北京郵電大學

实验报告

实验二： 线性表-栈-串综合应用实验

班级: 2022211320

学号: 2022212408

姓名: 胡宇杭

学院: 计算机学院 (国家示范性软件学院)

时间: 2023 年 11 月 9 日

目录

1 实验目的

1. 熟悉线性表的实现与应用;
2. 熟悉栈的实现与应用;
3. 熟悉串的实现与应用;
4. 培养根据实际问题合理选择数据结构的能力;
5. 学习自己查找相关资料以解决实际问题的能力。

$\oint \bigoplus$

2 实验环境

1. **操作系统：** *MacOS Ventura*
   - 版本：version 13.4.1
   - 架构：ARM
2. **C 环境：**
   - 编译器：Clang
   - IDE：Xcode
3. **编译器：** *Clang*
   - 版本：version 14.0.3
   - 编译命令 (开启 $O2$ 优化)：***clang -O2 -o output_file source_file.c***
   - 编译命令 (关闭 $O2$ 优化)：***clang -O0 -o output_file source_file.c***

3 实验内容 (简单行编辑程序)

3.1 问题描述:

文本编辑程序是利用计算机进行文字加工的基本软件工具，实现对文本文件的插入、删除等修改操作。限制这些这些操作以行为单位进行的编辑程序称为行编辑程序。

本实验将综合利用所学知识，设计并实现一个数据结构用来在计算机内存中建立一个行文本文件的映像，并实现后面要求的一些基础操作命令，为简化实验过程，可假设编辑处理的文本文件为 **C/C++** 源代码文件，且文件中均为 **ASCII** 码字符，不含中文等多字节字符。

3.2 要求:

- **功能 1**：实现打开并读入文件的命令，执行该命令可以将指定文件的内容读取到内存中自己建立的数据结构中;
- **功能 2**：在屏幕上输出指定范围的行，执行该命令可以将已经读入内存的文本文件的某几行显示到屏幕上;
- **功能 3**：行插入，执行该命令可以在指定位置插入行;
- **功能 4**：行删除，执行该命令可以删除一行或连续的几行;
- **功能 5**：行内文本插入，执行该命令可以在某一行的某个字符处插入一个或多个字符;
- **功能 6**：行内文本删除，执行该命令可以在某一行的某个字符处删除若干字符;
- **功能 7**：文本查找，执行该命令可从指定行开始查找某个字符串第一次出现的位置;
- **功能 8**：文本文件保存，执行该命令可以将修改后的文本内容写入到一个新的文件中。

3.3 附加要求:

检查源代码文本文件中的 **{}** **()** 是否匹配。

4 实验步骤

操作步骤 + 运行截图

4.1 代码解释

由于本次实验要求实现的功能繁多，因此采用分文件编写的方法，下面对每一个模块进行解释说明。

4.1.1 LINKSTRING

LinkString 用于存储每一行的字符串，由于代码文本编辑器需要频繁的进行插入和删除的功能，同时为了节省空间，这里采用不带头结点的链串进行实现。

```
7  typedef struct StringNode
8  {
9      char ch;
10     struct StringNode* next;
11 } StringNode;
12
13 typedef struct
14 {
15     StringNode* head;
16     int length;
17 } LinkString;
18
19 LinkString* init_linkstring(void);
20
21 void destory_linkstring(LinkString* ls);
22
23 bool insert_substring(LinkString* ls, int pos, const char* substr);
24
25 bool delete_substring(LinkString* ls, int pos, int len);
26
27 char* get_linkstring(LinkString* ls);
28
29 int find_substring(LinkString* ls, const char* pattern);
```

下面对每一个函数进行说明：

1. **LinkString* init_linkstring(void)**

   在初始化时，我们在堆区分配一块内存，并将头结点设为 **NULL** (因为是无头结点的链串)，并初始化 **length** 为 **0**。

   由于所有操作都是在常数时间内完成的，因此时间复杂度为 **O(1)**

```
6  LinkString* init_linkstring(void)
7  {
8      LinkString* ls = (LinkString*)malloc(sizeof(LinkString));
9      ls->head = NULL;
10     ls->length = 0;
11
12     return ls;
13 }
```

2. **void destory_linkstring(LinkString* ls)**

   该函数实现了销毁链串的功能，我们一次遍历每一个结点，将其释放掉在移动至下一个结点，最后将链串本身释放掉。

   函数需要遍历一次链串，因此时间复杂度为 **O(n)**

```
15 void destory_linkstring(LinkString* ls)
16 {
17     if (ls == NULL) return;
18
19     StringNode* curr = ls->head;
20
21     while (curr)
22     {
23         StringNode* temp = curr;
24         curr = curr->next;
25         free(temp);
26     }
27
28     free(ls);
29
30     return;
31 }
```

3. **bool insert_substring(LinkString* ls, int pos, const char* substr)**

   该函数实现了在特定位置插入子串的功能，同时做了特判，在输入的 **pos** 大于链串长度时，默认从串尾进行插入操作。在函数中，我们首先迭代至起始位置，同时由于链串不带头结点，因此需要利用 **prev** 记录当前结点的前驱结点。在插入操作时，我们需要查看前驱结点 **prev** 是否为空指针。如果为空指针，说明我们需要将结点插入至队头位置。

函数首先迭代至指定插入的结点，这部分的时间复杂度为 $\mathbf{O(n)}$，然后进行子串的插入操作，该过程需要遍历子串，故时间复杂度为 $\mathbf{O(m)}$。因此，总的时间复杂度为 $\mathbf{O(n+m)}$。

```
33 bool insert_substring(LinkString* ls, int pos, const char* substr)
34 {
35     if (pos < 0) return false;
36     pos = pos > ls->length ? ls->length : pos;
37 
38     StringNode* prev = NULL, * curr = ls->head;
39     for (int i = 0; i < pos; i++)
40     {
41         prev = curr;
42         curr = curr->next;
43     }
44 
45     for (int i = 0; substr[i] != '\0' && substr[i] != '\n'; i++)
46     {
47         StringNode* node = (StringNode*)malloc(sizeof(StringNode));
48         node->ch = substr[i];
49         node->next = curr;
50 
51         if (prev) prev->next = node;
52         else ls->head = node;
53 
54         prev = node;
55 
56         ls->length ++ ;
57     }
58 
59     return true;
60 }
```

4. **bool delete_substring(LinkString* ls, int pos, int len)**

   该函数实现了在特定位置删除指定长度子串的功能，与插入类似，我们对 **len** 越界的情况做了特殊处理，并且利用 **prev** 储存前驱结点，防止因当前结点为队头而错误地将队头指针变为野指针。

   函数的基本实现与插入函数类似，都需要迭代至需要删除的位置并删除指定长度的子串，因此时间复杂度也为 $\mathbf{O(n + m)}$

```
62 bool delete_substring(LinkString* ls, int pos, int len)
63 {
64     if (len < 0 || pos >= ls->length || len == 0) return false;
65     len = pos + len > ls->length ? ls->length - pos : len;
66 
67     StringNode* prev = NULL, * curr = ls->head;
68 
69     for (int i = 0; i < pos; i ++ )
70     {
71         prev = curr;
72         curr = curr->next;
73     }
74 
75     for (int i = 0; i < len; i++)
76     {
77         StringNode* temp = curr;
78         curr = curr->next;
79         free(temp);
80         ls->length -- ;
81     }
82 
83     if (prev) prev->next = curr;
84     else ls->head = curr;
85 
86     return true;
87 }
```

5. **char* get_linkstring(LinkString* ls)**

   该函数实现了遍历整个链串，并将其中元素提取出来，最后返回一个 **C** 的字符串的功能，同时使用了 **malloc** 开辟内存，防止链串长度超出 **res** 的最大长度导致截断。

   函数需要遍历整个链串，因此时间复杂度为 $\mathbf{O(n)}$。

```
89  char* get_linkstring(LinkString* ls)
90  {
91      char* res = (char*)malloc((ls->length + 1) * sizeof(char));
92      if (res == NULL) return NULL;
93
94      int index = 0;
95      StringNode* curr = ls->head;
96      while (curr)
97      {
98          res[index ++ ] = curr->ch;
99          curr = curr->next;
100     }
101
102     res[index] = '\0';
103
104     return res;
105 }
```

6. **int find_substring(LinkString* ls, const char* pattern)**

   该函数实现了使用 **KMP** 算法 (**next** 数组初始化 **0** 的版本) 完成了模式串与主串匹配的过程。

   我们知道 **KMP** 算法只需遍历一次主串就能实现查找功能，其中，**next** 数组的求解还需要遍历一次模式串，因此总的时间复杂度为 $\mathbf{O(n + m)}$。

```
107 int find_substring(LinkString* ls, const char* pattern)
108 {
109     if (ls->head == NULL) return -1;
110
111     int len = (int)strlen(pattern) - 1;
112     int* ne = (int*)malloc(len * sizeof(int));
113
114     memset(ne, 0, len * sizeof(int));
115
116     for (int i = 2, j = 0; i <= len; i ++ )
117     {
118         while (j && pattern[i] != pattern[j + 1]) j = ne[j];
119         if (pattern[i] == pattern[j + 1]) j ++ ;
120         ne[i] = j;
121     }
122
123     StringNode* curr = ls->head;
124     for (int i = 0, j = 0; curr; curr = curr->next, i ++ )
125     {
126         while (j && curr->ch != pattern[j + 1]) j = ne[j];
127         if (curr->ch == pattern[j + 1]) j ++ ;
128         if (j == len) return i - len;
129     }
130
131     free(ne);
132
133     return -1;
134 }
```

4.1.2 LINKLIST

将 **LinkString** 封装进 **LinkList**，这样我们就实现了最基本的文本存储访问功能：通过 **LinkList** 访问指定的行，然后通过 **LinkString** 访问指定的列。

```
typedef struct LNode
{
    LinkString* line;
    struct LNode* next;
} LNode;

typedef struct
{
    LNode* head;
    int length;
} LinkList;

LinkList* init_linklist(void);

LNode* get_row(LinkList* ll, int row_pos);

void destory_linklist(LinkList* ll);

bool insert_linklist(LinkList* ll, int row_pos, int col_pos, const char* s);

bool delete_linklist(LinkList* ll, int pos, int len);
```

下面对每一个函数进行说明：

1. **LinkList* init_linklist(void)**

   该函数的实现逻辑与链串的初始化一致，再次不再赘述。

   时间复杂度 **O(1)**

```
LinkList* init_linklist(void)
{
    LinkList* ll = (LinkList*)malloc(sizeof(LinkList));
    ll->head = NULL;
    ll->length = 0;

    return ll;
}
```

2. **LNode* get_row(LinkList* ll, int row_pos)**

   该函数用于通过指定的 **pos** 找到相应的结点。

   函数需要迭代至指定位置，因此时间复杂度为 **O(n)**

```
LNode* get_row(LinkList* ll, int row_pos)
{
    LNode* curr = ll->head;
    for (int i = 0; i < row_pos; i ++ ) curr = curr->next;
    return curr;
}
```

3. **void destory_linklist(LinkList* ll)**

   与 **LinkString** 类似，该函数用于销毁 **LinkList**，在此不再赘述。

   时间复杂度 $\mathbf{O(n)}$

```
13 void destory_linklist(LinkList* ll)
14 {
15     if (ll == NULL) return;
16 
17     LNode* curr = ll->head;
18 
19     while (curr)
20     {
21         LNode* temp = curr;
22         destory_linkstring(curr->line);
23         curr = curr->next;
24         free(temp);
25     }
26 
27     free(ll);
28 
29     return;
30 }
```

4. **bool insert_linklist(LinkList* ll, int row_pos, int col_pos, const char* s)**

   该函数实现了在指定位置插入行的功能，同时在函数中我们调用了之前的 **insert_substring()** 函数完成插入指定字符串的功能，防止了代码冗余的现象。

   函数需要迭代至指定位置，这部分的时间复杂度为 $\mathbf{O(n)}$，根据之前的分析，在链串中插入字符串的时间复杂度为 $\mathbf{O(n+m)(n=0)=O(m)}$。因此总的时间复杂度为 $\mathbf{O(n+m)}$。

```
bool insert_linklist(LinkList* ll, int row_pos, int col_pos, const char* s)
{
    if (row_pos < 0 || row_pos > ll->length) return false;

    LNode* prev = NULL, * curr = ll->head;
    for (int i = 0; i < row_pos; i++)
    {
        prev = curr;
        curr = curr->next;
    }

    LNode* node = (LNode*)malloc(sizeof(LNode));
    node->line = init_linkstring();

    insert_substring(node->line, col_pos, s);
    node->next = curr;

    if (prev) prev->next = node;
    else ll->head = node;

    ll->length ++ ;

    return true;
}
```

5. **bool delete_linklist(LinkList* ll, int pos, int len)**

   该函数与插入类似，都调用了 **LinkString** 中的函数，在此不再赘述。

   时间复杂度 $\mathbf{O(n + m)}$

```
bool delete_linklist(LinkList* ll, int pos, int len)
{
    if (len < 0 || pos >= ll->length || len == 0) return false;
    len = pos + len > ll->length ? ll->length - pos : len;

    LNode* prev = NULL, * curr = ll->head;
    for (int i = 0; i < pos; i ++ )
    {
        prev = curr;
        curr = curr->next;
    }

    for (int i = 0; i < len; i++)
    {
        LNode* temp = curr;
        curr = curr->next;
        destory_linkstring(temp->line);
        free(temp);
        ll->length -- ;
    }

    if (prev) prev->next = curr;
    else ll->head = curr;

    return true;
}
```

4.1.3 OPRTSTACK

OprtStack 用于实现附加要求中的运算符匹配算法，由于之前没写过数组版本的栈，所以在这就不再写链栈了 (~~绝对不是因为懒~~)。

```
11 typedef struct
12 {
13     char* base;
14     int top;
15     int stack_size;
16 } OpStack;
17
18 OpStack* init_stack(void);
19
20 void destory_stack(OpStack* s);
21
22 bool extend_stack(OpStack* s);
23
24 bool push(OpStack* s, char ch);
25
26 bool pop(OpStack* s);
27
28 char top(OpStack* s);
29
30 bool empty(OpStack* s);
```

1. **OpStack* init_stack(void)**

   初始化时为栈开辟指定的内存空间，由于使用动态数组实现，因此初始化 **top = -1** 代表一开始栈是空的。

```
5  OpStack* init_stack(void)
6  {
7      OpStack* s = (OpStack*)malloc(sizeof(OpStack));
8      s->base = (char*)malloc(STACK_INIT_SIZE * sizeof(char));
9      s->top = -1;
10     s->stack_size = STACK_INIT_SIZE;
11
12     return s;
13 }
```

2. **bool extend_stack(OpStack* s)**

   该函数用于实现在栈满时对栈容量的扩展，保证不会因为栈的容量问题导致判断运算符失败。为了防止 **realloc** 函数开辟空间失败返回空指针导致原空间无法访问造成内存泄漏，这里创建临时变量接受新的内存地址。

   由于 **realloc** 函数并不是在原有内存空间上进行扩展，而是开辟一个新的空间，并将原有数据拷贝过去，因此时间复杂度为 **O(n)**

```
bool extend_stack(OpStack* s)
{
    char* newBase = NULL;
    int newSize = s->stack_size + STACK_EXTEND_SIZE;
    newBase = (char*)realloc(s->base, newSize * sizeof(char));
    if (!newBase) return false;

    s->base = newBase;
    s->stack_size = newSize;

    return true;
}
```

3. 其他函数的实现比较简单，这里不再赘述。

4.1.4 TEXTEDITOR

在完成了 **LinkList** 的编写后，我们将其封装进 **TextEditor** 中，方便调用。

```
typedef struct
{
    LinkList* text;
} TextEditor;

TextEditor* init_editor(void);

void destory_editor(TextEditor* editor, int oprt);

void show_menu(void);

bool read_file(TextEditor** editor, const char* file_name);

bool save_file(TextEditor* editor, const char* file_name);

void print_text(TextEditor* editor, int start, int end);

bool check_oprt(TextEditor* editor);
```

下面对每一个函数进行说明：

1. **TextEditor* init_editor** 与 **void destory_editor(TextEditor* editor, int oprt)**

   在 **init_editor** 中，我们为其在堆区开辟了一片空间，并使用之前编写的 **init_linklist** 初始化封装的链表。时间复杂度 **O(1)**

   在 **destory_editor** 中，我们调用之前编写的 **destory_linklist** 函数销毁链表，但由于在某些操作中，我们可能并不希望将 **editor** 也释放掉，所以增加了变量

oprt，当 **oprt = 0** 时，才会执行 **free(editor)** 操作，彻底销毁整个 **TextEditor**。

时间复杂度 **O(nm)**

```
TextEditor* init_editor(void)
{
    TextEditor* editor = (TextEditor*)malloc(sizeof(TextEditor));
    editor->text = init_linklist();

    return editor;
}

void destory_editor(TextEditor* editor, int oprt)
{
    if (editor == NULL) return;

    destory_linklist(editor->text);

    if (oprt == 0) free(editor);

    return;
}
```

2. **void show_menu(void)**

   该函数用于打印可执行操作的选项。

3. **bool read_file(TextEditor** editor, const char* file_name)**

   该函数用于实现读取文件的功能。由于我们可能在程序运行时从不同的文件中读入数据，因此每次读入前都需要将当前的数据释放掉，这里使用了之前编写的 **destory_editor** 函数，同时我们希望能继续使用 **editor** 存储新的数据，因此 **oprt = 1**。

   需要注意的是，由于此处涉及到对指针分配内存，因此函数的参数应该是一个二重指针。

   由于需要将整个文本读入，因此时间复杂度为 **O(nm)**

```
bool read_file(TextEditor** editor, const char* file_name)
{
    destory_editor((*editor), 1);
    *editor = init_editor();
    FILE* file = fopen(file_name, "rb");
    if (file == NULL) return false;

    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), file)) insert_linklist((*editor)->text, (*editor)->text->length, 0, buffer);

    fclose(file);

    return true;
}
```

4. **bool save_file(TextEditor* editor, const char* file_name)**

   该函数用于实现将数据写入文件的功能，我们遍历链表的每一行，同时调用 **get_linkstring** 函数，将每一行存储的链串转换成字符串并存储到文件中。

   在函数中我们需要遍历链表的每一个结点，所以外层循环的迭代次数为 **n**，根据之前的分析，每次调用 **get_linkstring** 函数的时间复杂度为 **O(m)**，因此总的时间复杂度为 **O(nm)**

```
82  bool save_file(TextEditor* editor, const char* file_name)
83  {
84      FILE* file = fopen(file_name, "wb");
85      if (file == NULL) return false;
86
87      LNode* curr = editor->text->head;
88      while (curr)
89      {
90          LinkString* ls = curr->line;
91          char* s = get_linkstring(ls);
92
93          if (s != NULL)
94          {
95              fputs(s, file);
96              free(s);
97          }
98
99          fputs("\n", file);
100         curr = curr->next;
101     }
102
103     fclose(file);
104
105     return true;
106 }
```

5. **void print_text(TextEditor* editor, int start, int end)**

   该函数用于打印指定范围的行，在打印时，会显示当前行数，方面对其进行操作。与之前类似，当输入的数据大于行数是默认打印至末尾。我们先使用 **get_row** 函数获取起始结点，依次遍历剩下的结点，每次调用 **get_linkstring** 函数获得字符串并打印。

   根据之前的分析，**get_row** 函数和外层循环的时间复杂度为 **O(n)**，调用 **get_linkstring** 函数的时间复杂度为 **O(m)**，总的时间复杂度为 **O(nm)**

```
42 void print_text(TextEditor* editor, int start, int end)
43 {
44     if (editor->text->head == NULL || start >= editor->text->length) return;
45     end = end >= editor->text->length ? editor->text->length - 1 : end;
46 
47     LNode* curr = get_row(editor->text, start);
48 
49     for (int i = 0; i <= end - start; i ++ )
50     {
51         printf("%d  ", start + i);
52         if (curr->line->head)
53         {
54             char* s = (char*)malloc((curr->line->length + 1) * sizeof(char));
55             s = get_linkstring(curr->line);
56             printf("%s", s);
57             free(s);
58         }
59 
60         puts("");
61 
62         curr = curr->next;
63     }
64 
65     return;
66 }
```

6. **bool check_oprt(TextEditor* editor)**

   该函数用于实现检测运算符是否匹配。具体实现原理是通过利用栈的后进先出性质，按行优先的顺序访问整个文本，将所有的 **{**, **(** 运算符压入栈中，并查看之后出现的 **}**, **)** 运算符是否与栈顶元素匹配。在实现基本功能的同时，通过一定的算法将运算符是否在头文件中、**comment** 中、‘’ 中的情况过滤。

   由于需要遍历一遍文本，因此时间复杂度为 **O(nm)**

```
bool check_oprt(TextEditor* editor)
{
    OpStack* bracketStack = init_stack();
    bool isMultiComment = false;

    LNode* curr = editor->text->head;

    while (curr)
    {
        StringNode* node = curr->line->head;
        bool isQuote[2];
        memset(isQuote, false, sizeof(isQuote));

        if (node && node->ch == '#')
        {
            curr = curr->next;
            continue;
        }

        while (node)
        {
            char ch[3] = "\"\'";
            bool isMismatch = false;
            bool isSkip = false;

            if (node->ch == '/' && node->next && node->next->ch == '/') break;

            if (node->ch == '/' && node->next && node->next->ch == '*') isMultiComment = true;
            else if (node->ch == '*' && node->next && node->next->ch == '/') isMultiComment = false;
            else if (isMultiComment) isSkip = true;

            for (int i = 0; i < 2; i ++ )
            {
                if (node->ch == ch[i] && !isQuote[i]) isQuote[i] = true;
                else if (node->ch == ch[i] && isQuote[i]) isQuote[i] = false;
                else if (isQuote[i]) isSkip = true;
            }

            if (isSkip)
            {
                node = node->next;
                continue;
            }

            if (node->ch == '{' || node->ch == '(') push(bracketStack, node->ch);
            else if (node->ch == '}')
            {
                if (empty(bracketStack)) isMismatch = true;
                else if (top(bracketStack) != '{') isMismatch = true;
                else pop(bracketStack);
            }
            else if (node->ch == ')')
            {
                if (empty(bracketStack)) isMismatch = true;
                else if (top(bracketStack) != '(') isMismatch = true;
                else pop(bracketStack);
            }

            if (isMismatch)
            {
                destory_stack(bracketStack);
                return false;
            }

            node = node->next;
        }

        curr = curr->next;
    }

    bool flag = empty(bracketStack);
    destory_stack(bracketStack);

    return flag;
}
```

4.2 运行截图

1. **运行程序**

   运行程序，会先给出可用操作，并提示操作。

```
-----------------------------------
---------- 0. Exit ----------------
---------- 1. Load File -----------
---------- 2. Print Text ----------
---------- 3. Insert Line ---------
---------- 4. Delete Line ---------
---------- 5. Insert Text ---------
---------- 6. Delete Text----------
---------- 7. Search Text ---------
---------- 8. Check Oprt ----------
---------- 9. Save File -----------
-----------------------------------
Please enter the oprt:
```

2. **操作一：读入文件**

   输入 **1**，程序提示输入文件名，如果文件不存在，则会提示失败；相反，会提示成功

```
Please enter the oprt:
1
Please enter the file name.
error.dat
Load Failed.
```

```
Please enter the oprt:
1
Please enter the file name.
save.dat
Load Success.
```

3. **操作二：输出指定范围的行**

   输入 **2**，程序提示输入范围，在越界时会默认输出至末尾

```
Please enter the oprt:
2
Please enter the start and end position.
0 1
0  #include <iostream>
1  using namespace std;
```

```
Please enter the oprt:
2
Please enter the start and end position.
0 114514
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, world" << endl;
6
7   return 0;
8  }
```

4. **操作三：行插入**

   输入 **3**，程序提示输入指定行，在越界时默认在末尾插入

```
Please enter the oprt:
3
Please enter the row you want to insert.
6
Please enter the words.
    cout << "Hello, bupt" << endl;
Insert Success.
```

查看插入后的效果

```
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, world" << endl;
6   cout << "Hello, bupt" << endl;
7
8   return 0;
9  }
```

5. **操作四：行删除**

输入 **4**，程序提示输入指定行和长度，在越界时默认从起始位置开始全部删除

```
Please enter the oprt:
4
Please enter the row you want to delete, and the length.
5
1
Delete Success.
```

查看删除后的效果

```
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bupt" << endl;
6
7   return 0;
8  }
```

6. **操作五：行内文本插入**

输入 **5**，程序提示输入指定行和列，在越界时默认在末尾插入

```
Please enter the oprt:
5
Please enter the row and column you want to insert.
5 20
20
Please enter the words.
 ,world
Insert Success.
```

查看插入后的效果

```
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bup ,worldt" << endl;
6
7   return 0;
8  }
```

7. **操作六：行内文本删除**

   输入 **6**，程序提示输入指定行和列，以及长度，在越界时默认从起始位置开始全部删除

```
Please enter the oprt:
6
Please enter the row and column you want to delete, and the length.
5 20 7
Delete Success.
```

   查看插入后的效果

```
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bupt" << endl;
6
7   return 0;
8  }
```

8. **操作七：文本查找**

   输入 **7**，程序提示输入指定的行和想查找的字符串，如果成功，则返回字符串的起始位置 (下标从 **0** 开始)

```
Please enter the oprt:
7
Please enter the row.
5
Please enter the pattern string.
bupt
17
```

9. **操作八：检查运算符匹配**

输入 **8**，程序自动检查运算符是否匹配并输出

```
Please enter the oprt:
8
OK
```

```
Please enter the oprt:
8
ERROR
```

```
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bupt" << endl;
6
7   return 0;
8  }
```

```
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bupt" << endl;
6  }
7   return 0;
8  }
```

包含所有特殊情况的例子如下

```
0  #i{nclude <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << '{' << "H{ello, bupt" << endl;
6  //}
7   return 0;
8  }
-------------------------------------
---------- 0. Exit -----------------
---------- 1. Load File ------------
---------- 2. Print Text -----------
---------- 3. Insert Line ----------
---------- 4. Delete Line ----------
---------- 5. Insert Text ----------
---------- 6. Delete Text-----------
---------- 7. Search Text ----------
---------- 8. Check Oprt -----------
---------- 9. Save File ------------
-------------------------------------
Please enter the oprt:
8
OK
```

10. **操作九：保存文件**

输入 **9**，程序保存文件，结束运行

重新运行程序，将之前保存的文件读入并打印

```
Please enter the oprt:
9
Please enter the file name.
save.dat
Save Success.
```

```
Please enter the oprt:
1
Please enter the file name.
save.dat
Load Success.
-----------------------------------
---------- 0. Exit -----------------
---------- 1. Load File ------------
---------- 2. Print Text -----------
---------- 3. Insert Line ----------
---------- 4. Delete Line ----------
---------- 5. Insert Text ----------
---------- 6. Delete Text-----------
---------- 7. Search Text ----------
---------- 8. Check Oprt -----------
---------- 9. Save File ------------
-----------------------------------
Please enter the oprt:
2
Please enter the start and end position.
0 111
0  #include <iostream>
1  using namespace std;
2
3  int main()
4  {
5   cout << "Hello, bupt" << endl;
6  }
7   return 0;
8  }
```

5 实验分析和总结

5.1 数据结构分析

由于文本编辑器需要实现频繁的插入、删除操作，所以我选择了链串 + 链表的实现方法，从整体上来看，这个数据结构很好地实现实验所提出的各项要求，但仍有不足：

- 链串在应对大规模数据时的运行效率十分低下，只能处理较轻量的文本
- 由于链串的内存区域并不连续，不能善加利用缓存，导致在遍历时的运行时间要远大于其他数据结构

5.2 算法复杂度分析

在代码解释时说明了各函数的时间复杂度，故此处只对其进行汇总

- 操作一：根据之前的分析，时间复杂度为 $\mathbf{O(nm)}$

```
case 1:
{
    char file_name[20] = "";
    puts("Please enter the file name.");
    scanf("%s", file_name);
    if (read_file(&editor, file_name)) puts("Load Success.");
    else puts("Load Failed.");
    break;
}
```

- 操作二：根据之前的分析，时间复杂度为 $\mathbf{O(nm)}$

```
case 2:
{
    int start, end;
    puts("Please enter the start and end position.");
    scanf("%d%d", &start, &end);
    print_text(editor, start, end);
    break;
}
```

- 操作三：根据之前的分析，时间复杂度为 $\mathbf{O(n+m)}$

```
case 3:
{
    int pos = 0;
    char buffer[1024];
    puts("Please enter the row you want to insert.");
    scanf("%d", &pos);
    while (getchar() != '\n');

    puts("Please enter the words.");
    fgets(buffer, 1024, stdin);
    if (insert_linklist(editor->text, pos, 0, buffer)) puts("Insert Success.");
    else puts("Insert Failed.");
    break;
}
```

- 操作四：根据之前的分析，时间复杂度为 $\mathbf{O(n+m)}$

```
case 4:
{
    int pos = 0, len = 0;
    puts("Please enter the row you want to delete, and the length.");
    scanf("%d%d", &pos, &len);
    if (delete_linklist(editor->text, pos, len)) puts("Delete Success.");
    else puts("Delete Failed.");
    break;
}
```

- 操作五：根据之前的分析，时间复杂度为 $\mathbf{O(n+m)}$

```
case 5:
{
    int row_pos = 0, col_pos = 0;
    char buffer[1024];
    puts("Please enter the row and column you want to insert.");
    scanf("%d%d", &row_pos, &col_pos);
    while (getchar() != '\n');

    puts("Please enter the words.");
    fgets(buffer, 1024, stdin);
    if (insert_substring(get_row(editor->text, row_pos)->line, col_pos, buffer)) puts("Insert Success.");
    else puts("Insert Failed.");
    break;
}
```

- 操作六：根据之前的分析，时间复杂度为 $\mathbf{O(n+m)}$

```
case 6:
{
    int row_pos = 0, col_pos = 0, len = 0;
    puts("Please enter the row and column you want to delete, and the length.");
    scanf("%d%d%d",&row_pos, &col_pos, &len);
    if (delete_substring(get_row(editor->text, row_pos)->line, col_pos, len)) puts("Delete Success.");
    else puts("Delete Failed.");
    break;
}
```

- 操作七：根据之前的分析，时间复杂度为 $\mathbf{O(n+m)}$

```
case 7:
{
    int row_pos = 0;
    char pattern[1024];
    puts("Please enter the row.");
    scanf("%d",&row_pos);
    while (getchar() != '\n');

    puts("Please enter the pattern string.");
    fgets(pattern, 1024, stdin);
    pattern[strlen(pattern) - 1] = '\0';
    printf("%d\n", find_substring(get_row(editor->text, row_pos)->line, pattern));
    break;
}
```

- 操作八：根据之前的分析，时间复杂度为 $\mathbf{O(nm)}$

```
case 8:
{
    if (check_oprt(editor)) puts("OK");
    else puts("ERROR");
    break;
}
```

- 操作九：根据之前的分析，时间复杂度为 **O(nm)**

```
case 9:
{
    puts("Please enter the file name.");
    char file_name[20] = "";
    scanf("%s", file_name);
    if (save_file(editor, file_name)) puts("Save Success.");
    else puts("Save Failed.");
    break;
}
```

5.3 总结

通过这次实验，我学习了如何编写和使用链表、链串和栈来解决实际问题；在输入数据时，考虑到 **scanf(“%s”)** 不能读取空格，我使用了 **fgets()** 函数，但在读入时程序总会崩溃。通过查阅资料，我了解了缓冲区及相关概念，并解决了该问题。但这次实验仍有很多小问题，比如代码不够简洁，代码的复用性不强等，导致最后整体代码量比较大。

6 程序源代码

主函数

```
#include "texteditor.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, const char * argv[]) {
    TextEditor* editor = init_editor();
    int oprt = 0;

    do
    {
        show_menu();
        scanf("%d", &oprt);

        switch (oprt)
        {
            case 0:
            {
                exit(0);
            }
            case 1:
            {
                char file_name[20] = "";
                puts("Please enter the file name.");
                scanf("%s", file_name);
                if (read_file(&editor, file_name)) puts("Load Success.");
                else puts("Load Failed.");
                break;
            }
            case 2:
            {
                int start, end;
                puts("Please enter the start and end position.");
                scanf("%d%d", &start, &end);
                print_text(editor, start, end);
                break;
```

```
37              }
38              case 3:
39              {
40                  int pos = 0;
41                  char buffer[1024];
42                  puts("Please enter the row you want to insert.");
43                  scanf("%d", &pos);
44                  while (getchar() != '\n');
45
46                  puts("Please enter the words.");
47                  fgets(buffer, 1024, stdin);
48                  if (insert_linklist(editor->text, pos, 0, buffer)) puts("Insert Success.");
49                  else puts("Insert Failed.");
50                  break;
51              }
52              case 4:
53              {
54                  int pos = 0, len = 0;
55                  puts("Please enter the row you want to delete, and the length.");
56                  scanf("%d%d", &pos, &len);
57                  if (delete_linklist(editor->text, pos, len)) puts("Delete Success.");
58                  else puts("Delete Failed.");
59                  break;
60              }
61              case 5:
62              {
63                  int row_pos = 0, col_pos = 0;
64                  char buffer[1024];
65                  puts("Please enter the row and column you want to insert.");
66                  scanf("%d%d ",&row_pos, &col_pos);
67                  while (getchar() != '\n');
68
69                  puts("Please enter the words.");
70                  fgets(buffer, 1024, stdin);
71                  if (insert_substring(get_row(editor->text, row_pos)->line, col_pos, buffer)) puts("
                        Insert Success.");
72                  else puts("Insert Failed.");
73                  break;
74              }
75              case 6:
76              {
```

```
77                  int row_pos = 0, col_pos = 0, len = 0;
78                  puts("Please enter the row and column you want to delete, and the length.");
79                  scanf("%d%d%d",&row_pos, &col_pos, &len);
80                  if (delete_substring(get_row(editor->text, row_pos)->line, col_pos, len)) puts("
                        Delete Success.");
81                  else puts("Delete Failed.");
82                  break;
83              }
84              case 7:
85              {
86                  int row_pos = 0;
87                  char pattern[1024];
88                  puts("Please enter the row.");
89                  scanf("%d",&row_pos);
90                  while (getchar() != '\n');
91
92                  puts("Please enter the pattern string.");
93                  fgets(pattern, 1024, stdin);
94                  pattern[strlen(pattern) - 1] = '\0';
95                  printf("%d\n", find_substring(get_row(editor->text, row_pos)->line, pattern));
96                  break;
97              }
98              case 8:
99              {
100                 puts("Please enter the file name.");
101                 char file_name[20] = "";
102                 scanf("%s", file_name);
103                 if (save_file(editor, file_name)) puts("Save Success.");
104                 else puts("Save Failed.");
105                 break;
106             }
107             case 9:
108             {
109                 if (check_oprt(editor)) puts("OK");
110                 else puts("ERROR");
111                 break;
112             }
113             default:
114                 puts("Mistaken input, try again");
115                 break;
116         }
```

```
117     } while (oprt);
118
119
120     return 0;
121 }
```

linkstring.h

```
1  #pragma once
2  #ifndef LINKSTRING_H
3  #define LINKSTRING_H
4
5  #include <stdbool.h>
6
7  typedef struct StringNode
8  {
9      char ch;
10     struct StringNode* next;
11 } StringNode;
12
13 typedef struct
14 {
15     StringNode* head;
16     int length;
17 } LinkString;
18
19 LinkString* init_linkstring(void);
20
21 void destory_linkstring(LinkString* ls);
22
23 bool insert_substring(LinkString* ls, int pos, const char* substr);
24
25 bool delete_substring(LinkString* ls, int pos, int len);
26
27 char* get_linkstring(LinkString* ls);
28
29 int find_substring(LinkString* ls, const char* pattern);
30
31 #endif // !LINKSTRING_H
```

linkstring

```
#include "linkstring.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

LinkString* init_linkstring(void)
{
    LinkString* ls = (LinkString*)malloc(sizeof(LinkString));
    ls->head = NULL;
    ls->length = 0;

    return ls;
}

void destory_linkstring(LinkString* ls)
{
    if (ls == NULL) return;

    StringNode* curr = ls->head;

    while (curr)
    {
        StringNode* temp = curr;
        curr = curr->next;
        free(temp);
    }

    free(ls);

    return;
}

bool insert_substring(LinkString* ls, int pos, const char* substr)
{
    if (pos < 0) return false;
    pos = pos > ls->length ? ls->length : pos;

    StringNode* prev = NULL, * curr = ls->head;
    for (int i = 0; i < pos; i++)
    {
```

```
41          prev = curr;
42          curr = curr->next;
43      }
44
45      for (int i = 0; substr[i] != '\0' && substr[i] != '\n'; i++)
46      {
47          StringNode* node = (StringNode*)malloc(sizeof(StringNode));
48          node->ch = substr[i];
49          node->next = curr;
50
51          if (prev) prev->next = node;
52          else ls->head = node;
53
54          prev = node;
55
56          ls->length ++ ;
57      }
58
59      return true;
60  }
61
62  bool delete_substring(LinkString* ls, int pos, int len)
63  {
64      if (len < 0 || pos >= ls->length || len == 0) return false;
65      len = pos + len > ls->length ? ls->length - pos : len;
66
67      StringNode* prev = NULL, * curr = ls->head;
68
69      for (int i = 0; i < pos; i ++ )
70      {
71          prev = curr;
72          curr = curr->next;
73      }
74
75      for (int i = 0; i < len; i++)
76      {
77          StringNode* temp = curr;
78          curr = curr->next;
79          free(temp);
80          ls->length -- ;
81      }
```

```
82
83      if (prev) prev->next = curr;
84      else ls->head = curr;
85
86      return true;
87  }
88
89  char* get_linkstring(LinkString* ls)
90  {
91      char* res = (char*)malloc((ls->length + 1) * sizeof(char));
92      if (res == NULL) return NULL;
93
94      int index = 0;
95      StringNode* curr = ls->head;
96      while (curr)
97      {
98          res[index ++ ] = curr->ch;
99          curr = curr->next;
100     }
101
102     res[index] = '\0';
103
104     return res;
105 }
106
107 int find_substring(LinkString* ls, const char* pattern)
108 {
109     if (ls->head == NULL) return -1;
110
111     int len = (int)strlen(pattern) - 1;
112     int* ne = (int*)malloc(len * sizeof(int));
113
114     memset(ne, 0, len * sizeof(int));
115
116     for (int i = 2, j = 0; i <= len; i ++ )
117     {
118         while (j && pattern[i] != pattern[j + 1]) j = ne[j];
119         if (pattern[i] == pattern[j + 1]) j ++ ;
120         ne[i] = j;
121     }
122
```

```
123     StringNode* curr = ls->head;
124     for (int i = 0, j = 0; curr; curr = curr->next, i ++ )
125     {
126         while (j && curr->ch != pattern[j + 1]) j = ne[j];
127         if (curr->ch == pattern[j + 1]) j ++ ;
128         if (j == len) return i - len;
129     }
130 
131     free(ne);
132 
133     return -1;
134 }
```

linklist.h

```
1  #pragma once
2  #ifndef LINKLIST_H
3  #define LINKLIST_H
4  
5  #include "linkstring.h"
6  
7  typedef struct LNode
8  {
9      LinkString* line;
10     struct LNode* next;
11 } LNode;
12 
13 typedef struct
14 {
15     LNode* head;
16     int length;
17 } LinkList;
18 
19 LinkList* init_linklist(void);
20 
21 LNode* get_row(LinkList* ll, int row_pos);
22 
23 void destory_linklist(LinkList* ll);
24 
25 bool insert_linklist(LinkList* ll, int row_pos, int col_pos, const char* s);
```

```
26
27 bool delete_linklist(LinkList* ll, int pos, int len);
28
29 #endif // !LINKLIST_H
```

linklist

```
1  #include "linklist.h"
2  #include <stdlib.h>
3
4  LinkList* init_linklist(void)
5  {
6      LinkList* ll = (LinkList*)malloc(sizeof(LinkList));
7      ll->head = NULL;
8      ll->length = 0;
9
10     return ll;
11 }
12
13 void destory_linklist(LinkList* ll)
14 {
15     if (ll == NULL) return;
16
17     LNode* curr = ll->head;
18
19     while (curr)
20     {
21         LNode* temp = curr;
22         destory_linkstring(curr->line);
23         curr = curr->next;
24         free(temp);
25     }
26
27     free(ll);
28
29     return;
30 }
31
32 LNode* get_row(LinkList* ll, int row_pos)
33 {
```

```
34      LNode* curr = ll->head;
35      for (int i = 0; i < row_pos; i ++ ) curr = curr->next;
36      return curr;
37  }
38
39  bool insert_linklist(LinkList* ll, int row_pos, int col_pos, const char* s)
40  {
41      if (row_pos < 0 || row_pos > ll->length) return false;
42
43      LNode* prev = NULL, * curr = ll->head;
44      for (int i = 0; i < row_pos; i++)
45      {
46          prev = curr;
47          curr = curr->next;
48      }
49
50      LNode* node = (LNode*)malloc(sizeof(LNode));
51      node->line = init_linkstring();
52
53      insert_substring(node->line, col_pos, s);
54      node->next = curr;
55
56      if (prev) prev->next = node;
57      else ll->head = node;
58
59      ll->length ++ ;
60
61      return true;
62  }
63
64  bool delete_linklist(LinkList* ll, int pos, int len)
65  {
66      if (len < 0 || pos >= ll->length || len == 0) return false;
67      len = pos + len > ll->length ? ll->length - pos : len;
68
69      LNode* prev = NULL, * curr = ll->head;
70      for (int i = 0; i < pos; i ++ )
71      {
72          prev = curr;
73          curr = curr->next;
74      }
```

```
75
76      for (int i = 0; i < len; i++)
77      {
78          LNode* temp = curr;
79          curr = curr->next;
80          destory_linkstring(temp->line);
81          free(temp);
82          ll->length -- ;
83      }
84
85      if (prev) prev->next = curr;
86      else ll->head = curr;
87
88      return true;
89  }
```

oprtstack.h

```
1   #pragma once
2   #ifndef OPRTSTACK_H
3   #define OPRTSTACK_H
4
5   #include <stdbool.h>
6   #ifndef STACK_INIT_SIZE
7   #define STACK_INIT_SIZE 50
8
9   #ifndef STACK_EXTEND_SIZE
10  #define STACK_EXTEND_SIZE 20
11  typedef struct
12  {
13      char* base;
14      int top;
15      int stack_size;
16  } OpStack;
17
18  OpStack* init_stack(void);
19
20  void destory_stack(OpStack* s);
21
22  bool extend_stack(OpStack* s);
```

```
23
24 bool push(OpStack* s, char ch);
25
26 bool pop(OpStack* s);
27
28 char top(OpStack* s);
29
30 bool empty(OpStack* s);
31
32 #endif // !STACK_EXTEND_SIZE
33 #endif // !STACK_INIT_SIZE
34 #endif // !OPRTSTACK_H
```

texteditor

```
1  #include "oprtstack.h"
2  #include <string.h>
3  #include <stdlib.h>
4
5  OpStack* init_stack(void)
6  {
7      OpStack* s = (OpStack*)malloc(sizeof(OpStack));
8      s->base = (char*)malloc(STACK_INIT_SIZE * sizeof(char));
9      s->top = -1;
10     s->stack_size = STACK_INIT_SIZE;
11
12     return s;
13 }
14
15 bool extend_stack(OpStack* s)
16 {
17     char* newBase = NULL;
18     int newSize = s->stack_size + STACK_EXTEND_SIZE;
19     newBase = (char*)realloc(s->base, newSize * sizeof(char));
20     if (!newBase) return false;
21
22     s->base = newBase;
23     s->stack_size = newSize;
24
25     return true;
```

```
26  }
27
28  void destory_stack(OpStack* s)
29  {
30      if (s->base) free(s->base);
31      free(s);
32
33      return;
34  }
35
36  bool push(OpStack* s, char ch)
37  {
38      if (s->top >= s->stack_size)
39          if (!extend_stack(s)) return false;
40
41      s->base[ ++ s->top] = ch;
42
43      return true;
44  }
45
46  bool pop(OpStack* s)
47  {
48      if (s->top < 0) return false;
49
50      s->top -- ;
51
52      return true;
53  }
54
55  char top(OpStack* s)
56  {
57      if (s->top >= 0) return s->base[s->top];
58      return '\0';
59  }
60
61  bool empty(OpStack* s)
62  {
63      return s->top < 0;
64  }
```

texteditor.h

```h
#pragma once
#ifndef TEXTEDITOR_H
#define TEXTEDITOR_H

#include "linklist.h"
#include "oprtstack.h"

typedef struct
{
    LinkList* text;
} TextEditor;

TextEditor* init_editor(void);

void destory_editor(TextEditor* editor, int oprt);

void show_menu(void);

bool read_file(TextEditor** editor, const char* file_name);

bool save_file(TextEditor* editor, const char* file_name);

void print_text(TextEditor* editor, int start, int end);

bool check_oprt(TextEditor* editor);

#endif // !TEXTEDITOR_H
```

texteditor

```c
#include "texteditor.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

TextEditor* init_editor(void)
{
    TextEditor* editor = (TextEditor*)malloc(sizeof(TextEditor));
    editor->text = init_linklist();
```

```
10
11      return editor;
12  }
13
14  void destory_editor(TextEditor* editor, int oprt)
15  {
16      if (editor == NULL) return;
17
18      destory_linklist(editor->text);
19
20      if (oprt == 0) free(editor);
21
22      return;
23  }
24
25  void show_menu(void)
26  {
27      printf("-------------------------------------\n");
28      printf("---------- 1. Load File ------------\n");
29      printf("---------- 2. Print Text -----------\n");
30      printf("---------- 3. Insert Line ----------\n");
31      printf("---------- 4. Delete Line ----------\n");
32      printf("---------- 5. Insert Text ----------\n");
33      printf("---------- 6. Delete Text-----------\n");
34      printf("---------- 7. Search Text ----------\n");
35      printf("---------- 8. Save File ------------\n");
36      printf("---------- 9. Check Oprt -----------\n");
37      printf("-------------------------------------\n");
38
39      return;
40  }
41
42  void print_text(TextEditor* editor, int start, int end)
43  {
44      if (editor->text->head == NULL || start >= editor->text->length) return;
45      end = end >= editor->text->length ? editor->text->length - 1 : end;
46
47      LNode* curr = editor->text->head;
48      for (int i = 0; i < start; i ++ ) curr = curr->next;
49
50      for (int i = 0; i <= end - start; i ++ )
```

```
51      {
52          if (curr->line->head)
53          {
54              char* s = (char*)malloc((curr->line->length + 1) * sizeof(char));
55              s = get_linkstring(curr->line);
56              printf("%s", s);
57          }
58
59          puts("");
60
61          curr = curr->next;
62      }
63
64      return;
65  }
66
67  bool read_file(TextEditor** editor, const char* file_name)
68  {
69      destory_editor((*editor), 1);
70      *editor = init_editor();
71      FILE* file = fopen(file_name, "r");
72      if (file == NULL) return false;
73
74      char line[1024];
75      while (fgets(line, sizeof(line), file) != NULL)
76      {
77          insert_linklist((*editor)->text, (*editor)->text->length, 0, line);
78      }
79
80      fclose(file);
81
82      return true;
83  }
84
85  bool save_file(TextEditor* editor, const char* file_name)
86  {
87      FILE* file = fopen(file_name, "w");
88      if (file == NULL) return false;
89
90      LNode* curr = editor->text->head;
91      while (curr)
```

```
92      {
93          LinkString* ls = curr->line;
94          char* s = get_linkstring(ls);
95
96          if (s != NULL)
97          {
98              fputs(s, file);
99              free(s);
100         }
101
102         fputs("\n", file);
103
104         curr = curr->next;
105     }
106
107     fclose(file);
108
109     return true;
110 }
111
112 bool check_oprt(TextEditor* editor)
113 {
114     OpStack* bracketStack = init_stack();
115     bool isMultiComment = false;
116
117     LNode* curr = editor->text->head;
118
119     while (curr)
120     {
121         StringNode* node = curr->line->head;
122         bool isQuote[2];
123         memset(isQuote, false, sizeof(isQuote));
124
125         if (node && node->ch == '#')
126         {
127             curr = curr->next;
128             continue;
129         }
130
131         while (node)
132         {
```

```
133             char ch[3] = "\"\'";
134             bool isMismatch = false;
135             bool isSkip = false;
136 
137             if (node->ch == '/' && node->next && node->next->ch == '/') break;
138 
139             if (node->ch == '/' && node->next && node->next->ch == '*') isMultiComment = true;
140             else if (node->ch == '*' && node->next && node->next->ch == '/') isMultiComment = false;
141             else if (isMultiComment) isSkip = true;
142 
143             for (int i = 0; i < 2; i ++ )
144             {
145                 if (node->ch == ch[i] && !isQuote[i]) isQuote[i] = true;
146                 else if (node->ch == ch[i] && isQuote[i]) isQuote[i] = false;
147                 else if (isQuote[i]) isSkip = true;
148             }
149 
150             if (isSkip)
151             {
152                 node = node->next;
153                 continue;
154             }
155 
156             if (node->ch == '{' || node->ch == '(') push(bracketStack, node->ch);
157             else if (node->ch == '}')
158             {
159                 if (empty(bracketStack)) isMismatch = true;
160                 else if (top(bracketStack) != '{') isMismatch = true;
161                 else pop(bracketStack);
162             }
163             else if (node->ch == ')')
164             {
165                 if (empty(bracketStack)) isMismatch = true;
166                 else if (top(bracketStack) != '(') isMismatch = true;
167                 else pop(bracketStack);
168             }
169 
170             if (isMismatch)
171             {
172                 destory_stack(bracketStack);
173                 return false;
```

```
174             }
175
176             node = node->next;
177         }
178
179         curr = curr->next;
180     }
181
182     bool flag = empty(bracketStack);
183     destory_stack(bracketStack);
184
185     return flag;
186 }
```