基于二叉字典树的AC自动机

> Binary_Trie

- 技术说明：

  二叉字典树：通过把模式串编码为二进制字符串，将二进制编码插入字典树，进而达到节省空间的目的。

  - 结构定义：

    `typedef strucrt Binary_Trie {} Binary_Trie, *BTrie;`

    - `child[2]`：指向两个子节点。
    - `fail`：失败指针。
    - `char_flag`：标记独立成字母。
    - `word_flag`：标记独立成词。
  - 结构操作：
    - 获取新节点：通过`calloc()`和`malloc()`申请空间。
    - 插入函数：通过对每个字符进行编码，将二进制码自上而下插入到树中。
    - 失败指针和线索化建立函数：自上而下，逐层建立，顺便将利用空子节点指针建立线索化。
    - 匹配函数：正常遍历母串，自上而下，或沿着线索跳跃匹配。
  - 辅助工具：
    - 队列：

      通过顺序表，简单模拟队列。
    - 二进制转化函数：

      将字符的ASCII转化为二进制。
- 接口说明：
  - `BTrie BTrie_get_new_node()`：
    - 获取新节点。
    - 返回值：BTrie类型指针。
  - `int BTrie_insert(BTrie, char *)`：
    - 插入模式串到自动机中。
    - 参数：指向自动机根节点的指针和模式串。
    - 返回值：本次插入的新建立的结点的个数。
  - `void BTrie_build(BTrie , int)`：
    - 构建自动机。
    - 参数：指向自动机根节点的指针和结点个数。
  - `int BTrie_match_count(BTrie , const char *)`：
    - 参数：指向自动机根节点的指针和母串。
    - 返回值：母串中匹配成功的模式串的个数。
- 测试：
  - 由于二叉字典树是将字符串中每个字符转化为二进制码，插入到树中，因此，对于插入英文字符和汉字是一样的。
  - 采用计蒜客数据结构AC自动机习题：字符统计。

    - 测试结果：通过。
- 源代码：

```c
/*
* File Name:    Binary_Trie.cpp
* Author:       sunowsir
* Mail:         sunowsir@protonmail.com
* GitHub:       github.com/sunowsir
* Created Time: 2019年01月21日 星期一 20时21分14秒
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* BTrie 的结构定义 */
typedef struct Binary_Trie {
    /* char_flag: 标记独立成字, word_flag: 标记独立成词  */
    int char_flag, word_flag, count, flag[100];
    struct Binary_Trie **child, *fail;
} Binary_Trie, *BTrie;

/* Queue 的结构定义 */
typedef struct Queue{
    Binary_Trie **data;
    int head, tail, length;

} Queue;

/* 获取字符的编码 */
char *get_coding(char one_of_char_in_buffer) {
    int code = one_of_char_in_buffer, ret_len = 0;
    char *ret = (char *)calloc(sizeof(char), 20);
    while (code) {
        ret[ret_len++] = code % 2 + '0';
        code /= 2;
    }
    return ret;
}

/* Queue Start */

#define ERROR 0
#define OK 1

Queue *Queue_init(int length) {
    Queue *q = (Queue *)calloc(sizeof(Queue), 1);
    q->data = (Binary_Trie **)malloc(sizeof(Binary_Trie *) * length);
    q->length = length;
    q->head = 0;
    q->tail = -1;
    return q;
}
```

```c
int Queue_push(Queue *q, Binary_Trie *element) {
    if(q->tail + 1 > q->length) {
        return ERROR;

    }
    q->tail++;
    q->data[q->tail] = element;
    return OK;

}

Binary_Trie *Queue_front(Queue *q) {
    return q->data[q->head];

}
void Queue_pop(Queue *q) {
    q->head++;

}

int Queue_empty(Queue *q) {
    return q->head > q->tail;

}

void Queue_clear(Queue *q) {
    if (q == NULL) {
        return ;
    }
    free(q->data);
    free(q);
    return ;
}

#undef ERROR
#undef OK

/* Queue End  */

/* Binary Trie Start  */


/* BTrie 的结构操作:
*  1. BTrie_get_new_node()
*  2. BTrie_insert()
*  // 3. BTrie_search()
*  4. BTrie_build()
*  5. BTrie_match_count()
*  // 6. BTrie_clear()
*  */

#define SIZE 2
```

```c
#define BASE '0'

/* BTrie 的获取新节点的函数 */
BTrie BTrie_get_new_node() {
    BTrie bt = (BTrie)calloc(sizeof(Binary_Trie), 1);
    bt->child = (Binary_Trie **)calloc(sizeof(BTrie), 2);
    return bt;
}

/* BTrie 的插入函数 */
int BTrie_insert(BTrie root, char *buffer, int num) {
    BTrie p = root;
    int count = 0;
    for (int i = 0; buffer[i]; i++) {
        char *coding = get_coding(buffer[i]);
        // printf("%c : <%s>\n", buffer[i], coding);

        /* 根据获取到的每个字符的编码向下建树 */
        for (int j = 0; coding[j]; j++) {
            if (p->child[coding[j] - BASE] == NULL) {
                p->child[coding[j] - BASE] = BTrie_get_new_node();
                count++;
            }
            p = p->child[coding[j] - BASE];
        }
        /* 标记独立成字  */
        p->char_flag = 1;
        free(coding);
    }
    /* 标记独立成词 */
    p->word_flag = 1;
    p->count++;
    p->flag[++p->flag[0]] = num;
    return count;
}

// /* BTrie 的匹配函数 */
// int BTrie_search(BTrie root, char *str) {
//     BTrie p = root;
//     for (int i = 0; str[i]; i++) {
//         int now_char_same_flag = 1;
//         char *coding = get_coding(str[i]);
//
//         /* 根据获取到的每个字符的编码向下匹配 */
//         for (int j = 0; coding[j]; j++) {
//             if (p->child[coding[j]] == NULL) {
//                 return 0;
//             }
//             p = p->child[coding[j]];
//         }
//         if (!now_char_same_flag) {
//             return 0;
//         }
```

```
//         free(coding);
//     }
//     return p->word_flag;
// }

/* BTrie 的线索化和失败指针建立函数 */
void BTrie_build(BTrie root, int node_number) {
    Queue *q = Queue_init(node_number + 10);
    root->fail = NULL;
    Queue_push(q, root);
    while (!Queue_empty(q)) {
        BTrie now_node = Queue_front(q);
        Queue_pop(q);
        for (int i = 0; i < SIZE; i++) {
            if (now_node->child[i] == NULL) {
                if (now_node != root) {
                    now_node->child[i] = now_node->fail->child[i];
                }
                continue;
            }
            BTrie p = (now_node->fail ? now_node->fail->child[i] : root);
            if (p == NULL) {
                p = root;
            }
            now_node->child[i]->fail = p;
            Queue_push(q, now_node->child[i]);
        }
    }
    free(q);
    return ;
}

/* BTrie 的匹配函数 */
int BTrie_match_count(BTrie root, const char *str, int *num) {
    int ret = 0;
    BTrie p = root, q;
    while (str[0]) {
        char *coding = get_coding(str[0]);
        for (int i = 0; coding[i]; i++) {
            p = p->child[coding[i] - BASE];
            q = p;
        }
        while (q) {
            if (q->flag[0] > 0) {
                for (int i = 1; i <= q->flag[0]; i++) {
                    num[q->flag[i]] += q->word_flag;
                }
            }
            ret += q->count;
            // q->count = 0;
            q = q->fail;
        }
        if (p == NULL) {
```

```
            p = root;
        }
        free(coding);
        str++;
    }
    return ret;
}

// /* BTrie 的空间回收函数 */
// void BTrie_clear(BTrie root) {
// }

#undef SIZE

/* Binary Trie End */


int main() {
    int n, cnt = 0;
    BTrie root = BTrie_get_new_node();
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        char pattern[25] = {0};
        scanf("%s", pattern);
        cnt += BTrie_insert(root, pattern, i);
    }
    BTrie_build(root, cnt);
    char buffer[100005] = {0};
    scanf("%s", buffer);
    int ans[n] = {0};
    int total_count = BTrie_match_count(root, buffer, ans);

    for (int i = 0; i < n; i++) {
        printf("%d: %d\n", i, ans[i]);
    }

    return 0;
}
```