 5	S1<S2 datastructure 13 struct date datestructure	一切输入的参数都在合法范围之内
2	hello world 11 2 3 6 1	S1<S2 hello world 10 SubString failed! held worldheld	子串的位置不合法,其他输入的参数都合法
3	Zhe Jiang 3 7 4 3 6	S1>S2 ZheJiang 8 SubString failed! Zheng ZhengJiang	取子串长度不合法,其他输入的参数都合法

续表

序号	输　入	输　出	说　明
4	sun flower 4 4 10 3 4	S1＞S2 sunflower 9 flow Deletion failed! sunflowerflower	删除位置不合法，其他输入的参数都合法
5	swan lake 1 4 2 4 －2	S1＞S2 swanlake 8 swan sake Insertion failed!	插入位置不合法，其他输入的参数都合法

4. 解决方案

上述接口的实现方法分析如下。

(1) 串初始化操作 StrInit(&S)：即构造一个空串 S。根据堆分配存储表示的定义可知，空串满足的条件是指向存储空间的指针 ch 值为 NULL，且串长为 0。

(2) 串赋值操作 StrAssign(&S,chars[])：根据给定的字符数组(字符串)chars 构造一各堆分配存储表示的串 S。首先，如果串 S 不为空串，则需先释放原有空间；其次，根据给定的字符数组 chars 的大小进行存储空间的分配，如果分配失败，则本操作失败；如果分配成功，则将字符数组 chars 中的内容逐一复制至分配的存储空间；最后设置串长。

(3) 求串长操作 StrLength(S)：返回串的堆分配存储表示中串长数据域的值。

(4) 串比较操作 StrCompare(S, T)：按字典序依次比较给定的两个字符串 S 和 T 对应位置上字符的大小，直到遇到不相同字符，则返回不相同字符对应的 ASCII 码值的差值；如果其中一个字符串提前结束，则返回两串长的差值。即如果串 S＞T，则返回正值；如果 S＝T，则返回 0；如果 S＜T 则返回负值。

(5) 串连接操作 Concat(&T, S1, S2)：首先，如果串 T 不为空串，则需先释放原有空间；其次，根据给定的串 S1 和 S2 的串长进行存储空间的分配，如果分配失败，则本操作失败；如果分配成功，则设置串长为串 S1 和 S2 的串长之和，再将串 S1 的内容逐一复制至串 T 的相同位置，最后将串 S2 的内容逐一复制至串 T 刚才复制的内容之后。

(6) 求子串操作 SubString(&Sub, S, pos, len)：首先根据求子串的起始位置 pos 和串长判断操作参数是否合法，即要求 1≤pos≤S.length 且 0≤len≤S.length－pos＋1；其次，根据子串的长度为子串分配存储空间；再将主串中字符(位置从 pos 到 pos＋len－1)逐一复制到子串的存储空间；最后，设置子串串长为 len。

(7) 串删除操作 StrDelete(&S, pos, len)：首先根据要删除子串的起始位置和串长判断操作参数是否合法，即要求 1≤pos≤S.length－len＋1；其次，依次将删除子串后面的字

符(位置从 pos－1＋len 到 S.length－1)依次前移 len 个位置；最后，修正串 S 的长度为 S.length－len.

(8) 串插入操作 StrInsert(&S, pos, T)：在串 S 中第 pos 个字符之前插入串 T。首先判断插入位置 pos 是否合法，要求 1≤pos≤S.length＋1；如果合法，则为了能另外存储串 T 需增加分配存储空间的容量为 S.length＋T.length；再将串 S 的第 pos 个之后的字符后移 T.Length 个位置后，将串 T 插入至刚才空出的位置；最后，修正串 S 的长度为 S.length＋T.length。

5. 程序代码参考

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define ERROR 0
#define OK 1
#define OVERFLOW -2
typedef int Status;

typedef struct{
    char* ch;                    //若是非空串,则按串长分配存储区,否则 ch 为 NULL
    int length;                  //串长
}HString;

void StrInit(HString& S)
//创建一个空串
{   S.ch = NULL;
    S.length = 0;
}

Status StrAssign(HString& S, const char chars[])
//生成一个其值等于 chars 的串 S
{   if (S.ch)
          free(S.ch);            //释放旧空间
    int len = strlen(chars);
    if (!(S.ch = (char*)malloc(len * sizeof(char))))
          exit(OVERFLOW);
    for (int i = 0; i < len; i++)
          S.ch[i] = chars[i];
    S.length = len;              //设置子串长度
    return OK;
}

int StrLength(HString S)
//返回串 S 的长度
{   return S.length;
}

int StrCompare(HString S, HString T)
```

```
//比较串 S 和 T 的大小，如果 S = T,则返回 0;如果串 S > T,则返回正值;如果 S < T,则返回负值
{   for (int i = 0; i <= S.length && i <= T.length; i++)
        if (S.ch[i] != T.ch[i])
              return S.ch[i] - T.ch[i];
    return S.length - T.length;
}

Status Concat(HString& T, HString S1, HString S2)
//用 T 返回由 S1 和 S2 连接而成的新串
{   if (T.ch)
          free(T.ch);                          //释放旧空间
    if (!(T.ch = (char *)malloc((S1.length + S2.length) * sizeof(char))))
          exit(OVERFLOW);
    T.length = S1.length + S2.length;
    for (int i = 0; i < S1.length; i++)
          T.ch[i] = S1.ch[i];
    for (int i = 0; i < S2.length; i++)
          T.ch[S1.length + i] = S2.ch[i];
    return OK;
}

Status SubString(HString& Sub, HString S, int pos, int len)
//用 Sub 返回串 S 第 pos 个字符起长度为 len 的子串.
//其中：1≤pos≤S.length 且 0≤len≤S.length - pos + 1
{   if (pos < 1 || pos > S.length || len < 0 || len > S.length - pos + 1) //参数合法性检查
          return ERROR;
    if (Sub.ch)
          free(Sub.ch);
    if (!(Sub.ch = (char *)malloc(len * sizeof(char))))
          exit(OVERFLOW);
    for (int i = 0; i < len; i++)                //将主串中的字符复制到子串的相应位置
          Sub.ch[i] = S.ch[pos - 1 + i];
    Sub.length = len;                            //设置子串长度
    return OK;
}

Status StrDelete(HString& S, int pos, int len)
//从串 S 中删除第 pos 个字符起长度为 len 的子串.其中：1≤pos≤S.length - len + 1
{   if (pos < 1 || pos > S.length - len + 1 || len < 0)
          return ERROR;
    for (int i = pos - 1 + len; i < S.length; i++)
          S.ch[i - len] = S.ch[i];
    S.length -= len;
    return OK;
}

Status StrInsert(HString& S, int pos, HString T)
//在串 S 中第 pos 个字符之前插入串 T.其中：1≤pos≤S.length + 1
{   if (pos < 1 || pos > S.length + 1)
```

```
        return ERROR;
    int i;
    char * p;
    p = S.ch;
    if (!(S.ch = (char * )malloc((S.length + T.length) * sizeof(char))))
        exit(OVERFLOW);
    for (i = 0; i < pos - 1; i++)                    //将插入点之前的子串复制到新的位置
        S.ch[i] = p[i];
    for (i = S.length - 1; i >= pos - 1; i--)        //为插入串 T 而腾出位置
        S.ch[i + T.length] = p[i];
    free(p);
    for (i = 0; i < T.length; i++)                   //插入 T
        S.ch[pos - 1 + i] = T.ch[i];
    S.length += T.length;
    return OK;
}

void StrPrint(HString S)
//输出字符串 S
{   for (int i = 0; i < S.length; i++)
        putchar(S.ch[i]);
}

int main()
{   char s1[100], s2[100];
    int pos, len;
    HString S1, S2, S, Sub;
    StrInit(S1); StrInit(S2); StrInit(S); StrInit(Sub);
    printf("输入串 S1: ");
    scanf("%s", s1);
    StrAssign(S1, s1);
    printf("输入串 S2: ");
    scanf("%s", s2);
    StrAssign(S2, s2);

    printf("\n 两个串比较: ");
    if (StrCompare(S1, S2)< 0)
        printf("S1 < S2");
    else if (StrCompare(S1, S2) == 0)
        printf("S1 = S2");
    else
        printf("S1 > S2");

    Concat(S, S1, S2);
    printf("\n\n 由 S1 和 S2 连接而成的新串 S: ");
    StrPrint(S);
    printf("\n 新串长: ");
    printf("%d", StrLength(S));
```

```
    printf("\n\n 输入子串在 S 的起始位置: ");
    scanf("%d", &pos);
    printf("输入子串长: ");
    scanf("%d", &len);
    if (SubString(Sub, S, pos, len))
    {   printf("子串为: ");
        StrPrint(Sub);
    }
    else
        printf("SubString failed!");

    printf("\n\n 输入删除串 S 的起始位置: ");
    scanf("%d", &pos);
    printf("输入删除子串的长度: ");
    scanf("%d", &len);
    if (StrDelete(S, pos, len))
    {   printf("删除后的串 S 为: ");
        StrPrint(S);
    }
    else
        printf("Deletion failed!\n");

    printf("\n\n 输入在串 S 中插入串 S2 的位置: ");
    scanf("%d", &pos);
    if (StrInsert(S, pos, S2))
    {   printf("插入后的串 S 为: ");
        StrPrint(S);
    }
    else
        printf("Insertion failed!\n");
}
```

6. 运行结果参考

程序部分测试用例的运行结果如图 3-1 所示。

(a) 测试用例1　　(b) 测试用例2　　(c) 其他

图 3-1　程序部分测试用例的运行结果

7. 延伸思考

(1) 在串的操作中,串赋值 StrAssign、求串长 StrLength、串比较 StrCompare、串连接

Concat 以及求子串 SubString 这 5 种操作构成串类型的最小操作子集,其他串操作可在这个最小操作子集上实现,那么如何利用这些最小操作子集实现串的删除操作?

(2) 根据上述已实现的串的基本操作,编写算法模仿实现串的置换操作 StrReplace(&S,T,V),即以串 V 替代串 S 中出现的所有和串 T 相同的子串。

3.1.2 稀疏矩阵的转置操作

1. 实践目的

(1) 能够正确描述稀疏矩阵的三元组顺序表在计算机中的表示。

(2) 能够正确编写在稀疏矩阵的三元组顺序表上基本操作的实现算法。

(3) 能够编写程序验证在稀疏矩阵的三元组顺序表上实现的基本操作算法的正确性。

2. 实践内容

(1) 创建已知稀疏矩阵的三元组顺序表存储表示。

(2) 在稀疏矩阵的三元组顺序表上实现矩阵转置操作。

3. 实践要求

1) 数据结构

在稀疏矩阵的三元组顺序表上实现指定操作,先要将稀疏矩阵的三元组顺序表中每一非零元的存储结构描述如下:

```
#define MAXSIZE 100            //预定义的最大非零元个数
typedef struct
 {    int i, j;                //该非零元的行下标和列下标
      ElemType e;              //非零元素值
} Triple;
```

再将稀疏矩阵三元组顺序表的存储结构描述如下:

```
typedef struct
{    Triple data[MAXSIZE];     //非零元三元组表
     int    m;                 //矩阵的行数
     int    n;                 //矩阵的列数
     int    t;                 //矩阵的非零元个数
} TSMatrix;
```

2) 函数接口说明

```
void InitMatrix(int mat[MAXLEN][MAXLEN], int m, int n, TSMatrix& TSM);
//根据已知的稀疏矩阵 mat,创建三元组表 TSM,其中 m 和 n 分别是稀疏矩阵的行数和列数
void TransposeSMatrix(TSMatrix M, TSMatrix& T);
//求稀疏矩阵 M 的转置矩阵 T
```

3) 输入输出说明

输入说明:输入信息分为 2 部分。第 1 部分共 1 行,输入稀疏矩阵的行列数 m 和 n,中间用一个空格隔开;第 2 部分根据第 1 行输入的 m 和 n 的值对应输入 m 行 n 列的矩阵数据,数据之间用一个空格隔开。

输出说明:输出分为 2 部分。第 1 部分输出原矩阵的行数、列数和非零元素的个数,及其对应的三元组表;第 2 部分为转置矩阵输出原矩阵的行数、列数和非零元素的个数,及其

对应的三元组表。

4）测试用例

测试用例信息如表 3-2 所示。

表 3-2　测试用例

序号	输　　入	输　　出	说明
1	5 6 0 0 8　0 0　0 0 0 0　0 0　0 5 0 0　0 16 0 0 0 18 0 0　0 0 0 0　9 0　0	原稀疏矩阵：5 行 6 列共 5 个非零元素 行　列　元素值 0　2　8 2　0　5 2　4　16 3　2　18 4　3　9 转置后的稀疏矩阵：6 行 5 列共 5 个非零元素 行　列　元素值 0　2　5 2　0　8 2　3　18 3　4　9 4　2　16	无

4. 解决方案

上述接口的实现方法分析如下。

（1）创建稀疏矩阵的三元组顺序表操作 InitMatrix(mat,m,n,&TSM)：首先设置稀疏矩阵对应的三元组顺序表的行数、列数分别为 m、n；再依次将二维数组 mat 中的每个非零元所在的行、列和元素值存储在三元组表中并对非零元计数，最后将三元组顺序表的非零元个数设置为计数器值。

（2）稀疏矩阵转置操作 TransposeSMatrix(M,&T)：矩阵转置是一种简单的矩阵运算，指的是将矩阵中每个元素的行列号互换一下。对于一个 m×n 的矩阵 M，它的转置矩阵 T 是一个 n×m 的矩阵，且 T(i,j)= M(j,i)。当稀疏矩阵用三元组顺序表来表示时，是以行主序的原则存放非零元素的，这样存放有利于稀疏矩阵的运算。那么，根据转置的概念，首先可设置转置后矩阵 T 的行数、列数分别为原矩阵 M 的列数和行数，非零元素个数不变；再依次将原矩阵 M 中存储的非零元的三元组中的每个元素转存到转置矩阵 T 的非零元三元组中，注意每一非零元转置后的行号是转置前的列号，转置后的列号是转置前的行号，转置前后元素值不变。

然而，若按行列序号直接互换进行转置，则所得的三元组顺序表就不再满足行主序的原则。例如，图 3-2(a)中的三元组所表示的矩阵，转置后如图 3-2(b)所示，不再满足行主序的原则。为解决此问题，可按照以下方法进行矩阵转置：扫描转置前的三元组表，并按先列序、后行序的原则转置三元组。例如，对图 3-2(a)中的三元组，从第 0 行开始向下搜索列序号为 0 的元素，找到三元组(2,0,5)，则转置为(0,2,5)，并存入转置后的三元组顺序表中。接着搜索列序号为 1 的元素，没找到；再搜索列序号为 2 的元素，找到(0,2,8)，转置为

(2,0,8),并存入转置后的三元组顺序表中。以此类推,直到扫描完所有列,即可完成矩阵转置,并且转置后的三元组表应满足先行序、后列序的原则。正确转置后的三元组如图 3-2(c)所示。

	row	column	value
0	0	2	8
1	2	0	5
2	2	4	16
3	3	2	18
4	4	3	9

(a) 三元组

	row	column	value
0	2	0	8
1	0	2	5
2	4	2	16
3	2	3	18
4	3	4	9

(b) 行列互换转置后的三元组

	row	column	value
0	0	2	5
1	2	0	8
2	2	3	18
3	3	4	9
4	4	2	16

(c) 转置后的三元组(按行有序)

图 3-2　矩阵转置

5. 程序代码参考

```
#include <stdio.h>

#define MAXLEN 255                  //用户可在 255 以内定义数据最大长度
#define MAXSIZE 12500               //假设非零元个数的最大值为 12500

typedef int ElemType;

typedef struct {
    int i, j;                       //该非零元的行下标和列下标
    ElemType e;
} Triple;

typedef struct {
    Triple data[MAXSIZE];           //非零元三元组表
    int    m;                       //矩阵的行数
    int    n;                       //矩阵的列数
    int    t;                       //矩阵的非零元个数
} TSMatrix;

//从一个稀疏矩阵创建三元组表,mat 为稀疏矩阵
void InitMatrix(int mat[MAXLEN][MAXLEN], int m, int n, TSMatrix& TSM)
{   int i, j, k = 0, count = 0;
    TSM.m = m;
    TSM.n = n;
    for (i = 0; i < m; i++)
         for (j = 0; j < n; j++)
             if (mat[i][j] != 0)
             {   TSM.data[count].i = i;
                 TSM.data[count].j = j;
                 TSM.data[count].e = mat[i][j];
                 count++;
             }
```

```
    TSM.t = count;
}

void TransposeSMatrix(TSMatrix M, TSMatrix& T)
// 采用三元组存储表示,求稀疏矩阵 M 的转置矩阵 T
{   int p, q, col;
    T.m = M.n;
    T.n = M.m;
    T.t = M.t;
    if (T.t)
    {   q = 0;
        for (col = 0; col < M.n; col++)
             for (p = 0; p < M.t; p++)
                  if (M.data[p].j == col)
                  {   T.data[q].i = M.data[p].j;
                      T.data[q].j = M.data[p].i;
                      T.data[q].e = M.data[p].e;
                      q++;
                  }
    }
}

void PrintSMatrix(TSMatrix M)
//输出稀疏矩阵 M
{   int i;
    printf("%d行%d列共%d个非零元素\n", M.m, M.n, M.t);
    printf("行 列 元素值\n");
    for (i = 0; i < M.t; i++)
         printf("%2d%4d%8d\n", M.data[i].i, M.data[i].j, M.data[i].e);
}

int main()
{   int mat[MAXLEN][MAXLEN];
    int m, n;
    printf("输入稀疏矩阵的行列数:");
    scanf("%d %d", &m, &n);
    printf("输入稀疏矩阵的元素值:\n");
    for (int i = 0; i < m; i++)
         for (int j = 0; j < n; j++)
              scanf("%d", &mat[i][j]);
    TSMatrix M, T;
    InitMatrix(mat, m, n, M);
    printf("\n原稀疏矩阵:");
    PrintSMatrix(M);
    TransposeSMatrix(M, T);
    printf("\n转置后的稀疏矩阵:");
    PrintSMatrix(T);
}
```

6. 运行结果参考

程序测试用例的运行结果如图 3-3 所示。

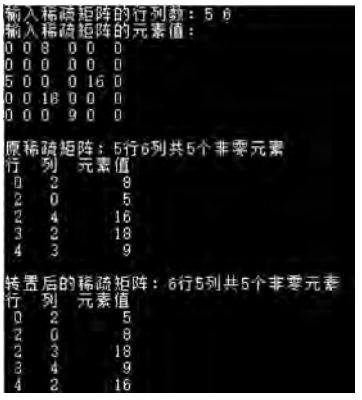

图 3-3　程序部分测试用例的运行结果

7. 延伸思考

(1) 分析一下上面实现的矩阵转置算法的时间复杂度,想想看这个转置算法的时间主要花费在什么地方？如何改进才能实现矩阵的快速转置？

(2) 如何在三元组顺序表存储表示下实现两个稀疏矩阵的加法运算？

3.2　进 阶 实 践

3.2.1　字符串加密、解密

1. 实践目的

(1) 能够正确分析字符串加密、解密涉及的关键问题及其解决思路。

(2) 能够根据字符串加密、解密的特点选择适当的存储结构。

(3) 能够运用串的基本操作方法设计字符串加密、解密中关键操作算法。

(4) 能够编写程序验证字符串加密、解密程序的正确性。

(5) 能够对实践结果的性能进行辩证分析或优化。

2. 实践内容

对给定的字符串按照指定的密钥进行加密处理,加密后生成密码串；另一方面,程序也可对给定的加密后的密码串按照密钥进行解密处理,还原成明文。加密规则是：根据输入的密钥,将需加密的字符串中每个字符与密钥进行异或运算。由于异或运算具有对称性,解密时,也只需要将加密后的字符串与密钥进行异或运算即可还原出原始字符串。

3. 实践要求

(1) 为使字符串加密、解密具有普适性,要求字符串由用户自主设定。

(2) 根据字符串加密、解密规则及其操作特性,自主选择恰当的存储结构。

(3) 给出关键操作功能模块的接口描述及其实现算法。

(4) 输入输出说明：本实践输入内容有 2 行,第 1 行为输入需要加密的一串字符,第 2 行输入一个数,作为加密时使用的密钥。输出内容也分 2 行,第 1 行为加密后的字符串,第

2 行为解密后的字符串。

(5) 测试用例见“运行结果”。

4. 解决方案

1) 数据结构

由于字符串加密、解密程序中处理的对象是字符串,一般情况下长度限定在一定范围内,而且在做加密、解密操作时,需要对各个字符进行运算,因此涉及随机存取串中的各个元素,为此本实践中最后拟采用定长顺序存储表示,其存储结构描述如下。

```
#define MAXSTRLEN 255                          //用户可在 255 以内定义最大串长
typedef unsigned char SString[MAXSTRLEN + 1];  //0 号单元存放串的长度
```

2) 关键操作实现要点

从实践内容及上述分析可知本实践的关键操作包括一个加密操作和一个解密操作,其中需要对字符串中每个字符与密钥进行异或运算。它们的实现要点简要说明如下:

(1) 加密操作。

本实践中的加密规则:根据输入的密钥,将需加密的字符串中每个字符与密钥进行异或运算。

(2) 解密操作。

由于异或运算具有对称性,解密时,也只需要将加密好的字符串与密钥进行异或运算即可还原出原始字符串。

3) 关键操作接口描述

```
Status StrEncode(SString S, int key, SString& T);
//串加密操作,其中 S 是原串,T 是加密后的串,key 为密钥
Status StrDecode(SString S, int key, SString& T);
//串解密操作,其中 S 是需解密的串,T 是解密后的串,key 为密钥
```

4) 关键操作算法参考

(1) 加密操作算法。

```
void StrEncode(SString S, int key, SString& T)
//串加密操作,其中 S 是原串,T 是加密后的串,key 为密钥
{   T[0] = S[0];
    for (int i = 1; i <= S[0]; i++)
        T[i] = S[i] ^ key;
}
```

(2) 解密操作算法。

```
void StrDecode(SString S, int key, SString& T)
//串解密操作,其中 S 是需解密的串,T 是解密后的串,key 为密钥
{   T[0] = S[0];
    for (int i = 1; i <= S[0]; i++)
        T[i] = S[i] ^ key;
}
```

5. 程序代码参考

扫码查看：3-2-1.cpp

6. 运行结果参考

程序测试用例的运行结果如图 3-4 所示。

(a) 测试用例1的运行结果　(b) 测试用例2的运行结果　(c) 测试用例3的运行结果

图 3-4　程序测试用例的运行结果

7. 延伸思考

(1) 如果加密规则改为恺撒加密法，则如何实现加密、解密算法？

(2) 如果需要对长文本进行加密、解密操作，则应该如何设计本实践案例？

3.2.2　求字符串的最小循环节问题

1. 实践目的

(1) 能够正确分析字符串的最小循环节问题所涉及的关键问题及其解决思路。

(2) 能够根据字符串的最小循环节问题的特点选择适当的存储结构。

(3) 能够运用模式匹配方法设计字符串的最小循环节问题中的关键算法。

(4) 能够编写程序验证求解字符串的最小循环节问题的正确性。

(5) 能够对实践结果的性能进行辩证分析或优化。

2. 实践内容

对于一个长度为$|S|$的字符串 S，如果 S 是由长度为$|T|$的字符串 T 循环 k 次构成，那么字符串 T 就是字符串 S 的最小循环节。

本问题要求编程实现求字符串的最小循环节，如组成字符串“ababab”的最小循环节就是“ab”。

3. 实践要求

(1) 为使求解字符串的最小循环节问题具有普适性，要求字符串由用户自主设定。

(2) 根据求解字符串的最小循环节问题的规则及其操作特性，自主选择恰当的数据存储结构。

(3) 给出关键操作功能模块的接口描述及其实现算法。

(4) 输入输出说明：本问题要求输入的字符串必须是由某个循环节循环若干次构成的。输出分两行，第 1 行为最小循环节长度，第 2 行为最小循环节。

4. 解决方案

1) 数据结构

本问题的处理对象是由某个循环节循环若干次构成的字符串，一般情况下长度限定在

一定范围内，而且本问题涉及计算串的 next[j]函数值，需要随机存取串中的各个字符，所以本实践中串也适用定长顺序存储表示，但考虑到串的堆分配存储表示其空间的大小可根据实际需要进行动态分配，其利用率更高，为此本实践中的串拟采用堆分配存储表示，其存储结构描述如下：

```
typedef struct
{    char * ch;              //若是非空串,则按串长分配存储区,否则 ch 为 NULL
     int length;             //串长
}HString;
```

2）关键操作实现要点

假定字符串 S 的长度为$|S|$，如果它是由长度为$|T|$的字符串 T（字符串 T 的最小循环节是其本身）循环 k 次构成，那么字符串 T 就是字符串 S 的最小循环节。

字符串的最小循环节有一个很重要的性质和 KMP 算法有关，即$|S|-$next[$|S|$]为字符串 T 的长度$|T|$。

证明：如果字符串 S 是由 T 循环 k 次构成，则有：

$$|S|/|T|=k$$

且字符串 S 中，前 $k-1$ 次 T 的循环和后 $k-1$ 次 T 的循环构成的字符串相等，即

$$S[0..|S|-|T|-1]=S[|T|..|S|-1]$$

考虑到 KMP 算法中 next 数组的 next[$|S|$]的值是 $k-1$ 个循环节的长度，即

$$\text{next}[|S|]=|S|-|T|$$

得到：$|S|-$next[$|S|$]$=|T|$。

因此，本问题最终就转化为求解串 next[j]的函数值的问题，这也是本问题的核心操作。其实现要点说明如下：

（1）求 next 函数值。

在 next 函数中，每个 j 值都有一个 k 值与之相对应，一般用 next[j]的函数值表示 j 值对应的 k 值。next 函数定义为

$$\text{next}[j]=\begin{cases}0, & \text{当 } j=1 \text{ 时}\\ \max\{k \mid 1<k<j \text{ 且} "t_1\cdots t_{k-1}"="t_{j-k+1}\cdots t_{j-1}"\}, & \text{当此集合不空时}\\ 1, & \text{其他情况}\end{cases}$$

由 next 函数的定义可知，求解 next[j]的函数值的过程是一个递推过程，初始时：next[1]$=0$。

若存在 next[j]$=k$，则表明在串 S 中有

$$"t_1\cdots t_{k-1}"="t_{j-k+1}\cdots t_{j-1}" \quad (1<k<j)$$

其中：k 为满足等式的最大值。此时，计算 next[$j+1$]的值存在以下两种情况：

情况 1：若 $t_k=t_j$，则表明在模式中存在

$$"t_1\cdots t_{k-1}t_k"="t_{j-k+1}\cdots t_{j-1}t_j" \quad (1<k<j)$$

并且不可能存在 $k'>k$ 满足上式，因此，可得到

$$\text{next}[j+1]=\text{next}[j]+1=k+1$$

情况 2：若 $t_k\neq t_j$，则表明在模式中存在

$$"t_1\cdots t_{k-1}t_k"\neq"t_{j-k+1}\cdots t_{j-1}t_j"$$

此时,可以把计算 next[j]的函数值的问题看成一个模式匹配过程。而整个串既是主串又是模式,如图 3-5 所示。

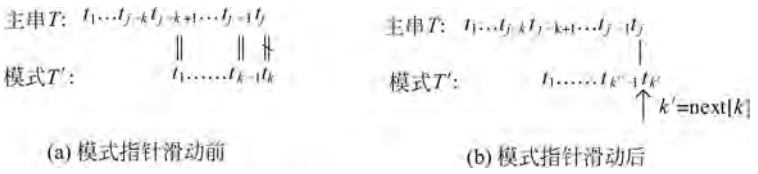

图 3-5　求 next[$j+1$]

在当前匹配过程中,已有"$t_1 \cdots t_{k-1}$"="$t_{j-k+1} \cdots t_{j-1}$"成立,则当 $t_k \neq t_j$ 时,应将模式 T' 向右滑动至 k'=next[k](1<k'<k<j),并把 k'位置上的字符与主串中 j 位置上的字符做比较。

若此时 $t_{k'}=t_j$,则表明在主串 T 中第 $j+1$ 个字符之前存在一个最大长度为 k'的子串,使得

$$"t_1 \cdots t_{k'-1} t_{k'}" = "t_{j-k+1'} \cdots t_{j-1} t_j" \quad (1 < k' < k < j)$$

因此,有

$$\text{next}[j+1] = k' + 1 = \text{next}[k] + 1$$

若此时 $t_{k'} \neq t_j$,则将模式 T'向右滑动至 k''=next[k']后继续匹配。以此类推,直至某次比较有 $t_k=t_j$(此即为上述情况),或某次比较有 $t_k \neq t_j$ 且 $k=1$,此时有 next[$j+1$]=1。

(2) 求最小循环节长度

在求得字符串 S 的 next 数组后,该字符串的最小循环节长度即为$|S|-$next[$|S|$]。

3) 关键操作接口描述

```
void getNext(HString S, int next[]);
//求模式 S 的 next 函数值并存入 next 数组
int CycleLen(HString T, int next[]);
//求串 T 中的最小循环节的长度,并返回其值
```

4) 关键操作算法参考

(1) 求 next 的函数值算法。

```
void getNext(H String T, int next[])
//求模式串 T 的 next 的函数值并存入数组 next
{   int i = 1, j = 0;
    next[1] = 0;
    while (i < T.length)
        if (j == 0 || T.ch[i] == T.ch[j])
        {   i++;
            j++;
            next[i] = j;
        }
        else
            j = next[j];
}
```

(2) 求最小循环节的长度。

```
int CycleLen(HString T, int next[])
//求串 T 中的最小循环节的长度
{   int len = StrLength(T);
    return len - next[len];
}
```

5. 程序代码参考

扫码查看：3-2-2.cpp

6. 运行结果参考

程序运行的结果如图 3-6 所示。

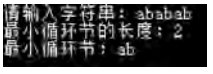

图 3-6　程序部分测试用例的运行结果

7. 延伸思考

(1) 如果输入字符串不是由某个循环节循环若干次构成的，程序运行结果将如何？

(2) 如果不借助 KMP 算法和 next 函数的性质，如何编写解决本问题的算法和验证程序？

3.2.3　基于协同过滤的图书推荐系统

1. 实践目的

(1) 能够正确分析基于协同过滤的图书推荐系统要解决的关键问题及其解决思路。

(2) 能够根据基于协同过滤的图书推荐系统需实现的功能选择恰当的存储结构。

(3) 能够运用行逻辑链接的顺序表的实现方法设计基于协同过滤的图书推荐系统中关键操作算法。

(4) 能够编写程序模拟基于协同过滤的图书推荐系统的实现，并验证其正确性。

(5) 能够对实践结果的性能进行辩证分析和优化提升。

2. 实践内容

编程实现一个基于协同过滤的图书推荐系统，模拟购书网站，根据网站用户的购书情况能够向特定用户推荐其感兴趣的图书。

3. 实践要求

(1) 分析基于协同过滤的图书推荐系统的关键要素及其操作特性，自主选择恰当的存储结构。

(2) 抽象出本系统的关键操作功能模块，并给出其接口描述及其实现算法。

(3) 输入输出说明：本系统要求输入用户人数、图书数量、用户姓名、图书名，以及用户

购买图书的情况,最后输入目标用户姓名。输出向目标用户推荐的图书。为了能使输入与输出信息更加人性化,可以在输入与输出信息之前适当添加有关输入或输出的提示信息。

4. 解决方案

1) 数据结构

协同过滤(collaborative filtering)是利用某些兴趣相投、拥有共同经验的群体的喜好推荐用户感兴趣的信息,个人通过合作的机制给予信息相当程度的评价(如购买、评分等),并记录下来以达到过滤的目的,进而帮助别人筛选信息。

协同过滤基于这样的思想,在某处总有和你兴趣相投的人。假设你和兴趣相投的人们评价方式都非常类似,而且你们都已经以这种方式评价了一组特定的项目,此外,你们每个人对其他人尚未评价的项目也有过评价。正如已经假设的那样,你们的兴趣是类似的,因此可以从兴趣相投的人们那里,提取具有很高评分而你尚未评价的项目,作为给你的推荐,反之亦然。

协同过滤的基础是效用矩阵(utility matrix),实际应用中的效用矩阵往往涉及大量的用户和项目,而用户不可能对所有的项目都给予评价,因此效用矩阵往往是一个高阶的稀疏矩阵。

如表 3-3 所示,其中行信息表达了用户购买图书的情况,表中的数据"1"表示该用户购买了对应列的图书,数据"0"表示该用户未购买对应列的图书。以用户张三为例,他购买了《数据结构》和《大数据基础》,而未购买《人工智能》。

表 3-3 效用矩阵

图 书	用 户		
	数 据 结 构	大数据基础	人 工 智 能
张三	1	1	0
李四	1	1	1
王五	0	1	1

由于效用矩阵往往是一个高阶的稀疏矩阵,所以需要对效用矩阵进行压缩存储,在找出与目标用户有相同兴趣的用户时,需要快速定位每位用户,也就是要在效用矩阵中快速定位到行,由于三元组顺序表以行主序存放矩阵的非零元,为取得某一行的非零元,必须从三元组顺序表的首元素开始进行查找,效率较低。如果能在三元组中增加一个"指示矩阵中每一行的首个非零元在三元组表中的序号"的数据,便可以随机存取稀疏矩阵中任意一行的非零元。这就是稀疏矩阵的"行逻辑链接"的顺序表表示,其类型描述如下:

```
typedef struct
{    Triple data[MAXSIZE];                    //非零元三元组表
     int rpos[MAXRC];                         //指示各行首个非零元在三元组表中的位置
     int m, n, t;                             //矩阵的行数、列数和非零元的个数
}RLSMatrix;
```

2) 关键操作实现要点

本系统是基于用户的协同过滤推荐,最核心的功能是获取用户间的相似度。本实践能够根据输入的效用矩阵计算出用户相似度矩阵,再根据用户相似度矩阵向目标用户推荐图

书。那么如何计算用户相似度矩阵呢，下面介绍一个比较简单的算法，也是本次实践所用的算法：基于共现度的算法。即当两个用户同时评价了同一个项目，则他们之间的相似度+1，这样就可以计算出各个用户之间的相似度了，最后得到的相似度矩阵如表 3-4 所示。

表 3-4　用户相似度矩阵

用　　户	张三	李四	王五
张三	*	2	1
李四	2	*	2
王五	1	2	*

以用户张三和李四为例，由于用户张三和李四均购买了《数据结构》和《大数据基础》，即他们对 2 个相同项目进行了评价，因此其相似度为 2。

如果系统有较多用户，推荐集合应定为多个元素（用户）的集合，这样能保证推荐的准确性。为了简洁，本示例仅有 3 位用户，所以推荐集合就定为单元集，如果要给张三推荐，那么先找到与张三最相似的用户集合{李四}，因张三与李四的相似度是 2，而与王五的相似度是 1。

既然找到了张三的最相似用户是李四，而且也可以发现李四购买过《人工智能》，而张三没有购买过，所以我们就给张三推荐《人工智能》。

为了实现基于协同过滤的图书推荐，得找出与目标用户有相同兴趣的用户，而效用矩阵中记录了原始信息。只需要在效用矩阵中找到与目标用户具有最高相似度的若干用户，然后再向目标用户推荐这些用户购买了而目标用户没有购买的图书即可。

根据上述分析可知：本系统可以抽象出 3 个关键性操作，其实现要点说明如下。

（1）创建行逻辑链接的顺序表。

创建稀疏矩阵的行逻辑链接的顺序表，只需在 3.1.2 节中创建稀疏矩阵的三元组顺序表的基础上将 rpos 这个“指示矩阵中每行的首个非零元在三元组表中的序号”的辅助数组固定在存储结构中，将便于随机存取稀疏矩阵中任意一行的非零元。具体做法是，在将稀疏矩阵的非零元存入三元组表中时，增加一个计数器计数非零元数目，当每行开始存放时，计数器中的值即为 rpos 数组对应行的值。

（2）取值操作，即求 r 行 c 列的元素的算法。

在行逻辑链接的顺序表存储下，由于有了 rpos 数组，就无须像在三元组顺序表上取值操作那样要对三元组表中元素顺序扫描来实现行的定位，而只需直接读取数组 rpos 中下标为 r 位置的元素值即可实现行的定位。为此，稀疏矩阵的行逻辑链接的顺序表表示是一种能实现随机存取任一行操作的存储结构，但要获取该行中的指定列的元素则还需顺序存取。

（3）计算两个用户的相似度。

有了对元素的取值操作，计算两个用户的相似度就转化为对两个用户对应行的相同图书统计共同的购买信息。只需增加一个计数器，当两个用户购买同一图书时计数器加 1，计数器的值即为两个用户的相似度。

（4）向目标用户进行推荐。

根据用户 v 的购买情况向目标用户 u 进行推荐，检索用户 u 和 v 对应的图书购买信息，将用户 v 已经购买而用户 u 尚未购买的图书推荐出来。由于这样的图书可能有多本，因此

需要引用一个数组保存推荐的图书,并将此数组作为推荐结果返回其值。

3) 关键操作接口描述

```
void InitMatrix(int mat[MAXRC][MAXRC], int m, int n, RLSMatrix& M);
//根据已知的稀疏矩阵 mat 创建其行逻辑链接的顺序表 M, 其中 m 和 n 是矩阵的行数和列数
ElemType Value(RLSMatrix M, int r, int c);
//求矩阵 M 中第 r 行和第 c 列的元素值
 int Cooccurrence(RLSMatrix M, int u, int v);
 //根据行逻辑链接的顺序表 M,计算两个用户 u 和 v 的相似度
 int * Recommend(RLSMatrix M, int u, int v);
 //根据用户 v 向目标用户 u 进行推荐,函数返回值是存放推荐结果的整型数组
```

4) 关键操作算法参考

(1) 创建行逻辑链接的顺序表的算法。

```
void InitMatrix(int mat[MAXRC][MAXRC], int m, int n, RLSMatrix& M)
//根据已知的稀疏矩阵 mat 创建其行逻辑链接的顺序表 M, 其中 m 和 n 是矩阵的行数和列数
{    int i, j, k = 0, count = 0;
     M.m = m;
     M.n = n;

     for (i = 0; i < m; i++)
     {    M.rpos[i] = count;
          for (j = 0; j < n; j++)
          {
               if (mat[i][j] != 0)
               {
                    M.data[count].i = i;
                    M.data[count].j = j;
                    M.data[count].e = mat[i][j];
                    count++;
               }
          }
    }
    M.t = count;
}
```

(2) 给定下标求矩阵的元素值的算法。

```
ElemType Value(RLSMatrix M, int r, int c)
//求矩阵 M 中第 r 行和第 c 列的元素值
{    int p = M.rpos[r];                          //第 r 行的首个非零元在三元组中的位置
     while (M.data[p].i == r && M.data[p].j < c)   //找第 c 列
          p++;
     if (M.data[p].i == r && M.data[p].j == c)
          return M.data[p].e;
    else
          return 0;
}
```

(3) 计算两个用户的相似度的算法。

```
int Cooccurrence(RLSMatrix M, int u, int v)
//根据行逻辑链接的顺序表 M,计算两个用户 u 和 v 的相似度
{     int c = 0;
      for (int j = 0; j < M.n; j++)
            if (Value(M, u, j) == Value(M, v, j))
                  c++;
      return c;
}
```

(4) 向目标用户进行推荐的算法。

```
int * Recommend(RLSMatrix M, int u, int v)
//根据用户 v 向目标用户 u 进行推荐,函数返回值是存放推荐结果的整型数组
{     int c = 0;
      int rec[MAXRC];
      for (int j = 0; j < M.n; j++)
            if (Value(M, v, j) == 1 && Value(M, u, j) == 0)
                  rec[c++] = j;
      rec[c] = -1;
      return rec;
}
```

5. 程序代码参考

扫码查看：3-2-3.cpp

6. 运行结果参考

程序运行的结果如图 3-7 所示。

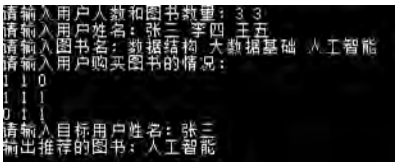

图 3-7 程序运行结果

7. 延伸思考

(1) 计算两个用户的相似度时,多次调用了取值操作 Value,但由于行逻辑链接的顺序表不具有完全的随机存取特性,这样多次调用取值操作会导致较高的时间复杂度。事实上,只需将两个用户评价的项目整行读出,然后再找出共同的项目数即可。那么应该怎样修改相似度的算法实现这一思想?

(2) 本实践根据一个最相似用户完成了向目标用户的推荐,请思考如何找出与某用户

相似度最高的若干用户,然后根据他们评价项目的情况对目标用户进行推荐?

3.3 拓展实践

3.3.1 字符串的解压缩

1. 实践目的

(1) 能够正确分析字符串解压缩过程中要解决的关键问题及其解决思路。

(2) 能够根据字符串解压缩的操作特点选择适当的存储结构。

(3) 能够运用字符串基本操作的实现方法设计字符串解压缩中的关键操作算法。

(4) 能够编写程序模拟字符串解压缩的实现,并验证其正确性。

(5) 能够对实践结果的性能进行辩证分析和优化提升。

2. 实践内容

行程编码(run length encoding,RLE),又称游程编码、行程长度编码、变动长度编码等,是一种统计编码。主要技术是检测重复的比特或字符序列,并用它们的出现次数取而代之。例如,有一个字符串"abcabccdcdcdef",经过行程编码后可以用"2[abc]3[cd]ef"来表示,从而实现对数据的压缩。行程编码尤其适用于计算机生成的图形图像,对减少存储容量很有效果。

本实践内容的具体描述是:给定一个经过行程编码压缩后的字符串,设计算法求出该字符串解压缩后的字符串,并编程验证算法的正确性。行程编码规则为:k[encoded_string]。其中,k 为数字;方括号内部的 encoded_string 为元字符串,数字 k 表示元字符串重复的次数。

3. 实践要求

(1) 输入要求:输入经过压缩的字符串,编码规则符合要求。

(2) 输出要求:输出经过解压缩后的字符串(不包含数字和方括号)。

(3) 测试用例信息如表 3-5 所示(只供参考,但不局限于此)。

表 3-5 测试用例

序号	输入	输出	说明
1	3[a]2[bc]	aaabcbc	含有两个编码点
2	3[a2[c]]	accaccacc	编码点存在嵌套
3	2[abc]3[cd]ef	abcabccdcdcdef	在编码点后存在非编码串
4	abc3[cd]xyz	abccdcdcdxyz	在编码点前后均存在非编码串
5	3[a2[c4[fg]]]	acfgfgfgfgcfgfgfgfgacfgfgfgfgcfgfgfgfgfgacfgfgfgfgcfgfgfgfg	一个短的压缩串由于编码点嵌套的存在,可能会解压缩出很长的串

4. 解决思路

1) 数据存储结构的设计要点提示

由于字符串解压缩问题中处理的对象仅仅是字符串,而一个短的压缩串由于编码点嵌套的存在,可能会解压缩出很长的串,另外,在操作过程中,需要经常对串中的字符进行随机访问,那么,你觉得本实践应该采用何种存储结构更为合适?

2）关键操作的实现要点提示

本实践主要完成的操作是对输入字符串的解压缩，由于输入字符串定义是递归的，因此解压缩算法也用递归思路处理，其实现要点说明如下：

依次读取每个字符，如果读到的字符是数字字符，则需要解析成对应的数字 k，作为后面元字符串重复出现的次数；如果读到的字符是“[”，说明是要重复的元字符串的开始，且 k 中已经存放了元字符串需要重复的次数，此时，需要根据重复次数 k 和当前需要重复的元字符串，将元字符串重复 k 次连接到解析字符串的后面；如果读到的字符是“]”说明当前重复出现的元字符串结束，相应的递归操作也结束。过程如图 3-8 所示。

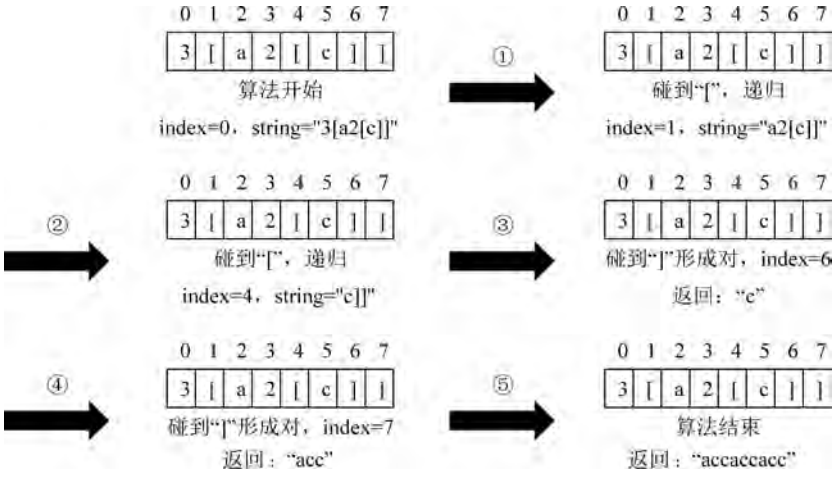

图 3-8　字符串解压缩过程

3）关键操作接口描述

从图 3-8 所示可知，整个解压缩过程就是一个不断从某个 index 位置开始解析串的过程，这个过程的接口可描述如下：

```
HString Analysis(HString S, int &index)
//从 index 位置开始解析串 S
```

5. 程序代码参考

扫码查看：3-3-1.cpp

6. 延伸思考

（1）由于输入字符串的定义具有递归特性，因此本题的解析算法也用递归的思想进行处理，请分析本算法的时间复杂度和空间复杂度。

（2）本问题能否不用递归算法进行解决？比如，应用第 2 章描述的栈结构进行解决，则代码应该如何编写？

3.3.2 《决议》高频词统计问题

1. 实践目的

(1) 能够通过《中国共产党关于党的百年奋斗重大成就和历史经验的决议》(简称《决议》)高频词的正确解读来领会中国共产党第十九届中央委员会第六次全体会议的精髓要义,从而深刻体悟中国共产党的执政理念,厚植爱党爱国的情怀,增强“四个自信”。

(2) 能够根据对《决议》高频词统计的操作特点选择恰当的存储结构。

(3) 能够运用 KMP 算法设计并实现词频统计功能。

(4) 能够编写程序测试对《决议》词频统计结果的正确性。

(5) 能够对实践结果的性能进行辩证分析和优化提升。

2. 实践背景

2021 年 11 月 11 日,中国共产党第十九届中央委员会第六次全体会议通过了《中共中央关于党的百年奋斗重大成就和历史经验的决议》。

一百年来,党领导人民浴血奋战、百折不挠,创造了新民主主义革命的伟大成就;自力更生、发愤图强,创造了社会主义革命和建设的伟大成就;解放思想、锐意进取,创造了改革开放和社会主义现代化建设的伟大成就;自信自强、守正创新,创造了新时代中国特色社会主义的伟大成就。党和人民百年奋斗,书写了中华民族几千年历史上最恢宏的史诗。

总结党的百年奋斗重大成就和历史经验,是在建党百年历史条件下开启全面建设社会主义现代化国家新征程、在新时代坚持和发展中国特色社会主义的需要;是增强政治意识、大局意识、核心意识、看齐意识,坚定道路自信、理论自信、制度自信、文化自信,做到坚决维护习近平同志党中央的核心、全党的核心地位,坚决维护党中央权威和集中统一领导,确保全党步调一致向前进的需要;是推进党的自我革命、提高全党斗争本领和应对风险挑战能力、永葆党的生机活力、团结带领全国各族人民为实现中华民族伟大复兴的中国梦而继续奋斗的需要。全党要坚持唯物史观和正确党史观,从党的百年奋斗中看清楚过去我们为什么能够成功、弄明白未来我们怎样才能继续成功,从而更加坚定、更加自觉地践行初心使命,在新时代更好坚持和发展中国特色社会主义[①]。

在《决议》全文中,“人民”这个高频词汇被提到 249 次,体现了我们党始终不忘团结带领全国各族人民不断为美好生活而奋斗的初心使命。“江山就是人民,人民就是江山”,彰显了我们党全心全意为人民服务的根本宗旨,展现了新时代共产党人不忘初心、牢记使命的坚定决心。

“社会主义”这个高频词汇共出现了 156 次,因为中国特色社会主义制度是我们的根本制度,也是我们实现中华民族伟大复兴的必经之路。坚持社会主义方向,坚持中国特色社会主义道路,这是一个根本性的要求。

3. 实践内容

《决议》全文共有 36 000 多字,编程统计《决议》全文中高频词“人民”“社会主义”出现的频数,并输出统计结果。

---

① 中共中央关于党的百年奋斗重大成就和历史经验的决议. 新华网. http://www.news.cn/2021-11/16/c_1128069706.htm.

4. 实践要求

(1) 根据词频统计的处理对象及其操作特性,自主选择恰当的数据存储结构。

(2) 抽象出本实践中所涉及的关键性操作模块,并给出其接口描述及其实现算法。

(3) 输入要求:通过读取磁盘文本文件的方式一次性读取《决议》全文。《决议》全文的文本文件可通过扫右边的二维码下载。

3-3-2
inputdata.txt

(4) 输出要求:分别输出“人民”和“社会主义”在《决议》中出现的频数。

5. 解决思路

1) 数据存储结构的设计要点提示

本实践处理的对象是《决议》全文构成的字符串,而本实践要解决的核心问题则是求指定词语如“人民”“社会主义”在《决议》全文中出现的频数。它可以基于字符串的模式匹配算法来实现,其中主串是《决议》全文,模式串是“人民”和“社会主义”。由于《决议》全文共有36 000多字,另外,在串的匹配过程中,需要经常对串中的字符进行随机访问,所以建议采用串的堆分配存储表示。

2) 关键操作的实现要点提示

本实践的处理过程可以简单表述为:引进一个计数器变量,从主串的串首位置开始进行模式匹配,匹配成功时计数器加1,然后再从上次匹配成功的下一位置开始进行模式匹配,匹配成功时计数器再加1,重复上述过程,直至整个主串都匹配完成为止,此时计数器的值即为模式串在主串中出现的频数。

根据上述表述,其中最关键的操作是串的模式匹配,有两种主要的模式匹配算法:Brute-Force算法(BF算法)和KMP算法。

由D. E. Knuth、V. R. Pratt和J. H. Morris提出的模式匹配改进算法,简称KMP算法。KMP算法对BF算法做了很大的改进,模式匹配的效率较高。KMP算法的主要思想是,每当某趟匹配失败时,主串指针不回退,而是利用已经得到的“部分匹配”的结果,将模式向右“滑动”尽可能远的一段距离后,继续进行比较。

具体地说,KMP算法在匹配过程中产生“失配”时,主串指针不回退,模式指针退回到next函数值所指示的位置上重新进行比较,并且当模式指针退至零时,主串指针和模式指针需同时增1。即若主串的第i个字符和模式的第1个字符不等,则应从主串的第i+1个字符起重新进行匹配。

KMP算法需要求模式的next函数,这个在3.2.2节中已经做过介绍。

6. 程序代码参考

扫码查看:3-3-2.cpp

7. 延伸思考

本实践实现的功能是给定具体词语,计算出该词语在指定文本中的词频。对于文本来说,假定全文已经进行了分词处理,请思考如何编程统计指定文本中每个词语出现的词频。

8. 结束语

透过高频词对《决议》全文进行精要解读,以更好地学习宣传贯彻党的十九届六中全会精神,从党的百年奋斗中看清楚过去我们为什么能够成功、弄明白未来我们怎样才能继续成功,从而更加坚定、更加自觉地践行初心使命。深刻理解党百年奋斗的历史逻辑、理论逻辑、实践逻辑,深化对共产党执政规律、社会主义建设规律、人类社会发展规律的认识,正确把握党百年奋斗的宝贵历史经验,能够使我们始终掌握新时代新征程党和国家事业发展的历史主动,以更加昂扬的姿态迈进新征程、建功新时代。