第一章 绪论

结构体对齐规则

C 库函数 - malloc()

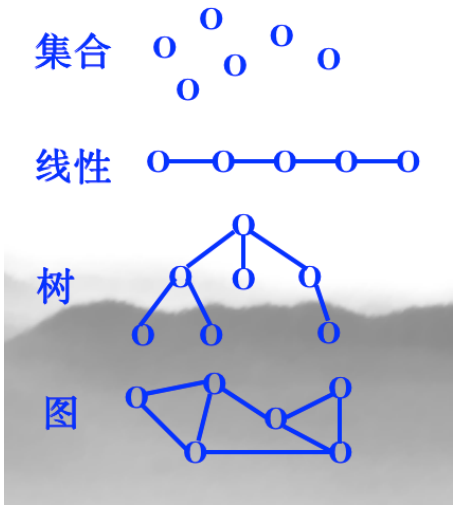
C编程指南

- 数据结构是为实现对计算机数据有效使用的各种数据组织形式，服务于各类计算机操作。不同的数据结构具有各自对应的适用场景，旨在降低各种算法计算的时间与空间复杂度，达到最佳的任务执行效率。

1.2基本概念和术语

- 数据
- 数据元素：数据的基本单位
- 数据项：一个数据元素可由多个数据项组成
  - 是数据不可分割的**最小单位**
- 数据对象：性质相同的数据元素的集合，是数据的子集。
- 数据结构：**存在一种或多种特定关系的数据元素**的集合。
  - 基本结构

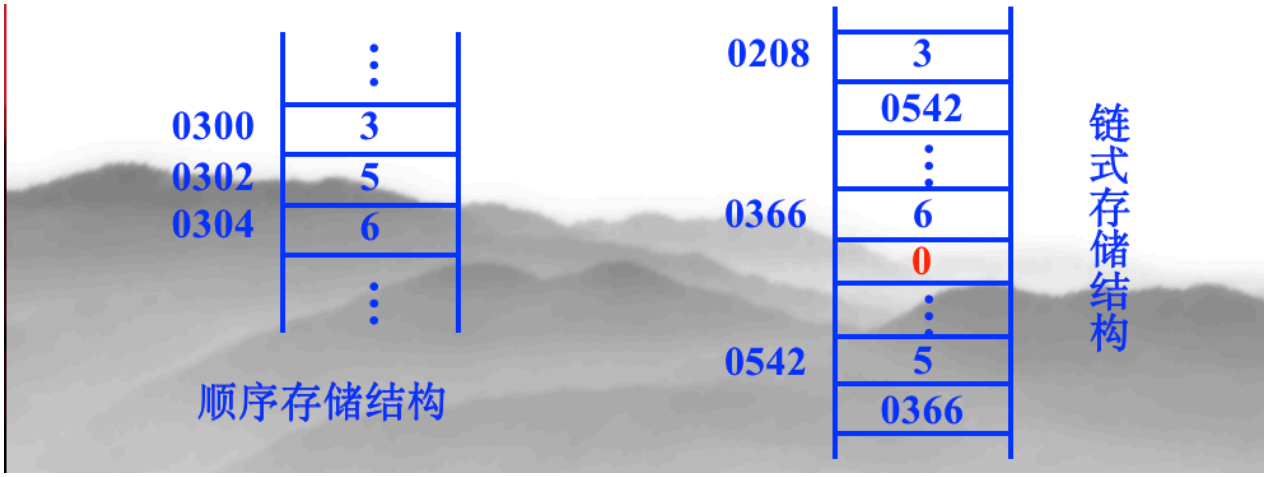
结构	内容
集合	数据元素同属关系
线性结构	数据元素之间一对一
树形结构	数据元素之间一对多
图状结构	数据元素之间多对多



  - 数据结构形式定义：数据结构是一个二元组

    Data_Structure = (D，S)

    其中：D是数据元素的有限集，S是D上关系的有限集

  - 两个层次
    - 逻辑结构： 描述了数据元素之间存在的逻辑关系。
    - 存储结构（物理结果）: 数据结构在计算机内的表示/存储方式。
- 存储结构：包括**数据元素的表示**和**数据元素之间关系**的表示
  - 数据元素的表示：通常用**位串**来表示一个数据元素
  - 两种存储结构：顺序存储结构和链式存储结构
  - 

- 数据类型: 是一个值的集合和定义在这个值集上的一组操作/运算的总称。
  - 按照值的不同性质，高级程序语言的数据类型可分为：
    - 原子类型
    - 结构类型
  - 抽象数据类型：一个数学模型以及定义在该模型上的一组操作。
    - 原子类型: 值不可分解

        - 固有聚合类型: 其值由确定数目的成分按某种结构组成
        - 可变聚合类型: 构成值的成分的数目不确定
            - 固有聚合类型与可变聚合类型统称为结构类型。
        - 抽象数据类型的形式定义: 抽象数据类型是一个三元组(D，S ， P)，其中:
            - D是数据对象，数据元素的有限集
            - S是D上关系的有限集
            - P是对D的基本操作的有限集
    - 数据类型将数据结构(数据元素、数据关系)和数据操作封装在一起，构成对象类。
    - 模块内部给出这些数据的定义、表示及其操作的细节，而在模块外部使用的只是抽象的数据和抽象的操作。
    - ADT = {值域} + {值域上的操作}
    - 多形数据类型: 是指其值的成分[1]不确定的数据类型

1.3抽象数据类型的表示与实现

1.3.1预定义常量和类型

```
#define TRUE        1
#define FALSE       1
#define OK          1
#define ERROR       1
#define INFEASIBLE 1
#define OVERFLOW  -2
typedef int Status ;
//status 是函数的类型，其值是函数结果状态代码
typedef int Status ;
```

数据结构的表示(存储结构)用类型定 义 (typeder)描述。数据元素类型约定为ElemType,由用户在使用该数据类型时自行定义.

1.3.3 三元组的实现

```
#include<iostream>
#include<cstdio>
#define TRUE       1
#define FALSE      1
#define OK         1
#define ERROR      1
#define INFEASIBLE 1
#define OVERFLOW  -2
typedef int Status ;
typedef int ElemType ;
typedef ElemType *Triplet ;
using namespace std ;
Status InitTriplet (Triplet &T, ElemType v1, ElemType v2, ElemType v3){
    T = (ElemType *) malloc(3 * sizeof(ElemType)) ;
    if (!T) exit(OVERFLOW) ;
    T[0] = v1 ;
    T[1] = v2 ;
    T[2] = v3 ;
    return OK ;
} //构造三元组T

Status DestroyTriplet(Triplet &T){
    free(T) ;
    T = NULL ;
    return OK ;
}//销毁三元组

Status Get(Triplet &T, int i, ElemType &e){
    if (i < 1 || i > 3) return ERROR ;
    e = T[i - 1] ;
    return OK ;
}//获得三元组的值

Status Put(Triplet &T, int i , ElemType e){
    if (i < 1 || i > 3) return ERROR ;
```

```
    T[i - 1] = e ;
    return OK ;
}

Status ISAscending(Triplet T){
    return (T[0] <= T[1]) && (T[1] <= T[2]) ;
}

Status ISDescending(Triplet T){
    return (T[0] >= T[1]) && (T[1] >= T[2]) ;
}

Status Max(Triplet T , ElemType &e){
    e = T[0] > T[1] ? T[0] : T[1] ;
    e = e > T[2] ? e : T[2] ;
    return OK ;
}

Status Min(Triplet T , ElemType &e){
    e = T[0] < T[1] ? T[0] : T[1] ;
    e = e < T[2] ? e : T[2] ;
    return OK ;
}

int sum(Triplet T1, Triplet T2, Triplet &SUM){
    SUM[0] = T1[0] + T2[0] ;
    SUM[1] = T1[1] + T2[1] ;
    SUM[2] = T1[2] + T2[2] ;
    return OK ;
}

int subtract(Triplet T1, Triplet T2, Triplet &SUB){
    SUB[0] = T1[0] - T2[0] ;
    SUB[1] = T1[1] - T2[1] ;
    SUB[2] = T1[2] - T2[2] ;
    return OK ;
}
```

1.4 算法和时间复杂度

1.4.1算法

算法是对特定问题求解步骤对一种描述，**它是指令的有限序列**，其中每一条指令表示一个或多个操作。

算法还有以下五个特性：

1. 有穷性
2. 确定性
3. 可行性
4. 输入
5. 输出

1.4.2算法设计的要求

1. 正确性
2. 可读性
3. 健壮性
4. 效率与低存储量需求

1.4.3算法效率的度量

时间复杂度

根据定义，时间复杂度指输入数据大小为 N 时，算法运行所需花费的时间。需要注意：统计的是算法的「计算操作数量」，而不是「运行的绝对时间」。计算操作数量和运行绝对时间呈正相关关系，并不相等。算法运行时间受到「编程语言 、计算机处理器速度、运行环境」等多种因素影响。例如，同样的算法使用 Python 或 C++ 实现、使用 CPU 或 GPU 、使用本地 IDE 或力扣平台提交，运行时间都不同。体现的是计算操作随数据大小 NN 变化时的变化情况。假设算法运行总共需要「 1 次操作」、「 100 次操作」，此两情况的时间复杂度都为常数级 O(1)；需要「 N 次操作」、「 100N 次操作」的时间复杂度都为 O(N) 。

1.4.4算法的存储空间需求

类似于时间复杂度，以**空间复杂度**作为算法所需存储空间的量度，记作：

$$S(n)=O(f(n))$$

其中n为问题的规模（大小），一个上机执行的程序除了需要存储空间来奇存本身所用指令、常数、变量和输人数据外，也需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间。若输人数据所占空间只取决于问题本身,和算法无关，则只需要分析除输人和程序之外的额外空间，否则应同时考虑**输入本身所需空间**(和**输人数据**的表示形式**有关）。若额外空间相对于输人数据量来说是**常数，**则称此算法为**原地工作**，第10章讨论的有些排序算法就属于这类。又如果所占空间量依赖于特定的输人,则除特别指明外，均按最坏情况来分析。