第八章 查找和排序

刘杰

人工智能学院

南開大學 Nankai University

查找

- 所有程序中最基本、最常用的操作之一
- 当查找的对象是一个庞大数量的数据集合中的元素时，查找的方法和效率就格外重要

主要内容

- 顺序表、有序表、树表和哈希表查找的各种实现方法
- 各种查找方法在等概率情况下的平均查找长度

查找的概念

- 查找表(Search Table)：相同类型的数据元素(对象)组成的集合，每个元素通常由若干数据项构成。
- 关键字(Key, 码)：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素。若关键字能唯一标识一个数据元素，则关键字称为主关键字；将能标识若干个数据元素的关键字称为次关键字。
- 查找/检索(Searching)：根据给定的K值，在查找表中确定一个关键字等于给定值的记录或数据元素。
  - 查找表中存在满足条件的记录：查找成功；结果：所查到的记录信息或记录在查找表中的位置。
  - 查找表中不存在满足条件的记录：查找失败。

查找的概念——基本形式

- 静态查找(Static Search)：在查找时只对数据元素进行查询或检索，查找表称为静态查找表。
- 动态查找(Dynamic Search)：在实施查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

查找的对象是查找表，采用何种查找方法，首先取决于查找表的组织。查找表是记录的集合，而集合中的元素之间是一种完全松散的关系，因此，查找表是一种非常灵活的数据结构，可以用多种方式来存储。

查找的概念——查找方法

根据存储结构不同分为三种：

- 顺序表和链表的查找：将给定的K值与查找表中记录的关键字逐个进行比较， 找到要查找的记录
- 散列表的查找：根据给定的K值直接访问查找表， 从而找到要查找的记录
- 索引查找表的查找：首先根据索引确定待查找记录所在的块 ， 然后再从块中找到要查找的记录。

查找的概念——查找方法

评价指标：

- 查找过程中主要操作是关键字的比较，查找过程中关键字的平均比较次数(平均查找长度ASL：Average Search Length)作为衡量一个查找算法效率高低的标准。ASL定义为：

$$\mathbf{ASL}=\sum_{i=1}^{n} P_i C_i$$

n为查找表中记录个数，$\sum_{i=1}^{n} P_i = 1$

查找的概念——评价指标

$$\mathbf{ASL}=\sum_{i=1}^{n} P_i C_i$$

- n为查找表中记录个数，$\sum_{i=1}^{n} P_i = 1$；
- P_i：查找第i个记录的概率，不失一般性，认为查找每个记录的概率相等，即$P_1=P_2=\ldots=P_n=\frac{1}{n}$；
- C_i：查找第i个记录需要进行比较的次数

查找的概念——评价指标

一般地，认为记录的关键字是一些可以进行比较运算的类型，如整型、字符型、实型等，本章以后各节中讨论所涉及的关键字、数据元素等的类型描述如下：

- 典型的关键字类型说明是：

```
typedef float  KeyType;    /* 实型 */
typedef int  KeyType;    /* 整型 */
typedef char KeyType;    /* 字符串型 */
```


查找的概念——评价指标

- 数据元素类型的定义是：

```
typedef struct RecType {
  KeyType key;      /* 关键字码 */
  ┇                 /* 其他域 */
}RecType;
```

- 对两个关键字的比较约定为如下带参数的宏定义：

```
  /* 对数值型关键字 */
#define EQ(a, b)  ((a)==(b))
#define LT(a, b)  ((a)<(b))
#define LQ(a, b)  ((a)<=(b))
  /* 对字符串型关键字 */
#define EQ(a, b)  (!strcmp((a), (b)) )
#define LT(a, b)  (strcmp((a), (b))<0 )
#define LQ(a, b)  (strcmp((a), (b))<=0 )
```


顺序查找

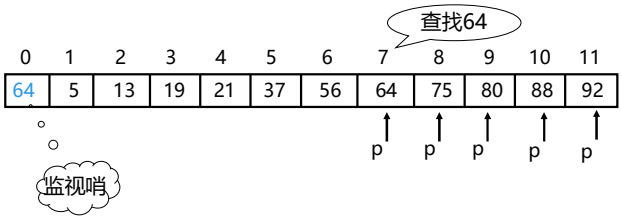
- 查找思想

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败。顺序表的类型定义如下：

顺序查找——查找思想

```
#define MAX_SIZE  100
typedef struct SSTable {
   RecType elem[MAX_SIZE];   /* 顺序表 */
   int length;    /* 实际元素个数 */
}SSTable;
int Seq_Search(SSTable ST, KeyType key) {
   int p;
   ST. elem[0].key=key;   /* 设置监视哨兵,失败返回0 */
   for (p=ST.length; !EQ(ST. elem[p].key, key); p--)
      return p;
}
   比较次数:
      查找第n个元素:    1
      ..........
      查找第i个元素:    n-i+1
      查找第1个元素:    n
      查找失败:         n+1
```

顺序查找——查找思想



比较次数=5

顺序查找示例

顺序查找

- 算法分析

不失一般性，设查找每个记录成功的概率相等，即$P_i=\frac{1}{n}$；查找第i个元素成功的比较次数C_i=n-i+1；

查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{i=1}^{n}(n-i+1)=\frac{n+1}{2}$$

折半查找

- 折半查找又称为二分查找，是一种效率较高的查找方法。
- 前提条件：查找表中的所有记录是按关键字有序(升序或降序) 。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

折半查找

- 查找思想

用Low、High和Mid表示待查找区间的下界、上界和中间位置指针，初值为Low=1，High=n。

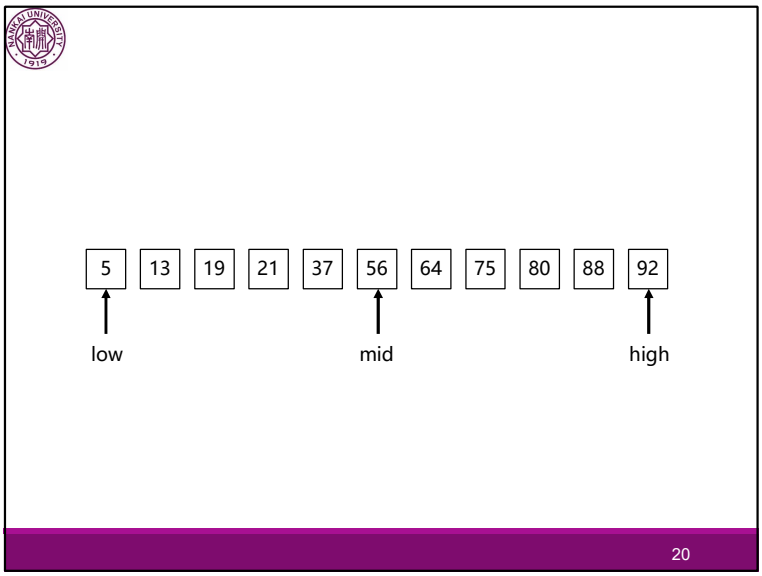
(1) 取中间位置Mid：Mid=(Low+High)/2；
(2) 比较中间位置记录的关键字与给定的K值：
① 相等： 查找成功；
② 大于：待查记录在区间的前半段，修改上界指针：High=Mid-1，转(1)；
③ 小于：待查记录在区间的后半段，修改下界指针：Low=Mid+1，转(1)；
直到越界(Low>High)，查找失败。

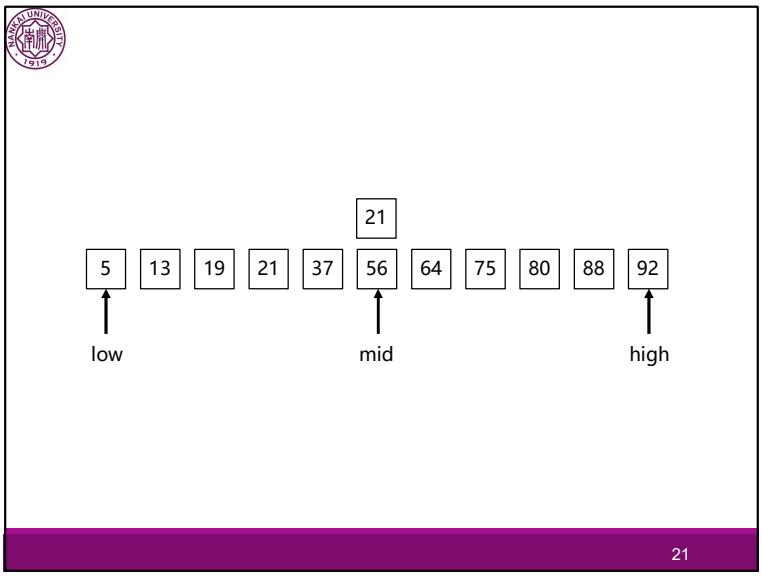
折半查找

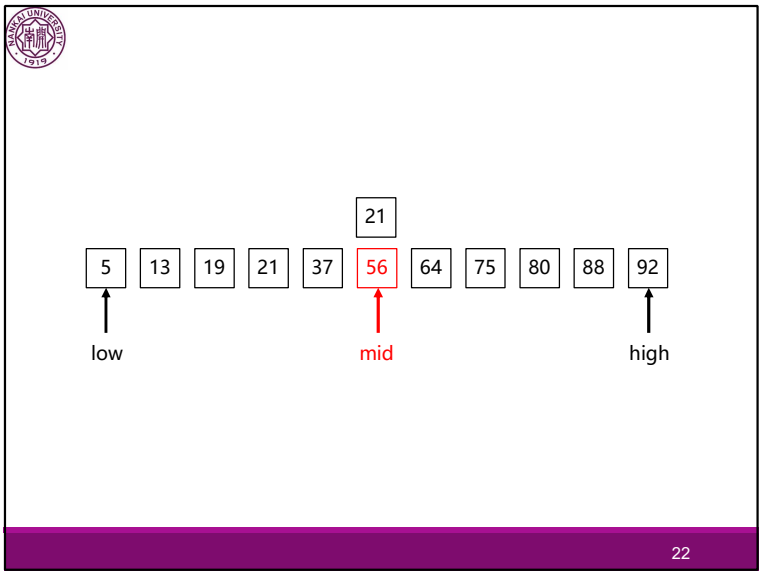
```
int Bin_Search(SSTable ST , KeyType key) {
  int Low=1, High=ST.length, Mid ;
  while (Low<High) {
    Mid=(Low+High)/2 ;
    if (EQ(ST. elem[Mid].key, key)) return Mid ;
    else if (LT(ST. elem[Mid].key, key)) Low=Mid+1 ;
    else High=Mid-1 ;
  }
  return 0 ;    /* 查找失败 */
}
```

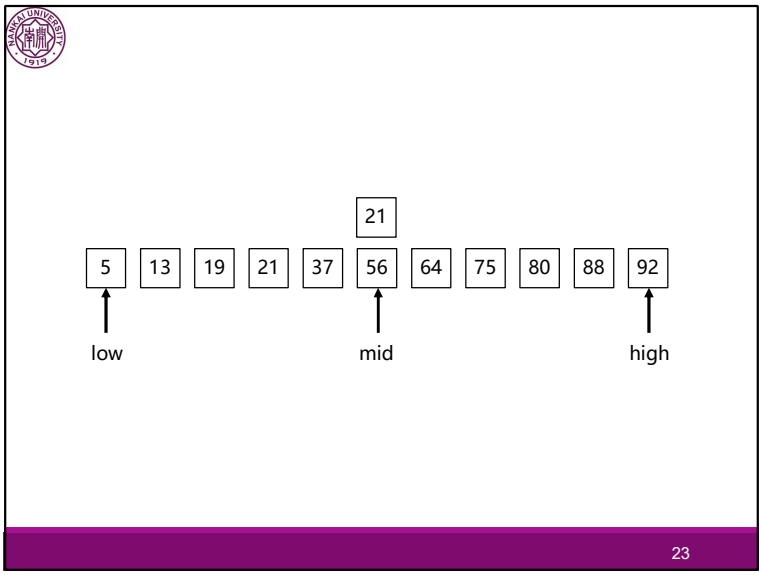

算法示例

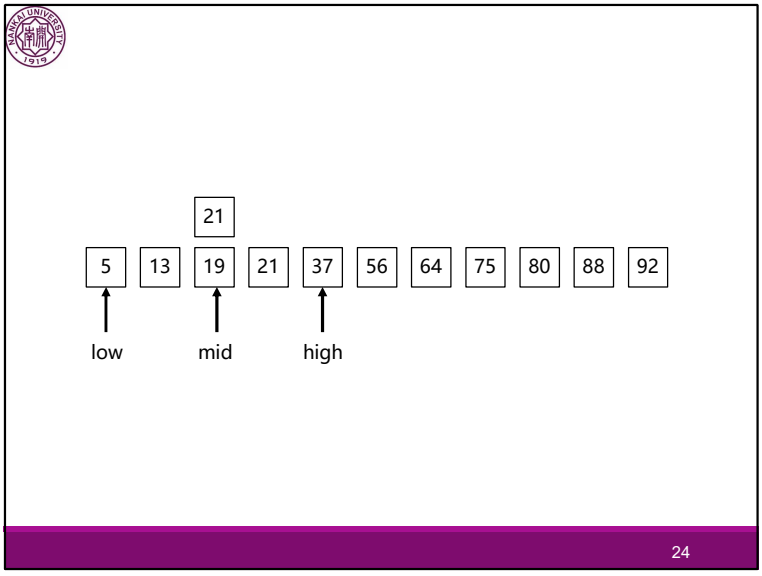
在[5, 13, 19, 21, 37, 56, 64, 75, 80, 88, 92]中查找21

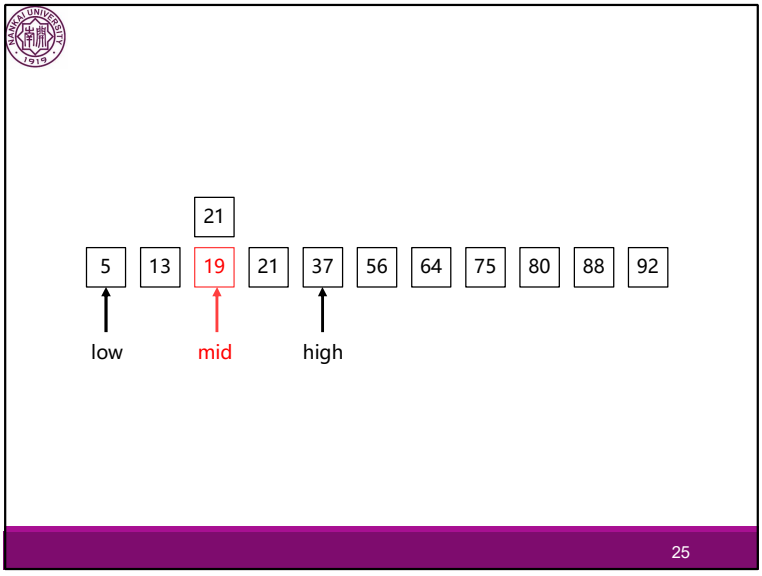

5 13 19 21 37 56 64 75 80 88 92
low
mid
high
20

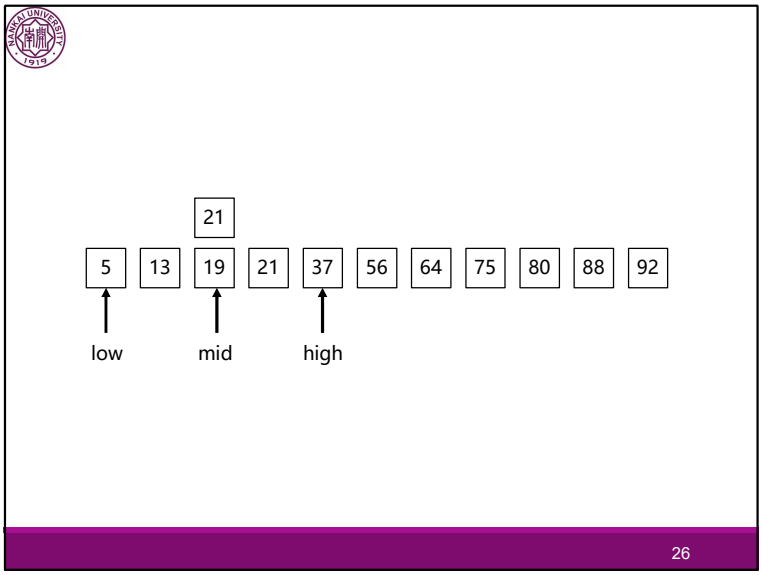

21
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
21

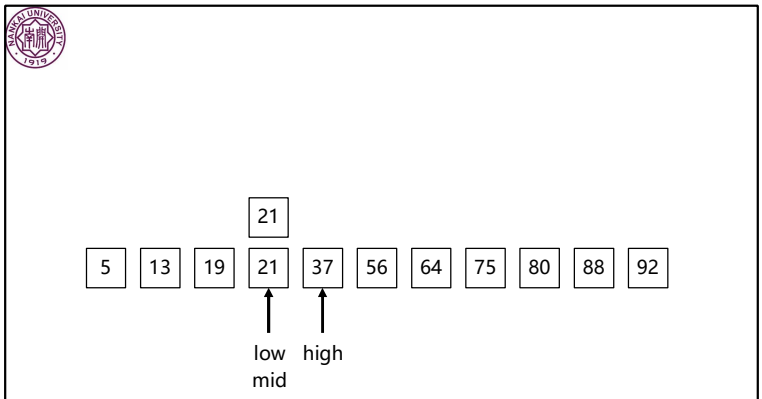

21
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
22

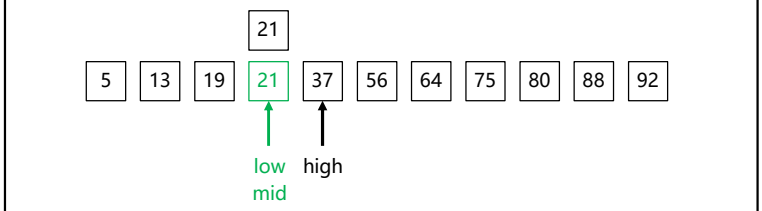

21
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
23


21
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
24


21
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
25

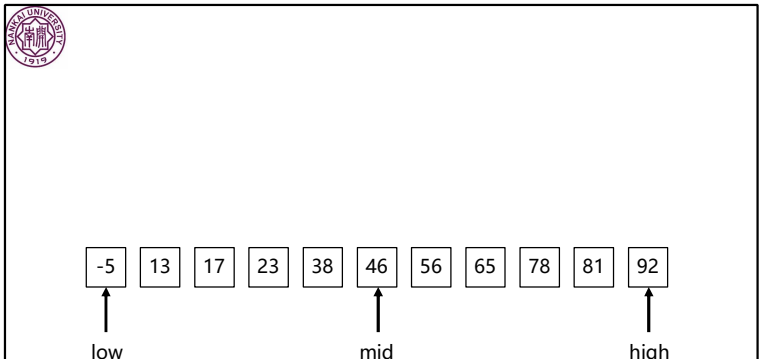


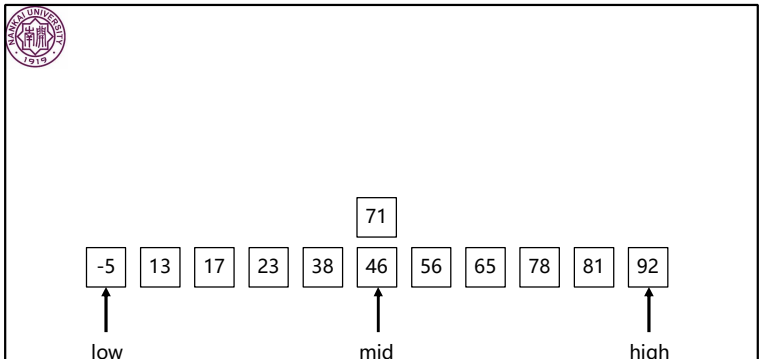


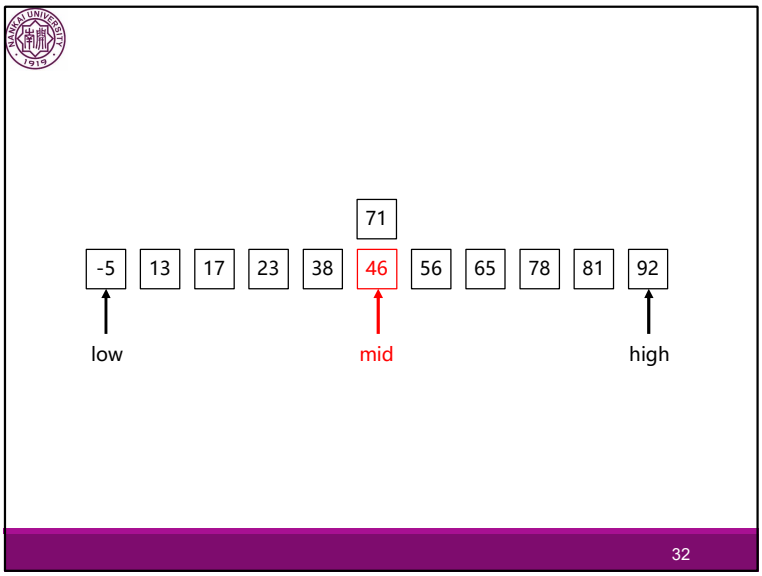


算法示例

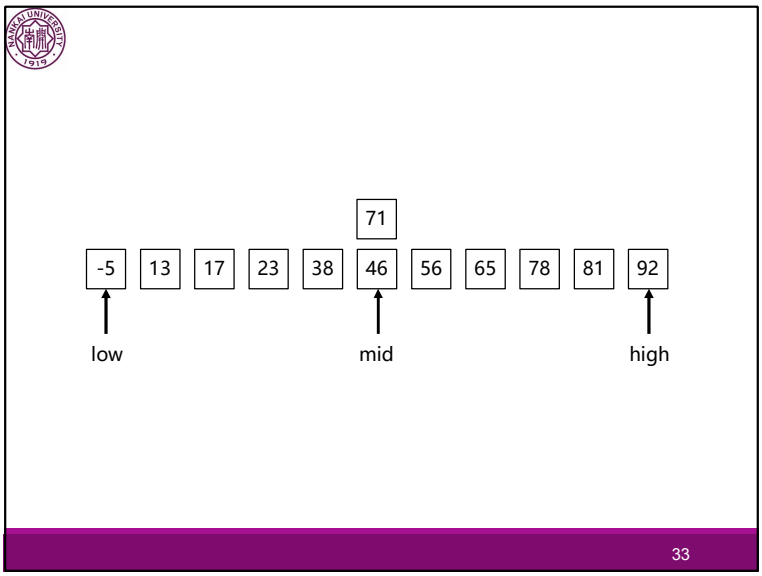
在[-5, 13, 17, 23, 38, 46, 56, 65, 78, 81, 92]中查找71



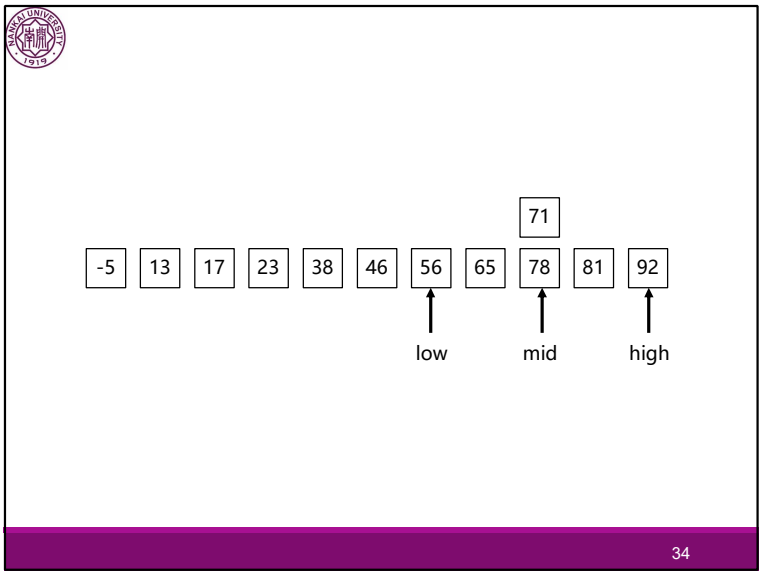




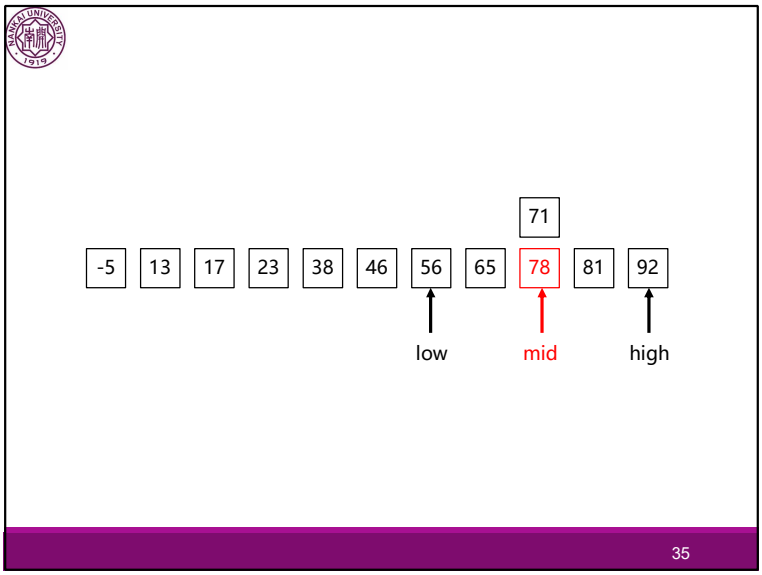
32



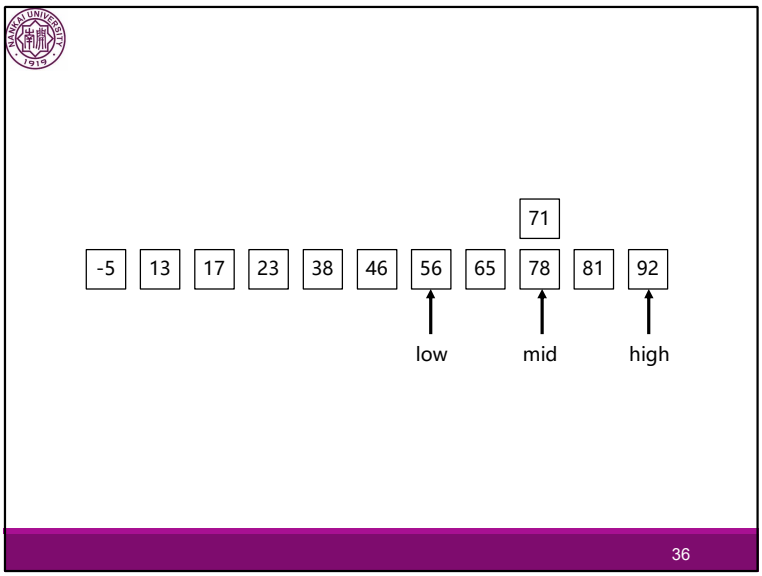
33



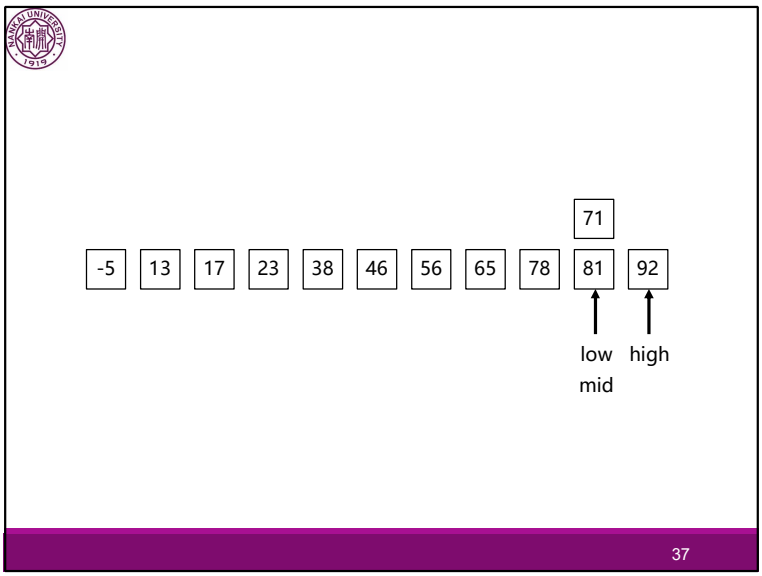
34



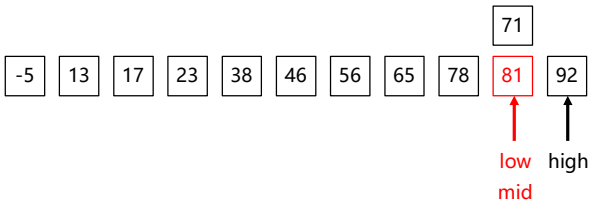
35



36



37

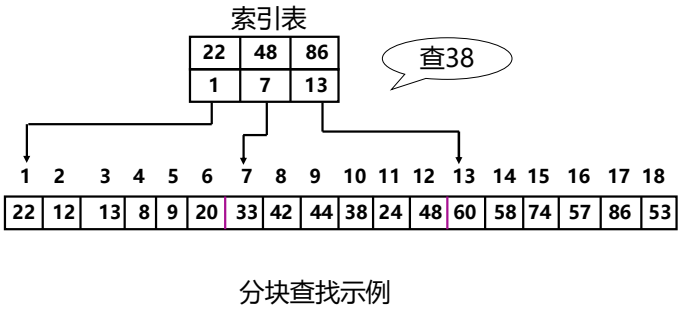


分块查找

- 分块查找(Blocking Search)又称索引顺序查找，是前面两种查找方法的综合。
- 查找表的组织
  - 将查找表分成几块。块间有序，即第i+1块的所有记录关键字均大于(或小于)第i块记录关键字；块内无序。
  - 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

最大关键字
起始指针

分块查找



分块查找示例

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。

利用树的形式组织查找表，可以对查找表进行动态高效的查找。

二叉排序树（BST）的定义

- 二叉排序树(Binary Sort Tree或Binary Search Tree)：二叉排序树或者是空树，或者是满足下列性质的二叉树。
  - 若左子树不为空，则左子树上所有结点的值(关键字)都小于根结点的值；
  - 若右子树不为空，则右子树上所有结点的值(关键字)都大于根结点的值；
  - 左、右子树都分别是二叉排序树。

二叉排序树（BST）的定义

- 结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。
- BST仍然可以用二叉链表来存储。

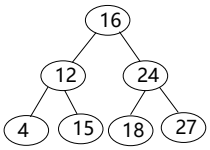
二叉排序树（BST）的定义

- 结点类型定义如下：

```
typedef  struct  Node {
    KeyType  key ;   /*  关键字域  */
    ...      /*  其它数据域  */
    struct  Node  *Lchild , *Rchild ;
}BSTNode ;
```


结点类型定义如下：

```
typedef  struct  Node {
    KeyType  key ;   /*  关键字域  */
    ...      /*  其它数据域  */
    struct  Node  *Lchild , *Rchild ;
}BSTNode ;
```



二叉排序树

BST树的查找

1 查找思想

首先将给定的K值与二叉排序树的根结点的关键字进行比较：

- 若相等： 则查找成功；
- 给定的K值小于BST的根结点的关键字：继续在该结点的左子树上进行查找；
- 给定的K值大于BST的根结点的关键字：继续在该结点的右子树上进行查找。

```
BSTNode *BST_Serach(BSTNode *T , KeyType key) {
    if (T==NULL)  return NULL;
    else {
        if  (EQ(T->key, key) )  return T;
        else if ( LT(key, T->key))
            return BST_Serach(T->Lchild, key);
        else  return BST_Serach(T->Rchild, key);
    }
}
```

BST树的插入

在BST树中插入一个新结点，要保证插入后仍满足BST的性质。

1 插入思想

在BST树中插入一个新结点x时，若BST树为空，则令新结点x为插入后BST树的根结点；否则，将结点x的关键字与根结点T的关键字进行比较：

- 若相等： 不需要插入；
- 若x.key < T->key：结点x插入到T的左子树中；
- 若x.key > T->key：结点x插入到T的右子树中。

由算法知，每次插入的新结点都是BST树的叶子结点，即在插入时不必移动其它结点，仅需修改某个结点的指针。

```
void  Insert_BST (BSTNode *T , KeyType  key) {
    BSTNode *x ;
    x=(BSTNode *)malloc(sizeof(BSTNode)) ;
    X->key=key; x->Lchild=x->Rchild=NULL ;
    if (T==NULL)  T=x ;
    else {
        if (EQ(T->key, x->key) ) return  ; /*  已有结点  */
        else if (LT(x->key, T->key) )
            Insert_BST(T->Lchild, key) ;
        else   Insert_BST(T->Rchild, key) ;
    }
}
```

哈希(散列)查找

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

例 30个地区的各民族人口统计表

编号	省、市(区)	总人口	汉族	回族	.....
1	北京				
2	上海				
.....	.....				

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1， H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19 H(Shenyang)=19

基本概念

- 哈希函数：在记录的关键字与记录的存储地址之间建立的一种对应关系叫哈希函数。
- 哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。可写成：$addr(a_i) = H(k_i)$，其中i是表中一个元素，$addr(a_i)$是a_i的地址，k_i是a_i的关键字。
- 哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表。
- 哈希查找(又叫散列查找)：利用哈希函数进行查找的过程叫哈希查找。

冲突：对于不同的关键字k_i、k_j，若$k_i \neq kj$，但$H(k_i) = H(k_j)$的现象叫冲突(collision) 。

同义词：具有相同函数值的两个不同的关键字，称为该哈希函数的同义词。

哈希函数通常是一种压缩映象，所以冲突不可避免，只能尽量减少；当冲突发生时，应该有处理冲突的方法。设计一个散列表应包括：

- 散列表的空间范围，即确定散列函数的值域；
- 构造合适的散列函数，使得对于所有可能的元素(记录的关键字)，函数值均在散列表的地址空间范围内，且出现冲突的可能尽量小；
- 处理冲突的方法。即当冲突出现时如何解决。

内部排序

在信息处理过程中，最基本的操作是查找。从查找来说，效率最高的是折半查找，折半查找的前提是所有的数据元素(记录)是按关键字有序的。需要将一个无序的数据文件转变为一个有序的数据文件。

将任一文件中的记录通过某种方法整理成为按(记录)关键字有序排列的处理过程称为排序。

排序是数据处理中一种最常用的操作。

排序的基本概念

(1) 排序(Sorting)

排序是将一批(组)任意次序的记录重新排列成按关键字有序的记录序列的过程，其定义为：

给定一组记录序列：$\{R_1, R_2, ..., R_n\}$，其相应的关键字序列是$\{K_1, K_2, ..., K_n\}$。确定1, 2, ... n的一个排列$p_1, p_2, ..., p_n$，使其相应的关键字满足如下非递减(或非递增)关系：$K_{p_1} \leq K_{p_2} \leq ... \leq K_{p_n}$的序列$\{K_{p_1}, K_{p_2}, ..., K_{p_n}\}$，这种操作称为排序。

关键字K_i可以是记录R_i的主关键字，也可以是次关键字或若干数据项的组合。

K_i是主关键字：排序后得到的结果是唯一的；

K_i是次关键字：排序后得到的结果是不唯一的

(2) 排序的稳定性

若记录序列中有两个或两个以上关键字相等的记录：$K_i = K_j (i \neq j,\ i, j = 1, 2, \ldots n)$，且在排序前$R_i$先于$R_j (i < j)$，排序后的记录序列仍然是$R_i$先于$R_j$，称排序方法是稳定的，否则是不稳定的。

排序算法有许多，但就全面性能而言，还没有一种公认为最好的。每种算法都有其优点和缺点，分别适合不同的数据量和硬件配置。

评价排序算法的标准有：执行时间和所需的辅助空间，其次是算法的稳定性。

若排序算法所需的辅助空间不依赖问题的规模n，即空间复杂度是O(1)，则称排序方法是就地排序，否则是非就地排序。

(3) 排序的分类

待排序的记录数量不同，排序过程中涉及的存储器的不同，有不同的排序分类。

- 待排序的记录数不太多：所有的记录都能存放在内存中进行排序，称为内部排序；
- 待排序的记录数太多：所有的记录不可能存放在内存中， 排序过程中必须在内、外存之间进行数据交换，这样的排序称为外部排序。

(4) 内部排序的基本操作

对内部排序地而言，其基本操作有两种：

- 比较两个关键字的大小；
- 存储位置的移动：从一个位置移到另一个位置。

第一种操作是必不可少的；而第二种操作却不是必须的，取决于记录的存储方式，具体情况是：

① 记录存储在一组连续地址的存储空间：记录之间的逻辑顺序关系是通过其物理存储位置的相邻来体现，记录的移动是必不可少的；

② 记录采用链式存储方式：记录之间的逻辑顺序关系是通过结点中的指针来体现，排序过程仅需修改结点的指针，而不需要移动记录；

③ 记录存储在一组连续地址的存储空间：构造另一个辅助表来保存各个记录的存放地址(指针)：排序过程不需要移动记录，而仅需修改辅助表中的指针，排序后视具体情况决定是否调整记录的存储位置。

①比较适合记录数较少的情况；而②、③则适合记录数较多的情况。

为讨论方便，假设待排序的记录是以①的情况存储，且设排序是按升序排列的；关键字是一些可直接用比较运算符进行比较的类型。

插入排序

采用的是以 “玩桥牌者” 的方法为基础的。即在考察记录R_i之前，设以前的所有记录$R_1, R_2, \ldots, R_{i-1}$已排好序，然后将$R_i$插入到已排好序的诸记录的适当位置。

最基本的插入排序是直接插入排序(Straight Insertion Sort)。

直接插入排序

1 排序思想

将待排序的记录R_i，插入到已排好序的记录表$R_1, R_2, \ldots, R_{i-1}$中，得到一个新的、记录数增加1的有序表。 直到所有的记录都插入完为止。

设待排序的记录顺序存放在数组$R[1 \ldots n]$中，在排序的某一时刻，将记录序列分成两部分：

- $R[1 \ldots i-1]$：已排好序的有序部分；
- $R[i \ldots n]$：未排好序的无序部分。

显然，在刚开始排序时，$R[1]$是已经排好序的。

算法示例

设有关键字序列为：7, 4, -2, 19, 13, 6，利用直接插入排序按照升序排列

62

7 4 -2 19 13 6

63

7 4 -2 19 13 6

64

7 4 -2 19 13 6

65

7 4 -2 19 13 6

66

4 7 -2 19 13 6

67

4 7 -2 19 13 6

68

4 7 -2 19 13 6

69

-2 4 7 19 13 6

70

-2 4 7 19 13 6

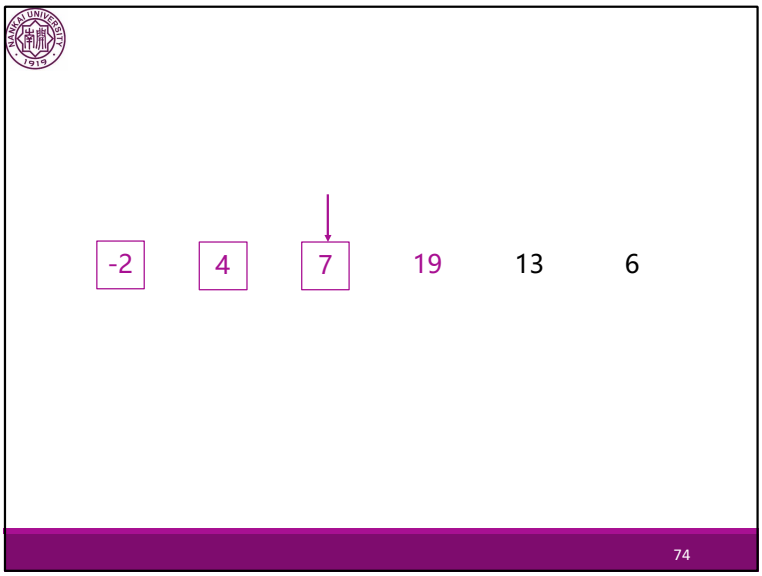
71

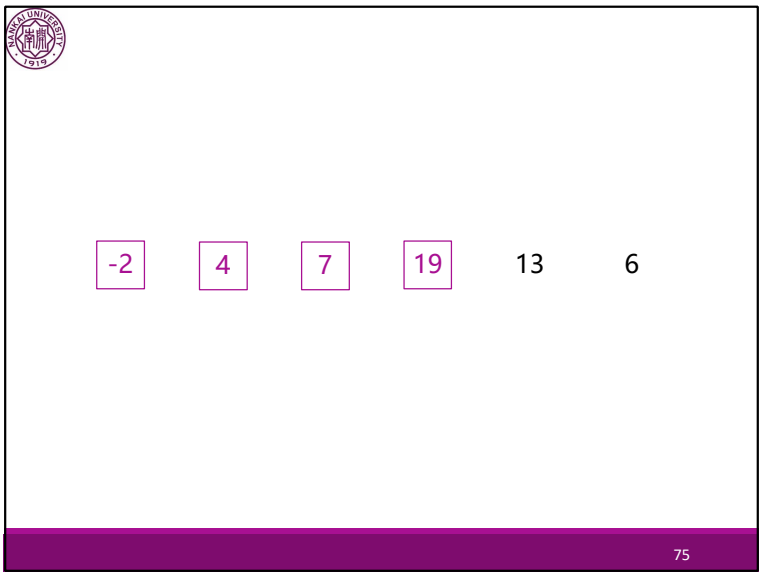
-2 4 7 19 13 6

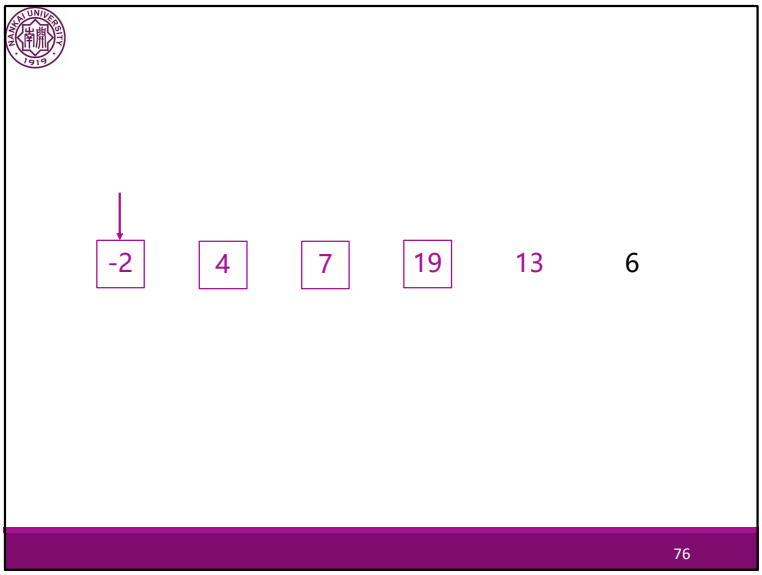
72

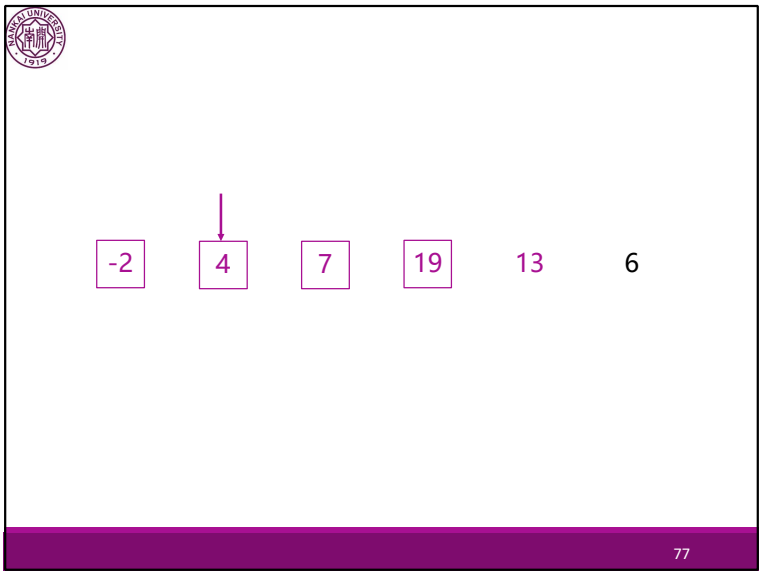
-2 4 7 19 13 6

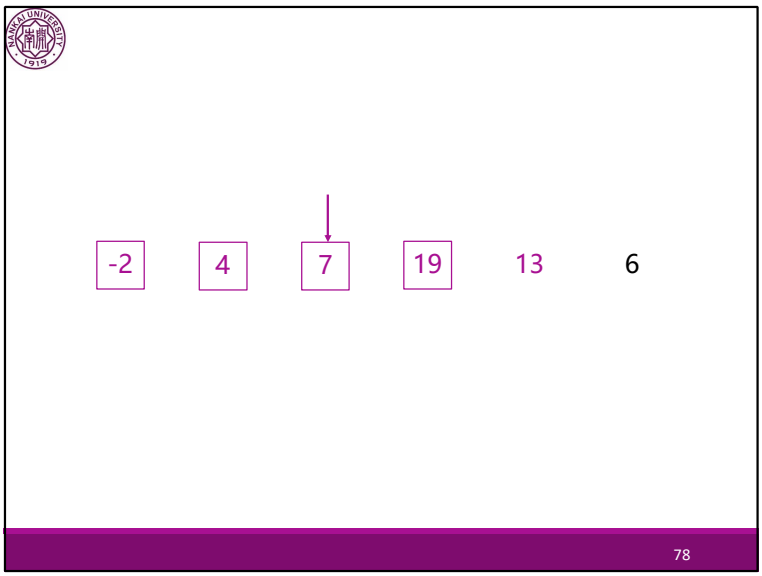
73

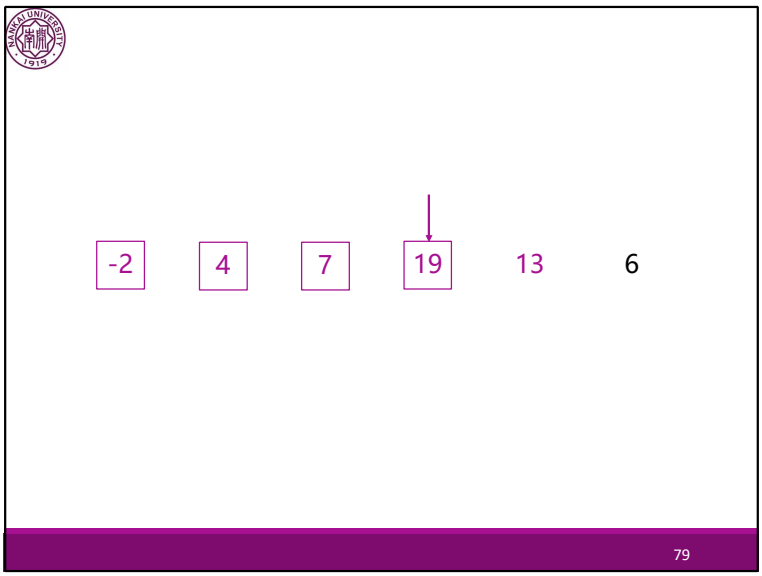

-2
4
7
19
13
6
74

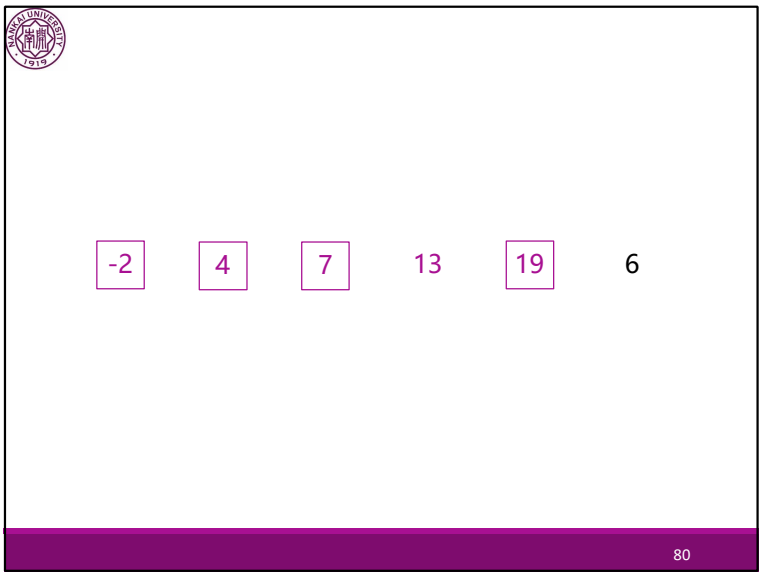

-2
4
7
19
13
6
75

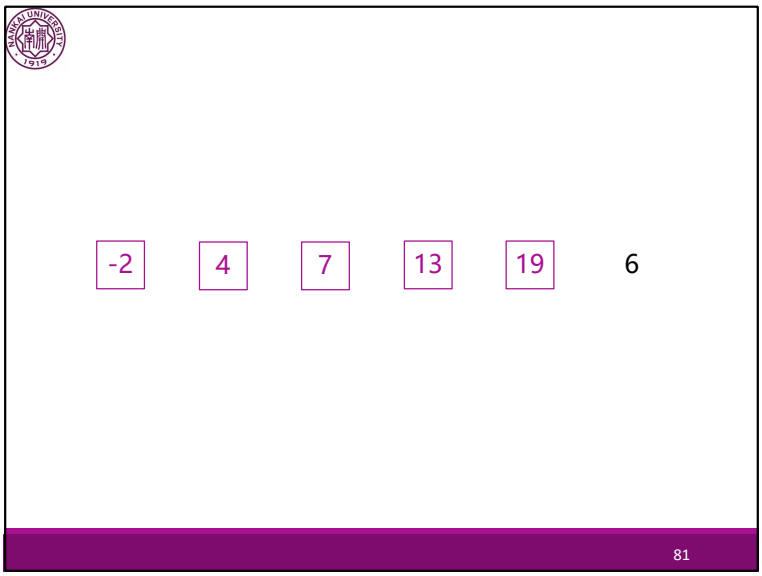

-2
4
7
19
13
6
76

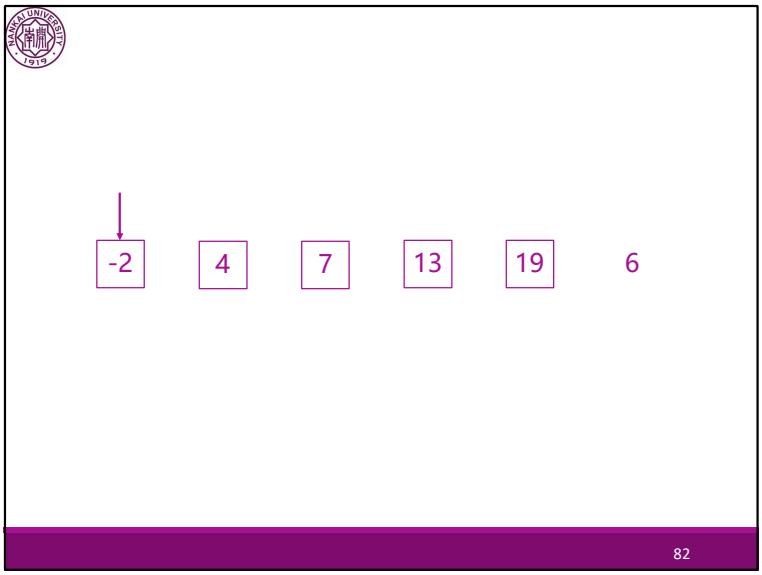

-2
4
7
19
13
6
77

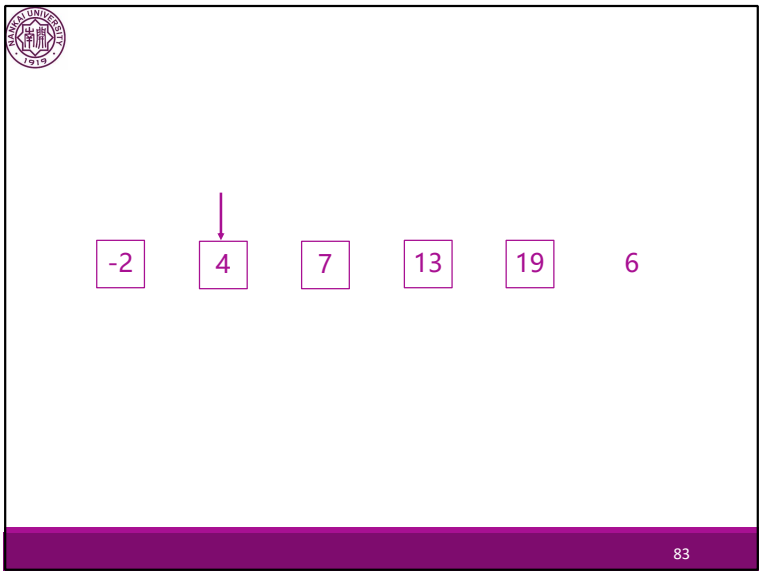

-2
4
7
19
13
6
78

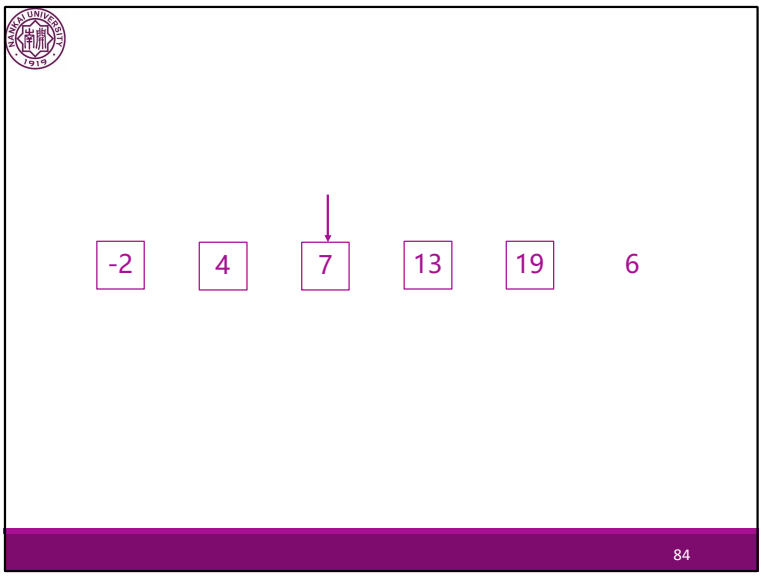

-2
4
7
19
13
6
79

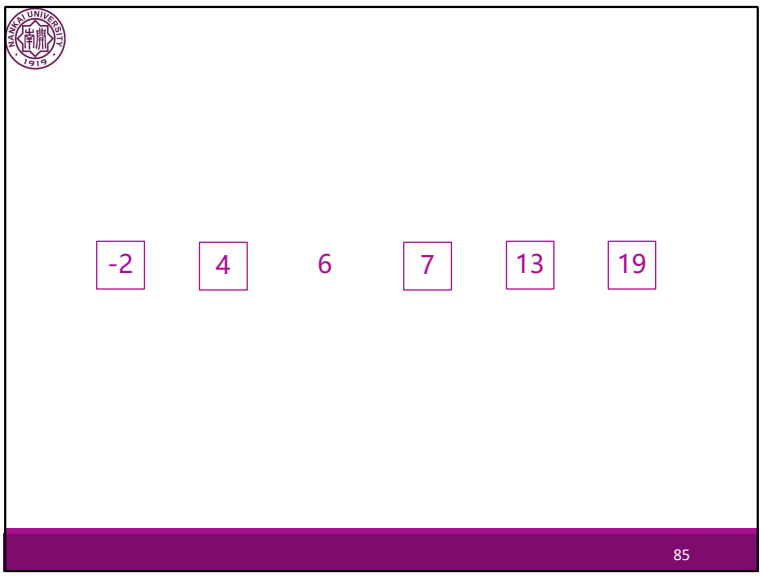

-2
4
7
13
19
6
80


-2
4
7
13
19
6
81


-2
4
7
13
19
6
82


-2
4
7
13
19
6
83


-2
4
7
13
19
6
84


-2
4
6
7
13
19
85

-2 4 6 7 13 19

-2 4 6 7 13 19

其它插入排序

折半插入排序

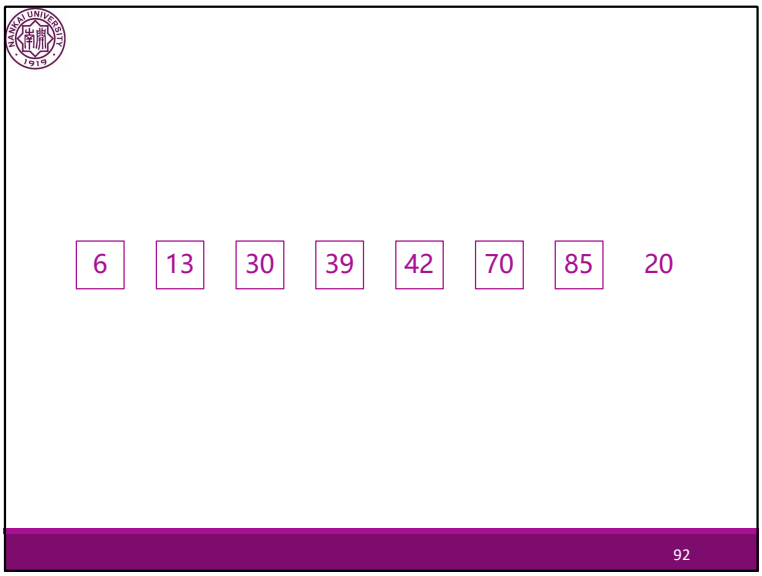
当将待排序的记录R[i] 插入到已排好序的记录子表R[1...i-1]中时，由于$R_1, R_2, ..., R_{i-1}$已排好序，则查找插入位置可以用“折半查找”实现，则直接插入排序就变成为折半插入排序。

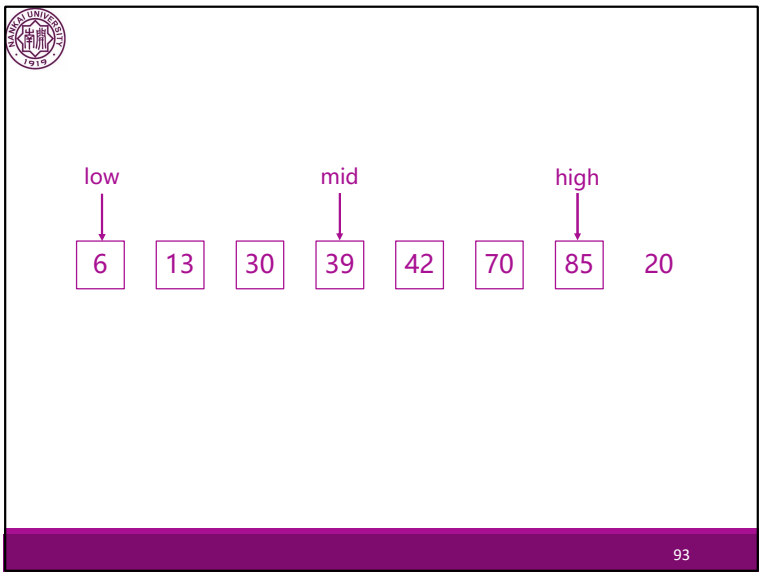
算法示例

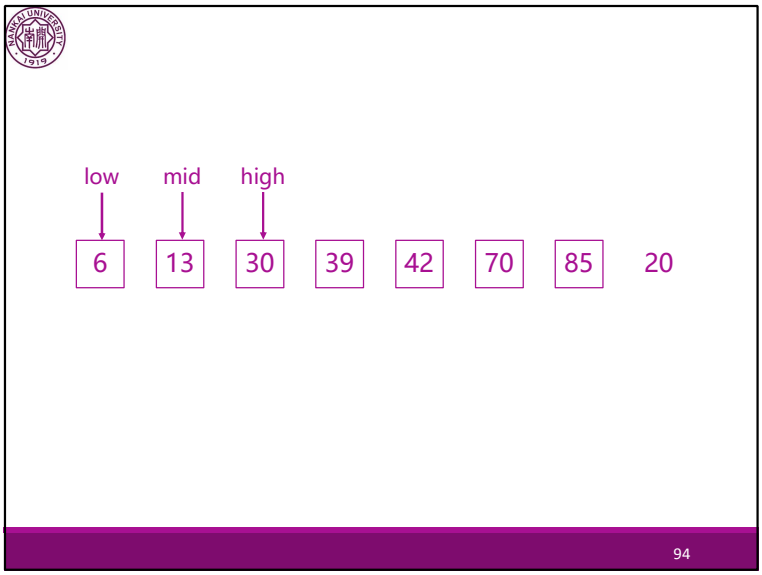
设有关键字序列为：30, 13, 70, 85, 39, 42, 6, 20，利用折半插入排序按照升序排列

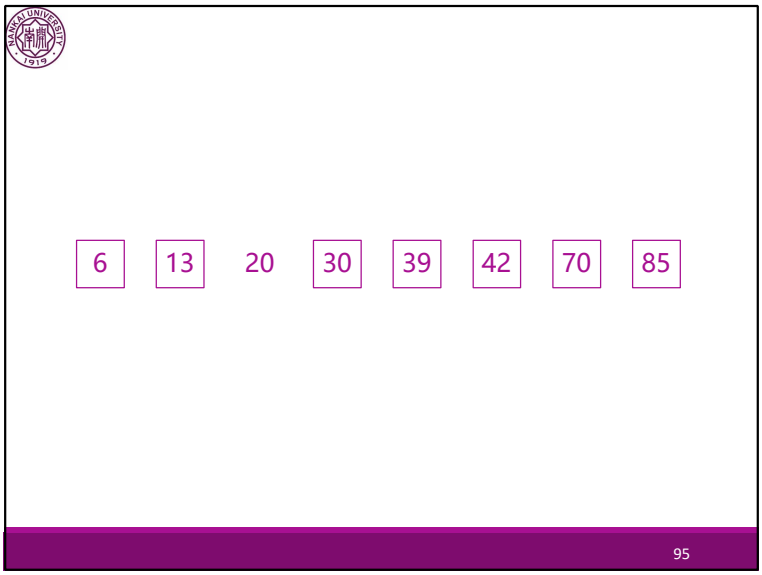
30 13 70 85 39 42 6 20

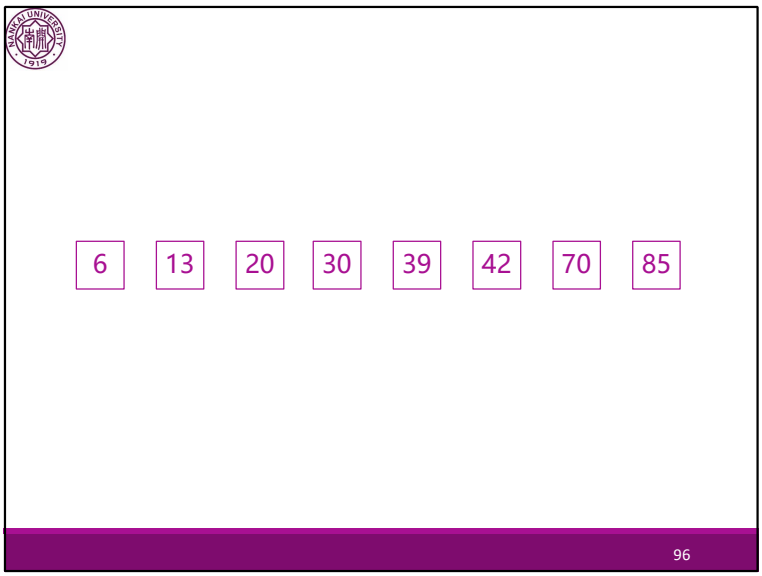
6 13 30 39 42 70 85 20

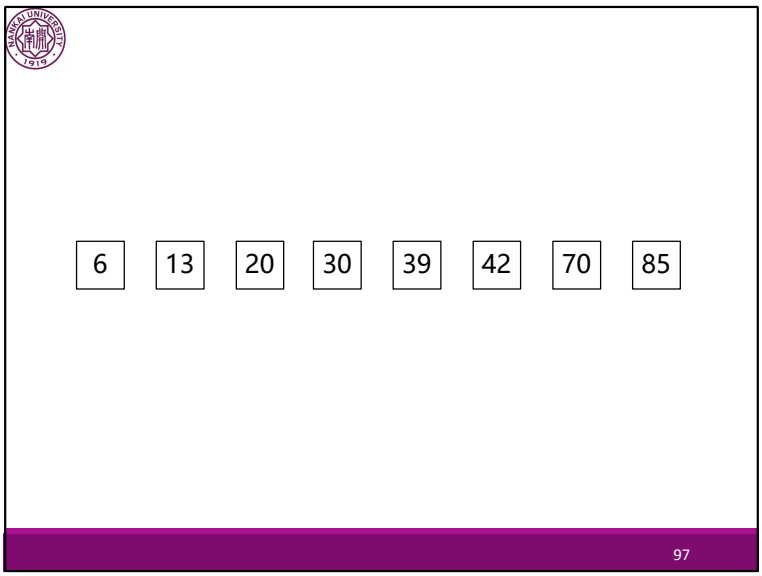

6 13 30 39 42 70 85 20
92


low
mid
high
6 13 30 39 42 70 85 20
93


low
mid
high
6 13 30 39 42 70 85 20
94


6 13 20 30 39 42 70 85
95


6 13 20 30 39 42 70 85
96


6 13 20 30 39 42 70 85
97

快速排序

是一类基于交换的排序，系统地交换反序的记录的偶对，直到不再有这样一来的偶对为止。其中最基本的是冒泡排序(Bubble Sort)。

冒泡排序

1 排序思想

依次比较相邻的两个记录的关键字，若两个记录是反序的(即前一个记录的关键字大于后前一个记录的关键字)，则进行交换，直到没有反序的记录为止。

① 首先将L->R[1]与L->R[2]的关键字进行比较，若为反序(L->R[1]的关键字大于L->R[2]的关键字)，则交换两个记录；然后比较L->R[2]与L->R[3]的关键字，依此类推，直到L->R[n-1]与L->R[n]的关键字比较后为止，称为一趟冒泡排序，L->R[n]为关键字最大的记录。

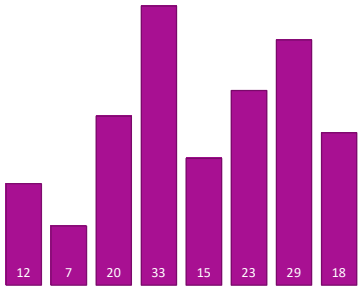
② 然后进行第二趟冒泡排序，对前n-1个记录进行同样的操作。

算法示例

设有关键字序列为： 12, 7, 20, 33, 15, 23, 29, 18，利用冒泡插入排序按照升序排列

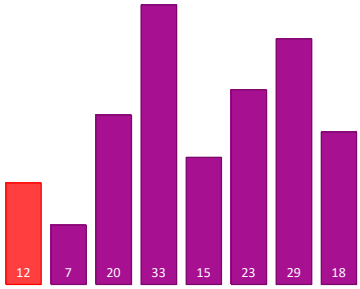
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



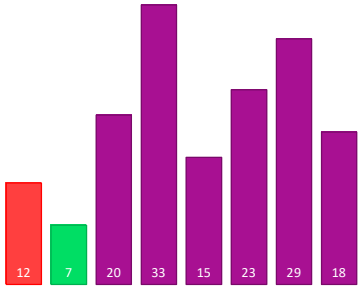
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



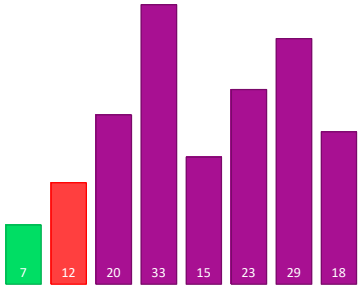
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



示例：冒泡排序

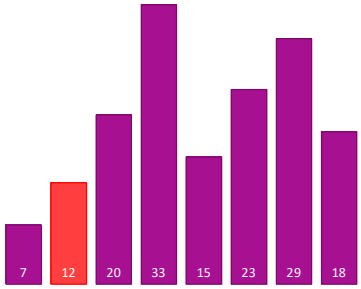
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



104

示例：冒泡排序

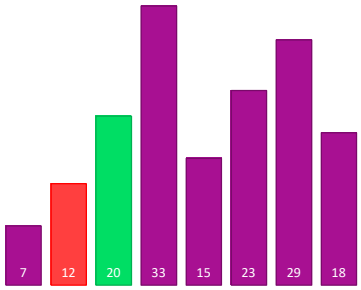
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



105

示例：冒泡排序

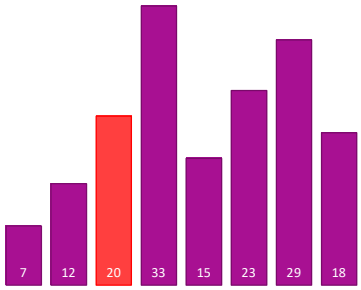
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



106

示例：冒泡排序

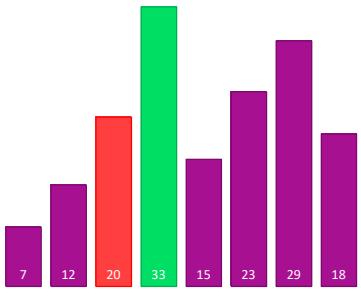
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



107

示例：冒泡排序

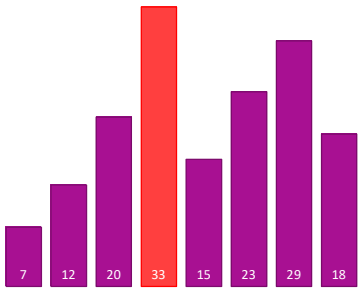
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



108

示例：冒泡排序

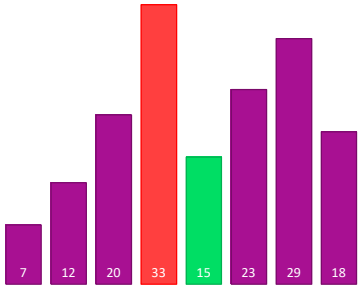
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



109

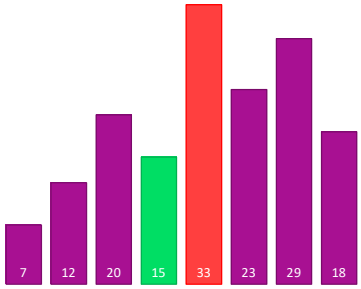
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



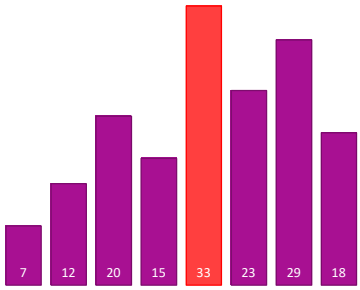
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



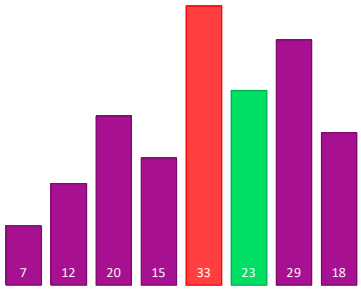
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



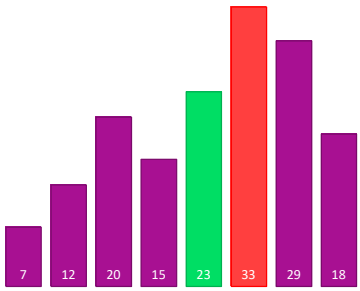
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



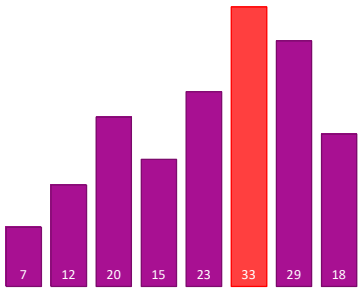
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



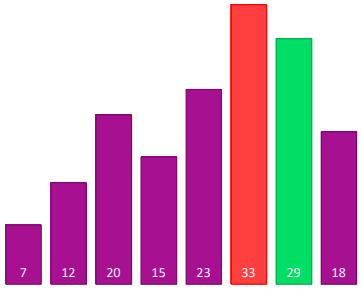
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



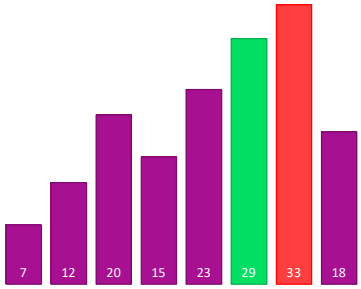
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



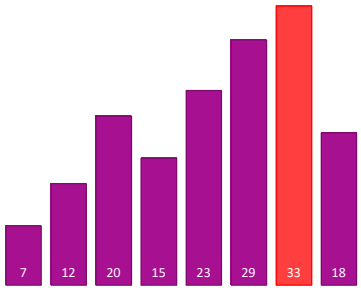
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



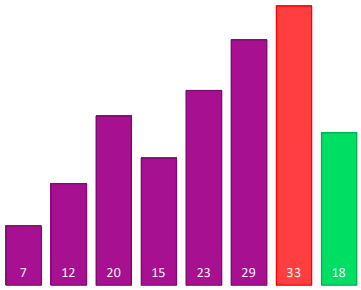
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



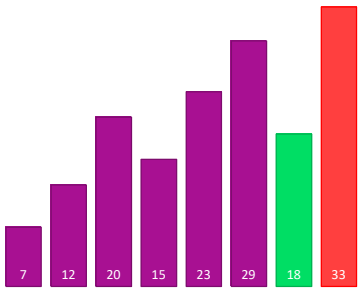
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



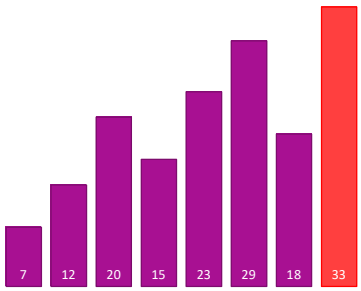
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



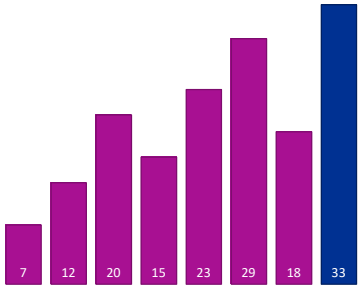
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



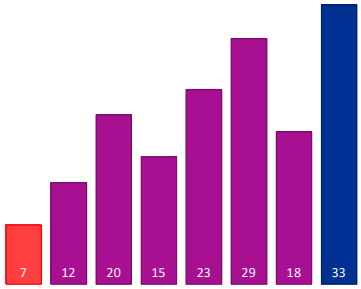
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



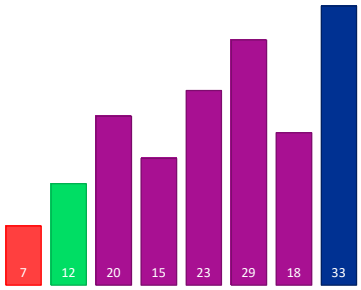
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



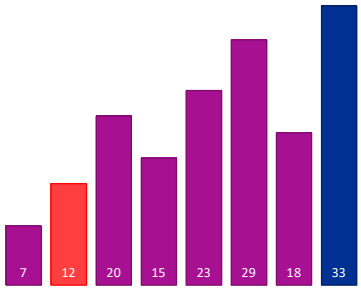
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



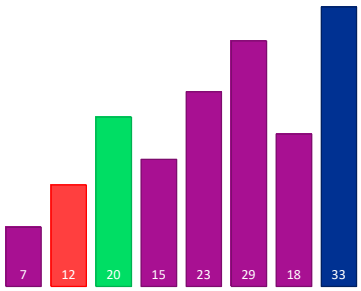
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



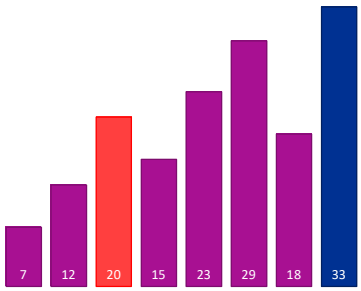
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



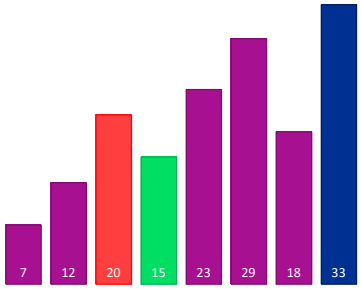
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



示例：冒泡排序

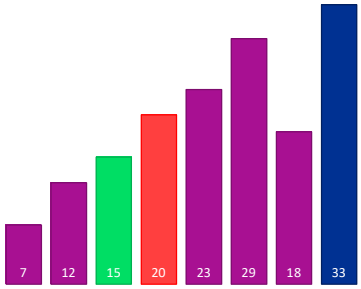
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



128

示例：冒泡排序

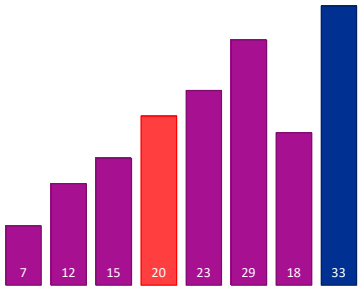
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



129

示例：冒泡排序

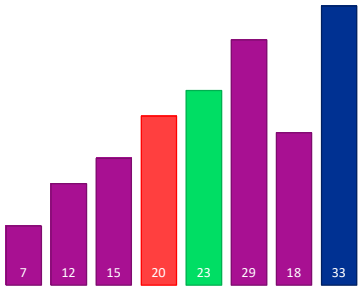
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



130

示例：冒泡排序

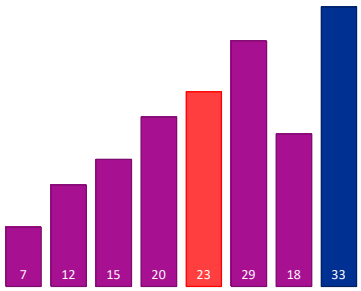
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



131

示例：冒泡排序

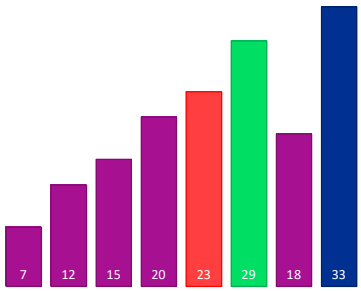
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



132

示例：冒泡排序

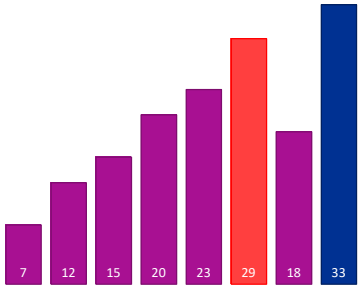
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



133

示例：冒泡排序

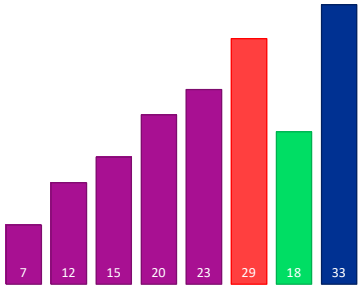
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



134

示例：冒泡排序

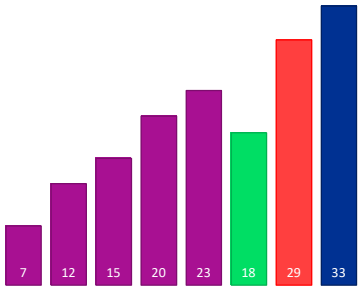
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



135

示例：冒泡排序

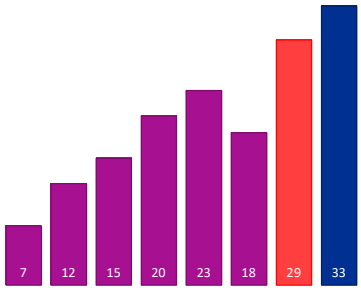
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



136

示例：冒泡排序

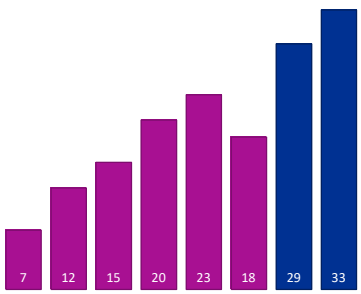
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



137

示例：冒泡排序

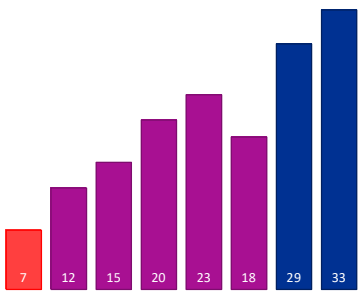
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



138

示例：冒泡排序

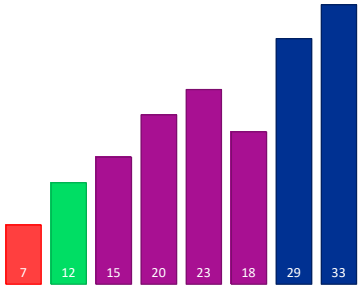
- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



139

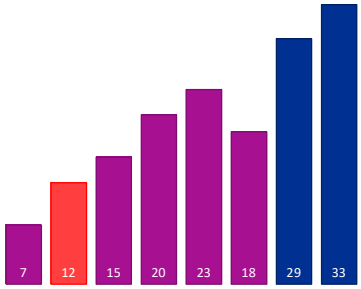
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



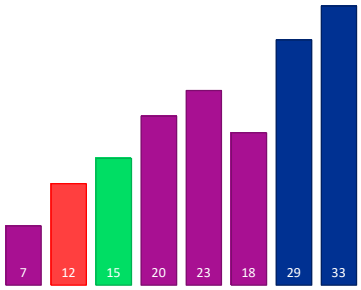
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



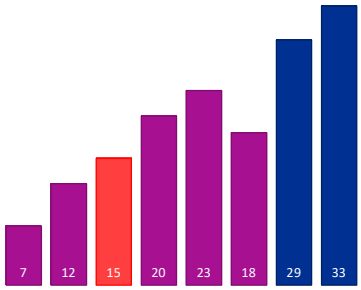
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



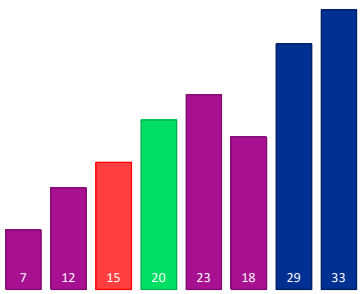
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



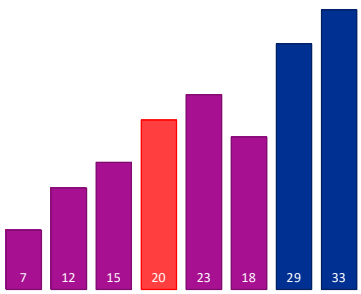
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



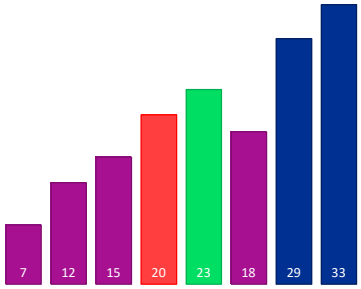
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



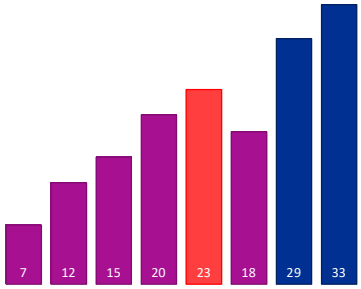
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



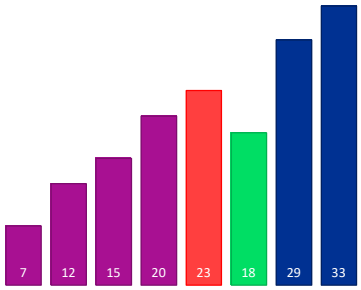
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



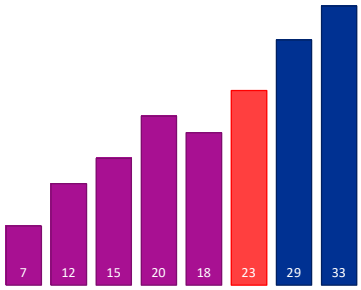
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



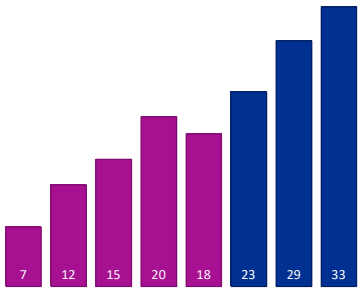
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



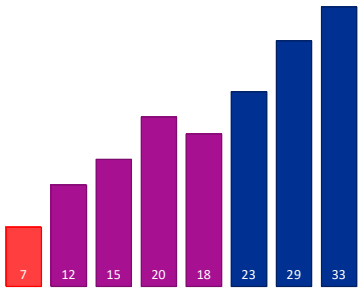
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



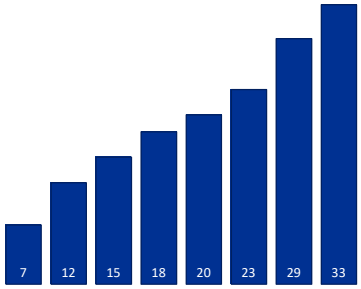
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



152

选择排序

选择排序(Selection Sort)的基本思想是：每次从当前待排序的记录中选取关键字最小的记录表，然后与待排序的记录序列中的第一个记录进行交换，直到整个记录序列有序为止。

简单选择排序

简单选择排序(Simple Selection Sort，又称为直接选择排序)的基本操作是：通过$n-i$次关键字间的比较，从$n-i+1$个记录中选取关键字最小的记录，然后和第i个记录进行交换，$i=1, 2, ..., n-1$。

算法示例

设有关键字序列为：7, 4, -2, 19, 13, 6，利用简单选择排序按照升序排列

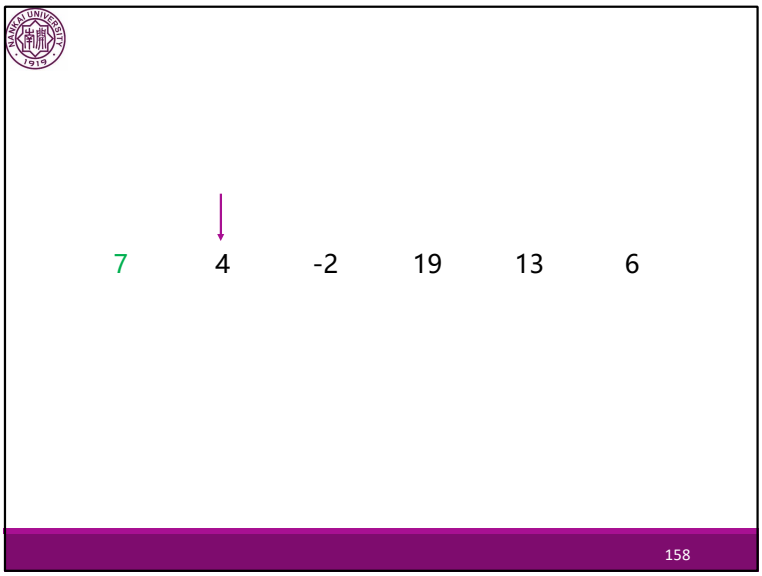
155

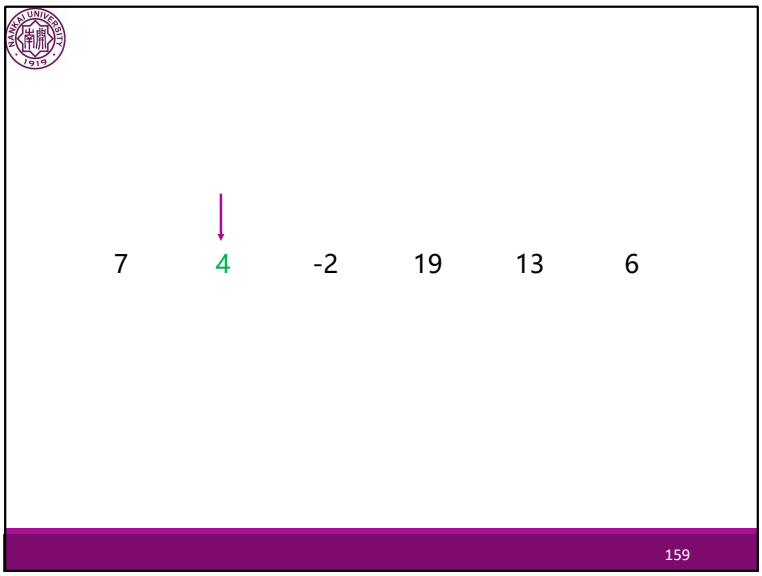
7 4 -2 19 13 6

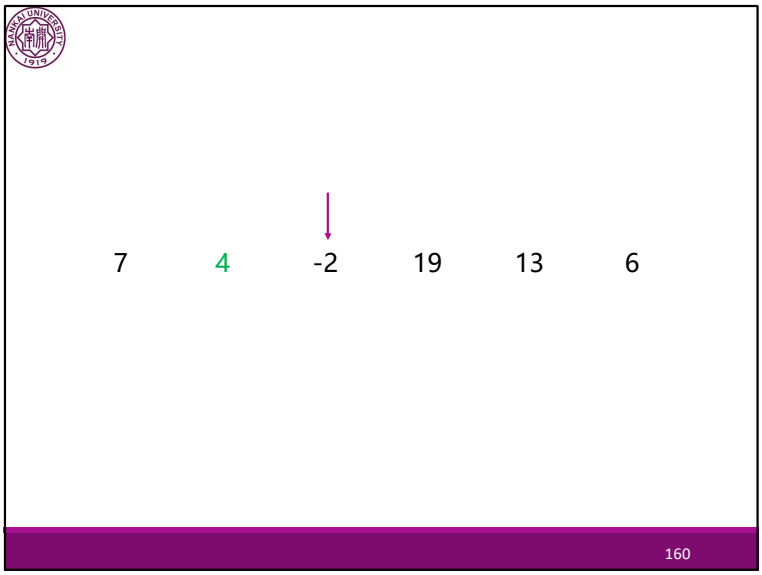
156

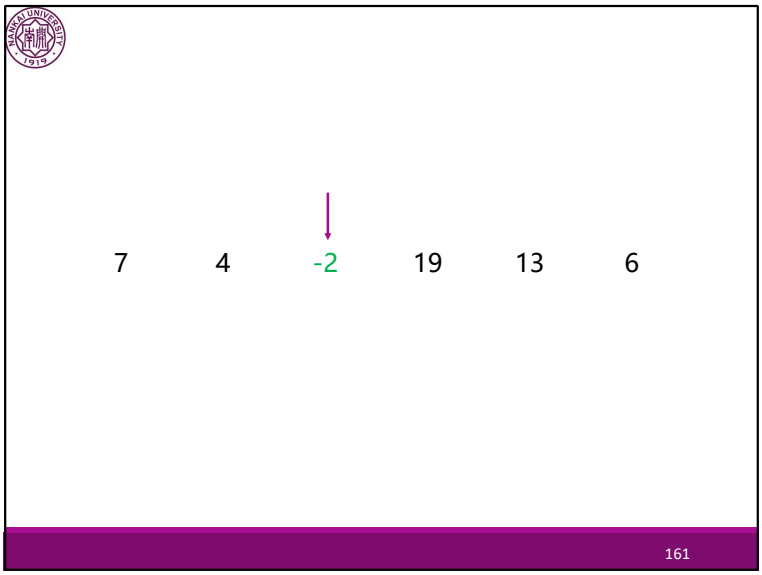
7 4 -2 19 13 6

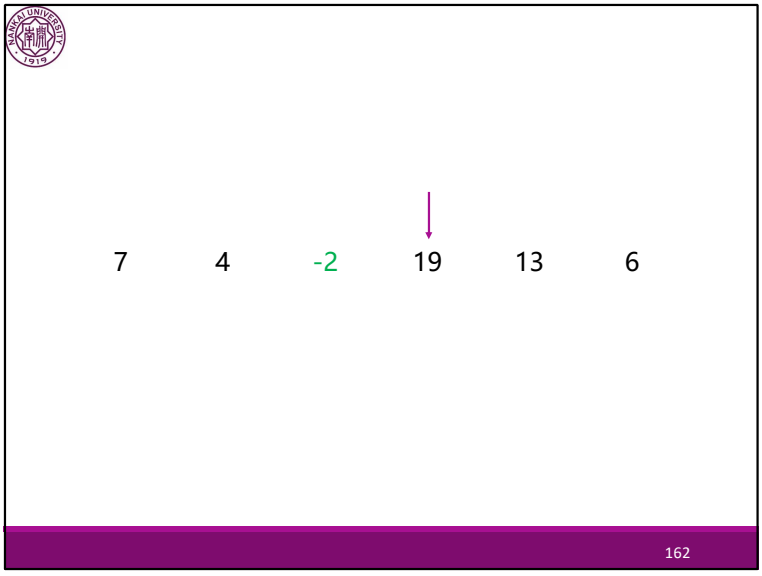
157

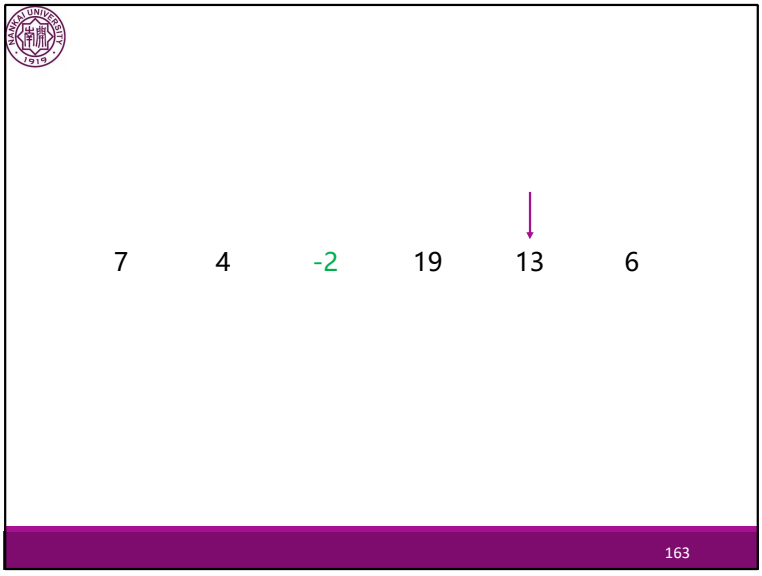

7 4 -2 19 13 6
158


7 4 -2 19 13 6
159


7 4 -2 19 13 6
160


7 4 -2 19 13 6
161


7 4 -2 19 13 6
162


7 4 -2 19 13 6
163

7 4 -2 19 13 6

164

-2 4 7 19 13 6

165

-2 4 7 19 13 6

166

-2 4 6 7 13 19

167

-2 4 6 7 13 19

168

169