这里可能要掌握一些基本的数据结构，防止到时候直接被问到了。

随便看了一下红黑树，发现原理还是比较简单的，以前的我为什么看半天都看不懂？真的是不明白为什么。

二、树

平衡树：AVL树，特殊的二叉排序树，左右子树全是平衡二叉树，左右子树高度之差的绝对值不超过1.

左子树深度-右子树深度得到平衡因子BF，这个BF的绝对值不可能超过1。

红黑树：二叉查找树。每个节点加一个存储位表示节点的颜色（红色或者黑色）

通过着色方式限制，它保证不存在一条路径比另一条路径的长度超出二倍。

基本限制：每条路径上的黑色节点数量相同，黑色下面可以是黑色，红色的子节点必须是黑色，红色子节点不可能重复。

插入限制：为了维持基本性质，首先按照二叉查找树方式找到位置，之后进行各种不同情况的旋转、着色，保持红黑树的性质。

插入最多两次旋转，删除最多三次旋转。

哈夫曼树：自底向上构建二叉树T

合并两颗最小值的树，合并后，根节点是两颗子树的加和。

B+树：

多路搜索树，为了磁盘读取而设计的一种平衡查找树。B+树中，每个节点可以有多个孩子，并且按照关键字的大小有序排列。

每个节点上指针上限为2d，内节点不存储data，只存储key，叶节点不存储指针。

B+树和B树可以稍微放弃一下，了解的并不是特别多哦。

1.快排手写代码：这个是很有可能会考的。

```
int once_quick_sort(vector &data, int left, int right) {
// 在这里是闭区间
```

```
// 快排的区间划分方法，原理还算是比较简单的，处理过程也是比较简洁的
// 注意这里的重要的区间划分
int key = data[left];

while(left < right) {
 while(left < right && key <= data[right]) {
     right--;
    }
    if(left < right) {
     data[left++] = data[right];
    }
    while(left < right && key > data[left]) {
     left++;
    }
    if(left < right) {
        // 不会产生数据覆盖的问题吗？
     data[right--] = data[left];
    }
}
data[left] = key;
return left;
```

}

x

1

int once_quick_sort(vector &data, int left, int right) {

2

```
// 在这里是闭区间
```

3

4

```
// 快排的区间划分方法，原理还算是比较简单的，处理过程也是比较简洁的
```

5

```
// 注意这里的重要的区间划分
```

6

```
int key = data[left];
```

7

8

```
while(left < right) {
```

9

```
    while(left < right && key <= data[right]) {
```

10

```
        right--;
```

11

```
    }
```

12

```
    if(left < right) {
```

13

```
        data[left++] = data[right];
```

14

```
    }
```

15

```
    while(left < right && key > data[left]) {
```

```
        left++;
    }
    if(left < right) {
        // 不会产生数据覆盖的问题吗?
        data[right--] = data[left];
    }
}
data[left] = key;
return left;
}
// 以下就是快排的划分过程
```

```
int quickSortOnce(vector &data, int left, int right) {
int val = data[left];
int state = 0;
while(left < right) {
while(state == 0 && left < right && data[right] >= val) {
right--;
}
if(state == 0 && left < right) {
data[left++] = data[right];
state = 1; //开始循环左边
}
while(state == 1 && left < right && data[left] < val) {
left++;
}
// 此时right是一直不变的
if(state == 1 && left < right) {
data[right--] = data[left];//left已经被存储了
state = 0;
}
}
// 最后补一个数据补全
data[left] = val;
return left; //left就是最终的位置location
}

int quickSort(vector &data, int left, int right) {
if(left <= right) return 1;
// 记得对每个区间都进行划分，这个是重要的
int middle = quickSortOnce(data, left, right);
quicksort(data, left, middle-1);
quicksort(data, middle+1, right);

// 最终返回1代表完成了复杂的排序过程
return 1;
}

x
```

1

// 以下就是快排的划分过程

2

int quickSortOnce(vector &data, int left, int right) {

3

```
int val = data[left];
```

4

```
int state = 0;
```

5

```
while(left < right) {
```

6

```
    while(state == 0 && left < right && data[right] >= val) {
```

7

```
        right--;
```

8

```
    }
```

9

```
    if(state == 0 && left < right) {
```

10

```
        data[left++] = data[right];
```

11

```
        state = 1; //开始循环左边
```

12

```
    }
```

13

```
    while(state == 1 && left < right && data[left] < val) {
```

14

```
        left++;
```

15

```
    }
```

16

```
    // 此时right是一直不变的
```

17

```
    if(state == 1 && left < right) {
```

18

```
        data[right--] = data[left];//left已经被存储了
```

19

```
        state = 0;
```

20

```
    }
```

21

```
}
```

22

```
// 最后补一个数据补全
```

23

```
data[left] = val;
```

24

```
return left; //left就是最终的位置location
```

25

}

26

27

int quickSort(vector &data, int left, int right) {

28

```
if(left <= right) return 1;
```

29

```
// 记得对每个区间都进行划分，这个是重要的
```

30

```
int middle = quickSortOnce(data, left, right);
```

31

```
quicksort(data, left, middle-1);
```

32

```
quicksort(data, middle+1, right);
```

33

34

```
// 最终返回1代表完成了复杂的排序过程
```

35

```
return 1;
```

36

}

2.归并排序代码：

void mergeSortMain(vector& data) {
int n = data.size();
vector temp(n, 0);

```
mergeSort(data, 0, data.size()-1, temp);
return;
```

}

void mergeSort(vector &data, int left, int right, vector &temp) {
if(left <= right) return;
int mid = (left+right)/2;

```
mergeSort(data, left, mid, temp);
mergeSort(data, mid+1, right, temp);
```

mergeArr(data, left, mid, right, temp);

```
return;
```

}

void mergeArr(vector &data, int left, int mid, int right, vector &temp) {
int loc1 = left;

int loc2 = mid + 1;
int loc = left;
while(loc1 <= mid && loc2 <=right) {
if(data[loc1] >= data[loc2]) {
// 在这里，使用降序排列
temp[loc++] = data[loc2++];
}
else {
temp[loc++] = data[loc1++];
}
}
while(loc1 <= mid) {
temp[loc++] = data[loc1++];
}
while(loc2 <= right) {
temp[loc++] = data[loc2++];
}
return;
}
// 归并排序，也就是40行代码，真的不算是特别的难

1

void mergeSortMain(vector& data) {

2

```
int n = data.size();
```

3

```
vector<int> temp(n, 0);
```

4

5

```
mergeSort(data, 0, data.size()-1, temp);
```

6

```
return;
```

7

}

8

void mergeSort(vector &data, int left, int right, vector &temp) {

9

```
if(left <= right) return;
```

10

```
int mid = (left+right)/2;
```

11

12

```
mergeSort(data, left, mid, temp);
```

13

```
mergeSort(data, mid+1, right, temp);
```

14

```
mergeArr(data, left, mid, right, temp);
```

15

16

```
return;
```

17

}

18

void mergeArr(vector &data, int left, int mid, int right, vector &temp) {

19

```
int loc1 = left;
```

20

```
int loc2 = mid + 1;
```

21

```
int loc = left;
```

22

```
while(loc1 <= mid && loc2 <=right) {
```

23

```
    if(data[loc1] >= data[loc2]) {
```

24

```
        // 在这里，使用降序排列
```

25

```
        temp[loc++] = data[loc2++];
```

26

```
    }
```

27

```
    else {
```

28

```
        temp[loc++] = data[loc1++];
```

```
    }
```

```
}
```

```
while(loc1 <= mid) {
```

```
    temp[loc++] = data[loc1++];
```

```
}
```

```
while(loc2 <= right) {
```

```
    temp[loc++] = data[loc2++];
```

```
}
```

```
return;
```

}

// 归并排序，也就是40行代码，真的不算是特别的难

3.了解B树和B+树

这两棵树是面试常考题型，需要我们一点点来了解的。

B树的概念

假设B树是m阶的

每个节点最多可以有m-1个关键字，根节点最少可以有1个关键字，非根节点至少有m/2个关键字。

每个节点中的关键字都按照从小到大的顺序排列，关键字左子树的所有关键字小与它，关键字右子树的所有关键字都大于它。这里和二叉查找树是极为相似的。

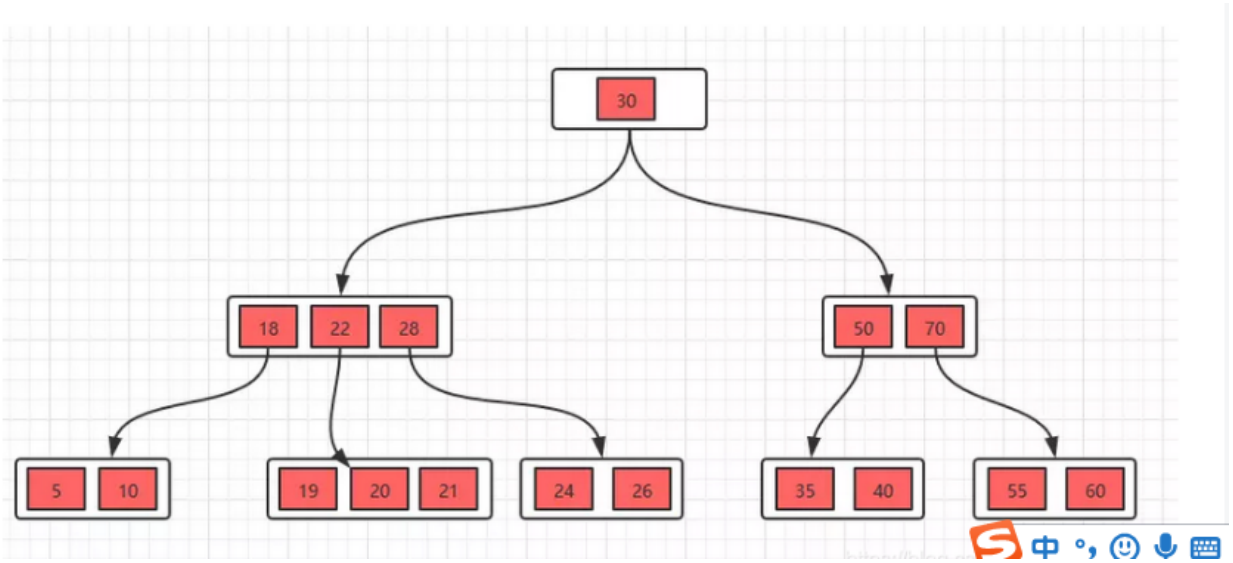
所有的叶子节点位于同一层，也就是说，根节点到每个叶子结点的长度都相同的。

每个节点都存有键值对，key、value

阶数表示一个节点最多有几个孩子节点，一般用字母m表示阶数。

也就是说，B树的子节点也会存储数值，B树的中间节点也会存储数值，这些点都是完全确定的。

下面展示一个典型的B树：另一种形式的多叉查找树



B+树的概念

根节点至少有一个元素，非根节点的元素数目 [m/2, m-1]。这个范围是非常明确的。

B+树有两种节点：内部节点（非叶子结点，索引节点，不存储数据，只存储索引，数据都在叶子上）、叶子结点（存储所有的数据）

叶子结点之间构成了一个链表。父节点存有右孩子第一个元素的索引，只存储索引，并不存储数据。

也就是内部节点拥有右边孩子的所有元素的索引。

倘若没有右孩子，是没有必要建立右孩子索引的。

数量满时，开始分裂，中间节点需要分到右边的子节点里面，这个logic是非常清晰地。

B+树相对于B树的优势：

单一节点存储元素更多，查询IO次数变少。

所有的查询都需要到达叶子结点，查询性能稳定。

所有的叶子结点构成了一个有序链表，更方便快速查找。这一点是非常明确的。

4.所有排序的比较

冒泡排序：从数组中的第一个数开始，依次遍历，相邻比较交换。每一次都找出未排序序列中的最大数，并且冒泡至序列的顶端。

插入排序：涉及到多个变量的协作处理，logic是简单的。

希尔排序：

选择排序：选择最小的和第一个位置交换

快排、堆排序、归并排序、

计数排序（比元素x小的元素有n个，则元素x的位置应当为n+1）

O(n+k)

桶排序：也算是稍微比较高级的算法了，但是实际上并不是特别的复杂哦。

计数排序会开辟一个n+k的数组空间，这个n+k的大小，就是原数组中的最大值，这一点是明确的。