**《数据结构》实验报告**

**姓名： 杨孟衡 学号： 8002118240**

**班级： 软件工程1809班 专业： 软件工程**

**报告日期： 2019 年 10 月 23 日**

**实验四 栈和字符串**

**一、问题描述与分析**

1. 设某线性数据元素类型为整型，以线性栈为存储结构。某线性数据元素类型为整形，以链式存储结构存储数据，试编程实现：

1. 已知栈顺序存储结构和基本操作，完成实验1；基于链式栈存储结构，完成实验2；利用字符顺序栈，实现算数表达式求值得程序-实验3；已知链式存储结构，完成取子串链表-实验4；字符串采用带头结点的链表存储，删除子串-实验5；字符串采用带头结点的链表存储，查找子串在主串中出现的位置-实验6；编写朴素模式匹配算法-实验7；编写KMP算法-实验8。
2. 已知顺序栈存储结构及基本操作定义(见seqstack.h)，利用顺序栈结构，编写算法函数void Dto16(unsigned int m)实现十进制无符号整数m到十六进制的转换。(lab4\_01.c)熟练掌握运用带头结点链表表示特定形式的数据的方法，并设计出有关算法。
3. 基于链式栈存储结构，重新实现栈的基本操作(linkstack.h)，并改写上题的进位制转换程序。(lab4\_02.c)
4. 利用字符串顺序栈，设计实现算数表达式求值。(lab4\_03.c)
5. 已知字符串采用带头结点的链式存储结构(linkstring.h)，请编写linkstring substring(linkstring s, int i, int len)函数，在字符串s中从第i个位置起取长度为len的子串，函数返回子串链表。(lab4\_04.c)
6. 字符串采用带头结点的链表存储，设计算法函数void delstring(linkstring s, int i, int len)在字符串s中删除从第i个位置开始，长度为len的子串。(lab4\_05.c)
7. 字符串采用带头结点的链表存储，编写函数linkstring index(linkstring s, linkstring t),查找子串t在主串s中第一次出现的位置，若匹配不成功，则返回NULL。(lab4\_06.c)
8. 利用朴素模式匹配算法，查找模式t在主串s中所有出现的位置，并将这些位置存储在带头结点的单链表中。(lab4\_07.c)
9. 编写快速模式匹配算法。

2.分析

(1)各种进制之间的转换算法为除以模取余数，所以在循环体内先将当前m对16求余然后把这个余数push栈中，接着将m右移四位（相当于除以16），再进入下一趟循环，直到m为0退出循环，此时此刻栈中存放的就是十进制整数转换以后的十六进制数，再根据ascll码表，数字0~9加上48是0~9的对应字符，若是数字A~F，则需要加上55才是对应的字符值，最后用putchar一个字符一个字符的弹出栈输出；

(2) 类似（1），只是数据结构不同，函数处理不同；

(3) 先创建两个线性空栈，提示输入数据元素，然后对实验数据进行转换计算处理，最后提示输出数据元素；

(4) 先根据i值将前指针p定位到子串第一个字符，再根据len值往后延伸指针temp，用这两个指针框定子串，然后将后指针temp指针域设置为NULL，头结点指针的指针域设置为前指针p，返回头结点指针即可；

(5) 先根据i值将后指针p定位到删除的起始位置，前指针temp定位到删除的起始位置的上一个位置，接着执行删除操作，每次让q指针指向当前删除的结点，p指向下一个可能删除的结点，temp指向当前删除结点的上一个结点，让temp的指针域指向p，free了q的结点，再接着使q=p继续删除操作，直到次数上限，或者链表到达结尾，返回头指针；

(6) 单重循环直到主串中的某个位置等于子串的第一个字符，然后直接返回当前结点；

(7) 朴素匹配算法，单重循环返回模式t在主串s中第一次出现的位置，否则返回-1，在indexall函数里，设置嵌套循环，外层循环确保主串s中的所有可能子串位置都能遍及到，内层循环根据每次临时数组temp与模式t是否匹配的结果做相应处理，若有匹配，那么放入新结点中，同时截断数组生成新数组temp，若不匹配那么退出循环；

(8) 分析同书。

**二、数据结构与算法设计**

1.数据结构：

线性栈代码：

#define MAXSIZE 100

typedef int datatype;

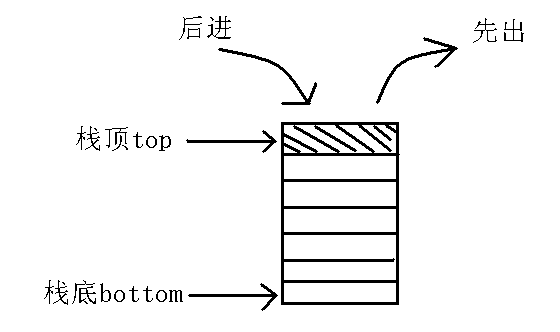
typedef struct

{       datatype  a[MAXSIZE];

        int top;

}seqstack;

线性栈图示如下：



链式栈代码：

typedef int datatype;

typedef struct node

{

        datatype data;

        struct node \*next;

}linknode;

typedef linknode \* linkstack;

静态链表采用基本数据类型数组作为基本元素构成，搭配一个指向自己本类的指针next。代码如下：

typedef char datatype;

typedef struct node

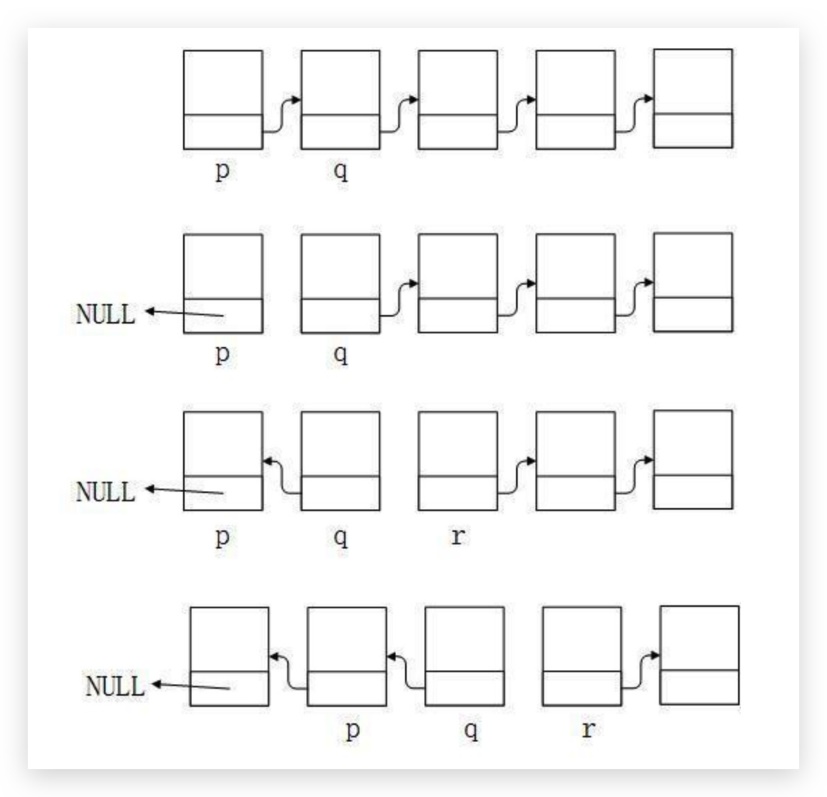
{   datatype data;

    struct node \*next;

}linknode;

typedef linknode \*linkstring;

链表图示如下：



链式字符串代码：

typedef char datatype;

typedef struct node

{   datatype data;

    struct node \*next;

}linknode;

typedef linknode \*linkstring;

**三、算法复杂度分析**

⑴ 线性栈的应用-十进制数转换十六进制数

转换只涉及到单重循环，算法复杂度为O（n）。

⑵ 链式栈的应用-十进制数转换十六进制数

单重循环，算法复杂度为O（n）。

（3）计算算术表达式

单重循环，算法复杂度为O(n)。

(4)基于带头结点的链式存储结构取子串

单重循环即可，算法复杂度为O（n）。

（5）基于带头结点的链式存储结构删除子串

单重循环即可，算法复杂度为O（n）。

（6）查找字串的位置

单重循环即可，算法复杂度为O（n）。

（7）朴素模式匹配算法

嵌套循环比较匹配子串和主串，算法复杂度为O（n^2）。

（8）快速模式匹配kmp算法

快速模式匹配，算法复杂度为O（n+m）。

**四、测试计划**

1.编写目的

加深对数据结构链表的存储结构，线性栈的理解。

2.开发及运行环境Dev c++

Visual Stdio Code 1.39

3.小项测试截图

（1）基于顺序栈存储结构实现十进制到十六进制的转换



（2）基于链式栈存储结构实现十进制到十六进制的转换

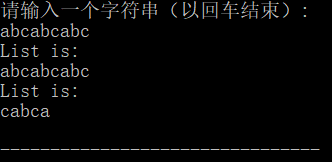


（3）算数表达式求值

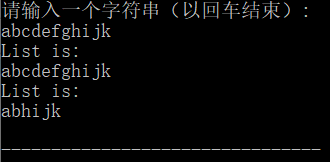


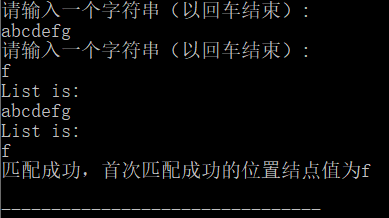

（4）基于带头结点的链式存储结构取子串



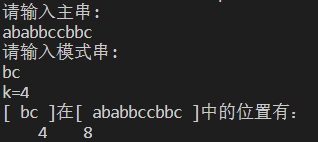
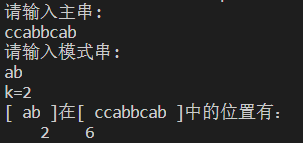
（5）基于带头结点的链式存储结构删除子串



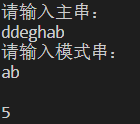
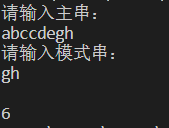
（6）查找子串的位置



（7）朴素模式匹配算法



（8）快速模式匹配KMP 算法



**五、源程序(仅贴算法函数部分)**

**<lab4\_01>:**

/\*

利用顺序栈结构，编写算法函数void Dto16(unsigned int m)实现十进制无符号整数m到十六进制数的转换功能。

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*文件名称：lab4\_01.c                 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "seqstack.h"

/\*请将本函数补充完整，并进行测试\*/

void Dto16(int m)

{   seqstack s;         /\*定义顺序栈\*/

    init(&s);

    printf("十进制数%u对应的十六进制数是：",m);

    while (m)

    {

        push(&s,m % 16);

        m = m >> 4;

    }

    while (!empty(&s))

                putchar(read(&s) < 10 ? pop(&s) + 48: pop(&s) + 55);

    printf("\n");

}

**<lab4\_02>:**

/\*

利用链式栈结构，编写算法函数void Dto16(unsigned int m)实现十进制无符号整数m到十六进制数的转换功能。

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*文件名称：lab4\_02.c                 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "linkstack.h"

/\*请将本函数补充完整，并进行测试\*/

void Dto16(unsigned int m)

{

    linkstack s;

    s=init();

    printf("十进制数%u对应的十六进制数是：",m);

    while (m)

    {

        s = push(s, m % 16);

        m = m << 4;

    }

    while (!empty(s))

               {

                   printf("%c", read(s) < 10? read(s) + 48:read(s) + 55);

                   s = pop(s);

               }

    printf("\n");

}

**<lab4\_03>:**

//利用栈处理中缀表达式，最终转换为后缀表达式，计算出结果输出结果

void CalPostFix(char\_stack\* c)

{

    int i = 0;

    int j = 0;

    int t;

    char temp[MAXSIZE];

    char after[MAXSIZE];

    double x1, x2;

    double result[MAXSIZE];

    int top = 0;

    //输入中缀表达式

    Initiate(c);

    Push(c, '#');

    printf("请输入中缀表达式(以#结束):\n");

    do {

        scanf("%c", &temp[i]);

        i++;

    } while (temp[i - 1] != '#');

    i = 0;

    //转换中缀表达式为后缀表达式

    while (temp[i] != '#')

    {

        if ((temp[i] >= '0' && temp[i] <= '9') || temp[i] == '.')

        {

            after[j++] = temp[i];

        }

        else if (temp[i] == '(')

        {

            Push(c, temp[i]);

        }

        else if (temp[i] == ')')

        {

            t = c->top - 1;

            while (c->a[t] != '(')

            {

                after[j++] = c->a[--c->top];

                t = c->top - 1;

            }

            Pop(c);

        }

        else if (IsOp(temp[i]))

        {

            after[j++] = ' ';

            while (Priority(c->a[c->top - 1]) >= Priority(temp[i]))

            {

                after[j++] = c->a[--c->top];

            }

            Push(c, temp[i]);

        }

        i++;

    }

    while (c->top) after[j++] = c->a[--c->top];

    printf("经过转换后的后缀表达式为:");

    for (int k = 0; after[k] != '#'; k++)

    {

        printf("%c", after[k]);

    }

    //计算后缀表达式

    i = 0;

    Initiate(c);

    while (after[i] != '#')

    {

        if (after[i] >= '0' && after[i] <= '9')

        {

            result[top] = readNumber(after, &i);

            top++;

        }

        else if (after[i] == ' ') i++;

        else if (after[i] == '+')

        {

            x2 = result[--top];

            x1 = result[--top];

            result[top] = x1 + x2;

            top++;

            i++;

        }

        else if (after[i] == '-')

        {

            x2 = result[--top];

            x1 = result[--top];

            result[top] = x1 - x2;

            top++;

            i++;

        }

        else if (after[i] == '\*')

        {

            x2 = result[--top];

            x1 = result[--top];

            result[top] = x1 \* x2;

            top++;

            i++;

        }

        else if (after[i] == '/')

        {

            x2 = result[--top];

            x1 = result[--top];

            result[top] = x1 / x2;

            top++;

            i++;

        }

    }

    //打印后缀表达式计算值

    printf("\n经过计算后，这个中缀表达式的值为: %.3lf\n", result[0]);

}

**<lab4\_04>:**

/\*

已知字符串采用带结点的链式存储结构（详见linksrting.h文件），

请编写函数linkstring substring(linkstring s,int i,int len)，

在字符串s中从第i个位置起取长度为len的子串，函数返回子串链表。

\*/

#include "linkstring.h"

/\*请将本函数补充完整，并进行测试\*/

linkstring substring(linkstring  s, int i, int len)

{

    linkstring p = s->next;

    linkstring q = s;

    linkstring temp = p;

    int k = 0;

    if(i < 0)

    {

        printf("没有这个位置的结点！请重新输入！\n");

        return s;

    }

    for(k = 1; k < i; k++)

    {

        if(p->next == NULL && (k + 1) < i)

        {

            printf("没有这个位置的结点！请重新输入！\n");

            return s;

        }

        temp = p;

        q->next = p->next;

        p = p->next;

        free(temp);

    }

    temp = p;

    for(k = 1;k < len; k++)

    {

        if(!temp)

        {

            break;

        }

        temp = temp->next;

    }

    temp->next = NULL;

    return s;

}

**<lab4\_05>:**

/\*

字符串采用带头结点的链表存储，设计算法函数void delstring(linkstring s, int i,int len)

在字符串s中删除从第i个位置开始，长度为len的子串。

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*文件名称：lab4\_05.c                 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "linkstring.h"

/\*请将本函数补充完整，并进行测试\*/

void delstring(linkstring  s, int i, int len)

{

    linkstring p = s->next;

    linkstring q = s;

    linkstring temp = q;

    int k = 0;

    if(i < 0)

    {

        printf("没有这个位置的结点！请重新输入！\n");

        exit(-1);

    }

    for(k = 1; k < i; k++)

    {

        if(p->next == NULL && (k + 1) < i)

        {

            printf("没有这个位置的结点！请重新输入！\n");

            exit(-1);

        }

        temp = temp->next;

        p = p->next;

    }

    k = 0;

    while(p && k < len)

    {

        q = p;

        p = p->next;

        temp->next = p;

        free(q);

        k++;

    }

    if(!p && k != len)

    {

        free(p);

        temp->next = NULL;

    }

}

**<lab4\_06>:**

/\*

字符串采用带头结点的链表存储，编写函数linkstring index(linkstring s, linkstring t)，

查找子串t在主串s中第一次出现的位置，若匹配不成功，则返回NULL。

\*/

#include "linkstring.h"

/\*请将本函数补充完整，并进行测试\*/

linkstring index(linkstring  s, linkstring t)

{

        linkstring p = s->next;

        linkstring q = t->next;

        int k = 1;

        while(p && q)

        {

                if(q->data == p->data)

                {

                        return p;

                }

                p = p->next;

                k++;

        }

        return NULL;

}

**<lab4\_07:>**

\*

利用朴素模式匹配算法，将模式t在主串s中所有出现的位置存储在带头结点的单链表中。

\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

typedef struct node

{

    int data;

    struct node \*next;

}linknode;

typedef linknode \*linklist;

/\*朴素模式匹配算法,返回t在s中第一次出现的位置，没找到则返回-1，请将程序补充完整\*/

int index(char \*s, char \*t)

{

    int i = 0, j = 0, succ = 0;

    int tlength = 0;

    int slength = 0;

    while (t[i] != '\0' && t[i] != EOF)

    {

        tlength++;

        i++;

    }

    while (s[j] != '\0' && s[j] != EOF)

    {

        slength++;

        j++;

    }

    i = j = 0;

    while ((i <= slength - tlength) && (!succ))

    {

        j = 0;

        succ = 1;

        while ((j <= tlength - 1) && succ)

        {

            if (t[j] == s[i + j])

            {

                j++;

            }

            else succ = 0;

        }

        i++;

    }

    if (succ) return (i - 1);

    else return -1;

}

/\*利用朴素模式匹配算法，将模式t在s中所有出现的位置存储在带头结点的单链表中,请将函数补充完整\*/

linklist indexall(char \*s, char \*t)

{

    linklist p = (linklist)malloc(sizeof(linknode));

    linklist head = p;

    linklist q;

    int tlength = 0;

    int slength = 0;

    int i = 0;

    int k = 0;

    int j = 0;

    char\* temp;

    p->next = NULL;

    p->data = 0;

    while (t[i] != '\0' && t[i] != EOF)

    {

        tlength++;

        i++;

    }

    i = 0;

    while (s[i] != '\0' && s[i] != EOF)

    {

        slength++;

        i++;

    }

    i = 0;

    temp = (char\*)malloc(sizeof(char) \* (slength + 1));

    for (k = 0,j = 0; k < slength; k++, j++)

    {

        temp[k] = s[j];

    }

    temp[slength] = '\0';

    while (i < slength)

    {

        k = index(temp, t);

        if (k == -1)

        {

            break;

        }

        else

        {

            q = (linklist)malloc(sizeof(linknode));

            q->data = k + i;

            i = q->data + 1;

            q->next = NULL;

            p->next = q;

            p = p->next;

            for (k = i, j = 0; j < slength - i; j++, k++)

            {

                temp[j] = s[k];

            }

            temp[j] = '\0';

        }

    }

    return head;

}

**<lab4\_08:>**

/\*

  编写快速模式匹配KMP算法，请将相关函数补充完整。

\*/

#define maxsize 100

typedef struct{

      char str[maxsize];

      int length ;

} seqstring;

/\*求模式p的next[]值，请将函数补充完整\*/

void getnext(seqstring p,int next[])

{

  int i = 0,j = -1;

  next[0] = -1;

  while(i < p.length)

  {

    if(j == -1  || p.str[i] == p.str[j])

    {

      ++i,++j;

      next[i] = j;

    }

    else j = next[j];

  }

}

/\*快速模式匹配算法，请将函数补充完整\*/

int kmp(seqstring t,seqstring p,int next[])

{

  int i = 0, j = 0;

  while(i < t.length && j < p.length)

  {

    if(j == -1 || t.str[i] == p.str[j])

    {

      i++, j++;

    }

    else j = next[j];

  }

  if(j == p.length) return (i - p.length);

  else return -1;

}