山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 级数据结构--数据/智能 实验 8 散列表		
实验学时：2	实验日期： 2024/11/06	

实验目的：

1. 掌握散列表结构的定义与实现；
2. 掌握散列表结构的使用。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 级数据结构--数据/智能 实验 8 散列表 A 线性开型寻址 和 B 链表散列。

其中需要分别实现使用数组描述的（即线性开型寻址）的散列表和使用链表进行存储和寻址的散列表。我们需要使用 insert、erase、find 这三个核心的方法。

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

A 题，本题使用的数据结构为线性开型寻址的散列表，这种散列表将所有的数据存储在一个线性的固定大小的数组上，其范围在确认除数的时候就已经确定了。我们分别来阐述这三种核心方法的实现方式（其实还应该有个修改的）。在这之前我们声明一下这个类中的数据成员，我们创建一个数据数组，再创将一个与这个数组一样长的 used 数组用于标记哪些位置已经存入了数据。首先是插入，我们通过键对除数取模可以获得一个索引，我们首先检查这个 used 中这个索引位置的真值来确认是否已经在这个数字原本该在的地方存储上了数据，如果没有则直接插入，如果有就向下遍历，直到遇到一个空位，将数据插入，插入后将 used 数组对应位置的真值进行更改向下遍历的方式为从(k + 1) % devisor 位置开始：index = (index + 1) % devisor;如果 index 在遍历过程中与 k % devisor 相等了，则说明这个散列表满了；再是 find 方法，这个方法的实现为，先对键按除数取模获得最初的起始位置，再检查这个位置的值是否为要找的值，如果不是就继续向下遍历，遍历方式与上文方式一致，如果遍历到终点，则说明不存在，或者遍历到空位，也说明不存在，如果找到，则可确认该数据在散列表中的下标。再是 erase 方法，这个方法的基础功能，即找到需要删除的数据并删除与 find 方法相同，而再删除过程完成后，我们还需要对散列表的序列进行更改，来让这个表维持散列的结构。这个过程实际就是，从当前被删除的数据的下一位开始遍历，按照以下原则进行处理：

①如果遇到空位，则停止 erase 过程

②如果遇到位置正确（即没有被挤走）（即 key % devisor == index）的数据，则跳过

③如果遇到值的正确位置在被删除数据的后方，且这个数据存在于被删除数据的后方，则跳过

④如果遇到的值的正确位置在被删除数据的前方，这个值的位置也在被删除数据的前方，则跳过

⑤如果遇到的值应该在的位置在被删除数据的后方，且被删除数据的位置在这个值的位置的后方，则跳过

⑥其余情况，则从该值的位置向前遍历，找到空位后移动过去，再将这个数据视为被删除的数据重复上述过程

经过以上过程，我们就成功完成了线性开型寻址的散列表中数据的删除操作，当然这也是这

个数据结构中最复杂的操作了。实际完成这个题所需要的操作过程的实现就是分别调用插入、查找、删除操作了，而且这个题最简单的地方在于他给的操作数可以保证散列表中的数据个数永远不会超过这个散列表的容量，因此对于溢出的处理可以简化很多。
B 题，本题让使用链表来描述散列表，实际上实现也是很简单的，我们再来详细说明 insert、erase、find 操作的实现方式，我们只需要让这个散列表存储一个存放链表指针的数组的指针即可。首先是 insert 方法，我们先将键对除数取模，可以得到一个他应该在的位置的索引，再按照这个索引从链表数组里获得一个链表的指针，再由该指针去遍历对应的链表，如果存在这个值就插入失败，如果不存在就执行这个链表的 push_back 方法；find 方法，和 insert 方法的操作类似，只需要先获取这个索引的位置，再到对应的链表中去寻找需要找的值即可；erase 方法确认要删除的这个值的位置的方法与 find 方法一致，确认之后我们再执行该链表的 erase 方法即可，操作数操作与 A 题类似。
3. 测试结果（测试输入，测试输出）
A 题测试输入：
7 12
1 21
0 1
0 13
0 5
0 23
0 26
0 33
1 33
1 33
1 13
1 5
1 1
输出：
-1
1
6
5
2
0
3
3
3
6
5
1
测试输入二：
20 30
0 84
0 15
0 54
2 15
2 84

1 54
2 54
0 89
1 89
0 13
0 48
2 89
0 60
0 24
1 13
0 6
1 24
0 31
2 60
2 48
0 49
0 9
1 6
1 13
0 33
2 49
0 60
1 6
2 9
1 60
输出：
4
15
14
0
0
14
0
9
9
13
8
0
0
4
13
6
4
11
0
0

9
10
6
13
14
1
0
6
0
0
B 题测试输入：
7 12
1 21
0 1
0 13
0 5
0 23
0 26
0 33
1 33
1 33
1 13
1 5
1 1
输出：
Not Found
3
3
1
3
1
测试输入二：
7 15
2 10
0 10
0 10
2 10
1 10
0 10
1 10
0 17
0 2
0 16
0 11
2 2
2 10

1 11
1 17
输出：
Delete Failed
Existed
0
Not Found
1
1
1
1
1
4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）
从测试结果来看，我们的算法成功解决了这个问题。存在的问题我们分别分析 A 题和 B 题，A 题，这个题的主要问题在于，我们并不考虑在散列表已满的情况下再次执行插入操作时的处理方式，无论是输出错误信息还是抛出错误，因为题目数据合法的关系，我们在编写代码的时候并没有考虑这些。实际编写的时候在插入部分检测一个检测是否已满的代码并单独处理散列表已经满了的情况即可。还有一个问题，就是如果我们输入的数据十分杂乱，或者十分集中，数据无法离散的存储在这个表里，我们实际的查询耗时是远大于 O（1）的，但是也是始终小于 O（n）的，这是这个线性开型寻址的散列表的数据结构自身的问题，在此只做指出，暂时不知道有没有什么优秀的方案来改进原本的这种数据结构。关于 B 题，链表存储，这种存储方式因为是动态大小，有效的避免了表会存满的问题。但同时，这个方式与数组描述有一个共有的问题就是查询速率的问题。在此给出一个优化方案，即在插入数据时使用插入排序的方式，让每个链表序列始终保持有序，并优化查询算法，让查询以二分的方式完成，依次来提升查询的速度。
5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）
本附录分为两部分，第一部分为 A 题代码，第二部分为 B 题代码
一、A 题代码

```cpp
/*2024 级数据结构--数据智能 实验 8 散列表 A 使用线性开型寻址实现.cpp*/
#include <iostream>

using namespace std;

template<class T>
class hashTable
{
private:
    T divisor;
    int size;
    T* table;
    bool* used;
public:
    hashTable(int size, T& divisor);
    void find(T& key);
    void insert(T& key);
    void erase(T& key);
```

```
19.       void display();
20.   };
21.
22.   template<class T>
23.   void hashTable<T>::display()
24.   {
25.       for (int i = 0; i < size; i++)
26.       {
27.           cout << table[i] << " ";
28.       }
29.       cout << endl;
30.   }
31.
32.   template<class T>
33.   hashTable<T>::hashTable(int size, T& divisor)
34.   {
35.       this->size = size;
36.       this->divisor = divisor;
37.       table = new T[size];
38.       for (int i = 0; i < size; i++)
39.       {
40.           table[i] = -1;
41.       }
42.       used = new bool[size];
43.       for (int i = 0; i < size; i++)
44.       {
45.           used[i] = false;
46.       }
47.   }
48.
49.   template<class T>
50.   void hashTable<T>::insert(T& key)
51.   {
52.       int index = key % size;
53.       while (index < size)
54.       {
55.           if (used[index] == false)
56.           {
57.               table[index] = key;
58.               used[index] = true;
59.               cout << index << endl;
60.               return;
61.           }
62.           if (table[index] == key)
63.           {
64.               cout << "Existed" << endl;
```

```
65.                 return;
66.             }
67.             index++;
68.             index = index % divisor;
69.         }
70.     }
71.
72.     template<class T>
73.     void hashTable<T>::find(T& key)
74.     {
75.         int index = key % size;
76.         while (index < size)
77.         {
78.             if (used[index] == false)
79.             {
80.                 cout << -1 << endl;
81.                 return;
82.             }
83.             if (table[index] == key)
84.             {
85.                 cout << index << endl;
86.                 return;
87.             }
88.             index++;
89.             index = index % divisor;
90.         }
91.     }
92.
93.
94.     template<class T>
95.     void hashTable<T>::erase(T& key) {
96.         int s = key % divisor;
97.         int j = s;
98.         int b = 0;
99.         do {
100.            if (table[j] == key) {
101.                table[j] = -1;
102.                used[j] = false;
103.                b = 1;
104.                break;
105.            }
106.            else {
107.                j = (j + 1) % divisor;
108.            }
109.        } while (j != s && used[j] == true);
110.        if (b == 0) {
```

```
            cout << "Not Found" << endl;
            return;
        }
        s = j;
        j = (j + 1) % divisor;
        int k = s;
        int sum = 0;
        for (int i = j; i != (j + divisor - 1) % divisor; i = (i + 1) %
    divisor)
        {
            if (used[i] == false)
            {
                break;
            }
            int x2 = table[i];
            if (i == (x2 % divisor) || ((x2 % divisor) > s) && i > (x2
   % divisor))
            {
                continue;
            }
            else {
                if (i < s && i>(x2 % divisor))
                {
                    continue;
                }
                else if ((x2 % divisor) > s && s > i)
                {
                    continue;
                }
                else
                {
                    int a = i;
                    sum++;
                    while (used[(a + divisor - 1) % divisor] == true)
                    {
                        a = (a + divisor - 1) % divisor;
                    }
                    table[(a + divisor - 1) % divisor] = x2;
                    used[(a + divisor - 1) % divisor] = true;
                    used[i] = false;
                    s = i;
                }
            }
        }
        cout << sum << endl;
    }
```

```
155.
156.
157.
158.
159. // template<class T>
160. // void hashTable<T>::erase(T& key)
161. // {
162. //     int index = key % size;
163. //     while (index < size)
164. //     {
165. //         if (index == 0 && table[index] == 1)
166. //         {
167. //             cout << "Not Found" << endl;
168. //             return;
169. //         }
170. //         if (table[index] == key && index == 0)
171. //         {
172. //             table[index] = 1;
173. //             int i = index;
174. //             int sum = 0;
175. //             int tem_index = index;
176. //             while (i < size)
177. //             {
178. //                 i = (i + 1) % divisor;
179. //                 if (i == key % divisor)
180. //                 {
181. //                     break;
182. //                 }
183. //                 if (table[i] == divisor && i != 0)
184. //                 {
185. //                     continue;
186. //                 }
187. //                 if (table[i] == 1 && i == 0)
188. //                 {
189. //                     continue;
190. //                 }
191. //                 if (table[i] != divisor && i != 0)
192. //                 {
193. //                     if (table[i] % divisor != i)
194. //                     {
195. //                         sum++;
196. //                         table[tem_index] = table[i];
197. //                         if (i == 0)
198. //                         {
199. //                             table[i] = 1;
200. //                         }
```

```
201. //                          if (i != 0)
202. //                          {
203. //                              table[i] = divisor;
204. //                          }
205. //                          tem_index = i;
206. //                      }
207. //                  }
208. //              }
209. //              cout << sum << endl;
210. //              return;
211. //          }
212. //          if (table[index] == key && index != 0)
213. //          {
214. //              table[index] = divisor;
215. //              int i = index;
216. //              int sum = 0;
217. //              int tem_index = index;
218. //              while (i < size)
219. //              {
220. //                  i = (i + 1) % divisor;
221. //                  if (i == key % divisor)
222. //                  {
223. //                      break;
224. //                  }
225. //                  if (table[i] == divisor && i != 0)
226. //                  {
227. //                      break;
228. //                  }
229. //                  if (table[i] == 1 && i == 0)
230. //                  {
231. //                      break;
232. //                  }
233. //                  if (table[i] != divisor && i != 0)
234. //                  {
235. //                      if (table[i] % divisor != i)
236. //                      {
237. //                          sum++;
238. //                          table[tem_index] = table[i];
239. //                          if (i == 0)
240. //                          {
241. //                              table[i] = 1;
242. //                          }
243. //                          if (i != 0)
244. //                          {
245. //                              table[i] = divisor;
246. //                          }
```

```
247. //                         tem_index = i;
248. //                     }
249. //                 }
250. //             }
251. //             cout << sum << endl;
252. //             return;
253. //         }
254. //         if (table[index] == divisor && index != 0)
255. //         {
256. //             cout << "Not Found" << endl;
257. //             return;
258. //         }
259. //         index++;
260. //         index = index % divisor;
261. //     }
262. //     cout << "Not Found" << endl;
263. // }
264. class Solution
265. {
266. public:
267.     void solve();
268. };
269. 
270. void Solution::solve()
271. {
272.     int D, m;
273.     cin >> D >> m;
274.     hashTable<int> table(D, D);
275.     for (int i = 0; i < m; i++)
276.     {
277.         int op, x;
278.         cin >> op >> x;
279.         if (op == 0)
280.         {
281.             table.insert(x);
282.             // table.display();
283.         }
284.         else if (op == 1)
285.         {
286.             table.find(x);
287.             // table.display();
288.         }
289.         else if (op == 2)
290.         {
291.             table.erase(x);
292.             // table.display();
```

```
            }
        }
    }

    int main()
    {
        Solution solution;
        solution.solve();
        return 0;
    }
```

二、B 题代码

```
    /*2024 级数据结构--数据智能 实验 8 散列表 B 链表散列.cpp*/
    #include<iostream>

    using namespace std;


    template<class T>
    class list_node
    {
    private:
        T data;

    public:
        list_node<T>* front;
        list_node<T>* next;
        bool operator==(const list_node<T>& list_node) const { return this->data == list_node.data; };
        T& getData() { return data; };
        list_node<T>* getFront() { return front; };
        list_node<T>* getNext() { return next; };
        list_node(){};
        list_node(const T& data, list_node<T>* front = nullptr, list_node<T>* next = nullptr);
        list_node(list_node<T>* const list_node) { this->data = list_node->data;this->front = list_node->front;this->next = list_node->next; };
    };

    template<class T>
    list_node<T>::list_node(const T& data,list_node<T>* front,list_node<T>* next)
    :data(data),front(front),next(next)
    {

    }
```

```
31.
32.  template<class T>
33.  class chainlist
34.  {
35.  private:
36.      list_node<T>* head;
37.      list_node<T>* back;
38.      unsigned int size;
39.  public:
40.      void push_back(list_node<T>* operated_list_node);
41.      void push_back(const T& data);
42.      void push_front(list_node<T>* operated_list_node);
43.      void push_front(const T& data);
44.      void pop_back();
45.      void pop_front();
46.      void insert(T data, int index);
47.      void insert(list_node<T>* operated_list_node, int index);
48.      void erase(const unsigned int index);
49.      unsigned int getSize() { return size; };
50.      unsigned int find_first(list_node<T>* operated_list_node);
51.      unsigned int find_first(const T& data);
52.      void inverse();
53.      chainlist();
54.      chainlist(const chainlist<T>& chainlist);
55.      void print();
56.      list_node<T>* getHead() { return head; };
57.      list_node<T>* getBack() { return back; };
58.
59.  };
60.
61.  template<class T>
62.  void chainlist<T>::push_back(list_node<T>* operated_list_node)
63.  {
64.      list_node<T>* new_list_node = new list_node<T>(operated_list_no
   de);
65.
66.
67.      if (size == 0)
68.      {
69.          this->head = new_list_node;
70.          this->back = new_list_node;
71.          size++;
72.          return;
73.      }
74.
75.      this->back->next = new_list_node;
```

```
76.       new_list_node->front = this->back;
77.       new_list_node->next = nullptr;
78.       this->back = new_list_node;
79.
80.       size++;
81.   }
82.
83.   template<class T>
84.   void chainlist<T>::push_back(const T& data)
85.   {
86.       T new_data = data;
87.       list_node<T>* new_list_node = new list_node<T>(new_data);
88.       this->push_back(new_list_node);
89.       delete new_list_node;
90.   }
91.
92.   template<class T>
93.   void chainlist<T>::push_front(list_node<T>* operated_list_node)
94.   {
95.       list_node<T>* new_list_node = new list_node<T>(operated_list_no
   de);
96.
97.       if (size == 0)
98.       {
99.           this->back = new_list_node;
100.          this->head = new_list_node;
101.          size++;
102.          return;
103.      }
104.      new_list_node->next = this->head;
105.      new_list_node->front = nullptr;
106.      this->head->front = new_list_node;
107.      this->head = new_list_node;
108.
109.      size++;
110.  }
111.
112.  template<class T>
113.  void chainlist<T>::push_front(const T& data)
114.  {
115.      T new_data = data;
116.      list_node<T>* new_list_node = new list_node<T>(new_data);
117.      this->push_front(new_list_node);
118.      delete new_list_node;
119.  }
120.
```

```
121. template<class T>
122. void chainlist<T>::pop_back()
123. {
124.     if (this->back == nullptr) return;
125.     list_node<T>* temp = this->back;
126.     this->back = this->back->front;
127.     delete temp;
128.     if (size == 1)
129.     {
130.         this->head = nullptr;
131.         this->back = nullptr;
132.     }
133.
134.     size--;
135. }
136.
137. template<class T>
138. void chainlist<T>::pop_front()
139. {
140.     if (this->head == nullptr) return;
141.     list_node<T>* temp = this->head;
142.     this->head = this->head->next;
143.     delete temp;
144.     if (size == 1)
145.     {
146.         this->head = nullptr;
147.         this->back = nullptr;
148.     }
149.     size--;
150. }
151.
152.
153. template<class T>
154. void chainlist<T>::insert(list_node<T>* operated_list_node, int ind
   ex)
155. {
156.     if (index == 0)
157.     {
158.         push_front(operated_list_node);
159.         return;
160.     }
161.     if (index == size)
162.     {
163.         push_back(operated_list_node);
164.         return;
165.     }
```

```
166.
167.     list_node<T>* temp = this->head;
168.     for (int i = 0; i < index - 1; i++)
169.     {
170.         temp = temp->next;
171.     }
172.     list_node<T>* new_list_node = new list_node<T>(operated_list_no
    de);
173.     new_list_node->next = temp->next;
174.     temp->next->front = new_list_node;
175.     temp->next = new_list_node;
176.     new_list_node->front = temp;
177.     size++;
178. }
179.
180. template<class T>
181. void chainlist<T>::insert(T data, int index)
182. {
183.     list_node<T>* new_list_node = new list_node<T>(data);
184.     this->insert(new_list_node, index);
185.     delete new_list_node;
186.
187. }
188.
189. template<class T>
190. void chainlist<T>::erase(const unsigned int index)
191. {
192.     if (index == 0)
193.     {
194.         pop_front();
195.         return;
196.     }
197.     if (index == size - 1)
198.     {
199.         pop_back();
200.         return;
201.     }
202.
203.     list_node<T>* temp = this->head;
204.     for (int i = 0; i < index - 1; i++)
205.     {
206.         temp = temp->next;
207.     }
208.     list_node<T>* temp2 = temp->next;
209.     temp->next = temp2->next;
210.     temp->next->front = temp;
```

```
211.     delete temp2;
212.     size--;
213. }
214. 
215. template<class T>
216. unsigned int chainlist<T>::find_first(list_node<T>* operated_list_n
   ode)
217. {
218.     list_node<T>* temp = this->head;
219.     unsigned int index = 0;
220.     while (temp != nullptr)
221.     {
222.         if ((*temp) == (*operated_list_node)) return index;
223.         temp = temp->next;
224.         index++;
225.     }
226.     return -1;
227. }
228. 
229. template<class T>
230. unsigned int chainlist<T>::find_first(const T& data)
231. {
232.     list_node<T>* temp = this->head;
233.     unsigned int index = 0;
234.     while (temp != nullptr)
235.     {
236.         if (temp->getData() == data) return index;
237.         temp = temp->next;
238.         index++;
239.     }
240.     return -1;
241. }
242. 
243. template<class T>
244. void chainlist<T>::inverse()
245. {
246.     list_node<T>* temp1 = this->head;
247.     list_node<T>* temp2 = nullptr;
248.     list_node<T>* temp3 = nullptr;
249.     for (int i = 0; i < size; i++)
250.     {
251.         temp2 = temp1->next;
252.         temp3 = temp1->front;
253.         temp1->front = temp1->next;
254.         temp1->next = temp3;
255.         temp1 = temp2;
```

```
256.     }
257.     temp1 = this->head;
258.     this->head = this->back;
259.     this->back = temp1;
260. }
261.
262. template<class T>
263. chainlist<T>::chainlist()
264. {
265.     this->head = nullptr;
266.     this->back = nullptr;
267.     this->size = 0;
268. }
269.
270. template<class T>
271. chainlist<T>::chainlist(const chainlist<T>& chainlist)
272. {
273.     this->head = nullptr;
274.     this->back = nullptr;
275.     this->size = 0;
276.     list_node<T>* temp = chainlist.head;
277.     while (temp != nullptr)
278.     {
279.         push_back(temp);
280.         temp = temp->next;
281.     }
282. }
283.
284. template<class T>
285. void chainlist<T>::print()
286. {
287.     list_node<T>* temp = this->head;
288.     while (temp != nullptr)
289.     {
290.         cout << temp->getData() << " ";
291.         temp = temp->next;
292.     }
293.     cout << endl;
294. }
295.
296.
297. template<class T>
298. class chainHashTable
299. {
300. private:
301.     chainlist<T>* data;
```

```
302.     T divisor;
303.     int capacity;
304. public:
305.     chainHashTable(const T& ini_divisor);
306.     chainHashTable();
307.     void insert(const T& key);
308.     void find(const T& key);
309.     void erase(const T& key);
310. };
311.
312. template<class T>
313. chainHashTable<T>::chainHashTable()
314. {
315.     data = nullptr;
316.     divisor = T(0);
317.     capacity = 0;
318. }
319.
320. template<class T>
321. chainHashTable<T>::chainHashTable(const T& ini_divisor)
322. {
323.     this->divisor = ini_divisor;
324.     this->capacity = ini_divisor;
325.     data = new chainlist<T>[capacity];
326. }
327.
328. template<class T>
329. void chainHashTable<T>::insert(const T& key)
330. {
331.     int index = key % divisor;
332.     if (data[index].find_first(key) != -1)
333.     {
334.         cout << "Existed" << endl;
335.     }
336.     else
337.     {
338.         data[index].push_back(key);
339.     }
340. }
341.
342. template<class T>
343. void chainHashTable<T>::find(const T& key)
344. {
345.     int index = key % divisor;
346.     if (data[index].find_first(key) == -1)
347.     {
```

```
348.            cout << "Not Found" << endl;
349.        }
350.        else
351.        {
352.            cout << data[index].getSize() << endl;
353.        }
354.    }
355.
356.    template<class T>
357.    void chainHashTable<T>::erase(const T& key)
358.    {
359.        int index = key % divisor;
360.        if (data[index].find_first(key) == -1)
361.        {
362.            cout << "Delete Failed" << endl;
363.        }
364.        else
365.        {
366.            data[index].erase(data[index].find_first(key));
367.            cout << data[index].getSize() << endl;
368.        }
369.    }
370.
371.    class Solution
372.    {
373.    public:
374.        void solve();
375.    };
376.    void Solution::solve()
377.    {
378.        int n, m;
379.        cin >> n >> m;
380.        chainHashTable<int> hashTable(n);
381.        for (int i = 0; i < m; i++)
382.        {
383.            int a, b;
384.            cin >> a >> b;
385.            if (a == 0)
386.            {
387.                hashTable.insert(b);
388.            }
389.            else if (a == 1)
390.            {
391.                hashTable.find(b);
392.            }
393.            else if (a == 2)
```

```
            {
                hashTable.erase(b);
            }
        }
    }

    int main()
    {
        Solution solute;
        solute.solve();
        return 0;
    }
```