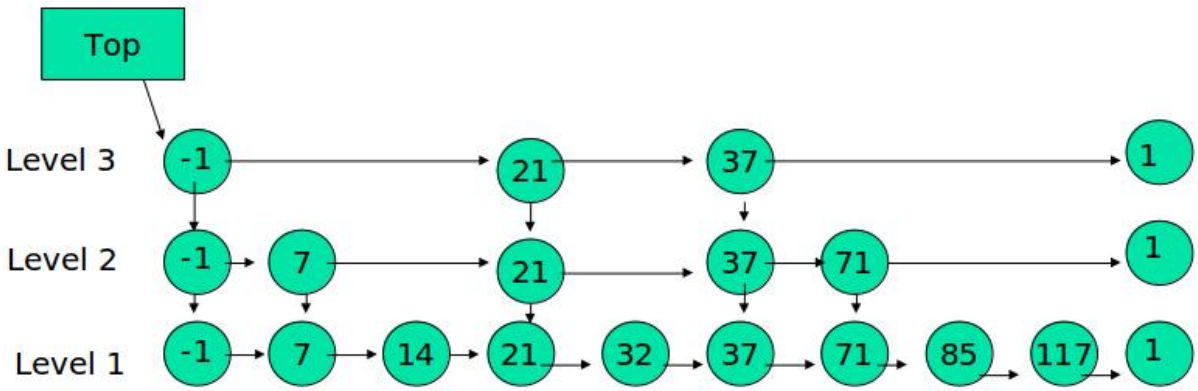
概念

跳跃表示一种数据结构，允许快速查询一个有序连续元素的数据链表。快速查询是通过维护一个多层次的链表，且每一层链表中的原始是前一层链表元素的自己。

效率和平衡树媲美---查找、删除、添加等操作都可以在$O(\log n)$ 期望时间下完成，并且比平衡树来说，跳跃表的实现要简单直观的多。基本上跳跃列表是对有序链表增加上附件的浅见链接，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分列表，因此得名

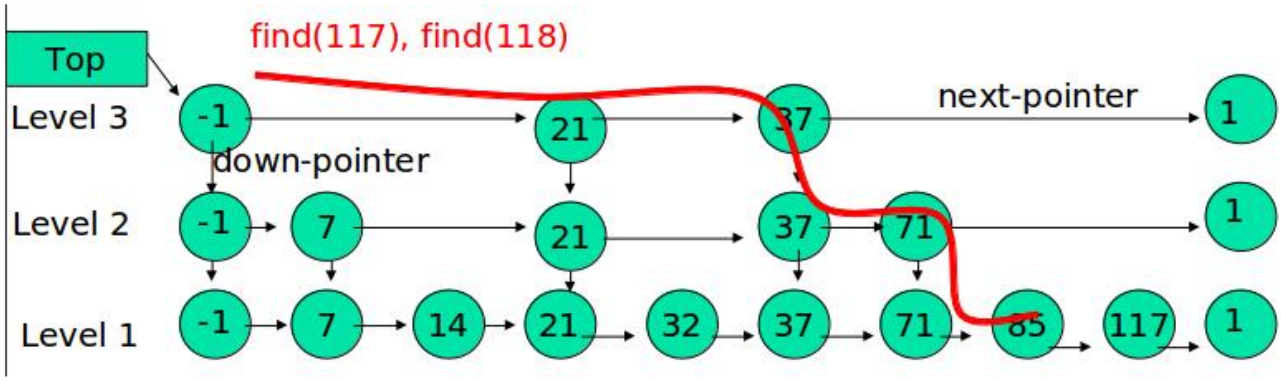
数据结构



跳跃表具有如下性质：

1. 由很多层结构组成
2. 每一层都有一个有序的链表
3. 最底层链表包含所有元素
4. 如果一个元素出现在`Level i`的链表中，则他在`Level i`之下的链表中都会出现
5. 每个节点包含两个指正，一个指向统一链表中的下一个元素，一个指向下面一层的元素

跳表的搜索



查找流程

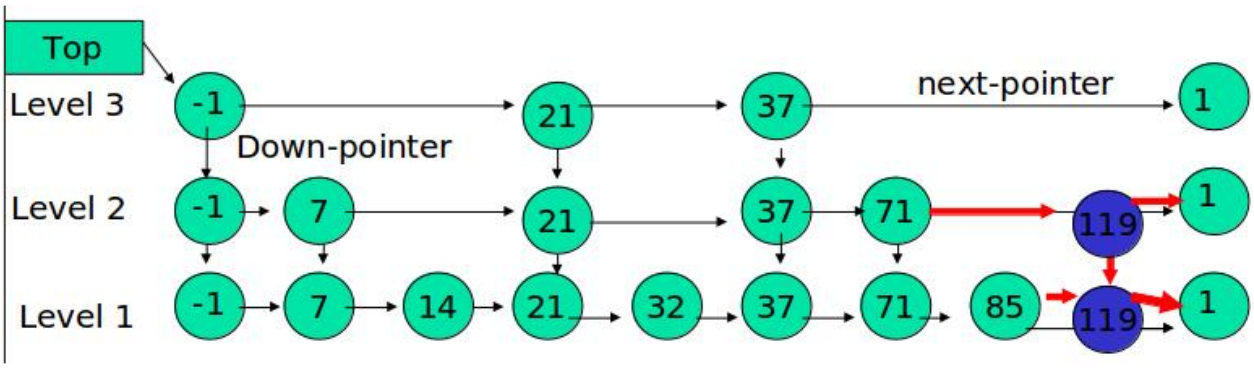
```
1. 比较21，比21大，往后面找
2. 比较 37，  比 37大，比链表最大值小，从 37 的下面一层开始找
3. 比较 71，  比 71 大，比链表最大值小，从 71 的下面一层开始找
4. 比较 85，  比 85 大，从后面找
5. 比较 117，  等于 117，  找到了节点。
```

跳跃表的插入

先确定该元素要占据的层数 K（采用丢硬币的方式，这完全是随机的）

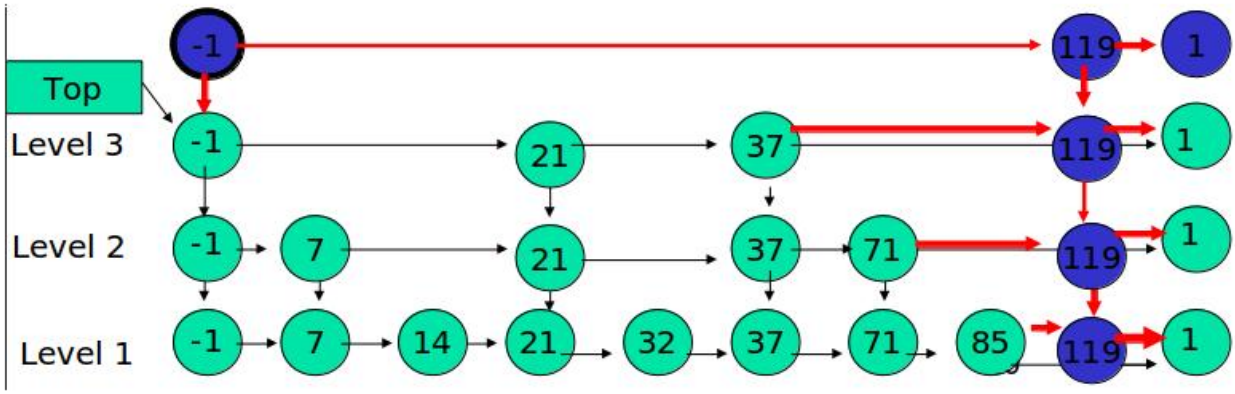
然后在 Level 1 ... Level K 各个层的链表都插入元素。

例子：插入 119， K = 2



如果 K 大于链表的层数，则要添加新的层。

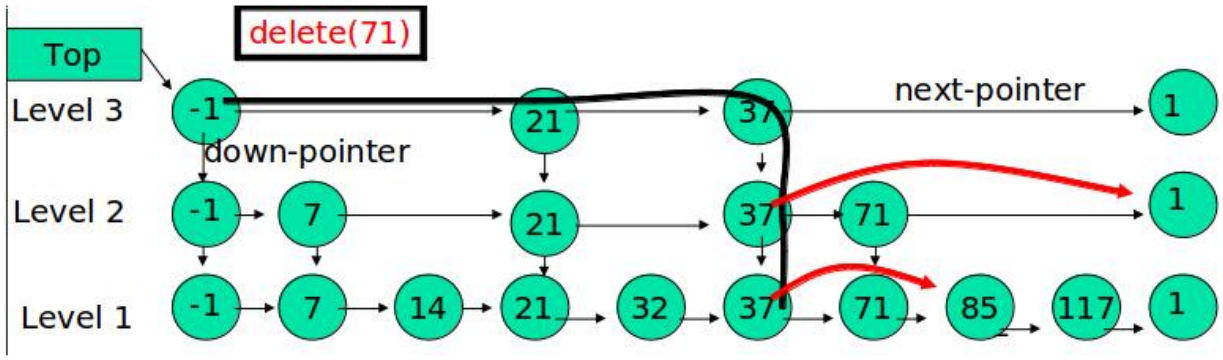
例子：插入 119， K = 4



跳跃表删除

在各个层中找到包含 x 的节点，使用标准的 delete from list 方法删除该节点。

例子：删除 71



跳表的空间复杂度分析

n 个元素的跳表，每个元素插入的时候都要做一次实验，用来决定元素占据的层数 K，

跳表的高度等于这 n 次实验中产生的最大 K，待续。。。

根据上面的分析，每个元素的期望高度为 2， 一个大小为 n 的跳表，其节点数目的

期望值是 2n。

java代码实现（查找、删除）

```
public class SkipList {

    public int level = 0;
    public SkipListNode top = null;
```

```
public SkipList() {
    this(4);
}

//跳跃表的初始化
public SkipList(int level) {
    this.level = level;
    SkipListNode skipListNode = null;
    SkipListNode temp = top;
    SkipListNode tempDown = null;
    SkipListNode tempNextDown = null;
    int tempLevel = level;
    while(tempLevel -- != 0){
        skipListNode = createNode(Integer.MIN_VALUE);
        temp = skipListNode;
        skipListNode = createNode(Integer.MAX_VALUE);
        temp.setNext(skipListNode);
        temp.setDownNext(tempDown);
        temp.getNext().setDownNext(tempNextDown);
        tempDown = temp;
        tempNextDown = temp.getNext();
    }
    top = temp;
}

//随机产生数k，k层下的都需要将值插入
public int randomLevel(){
    int k = 1;
    while(new Random().nextInt()%2 == 0){
        k ++;
    }
    return k > level ? level : k;
}

//查找
public SkipListNode find(int value){
    SkipListNode node = top;
    while(true){
        while(node.getNext().getValue() < value){
            node = node.getNext();
        }
        if(node.getDownNext() == null){
            //返回要查找的节点的前一个节点
            return node;
        }
        node = node.getDownNext();
    }
}

//删除一个节点
public boolean delete(int value){
```

```
        int tempLevel = level;
        SkipListNode skipListNode = top;
        SkipListNode temp = skipListNode;
        boolean flag = false;
        while(tempLevel -- != 0){
            while(temp.getNext().getValue() < value){
                temp = temp.getNext();
            }
            if(temp.getNext().getValue() == value){
                temp.setNext(temp.getNext().getNext());
                flag = true;
            }
            temp = skipListNode.getDownNext();
        }
        return flag;
    }

    //插入一个节点
    public void insert(int value){
        SkipListNode skipListNode = null;
        int k = randomLevel();
        SkipListNode temp = top;
        int tempLevel = level;
        SkipListNode tempNode = null;
        //当在第n行插入后，在第n - 1行插入时需要将第n行backTempNode的DownNext域指向第n -
1的域
        SkipListNode backTempNode = null;
        int flag = 1;
        while(tempLevel-- != k){
            temp = temp.getDownNext();
        }

        tempLevel++;
        tempNode = temp;
        //小于k层的都需要进行插入
        while(tempLevel-- != 0){
            //在第k层寻找要插入的位置
            while(tempNode.getNext().getValue() < value){
                tempNode = tempNode.getNext();
            }
            skipListNode = createNode(value);
            //如果是顶层
            if(flag != 1){
                backTempNode.setDownNext(skipListNode);
            }
            backTempNode = skipListNode;
            skipListNode.setNext(tempNode.getNext());
            tempNode.setNext(skipListNode);
            flag = 0;
            tempNode = tempNode.getDownNext();
        }
```

```
    }

    //创建一个节点
    private SkipListNode createNode(int value){
        SkipListNode node =  new SkipListNode();
        node.setValue(value);
        return node;
    }


}

/**
 * Node节点
 */
class SkipListNode implements Comparable {

    //节点存储的值
    private int value;
    //当前节点指向后面的节点
    private SkipListNode next = null;

    //当前节点指向下一层级节点
    private SkipListNode downNext = null;

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.printf("123");
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public SkipListNode getNext() {
        return next;
    }

    public void setNext(SkipListNode next) {
        this.next = next;
    }

    public SkipListNode getDownNext() {
        return downNext;
    }
```

```
    public void setDownNext(SkipListNode downNext) {
        this.downNext = downNext;
    }

    @Override
    public int compareTo(Object o) {
        return this.value > ((SkipListNode)o).value ? 1 : -1;
    }
}
```