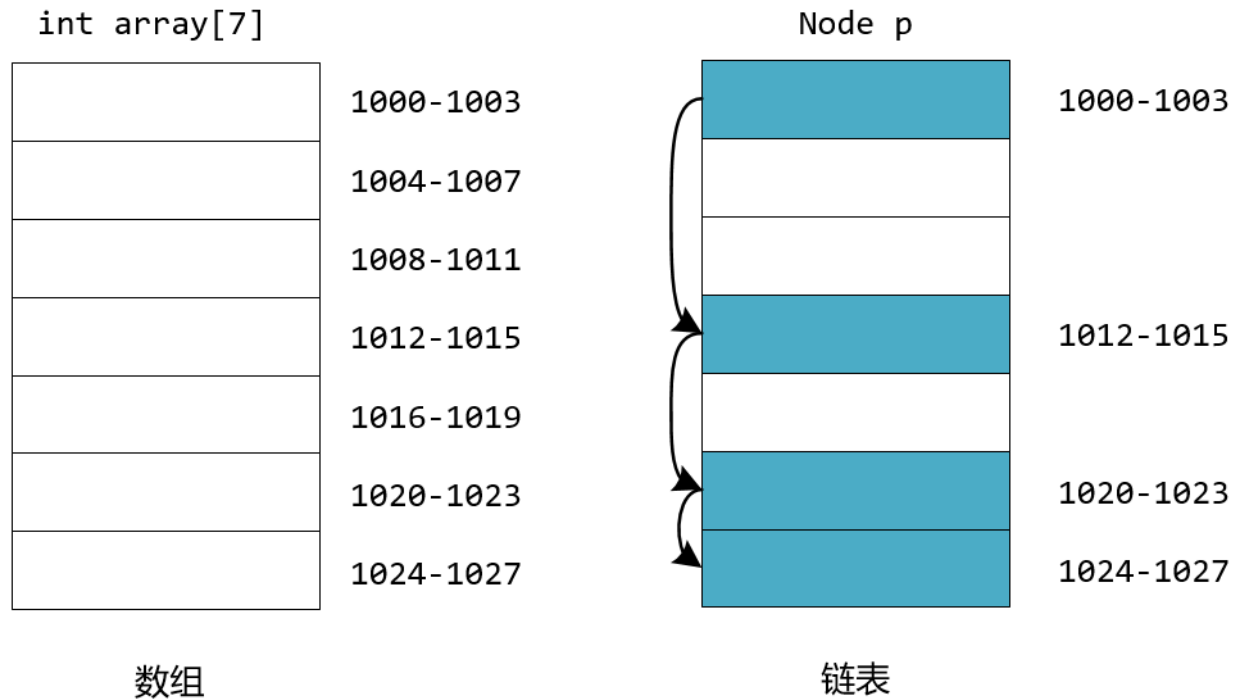


2018年10月25日

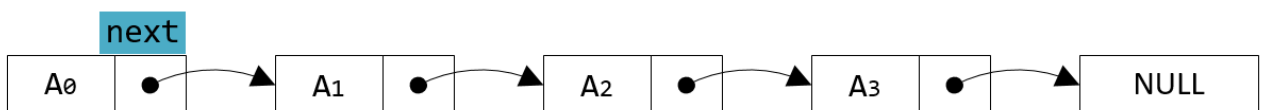
由于数组需要一块连续的内存空间，所以插入和删除时会使部分元素挪动，其时间复杂度为 $O(N)$ ，为了避免这种开销，可以使用链表这种不连续内存的数据结构。数组和链表的内存分布：



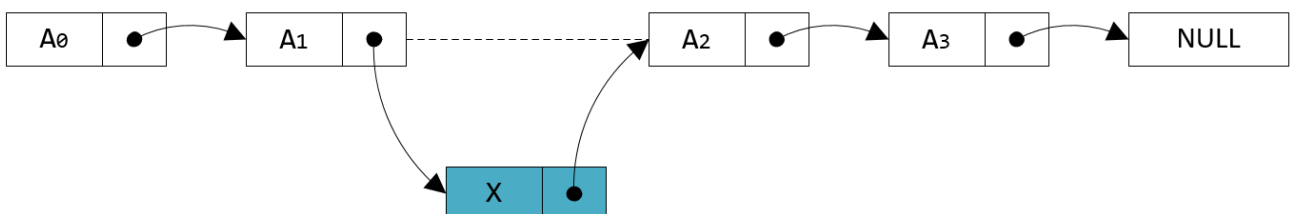
假如内存剩余可用空间大于100MB，但是其可用空间不是连续的，所以申请100MB数组时会失败，而链表由于不需要连续的内存，其通过指针将一组零散的内存块串起来，所以此时使用链表就不会有问题。

1, 单链表

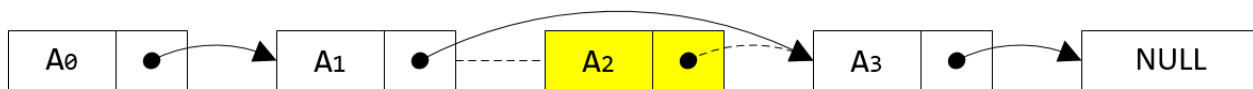
链表是由一系列的节点组成，对于单链表而言，每个节点除了存储元素外，还要存储指向下一个节点的地址，称之为 `next`；最后一个节点的 `next` 指向 `NULL`：



向链表中插入元素，如向 A_1 与 A_2 之间插入节点 x ，首先将 A_1 的 `next` 指向 x ，然后将 x 指向 A_2 ：

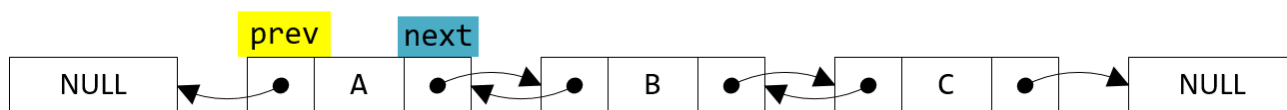


删除链表中的元素，如删除 A_2 ，那么就将 A_1 的 next 指向 A_3 ：



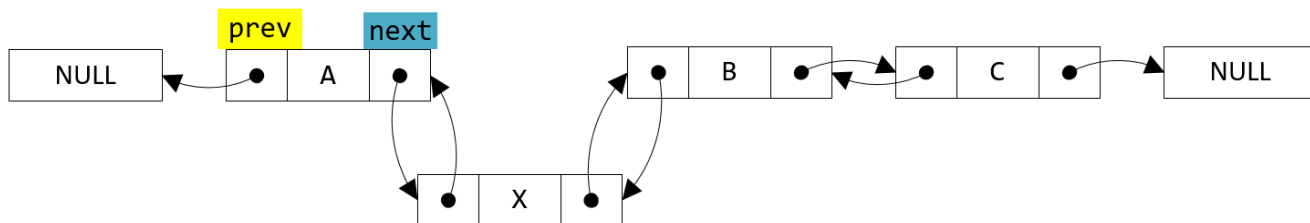
由此可见，链表的插入和删除操作，只需考虑相邻节点指针的改变，因此其时间复杂度均为 $O(1)$ 。但是，在链表中随机访问某个位置的元素就不如数组高效了，只能从头开始遍历，直至找到所需的节点，因此其随机访问的时间复杂度为 $O(N)$ 。

2，双向链表

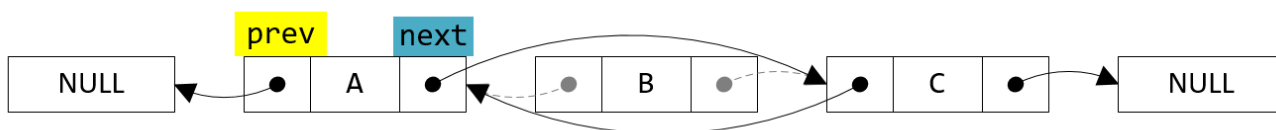


如上图，双向链表比单链表多了一个前向指针 prev，prev 指向前一个节点，next 指向后一个节点。

向链表中插入元素，如在 A 节点与 B 节点之间插入节点 X，首先将 A 节点的 next 指向 X 节点，X 节点的 prev 指向 A 节点；然后将 X 节点的 next 指向 B 节点，B 节点的 prev 指向 X：

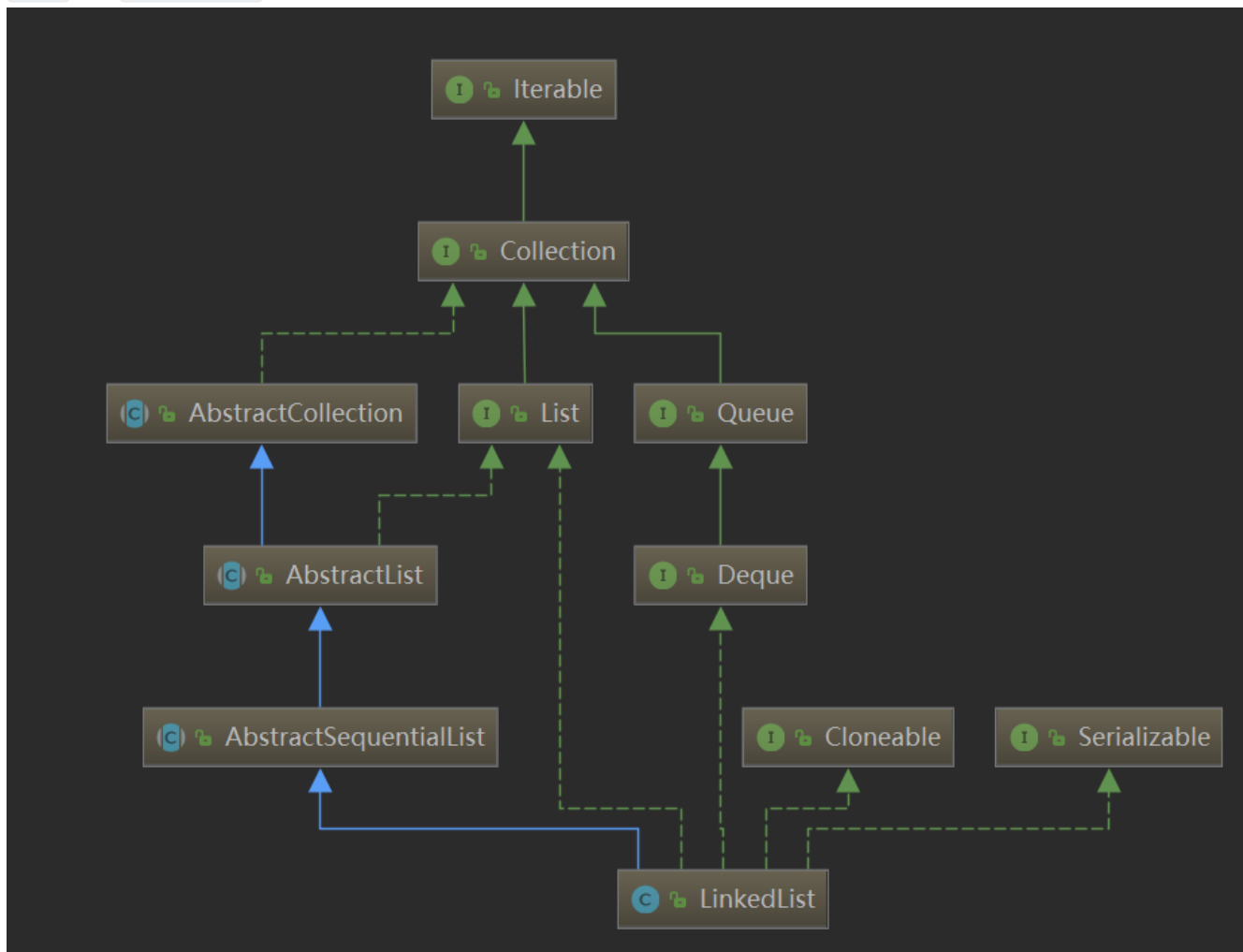


删除链表中的元素，如删除元素 B，那么首先将 A 节点的 next 指向 B 节点的下一个节点即 C 节点，再将 C 节点的 prev 指向 A 节点：



3，容器

JAVA 中的 `LinkedList` 容器就是使用双向链表实现的，其结构如下：



可以看出，其实现了 `Queue` 和 `Deque` 接口，因此可以把其当做队列或双端队列来使用，同时也可以将其当做栈来使用。

使用 `LinkedList` 时，需要注意的是，当需要删除某些特定元素时，使用迭代器中的 `remove` 方法会更节省时间；例如，删除集合中的偶数，常规操作如下：

```
1  int i = 0;
2  while (i < list.size()) {
3      if (list.get(i) % 2 == 0) {
4          list.remove(i);
5      } else {
6          i++;
7      }
8  }
```

因为 `get` 和 `remove` 都需要遍历集合，因此时间复杂度为 $O(N)$ ，而 `while` 循环也需要 N 次，因此总的时间复杂度为 $O(N^2)$ 。

使用迭代器中的 `remove`：

```
1 Iterator<Integer> iterator = list.iterator();
2 while (iterator.hasNext()) {
3     if (iterator.next() % 2 == 0) {
4         iterator.remove();
5     }
6 }
```

因为要删除的元素正好是迭代器所指向的前一个元素，由于链表的特性将其直接删除只需花费 $O(1)$ 时间，while 循环需要 N 次，那么总的时间复杂度为 $O(N)$ 。

Talk is cheap, 验证一下:

```

1 class TestLinkedList {
2     public static void main(String[] args) {
3         System.out.printf("输入规模 \t自带remove方法 \t迭代器中remove方法 \n");
4         for (int n = 100; n <= 1000000; n *= 10) {
5             System.out.print(String.format("N=%7d", n));
6             for (int alg = 0; alg <= 1; alg++) {
7                 getTimingInfo(n, alg);
8             }
9             System.out.println();
10        }
11    }
12
13    private static void getTimingInfo(int n, int alg) {
14        LinkedList<Integer> list;
15        Random random=new Random();
16        long startTime = System.currentTimeMillis();
17        long totalTime = 0;
18
19        int k;
20        for (k = 0; totalTime < 4000; k++) {
21            list = new LinkedList<>();
22            for (int j = 0; j <n; j++) {
23                list.add(random.nextInt(n));
24            }
25            if (alg == 0) {
26                int i = 0;
27                while (i < list.size()) {
28                    if (list.get(i) % 2 == 0) {
29                        list.remove(i);
30                    } else {
31                        i++;
32                    }
33                }
34            } else if (alg == 1) {
35                Iterator<Integer> iterator = list.iterator();
36
37                while (iterator.hasNext()) {
38                    if (iterator.next() % 2 == 0) {
39                        iterator.remove();
40                    }
41                }
42            }
43            totalTime += System.currentTimeMillis() - startTime;
44        }
45    }
46 }

```

```

41         }
42     }
43     totalTime = System.currentTimeMillis() - startTime;
44 }
45
46 System.out.print(String.format("\t%12.6f", (totalTime * 1000 / k) / (double)
1000000));
47 }
48 }

```

Run: TestLinkedList x

S:\Java\jdk\bin\java.exe ...

输入规模	自带remove方法	迭代器中remove方法
N= 100	0.000006	0.000002
N= 1000	0.000341	0.000026
N= 10000	0.070877	0.000252
N= 100000	9.007000	0.002668
N=1000000	1738.076000	0.039450

从图中可以看出，集合自带的 remove 方法所消耗的时间确实是以平方级增长的，而迭代器中的 remove 方法则是以线性增长的。

还需注意的是，在使用迭代器时，不能使用集合中的方法来进行添加和删除的操作，否则会有 `ConcurrentModificationException` 异常，看看 `LinkedList` 中迭代器部分源码：

```

1 private class ListItr implements ListIterator<E> {
2     private Node<E> lastReturned;
3     private Node<E> next;
4     private int nextIndex;
5     private int expectedModCount = modCount;
6
7     public boolean hasNext() {
8         return nextIndex < size;
9     }
10
11     public E next() {
12         checkForComodification();
13         if (!hasNext())
14             throw new NoSuchElementException();
15
16         lastReturned = next;
17         next = next.next;
18         nextIndex++;
19         return lastReturned.item;
20     }
21     public void remove() {
22         checkForComodification();
23         if (lastReturned == null)
24             throw new IllegalStateException();
25

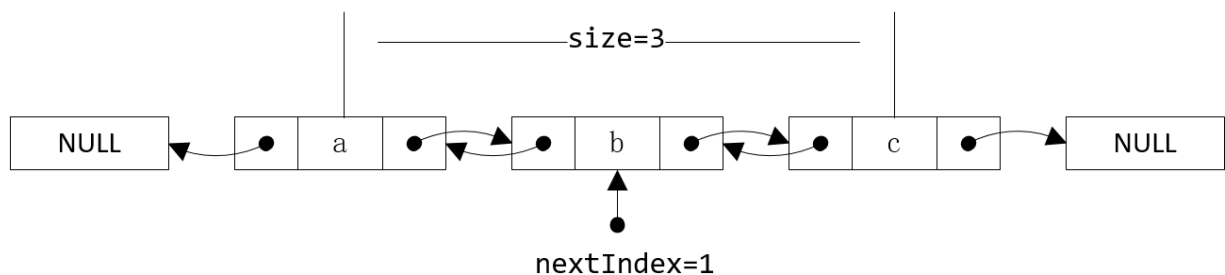
```

```

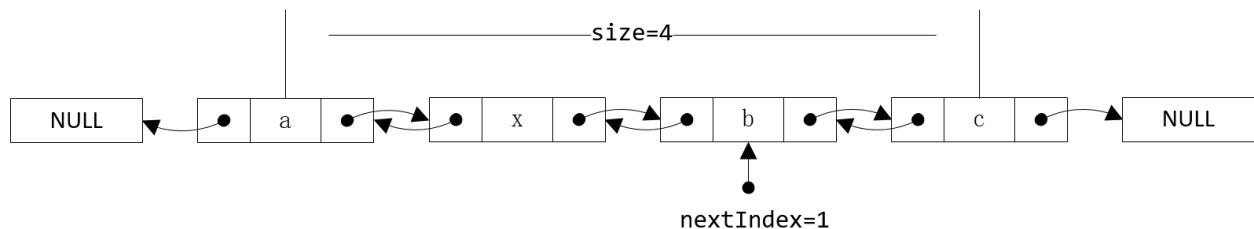
26         Node<E> lastNext = lastReturned.next;
27         unlink(lastReturned);
28         if (next == lastReturned)
29             next = lastNext;
30         else
31             nextIndex--;
32         lastReturned = null;
33         expectedModCount++;
34     }

```

假设集合中有三个元素 a,b,c，现在使用迭代器遍历完元素 a，那么此时 next 应返回元素 a，其中 nextIndex=1，而 size=3：

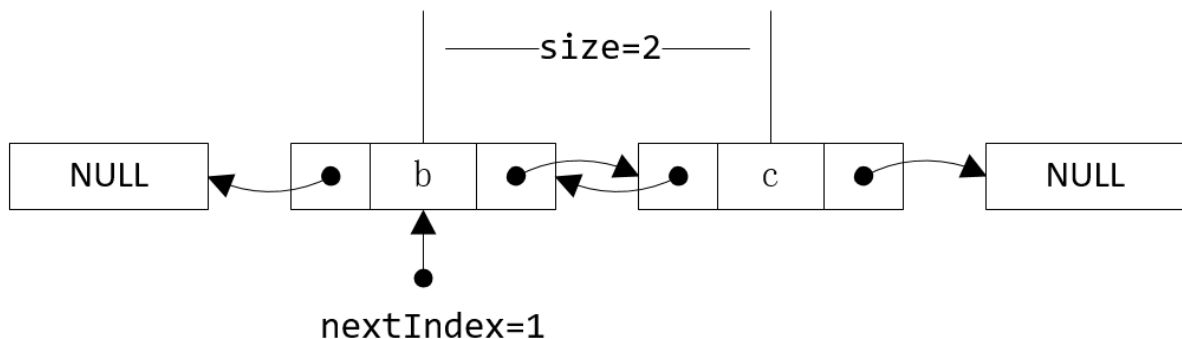


若此时在元素 b 之前插入元素 x，那么此时 size=4，但是 nextIndex 依然为 1：



那么遍历完 b 之后 nextIndex=2，遍历完 c 后，应该终止遍历了，但是 nextIndex=3，此时 nextIndex < size，hasNext 进行判断时依然有效，显然，此时调用 next 将会返回 NULL，会造成空指针异常。

若此时删除元素 a，那么此时 size=2，nextIndex=1：



遍历完元素 b 后，nextIndex=2 与 size 相等，此时调用 hasNext 会返回 false，但是集合中还有元素 c 没有被遍历到。

而迭代器中添加和删除操作会对 nextIndex 以及其他属性进行调整，因此，在使用迭代器时，应使用迭代器中的方法进行添加或删除操作。

4, LinkedList的简单实现

```
1 public class MyLinkedList<AnyType> implements Iterable<AnyType> {
2     private int theSize;
3     private int modCount = 0;
4     private Node<AnyType> beginMarker;
5     private Node<AnyType> endMarker;
6
7     private static class Node<AnyType> {
8         public AnyType data;
9         public Node<AnyType> prev;
10        public Node<AnyType> next;
11
12        public Node(AnyType data, Node<AnyType> prev, Node<AnyType> next) {
13            this.data = data;
14            this.prev = prev;
15            this.next = next;
16        }
17    }
18
19    public MyLinkedList() {
20        doClear();
21    }
22
23    public void clear() {
24        doClear();
25    }
26
27    private void doClear() {
28        beginMarker = new Node<>(null, null, null);
29        endMarker = new Node<>(null, beginMarker, null);
30        beginMarker.next = endMarker;
31
32        theSize = 0;
33        modCount++;
34    }
35
36    public int size() {
37        return theSize;
38    }
39
40    public boolean isEmpty() {
41        return size() == 0;
42    }
43
44    public boolean add(AnyType element) {
45        add(size(), element);
46        return true;
47    }
48
49    public void add(int index, AnyType element) {
50        addBefore(getNode(index, 0, size()), element);
```

```

51     }
52
53     private void addBefore(Node<AnyType> p, AnyType element) {
54         Node<AnyType> newNode = new Node<>(element, p.prev, p);
55         newNode.prev.next = newNode;
56         p.prev = newNode;
57         theSize++;
58         modCount++;
59     }
60
61     public AnyType get(int index) {
62         return getNode(index).data;
63     }
64
65     public AnyType set(int index, AnyType newVal) {
66         Node<AnyType> p = getNode(index);
67         AnyType oldVal = p.data;
68
69         p.data = newVal;
70         return oldVal;
71     }
72
73     private Node<AnyType> getNode(int index) {
74         return getNode(index, 0, size() - 1);
75     }
76
77     private Node<AnyType> getNode(int index, int lower, int upper) {
78         Node<AnyType> p;
79         if (index < lower || index > upper) {
80             throw new IndexOutOfBoundsException("getNode index: " + index + "; size:
81 " + size());
82         }
83         if (index < size() / 2) {
84             p = beginMarker.next;
85             for (int i = 0; i < index; i++) {
86                 p = p.next;
87             }
88         } else {
89             p = endMarker;
90             for (int i = size(); i > index; i--) {
91                 p = p.prev;
92             }
93         }
94         return p;
95     }
96
97     public AnyType remove(int index) {
98         return remove(getNode(index));
99     }
100
101     private AnyType remove(Node<AnyType> p) {
102         p.next.prev = p.prev;
103         p.prev.next = p.next;

```



```

103     theSize--;
104     modCount++;
105
106     return p.data;
107 }
108
109 @Override
110 public String toString() {
111     StringBuilder sb = new StringBuilder("[ ");
112
113     for (AnyType element : this) {
114         sb.append(element + " ");
115     }
116     sb.append("]");
117     return sb.toString();
118 }
119
120 @Override
121 public Iterator<AnyType> iterator() {
122     return new LinkedListIterator();
123 }
124
125 private class LinkedListIterator implements Iterator<AnyType> {
126     private Node<AnyType> current = beginMarker.next;
127     private int expectedModCount = modCount;
128     private boolean okToRemove = false;
129
130     @Override
131     public boolean hasNext() {
132         return current != endMarker;
133     }
134
135     @Override
136     public AnyType next() {
137         if (modCount != expectedModCount) {
138             throw new ConcurrentModificationException();
139         }
140         if (!hasNext()) {
141             throw new NoSuchElementException();
142         }
143         AnyType nextElement = current.data;
144         current = current.next;
145         okToRemove = true;
146         return nextElement;
147     }
148
149     @Override
150     public void remove() {
151         if (modCount != expectedModCount) {
152             throw new ConcurrentModificationException();
153         }
154         if (!okToRemove) {
155             throw new IllegalStateException();

```

```
156         }
157
158         MyLinkedList.this.remove(current.prev);
159         expectedModCount++;
160         okToRemove = false;
161     }
162 }
163 }
```

5, 比较

时间复杂度	数组	链表
插入删除	$O(N)$	$O(1)$
随机访问	$O(1)$	$O(N)$

数组使用连续空间，因此可以很容易计算其中某个元素的地址，实现常数级的随机访问，这也是数组的一个缺点，使用连续空间那么其大小要是固定的，并且一经申明就不可以更改，若申明过大，一是系统可能没有足够的连续内存，二是若用不了这么多空间，会造成浪费；即使 `ArrayList` 支持动态扩容，若此时已经存储了1GB的数据，而且没有多余的空间了，就要扩容至1.5GB，并把之前的1GB数据拷贝过来，会非常的耗时。

但是，如果对内存的使用比较苛刻，那么久适合使用数组，因为链表中每个节点都需要消耗额外的空间去存储 `next` 和 `prev`。对链表进行频繁的插入和删除时，会频繁的造成内存的申请和释放，容易造成内存碎片，在 `JAVA` 中，还会造成频繁的 `GC`。