数据结构 04

2025 年 3 月 13 日

考纲内容：

- **查找**
  查找的基本概念，线性表的查找，二叉排序树，哈希表的查找。
- **内排序**
  排序的基本概念，各种排序（插入排序、交换排序、选择排序、归并排序和基数排序）的基本思想和算法分析。

1 查找

1.1 查找的基本概念

查找（Searching）是在一个数据集合中寻找特定元素的过程。
基本术语：

- 查找表（Search Table）：由同一类型的数据元素组成的集合。
- 关键字（Key）：数据元素中唯一标识该元素的属性。
- 查找长度（Search Length）：查找过程中对比关键字的次数。
- 平均查找长度（Average Search Length, ASL）：所有查找过程中对比关键字次数的期望值。

查找算法的性能评价依据主要为：

- 平均查找长度（ASL）
- 时间复杂度
- 空间复杂度

1.2 线性表的查找

1.2.1 顺序查找

顺序查找（Sequential Search）是最基本的查找算法，适用于顺序表和链表等线性结构。
顺序查找的平均时间复杂度为 $O(n)$，最坏时间复杂度为 $O(n)$。

Algorithm 1 顺序查找算法

```
1: procedure SequentialSearch(ST, key)
2:     i ← 0
3:     while i < n and ST[i].key ≠ key do
4:         i ← i + 1
5:     end while
6:     if i < n then return i                          ▷ 找到，返回位置
7:     elsereturn −1                                   ▷ 未找到，返回-1
8:     end if
9: end procedure
```

Algorithm 2 折半查找算法

```
1: procedure BinarySearch(ST, key, low, high)
2:     while low ≤ high do
3:         mid ← ⌊(low + high)/2⌋
4:         if ST[mid].key = key then return mid         ▷ 找到，返回位置
5:         else if ST[mid].key > key then
6:             high ← mid − 1
7:         else
8:             low ← mid + 1
9:         end if
10:    end whilereturn −1                               ▷ 未找到，返回-1
11: end procedure
```

1.2.2 折半查找

折半查找（Binary Search）要求表中元素按关键字有序排列，且存储结构必须是随机存取的（如顺序表）。

折半查找的查找效率分析：

- 时间复杂度：$O(\log n)$
- 空间复杂度：$O(1)$

折半查找比顺序查找效率高，但要求表必须有序且采用顺序存储结构。

1.2.3 分块查找

分块查找（Block Search）介于顺序查找与折半查找之间，将表分为若干块，块内无序但块间有序。

步骤：

1. 先在索引表中确定要查找的数据所在的块
2. 然后在块内进行顺序查找

分块查找的平均查找长度：

$$ASL = \frac{s+1}{2} + \frac{b+1}{2} \tag{1}$$

其中，s 为块数，b 为每块的记录数。

1.3 二叉排序树

二叉排序树（Binary Search Tree, BST）是一种特殊的二叉树，满足以下条件：

- 若左子树非空，则左子树上所有结点的值均小于根结点的值
- 若右子树非空，则右子树上所有结点的值均大于根结点的值
- 左右子树也分别是二叉排序树

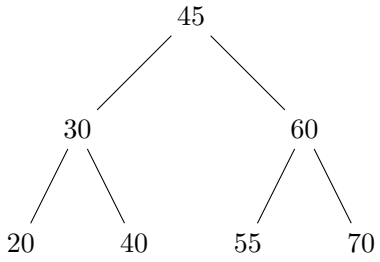

图 1: 二叉排序树示例

Algorithm 3 二叉排序树查找算法

```
procedure BSTSearch(T, key)
    if T = NULL or key = T.data.key then return T
    else if key < T.data.key then return BSTSearch(T.lchild, key)
    elsereturn BSTSearch(T.rchild, key)
    end if
end procedure
```

1.3.1 二叉排序树的基本操作

查找操作

插入操作 在二叉排序树中插入一个结点，需保持其二叉排序树的性质。插入位置一定是某个叶结点的位置。

删除操作 删除操作较为复杂，需考虑三种情况：

- 删除叶结点：直接删除
- 删除只有一个子树的结点：用子树替代该结点
- 删除有两个子树的结点：用直接后继（中序遍历下的后继）替代，然后删除后继结点

二叉排序树的查找、插入和删除操作的平均时间复杂度均为 $O(\log n)$，但最坏情况下会退化为 $O(n)$（如树退化为单链表）。

1.4 平衡二叉树

平衡二叉树（AVL 树）是一种特殊的二叉排序树，要求任一结点的左右子树高度差不超过 1。

平衡因子 BF(node) = 左子树高度 - 右子树高度，平衡二叉树中所有结点的平衡因子只能是-1、0 或 1。

插入结点后可能破坏平衡性，需通过旋转操作恢复：

- LL 型：单右旋
- RR 型：单左旋
- LR 型：先左后右双旋
- RL 型：先右后左双旋

平衡二叉树的查找、插入和删除操作的时间复杂度均为 $O(\log n)$。

1.5 哈希表的查找

哈希表（Hash Table）是一种以” 键-值” 形式存储数据的数据结构，它通过哈希函数将关键字映射到表中的位置，实现高效查找。

1.5.1 哈希函数

常用的哈希函数构造方法：

- 除留余数法：$h(key) = key \mod p$（p 通常选取小于表长的最大质数）
- 直接定址法：$h(key) = a \times key + b$
- 数字分析法：选取关键字的某些位作为哈希地址
- 平方取中法：取关键字平方后的中间几位作为哈希地址

1.5.2 冲突处理

冲突（Collision）指不同关键字映射到哈希表的同一位置。

开放定址法 当发生冲突时，使用某种探测方法寻找下一个空位置：

- 线性探测法：$h_i(key) = (h(key) + i) \mod m$
- 平方探测法：$h_i(key) = (h(key) + i^2) \mod m$
- 双散列法：$h_i(key) = (h(key) + i \times h_2(key)) \mod m$

链地址法 将哈希表的每个位置设为链表的头节点，相同哈希地址的元素存储在同一链表中。

0	→	10	→	20		
1	→	31				
2	→	12	→	22	→	32
3						
4	→	34				

图 2: 链地址法示例

1.5.3 哈希表性能分析

装填因子（Load Factor）：$\alpha = \frac{n}{m}$，其中 n 是表中记录数，m 是哈希表长度。
装填因子衡量哈希表的满度，它直接影响查找效率。一般 $\alpha \leq 0.75$ 时查找效率较高。
哈希表的查找效率：

- 最好情况：$O(1)$，没有冲突直接定位
- 最坏情况：$O(n)$，全部冲突退化为线性查找
- 平均情况：取决于哈希函数和冲突处理方法，通常近似为 $O(1)$

哈希表的优点是查找、插入、删除操作的时间复杂度接近 $O(1)$，但缺点是需要额外的存储空间，且无法实现有序操作。

2 内排序

2.1 排序的基本概念

排序（Sorting）是将一组数据按照特定的顺序进行重新排列的过程。

基本术语：

- 内排序：排序过程中数据元素全部存放在内存中进行的排序
- 外排序：排序过程中需要在内外存间多次交换数据的排序
- 稳定性：若两个记录 R_i 和 R_j 的关键字相同且排序前 R_i 在 R_j 之前，排序后 R_i 仍在 R_j 之前，则称排序算法是稳定的

排序算法的评价指标：

- 时间复杂度：比较次数和移动次数
- 空间复杂度：执行算法所需的额外存储空间
- 算法的稳定性：关键字相同的记录是否保持原有的相对位置

常见排序算法的时间复杂度比较：

算法	平均时间	最好情况	最坏情况	空间复杂度	稳定性
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	不稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定

其中，n 为待排序记录数，d 为关键字位数，r 为关键字取值范围。

2.2 插入排序

2.2.1 直接插入排序

直接插入排序（Straight Insertion Sort）的基本思想是将一个记录插入到已经排好序的有序表中，得到一个新的、记录数增加 1 的有序表。

直接插入排序的性能分析：

- 时间复杂度：平均 $O(n^2)$，最好 $O(n)$，最坏 $O(n^2)$
- 空间复杂度：$O(1)$
- 稳定性：稳定

直接插入排序适用于基本有序的数据集合或者数据集合较小的情况。

Algorithm 4 直接插入排序算法

```
1: procedure InsertionSort(A[0...n − 1])
2:     for i ← 1 to n − 1 do
3:         temp ← A[i]                          ▷ 保存当前元素
4:         j ← i − 1                            ▷ 前一位置
5:         while j ≥ 0 and A[j] > temp do
6:             A[j + 1] ← A[j]                  ▷ 元素后移
7:             j ← j − 1
8:         end while
9:         A[j + 1] ← temp                      ▷ 插入到正确位置
10:    end for
11: end procedure
```

49	38	65	97	76	13	27	49	初始数组
38	49	65	97	76	13	27	49	第 1 次插入后
13	27	38	49	49	65	76	97	最终排序结果

图 3: 直接插入排序示例

2.2.2 希尔排序

希尔排序（Shell Sort）是插入排序的一种改进版本，也称为" 缩小增量排序"。其基本思想是将整个待排记录序列分割成若干子序列分别进行直接插入排序。

Algorithm 5 希尔排序算法

```
1: procedure ShellSort(A[0...n − 1])
2:     选择一个增量序列 d_1, d_2, ..., d_t，其中 d_t = 1
3:     for k ← 1 to t do
4:         按增量 d_k 对各子序列进行直接插入排序
5:     end for
6: end procedure
```

希尔排序的性能分析：

- 时间复杂度：取决于增量序列，一般为 $O(n^{1.3})$
- 空间复杂度：$O(1)$
- 稳定性：不稳定

2.3 交换排序

2.3.1 冒泡排序

冒泡排序（Bubble Sort）的基本思想是通过相邻元素的比较和交换，使较大的元素逐渐” 浮” 到后面。

Algorithm 6 冒泡排序算法

```
1:  procedure BubbleSort(A[0...n − 1])
2:      for i ← 0 to n − 2 do
3:          swapped ← false
4:          for j ← 0 to n − 2 − i do
5:              if A[j] > A[j + 1] then
6:                  交换 A[j] 和 A[j + 1]
7:                  swapped ← true
8:              end if
9:          end for
10:         if swapped = false then
11:             return                          ▷ 数组已经有序，提前结束
12:         end if
13:     end for
14: end procedure
```

冒泡排序的性能分析：

- 时间复杂度：平均 $O(n^2)$，最好 $O(n)$，最坏 $O(n^2)$
- 空间复杂度：$O(1)$
- 稳定性：稳定

2.3.2 快速排序

快速排序（Quick Sort）的基本思想是通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，然后再分别对这两部分记录继续进行排序。

快速排序的性能分析：

- 时间复杂度：平均 $O(n \log n)$，最好 $O(n \log n)$，最坏 $O(n^2)$
- 空间复杂度：$O(\log n)$
- 稳定性：不稳定

快速排序是实践中最高效的排序算法之一，但在最坏情况下性能会显著下降。

Algorithm 7 快速排序算法

```
1: procedure QuickSort(A, low, high)
2:     if low < high then
3:         pivot ← Partition(A, low, high)
4:         QuickSort(A, low, pivot − 1)
5:         QuickSort(A, pivot + 1, high)
6:     end if
7: end procedure
8:
9: procedure Partition(A, low, high)
10:    pivot ← A[low]                                  ▷ 选择第一个元素作为基准
11:    while low < high do
12:        while low < high and A[high] ≥ pivot do
13:            high ← high − 1
14:        end while
15:        A[low] ← A[high]
16:        while low < high and A[low] ≤ pivot do
17:            low ← low + 1
18:        end while
19:        A[high] ← A[low]
20:    end while
21:    A[low] ← pivot return low
22: end procedure
```

2.4 选择排序

2.4.1 简单选择排序

简单选择排序（Simple Selection Sort）的基本思想是每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。

Algorithm 8 简单选择排序算法

```
1:  procedure SelectionSort(A[0...n − 1])
2:      for i ← 0 to n − 2 do
3:          min ← i
4:          for j ← i + 1 to n − 1 do
5:              if A[j] < A[min] then
6:                  min ← j
7:              end if
8:          end for
9:          if min ≠ i then
10:             交换 A[i] 和 A[min]
11:         end if
12:     end for
13: end procedure
```

简单选择排序的性能分析：

- 时间复杂度：平均 $O(n^2)$，最好 $O(n^2)$，最坏 $O(n^2)$
- 空间复杂度：$O(1)$
- 稳定性：不稳定

2.4.2 堆排序

堆排序（Heap Sort）是利用堆这种数据结构所设计的一种排序算法。

堆（Heap）是具有以下性质的完全二叉树：每个节点的值都大于或等于其左右孩子节点的值（大顶堆）；或者每个节点的值都小于或等于其左右孩子节点的值（小顶堆）。

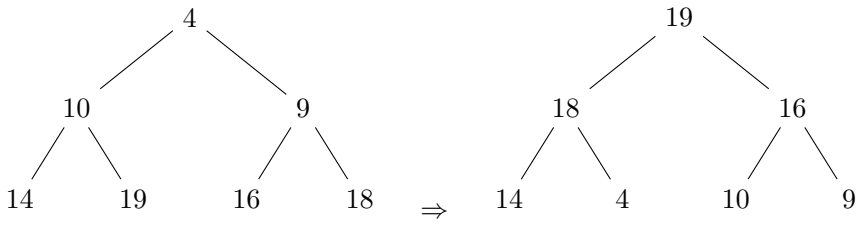

图 4: 从无序数组构建大顶堆的示例

堆排序的性能分析：

Algorithm 9 堆排序算法

```
1: procedure HeapSort(A[0...n − 1])
2:     for i ← ⌊n/2⌋ − 1 downto 0 do                          ▷ 建堆
3:         Heapify(A, n, i)
4:     end for
5:     for i ← n − 1 downto 1 do                              ▷ 堆排序
6:         交换 A[0] 和 A[i]                                  ▷ 将堆顶元素与末尾元素交换
7:         Heapify(A, i, 0)                                   ▷ 重新调整堆
8:     end for
9: end procedure
10:
11: procedure Heapify(A, n, i)                                ▷ 调整以 i 为根的子树
12:     largest ← i
13:     left ← 2 × i + 1
14:     right ← 2 × i + 2
15:     if left < n and A[left] > A[largest] then
16:         largest ← left
17:     end if
18:     if right < n and A[right] > A[largest] then
19:         largest ← right
20:     end if
21:     if largest ≠ i then
22:         交换 A[i] 和 A[largest]
23:         Heapify(A, n, largest)
24:     end if
25: end procedure
```

- 时间复杂度：平均 $O(n\log n)$，最好 $O(n\log n)$，最坏 $O(n\log n)$
- 空间复杂度：$O(1)$
- 稳定性：不稳定

堆排序是选择排序的一种优化，具有比简单选择排序更好的时间复杂度。

2.5 归并排序

归并排序（Merge Sort）是采用分治法的一个典型应用。其基本思想是将两个或两个以上的有序表合并成一个新的有序表。

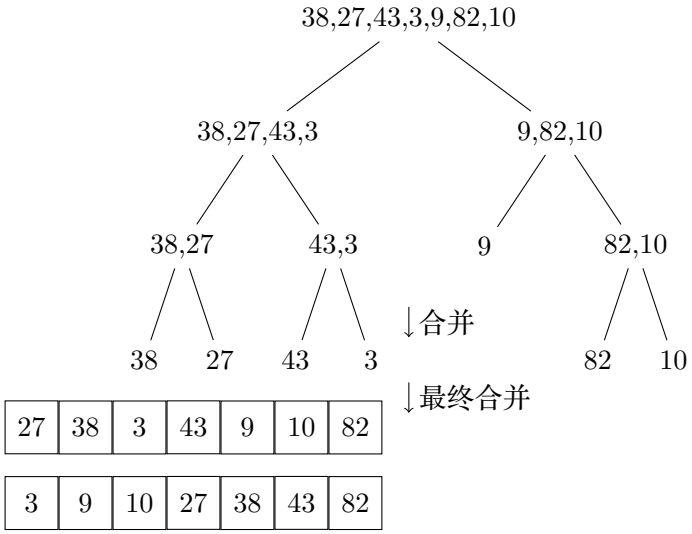

图 5: 归并排序示例

归并排序的性能分析：

- 时间复杂度：平均 $O(n\log n)$，最好 $O(n\log n)$，最坏 $O(n\log n)$
- 空间复杂度：$O(n)$
- 稳定性：稳定

归并排序是稳定的排序方法，并且具有稳定的时间复杂度，但需要额外的空间。

2.6 基数排序

基数排序（Radix Sort）是一种非比较型排序算法，它根据关键字中的位值，将待排序的元素分配到桶中，达到排序的目的。

基数排序的性能分析：

- 时间复杂度：$O(d(n+r))$，其中 d 是关键字位数，r 是关键字范围（桶的个数）

Algorithm 10 归并排序算法

```
1: procedure MergeSort(A, left, right)
2:     if left < right then
3:         mid ← ⌊(left + right)/2⌋
4:         MergeSort(A, left, mid)
5:         MergeSort(A, mid + 1, right)
6:         Merge(A, left, mid, right)
7:     end if
8: end procedure
9:
10: procedure Merge(A, left, mid, right)
11:     创建临时数组 temp[left...right]
12:     i ← left, j ← mid + 1, k ← left
13:     while i ≤ mid and j ≤ right do
14:         if A[i] ≤ A[j] then
15:             temp[k] ← A[i]
16:             i ← i + 1, k ← k + 1
17:         else
18:             temp[k] ← A[j]
19:             j ← j + 1, k ← k + 1
20:         end if
21:     end while
22:     while i ≤ mid do                                  ▷ 处理剩余元素
23:         temp[k] ← A[i]
24:         i ← i + 1, k ← k + 1
25:     end while
26:     while j ≤ right do                                ▷ 处理剩余元素
27:         temp[k] ← A[j]
28:         j ← j + 1, k ← k + 1
29:     end while
30:     for k ← left to right do                          ▷ 将临时数组复制回原数组
31:         A[k] ← temp[k]
32:     end for
33: end procedure
```

Algorithm 11 基数排序算法（LSD，从低位到高位）

```
procedure RadixSort(A[0...n − 1], d)                ▷ d 为最大位数
    for i ← 1 to d do                               ▷ 从最低位开始
        按第 i 位数字将所有元素分配到桶 bucket[0...9] 中
        将所有桶中的元素依次收集到数组 A 中
    end for
end procedure
```

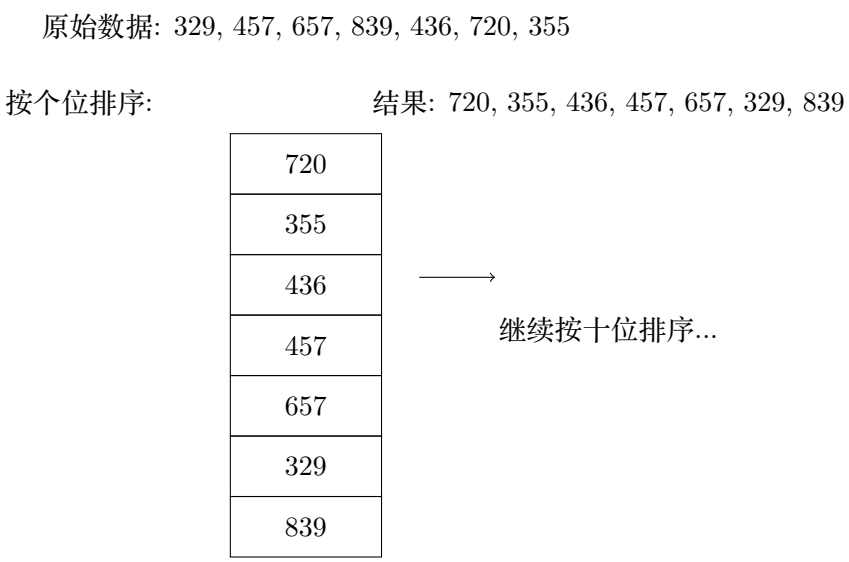

图 6: 基数排序示例（按个位排序的第一步）

- 空间复杂度：$O(n+r)$
- 稳定性：稳定

基数排序不进行关键字之间的比较，适用于数据量大但是每个数据位数不是很多的情况。

2.7 排序算法选择

在实际应用中，选择合适的排序算法需要考虑以下因素：

- **数据规模**：对于小规模数据，简单的排序算法如插入排序可能更高效；对于大规模数据，复杂度为 $O(n\log n)$ 的排序算法更合适
- **数据分布**：对于接近有序的数据，插入排序和冒泡排序可能表现更好
- **稳定性要求**：如果需要保持相同关键字元素的相对位置，应选择稳定的排序算法
- **存储要求**：如果内存空间有限，应避免使用需要额外空间的算法
- **关键字的性质**：对于整数或者可以转化为整数的关键字，可以考虑使用桶排序或基数排序

在实践中，许多编程语言的标准库采用混合策略，例如对小规模数据使用插入排序，对大规模数据使用快速排序或归并排序。