LinkedList，Binary Search，和Time Complexity

### 理解LinkedList

1. **Java内的LinkedList码源分析**

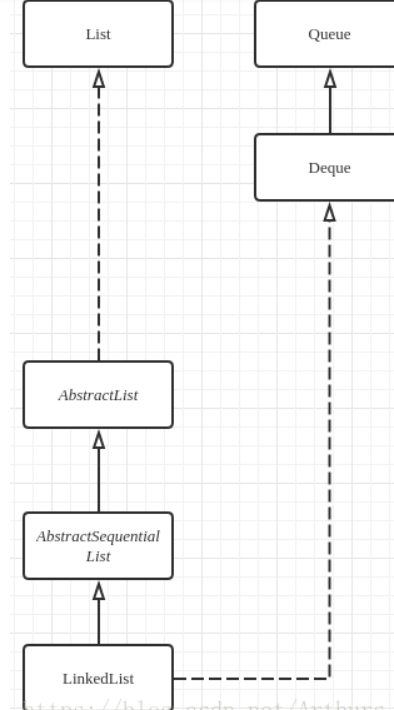
**下面是LinkedList的java自带jdk的设计**

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。

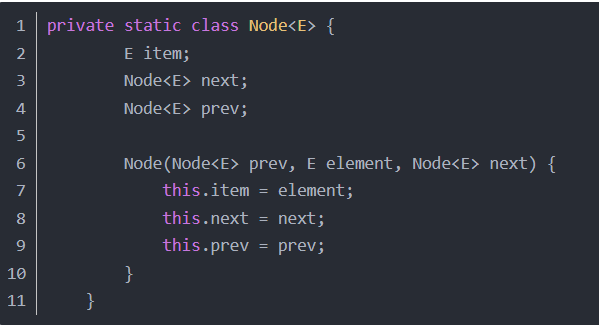
**注意，java的底层自带的linked list，也就是我们每次import是doubly linkedlist而非singly linkedlist**

public int numberOfMatches(int n) {

return n - 1;

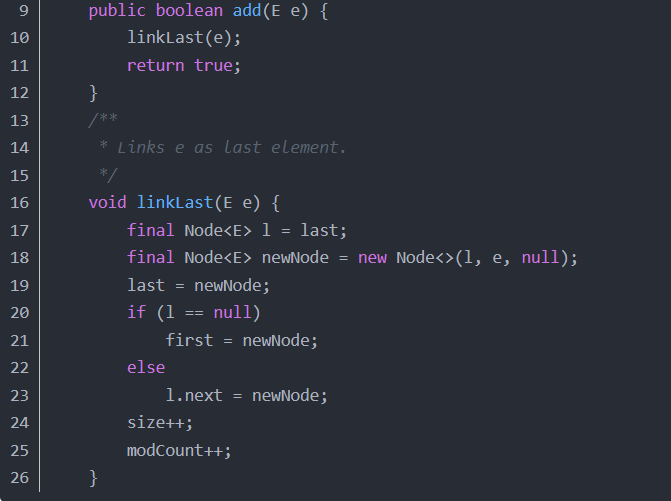
}

1. **LinkedList自带的Node class**

****

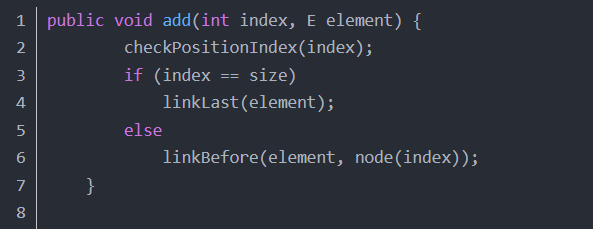
next和prev都有，再次证明了这是doubly linkedlist

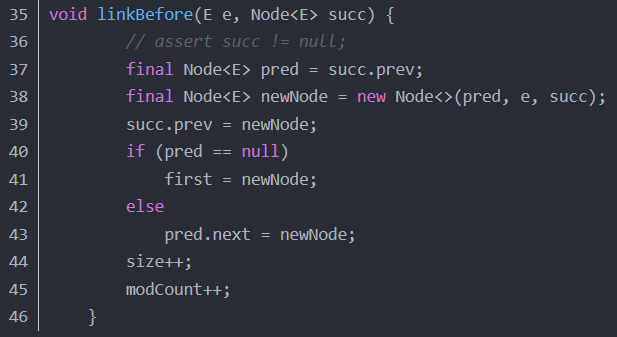
1. **LinkedList的具体方法**
2. **add的方法：**

****

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

1. **另一个add方法，在具体位置增加node：**

****

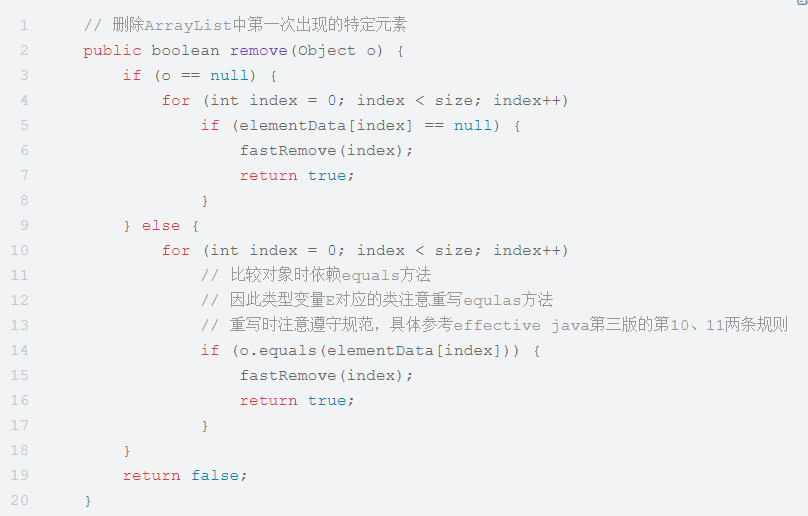
****

**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**

1. **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







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

1. **LinkedList的算法思考题和更深度的对于LinkedList的理解：**
2. **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**
3. **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**
4. **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**

1. **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;

}

1. **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;

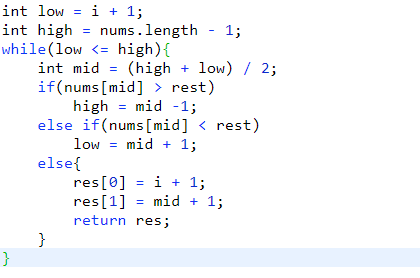
}

### 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）：

****

1. **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;

}

1. **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)

1. **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 ''loop body'' you can assume that a constant number of lines of code are there. Briefly explain how you obtained your answer.

*\*Hint: Write a nested summation to express the number of times the loop 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:

6n^2

lim -------- = 0

(n^2) log n

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:

3/2 n^2 + 7n - 4 3n + 7 3

lim ---------------- = lim ----------- = lim ---- = 3/16

8n^2 16 n 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 statement 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.