

# 链表

数组各种操作的时间复杂度

插入: $O(n)$

删除: $O(n)$

修改: $O(1)$

查询: $O(n)$

## 线性表:

表中任意元素只有唯一的前驱,唯一的后继

头元素没有前驱,尾元素没有后继

```
5
6 class CA
7 {
8 public:
9     CA() { cout << "CA::CA()" << endl; } 已用时间 <= 6ms
10    CA(int n) { cout << "CA::CA()" << endl; }
11 };
12
13 int main()
14 {
15     CA a{};
16
17     return 0;
18 }
```

这样写才调默认构造

## CList.h

```
1  #pragma once
2
3  #include <utility>
4
5  // 双向链表
6  template<typename T>
7  class CList
8  {
9  public:
10     struct NODE
11     {
12         NODE() :m_pPre(nullptr), m_pNext(nullptr), m_val{} {}
13         NODE(const T& val):m_pPre(nullptr), m_pNext(nullptr), m_val(val) {}
14         T m_val; // 存储数据
15         NODE* m_pPre; // 前驱节点的指针
16         NODE* m_pNext; // 后继节点的指针
17     };
18     using PNODE = NODE*;
19 public:
```

```

20     CList();
21     CList(const CList& obj);
22     CList(CList&& obj);
23     CList(std::initializer_list<T> il);
24     CList& operator=(const CList& obj);
25     virtual ~CList();
26
27     // 插入
28     void PushHead(const T& val);
29     void PushBack(const T& val);
30     void Insert(PNODE pNode, const T& val); // 插入到指定位置的前面
31
32     // 删除
33     void PopHead();
34     void PopBack();
35     void Delete(PNODE pNode);
36
37     // 修改
38     void Modify(PNODE pNode, const T& val);
39
40     // 查询
41     PNODE Find(const T& val);
42
43     // 清空
44     void Clear();
45
46     // 获取个数
47     size_t GetCount();
48
49     // 判断是否为空
50     bool IsEmpty();
51 private:
52     void Init(); // 置空
53 private:
54     PNODE m_pHead; // 头哨兵指针
55     PNODE m_pTail; // 尾哨兵指针
56     size_t m_nCount; // 节点的个数
57 };
58
59 template<typename T>
60 inline CList<T>::CList()
61 {
62     Init();
63 }
64
65 template<typename T>
66 inline CList<T>::CList(const CList& obj)
67 {
68     Init();
69     *this = obj;
70 }
71
72 template<typename T>
73 inline CList<T>::CList(CList&& obj)
74 {
75     Init();
76     m_pHead = obj.m_pHead;
77     m_pTail = obj.m_pTail;

```

```

78     m_nCount = obj.m_nCount;
79     obj.m_pHead = nullptr;
80     obj.m_pTail = nullptr;
81     obj.m_nCount = 0;
82 }
83
84 template<typename T>
85 inline CList<T>::CList(std::initializer_list<T> il)
86 {
87     Init();
88
89     for (auto& val : il)
90     {
91         PushBack(val);
92     }
93 }
94
95 template<typename T>
96 inline CList<T>& CList<T>::operator=(const CList& obj)
97 {
98     if (*this == obj)
99     {
100         return *this;
101     }
102     // 清空自己
103     Clear();
104
105     // 循环插入
106     PNODE pNode = obj.m_pHead->m_pNext;
107     while (pNode != obj.m_pTail) // 从头结点开始遍历,直到尾哨兵
108     {
109         PushBack(pNode->obj.m_val);
110         pNode = pNode->m_pNext;
111     }
112     return *this;
113 }
114
115 template<typename T>
116 inline CList<T>::~~CList()
117 {
118     // 元素内容清掉
119     Clear();
120     // 释放哨兵
121     if (m_pHead != nullptr)
122     {
123         delete m_pHead;
124         m_pHead = nullptr;
125     }
126     if (m_pTail != nullptr)
127     {
128         delete m_pTail;
129         m_pTail = nullptr;
130     }
131 }
132
133 template<typename T>
134 inline void CList<T>::PushHead(const T& val)
135 {

```

```

136 // 头哨兵后面是头结点, val插到头节点前面, val就是新的头结点
137 Insert(m_pHead→m_pNext, val);
138 }
139
140 template<typename T>
141 inline void CList<T>::PushBack(const T& val)
142 {
143     Insert(m_pTail, val); // 插到尾哨兵前面
144 }
145
146 template<typename T>
147 inline void CList<T>::Insert(PNODE pNode, const T& val)
148 {
149     // 检查参数
150     if (pNode == nullptr)
151     {
152         return;
153     }
154     // 构造新结点
155     PNODE pNewNode = new NODE(val);
156     if (pNewNode == nullptr)
157     {
158         return;
159     }
160
161     // 插入新结点
162     PNODE pPre = pNode→m_pPre;
163     pPre→m_pNext = pNewNode;
164     pNewNode→m_pPre = pPre;
165     pNewNode→m_pNext = pNode;
166     pNode→m_pPre = pNewNode;
167
168     // 个数增加
169     ++m_nCount;
170 }
171
172 template<typename T>
173 inline void CList<T>::PopHead()
174 {
175     Delete(m_pHead→m_pNext);
176 }
177
178 template<typename T>
179 inline void CList<T>::PopBack()
180 {
181     Delete(m_pTail→m_pPre);
182 }
183
184 template<typename T>
185 inline void CList<T>::Delete(PNODE pNode)
186 {
187     // 检查参数
188     if (pNode == nullptr || pNode == m_pHead || pNode == m_pTail)
189     {
190         return;
191     }
192     // 拿到前驱和后继
193     PNODE pPre = pNode→m_pPre;

```

```

194     PNODE pNext = pNode->m_pNext;
195
196     pPre->m_pNext = pNext;
197     pNext->m_pPre = pPre;
198
199     // 删除
200     delete pNode;
201
202     // 个数减少
203     --m_nCount;
204 }
205
206 template<typename T>
207 inline void CList<T>::Modify(PNODE pNode, const T& val)
208 {
209     // 检查参数
210     if (pNode == nullptr || pNode == m_pHead || pNode == m_pTail)
211     {
212         return;
213     }
214     pNode->m_val = val;
215 }
216
217 template<typename T>
218 inline typename CList<T>::PNODE CList<T>::Find(const T& val)
219 {
220     // 保存头结点
221     PNODE pNode = m_pHead->m_pNext;
222     while (pNode != m_pTail) // 从头结点开始遍历,直到尾哨兵
223     {
224         if (pNode->m_val == val)
225         {
226             return pNode;
227         }
228         pNode = pNode->m_pNext;
229     }
230     return nullptr;
231 }
232
233 template<typename T>
234 inline void CList<T>::Clear()
235 {
236     while (!IsEmpty())
237     {
238         PopBack();
239     }
240 }
241
242 template<typename T>
243 inline size_t CList<T>::GetCount()
244 {
245     return m_nCount;
246 }
247
248 template<typename T>
249 inline bool CList<T>::IsEmpty()
250 {
251     return m_nCount == 0;

```

```

252 }
253
254 template<typename T>
255 inline void CList<T>::Init()
256 {
257     m_pHead = new NODE;
258     m_pTail = new NODE;
259     m_pHead->m_pNext = m_pTail;
260     m_pTail->m_pPre = m_pHead;
261     m_nCount = 0;
262 }

```

## 双向链表.cpp

```

1  // 链表.cpp
2
3  #include <iostream>
4  #include "CList.h"
5  using namespace std;
6
7  class CA
8  {
9  public:
10     CA() :m_p(nullptr) {}
11     CA(int n) :m_p(new int(n)) {}
12     CA(const CA& obj) :m_p(new int(*obj.m_p)) {}
13     CA& operator=(const CA& obj)
14     {
15         if (m_p != nullptr)
16         {
17             delete m_p;
18         }
19         m_p = new int(*obj.m_p);
20         return *this;
21     }
22     ~CA()
23     {
24         if (m_p != nullptr)
25         {
26             delete m_p;
27         }
28     }
29 private:
30     int* m_p;
31 };
32
33 CList<CA> Foo()
34 {
35     CList<CA> lst({2, 3, 4, 5, 6, 7, 8, 9});
36     return lst;
37 }
38
39 int main()
40 {
41     auto lst1 = Foo();
42
43     // 14 7 3 5

```

```

44     CList<int> lst;
45     lst.PushBack(3);
46     lst.PushBack(5);
47     lst.PushHead(7);
48     lst.PushHead(14);
49
50     auto pNode = lst.Find(7);
51     lst.Insert(pNode, 13);
52
53     pNode = lst.Find(66);
54
55     lst.Delete(lst.Find(5));
56     lst.Delete(lst.Find(99));
57     lst.PopHead();
58     lst.PopBack();
59
60     return 0;
61 }

```

## 迭代器

### CList.hpp

```

1  #pragma once
2  #include <utility>
3  #include <assert.h>
4
5  // 双向链表
6  template<typename T>
7  class CList
8  {
9  public:
10     struct NODE
11     {
12         NODE() :m_pPre(nullptr), m_pNext(nullptr), m_val{} {}
13         NODE(const T& val) :m_pPre(nullptr), m_pNext(nullptr), m_val(val) {}
14         T m_val; // 存储数据
15         NODE* m_pPre; // 前驱节点的指针
16         NODE* m_pNext; // 后继节点的指针
17     };
18     using PNODE = NODE*;
19
20     class CIterator
21     {
22     public:
23         CIterator(PNODE pCur, PNODE pHead, PNODE pTail) :
24             m_pCur(pCur), m_pHead(pHead), m_pTail(pTail) {}
25         CIterator& operator++() // 前++
26         {
27             // 检查,不能移动到哨兵后面
28             assert(m_pCur != m_pTail);
29             // 位置后移
30             m_pCur = m_pCur->m_pNext;
31
32             return *this;
33         }
34         bool operator!=(const CIterator& obj)

```

```

35     {
36         return m_pCur != obj.m_pCur;
37     }
38     T& operator*()
39     {
40         // 尾哨兵不能取内容
41         assert(m_pCur != m_pTail);
42         return m_pCur->m_val;
43     }
44
45     private:
46         PNODE m_pCur;
47         PNODE m_pHead; // 头哨兵,负责边界检查
48         PNODE m_pTail; // 尾哨兵,负责边界检查
49     };
50 public:
51     CList();
52     CList(const CList& obj);
53     CList(CList&& obj);
54     CList(std::initializer_list<T> il);
55     CList& operator=(const CList& obj);
56     virtual ~CList();
57
58     // 获取迭代器
59     CIterator begin();
60     CIterator end();
61
62     // 插入
63     void PushHead(const T& val);
64     void PushBack(const T& val);
65     void Insert(PNODE pNode, const T& val); // 插入到指定位置的前面
66
67     // 删除
68     void PopHead();
69     void PopBack();
70     void Delete(PNODE pNode);
71
72     // 修改
73     void Modify(PNODE pNode, const T& val);
74
75     // 查询
76     PNODE Find(const T& val);
77
78     // 清空
79     void Clear();
80
81     // 获取个数
82     size_t GetCount();
83
84     // 判断是否为空
85     bool IsEmpty();
86 private:
87     void Init(); // 置空
88 private:
89     PNODE m_pHead; // 头哨兵指针
90     PNODE m_pTail; // 尾哨兵指针
91     size_t m_nCount; // 节点的个数
92 };

```



```

93
94     template<typename T>
95     inline CList<T>::CList()
96     {
97         Init();
98     }
99
100    template<typename T>
101    inline CList<T>::CList(const CList& obj)
102    {
103        Init();
104        *this = obj;
105    }
106
107    template<typename T>
108    inline CList<T>::CList(CList&& obj)
109    {
110        Init();
111        m_pHead = obj.m_pHead;
112        m_pTail = obj.m_pTail;
113        m_nCount = obj.m_nCount;
114        obj.m_pHead = nullptr;
115        obj.m_pTail = nullptr;
116        obj.m_nCount = 0;
117    }
118
119    template<typename T>
120    inline CList<T>::CList(std::initializer_list<T> il)
121    {
122        Init();
123
124        for (auto& val : il)
125        {
126            PushBack(val);
127        }
128    }
129
130    template<typename T>
131    inline CList<T>& CList<T>::operator=(const CList& obj)
132    {
133        if (*this == obj)
134        {
135            return *this;
136        }
137        // 清空自己
138        Clear();
139
140        // 循环插入
141        PNODE pNode = obj.m_pHead->m_pNext;
142        while (pNode != obj.m_pTail) // 从头结点开始遍历,直到尾哨兵
143        {
144            PushBack(pNode->obj.m_val);
145            pNode = pNode->m_pNext;
146        }
147        return *this;
148    }
149
150    template<typename T>

```

```
151 inline CList<T>::~~CList()
152 {
153     // 元素内容清掉
154     Clear();
155     // 释放哨兵
156     if (m_pHead != nullptr)
157     {
158         delete m_pHead;
159         m_pHead = nullptr;
160     }
161     if (m_pTail != nullptr)
162     {
163         delete m_pTail;
164         m_pTail = nullptr;
165     }
166 }
167
168 template<typename T>
169 inline typename CList<T>::CIterator CList<T>::begin()
170 {
171     return CIterator(m_pHead→m_pNext, m_pHead, m_pTail);
172 }
173
174 template<typename T>
175 inline typename CList<T>::CIterator CList<T>::end()
176 {
177     return CIterator(m_pTail, m_pHead, m_pTail);
178 }
179
180 template<typename T>
181 inline void CList<T>::PushHead(const T& val)
182 {
183     // 头哨兵后面是头结点, val插到头节点前面, val就是新的头结点
184     Insert(m_pHead→m_pNext, val);
185 }
186
187 template<typename T>
188 inline void CList<T>::PushBack(const T& val)
189 {
190     Insert(m_pTail, val); // 插到尾哨兵前面
191 }
192
193 template<typename T>
194 inline void CList<T>::Insert(PNODE pNode, const T& val)
195 {
196     // 检查参数
197     if (pNode == nullptr)
198     {
199         return;
200     }
201     // 构造新结点
202     PNODE pNewNode = new NODE(val);
203     if (pNewNode == nullptr)
204     {
205         return;
206     }
207
208     // 插入新结点
```

```

209     PNODE pPre = pNode→m_pPre;
210     pPre→m_pNext = pNewNode;
211     pNewNode→m_pPre = pPre;
212     pNewNode→m_pNext = pNode;
213     pNode→m_pPre = pNewNode;
214
215     // 个数增加
216     ++m_nCount;
217 }
218
219 template<typename T>
220 inline void CList<T>::PopHead()
221 {
222     Delete(m_pHead→m_pNext);
223 }
224
225 template<typename T>
226 inline void CList<T>::PopBack()
227 {
228     Delete(m_pTail→m_pPre);
229 }
230
231 template<typename T>
232 inline void CList<T>::Delete(PNODE pNode)
233 {
234     // 检查参数
235     if (pNode == nullptr || pNode == m_pHead || pNode == m_pTail)
236     {
237         return;
238     }
239     // 拿到前驱和后继
240     PNODE pPre = pNode→m_pPre;
241     PNODE pNext = pNode→m_pNext;
242
243     pPre→m_pNext = pNext;
244     pNext→m_pPre = pPre;
245
246     // 删除
247     delete pNode;
248
249     // 个数减少
250     --m_nCount;
251 }
252
253 template<typename T>
254 inline void CList<T>::Modify(PNODE pNode, const T& val)
255 {
256     // 检查参数
257     if (pNode == nullptr || pNode == m_pHead || pNode == m_pTail)
258     {
259         return;
260     }
261     pNode→m_val = val;
262 }
263
264 template<typename T>
265 inline typename CList<T>::PNODE CList<T>::Find(const T& val)
266 {

```

```

267     // 保存头结点
268     PNODE pNode = m_pHead->m_pNext;
269     while (pNode != m_pTail) // 从头结点开始遍历,直到尾哨兵
270     {
271         if (pNode->m_val == val)
272         {
273             return pNode;
274         }
275         pNode = pNode->m_pNext;
276     }
277     return nullptr;
278 }
279
280 template<typename T>
281 inline void CList<T>::Clear()
282 {
283     while (!IsEmpty())
284     {
285         PopBack();
286     }
287 }
288
289 template<typename T>
290 inline size_t CList<T>::GetCount()
291 {
292     return m_nCount;
293 }
294
295 template<typename T>
296 inline bool CList<T>::IsEmpty()
297 {
298     return m_nCount == 0;
299 }
300
301 template<typename T>
302 inline void CList<T>::Init()
303 {
304     m_pHead = new NODE;
305     m_pTail = new NODE;
306     m_pHead->m_pNext = m_pTail;
307     m_pTail->m_pPre = m_pHead;
308     m_nCount = 0;
309 }

```

## 迭代器.cpp

```

1  // 迭代器.cpp
2  #include <iostream>
3  #include <list>
4  #include "CList.hpp"
5  using namespace std;
6
7  int main()
8  {
9      #if 0
10         list<int> lst({ 1, 4, 5, 6, 7, 8, 9 });
11         list<int>::iterator itr = lst.begin();

```

