

定义
分类
单向链表
代码实现
优缺点说明
双向链表
代码实现
优缺点说明
循环链表
代码实现
代码实现
优缺点
有序链表
代码实现

定义

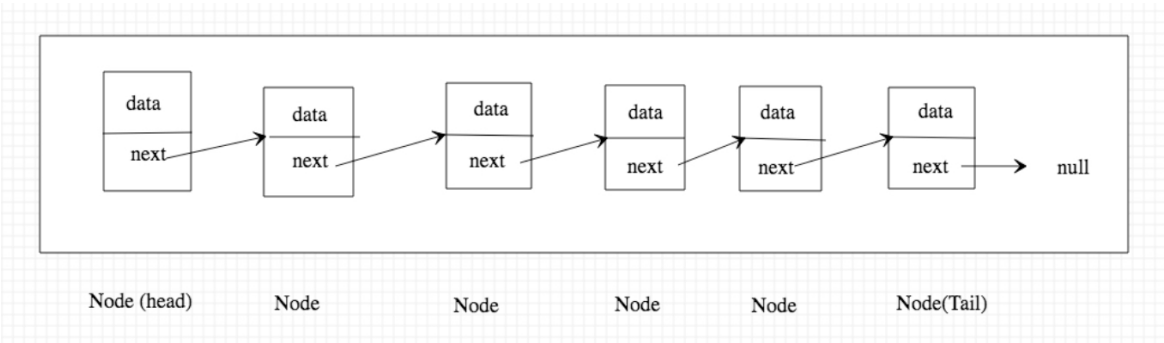
线性表的数据元素可以存储在随意的存储单元，每一个节点不仅仅包括数据元素还有指向下一个节点的指针

分类

链式线性表在存储结构上可以分为单向链表、双向链表和循环链表

链式线性表在顺序上也可以分为有序的和无序的

单向链表



单链表是链表中结构最简单的。一个单链表的节点(Node)分为两个部分，第一个部分(data)保存节点的信息，另一个部分存储下一个节点，分指向空值

代码实现

```
1 package day1.单链表;
2
3 public class SingleLink<E> {
4
5     private int size = 0;          //链表节点数量(包含头结点)
6     private Node head;             //链表的头结点
7
8     static class Node<E>{
9         private E data;           // 节点数据
10        private Node next;        // 下一个节点
11
12        public Node(E e){
13            this.data = e;
14        }
15
16        @Override
17        public String toString() {
18            return data.toString();
19        }
20    }
21
22    /**
23     * 构造函数, 初始化 头结点
24     */
25    public SingleLink(){
26        Node<E> node = new Node<>(null);
27        node.next = null;
28        head = node;
29        size++;
30    }
31
32    /**
33     * 根据节点 数据 获取在链表的下标值 (从零开始)
34     * @param e
35     * @return
36     */
37    public int find(E e){
38        if (e == null){
39            throw new NullPointerException("数据不能为空");
40        }
41        Node tmp = head;
42        int cnt = 0;
43        while (tmp.next != null){
44            tmp = tmp.next;
45            cnt++;
46            if (e.equals(tmp.data)){
47                return cnt;
48            }
49        }
50        return -1;
51    }
52}
```

```

53  /**
54   * 根据下标获取链表的元素
55   * @param index
56   * @return
57   */
58  public Node get(int index){
59      if (index < 0 || index >= size){
60          throw new IndexOutOfBoundsException("元素位置不合法");
61      }
62      Node tmp = head;
63      int cnt = 0;
64      while (tmp.next != null){
65          tmp = tmp.next;
66          cnt++;
67          if (cnt == index){
68              return tmp;
69          }
70      }
71      return null;
72  }
73
74  /**
75   * addFirst
76   * @param e
77   */
78  public void addFirst(E e){
79      Node node = new Node(e);
80      node.next = head.next;
81      head.next = node;
82      size++;
83  }
84
85  /**
86   * addLast
87   * @param e
88   */
89  public void addLast(E e){
90      Node tmp = head;
91      while (tmp.next != null){
92          tmp = tmp.next;
93      }
94      tmp.next = new Node(e);
95      size++;
96  }
97
98  /**
99   * 根据下标移除
100   * @param index
101   */
102  public void remove(int index){
103      if (index < 1 || index >= size){
104          throw new IndexOutOfBoundsException("元素位置不合法");
105      }
106      Node curr = get(index);

```

```

107         if (index == 1){// 移除第一个节点
108             head.next = curr.next;
109         }else {
110             Node prev = get(index - 1);
111             prev.next = curr.next;
112         }
113         size--;
114     }
115
116     /**
117     * 根据节点的 值 移除该节点
118     * @param e
119     * @return
120     */
121     public boolean remove(E e){
122         int index = find(e);
123         if (index == -1){
124             return false;
125         }
126         remove(index);
127         return true;
128     }
129
130     public void print(){
131         Node tmp = head;
132         StringBuffer buffer = new StringBuffer();
133         buffer.append(" size is : " + size + " , 遍历链表 : head --> ");
134         while (tmp.next != null){
135             tmp = tmp.next;
136             buffer.append(tmp.toString() + " --> ");
137         }
138         System.out.println(buffer.toString());
139     }
140
141
142     public static void main(String[] args) {
143         System.out.println("初始化链表");
144         SingleLink<String> link = new SingleLink<>();
145         link.print();
146
147         System.out.println("addFirst a ");
148         link.addFirst("a");
149         link.print();
150
151         System.out.println("addFirst b ");
152         link.addFirst("b");
153         link.print();
154
155         System.out.println("addLast c ");
156         link.addLast("c");
157         link.print();
158
159         System.out.println("查找节点 a 在第链表的下标为 : " + link.find("a"));
160         System.out.println("查找节点 x 在第链表的下标为 : " + link.find("x"));

```

```

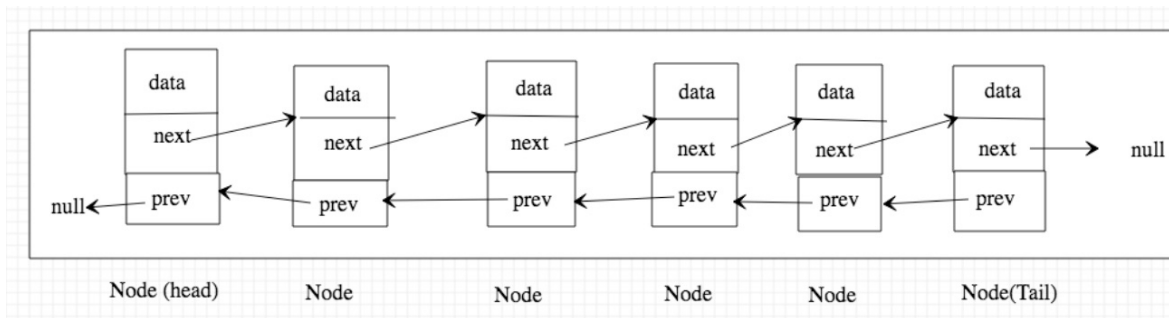
161
162
163     System.out.println("获取下标为的 1 个节点 : " + link.get(1));
164
165     System.out.println("移除下标为 2 个节点");
166     link.remove(2);
167     link.print();
168
169     System.out.println("移除 c ");
170     link.remove("c");
171     link.print();
172     System.out.println("移除 b ");
173     link.remove("b");
174     link.print();
175 }
176
177 }
178
179 /*输出
180
181 初始化链表
182 size is : 1 , 遍历链表 : head -->
183 addFirst a
184 size is : 2 , 遍历链表 : head --> a -->
185 addFirst b
186 size is : 3 , 遍历链表 : head --> b --> a -->
187 addLast c
188 size is : 4 , 遍历链表 : head --> b --> a --> c -->
189 查找节点 a 在第链表的下标为 : 2
190 查找节点 x 在第链表的下标为 : -1
191 获取下标为的 1 个节点 : b
192 移除下标为 2 个节点
193 size is : 3 , 遍历链表 : head --> b --> c -->
194 移除 c
195 size is : 2 , 遍历链表 : head --> b -->
196 移除 b
197 size is : 1 , 遍历链表 : head -->
198
199 */

```

优缺点说明

1. 插入到头部速度快，移动指针即可
2. 插入到尾部速度慢，需要从头部开始遍历查找整张链表
3. 删除头部比较快，移动指针即可
4. 删除其他节点就要进行遍历
5. 查询需要进行遍历

双向链表



单链表与双向链表的区别在于，链表中的节点多了一个指针，不仅仅有指向下一个节点的指针，还有一个指向上一个节点的指针，头结双向链表的好处就是循环方向多了一条，可以从前往后，也可以从后往前。

代码实现

```

1 package day1.双向链表;
2
3 public class DBLink<E> {
4     private int size;        // 节点数量 包含头结点 和 尾节点
5     private Node head;       // 头结点
6     private Node tail;       // 尾节点
7
8     static class Node<E>{
9         private E data;      // 节点数据
10        private Node next;    // 下一个节点
11        private Node prev;    // 上一个节点
12
13        public Node(E e){
14            this.data = e;
15        }
16
17        @Override
18        public String toString() {
19            return data.toString();
20        }
21    }
22
23    /**
24     * 构造函数，初始化 头结点、尾节点
25     */
26    public DBLink(){
27        head = new Node(null);
28        tail = new Node(null);
29        head.next = tail;
30        tail.prev = head;
31        size = 2;
32    }
33
34    /**
35     * 根据节点 数据 获取在链表的下标值
36     * @param e
37     * @return
38     */
39    public int find(E e){

```

```

40         if (e == null){
41             throw new NullPointerException("数据不能为空");
42         }
43         Node tmp = head;
44         int cnt = 0;
45         while (tmp.next != null){
46             tmp = tmp.next;
47             cnt++;
48             if (e.equals(tmp.data)){
49                 return cnt;
50             }
51         }
52         return -1;
53     }
54
55     /**
56      * 根据下标获取链表中的元素
57      * @param index
58      * @return
59      */
60     public Node get(int index){
61         if (index < 0 || index >= size){
62             throw new IndexOutOfBoundsException("元素下标不合法");
63         }
64
65         // 根据 index 判断 从头开始找，还是从尾部 开始找更快
66         if (index * 2 <= size){
67             Node tmp = head;
68             int cnt = 0;
69             while (tmp.next != null){
70                 tmp = tmp.next;
71                 cnt++;
72                 if (cnt == index){
73                     return tmp;
74                 }
75             }
76         }else {
77             Node tmp = tail;
78             int cnt = 0;
79             while (tmp.prev != null){
80                 tmp = tmp.prev;
81                 cnt++;
82                 if (cnt == size - index - 1){
83                     return tmp;
84                 }
85             }
86         }
87         return null;
88     }
89
90     /**
91      * addFirst
92      */
93     public void addFirst(E e){

```

```

94     Node node = new Node(e);
95     Node next = head.next;
96
97     node.next = next;
98     next.prev = node;
99     head.next = node;
100    node.prev = head;
101    size++;
102 }
103
104 /**
105  * addLast
106  * @param e
107  */
108 public void addLast(E e){
109     Node node = new Node(e);
110     Node prev = tail.prev;
111
112     node.prev = prev;
113     prev.next = node;
114     tail.prev = node;
115     node.next = tail;
116     size++;
117 }
118
119 /**
120  * 根据元素的值删除
121  * @param e
122  */
123 public boolean remove(E e){
124     int index = find(e);
125     if (index == -1){
126         return false;
127     }
128     remove(index);
129     return true;
130 }
131
132 /**
133  * 根据 元素 下标 删除元素
134  * @param index
135  */
136 public void remove(int index){
137     if (index < 1 || index >= size - 1){
138         throw new IndexOutOfBoundsException("元素下标不合法");
139     }
140     Node curr = get(index);
141     Node prev = curr.prev;
142     Node next = curr.next;
143     prev.next = next;
144     next.prev = prev;
145     size--;
146 }
147

```



```

148 public void print(){
149     Node tmp = head;
150     StringBuffer buffer = new StringBuffer();
151     buffer.append(" size is : " + size + " , 正向遍历链表 : head --> ");
152     while (tmp.next != null && !tmp.next.equals(tail)){
153         tmp = tmp.next;
154         buffer.append(tmp.toString() + " --> ");
155     }
156     buffer.append(" tail ");
157     System.out.print(buffer.toString());
158
159     tmp = tail;
160     buffer = new StringBuffer();
161     buffer.append(" 反向遍历链表 : tail -->");
162     while (tmp.prev != null && !tmp.prev.equals(head)){
163         tmp = tmp.prev;
164         buffer.append(tmp.toString() + " <-- ");
165     }
166     buffer.append(" head ");
167     System.out.print(buffer.toString());
168     System.out.println();
169 }
170
171 public static void main(String[] args) {
172
173     DBLink<String> link = new DBLink<>();
174     System.out.println("构造完成");
175     link.print();
176     System.out.println("addFirst a ");
177     link.addFirst("a");
178     System.out.println("addFirst b ");
179     link.addFirst("b");
180     System.out.println("addFirst c ");
181     link.addFirst("c");
182     link.print();
183
184     System.out.println("addLast 1 ");
185     link.addLast("1");
186     System.out.println("addLast 2 ");
187     link.addLast("2");
188     System.out.println("addLast 3 ");
189     link.addLast("3");
190     link.print();
191
192     System.out.println("查找 a 在第 " + link.find("a") + " 个");
193     System.out.println("查找 x 在第 " + link.find("x") + " 个");
194
195     System.out.println("获取下标为2的元素 " + link.get(2));
196
197
198     System.out.println("移除下标为2的元素");
199     link.remove(2);
200     link.print();
201

```

```

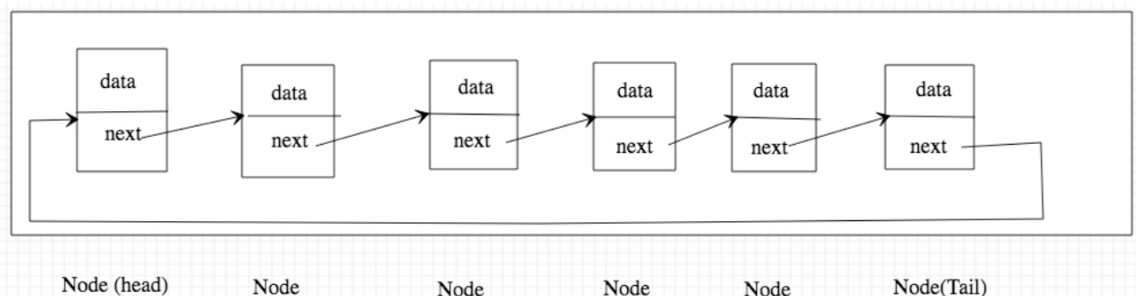
202     System.out.println("移除 元素 a");
203     link.remove("a");
204     link.print();
205 }
206 }
207
208 /* 输出
209
210 构造完成
211 size is : 2 , 正向遍历链表 : head --> tail 反向遍历链表 : tail --> head
212 addFirst a
213 addFirst b
214 addFirst c
215 size is : 5 , 正向遍历链表 : head --> c --> b --> a --> tail 反向遍历链表 : tail --> a <-- b <-- c
216 addLast 1
217 addLast 2
218 addLast 3
219 size is : 8 , 正向遍历链表 : head --> c --> b --> a --> 1 --> 2 --> 3 --> tail 反向遍历链表 : tail --> 3 <-- 2 <-- 1 <-- a <-- b <-- c
220 查找 a 在第 3 个
221 查找 x 在第 -1 个
222 获取下标为2的元素 b
223 移除下标为2的元素
224 size is : 7 , 正向遍历链表 : head --> c --> a --> 1 --> 2 --> 3 --> tail 反向遍历链表 : tail --> 3 <-- 2 <-- 1 <-- a <-- c
225 移除 元素 a
226 size is : 6 , 正向遍历链表 : head --> c --> 1 --> 2 --> 3 --> tail 反向遍历链表 : tail --> 3 <-- 2 <-- 1 <-- c
227
228 */

```

优缺点说明

1. 头部插入和尾部插入都较快，只需要移动指针即可
2. 根据查询位置查询速度快于单链表，通过位置与链表长度比较，决定从头部还是尾部开始遍历
3. 根据元素的值查询需要遍历表
4. 根据位置删除速度快于单链表
5. 根据元素的值删除需要遍历表
6. 多一个指针，多占用一点空间

循环链表



循环链表就是在单链表的基础上，将尾节点的指针指向头结点。这样的好处是可以通过任意节点出发，访问到链表的全部节点

代码实现

```

1 package day1.循环链表;
2
3 import javax.sound.midi.Soundbank;
4
5 public class LoopLink<E> {
6
7     private int size = 0;        //链表节点数量(包含头结点和 尾节点)
8     private Node head;           //链表的头结点
9     private Node tail;           //链表的尾节点
10
11     static class Node<E>{
12         private E data;          // 节点数据
13         private Node next;       // 下一个节点
14
15         public Node(E e){
16             this.data = e;
17         }
18
19         @Override
20         public String toString() {
21             return data.toString();
22         }
23     }
24
25     public LoopLink(){
26         head = new Node(null);
27         tail = new Node(null);
28         head.next = tail;
29         tail.next = head;
30         size = 2;
31     }
32
33     /**
34      * 根据节点 数据 获取在链表的下标(从0开始)
35      * @param e
36      * @return
37      */
38     public int find(E e){
39         if (e == null){
40             throw new NullPointerException("数据不能为空");
41         }
42         Node tmp = head;
43         int _index = -1;
44         int cnt = 0;
45         while (tmp.next != null && !tmp.next.equals(head)){
46             tmp = tmp.next;
47             cnt++;
48             if (e.equals(tmp.data)){
49                 return cnt;
50             }
51         }
52         return -1;
53     }
54

```

```

55     /**
56      * 根据下表获取链表的元素
57      * @param index
58      * @return
59      */
60     public Node get(int index){
61         if (index < 0 || index >= size){
62             throw new IndexOutOfBoundsException("元素位置不合法");
63         }
64         Node tmp = head;
65         int cnt = 0;
66         while (tmp.next != null && !tmp.next.equals(tail)){
67             tmp = tmp.next;
68             cnt++;
69             if (cnt == index){
70                 return tmp;
71             }
72         }
73         return null;
74     }
75
76
77     /**
78      * addFirst
79      */
80     public void addFirst(E e){
81         Node node = new Node(e);
82         Node next = head.next;
83         node.next = next;
84         head.next = node;
85         size++;
86     }
87
88     /**
89      * 根据下标移除
90      * @param index
91      */
92     public void remove(int index){
93         if (index < 1 || index >= size - 1){
94             throw new IndexOutOfBoundsException("元素位置不合法");
95         }
96         Node curr = get(index);
97         if (index == 1){// 移除第一个节点
98             head.next = curr.next;
99         }else {
100             Node prev = get(index - 1);
101             prev.next = curr.next;
102         }
103         size--;
104     }
105
106     /**
107      * 根据节点的 值 移除该节点
108      * @param e

```

```

109     * @return
110     */
111     public boolean remove(E e){
112         int index = find(e);
113         if (index == -1){
114             return false;
115         }
116         remove(index);
117         return true;
118     }
119
120     public void print(){
121         Node tmp = head;
122         StringBuffer buffer = new StringBuffer();
123         buffer.append(" size is : " + size + " , 遍历链表 : head -->");
124         while (tmp.next != null && !tmp.next.equals(tail)){
125             tmp = tmp.next;
126             buffer.append(tmp.toString() + " --> ");
127         }
128         buffer.append(" tail ");
129         System.out.println(buffer.toString());
130     }
131
132
133     public static void main(String[] args) {
134
135         LoopLink<String> loopLink = new LoopLink();
136         System.out.println("构造");
137         loopLink.print();
138
139         System.out.println("addFirst a");
140         loopLink.addFirst("a");
141         loopLink.print();
142
143         System.out.println("移除下标为1的元素");
144         loopLink.remove(1);
145         loopLink.print();
146
147
148         System.out.println("addFirst b1");
149         loopLink.addFirst("b1");
150         System.out.println("addFirst b2");
151         loopLink.addFirst("b2");
152         System.out.println("addFirst b3");
153         loopLink.addFirst("b3");
154         loopLink.print();
155
156
157         System.out.println("获取下标为2 : " + loopLink.get(2));
158         System.out.println("查询 b2 的下标 : " + loopLink.find("b2"));
159
160         System.out.println("移除b2");
161         loopLink.remove("b2");
162         loopLink.print();
163

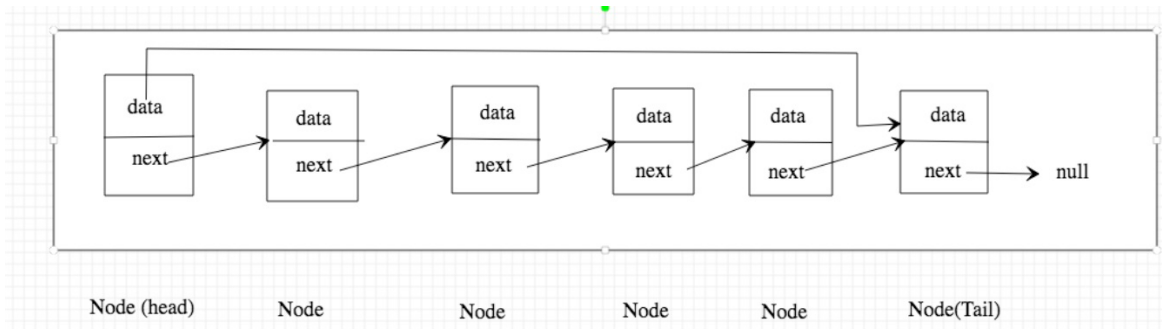
```

```

164     System.out.println("移除下标为1");
165     loopLink.remove(2);
166     loopLink.print();
167 }
168 }
169
170 /* 输出
171
172 构造
173 size is : 2 , 遍历链表 : head --> tail
174 addFirst a
175 size is : 3 , 遍历链表 : head --> a --> tail
176 移除下标为1的元素
177 size is : 2 , 遍历链表 : head --> tail
178 addFirst b1
179 addFirst b2
180 addFirst b3
181 size is : 5 , 遍历链表 : head --> b3 --> b2 --> b1 --> tail
182 获取下标为2 : b2
183 查询 b2 的下标 : 2
184 移除b2
185 size is : 4 , 遍历链表 : head --> b3 --> b1 --> tail
186 移除下标为1
187 size is : 3 , 遍历链表 : head --> b3 --> tail
188
189 */

```

还有一种变式如下图，头节点的data本身为空，利用该段空间，存储一个指针，指向尾部，这样的好处是找到尾节点不用从头循环查找



代码实现

```

1 package day1.循环链表;
2
3 public class LoopLinkExt<E> {
4
5     private int size = 0; //链表节点数量(包含头结点)
6     private Node head; //链表的头结点
7
8     static class Node<E>{
9         private E data; // 节点数据
10        private Node next; // 下一个节点
11    }

```

```

12     public Node(E e){
13         this.data = e;
14     }
15
16     @Override
17     public String toString() {
18         return data.toString();
19     }
20 }
21
22 /**
23  * 构造函数，初始化 头结点
24  */
25 public LoopLinkExt(){
26     Node<E> node = new Node<>(null);
27     node.next = null;
28     head = node;
29     size++;
30 }
31
32 /**
33  * 根据节点 数据 获取在链表的下标值（从零开始）
34  * @param e
35  * @return
36  */
37 public int find(E e){
38     if (e == null){
39         throw new NullPointerException("数据不能为空");
40     }
41     Node tmp = head;
42     int cnt = 0;
43     while (tmp.next != null){
44         tmp = tmp.next;
45         cnt++;
46         if (e.equals(tmp.data)){
47             return cnt;
48         }
49     }
50     return -1;
51 }
52
53 /**
54  * 根据下标获取链表的元素
55  * @param index
56  * @return
57  */
58 public Node get(int index){
59     if (index < 0 || index >= size){
60         throw new IndexOutOfBoundsException("元素位置不合法");
61     }
62     Node tmp = head;
63     int cnt = 0;
64     while (tmp.next != null){
65         tmp = tmp.next;

```

```

66         cnt++;
67         if (cnt == index){
68             return tmp;
69         }
70     }
71     return null;
72 }
73
74 /**
75  * addFirst
76  * @param e
77  */
78 public void addFirst(E e){
79     if (size == 1){ // 长度等于 1，只有头结点，加入在头结点后面，从头结点和尾节点是一样的，效率也一样
80         addLast(e);
81     }else {
82         Node node = new Node(e);
83         node.next = head.next;
84         head.next = node;
85         size++;
86     }
87 }
88
89 /**
90  * addLast
91  * @param e
92  */
93 public void addLast(E e){
94     Node curr = new Node(e);
95     if (size == 1){
96         head.next = curr;
97         head.data = curr;
98     }else {
99         Node tail = (Node) head.data;
100         tail.next = curr;
101         head.data = curr;
102     }
103     size++;
104 }
105
106 /**
107  * 根据下标移除
108  * @param index
109  */
110 public void remove(int index){
111     if (index < 1 || index >= size){
112         throw new IndexOutOfBoundsException("元素位置不合法");
113     }
114     Node curr = get(index);
115     if (index == 1){ // 移除第一个节点
116         head.next = curr.next;
117     }else {
118         Node prev = get(index - 1);
119         prev.next = curr.next;

```



```

120     }
121     size--;
122 }
123
124 /**
125  * 根据节点的 值 移除该节点
126  * @param e
127  * @return
128  */
129 public boolean remove(E e){
130     int index = find(e);
131     if (index == -1){
132         return false;
133     }
134     remove(index);
135     return true;
136 }
137
138 public void print(){
139     Node tmp = head;
140     StringBuffer buffer = new StringBuffer();
141     buffer.append(" size is : " + size + " , 遍历链表 : head --> ");
142     while (tmp.next != null){
143         tmp = tmp.next;
144         buffer.append(tmp.toString() + " --> ");
145     }
146     System.out.println(buffer.toString());
147 }
148
149
150 public static void main(String[] args) {
151     System.out.println("初始化链表");
152     LoopLinkExt<String> link = new LoopLinkExt<>();
153     link.print();
154
155     System.out.println("addFirst a ");
156     link.addFirst("a");
157     link.print();
158
159     System.out.println("addFirst b ");
160     link.addFirst("b");
161     link.print();
162
163     System.out.println("addLast c ");
164     link.addLast("c");
165     link.print();
166
167     System.out.println("查找节点 a 在第链表的下标为 : " + link.find("a"));
168     System.out.println("查找节点 x 在第链表的下标为 : " + link.find("x"));
169
170
171     System.out.println("获取下标为的 1 个节点 : " + link.get(1));
172
173     System.out.println("移除下标为 2 个节点");

```

```

174         link.remove(2);
175         link.print();
176
177         System.out.println("移除 c ");
178         link.remove("c");
179         link.print();
180         System.out.println("移除 b ");
181         link.remove("b");
182         link.print();
183     }
184 }

```

优缺点

实际上这种实现与单链表一致，不过插入尾节点做了优化，不用循环到尾部节点

有序链表

代码实现

```

1 package day1.有序链表;
2
3 import java.util.Comparator;
4
5 public class SortLink<E> {
6
7     private int size = 0;           //链表节点数量(包含头结点)
8     private Node head;              //链表的头结点
9     private Comparator<E> comparator;
10
11     static class Node<E>{
12         private E data;             // 节点数据
13         private Node next;          // 下一个节点
14
15         public Node(E e){
16             this.data = e;
17         }
18
19         @Override
20         public String toString() {
21             return data.toString();
22         }
23     }
24
25     public void init(){
26         Node<E> node = new Node<>(null);
27         node.next = null;
28         head = node;
29         size++;
30     }
31
32     public SortLink(){
33         init();
34         comparator = new Comparator<E>() {

```

```

35         @Override
36         public int compare(E o1, E o2) {
37             if (o1.hashCode() > o2.hashCode()){
38                 return 1;
39             }else if (o1.hashCode() < o2.hashCode()){
40                 return -1;
41             }
42             return 0;
43         }
44     };
45 }
46
47 public SortLink(Comparator<E> comparator){
48     init();
49     this.comparator = comparator;
50 }
51
52 /**
53  * 根据节点 数据 获取在链表的下标值 (从零开始)
54  * @param e
55  * @return
56  */
57 public int find(E e){
58     if (e == null){
59         throw new NullPointerException("数据不能为空");
60     }
61     Node tmp = head;
62     int cnt = 0;
63     while (tmp.next != null){
64         tmp = tmp.next;
65         cnt++;
66         if (e.equals(tmp.data)){
67             return cnt;
68         }
69     }
70     return -1;
71 }
72
73 /**
74  * 根据下标获取链表的元素
75  * @param index
76  * @return
77  */
78 public Node get(int index){
79     if (index < 0 || index >= size){
80         throw new IndexOutOfBoundsException("元素位置不合法");
81     }
82     Node tmp = head;
83     int cnt = 0;
84     while (tmp.next != null){
85         tmp = tmp.next;
86         cnt++;
87         if (cnt == index){
88             return tmp;

```

```

89         }
90     }
91     return null;
92 }
93
94 /**
95  * add
96  * @param e
97  */
98 public void add(E e){
99     if(e == null){
100         throw new NullPointerException("元素值不能为空");
101     }
102     Node node = new Node(e);
103     if (size == 1){
104         head.next = node;
105     }else {
106         Node tmp = head;
107         int _index = -1;
108         int cnt = 0;
109         while (tmp.next != null){
110             cnt++;
111             tmp = tmp.next;
112             int cmp = comparator.compare((E) tmp.data, (E) node.data);
113             if (cmp > 0){
114                 _index = cnt;
115                 break;
116             }
117         }
118         if (_index == -1){// 加到末尾
119             Node last = get(size - 1);
120             last.next = node;
121         }else {
122             Node prev = get(cnt - 1);
123             Node next = prev.next;
124             node.next = next;
125             prev.next = node;
126         }
127     }
128     size++;
129 }
130
131 /**
132  * 根据下标移除
133  * @param index
134  */
135 public void remove(int index){
136     if (index < 1 || index >= size){
137         throw new IndexOutOfBoundsException("元素位置不合法");
138     }
139     Node curr = get(index);
140     if (index == 1){// 移除第一个节点
141         head.next = curr.next;
142     }else {

```

```

143         Node prev = get(index - 1);
144         prev.next = curr.next;
145     }
146     size--;
147 }
148
149 /**
150  * 根据节点的 值 移除该节点
151  * @param e
152  * @return
153  */
154 public boolean remove(E e){
155     int index = find(e);
156     if (index == -1){
157         return false;
158     }
159     remove(index);
160     return true;
161 }
162
163 public void print(){
164     Node tmp = head;
165     StringBuffer buffer = new StringBuffer();
166     buffer.append(" size is : " + size + " , 遍历链表 : head --> ");
167     while (tmp.next != null){
168         tmp = tmp.next;
169         buffer.append(tmp.toString() + " --> ");
170     }
171     System.out.println(buffer.toString());
172 }
173
174
175 public static void main(String[] args) {
176     System.out.println("构造实例");
177     SortLink<String> sortLink = new SortLink<>();
178     sortLink.print();
179     System.out.println("增加 a ");
180     sortLink.add("a");
181     sortLink.print();
182
183     System.out.println("增加 c ");
184     sortLink.add("c");
185     sortLink.print();
186
187     System.out.println("增加 e ");
188     sortLink.add("e");
189     sortLink.print();
190
191     System.out.println("增加 d ");
192     sortLink.add("d");
193     sortLink.print();
194
195     System.out.println("增加 f ");
196     sortLink.add("f");

```

```

197         sortLink.print();
198
199         System.out.println("增加 b ");
200         sortLink.add("b");
201         sortLink.print();
202
203         System.out.println("查找节点 a 在第链表的下标为 : " + sortLink.find("a"));
204         System.out.println("查找节点 x 在第链表的下标为 : " + sortLink.find("x"));
205
206
207         System.out.println("获取下标为的 1 个节点 : " + sortLink.get(1));
208
209         System.out.println("移除下标为 2 个节点");
210         sortLink.remove(2);
211         sortLink.print();
212
213         System.out.println("移除 c ");
214         sortLink.remove("c");
215         sortLink.print();
216         System.out.println("移除 b ");
217         sortLink.remove("b");
218         sortLink.print();
219     }
220 }
221
222 /*
223
224 构造实例
225 size is : 1 , 遍历链表 : head -->
226 增加 a
227 size is : 2 , 遍历链表 : head --> a -->
228 增加 c
229 size is : 3 , 遍历链表 : head --> a --> c -->
230 增加 e
231 size is : 4 , 遍历链表 : head --> a --> c --> e -->
232 增加 d
233 size is : 5 , 遍历链表 : head --> a --> c --> d --> e -->
234 增加 f
235 size is : 6 , 遍历链表 : head --> a --> c --> d --> e --> f -->
236 增加 b
237 size is : 7 , 遍历链表 : head --> a --> b --> c --> d --> e --> f -->
238 查找节点 a 在第链表的下标为 : 1
239 查找节点 x 在第链表的下标为 : -1
240 获取下标为的 1 个节点 : a
241 移除下标为 2 个节点
242 size is : 6 , 遍历链表 : head --> a --> c --> d --> e --> f -->
243 移除 c
244 size is : 5 , 遍历链表 : head --> a --> d --> e --> f -->
245 移除 b
246 size is : 5 , 遍历链表 : head --> a --> d --> e --> f -->
247
248 */

```

