LinkedList，Binary Search，和 Time Complexity

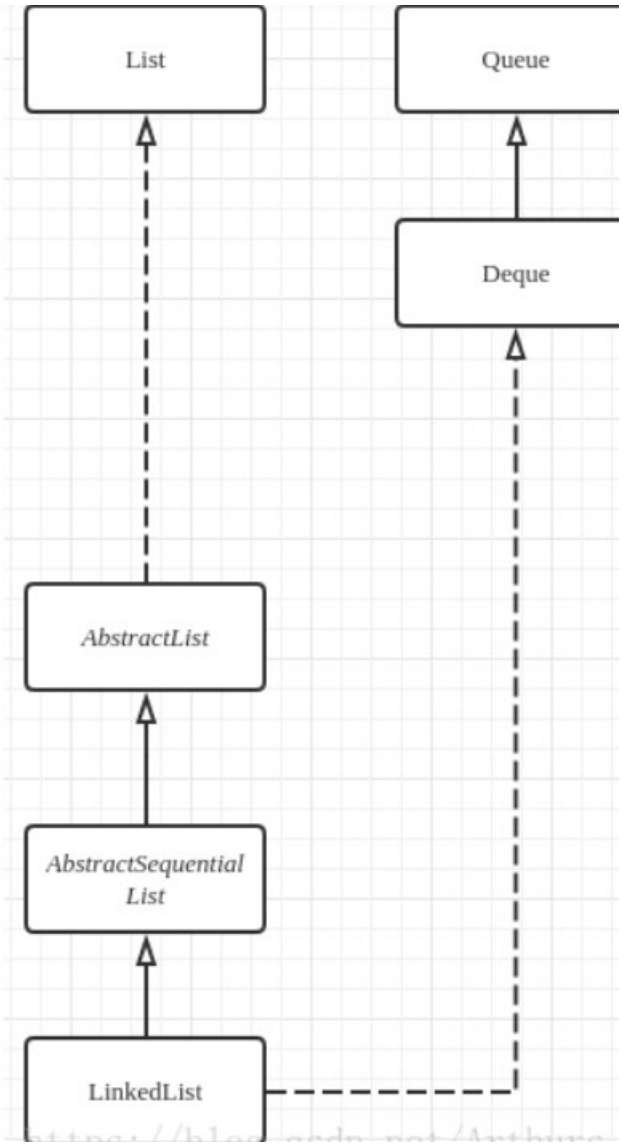
理解 LinkedList

1. **Java 内的 LinkedList 码源分析**

   下面是 LinkedList 的 java 自带 jdk 的设计

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>{...}
```

我们可以看到，LinkedList 继承 1 个类，2 个接口，具体的结构可以用下图表示，虚线表示的是接口，实现表示的是 abstract class 的继承：



它和 ArrayList 的底层实现方式的不同在于，ArrayList 的底层是由数组来实现，那么这样的底层就决定了 ArrayList 的读取速度会比较快，毕竟访问的是一个连续的内存地址，但是带来的弊端就是查询和修改会比较麻烦，需要遍历，再进行对应的读取和位移

这个时候 LinkedList 的优势便体现了出来，其底层由双向链表构成这点可以保证其修改，只需要针对指定节点的前后指向进行修改即可。

注意，java 的底层自带的 linked list，也就是我们每次 import 是 doubly linkedlist 而非 singly linkedlist

```
public int numberOfMatches(int n) {
        return n - 1;
    }
```

2. **LinkedList 自带的 Node class**

```
1  private static class Node<E> {
2          E item;
3          Node<E> next;
4          Node<E> prev;
5
6          Node(Node<E> prev, E element, Node<E> next) {
7              this.item = element;
8              this.next = next;
9              this.prev = prev;
10         }
11     }
```

next 和 prev 都有，再次证明了这是 doubly linkedlist

3. **LinkedList 的具体方法**
    1. **add 的方法：**

```
9       public boolean add(E e) {
10          linkLast(e);
11          return true;
12      }
13      /**
14       * Links e as last element.
15       */
16      void linkLast(E e) {
17          final Node<E> l = last;
18          final Node<E> newNode = new Node<>(l, e, null);
19          last = newNode;
20          if (l == null)
21              first = newNode;
22          else
23              l.next = newNode;
24          size++;
25          modCount++;
26      }
```

可以看到它 java 自带的这个 linkedlist 能直接统计 size，但是平时自己写的 linkedlist 没写 size，每次要找到 size 都得从头遍历到尾

2. 另一个 add 方法，在具体位置增加 node：

```
1   public void add(int index, E element) {
2           checkPositionIndex(index);
3           if (index == size)
4               linkLast(element);
5           else
6               linkBefore(element, node(index));
7       }
8
```

```
35  void linkBefore(E e, Node<E> succ) {
36          // assert succ != null;
37          final Node<E> pred = succ.prev;
38          final Node<E> newNode = new Node<>(pred, e, succ);
39          succ.prev = newNode;
40          if (pred == null)
41              first = newNode;
42          else
43              pred.next = newNode;
44          size++;
45          modCount++;
46      }
```

注意，这里的 add index 和上面 add 不是 overload!!! 原因：return type 是不一样的，overload 要求 return type 也是一样，只有 parameter 不一样才算 overload

4. **与 ArrayList 的对比**

arraylist 底层是 array，扩容就是如果发现不够了，就新建一个更大的 array，然后把原来 copy 到现在的里面

我们之前正课讲的 arraylist 是 add 和扩容都讲过了，我们现在来看看 remove

```
1       // 删除ArrayList中第一次出现的特定元素
2       public boolean remove(Object o) {
3           if (o == null) {
4               for (int index = 0; index < size; index++)
5                   if (elementData[index] == null) {
6                       fastRemove(index);
7                       return true;
8                   }
9           } else {
10              for (int index = 0; index < size; index++)
11                  // 比较对象时依赖equals方法
12                  // 因此类型变量E对应的类注意重写equlas方法
13                  // 重写时注意遵守规范，具体参考effective java第三版的第10、11两条规则
14                  if (o.equals(elementData[index])) {
15                      fastRemove(index);
16                      return true;
17                  }
18          }
19          return false;
20      }
```

```java
    // 根据下标删除元素
    private void fastRemove(int index) {
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            // 将elemenData中index+1及其后面的元素都向前移动一个下标
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        // 根据上一步的操作，size-1位置的对象向前移动了一个下标
        // 如果没有elementData[--size]==null，可能会导致内存泄漏
        // 试想，ArrayList被add了100个对象，然后被remove了100次。按照GC的机制来说，100个对象应
        elementData[--size] = null;
    }
```

```java
    // 根据下标删除元素
    // 注意：java5后引入自动装箱、拆箱的机制，因此产生了一个有趣的问题：
    // 当类型变量为Integer的ArrayList调用remove时，可能调用remove(Object)，也可能调用remove(
    // 一定要注意测试是否符合自己的预期
    public E remove(int index) {
        rangeCheck(index);

        modCount++;
        E oldValue = elementData(index);

        int numMoved = size - index - 1;
        // 如果被删除元素不是ArrayList的最后一个元素
        if (numMoved > 0)
            // 对应下标之后的元素向前移动一个下标
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        // 最后一个元素只为null，方便GC
        elementData[--size] = null;

        return oldValue;
    }
```

看到了以后大家应该明白了，以后想要把 array 后面一堆东西 copy 到前面去，可以选择用 System.arrayCopy 方法哦

这个是 arraylist 的 remove 方法，再结合课上讲的 add，基本就是最基本的雏形了

5. **LinkedList 的算法思考题和更深度的对于 LinkedList 的理解：**

    1. **虽然 linkedlist 是一个个 node 组成的，但是如果我有了 head，我同样可以用这个 head 去表示目前的 linkedlist，也就是说我们完全可以使用先只用 node 来表示我想要 return 的那个 linkedlist，先把 node 建好连在一起，然后把它们组成一个 linkedlist**

    2. **承接上面我说的有关 linkedlist 的 node 的表达方法，我们在 iterate linkedlist 时候，可以用指针表示现在 node 的位置，指针就是一个标签，它的箭头指向虚拟机里存的 node 的地址，我们需要找下一个，就让给指针分配新的内存和地址，让它指代别的，我们要 delete，其实就是省略中间的 node，让现在的 node 直接指向下下个，所以两个两个指针（标签）可以一个保存现在的值，一个保存下一个的值，那么再有一个不动的 head，就可以去 iterate 整个 linkedlist，如果我们想要找到 linkedlist**

的中点，其实就是从 head 开始，然后一个慢指针，一个快指针往下搜索，慢指针一次只往下 move 一个 index，快指针一次 move 两个 index，最后 return 慢指针就好，这个是 linkedlist 的独特的思考方式

3. 总结一下：
   1. linkedlist 的 head 就是 linkedlist 的本身，head 可以表示整个 linkedlist
   2. 通过 2 个指针我们可以 iterate 整个 linkedlist 对其进行复杂高深的操作，PS：如果有时操作还不够可以思考是不是还需要第三个指针，可以想一想反转链表是不是这个道理

6. 关于 LinkedList 的算法 example：

   在我们课程 linkedlist 里面再建一个 method，用来 return 在整个 linkedlist 的中间位置的 node

   思考角度：
   建立两个 node，一个 node 每次往前移动 1 一个单位，一个 node 每次往前移动两个单位，然后 return 第一个 node 的 data 就可

   代码示例：

```
public int returnMiddle(LinkedList a) {
 Node fast = head;
 Node slow = head;
 while(fast != null && slow != null && fast.next != null) {
  slow = slow.next;
  fast = fast.next.next;
 }
 return slow.data;
}
```

7. 打卡作业 HW9 讲解：

   题目：You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself.

   Example 1 : Input: l1 = [2,4,3], l2 = [5,6,4] Output: [7,0,8]
   Explanation: 342 + 465 = 807.

Example 2: Input: l1 = [0], l2 = [0] Output: [0]

Example 3: Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9] Output: [8,9,9,9,0,0,0,1]

Source: LeetCode

思路：

1. 将两个链表看成是相同长度的进行遍历，如果一个链表较短则在前面补 00，比如 987 + 23 = 987 + 023 = 1010

2. 每一位计算的同时需要考虑上一位的进位问题，而当前位计算结束后同样需要更新进位值

3. 如果两个链表全部遍历完毕后，进位值为 11，则在新链表最前方添加节点 11

小技巧：对于链表问题，返回结果为头结点时，通常需要先初始化一个预先指针 pre，该指针的下一个节点指向真正的头结点 head。使用预先指针的目的在于链表初始化时无可用节点值，而且链表构造过程需要指针移动，进而会导致头指针丢失，无法返回结果。

代码示例：

```
public Node addTwoNumbers(Node l1, Node l2) {
       Node pre = new Node(0);
       Node cur = pre;
       int carry = 0;
       while(l1 != null || l2 != null) {
           int x = l1 == null ? 0 : l1.val;
           int y = l2 == null ? 0 : l2.val;
           int sum = x + y + carry;

           carry = sum / 10;
           sum = sum % 10;
           cur.next = new Node(sum);

           cur = cur.next;
           if(l1 != null)
               l1 = l1.next;
           if(l2 != null)
               l2 = l2.next;
       }
       if(carry == 1) {
```

```
            cur.next = new Node(carry);
        }
        return pre.next;
    }
```

Binary Search：

1. **模板：**

   先找到 high 和 low，然后在 while loop 里面，第一行，middle = （low + high）/ 2
   第二和第三行：关于 low 和 high 的值的变更就是 low = mid + 1 或者 high = mid – 1，
   下面是伪代码（python）：

```
int low = i + 1;
int high = nums.length - 1;
while(low <= high){
    int mid = (high + low) / 2;
    if(nums[mid] > rest)
        high = mid -1;
    else if(nums[mid] < rest)
        low = mid + 1;
    else{
        res[0] = i + 1;
        res[1] = mid + 1;
        return res;
    }
}
```

2. **使用 recursion 实现 Binary Search**

   如果要使用 recursion 来写 binary search 的话，那每次 if 判断完了 return 这个 binary search 的 method 就好。

```
public static int rank (int key,int[] a) {
    return rank(key,a,0,16,1);
}
public static int rank (int key,int[] a,int lo,int hi,int deep) {
    if (hi < lo) return - 1;
    int mid = lo + (hi - lo) / 2;
    for(int i = 0 ; i < deep ; i++)
        System.out.print(" ");
    System.out.println("lo: "+lo+" hi: "+hi);
    if (key < a[mid])
        return rank (key,a,lo,mid - 1,deep + 1);
    else if (key > a[mid])
```

```
        return rank (key,a,mid + 1,hi,deep + 1);
    else
        return mid;
}
```

3. 算法 example

给定一个已经 sort 完成的 array，和一个 target，寻找在 array 里面两个不同 index 位置加起来能等于目前的 target 的 index 并返回(可以使用 binary search，不过也可以考虑其他的方法哦，binary search 不一定是最简便的)

思考角度一：
我们可以暴力列举所有可能的情况，也就是第一个 for loop 遍历整个 array，找到每个位置，然后第二个 for loop 从这个位置的下一个位置开始找，如果两个位置相加等于 target 就 return

这个方法的话，时间复杂度是 O(n^2)，但是因为我们学过使用 Binary Search，取代第二个 for loop，找到第一个 index 后，我完全可以用 binary search 找到第二个 index，这样时间复杂度就会被减少到 O(n log n)

代码示例：

```
public int[] twoSum(int[] nums, int target) {
        int[] res = new int[2];
        for(int i = 0; i < nums.length - 1; i++){
            int rest = target - nums[i];
            int low = i + 1;
            int high = nums.length - 1;
            while(low <= high){
                int mid = (high + low) / 2;
                if(nums[mid] > rest)
                    high = mid -1;
                else if(nums[mid] < rest)
                    low = mid + 1;
                else{
                    res[0] = i;
                    res[1] = mid;
                    return res;
                }
            }
        }
        return res;
    }
```

思考角度二：

既然是已经 sort 过了，那我们完全可以从左边和右边同时找，加起来小于 target 就让左边的 index + 1，加起来大于 target 就让左边的 index – 1，那我们就只需要一个 while loop 就可以实现了，所以时间复杂度就下降到 O(n)了，比刚刚更快了

代码示例：

```
public int[] twoSum(int[] numbers, int target) {
       int n = numbers.length,i=0,j=n-1;
       while(i<j){
           int sum = numbers[i] + numbers[j];
           if(sum == target) return new int[]{i+1,j+1};
           if(sum > target) j--;
           else i++;
       }
       return null;
    }
```

Time Complexity：

1. **大作业 HW3 讲解：Time complexity 的 big - oh， Omega，和 theta**

回顾一下关于大作业 HW3 里的几道题，以及 time complexity 的 big - oh， Omega，和 theta 时间的复杂度是看一个函数里最大的复杂度，比如同时出现了 n^2 和 n log n，那我们只需要看 n^2 就够了

所以比较两个时间复杂度，一个具体时间的和 big O 比较， big Omega 或者 theta 比较，或者使用 big O 的话，我们最快的方式是把两边都花化简成函数中的最大复杂度的那一项

总结：

判断时间复杂度，首先把下面的不等式列举出来

$$1 < \log n < n^{(-x)} < n < n \log n < n^2 < x^n$$

其次删除我们式子中的常数，原本 2n 直接化简成 n，原本 2logn 直接化简成 logn，其次找到我们的式子中最大的复杂度来表示整个式子，比说对于 2SQRT(n) + log n 可以直接化简成 SQRT(n)

接下来和 big O 或者 big theta 或者 big Omega 来比较
如果是 big O,我们需要找到在上面不等式 SQRT(n）右边的，也就是说可以用 O(n), O(n long n)等来表示现在的

如果是 big theta，那就是和目前这个化简的相等的，所以 big theta 就是 Theta(SQRT(n)

如果是 big Omega，那就是找不等式左边的，也就是可以用 Omega(log n), Omega(1)来表示 SQRT(n)

2. 大作业 HW3 题目与答案：

```
1. For the following program fragment compute the worst-
case asymptotic time complexity (as a function of n). Where it says ''l
oop body'' you can assume that a constant number of lines of code are t
here. Briefly explain how you obtained your answer.

  *Hint: Write a nested summation to express the number of times the lo
op body is executed.*

          for (i=0; i<=n-1; i++){
              for (j=i+1; j<=n-1; j++){
              loop body
              }
          }
```

```
1）The number of times the loop body is executed is:

  n-1       n-1           n-1
  -----    -----         -----
  \        \             \
   \        \   1    =    \     (n-1) - (i+1) + 1
   /        /             /
  /        /             /
  -----   ------         -----
  i=0     j = i+1       i=0

Since (n-1) - (i+1) + 1 = (n-1) - i we get that the number of times the
loop
body is executed is:  (n-1) + (n-2) + ... + 0 = n(n-1)/2 = n^2/2 - n/2

Hence the time complexity is Theta(n^2)
```

2. For each of the following pairs of functions T1(n) and T2(n) clearly answer the following 4 questions: Is T1(n) = O(T2(n))?, Is T1(n) = Omega(T2(n))?, Is T1(n) = Theta(T2(n))? If you were given two algorithms A1 with time complexity T1(n) and A2 with time complexity T2(n), which would you pick if your goal was to have the fastest algorithm?

You should justify your answer using either the definition of big-oh, big-theta, big-Omega.

Hint: Limit Test sometimes works for this problem

- T1(n) = 6n^2, T2(n) = n^2 log n

- T1(n) = 3/2 n^2 + 7n -4, T2(n) = 8n^2

- T1(n) = n^4, T2(n) = (n^3) log n

2a)

In all of the below lim is used for the limit as n approaches infinity:

$$\lim \frac{6n^2}{(n^2)\log n} = 0$$

Hence (n^2) log n is asymptotically faster growing than 6n^2. Thus

6n^2 = O(n^2 log n), 6n^2 != Omega(n^2 log n), 6n^2 != Theta(n^2 log n)

A1 is the faster algorithm.

2b)

In all of the below lim is used for the limit as n approaches infinity:

$$\lim \frac{3/2\ n^2 + 7n - 4}{8n^2} = \lim \frac{3n + 7}{16\ n} = \lim \frac{3}{16} = 3/16$$

Hence these two functions grow at the same asymptotic growth rate, though

```
T1 has the smaller constant.  Thus

  T1(n) = O(T2(n)),  T1(n) = Omega(T2(n)) and T1(n) = Theta(T2(n))

  A1 is the faster algorithm since T1 has the smaller constant.
```

```
2c)

In all of the below lim is used for the limit as n approaches infinity:

     n^4
lim --------    = infinity
    n^3 log n

Hence n^4 is asymptotically faster growing.  Thus

  n^4 != O(n^3 log n),  n^4 = Omega(n^3 log n), n^4 != Theta(n^3 log n)

  A2 is the faster algorithm.
```

```
3. Prove whether or not each of the following statements are true. For
those that you believe are false, prove this by giving a counterexample
 (i.e. particular functions for f(n) and g(n) for which the given state
ment is not true). For those that you believe are true, use the formal
definitions of big-oh, big-Omega, and big-
Theta to prove it. In all problems, you are given that for all n, f(n)
>= 0 and g(n) >= 0.
  - If f(n) = O(g(n)) then g(n) = O(f(n))

  - f(n)+g(n) = O(max(f(n),g(n))) (covered in class!)

  - If f(n) = Omega(g(n)) then g(n) = O(f(n))
```

```
3a)
This is false as demonstrated by the counterexample, f(n)=n and
g(n)=n^2.
Note that n = O(n^2) but n^2 != O(n).
```

```
3b)
We prove this is true.  The key observation is that
   f(n) + g(n) <= 2 max(f(n) + g(n)) for all n
Thus by letting n_0 = 1 and c = 2 we get the required conditions to
have that f(n) + g(n) = O(max(f(n),g(n))
```

```
3c)

We prove this is true.  To prove  If p then q we assume p is true and
show that q must follow.  Thus we assume that f(n) = Omega(g(n)).  By
the definition
of Omega we have that constants c and n_0 exists such that

   f(n) >= c g(n) for all n >= n_0    (*)

We must show that there exists constants n'_0 and c' for which

   g(n) <= c' f(n) for all n >= n'_0

Solving for f(n) in the above, we get the equivalent requirement that

   f(n) >= 1/c' g(n) for all n >= n'_0

By letting 1/c' = c (so let c' = 1/c) and n'_0 = n_0 this is clearly
true since it is the same as the equation marked by a (*)
```

```
4. Are each of the following true or false?
 - 3 n^2 + 10 n log n = O(n log n) false
 - 3 n^2 + 10 n log n = Omega(n^2) true
 - 3 n^2 + 10 n log n = Theta(n^2) true
 - n log n + n/2 = O(n) false
 - 10 SQRT(n) + log n = O(n) true
 - SQRT(n) + log n = O(log n) false
 - SQRT(n) + log n = Theta(log n) false
 - SQRT(n) + log n = Theta(n) false
 - 2 SQRT(n) + log n = Theta(SQRT(n)) true
 - SQRT(n) + log n = Omega(1) true
 - SQRT(n) + log n = Omega(log n) true
 - SQRT(n) + log n = Omega(n) false
```

4) (a) False, since n^2 (the dominate term on the left) is asymptotically faster growing than n log n and hence not upperbounded by it.
(b,c) True, since n^2 (the dominate term on the left) asymptotically grows like n^2 and hence it is Omega(n^2) and also Theta(n^2). faster growing than n log n and hence not upperbounded by it.
(d) False since n log n (the dominate term on the left) is not asymptotically upperbounded by n.
(e) True, since the dominate term on the left, 10 SQRT(n), is asymptotically upperbounded by n.
(f,g) False, since the dominate term on the left, SQRT(n), is not asymptotically upperbounded by n. See the class notes where we showed that that lim as n -> infinity of log n / SQRT(n) = 0 giving that SQRT(n) is asymptotically faster growing.
(h) False, since the dominate term on the left, SQRT(n), is asymptotically faster slower growing than n.
(i) True, since the dominate term on the left, 2 SQRT(n), grows asymptotically at the same rate as SQRT(n).
(j) True, since the dominate term on the left, SQRT(n), is asymptotically faster growing than 1.
(k) True, since the dominate term on the left, SQRT(n), is asymptotically faster growing than log n.
(l) False, since the dominate term on the left, SQRT(n), is asymptotically slower growing than n.