数据结构作业 第五周&第六周

霍斌 PB24111627

Hanoi

如下表所示:

Hanoi(3,a,b,c)的工作栈

步骤	栈的状态
1	Hanoi(3,a,b,c)
2	Hanoi(3,a,b,c), Hanoi(2,a,c,b)
3	Hanoi(3,a,b,c), Hanoi(2,a,c,b), Hanoi(1,a,b,c)
4	Hanoi(3,a,b,c), Hanoi(2,a,c,b)
5	Hanoi(3,a,b,c), Hanoi(2,a,c,b), Hanoi(1,c,a,b)
6	Hanoi(3,a,b,c), Hanoi(2,a,c,b)
7	Hanoi(3,a,b,c)
8	Hanoi(3,a,b,c), Hanoi(2,b,a,c)
9	Hanoi(3,a,b,c), Hanoi(2,b,a,c), Hanoi(1,b,c,a)
10	Hanoi(3,a,b,c), Hanoi(2,b,a,c)
11	Hanoi(3,a,b,c), Hanoi(2,b,a,c), Hanoi(1,a,b,c)
12	Hanoi(3,a,b,c), Hanoi(2,b,a,c)
13	Hanoi(3,a,b,c)
14	null

3.28

由题意实现了一个循环队列的头文件CircularQueue.h，包含初始化、出队、入队算法，并提供了q2.cpp以供测试:

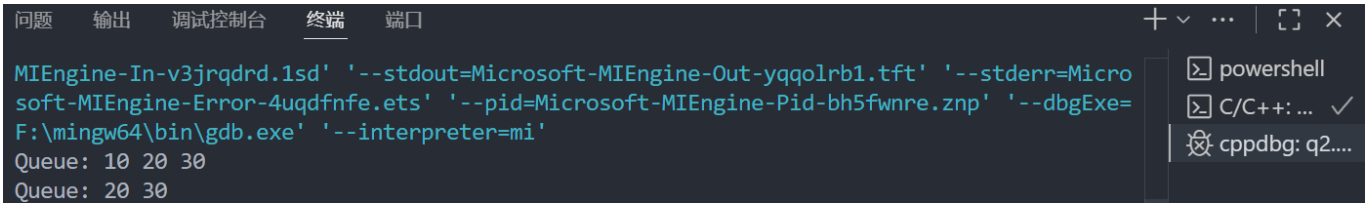
```
#ifndef CIRCULARQUEUE_H
#define CIRCULARQUEUE_H

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

typedef struct QueueNode {
```

```
    int data;
    struct QueueNode* next;
} QueueNode;

typedef struct CircularQueue {
    QueueNode* dummyHead;
    QueueNode* tail;
    int size;
} CircularQueue;

// 初始化循环队列
void initQueue(CircularQueue* queue) {
    queue->dummyHead = (QueueNode*)malloc(sizeof(QueueNode));
    // 让dummyHead->next 指向自身
    queue->dummyHead->next = queue->dummyHead;
    queue->tail = queue->dummyHead;
    queue->size = 0;
}

// 入队操作
void enqueue(CircularQueue* queue, int value) {
    // 创建新的节点
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    newNode->data = value;

    // 将新节点插入到 tail 之后
    newNode->next = queue->dummyHead;

    if (queue->size == 0) {
        queue->dummyHead->next = newNode;
        queue->tail = newNode;
    } else {
        queue->tail->next = newNode;
        queue->tail = newNode;
    }

    queue->size++; // 更新队列的大小
}

// 出队操作
void dequeue(CircularQueue* queue) {
    // 检测队列为空的情况
    if (queue->size == 0) {
        std::cout << "Queue is empty." << std::endl;
        return;
    }

    QueueNode* toDeleteNode = queue->dummyHead->next;
    queue->dummyHead->next = toDeleteNode->next;
    free(toDeleteNode);
    queue->size--;

    // 如果出队后队列为空，恢复到空队列初始状态
    if (queue->size == 0) {
```

```
        queue->tail = queue->dummyHead;
        queue->dummyHead->next = queue->dummyHead;
    }
}

// 销毁队列
void destroyQueue(CircularQueue* queue) {
    // 销毁所有节点
    while (queue->size > 0) {
        dequeue(queue);
    }
    // 释放dummyHead节点
    free(queue->dummyHead);
    queue->dummyHead = NULL;
    queue->tail = NULL;
    queue->size = 0;
}

void printQueue(CircularQueue* queue) {
    QueueNode* current = queue->dummyHead->next;
    std::cout << "Queue: ";
    for (int i = 0; i < queue->size; i++) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
#endif // CIRCULARQUEUE_H
```

```
#include "CircularQueue.h"

int main() {
    CircularQueue queue;
    initQueue(&queue);

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);
    printQueue(&queue);

    dequeue(&queue);
    printQueue(&queue);

    destroyQueue(&queue);

    return 0;
}
```

测试结果如下:

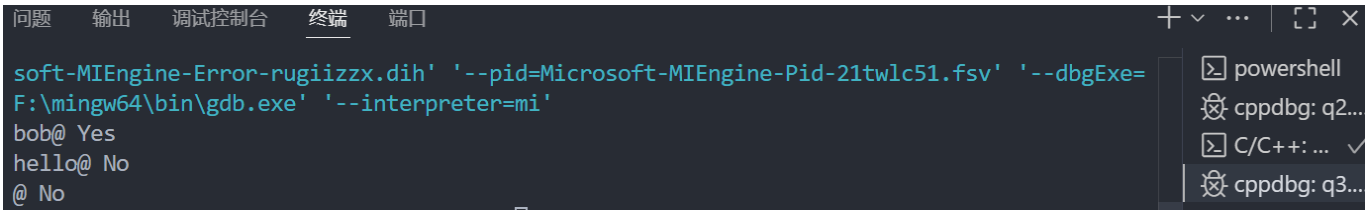


3.31

使用双指针法判断字符串是否回文，而且会排除空字符串课空指针的情况.

```
// 判断以@为结束符的字符串是否回文
#include <iostream>
#include <cstring>

bool isPalindrome(const char* str) {
    if(str == nullptr || strlen(str) == 1) {
        return false; // 空指针或者空字符串不是回文
    }

    // 左右双指针法
    int left = 0;
    int right = strlen(str) - 2; // -2是为了跳过结尾的@符号

    // 比较左右两端的字符，逐渐向中间靠拢
    while(left < right) {
        if(str[left] != str[right]) {
            return false; // 发现不匹配直接返回false
        }
        left++;
        right--;
    }

    // 未发现不匹配的字符，说明是回文
    return true;
}

int main() {
    const char* testStr1 = "bob@";
    const char* testStr2 = "hello@";
    const char* testStr3 = "@";

    // 测试回文函数
    std::cout << testStr1 << " " << (isPalindrome(testStr1) ? "Yes" : "No")
<< std::endl;
    std::cout << testStr2 << " " << (isPalindrome(testStr2) ? "Yes" : "No")
<< std::endl;
    std::cout << testStr3 << " " << (isPalindrome(testStr3) ? "Yes" : "No")
<< std::endl;

    return 0;
}
```

函数测试结果:



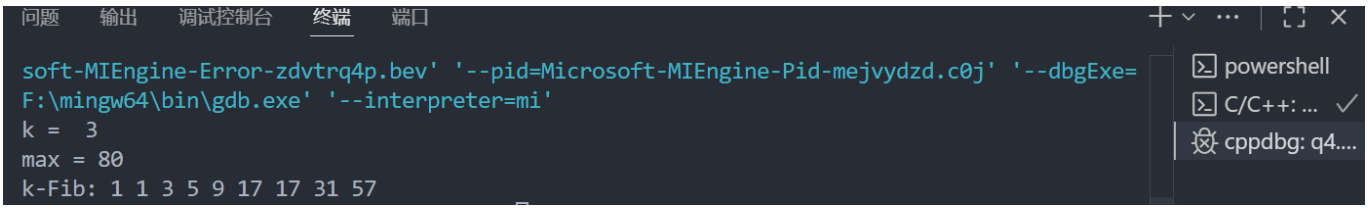
3.32

使用上文中的CircularQueue.h解决问题.

```
#include "CircularQueue.h"
#include <iostream>
using namespace std;

int main() {
    int k, max;
    cout << "k =  ";
    cin >> k;
    cout << "max = ";
    cin >> max;

    CircularQueue queue;
    initQueue(&queue);

    // 将前k项入队，均为1
    for (int i = 0; i < k; ++i) {
        enqueue(&queue, 1);
    }

    std::cout << "k-Fib: ";
    int fn = 1; // 队尾项
    while (true) {
        // 计算下一项
        int sum = 0;
        QueueNode* cur = queue.dummyHead->next;
        for (int i = 0; i < k; ++i) {
            sum += cur->data;
            cur = cur->next;
        }
        // sum为f_{n+1}
        if (fn <= max && sum > max) {
            break;
        }
        // 队列弹出最早的，加入新项
        dequeue(&queue);
        // dequeue之前输出队首的值，表示出整个数列
        cout << queue.dummyHead->next->data << ' ';
        enqueue(&queue, sum);
        fn = sum;
```

```
    }

    // 输出循环队列中的k项
    QueueNode* cur = queue.dummyHead->next;
    for (int i = 0; i < k; ++i) {
        cout << cur->data << " ";
        cur = cur->next;
    }
    cout << endl;

    destroyQueue(&queue);
    return 0;
}
```

测试结果:



4.18

```
// 计算串s中不同字符的个数和每个字符的出现次数
#include <iostream>
#include <cstring>
#include <cstdint>

int main() {
    char* str = "Hello world!";
    if(str == nullptr) {
        std::cout << "The input string is null." << std::endl;
        return -1;
    }

    unsigned int CHAR_NUM = 256; // 对于ASCII字符集一共有256个字符，创建一个顺序
表来存储每个字符的出现次数
    int charCount[CHAR_NUM] = {0}; // 初始化字符计数数组

    // 统计每个字符的出现次数
    for(int i = 0; i < strlen(str); ++i) {
        unsigned char ch = str[i];
        charCount[ch]++;
    }

    // 计算不同字符的个数
    int uniqueCharCount = 0;
    for(int i = 0; i < CHAR_NUM; ++i) {
        if(charCount[i] != 0) {
            uniqueCharCount++;
            std::cout << "Character '" << static_cast<char>(i) << "': " <<
```

```
    charCount[i] << "." << std::endl;
            }
        }

        std::cout << "Total unique characters: " << uniqueCharCount <<
    std::endl;

        return 0;
    }
```

用字符串Hello world!测试，结果如下:

4.22

在BlockListString.h中实现要求功能，并在q6.cpp主函数中验证:

```
#ifndef BLOCKLISTSTRING_H
#define BLOCKLISTSTRING_H

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

#define CHUNK_SIZE 8

typedef struct BStringNode {
    char data[CHUNK_SIZE];
    struct BStringNode* next;
} BStringNode;

typedef struct BString {
    BStringNode* dummyHead;
    BStringNode* tail;
    unsigned int size; // 字符串总长度
    unsigned int nodeCount; // 节点数量
} BString;


// 销毁字符串
void destroyBString(BString* str) {
    // 删除所有节点
    BStringNode* current = str->dummyHead->next;
    while (current != NULL) {
        BStringNode* toDelete = current;
        current = current->next;
```

```
            free(toDelete);
        }

        // 释放dummyHead节点
        free(str->dummyHead);
        str->dummyHead = NULL;
        str->tail = NULL;
        str->size = 0;
        str->nodeCount = 0;
    }

    // 向BString末尾添加一个字符
    void appendBString(BString* str, char ch) {
        unsigned int lastNodeSize = (str->size == 0) ? 0 : ((str->size - 1) %
    CHUNK_SIZE + 1);
        if (str->nodeCount == 0 || lastNodeSize == CHUNK_SIZE) {
            BStringNode* newNode = (BStringNode*)malloc(sizeof(BStringNode));
            newNode->data[0] = ch;
            newNode->next = NULL;
            str->tail->next = newNode;
            str->tail = newNode;
            str->nodeCount++;
        } else {
            str->tail->data[lastNodeSize] = ch;
        }
        str->size++;
    }

    // 初始化字符串
    void initBString(BString* str) {
        str->dummyHead = (BStringNode*)malloc(sizeof(BStringNode));
        str->dummyHead->next = NULL;
        str->tail = str->dummyHead;
        str->size = 0;
        str->nodeCount = 0;
    }

    void assignBString(BString* str, const char* cstr) {
        destroyBString(str);
        initBString(str);

        while (*cstr) {
            appendBString(str, *cstr);
            cstr++;
        }
    }

    // 打印字符串
    void printBString(BString* str) {
        std::cout << "BString: ";
        BStringNode* current = str->dummyHead->next;
        unsigned int printed = 0;

        while (current != NULL && printed < str->size) {
```

```
        unsigned int charsInNode = (current == str->tail) ?
            ((str->size - 1) % CHUNK_SIZE + 1) : CHUNK_SIZE;

        for (unsigned int i = 0; i < charsInNode; i++) {
            std::cout << current->data[i];
            printed++;
        }
        current = current->next;
    }
    std::cout << std::endl;
}

// 将串s插入到串t中字符ch之后，如果ch不存在则连接在t末尾
void insertAfterChar(BString* t, BString* s, char ch) {
    if (s == NULL || s->size == 0) return;

    // 查找字符ch在t中最后一次出现的位置
    BStringNode* targetNode = NULL;
    unsigned int targetPosInNode = 0;
    BStringNode* currentNode = t->dummyHead->next;
    unsigned int charIndex = 0;

    while (currentNode != NULL) {
        unsigned int charsInNode = (currentNode == t->tail) ?
            ((t->size - 1) % CHUNK_SIZE + 1) : CHUNK_SIZE;

        for (unsigned int i = 0; i < charsInNode; i++) {
            if (currentNode->data[i] == ch) {
                targetNode = currentNode;
                targetPosInNode = i;
            }
            charIndex++;
        }
        currentNode = currentNode->next;
    }

    // 如果没找到ch，将s连接在t末尾
    if (targetNode == NULL) {
        BStringNode* sCurrent = s->dummyHead->next;
        while (sCurrent != NULL) {
            unsigned int charsInNode = (sCurrent == s->tail) ? ((s->size -
1) % CHUNK_SIZE + 1) : CHUNK_SIZE;
            for (unsigned int i = 0; i < charsInNode; i++) {
                if (t->nodeCount > 0) {
                    unsigned int lastNodeSize = (t->size - 1) % CHUNK_SIZE
+ 1;
                    if (lastNodeSize < CHUNK_SIZE) {
                        t->tail->data[lastNodeSize] = sCurrent->data[i];
                        t->size++;
                        sCurrent = (i == charsInNode - 1) ? sCurrent->next
: sCurrent;
                        continue;
                    }
                }
```

```
                // 需要创建新节点
                BStringNode* newNode =
(BStringNode*)malloc(sizeof(BStringNode));
                newNode->data[0] = sCurrent->data[i];
                newNode->next = NULL;

                t->tail->next = newNode;
                t->tail = newNode;
                t->nodeCount++;
                t->size++;
            }
            sCurrent = sCurrent->next;
        }
        return;
    }

    // 开始插入，将t分为三部分 ch前(包括ch),s和ch之后部分

    // 创建新的BString
    BString* result = (BString*)malloc(sizeof(BString));
    initBString(result);

    // 前部分
    currentNode = t->dummyHead->next;
    unsigned int copiedChars = 0;
    bool reachedTarget = false;

    while (currentNode != NULL && !reachedTarget) {
        unsigned int charsInNode = (currentNode == t->tail) ?
            ((t->size - 1) % CHUNK_SIZE + 1) : CHUNK_SIZE;

        unsigned int endPos = (currentNode == targetNode) ? targetPosInNode
+ 1 : charsInNode;

        for (unsigned int i = 0; i < endPos; i++) {
            // 添加到result
            if (result->nodeCount > 0) {
                unsigned int lastNodeSize = (result->size - 1) % CHUNK_SIZE
+ 1;
                if (lastNodeSize < CHUNK_SIZE) {
                    result->tail->data[lastNodeSize] = currentNode-
>data[i];
                    result->size++;
                    continue;
                }
            }

            BStringNode* newNode =
(BStringNode*)malloc(sizeof(BStringNode));
            newNode->data[0] = currentNode->data[i];
            newNode->next = NULL;
            result->tail->next = newNode;
            result->tail = newNode;
```

```
            result->nodeCount++;
            result->size++;
        }

        if (currentNode == targetNode) {
            reachedTarget = true;
            copiedChars = targetPosInNode + 1;
        }
        currentNode = currentNode->next;
    }

    // 加入s
    BStringNode* sCurrent = s->dummyHead->next;
    while (sCurrent != NULL) {
        unsigned int charsInNode = (sCurrent == s->tail) ?
            ((s->size - 1) % CHUNK_SIZE + 1) : CHUNK_SIZE;

        for (unsigned int i = 0; i < charsInNode; i++) {
            if (result->nodeCount > 0) {
                unsigned int lastNodeSize = (result->size - 1) % CHUNK_SIZE
+ 1;
                if (lastNodeSize < CHUNK_SIZE) {
                    result->tail->data[lastNodeSize] = sCurrent->data[i];
                    result->size++;
                    continue;
                }
            }

            BStringNode* newNode =
(BStringNode*)malloc(sizeof(BStringNode));
            newNode->data[0] = sCurrent->data[i];
            newNode->next = NULL;
            result->tail->next = newNode;
            result->tail = newNode;
            result->nodeCount++;
            result->size++;
        }
        sCurrent = sCurrent->next;
    }

    // 加入后半部分
    currentNode = targetNode;
    unsigned int startPos = targetPosInNode + 1;

    while (currentNode != NULL) {
        unsigned int charsInNode = (currentNode == t->tail) ?
            ((t->size - 1) % CHUNK_SIZE + 1) : CHUNK_SIZE;

        for (unsigned int i = startPos; i < charsInNode; i++) {
            if (result->nodeCount > 0) {
                unsigned int lastNodeSize = (result->size - 1) % CHUNK_SIZE
+ 1;
                if (lastNodeSize < CHUNK_SIZE) {
                    result->tail->data[lastNodeSize] = currentNode-
```

```
>data[i];
                    result->size++;
                    continue;
                }
            }

            BStringNode* newNode =
(BStringNode*)malloc(sizeof(BStringNode));
            newNode->data[0] = currentNode->data[i];
            newNode->next = NULL;
            result->tail->next = newNode;
            result->tail = newNode;
            result->nodeCount++;
            result->size++;
        }

        currentNode = currentNode->next;
        startPos = 0; // 后续节点从0开始
    }

    // 消去t的内容
    BStringNode* toDelete = t->dummyHead->next;
    while (toDelete != NULL) {
        BStringNode* next = toDelete->next;
        free(toDelete);
        toDelete = next;
    }



    // 让t指向result内容
    t->dummyHead->next = result->dummyHead->next;
    t->tail = result->tail;
    t->size = result->size;
    t->nodeCount = result->nodeCount;
}

#endif // BLOCKLISTSTRING_H
```

以下是测试程序的代码:创建两个字符串然后再使用insertAfterChar函数进行测试:

```
#include "BlockListString.h"

int main() {
    BString s;
    BString t;
    initBString(&s);
    initBString(&t);

    // Example usage: assign values, concatenate, and print
    assignBString(&s, "Hello, ");
    assignBString(&t, "world!");
```

```
    printBString(&s);
    printBString(&t);

    insertAfterChar(&t, &s, 'l');
    printBString(&t);
    // insertAfterChar();
    return 0;
}
```

发现结果符合预期，如下图所示: