数据结构与算法 C++ 教材知识汇总

第 1 章 绪论

1.1 数据结构基本概念

数据是对客观事物的符号表示，在计算机科学中，它是能输入到计算机中并被计算机程序处理的符号总称。数据元素是数据的基本单位，在程序中通常作为一个整体进行考虑和处理。而数据项是构成数据元素的不可分割的最小单位。

抽象数据类型（Abstract Data Type，ADT）是指一个数学模型以及定义在该模型上的一组操作。在 C++ 中，我们通过类来实现抽象数据类型。例如，定义一个简单的复数类：

```
class Complex {
private:
 double real;
 double imag;
public:
 Complex(double r = 0, double i = 0) : real(r), imag(i) {}
 Complex operator+(const Complex& other);
 Complex operator-(const Complex& other);
};
Complex Complex::operator+(const Complex& other) {
 return Complex(real + other.real, imag + other.imag);
}
Complex Complex::operator-(const Complex& other) {
 return Complex(real - other.real, imag - other.imag);
}
```

这里通过类定义了复数的数据结构，并通过运算符重载实现了复数的加法和减法操作，体现了抽象数据类型的封装性和数据抽象特性。友元函数则打破了类的封装性，允许外部函数直接访问类的私有成员，在某些特殊场景下具有重要作用。

1.2 算法和算法分析

算法是对特定问题求解步骤的一种描述，是指令的有限序列，必须具备有穷性、确定性、可行性、输入和输出五个特性。有穷性要求算法在执行有限个步骤后必须结束；确定性确保算法的每一步都有明

确的定义，不会产生歧义；可行性表示算法中的操作都可以通过已经实现的基本运算执行有限次来实现；输入允许算法在执行过程中从外部获取数据；输出则是算法处理后的结果。

算法分析主要是对算法的时间复杂度和空间复杂度进行评估。时间复杂度反映了算法执行时间随输入规模增长的变化趋势，通过分析算法中基本操作的执行次数来确定。例如，对于一个简单的循环累加数组元素的算法：

```
int sum(int arr[], int n) {
 int s = 0;
 for (int i = 0; i < n; i++) {
 s += arr[i];
}
 return s;
}
```

其中循环内的 s += arr[i]; 是基本操作，执行次数为 n 次，所以该算法的时间复杂度为 O(n) 。空间复杂度则衡量算法在执行过程中所需的额外存储空间，上述算法中除了输入数组和少量临时变量外，没有占用与输入规模相关的额外空间，所以空间复杂度为 O(1) 。

第 2 章 线性表

2.1 逻辑结构

线性表是具有相同数据类型的 n 个数据元素的有限序列，记为 (a_1, a_2, \cdots, a_n) ，其中 a_i 表示第 i 个数据元素，n 为线性表的长度，当 n = 0 时称为空表。线性表具有线性结构的特点，即除第一个元素外，每个元素都有且仅有一个直接前驱；除最后一个元素外，每个元素都有且仅有一个直接后继。

2.2 顺序存储结构

顺序存储结构是用一组地址连续的存储单元依次存储线性表的数据元素。在 C++ 中，可以使用数组来实现顺序表。例如：

```
const int MAXSIZE = 100;
class SeqList {
private:
 int data[MAXSIZE];
 int length;
public:
 SeqList() : length(0) {}
```

```
 bool insert(int i, int e);
 bool deleteElem(int i, int& e);
 int getElem(int i);
};
bool SeqList::insert(int i, int e) {
 if (i < 1 || i > length + 1 || length == MAXSIZE) {
 return false;
}
 for (int j = length; j >= i; j--) {
 data[j] = data[j - 1];
}
 data[i - 1] = e;
 length++;
 return true;
}
bool SeqList::deleteElem(int i, int& e) {
 if (i < 1 || i > length) {
 return false;
}
 e = data[i - 1];
 for (int j = i; j < length; j++) {
 data[j - 1] = data[j];
}
 length--;
 return true;
}
int SeqList::getElem(int i) {
 if (i < 1 || i > length) {
 throw "Out of range";
}
 return data[i - 1];
}
```

顺序表的优点是可以随机访问元素，时间复杂度为 O(1)；缺点是插入和删除操作时，平均需要移动约一半的元素，时间复杂度为 O(n)。

2.3 链式存储结构

链式存储结构通过指针将节点链接起来，每个节点包含数据域和指针域。单链表是最基本的链表形式，每个节点只有一个指向后继节点的指针。例如：

```
struct ListNode {
 int val;
 ListNode* next;
 ListNode(int x) : val(x), next(NULL) {}
};
class SinglyLinkedList {
```

```
private:
 ListNode* head;
public:
 SinglyLinkedList() : head(NULL) {}
 void insert(int val);
 bool deleteNode(int val);
 ListNode* getHead();
};
void SinglyLinkedList::insert(int val) {
 ListNode* newNode = new ListNode(val);
 newNode->next = head;
 head = newNode;
}
bool SinglyLinkedList::deleteNode(int val) {
 ListNode* prev = NULL;
 ListNode* curr = head;
 while (curr != NULL && curr->val != val) {
 prev = curr;
 curr = curr->next;
}
 if (curr == NULL) {
 return false;
}
 if (prev == NULL) {
 head = curr->next;
} else {
 prev->next = curr->next;
}
 delete curr;
 return true;
}
ListNode* SinglyLinkedList::getHead() {
 return head;
}
```

除了单链表，还有循环链表，其尾节点的指针指向头节点，形成一个环；双向链表则每个节点有两个指针，分别指向前驱和后继节点，使得链表的遍历和操作更加灵活。链式存储结构在插入和删除操作时，只需修改指针，时间复杂度为 O(1) ，但访问元素需要顺序查找，时间复杂度为 O(n) 。

第 3 章 栈和队列

3.1 栈

栈是一种后进先出（LIFO）的数据结构，就像一个只能从顶部取放物品的容器。顺序栈可以使用数组实现，链式栈则通过链表实现。以顺序栈为例：

```
const int MAXSIZE = 100;
class SeqStack {
private:
 int data[MAXSIZE];
 int top;
public:
 SeqStack() : top(-1) {}
 bool push(int e);
 bool pop(int& e);
 bool getTop(int& e);
};
bool SeqStack::push(int e) {
 if (top == MAXSIZE - 1) {
 return false;
}
 data[++top] = e;
 return true;
}
bool SeqStack::pop(int& e) {
 if (top == -1) {
 return false;
}
 e = data[top--];
 return true;
}
bool SeqStack::getTop(int& e) {
 if (top == -1) {
 return false;
}
 e = data[top];
 return true;
}
```

栈在表达式求值、函数调用、递归过程实现等场景中广泛应用。例如，在计算后缀表达式时，利用栈可以方便地进行操作数和运算符的处理。

3.2 队列

队列是一种先进先出（FIFO）的数据结构，类似于排队的场景。链队列使用链表实现，循环队列则通过数组实现，并且通过巧妙地处理队头和队尾指针来实现循环。以循环队列为例：

```
const int MAXSIZE = 100;
class CircularQueue {
private:
 int data[MAXSIZE];
```

```
 int front;
 int rear;
public:
 CircularQueue() : front(0), rear(0) {}
 bool enQueue(int e);
 bool deQueue(int& e);
 bool getFront(int& e);
 bool isEmpty();
};
bool CircularQueue::enQueue(int e) {
 if ((rear + 1) % MAXSIZE == front) {
 return false;
}
 data[rear] = e;
 rear = (rear + 1) % MAXSIZE;
 return true;
}
bool CircularQueue::deQueue(int& e) {
 if (front == rear) {
 return false;
}
 e = data[front];
 front = (front + 1) % MAXSIZE;
 return true;
}
bool CircularQueue::getFront(int& e) {
 if (front == rear) {
 return false;
}
 e = data[front];
 return true;
}
bool CircularQueue::isEmpty() {
 return front == rear;
}
```

队列常用于处理需要按顺序处理的数据，如操作系统中的作业排队、广度优先搜索算法等。优先队列则是一种特殊的队列，其中每个元素都有一个优先级，出队时总是优先取出优先级最高的元素，在 Dijkstra 算法等场景中有重要应用。

第 4 章 串

4.1 串类型的定义

串（String）是由零个或多个字符组成的有限序列，一般记为 s = "a_1a_2\cdots a_n" ，其中 s 是串的名称，a_i 可以是字母、数字或其他字符，n 为串的长度，当 n = 0 时称为空串。与线性表不同，串的基本操作通常以子串为单位进行。

4.2 字符串的实现

在 C++ 中，字符串可以使用字符数组或 string 类来实现。字符数组实现方式如下：

```
#include <iostream>
using namespace std;
int main() {
 char str1[10] = "hello";
 char str2[] = "world";
 cout << str1 << " " << str2 << endl;
 return 0;
}
```

string 类则提供了更丰富的操作接口，如字符串连接、查找、替换等：

```
#include <iostream>
#include <string>
using namespace std;
int main() {
 string s1 = "hello";
 string s2 = "world";
 string s3 = s1 + " " + s2;
 cout << s3 << endl;
 cout << s3.find("world") << endl;
 return 0;
}
```

4.3 字符串模式匹配算法

简单字符串模式匹配算法（Brute-Force 算法）通过逐个比较主串和模式串的字符来查找模式串在主串中的位置，时间复杂度为 O(mn) ，其中 m 为模式串长度，n 为主串长度。首尾字符串模式匹配算法通过从模式串的首尾两端同时进行比较，在一些情况下可以提高效率。KMP（Knuth-Morris-Pratt）算法则利用模式串的部分匹配信息，避免不必要的字符比较，将时间复杂度降低到 O(m + n) 。

第 5 章 数组和广义表

5.1 数组

数组是由一组类型相同的数据元素构成的有限序列，每个数据元素称为数组元素。数组可以是一维、二维或多维的。在 C++ 中，二维数组可以表示为 `int arr[m][n]` ，它在内存中按行优先存储。例如：

```
#include <iostream>
using namespace std;
int main() {
 int arr[3][4] = {
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12}
};
 for (int i = 0; i < 3; i++) {
 for (int j = 0; j < 4; j++) {
 cout << arr[i][j] << " ";
}
 cout << endl;
}
 return 0;
}
```

对于特殊矩阵，如对称矩阵、三角矩阵等，可以利用其特殊性质进行压缩存储，以节省存储空间。稀疏矩阵则是指非零元素个数远小于矩阵元素总数的矩阵，常用的存储方式有三元组表和十字链表法。

5.2 广义表

广义表是线性表的推广，它的元素可以是单个元素，也可以是广义表。例如，LS = ((a, b), (c, (d, e))) 就是一个广义表，其中 (a, b) 和 (c, (d, e)) 都是它的子表。广义表的存储结构通常采用链式存储，每个节点可以表示为原子节点或子表节点，通过不同的指针域来区分。

第 6 章 树和二叉树

6.1 树的基本概念

树是 n 个节点的有限集，当 n = 0 时称为空树。在任意一棵非空树中：（1）有且仅有一个特定的称为根的节点；（2）当 n > 1 时，其余节点可分为 m 个互不相交的有限集 $T_1, T_2, \cdots, T_m$ ，其中每一个集合本身又是一棵树，并且称为根的子树。树的基本术语包括节点的度、树的度、叶子节点、分支节点、层次、树的深度等。

6.2 二叉树

二叉树是一种特殊的树，每个节点最多有两个子树，分别称为左子树和右子树。二叉树具有一些重要的性质，如第 i 层上最多有 2^{i - 1} 个节点；深度为 k 的二叉树最多有 2^k - 1 个节点等。二叉树的存储结构有顺序存储和链式存储两种，链式存储使用二叉链表，每个节点包含数据域、左指针域和右指针域。

二叉树的遍历算法包括前序遍历（根 - 左 - 右）、中序遍历（左 - 根 - 右）和后序遍历（左 - 右 - 根）。例如，对于二叉链表存储的二叉树：

```
struct TreeNode {
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
void preOrder(TreeNode* root) {
 if (root != NULL) {
 cout << root->val << " ";
 preOrder(root->left);
 preOrder(root->right);
}
}
void inOrder(TreeNode* root) {
 if (root != NULL) {
 inOrder(root->left);
 cout << root->val << " ";
 inOrder(root->right);
}
}
void postOrder(TreeNode* root) {
 if (root != NULL) {
 postOrder(root->left);
 postOrder(root->right);
 cout << root->val << " ";
}
}
```

线索二叉树通过利用二叉链表中的空指针，记录节点的前驱和后继信息，提高了遍历效率。树和森林与二叉树之间可以进行相互转换，这种转换在一些算法实现中具有重要作用。哈夫曼树是一种带权路径长度最短的二叉树，常用于哈夫曼编码，以实现数据的压缩和编码。

第 7 章 图

7.1 图的定义和术语

图 G 由顶点集 V(G) 和边集 E(G) 组成，记为 G = (V, E) 。根据边是否有方向，图可分为无向图和有向图。图的基本术语包括顶点的度（无向图）、入度和出度（有向图）、路径、回路、连通图等。

7.2 图的存储表示

邻接矩阵是一种用二维数组表示图的方法，对于无向图，邻接矩阵是对称矩阵；对于有向图，邻接矩阵不一定对称。例如：

```
const int MAX_VERTEX_NUM = 100;
class AdjMatrixGraph {
private:
 int vertex[MAX_VERTEX_NUM];
 int arc[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
```

> （注：文档部分内容可能由 AI 生成）