

# Python程序设计报告

设计题目：实现LRU缓存机制

学生姓名：

专 业：计算机科学与技术

班 级：一班

学 号：

指导教师：

完成日期

# （一）需求和规格说明

**问题描述：**LRU是Least Recently Used的缩写，即最近最少使用，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。

**编程任务：**

实现 LRUCache 类：

1. LRUCache(int capacity) 以 正整数 作为容量 capacity 初始化 LRU 缓存
2. int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
3. void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。

函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

# （二）设计

## 1．设计思想

本题主要考察对于双向链表和哈希表的综合运用。直接实现get和put总要遍历一遍整个数组或者链表，再根据其存放时间做出对应的操作，但是这样的话时间复杂度就到了O(n)。所以采用两种不同的存储方式共同存储同一份数据，虽然用的空间多了，但是时间复杂度从O(n) 降低到了O(1)，符合要求。首先使用双向链表按照被使用的顺序存储了这些键值对，**靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。**哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置，用C++描述哈希表就是unordered\_map<int,Node\*>。通过key得到对应的节点，拿到了节点就在双向链表中进行操作。对于 get 操作，首先判断 key 是否存在：如果 key 不存在，则返回 −1；如果 key 存在，则 key 对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。对于 put 操作，首先判断 key 是否存在：如果 key 不存在，使用 key 和 value 创建一个新的节点，在双向链表的头部添加该节点，并将 key 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；如果 key 存在，则与 get 操作类似，先通过哈希表定位，再将对应的节点的值更新为 value，并将该节点移到双向链表的头部。这样的话，无论是存还是取数据都能在O(1)的时间复杂度内完成。

## 2. 设计表示

为了实现LRU机制，设计了一个双向链表，实现了链表的常见方法，头尾增删，指定位置增删，反转，赋值等操作。但是要模拟LRU机制只需要用到其中一部分即可

1. 结点结构：

|  |  |  |  |
| --- | --- | --- | --- |
| **结构体名称** | **成员数据类型** | **成员名** | **描述** |
| Node | Node\* | \_next\_node | 下一个节点指针 |
| Node\* | \_prev\_node | 上一个节点指针 |
| int | \_data | 数据 |
| Function | Set\_data() | 设置数据 |
| Function | Get\_data() | 得到数据 |
| Function | Set\_prev\_node | 设置前一个节点 |
| Function | Get\_prev\_node | 得到前一个节点 |
| Function | Set\_next\_node | 设置下一个节点 |
| Function | Get\_next\_node | 得到下一个节点 |

1. List类：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| list | 函数 | void | Append(val) | 尾插 |
| void | Insert(val,index) | 指定位置插入元素 |
| void | Remove(index) | 指定位置移除元素 |
| void | Pop() | 尾删 |
| void | At(index) | 根据下标找到节点 |
| string | \_\_str\_\_ | 打印所有结点数据 |
| void | Push\_front() | 头插 |
| Void | Pop\_front() | 头删 |
| void | Assign(list[int]) | 用数组赋值给链表 |
| void | Front() | 返回头节点 |
| void | Back() | 返回尾节点 |
| Void | Reverse() | 反转链表 |
| void\* | Clear() | 清空链表 |
| 数据 | int | \_size | 结点数 |
| Node\* | \_head\_node | 指向头结点的指针 |
| Node\* | \_tail\_node | 尾指针 |

1. LRUCache类

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| LRUCache | 数据 | int | \_cap | 容量 |
| map | \_dataMap | 存放数据的哈希表 |
| List | \_dataList | 存放数据的双向链表 |
| 函数 | int | Get() |  |
| void | put() |  |
| void | \_move\_to\_front(Node\*) | 把目标节点移动到头节点位置 |

## 3. 核心算法

# 核心算法是通过双向链表维护缓存数据的访问顺序，最近访问的数据始终在链表头部，最久未访问的数据在链表尾部；同时通过哈希表快速定位链表中的节点。每次访问数据时，先通过哈希表O(1) 找到对应节点，将其移动到链表头部以标记为最近使用；当插入新数据时，如果缓存未满，则直接将数据插入链表头部并更新哈希表；若缓存已满，则删除链表尾部的节点（表示最久未使用的数据），同时移除哈希表中的对应键，然后将新数据插入链表头部并添加到哈希表中。通过这种方式，双向链表保证了节点的有序性，哈希表保证了数据的快速定位，从而实现了O(1) 时间复杂度的访问和更新操作。

# （三）用户手册

# 首先需要实例化一个带有LRUCache 对象，例如 LRUCache(2) 创建一个容量为 2 的缓存。通过 put(key, value) 方法可以向缓存中添加数据，如果插入的新数据导致缓存超出容量，则会删去最久未使用的键值对。使用 get(key) 方法可以根据键查询对应的值，如果该键存在，则返回其值并将该键标记为最近使用，并在内部更新它的值，若键不存在则返回 -1。缓存内部通过一个双向链表维护访问顺序，最近访问的数据在链表头部，最久未使用的数据在链表尾部；同时，哈希表用于快速定位链表节点以支持高效操作。使用此缓存时，无需关心底层逻辑，只需通过 put 和 get 方法操作数据即可实现O(1)时间复杂度的增改查。

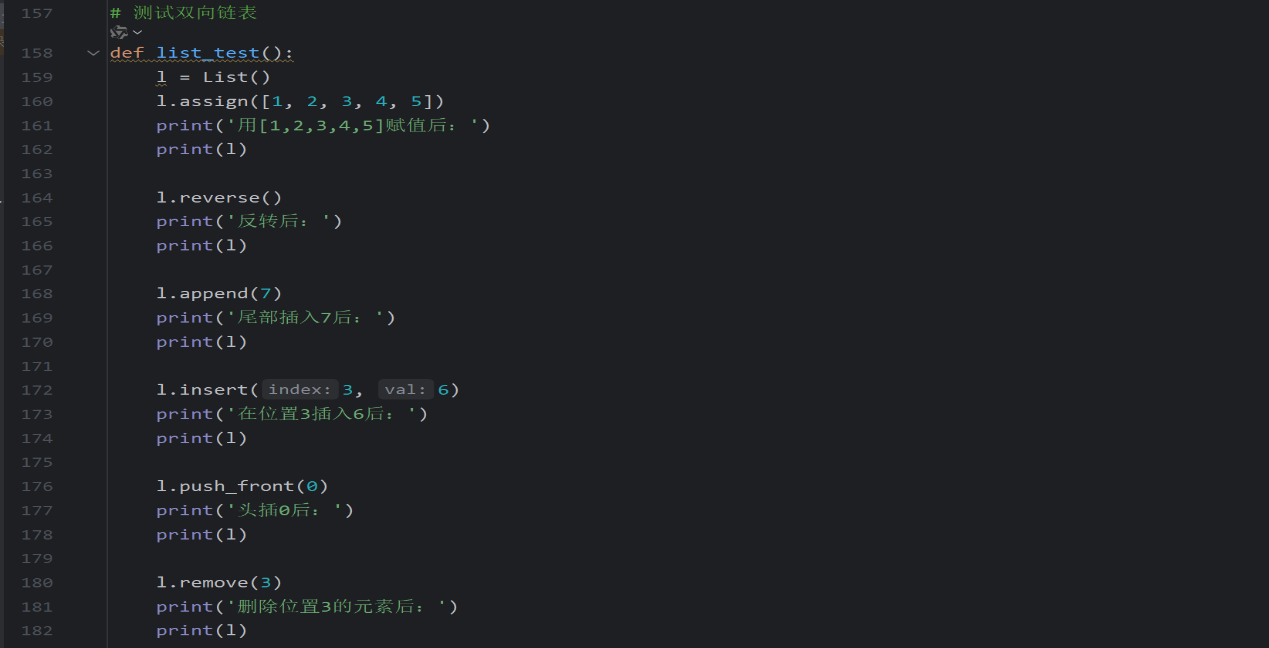
# （四）调试及测试

## 1. 测试数据：

首先，由于我封装了一个双向链表，所以先测试下链表的基本操作，其次再测试LRU缓存机制对不对。

双向链表测试；

文本

描述已自动生成

下面是LRU机制测试：

文本

描述已自动生成

## 2. 测试结果：

双向链表测试结果：

图片包含 形状

描述已自动生成

图片包含 文本

描述已自动生成

LRU缓存测试结果：

形状

中度可信度描述已自动生成

可以看到都是符合预期的

## 进一步改进：

现实生活中，LRU缓存机制应用广泛，但是底层就是这个机制，我们可以再新加功能以符合不同的使用场景，比如容量在构造的时候就已经确定好了，我们可以再设置一个set\_cap函数，再去动态的调整容量，然后在实现双向链表时，需要多次进行指针操作，一个错误就可能导致程序崩溃，所以可以通过python内置的高效链表实现，不过选这个LRU机制就是想练练双向链表的写法

# (五) 感想

这道题并不难，但是涉及到的知识点都是必须要掌握的，这道题的原型LRU缓存机制也是广为使用的，在实现 LRU 缓存机制的过程中，主要挑战在于如何在O(1) 时间复杂度内完成插入、删除和访问操作。解决这个问题需要同时利用哈希表和双向链表，哈希表用于快速定位节点，双向链表维护数据的访问顺序，实现了这两种数据结构的互补。 跟让人感受深刻的是，LRU 缓存机制不仅是一个数据结构问题，它更是高效资源管理的经典案例。它在操作系统的页面置换、数据库的缓存、浏览器的历史记录管理等领域都有广泛应用。这种基础算法的实现和优化过程，帮助了我理解计算机系统设计背后的核心思想，同时也能提升代码实现和优化的能力。

# 附录：

1. **class** Node:
2. **def** \_\_init\_\_(self, val: int = 0):
3. self.\_data: int = val
4. self.\_next\_node: 'Node' = None
5. self.\_prev\_node: 'Node' = None
7. # getter 和 setter方法
8. **def** set\_data(self, val: int):
9. self.\_data = val
11. **def** get\_data(self) -> int:
12. **return** self.\_data
14. **def** set\_next\_node(self, node: 'Node'):
15. self.\_next\_node = node
17. **def** get\_next\_node(self) -> 'Node':
18. **return** self.\_next\_node
20. **def** set\_prev\_node(self, node: 'Node'):
21. self.\_prev\_node = node
23. **def** get\_prev\_node(self) -> 'Node':
24. **return** self.\_prev\_node

27. # 双向循环链表
28. **class** List:
29. **def** \_\_init\_\_(self):
30. self.\_head\_node = Node()
31. self.\_tail\_node = self.\_head\_node
32. self.\_size = 0
34. # 尾部插入元素
35. **def** append(self, val: int):
36. self.\_tail\_node.set\_next\_node(Node(val))
37. self.\_tail\_node.get\_next\_node().set\_prev\_node(self.\_tail\_node)
38. self.\_tail\_node = self.\_tail\_node.get\_next\_node()
39. self.\_size += 1
41. # 指定位置插入元素
42. **def** insert(self, index: int, val: int):
43. # 索引越界，直接抛出异常
44. **if** index > self.\_size:
45. **raise** IndexError('Index out of range.')
46. **elif** index == self.\_size:  # 如果索引是长度，那就等价于尾插法
47. self.append(val)
48. **else**:
49. temp = self.\_head\_node
50. **for** \_ **in** range(index):
51. temp = temp.get\_next\_node()
52. temp.get\_next\_node().set\_prev\_node(Node(val))
53. temp.get\_next\_node().get\_prev\_node().set\_next\_node(temp.get\_next\_node())
54. temp.set\_next\_node(temp.get\_next\_node().get\_prev\_node())
55. temp.get\_next\_node().set\_prev\_node(temp)
56. self.\_size += 1
58. # 指定位置移除元素
59. **def** remove(self, index: int):
60. **if** index >= self.\_size:
61. **raise** IndexError('Index out of range.')
62. t = self.\_head\_node
63. **for** \_ **in** range(index):
64. t = t.get\_next\_node()  #循环找到要删除元素的前一个元素
65. rm = t.get\_next\_node()
66. t.set\_next\_node(rm.get\_next\_node())
67. **if** rm.get\_next\_node():
68. rm.get\_next\_node().set\_prev\_node(t)
69. **else**:
70. self.\_tail\_node = t
71. self.\_size -= 1
73. # 尾部删除元素，并返回
74. **def** pop(self) -> int:
75. **if** self.\_size == 0:
76. **raise** RuntimeError('List is empty.')
77. val = self.\_tail\_node.get\_data()
78. self.\_tail\_node = self.\_tail\_node.get\_prev\_node()
79. **if** self.\_tail\_node:
80. self.\_tail\_node.set\_next\_node(None)
81. self.\_size -= 1
82. **return** val
84. # 根据下标找到对应元素
85. **def** at(self, index: int) -> int:
86. **if** index >= self.\_size:
87. **raise** IndexError('Index out of range.')
88. t = self.\_head\_node.get\_next\_node()
89. **for** \_ **in** range(index):
90. t = t.get\_next\_node()
91. **return** t.get\_data()
93. # 头插
94. **def** push\_front(self, val: int):
95. self.insert(0, val)
97. # 头删，并返回值
98. **def** pop\_front(self) -> int:
99. **return** self.remove(0)
101. **def** \_\_str\_\_(self):
102. temp = self.\_head\_node.get\_next\_node()
103. **while** temp:
104. **print**(temp.get\_data(), end=' ')
105. temp = temp.get\_next\_node()
106. **return** '\n'
108. **def** size(self) -> int:
109. **return** self.\_size
111. **def** empty(self) -> bool:
112. **return** self.\_size == 0
114. # 反转链表
115. **def** reverse(self):
116. **if** self.\_size == 0:
117. **return**
118. t = self.\_head\_node.get\_next\_node()
119. nxt = self.\_head\_node.get\_next\_node()
120. **while** t **and** t.get\_next\_node():
121. nxt = t.get\_next\_node()
122. t.set\_next\_node(t.get\_prev\_node())
123. t.set\_prev\_node(nxt)
124. t = nxt
126. self.\_tail\_node = self.\_head\_node.get\_next\_node()
127. self.\_tail\_node.set\_next\_node(None)
128. t.get\_prev\_node().set\_prev\_node(t)
129. t.set\_next\_node(t.get\_prev\_node())
130. t.set\_prev\_node(self.\_head\_node)
131. self.\_head\_node.set\_next\_node(t)
133. # 清空整个链表
134. **def** clear(self):
135. self.\_head\_node.set\_next\_node(None)
136. self.\_tail\_node = self.\_head\_node
137. self.\_size = 0
139. # 返回头节点，注意head的next节点才是真正的头节点
140. **def** front(self) -> int:
141. **if** self.\_size == 0:
142. **raise** RuntimeError('List is empty.')
143. **return** self.\_head\_node.get\_next\_node().get\_data()
145. # 返回尾节点
146. **def** back(self) -> int:
147. **if** self.\_size == 0:
148. **raise** RuntimeError('List is empty.')
149. **return** self.\_tail\_node.get\_data()
151. # 传入列表的赋值操作
152. **def** assign(self, values: list[int]):
153. self.clear()  #赋值前先清空容器
154. **for** val **in** values:
155. self.append(val)
157. # 测试双向链表
158. **def** list\_test():
159. l = List()
160. l.assign([1, 2, 3, 4, 5])
161. **print**('用[1,2,3,4,5]赋值后：')
162. **print**(l)
164. l.reverse()
165. **print**('反转后：')
166. **print**(l)
168. l.append(7)
169. **print**('尾部插入7后：')
170. **print**(l)
172. l.insert(3, 6)
173. **print**('在位置3插入6后：')
174. **print**(l)
176. l.push\_front(0)
177. **print**('头插0后：')
178. **print**(l)
180. l.remove(3)
181. **print**('删除位置3的元素后：')
182. **print**(l)
184. l.pop\_front()
185. **print**('头删后：')
186. **print**(l)
188. **print**(f'链表是否为空: {l.empty()}')
190. **print**(f'链表长度为: {l.size()}')
192. **print**(f'链表头元素为: {l.front()}')
194. **print**(f'链表尾元素为: {l.back()}')
196. **print**(f'链表是否为空: {l.empty()}')
197. **print**('链表清空后：')
198. l.clear()
199. **print**(f'链表是否为空: {l.empty()}')

202. # 实现LRU缓存机制类
203. **class** LRUCache:
204. **def** \_\_init\_\_(self, capacity: int):
205. self.\_cap = capacity  # 总容量
206. self.\_dataMap = {}  # 哈希表
207. self.\_dataList = List()  # 双向链表
209. **def** get(self, key: int) -> int:
210. **if** key **not** **in** self.\_dataMap:
211. **return** -1
212. # 移动到链表头部,表示最近访问
213. node = self.\_dataMap[key]
214. self.\_move\_to\_front(node)
215. **return** node.get\_data()
217. **def** put(self, key: int, value: int):
218. **if** key **in** self.\_dataMap:
219. # 如果键已存在，更新值并移动到头部
220. node = self.\_dataMap[key]
221. node.set\_data(value)
222. self.\_move\_to\_front(node)
223. **else**:
224. # 如果键不存在，需要插入新节点
225. **if** self.\_dataList.size() == self.\_cap:   # 缓存已满，移除链表尾部节点
226. # 先拿到尾节点
227. tail = self.\_dataList.\_tail\_node
228. # 哈希表同步删除
229. **del** self.\_dataMap[tail.get\_data()]
230. # 链表删除
231. self.\_dataList.pop()
232. # 插入新节点到链表头部
233. self.\_dataList.push\_front(value) # 链表头部插入
234. self.\_dataMap[key] = self.\_dataList.\_head\_node.get\_next\_node() # 哈希表同步插入

237. **def** \_move\_to\_front(self, node: Node):
238. # 找到移除节点的前后节点，分别指向其前后节点进行操作
239. pre = node.get\_prev\_node()
240. nxt = node.get\_next\_node()
241. # 从链表中移除当前节点
242. **if** pre:
243. pre.set\_next\_node(nxt)
244. **if** nxt:
245. nxt.set\_prev\_node(pre)
246. # 如果要删除的是尾节点，则更新尾节点
247. **if** node == self.\_dataList.\_tail\_node:
248. self.\_dataList.\_tail\_node = pre

251. # 插入到链表头部
252. node.set\_next\_node(self.\_dataList.\_head\_node.get\_next\_node())
253. node.set\_prev\_node(self.\_dataList.\_head\_node)
255. **if** self.\_dataList.\_head\_node.get\_next\_node():
256. self.\_dataList.\_head\_node.get\_next\_node().set\_prev\_node(node)
257. self.\_dataList.\_head\_node.set\_next\_node(node)


261. # 测试LRU缓存机制
262. **if** \_\_name\_\_ == "\_\_main\_\_":
263. list\_test()
264. lru = LRUCache(2)
265. lru.put(1,1) # 缓存是 {1=1}
266. lru.put(2,2) # 缓存是 {1=1, 2=2}
267. **print**(lru.get(1))  # 返回1
268. lru.put(3,3) # 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
269. **print**(lru.get(2)) # 返回 -1 (未找到)
270. lru.put(4,4) # 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
271. **print**(lru.get(1))  # 返回 -1 (未找到)
272. **print**(lru.get(3))  # 返回3
273. **print**(lru.get(4))  # 返回4

