



# **数据结构实验报告**

题目:基于红黑树的哈希表实现

学 院 国际教育学院

专 业 计算机科学与技术国际

年级班别 一班

学 号 3124009862

学生姓名 杨恒熠

指导教师 李小妹

编 号

成 绩 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2025年 11月

**报告：**

**报告内容：** □详细　 □完整　 □基本完整 □不完整

**设计方案：** □非常合理　 □合理　 □基本合理 □较差

**算**法实**现：** □全部实现　 □基本实现　 □部分实现 □实现较差

**测试样例：** □完备　 □比较完备　 □基本完备 □不完备

**文档格式：** □规范　 □比较规范　 □基本规范 □不规范

**答辩：**

□理解题目透彻，问题回答流利

□理解题目较透彻，回答问题基本正确

□部分理解题目，部分问题回答正确

□未能完全理解题目，答辩情况较差

**总评成绩：**

□优　　　□良　　　□中　　　□及格　　　□不及格

## 一、实验目的

1. 掌握哈希表的基本概念和实现原理
2. 理解红黑树的性质和操作方法
3. 学会将红黑树作为哈希冲突解决策略应用于哈希表
4. 实现哈希表的基本操作：插入、查找、删除等
5. 分析基于红黑树的哈希表的时间复杂度和性能特点

## 二、实验环境

CPU：Intel(R) Core(TM) i9-14900HX 2.20 GHz

Memory：31.6 GB

OS：Windows 11 专业版

IDE:CLion 2025.2.1

编译器:GCC 8.1.0

## 三、实验内容

设计并实现一个基于红黑树的哈希表，要求实现以下功能： 1. 初始化哈希表 2. 销毁哈希表 3. 插入键值对 4. 查找指定键的值 5. 删除指定键的元素 6. 获取哈希表中元素个数 7. 显示哈希表内容

## 四、实验原理

### 3.1 哈希表基本概念

哈希表(Hash Table)是一种根据关键码值(Key value)而直接进行访问的数据结构。它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

### 3.2 红黑树基本概念

红黑树是一种自平衡的二叉搜索树，具有以下性质： 1. 每个节点是红色或黑色 2. 根节点是黑色 3. 每个叶节点(NIL节点)是黑色 4. 如果一个节点是红色，则它的两个子节点都是黑色 5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点

### 3.3 哈希冲突解决

当不同的键通过哈希函数映射到同一个位置时，就会发生哈希冲突。本实验采用链地址法解决冲突，但不同于普通的链表，而是使用红黑树来存储同一位置的多个元素，这样可以保证较好的查找性能。

## 五、数据结构设计

### 4.1 红黑树节点定义

typedef struct RBNode {  
 int key;  
 int value;  
 int color; /\* RED or BLACK \*/  
 struct RBNode \*left, \*right, \*parent;  
} RBNode;

### 4.2 哈希表结构定义

typedef struct HashTable {  
 RBNode \*\*buckets; /\* 指向根节点指针数组，根节点使用全局 NIL 作为空 \*/  
 int size; /\* 桶数量 \*/  
 int count; /\* 键值对总数 \*/  
} HashTable;

## 五、算法设计与实现

### 5.1 哈希函数

采用简单的除留余数法：

static int hash\_func(HashTable \*ht, int key) {  
 int idx = key % ht->size;  
 if (idx < 0) idx += ht->size;  
 return idx;  
}

### 5.2 红黑树操作

#### 插入操作

1. 按照二叉搜索树的方式插入新节点
2. 将新节点着色为红色
3. 通过旋转和重新着色来维护红黑树性质

#### 删除操作

1. 按照二叉搜索树的方式删除节点
2. 如果删除的是黑色节点，需要通过旋转和重新着色来维护红黑树性质

#### 查找操作

按照二叉搜索树的查找方式进行查找。

### 5.3 哈希表操作实现

#### 初始化哈希表

void InitHashTable(HashTable \*ht, int size) {  
 if (size <= 0) size = 8;  
 init\_nil();  
 ht->size = size;  
 ht->count = 0;  
 ht->buckets = (RBNode \*\*)malloc(sizeof(RBNode \*) \* size);  
 for (int i = 0; i < size; i++) ht->buckets[i] = NIL;  
}

#### 插入操作

Status InsertHash(HashTable \*ht, int key, int value) {  
 if (!ht || !ht->buckets) return ERROR;  
 int idx = hash\_func(ht, key);  
 int inserted\_new = 0;  
 RBNode \*root = ht->buckets[idx];  
 RBNode \*ret = rb\_insert\_node(&root, key, value, &inserted\_new);  
 if (!ret) return ERROR;  
 ht->buckets[idx] = root;  
 if (inserted\_new) ht->count++;  
 return OK;  
}

#### 查找操作

Status SearchHash(HashTable \*ht, int key, int \*value) {  
 if (!ht || !ht->buckets) return ERROR;  
 int idx = hash\_func(ht, key);  
 RBNode \*node = rb\_search(ht->buckets[idx], key);  
 if (!node) return ERROR;  
 if (value) \*value = node->value;  
 return OK;  
}

#### 删除操作

Status DeleteHash(HashTable \*ht, int key) {  
 if (!ht || !ht->buckets) return ERROR;  
 int idx = hash\_func(ht, key);  
 int ok = rb\_delete\_node(&ht->buckets[idx], key);  
 if (ok) ht->count--;  
 return ok ? OK : ERROR;  
}

## 六、测试与分析

### 6.1 测试代码

#include <stdio.h>  
#include "RBTree\_Hash.h"  
  
int main(void) {  
 HashTable ht;  
 InitHashTable(&ht, 8);  
  
 InsertHash(&ht, 10, 100);  
 InsertHash(&ht, 18, 180);  
 InsertHash(&ht, 26, 260);  
 InsertHash(&ht, 2, 20);  
 InsertHash(&ht, -6, -60);  
 InsertHash(&ht, 3, 30);  
  
 printf("Initial table:\n");  
 DisplayHash(&ht);  
  
 int v;  
 if (SearchHash(&ht, 18, &v)) printf("Found 18 -> %d\n", v);  
 else printf("18 not found\n");  
  
 if (SearchHash(&ht, 3, &v)) printf("Found 3 -> %d\n", v);  
 else printf("3 not found\n");  
  
 DeleteHash(&ht, 18);  
 printf("After deleting 18:\n");  
 DisplayHash(&ht);  
  
 printf("Count = %d\n", GetCount(&ht));  
  
 DestroyHashTable(&ht);  
 return 0;  
}

### 6.2 测试结果

Initial table:  
Bucket 0: (empty)  
Bucket 1: (empty)  
Bucket 2: (-6,-60) (2,20) (10,100) (18,180) (26,260)   
Bucket 3: (3,30)   
Bucket 4: (empty)  
Bucket 5: (empty)  
Bucket 6: (empty)  
Bucket 7: (empty)  
Found 18 -> 180  
Found 3 -> 30  
After deleting 18:  
Bucket 0: (empty)  
Bucket 1: (empty)  
Bucket 2: (-6,-60) (2,20) (10,100) (26,260)   
Bucket 3: (3,30)   
Bucket 4: (empty)  
Bucket 5: (empty)  
Bucket 6: (empty)  
Bucket 7: (empty)  
Count = 5

### 6.3 结果分析

从测试结果可以看出： 1. 哈希函数将键值正确地分配到了不同的桶中 2. 在桶2中，由于10、18、26、-6、2的哈希值相同，它们被存储在同一棵红黑树中，并按照键值大小有序排列 3. 查找操作能够正确找到存在的键值对 4. 删除操作能够正确删除指定的键，并维护红黑树的结构 5. 元素计数功能正确反映了哈希表中元素的数量

## 七、时间复杂度分析

| 操作 | 平均时间复杂度 | 最坏时间复杂度 |
| --- | --- | --- |
| 插入 | O(1) | O(log n) |
| 查找 | O(1) | O(log n) |
| 删除 | O(1) | O(log n) |

其中n为哈希表中元素的总数。在理想情况下，哈希函数能够均匀分布元素，每个桶中的红黑树节点数较少，操作时间接近O(1)。在最坏情况下，所有元素都映射到同一个桶中，此时操作时间取决于红黑树的高度，为O(log n)。

## 八、实验总结

通过本次实验，我深入理解了哈希表和红黑树的数据结构特点以及它们的实现方式。使用红黑树作为哈希冲突的解决方案相比链表具有更好的性能保证，特别是在数据量较大时能够提供更稳定的查找、插入和删除时间复杂度。

在实现过程中，我遇到了以下问题和挑战：

1. 红黑树的旋转和着色操作较为复杂，需要仔细处理各种情况
2. 哈希函数的设计对性能影响较大，需要根据实际数据分布选择合适的函数
3. 内存管理需要特别注意，避免内存泄漏

通过解决这些问题，我不仅提高了编程能力，也加深了对数据结构和算法的理解。这种将不同数据结构组合使用的方法在实际开发中具有重要意义，能够根据具体需求选择最适合的数据结构组合来解决问题。