

数据结构与算法 II 上机实验 (9.29)

中国人民大学 信息学院 崔冠宇 2018202147

上机题 1 用蒙特卡洛法和分治策略找到数组中的多数（出现次数多于一半的数）。

解：

多数问题的蒙特卡洛法是采取随机抽样，假定该元素是多数，然后扫描数组进行验证，若超出一半，则确实是多数。重复多次，直至错误判断的概率小于给定阈值为止。

根据老师课上的 PPT 分析，这是一个偏真的 $1/2$ 正确的蒙特卡洛算法，即如果数组含有主元素，则算法以 $> 1/2$ 的概率返回真；如果数组没有主元素，则算法肯定返回假。

采用重复调用方法，要使算法返回错误结果的概率不超过 ε ，则要至多重复调用 $\lceil \log(1/\varepsilon) \rceil$ 次。由于一次运行的时间复杂度为 $\Theta(n)$ ，则重复调用的时间复杂度为 $T(n) = O(n \log(1/\varepsilon))$ ，其中 ε 是指定阈值。

多数问题的分治算法则是将数组对半划分 ($2T(n/2)$)，分别递归求解多数，然后进行比较：

1. 如果左右多数相同，则它就是整个数组的多数；
2. 若左右多数不同，或有一边不存在多数，则需要扫描整个数组，确认二者究竟谁为多数 ($\Theta(n)$)。

所以根据分析可以写出 $T(n) \leq 2T(n/2) + \Theta(n)$ ，根据主定理， $T(n) = O(n \log n)$ 。

程序运行截图：运行 majority.cpp，可以发现蒙特卡洛法并不能保证结果的正确性，有一定的错误率；而确定性的分治法则能保证结果的正确性：

```
CuiGuanyu@Cui: ~  
Last login: Sat Oct 3 11:54:15 on ttys000  
CuiGuanyu@Cui ~$ ./Users/CuiGuanyu/Desktop/9.29/上机实验2/majority  
分治法求多数：  
多数：1  
蒙特卡洛法求多数：  
多数：1  
CuiGuanyu@Cui ~$ ./Users/CuiGuanyu/Desktop/9.29/上机实验2/majority  
分治法求多数：  
多数：1  
蒙特卡洛法求多数：  
不存在多数。  
CuiGuanyu@Cui ~$
```

上机题 2 用随机技术实现跳跃表。

解：

跳跃表在操作系统中地址变换一节有重要的作用，它与链表类似，只是有一些节点有多级指针，可以理解为是不同“步长”的链表贴在了一起。

查询时，要注意从最高级的头节点开始查找；删除时，也要注意修改前面各级节点的指针；体现跳跃表随机性的一点就是插入操作，在插入时，利用随机数生成器反复生成随机数，以 p 的概率升级节点，但不能超过某个给定上界，然后再查找到新节点的插入位置，并设置指针。

算法复杂度分析

时间复杂度不做要求，所以直接利用老师给出的结论：

- 最坏情况下：退化为链表， $T(n) = O(n + \max Level)$ ；
- 一般情况下，利用随机技术， $T(n) = O(\log n)$ 。

空间复杂度：根据插入时的规则，容易看出 i 级链有 np^i 个，所以额外辅助空间为 $S(n) = \sum_{i=1}^{\infty} np^i = \frac{n}{1-p} = O(n)$ 。

程序运行截图：运行 `skiplist.cpp`，因为多级链不容易画出，所以主要测试了表的插入和删除：

```
CuiGuanyu@Cui ~ /Users/CuiGuanyu/Desktop/9.29/上机实验2/skiplist
测试插入节点：
(空)
(0, 2)
(0, 2) -> (1, 2)
(0, 2) -> (1, 2) -> (2, 1)
(0, 2) -> (1, 2) -> (2, 1) -> (3, 2)
测试节点赋值：
随机赋值键为0的节点：
(0, 1) -> (1, 2) -> (2, 1) -> (3, 2)
随机赋值键为1的节点：
(0, 1) -> (1, 1) -> (2, 1) -> (3, 2)
随机赋值键为2的节点：
(0, 1) -> (1, 1) -> (2, 1) -> (3, 2)
随机赋值键为3的节点：
(0, 1) -> (1, 1) -> (2, 1) -> (3, 1)
测试节点删除：
生成的键随机序列为：
2 0 1 3
随机删除键为2的节点：
(0, 1) -> (1, 1) -> (3, 1)
随机删除键为0的节点：
(1, 1) -> (3, 1)
随机删除键为1的节点：
(3, 1)
随机删除键为3的节点：
(空)
CuiGuanyu@Cui ~
```

附录——多数问题的两种解法: **majority.cpp**:

```
1 #include <iostream>
2 #include <vector>
3 #include <random>
4 #include <cmath>
5 #include <optional>
6
7 // 多数问题-蒙特卡洛
8 // 输入: std::vector<T> 数组, double 容忍错误率
9 // 输出: std::optional<T> 多数(可以不存在)
10 // 时间复杂度:  $T(n) = O(n \log(1/\epsilon))$ 
11 template <typename T>
12 std::optional<T> Majority_MC(const std::vector<T> & vec, double epsilon)
13 {
14     // 将用于为随机数引擎获得种子
15     std::random_device rd;
16     // 以播种标准 mersenne_twister_engine
17     std::mt19937 gen(rd());
18     // 随机整数生成器 [a, b]
19     std::uniform_int_distribution<> dis(0, vec.size() - 1);
20     // 重复多次
21     unsigned long long times = static_cast<unsigned long long>(std::log(1 / epsilon)
22 );
23     for(unsigned long long i = 0; i < times; i++)
24     {
25         // 随机数
26         int rnd = dis(gen);
27         // 假定多数
28         T majority = vec[rnd];
29         // 计数
30         unsigned long long count = 0;
31         // 扫一遍
32         for(size_t j = 0; j < vec.size(); j++)
33         {
34             // 增加计数
35             if(majority == vec[j])
```

```

36         count++;
37     }
38 }
39 // 超过半数
40 if(count >= vec.size() / 2)
41 {
42     return std::optional<T>(majority);
43 }
44 }
45 // 无多数
46 return std::optional<T>(std::nullopt);
47 }
48
49 // 多数问题-分治法
50 // 输入: std::vector<T> 数组, size_t 左端下标, size_t 右端下标
51 // 输出: std::optional<T> 多数(可以不存在)
52 // 复杂度:  $T(n) = O(n \log n)$ 
53 template <typename T>
54 std::optional<T> Majority_DC(const std::vector<T> & vec, size_t left, size_t right)
55 {
56     // 个数
57     size_t size = right - left + 1;
58     // 递归出口: 仅一个元素
59     if(size == 1)
60     {
61         return std::optional<T>(vec[left]);
62     }
63     // 中间
64     size_t middle = (left + right) / 2;
65     // 两个子问题
66     //  $2T(n/2)$ 
67     auto leftMajority = Majority_DC(vec, left, middle);
68     auto rightMajority = Majority_DC(vec, middle + 1, right);
69     // 左右多数相同 (必须有值), 则就是整个数组多数
70     if(leftMajority == rightMajority
71         && leftMajority.has_value()
72         && rightMajority.has_value())

```

```

73     {
74         return std::optional<T>(leftMajority);
75     }
76     // 左右不同(或有一边无值), 扫描整个数组, 确定二者谁多
77     // O(n)
78     unsigned long long leftCount = 0, rightCount = 0;
79     for(size_t i = left; i <= right; i++)
80     {
81         if(vec[i] == leftMajority)
82         {
83             leftCount++;
84         }
85         if(vec[i] == rightMajority)
86         {
87             rightCount++;
88         }
89     }
90     // 多于一半
91     if(leftCount > size / 2)
92     {
93         return std::optional<T>(leftMajority);
94     }
95     if(rightCount > size / 2)
96     {
97         return std::optional<T>(rightMajority);
98     }
99     // 无多数
100    //  $T(n) = 2T(n/2) + O(n)$ 
101    // -->  $T(n) = O(n \log n)$ 
102    return std::optional<T>(std::nullopt);
103 }
104
105 int main(int argc, char *argv[])
106 {
107     // 一个简单的例子
108     // 多次运行观察不同结果
109     std::vector<int> v = {1, 2, 3, 4, 1, 1, 1, 2, 1};

```

```

110
111 // 分治法
112 // 总是输出多数为1，结果正确
113 std::cout << "分治法求多数:" << std::endl;
114 auto retDC = Majority_DC(v, 0, v.size() - 1);
115 if(retDC.has_value())
116 {
117     std::cout << "多数: " << retDC.value() << std::endl;
118 }
119 else
120 {
121     std::cout << "不存在多数。" << std::endl;
122 }
123
124 // 蒙特卡洛法
125 // 多数情况输出为1，少数情况输出不存在多数
126 std::cout << "蒙特卡洛法求多数:" << std::endl;
127 // 错误概率容忍度
128 // 稍微写大一些容忍度，使得结果更多样
129 auto retMC = Majority_MC(v, 0.2);
130 if(retMC.has_value())
131 {
132     std::cout << "多数: " << retMC.value() << std::endl;
133 }
134 else
135 {
136     std::cout << "不存在多数。" << std::endl;
137 }
138 return 0;
139 }

```

附录——跳跃表的定义及实现: **skiplist.h**:

```

1 #ifndef SKIP_LIST_H
2 #define SKIP_LIST_H
3
4 #include <vector>
5 #include <utility>

```

```

6 #include <optional>
7 #include <random>
8 #include <functional>
9 #include <iostream>
10
11 template <typename K, typename V>
12 class SkipListNode
13 {
14     public:
15         // 构造函数
16         SkipListNode(std::optional<K> key, std::optional<V> value, long long level)
17         ;
18         // 析构函数
19         ~SkipListNode();
20         // 获取键
21         K getKey();
22         // 获取值
23         V getValue();
24         // 设置新值
25         void setValue(V value);
26         // 设置 _level 级别的指针
27         void setNext(long long level, SkipListNode<K, V> * p);
28         // 获得下一个
29         std::vector<SkipListNode<K, V> *> & getNext();
30     private:
31         // 键(头节点的键可以不存在)
32         std::optional<K> _key;
33         // 值(头节点的值可以不存在)
34         std::optional<V> _value;
35         // 各级指针
36         std::vector<SkipListNode<K, V> *> _next;
37 };
38
39 template <typename K, typename V>
40 SkipListNode<K, V>::SkipListNode(std::optional<K> key, std::optional<V> value, long
    long level)
41 : _key(key), _value(value)

```

```
41 {
42     // 0 ~ level 个空指针
43     for(long long i = 0; i <= level; i++)
44     {
45         _next.push_back(nullptr);
46     }
47 }
48
49 template <typename K, typename V>
50 SkipListNode<K, V>::~SkipListNode(){}
51
52 template <typename K, typename V>
53 K SkipListNode<K, V>::getKey()
54 {
55     return _key.value();
56 }
57
58 template <typename K, typename V>
59 V SkipListNode<K, V>::getValue()
60 {
61     return _value.value();
62 }
63
64 template <typename K, typename V>
65 void SkipListNode<K, V>::setValue(V value)
66 {
67     _value = value;
68 }
69
70 template <typename K, typename V>
71 void SkipListNode<K, V>::setNext(long long level, SkipListNode<K, V> * p)
72 {
73     assert(0 <= level && level < _next.size());
74     _next[level] = p;
75 }
76
77 template <typename K, typename V>
```



```

78 std::vector<SkipListNode<K, V> *> & SkipListNode<K, V>::getNext()
79 {
80     return _next;
81 }
82
83 template <typename K, typename V, class Compare = std::less<K>>
84 class SkipList
85 {
86     //using iterator = SkipListNode<K, V> *;
87     public:
88         // 构造函数
89         SkipList(double prob = 0.5, long long capacity = 1);
90         // 析构函数
91         ~SkipList();
92         // 查找
93         const SkipListNode<K, V> * find(K key);
94         // 改值
95         bool assign(K key, V value);
96         // 增加
97         std::pair<SkipListNode<K, V> *, bool> insert(K key, V value);
98         // 删除
99         SkipListNode<K, V> * erase(K key);
100        // 返回元素个数
101        long long size();
102        // 返回容量
103        long long capacity();
104        // 打印
105        void print();
106    private:
107        // 找到 _key 对应位置的前一个位置
108        // 返回值第一个是等于 _key 的位置，如果没有，返回tail
109        // 返回值第二个是等于 _key 之前的各级节点的位置
110        std::pair<SkipListNode<K, V> *,
111            std::vector<SkipListNode<K, V> *>> findPrevPos(K key);
112        // 首节点指针
113        SkipListNode<K, V> * _head;
114        // 尾节点指针

```

```

115     SkipListNode<K, V> * _tail;
116     // 当前元素个数
117     long long _size;
118     // 元素个数达到的上界, 即STL容器库容量的概念
119     long long _capacity;
120     // 最大允许层数 =  $\log_{1/p}(\text{capacity})$ 
121     long long _maxLevel;
122     // 目前层数
123     long long _nowLevel;
124     // 将用于为随机数引擎获得种子
125     std::random_device _rd;
126     // 以播种标准 mersenne_twister_engine
127     std::mt19937 _gen;
128     // 随机数分布生成器 [a, b)
129     std::uniform_real_distribution<double> _dis;
130     // 概率
131     double _prob;
132 };
133
134 template <typename K, typename V, class Compare>
135 SkipList<K, V, Compare>::SkipList(double prob, long long capacity)
136 : _head(nullptr), _tail(nullptr), _size(0), _capacity(capacity),
137 _nowLevel(0), _dis(0.0, 1.0), _gen(_rd()),
138 _prob(prob)
139 {
140     // 保证 prob 合法
141     assert(0 < _prob && _prob <= 1);
142     // 当前允许的最大层次 =  $\log_{1/p}(\text{capacity})$ 
143     _maxLevel = std::log(_capacity) / std::log(1 / _prob);
144     // 空头节点
145     _head = new SkipListNode<K, V>(std::nullopt, std::nullopt, _maxLevel);
146     // 空尾节点
147     _tail = new SkipListNode<K, V>(std::nullopt, std::nullopt, 0);
148     // 头节点连尾节点
149     for(long long i = 0; i <= _maxLevel; i++)
150     {
151         _head -> setNext(i, _tail);

```

```

152     }
153 }
154
155 template <typename K, typename V, class Compare>
156 SkipList<K, V, Compare>::~SkipList()
157 {
158     // 析构所有节点
159     SkipListNode<K, V> * p = _head;
160     // 沿着节点往后捋
161     while(p != nullptr)
162     {
163         // 下一个节点
164         SkipListNode<K, V> * pNext = p -> getNext()[0];
165         // 析构本节点
166         delete p;
167         // 前进
168         p = pNext;
169     }
170 }
171
172 //
173 template <typename K, typename V, class Compare>
174 std::pair<SkipListNode<K, V> *,
175         std::vector<SkipListNode<K, V> *>>
176         SkipList<K, V, Compare>::findPrevPos(K key)
177 {
178     Compare cmp = Compare();
179     // 找到的各级指针(从高级到低级)
180     std::vector<SkipListNode<K, V> *> prev;
181     // 从最高级往下找
182     for(long long i = _maxLevel; i >= 0; i--)
183     {
184         // 本节点指针与下一节点指针
185         SkipListNode<K, V> * p = _head;
186         SkipListNode<K, V> * pNext = p -> getNext()[i];
187         // pNext 为尾节点表示 p 已到最后, 不必前进了
188         while(pNext != _tail)

```

```

189         {
190             // 直到 pNext -> key >= _key
191             if(!cmp(pNext -> getKey(), key))
192             {
193                 break;
194             }
195             p = pNext;
196             pNext = pNext -> getNext()[i];
197         }
198         prev.push_back(p);
199     }
200     // 后一个0级节点
201     SkipListNode<K, V> * found = prev[prev.size() - 1] -> getNext()[0];
202     // 后一个节点是结尾
203     if(found == _tail)
204     {
205         return std::pair(_tail, prev);
206     }
207     // 后一个节点的键是要查找的键
208     if(found -> getKey() == key)
209     {
210         return std::pair(found, prev);
211     }
212     // 后一个节点的键并不是待查找的键
213     return std::pair(_tail, prev);
214 }
215
216 template <typename K, typename V, class Compare>
217 const SkipListNode<K, V> * SkipList<K, V, Compare>::find(K key)
218 {
219     return findPrevPos(key).first;
220 }
221
222 template <typename K, typename V, class Compare>
223 bool SkipList<K, V, Compare>::assign(K key, V value)
224 {
225     std::pair<SkipListNode <K, V> *,

```

```
226         std::vector<SkipListNode<K, V> *>> res = findPrevPos(key);
227     // 没找到
228     if(res.first == _tail)
229     {
230         return false;
231     }
232     res.first -> setValue(value);
233     return true;
234 }
235
236 template <typename K, typename V, class Compare>
237 std::pair<SkipListNode<K, V> *, bool> SkipList<K, V, Compare>::insert(K key, V
    value)
238 {
239     std::pair<SkipListNode<K, V> *,
240         std::vector<SkipListNode<K, V> *>> res = findPrevPos(key);
241
242     // 有重复的键
243     if(res.first != _tail)
244     {
245         std::cout << "键重复!" << std::endl;
246         return std::pair(res.first, false);
247     }
248
249     // 产生新节点的等级
250     auto generateLevel = [&]() -> long long
251     {
252         long long _level = 0;
253         while(_dis(_gen) <= _prob && _level < _maxLevel)
254         {
255             _level++;
256         }
257         return _level;
258     };
259     // 获得该新节点的等级
260     long long level = generateLevel();
261     // 可能更新目前等级
```

```

262     if(level > _nowLevel)
263     {
264         _nowLevel = level;
265     }
266
267     // 生成节点
268     SkipListNode<K, V> * newNode =
269         new SkipListNode<K, V>(key, value, level);
270     // 插入, 修改指针
271     // 注意返回的vector中的指针是从高级到低级的
272     for(long long i = 0; i <= level; i++)
273     {
274         // 将新节点的第 i 级指针设置为
275         // 插入位置之前的 i 级节点的下一个 i 级节点
276         newNode -> setNext(i,
277             res.second[res.second.size() - 1 - i] -> getNext()[i]);
278         res.second[res.second.size() - 1 - i] -> setNext(i, newNode);
279     }
280
281     // 因为节点个数增加, 一系列操作:
282     // 更新容量、计算最大允许等级、修改头节点等级.....
283     // 节点计数增加
284     _size++;
285     // 超过容量
286     if(_size > _capacity)
287     {
288         _capacity *= 2;
289         // 当前允许的最大层次 =  $\log_{1/p}(\text{capacity})$ 
290         long long newMaxLevel = std::log(_capacity) / std::log(1 / _prob);
291         // 增加头节点等级
292         if(newMaxLevel > _maxLevel)
293         {
294             for(long long i = 0; i < newMaxLevel - _maxLevel; i++)
295             {
296                 _head -> getNext().push_back(_tail);
297             }
298             _maxLevel = newMaxLevel;

```

```

299     }
300 }
301 return std::pair(newNode, true);
302 }
303
304 template <typename K, typename V, class Compare>
305 SkipListNode<K, V> * SkipList<K, V, Compare>::erase(K key)
306 {
307     std::pair<SkipListNode<K, V> *,
308         std::vector<SkipListNode<K, V> *>> ret = findPrevPos(key);
309     // 没找到
310     if(ret.first == _tail)
311     {
312         return nullptr;
313     }
314     // 获取本节点等级
315     long long level = ret.first -> getNext().size() - 1;
316     // 设置之前的节点的后面指针
317     for(long long i = 0; i <= level; i++)
318     {
319         ret.second[ret.second.size() - 1 - i] -> setNext(i, ret.first -> getNext()[
320 i]);
321     }
322     // 仅减少元素数目，不减少容量
323     _size--;
324     return ret.first;
325 }
326
327 template <typename K, typename V, class Compare>
328 long long SkipList<K, V, Compare>::size()
329 {
330     return _size;
331 }
332
333 template <typename K, typename V, class Compare>
334 long long SkipList<K, V, Compare>::capacity()
335 {

```

```

335     return _capacity;
336 }
337
338 template <typename K, typename V, class Compare>
339 void SkipList<K, V, Compare>::print()
340 {
341     SkipListNode<K, V> * p = _head -> getNext()[0];
342     if(p == _tail)
343     {
344         std::cout << "(空)" << std::endl;
345         return;
346     }
347     while(p -> getNext()[0] != _tail)
348     {
349         std::cout << "(" << p -> getKey() << ", " << p -> getValue() << ")" -> ";
350         p = p -> getNext()[0];
351     }
352     std::cout << "(" << p -> getKey() << ", " << p -> getValue() << ")" << std::
    endl;
353 }
354 #endif

```

附录——跳跃表测试: **skiplist.cpp**:

```

1 #include <iostream>
2 #include <iterator>
3 #include "skiplist.h"
4
5 int main(int argc, char * argv[])
6 {
7     // 测试节点数
8     const int testTimes = 4;
9
10    // 将用于为随机数引擎获得种子
11    std::random_device rd;
12    // 以播种标准 mersenne_twister_engine
13    std::mt19937 gen(rd());
14    // 随机数分布生成器 [a, b]

```



```
15     std::uniform_int_distribution<> dis(0, testTimes - 1);
16
17     SkipList<int, int> l(.8, 10);
18     // 测试插入节点
19     std::cout << "测试插入节点: " << std::endl;
20     l.print();
21     for(int i = 0; i < testTimes; i++)
22     {
23         l.insert(i, dis(gen));
24         l.print();
25     }
26     // 测试赋值节点
27     std::cout << "测试节点赋值: " << std::endl;
28     for(int i = 0; i < testTimes; i++)
29     {
30         std::cout << "随机赋值键为 " << i << "的节点: " << std::endl;
31         l.assign(i, dis(gen));
32         l.print();
33     }
34     // 测试删除节点
35     std::cout << "测试节点删除: " << std::endl;
36     // 先生成随机序列
37     std::vector<int> v;
38     for(int i = 0; i < testTimes; i++)
39     {
40         v.push_back(i);
41     }
42     std::shuffle(v.begin(), v.end(), gen);
43     std::cout << "生成的键随机序列为:" << std::endl;
44     std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
45     std::cout << std::endl;
46     for(int i = 0; i < testTimes; i++)
47     {
48         std::cout << "随机删除键为 " << v[i] << "的节点:" << std::endl;
49         l.erase(v[i]);
50         l.print();
51     }
```

```
52     return 0;
53 }
```