栈

基本概念

栈（Stack）是一种遵循后进先出（LIFO, Last In First Out）原则的有序集合。新添加的或待删除的元素都保存在栈的同一端，称为栈顶，另一端就是栈底。

操作

在栈的使用中，主要涉及以下几个操作：

- `Push`（入栈）：在栈顶添加元素
- `Pop`（出栈）：删除栈顶元素
- `Top`（获取栈顶元素）：返回栈顶元素，但不删除
- `isEmpty`（判断栈空）：检查栈是否为空
- `isFull`（判断栈满）：检查栈是否已满

顺序存储

定义

顺序栈是使用一维数组存储栈中元素，并使用一个变量`top`记录栈顶元素的位置。

特点

- 当`top = -1`时，栈为空
- 当`top = maxSize - 1`时，栈为满

代码实现

```
#include <iostream>
using namespace std;

#define MAXSIZE 10

class Stack {
private:
    int data[MAXSIZE];
    int top;

public:
    Stack() { top = -1; }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == MAXSIZE - 1;
    }

    void push(int x) {
```

```
        if (isFull()) {
            cout << "Stack Overflow" << endl;
            return;
        }
        data[++top] = x;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow" << endl;
            return -1;
        }
        return data[top--];
    }

    int getTop() {
        if (isEmpty()) {
            cout << "Stack is Empty" << endl;
            return -1;
        }
        return data[top];
    }
};
```

链式存储

定义

链式栈是栈的链式存储结构，实际上是一个单链表，通常称为链栈。

特点

- 插入和删除操作只能在链栈的栈顶进行。

代码实现

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

class LinkedStack {
private:
    Node* top;

public:
    LinkedStack() { top = nullptr; }
```

```
    bool isEmpty() {
        return top == nullptr;
    }

    void push(int x) {
        Node* node = new Node(x);
        node->next = top;
        top = node;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow" << endl;
            return -1;
        }
        Node* temp = top;
        top = top->next;
        int poppedValue = temp->data;
        delete temp;
        return poppedValue;
    }

    int getTop() {
        if (isEmpty()) {
            cout << "Stack is Empty" << endl;
            return -1;
        }
        return top->data;
    }
};
```

队列

队列的特点

队列（Queue）是一种先进先出（FIFO, First In First Out）的数据结构，具有以下特点：

- **插入和删除操作限定**：插入操作仅在队尾（rear）进行，删除操作仅在队头（front）进行。
- **先进先出原则**：最早进入队列的元素将是最先被删除的元素。

队列的存储结构

队列的两种基本存储结构：

顺序存储

1. **定义**：顺序队列是用一维数组来实现的队列。需要两个变量`front`和`rear`分别记录队列的首元素和尾元素的位置。
2. **队空和队满的判定**：
    - **队空条件**：`front == rear`
    - **队满条件**：当(rear+1)%maxSize==front时，队列满。

链式存储

1. **定义**：链式队列通常使用单链表实现，链表的一端作为队列的头部（队首元素），另一端作为尾部（队尾元素）。
2. **队空和队满的判定**：
   - **队空条件**：当头指针`front`指向`NULL`时。
   - **队满条件**：由于链式存储的动态性，理论上不会“队满”，除非系统内存耗尽。

实现

顺序队列的实现

```
#include <iostream>
using namespace std;

#define MAXSIZE 10

class CircularQueue {
private:
    int data[MAXSIZE];
    int front;
    int rear;

public:
    CircularQueue() {
        front = 0;
        rear = 0;
    }

    bool isEmpty() {
        return front == rear;
    }

    // 修改队满的判断条件
    bool isFull() {
        return (rear + 1) % MAXSIZE == front;
    }

    void enqueue(int x) {
        if (isFull()) {
            cout << "Queue Overflow" << endl;
            return;
        }
        data[rear] = x;
        rear = (rear + 1) % MAXSIZE;
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow" << endl;
            return -1;
        }
        int x = data[front];
        front = (front + 1) % MAXSIZE;
        return x;
    }
```

```
    int getFront() {
        if (isEmpty()) {
            cout << "Queue is Empty" << endl;
            return -1;
        }
        return data[front];
    }
};
```

链式队列的实现

```
class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

class LinkedQueue {
private:
    Node* front;
    Node* rear;

public:
    LinkedQueue() {
        front = nullptr;
        rear = nullptr;
    }

    bool isEmpty() {
        return front == nullptr;
    }

    void enqueue(int x) {
        Node* newNode = new Node(x);
        if (rear == nullptr) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow" << endl;
            return -1;
        }
        Node* temp = front;
        front = front->next;
        if (front == nullptr) {
```

```
            rear = nullptr;
        }
        int poppedValue = temp->data;
        delete temp;
        return poppedValue;
    }

    int getFront() {
        if (isEmpty()) {
            cout << "Queue is Empty" << endl;
            return -1;
        }
        return front->data;
    }
};
```

链表

链表概述

链表是一种常用的基础数据结构，是一系列节点的集合。每个节点至少包含两个部分：一个是存储数据元素的数据域，另一个是存储下一个节点地址的指针域。

单链表

定义

单链表是链表的一种，每个节点只有一个指针指向下一个节点，通过这种方式，链表将所有节点连接在一起。

基本操作

1. **创建**：初始化链表。
2. **插入**：在指定位置插入一个新的节点。
3. **删除**：删除指定位置的节点。
4. **搜索**：遍历链表寻找具有指定值的节点。
5. **更新**：更新指定位置节点的数据。
6. **清空**：清空整个链表。

代码示例

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

class SingleLinkedList {
private:
    ListNode* head;
public:
    SingleLinkedList() { head = nullptr; }
    void insert(int val) {
        ListNode* newNode = new ListNode(val);
```

```
        newNode->next = head;
        head = newNode;
    }
    // ... 其他操作（删除、搜索、更新等）
};
```

双链表

定义

双链表与单链表类似，但每个节点有两个指针，一个指向下一个节点，另一个指向前一个节点。

基本操作

双链表的基本操作与单链表类似，但因为其双向性质，某些操作（如反向遍历）会更方便。

代码示例

```
// 定义双向链表的节点结构
struct Node {
    int data;
    Node* prev;
    Node* next;
    // 构造函数
    Node(int data) : data(data), prev(nullptr), next(nullptr) {}
};
// 定义双向链表类
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;
public:
    // 构造函数
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}
    // 析构函数
    ~DoublyLinkedList() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
    // 向链表尾部添加节点
    void append(int data) {
        Node* newNode = new Node(data);
        if (!head) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
    // 在链表头部添加节点
    void prepend(int data) {
        Node* newNode = new Node(data);
        if (!head) {
```

```
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }
};
```

循环链表

定义

循环链表是另一种形式的链式存储结构，它的特点是表中最后一个节点的指针不是指向`NULL`，而是指向头节点，形成一个环。

基本操作

循环链表的基本操作与单链表类似，但需要特别注意处理尾节点的指针。

代码示例

```
// 定义循环链表的节点结构
struct Node {
    int data;
    Node* next;

    // 构造函数
    Node(int data) : data(data), next(nullptr) {}
};

// 定义循环链表类
class CircularLinkedList {
private:
    Node* head;
    Node* tail;

public:
    // 构造函数
    CircularLinkedList() : head(nullptr), tail(nullptr) {}

    // 析构函数
    ~CircularLinkedList() {
        if (head) {
            Node* temp = head;
            tail->next = nullptr; // 断开循环链接，以便正常删除
            while (temp) {
                Node* next = temp->next;
                delete temp;
                temp = next;
            }
        }
    }
};
```

堆

最小堆的概述

最小堆（Min Heap）是一种特殊的完全二叉树。在最小堆中，任何一个父节点的值都小于或等于它的左右子节点的值。这个特性确保了堆的最小元素总是位于根节点。

堆的定义

C++中最小堆的结构可以这样定义：

```
struct minHeap {
    int *data;      // 存储堆元素的数组
    int size;       // 当前堆中元素的数量
    int capacity;  // 堆的最大容量
};
```

建立最小堆

调整为最小堆

调整为最小堆主要是通过`percDown`函数实现的，该函数确保从当前节点到叶节点的路径上的节点满足最小堆的性质。

```
void percDown(minHeap* H, int index) {
    int tmp = H->data[index];
    int parent = index, child = 0;
    for (; parent * 2 + 1 <= H->size; parent = child) {
        child = parent * 2;
        if (child + 1 <= H->size && H->data[child] > H->data[child + 1])
            child++;
        if (tmp <= H->data[child])
            break;
        H->data[parent] = H->data[child];
    }
    H->data[parent] = tmp;
}
```

将整个数组调整为最小堆

对半数以上的非叶子节点依次执行`percDown`操作。

```
void toMinHeap(minHeap* H) {
    for (int i = H->size / 2; i > 0; i--) {
        percDown(H, i);
    }
}
```

创建并初始化最小堆

首先创建一个空堆，然后将数组元素拷贝到堆中，并调整为最小堆。

```
minHeap* createMinHeap() {
    minHeap* H = new minHeap();
    H->data = new int[maxSize + 1];
    H->data[0] = INT_MIN;  // 哨兵，比堆中所有元素都要小
```

```
    H->size = 0;
    H->capacity = maxSize;
    return H;
}

minHeap* initMinHeap(int nums[], int n) {
    minHeap* H = createMinHeap();
    for (int i = 1; i <= n; i++) {
        H->data[i] = nums[i - 1];
    }
    H->size = n;  // 关键
    toMinHeap(H);
    return H;
}
```

堆的插入

插入操作需要保证堆的结构和性质不变。新元素首先被插入到数组的尾部，然后沿着树向上调整到正确的位置。

```
void insert(minHeap* H, int X) {
    if (H->size == H->capacity) {
        return;  // 堆满了
    }
    int i = ++H->size;
    for (; H->data[i / 2] > X; i /= 2) {
        H->data[i] = H->data[i / 2];
    }
    H->data[i] = X;
}
```

堆的删除

删除操作通常删除堆顶元素（最小元素）。删除后，将堆中最后一个元素放到根节点，然后执行`percDown`操作，保证最小堆的性质。

```
int pop(minHeap* H) {
    int res = H->data[1];  // 获取堆顶元素
    H->data[1] = H->data[H->size--];  // 将最后一个元素移动到堆顶
    percDown(H, 1);  // 从新的堆顶开始调整
    return res;
}
```

哈希表

哈希表（Hash Table），也称为散列表，是根据关键码值(Key value)而直接进行访问的数据结构。通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数称作散列函数（Hash Function），存放记录的数组称为哈希表（或散列表）。

散列查找函数的构建

散列查找函数的构建主要考虑两个因素：计算简单和减少冲突。

1. 直接定址法
   - 直接取关键词或关键词的某个线性函数值作为散列地址。公式如下：

$$h(key) = a \times key + b$$

   - 其中，`a` 和 `b` 是常数。
2. 除留余数法
   - 选择一个合适的不大于哈希表长的数p，然后对于给定的关键字key，采用除法求余数的方法，计算位置：

$$h(key) = key \mod p$$

3. 数字分析法
   - 考虑关键词的数字特点，选取其中部分数字作为散列地址。
4. 折叠法
   - 将关键词分割为几部分，然后取这几部分数字的叠加和作为散列地址。
5. 平方取中法
   - 取关键词平方后的中间几位作为散列地址。
6. 字符关键词的散列函数构造
   - 对于字符类型的关键词，通常将每个字符转换成一个整数，然后使用上述数值关键词的散列函数。

散列查找的冲突解决方法

当两个或多个关键词散列到同一位置时，称为冲突（Collision）。解决冲突的方法主要有：

线性探查法

当冲突发生时，以固定的增量序列（通常是1，2……，(TableSize-1)）在表中循环探测下一存储地址。

拉链法

拉链法将所有散列到同一个值的关键词存储在同一个链表中。

定义结点

```
#define Capacity 10  // 散列表的长度
// 结点
struct HashNode{
    int data;
    HashNode* next;
    HashNode(){
        data = 0;
        next = nullptr;
    }
    HashNode(int d){
        data = d;
        next = nullptr;
    }
};
```

定义表

```
// 散列表
struct HashSet{
    HashNode *table;
    int tableSize;
    HashSet(){
        table = new HashNode[Capacity];
        tableSize = 0;
    }
};
```

插入

头插法

```
// 插入
void insertHashSet(HashSet* table, int data){
    int key = hashcode(data);
    HashNode *hashNode = new HashNode(data);
    hashNode->next = table->table[key].next;
    table->table[key].next = hashNode;
}
```

查找

```
// 查找
HashNode* findHashSet(HashSet* table, int data){
    int key = hashcode(data);
    HashNode *ptr = table->table[key].next;
    while(ptr != nullptr){
        if(ptr->data == data){
            return ptr;
        }else{
            ptr = ptr->next;
        }
    }
    return ptr;
}
```

删除

```
// 删除
bool deleteHashSet(HashSet* table, int data){
    int key = hashcode(data);
    HashNode *ptr = table->table[key].next;
    HashNode *prev = &table->table[key];
    while(ptr != nullptr){
        if(ptr->data == data){
            prev->next = ptr->next;
            delete ptr;
            return true;
        }else{
            prev = ptr;
            ptr = ptr->next;
        }
    }
```

```
    return false;
}
```

哈希表的主要优点是，当散列表足够大时，时间复杂度可以接近O(1)，即达到常数级时间。其缺点是，对散列函数要求较高，并且当表填满时性能急剧下降。因此，在实际应用中，合理设计散列函数和冲突解决方法至关重要。

广义表

广义表（Generalized List）是线性表的推广。对于线性表而言，元素具有相同的类型；而在广义表中，元素可以是单个元素，也可以是另一个广义表，这种元素的递归性质给广义表带来了极大的灵活性和表达能力。

存储

广义表可以通过多种方式存储，例如用链表的形式。在链表实现中，每个节点可以有两个指针，一个指向该节点的下一个节点（表尾），另一个可能指向与该节点相关联的另一个广义表（表头或子表）。

特性

1. **长度（Length）**：广义表的长度是指最外层的元素个数。
2. **深度（Depth）**：广义表的深度是指表中括号的最大嵌套层数。
3. **表头（Head）**：广义表非空时，第一个元素是表头。表头可以是单个元素，也可以是一个子表。
4. **表尾（Tail）**：广义表的表尾是指除去表头的其余部分，表尾本身构成一个新的广义表。

操作

1. **求表头（GetHead/Head）**：获取广义表的第一个元素。
2. **求表尾（GetTail/Tail）**：获取广义表除第一个元素以外的其余部分。

例子

例1：

```
广义表A=(a,b,(c,d),(e,(f,g)))
求：GetHead(GetTail(GetHead(GetTail(GetTail(A)))))
```

步骤：

- GetTail(A) 去除表头a，得到 (b,(c,d),(e,(f,g)))
- GetTail(GetTail(A)) 得到 ((c,d),(e,(f,g)))
- GetHead(GetTail(GetTail(A))) 得到 (c,d)
- GetTail(GetHead(GetTail(GetTail(A)))) 得到 (d)
- GetHead(GetTail(GetHead(GetTail(GetTail(A))))) 得到 d

例2：

```
广义表L=((x,y,z),a,(u,t,w))，求取出原子项t的运算。
```

步骤：

- Tail(L) 得到 (a,(u,t,w))
- Tail(Tail(L)) 得到 ((u,t,w))
- Head(Tail(Tail(L))) 得到 (u,t,w)

- Tail(Head(Tail(Tail(L)))) 得到 (t,w)
- Head(Tail(Head(Tail(Tail(L))))) 得到 t

所以正确的操作是 D.Head(Tail(Head(Tail(Tail(L))))).

代码实现

```
#include <iostream>
#include <memory>
#include <variant>
#include <vector>
// 前置声明
class GenList;
// 定义广义表节点可以是单个元素（此处简单使用int）或另一个广义表
using Node = std::variant<int, std::shared_ptr<GenList>>;
// 定义广义表类
class GenList {
public:
    // 构造函数
    GenList() = default;
    // 添加元素
    void add(const Node& node) {
        nodes.push_back(node);
    }
    // 获取表头
    Node head() const {
        if (nodes.empty()) {
            throw std::runtime_error("The generalized list is empty");
        }
        return nodes.front();
    }
    // 获取表尾
    GenList tail() const {
        if (nodes.size() < 2) {
            throw std::runtime_error("No tail in a list with less than two
elements");
        }
        GenList tailList;
        for (size_t i = 1; i < nodes.size(); ++i) {
            tailList.add(nodes[i]);
        }
        return tailList;
    }
    // 打印广义表
    void print() const {
        std::cout << "(";
        for (size_t i = 0; i < nodes.size(); ++i) {
            if (i > 0) {
                std::cout << ", ";
            }
            std::visit([](const auto& arg) {
                using T = std::decay_t<decltype(arg)>;
                if constexpr (std::is_same_v<T, int>) {
                    std::cout << arg;
                } else {
                    arg->print();
                }
```

```
            }, nodes[i]);
        }
        std::cout << ")";
    }
private:
    std::vector<Node> nodes;
};

int main() {
    // 创建广义表
    auto list = std::make_shared<GenList>();
    list->add(1);
    list->add(2);
    auto sublist = std::make_shared<GenList>();
    sublist->add(3);
    sublist->add(4);
    list->add(sublist);
    // 打印广义表
    list->print(); // 输出: (1, 2, (3, 4))
    // 获取并打印表头
    auto head = list->head();
    std::visit([](const auto& arg) {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, int>) {
            std::cout << "\nHead: " << arg << std::endl;
        } else {
            std::cout << "\nHead: ";
            arg->print();
            std::cout << std::endl;
        }
    }, head);

    // 获取并打印表尾
    auto tail = list->tail();
    std::cout << "Tail: ";
    tail.print();
    std::cout << std::endl;

    return 0;
}
```

并查集

并查集（Disjoint Set Union，DSU）是一种处理不交集的合并及查询问题的数据结构。能够高效地进行以下两种操作：

1. **Find**：确定某个元素属于哪一个子集。这可以用于确定两个元素是否属于同一子集。
2. **Union**：将两个子集合并成同一个集合。

并查集的关键点

1. **代表元素**：每个集合都有一个代表元素，用于标识这个集合。
2. **路径压缩**（Path Compression）：在执行Find操作时优化查找路径，使得每个节点直接或间接地指向代表元素。

3. **按秩合并**（Union by Rank）：在执行Union操作时，总是将较小的集合合并到较大的集合上，这样可以避免树的不平衡，优化性能。

并查集的实现

包含了路径压缩和按秩合并的优化：

```
#include <iostream>
#include <vector>

class UnionFind {
private:
    std::vector<int> parent;  // 父节点
    std::vector<int> rank;    // 树的高度（秩）
public:
    // 构造函数，初始化n个元素
    UnionFind(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    // 查找x的根节点（代表元素）
    int find(int x) {
        if (x != parent[x]) {
            parent[x] = find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }
    // 合并x和y所在的集合
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                if (rank[rootX] == rank[rootY]) {
                    rank[rootX]++;
                }
            }
        }
    }
    // 判断x和y是否属于同一集合
    bool isConnected(int x, int y) {
        return find(x) == find(y);
    }
};
// 主函数，用于测试
int main() {
    UnionFind uf(10);
    uf.unite(1, 2);
    uf.unite(2, 3);
    uf.unite(4, 5);
    std::cout << "1 and 3 are connected: " << uf.isConnected(1, 3) << std::endl;
 // 输出: 1
```

```
    std::cout << "3 and 5 are connected: " << uf.isConnected(3, 5) << std::endl;
 // 输出: 0
    return 0;
}
```

在这个实现中，使用两个数组来表示森林，其中`parent`数组保存每个节点的父节点，而`rank`数组保存每棵树的秩（大致可以认为是树的高度）。`find`函数用于查找一个元素的代表元素，并应用了路径压缩来优化查找效率。`unite`函数用于合并两个元素所在的集合，并应用了按秩合并来优化树的结构。最后，`isConnected`函数可以用来检查两个元素是否属于同一个集合。

树

二叉树的遍历

前序遍历

递归算法

前序遍历的顺序是：根节点 -> 左子树 -> 右子树。在递归算法中，首先访问根节点，然后递归地进行左子树的前序遍历，接着递归地进行右子树的前序遍历。

```
void preOrder(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    cout << root->data << " ";
    preOrder(root->left);
    preOrder(root->right);
}
```

非递归算法

非递归算法需要手动模拟递归过程，通常利用栈（Stack）来实现。遍历过程中，先访问节点并输出节点值，然后先将右孩子入栈（如果有），再将左孩子入栈（如果有），这样可以保证左孩子先于右孩子处理。

```
void preOrderNonRec(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    stack<BinaryTree*> Stack;
    BinaryTree *tmp = root;
    while(!Stack.empty() || tmp){
        while(tmp != nullptr){
            cout << tmp->data << " ";
            Stack.push(tmp);
            tmp = tmp->left;
        }
        if(!Stack.empty()){
            tmp = Stack.top();
            Stack.pop();
            tmp = tmp->right;
        }
    }
}
```

中序遍历

递归算法

中序遍历的顺序是：左子树 -> 根节点 -> 右子树。在递归算法中，首先递归地进行左子树的中序遍历，然后访问根节点，最后递归地进行右子树的中序遍历。

```
void inOrder(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}
```

非递归算法

在非递归的中序遍历中，使用栈来模拟递归过程。首先将节点的所有左孩子入栈，然后访问节点，并向右移动。

```
void inOrderNonRec(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    stack<BinaryTree*> Stack;
    BinaryTree *tmp = root;
    while(!Stack.empty() || tmp){
        while(tmp != nullptr){
            Stack.push(tmp);
            tmp = tmp->left;
        }
        if(!Stack.empty()){
            tmp = Stack.top();
            Stack.pop();
            cout << tmp->data << " ";
            tmp = tmp->right;
        }
    }
}
```

后序遍历

递归算法

后序遍历的顺序是：左子树 -> 右子树 -> 根节点。在递归算法中，首先递归地进行左子树的后序遍历，接着递归地进行右子树的后序遍历，最后访问根节点。

```
void proOrder(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    proOrder(root->left);
    proOrder(root->right);
    cout << root->data << " ";
}
```

非递归算法

后序遍历的非递归实现是三种遍历中最复杂的，因为在访问完左子树和右子树之后，还需要访问根节点。使用一个前置指针`pre`来记录上一个访问过的节点。

```
void proOrderNonRec(BinaryTree *root){
    if(root == nullptr){
        return;
    }
    stack<BinaryTree*> Stack;
    BinaryTree *tmp = root;
    BinaryTree *pre = nullptr;
    while(!Stack.empty() || tmp){
        while(tmp != nullptr){
            Stack.push(tmp

);
            tmp = tmp->left;
        }
        if(!Stack.empty()){
            tmp = Stack.top();
            // 仅当当前节点的右子节点为空或已经被访问过，才能访问当前节点
            if(tmp->right == nullptr || pre == tmp->right){
                cout << tmp->data << " ";
                Stack.pop();
                pre = tmp;
                tmp = nullptr;
            } else {
                // 否则，处理右子树
                tmp = tmp->right;
            }
        }
    }
}
```

在后序遍历的非递归实现中，需要特别注意处理节点的右子树，并且需要使用一个额外的指针来记录最近访问过的节点，以避免重复访问。

层序遍历

```
void levelOrderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    std::queue<TreeNode*> queue;
    queue.push(root);
    while (!queue.empty()) {
        TreeNode* node = queue.front();
        queue.pop();
        std::cout << node->val << " ";
        if (node->left) queue.push(node->left);
        if (node->right) queue.push(node->right);
    }
}
```

二叉搜索树

二叉搜索树（Binary Search Tree, BST）是一种特殊的二叉树，它支持许多动态集合操作，包括搜索、插入、删除等。在二叉搜索树中，左子树上的所有节点的值均小于它的根节点的值，右子树上的所有节点的值均大于它的根节点的值。

查找

查找操作是在树中查找一个指定的值，如果该值存在，返回该节点，否则返回null。

```
BinaryTree* find(BinaryTree* root, int target) {
    if (root == nullptr || root->data == target) {
        return root;
    }
    if (root->data < target) {
        return find(root->right, target);
    }
    return find(root->left, target);
}
```

插入

插入操作是将一个新值插入到二叉搜索树中。插入过程需要保证树的性质：即左子树的所有节点值小于根节点值，右子树的所有节点值大于根节点值。

```
bool insert(BinaryTree*& root, int data) {
    if (root == nullptr) {
        root = new BinaryTree(data);
        return true;
    }
    BinaryTree* tmp = root;
    while (tmp) {
        if (data < tmp->data) {
            if (!tmp->left) {
                tmp->left = new BinaryTree(data);
                return true;
            }
            tmp = tmp->left;
        } else if (data > tmp->data) {
            if (!tmp->right) {
                tmp->right = new BinaryTree(data);
                return true;
            }
```

```
            tmp = tmp->right;
        } else {
            // 如果元素已存在，不插入，返回false。
            return false;
        }
    }
    return true;
}
```

删除

删除操作用于从二叉搜索树中删除一个节点。如果节点有两个子节点，通常的做法是用其右子树中的最小节点（或左子树中的最大节点）来替换它。

```
BinaryTree* deleteNode(BinaryTree*& root, int target) {
    if (root == nullptr) {
        return nullptr;
    }
    if (target < root->data) {
        root->left = deleteNode(root->left, target);
    } else if (target > root->data) {
        root->right = deleteNode(root->right, target);
    } else {
        if (root->left != nullptr && root->right != nullptr) {
            BinaryTree* tmp = findMin(root->right);
            root->data = tmp->data;
            root->right = deleteNode(root->right, tmp->data);
        } else {
            BinaryTree* tmp = root;
            if (root->left == nullptr) {
                root = root->right;
            } else if (root->right == nullptr) {
                root = root->left;
            }
            delete tmp;
        }
    }
    return root;
}
```

最值

最值操作是找到树中的最大或最小元素。在二叉搜索树中，最小元素位于最左侧，最大元素位于最右侧。

```
// 查找最大值
BinaryTree* findMax(BinaryTree* root) {
    if (root == nullptr) {
        return nullptr;
    }
    while (root->right != nullptr) {
        root = root->right;
    }
    return root;
}
```

```
// 查找最小值
BinaryTree* findMin(BinaryTree* root) {
    if (root == nullptr) {
        return nullptr;
    }
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}
```

完全二叉树

完全二叉树（Complete Binary Tree）是一种特殊的二叉树，其中每一层都是完全填满的，除了可能的最后一层。在最后一层，所有的节点都尽可能地靠左排列。以下是完全二叉树的一些重要特性和相关操作的说明：

特性

1. **节点编号**：在完全二叉树中，如果按从上至下、从左至右的顺序对节点编号（从1开始），对于任意节点编号为 `i` 的节点：
   - 父节点的编号为 `i/2`（如果 `i` 是偶数，则其为左孩子；如果 `i` 是奇数，则其为右孩子）。
   - 左孩子的编号为 `2*i`。
   - 右孩子的编号为 `2*i + 1`。
   - 这种编号特性使得完全二叉树特别适合用数组来表示。
2. **高度平衡**：在完全二叉树中，除了最后一层外，其他每一层的节点数都达到最大值。最后一层的节点尽可能集中在左侧。
3. **节点数量与高度关系**：一个高度为 `h` 的完全二叉树，最多有 `2^h - 1` 个节点（`h` 从1开始计数）。

应用

完全二叉树经常用于实现优先队列（Priority Queue），特别是二叉堆（Binary Heap）的形式，因为它可以有效地支持插入（Insert）、删除（Delete）等操作。

示例

一个完全二叉树的例子（节点值即为节点编号）：

```
        1
      /   \
     2     3
    / \   /
   4   5 6
```

在这个例子中：

- 节点1是根节点。
- 节点2和3是节点1的左孩子和右孩子。
- 节点4和5是节点2的左孩子和右孩子，节点6是节点3的左孩子。
- 最后一层（第3层）没有完全填满，但所有的节点都尽可能地靠左排列。

代码示例

```
class CompleteBinaryTree {
private:
    vector<int> tree; // 使用数组来存储树的节点，索引0不使用
public:
    CompleteBinaryTree(const vector<int> &nodes) : tree(nodes.size() + 1) {
        for (size_t i = 0; i < nodes.size(); ++i) {
            tree[i + 1] = nodes[i]; // 将节点值填入数组，从索引1开始
        }
    }

    int getParent(int index) {
        if (index > 1 && index < tree.size()) {
            return tree[index / 2];
        }
        return -1; // 无效或根节点
    }

    int getLeftChild(int index) {
        int leftIndex = 2 * index;
        if (leftIndex < tree.size()) {
            return tree[leftIndex];
        }
        return -1; // 没有左孩子
    }

    int getRightChild(int index) {
        int rightIndex = 2 * index + 1;
        if (rightIndex < tree.size()) {
            return tree[rightIndex];
        }
        return -1; // 没有右孩子
    }
};

int main() {
    CompleteBinaryTree cbt({1, 2, 3, 4, 5, 6}); // 创建一个完全二叉树
    cout << "Left child of 2: " << cbt.getLeftChild(2) << endl; // 输出: 4
    cout << "Right child of 2: " << cbt.getRightChild(2) << endl; // 输出: 5
    cout << "Parent of 5: " <<

 cbt.getParent(5) << endl; // 输出: 2
    return 0;
}
```

平衡二叉树

平衡二叉树（也称为AVL树）是一种自平衡的二叉搜索树。在这种树中，任何节点的两个子树的高度最多相差1，这一属性保证了树的平衡，从而保证了各种操作（查找、插入、删除）的时间复杂度在最坏情况下仍然是O(log n)。为了维护这种高度平衡，AVL树在进行插入和删除操作后可能需要通过旋转来重新平衡自己。

求树高

计算树的高度是判断树是否平衡以及如何旋转的基础。

```
int depth(BinaryTree* root) {
    if(root == nullptr) {
        return 0;
    }
    int left = depth(root->left);
    int right = depth(root->right);
    return max(left, right) + 1;
}
```

旋转操作

AVL树通过四种旋转操作来维护平衡：左旋（LL）、右旋（RR）、左右旋（LR）、右左旋（RL）。

1. **右单旋（RR）**：

   用于处理右子树的右子树插入情况，导致失衡。

   ```
   BinaryTree* RR(BinaryTree* root) {
       BinaryTree* right = root->right;
       root->right = right->left;
       right->left = root;
       return right;
   }
   ```

2. **左单旋（LL）**：

   用于处理左子树的左子树插入情况，导致失衡。

   ```
   BinaryTree* LL(BinaryTree* root) {
       BinaryTree* left = root->left;
       root->left = left->right;
       left->right = root;
       return left;
   }
   ```

3. **先右后左旋（LR）**：

   用于处理左子树的右子树插入情况，导致失衡。

   ```
   BinaryTree* LR(BinaryTree* root) {
       root->left = RR(root->left);
       return LL(root);
   }
   ```

4. **先左后右旋（RL）**：

   用于处理右子树的左子树插入情况，导致失衡。

   ```
   BinaryTree* RL(BinaryTree* root) {
       root->right = LL(root->right);
       return RR(root);
   }
   ```

插入操作

插入操作是平衡二叉树中最复杂的操作之一，因为它可能需要在插入后进行一系列旋转来保持树的平衡。

```
BinaryTree* insert(BinaryTree* root, int X) {
    if(root == nullptr) {
        root = new BinaryTree(X);
        return root;
    }
    if(root->data < X) {
        root->right = insert(root->right, X);
        if(depth(root->right) - depth(root->left) > 1) {
            if(X > root->right->data) {
                root = RR(root);
            } else {
                root = RL(root);
            }
        }
    } else if(root->data > X) {
        root->left = insert(root->left, X);
        if(depth(root->left) - depth(root->right) > 1) {
            if(X < root->left->data) {
                root = LL(root);
            } else {
                root = LR(root);
            }
        }
    }
    // 如果root->data == X，即元素已存在，这里不做处理
    return root;
}
```

红黑树

红黑树是一种自平衡的二叉搜索树，每个节点包含一个颜色属性，可以是红色或黑色。红黑树通过强制维护特定的平衡条件来保证操作的最坏时间复杂度为O(log n)。

特性：

1. **节点颜色**：每个节点要么是红色，要么是黑色。
2. **根节点性质**：根节点是黑色。
3. **叶子节点性质**：所有叶子节点（NIL节点，空节点）是黑色。
4. **红色节点性质**：如果一个节点是红色的，那么它的两个子节点都是黑色的（不会有两个连续的红色节点）。
5. **黑色高度性质**：从任一节点到其每个叶子的所有路径上包含相同数目的黑色节点。

操作：

- 插入、删除操作可能会违反红黑树的特性，因此需要通过旋转和重新着色来调整树，确保继续满足红黑树的特性。
- 插入、删除和查找操作的时间复杂度保持在O(log n)。

B树

B树是一种自平衡的多路搜索树，适合用于读写相对较大的数据块的存储系统。B树通过保持数据有序且平衡来优化访问磁盘数据的过程。

特性：

1. **节点最大和最小度数**：定义了节点的最大和最小子节点数目。B树的度数通常与磁盘页的大小相关。
2. **所有叶子在同一层**：所有的叶子节点都位于同一层。
3. **节点关键字数**：一个节点所包含的关键字数在特定范围内，通常是节点度数的函数。
4. **数据存储在叶子节点**（在某些B树变种中）：所有的数据都存储在叶子节点，内部节点只存储关键字和子节点指针。

操作：

- 查找、插入和删除操作的时间复杂度为O(log n)。
- B树通过在节点中保持多个键和子节点的链接来减少树的高度，从而优化大量数据的存储和检索。

B+树

B+树是B树的变种，广泛应用于数据库和文件系统。

特性：

1. **所有关键字都出现在叶子节点**：叶子节点包含了所有关键字的信息，以及指向记录的指针。
2. **内部节点关键字作为分割点**：内部节点的关键字仅作为分割点，用于引导搜索，内部节点不保存数据记录的指针。
3. **叶子节点形成链表**：所有叶子节点都通过指针连接，形成一个有序链表，便于范围查询。

操作：

- 查找、插入和删除操作的时间复杂度为O(log n)。
- B+树通过将数据全部存储在叶子节点并通过链表连接来优化范围搜索和顺序访问，同时保持了B树的所有优点，如低树高和数据平衡。

图

图的表示

邻接矩阵

使用二维数组表示图中节点之间的连接关系。如果`matrix[i][j]`非零，则表示节点i和节点j之间存在边。

邻接表

定义图中边的结构体`EdgeNode`，每个`EdgeNode`包含指向邻接节点的索引（或指针）和指向下一条边的指针：

```
struct EdgeNode {
    int adjIndex; // 邻接点索引
    int weight; // 边的权重，用于加权图，不需要时可以省略
    EdgeNode* next; // 指向下一条边的指针
    EdgeNode(int adj, int w, EdgeNode* n) : adjIndex(adj), weight(w), next(n) {}
};
```

定义图中节点的结构体 `VertexNode`，每个 `VertexNode` 包含节点数据和一个指向其边链表第一条边的指针：

```
struct VertexNode {
    int data; // 节点的数据
    EdgeNode* firstEdge; // 指向第一条依附于该节点的边
    VertexNode() : data(0), firstEdge(nullptr) {}
    VertexNode(int d) : data(d), firstEdge(nullptr) {}
};
```

可以定义一个图的类，包含所有顶点的列表（这里使用向量来表示），并提供添加边、节点等操作的方法：

```
class Graph {
private:
    vector<VertexNode> vertices; // 所有顶点的列表

public:
    Graph(int numVertices) {
        // 初始化有固定数量顶点但没有边的图
        for(int i = 0; i < numVertices; ++i) {
            vertices.push_back(VertexNode(i));
        }
    }

    void addEdge(int start, int end, int weight) {
        // 向图中添加边，从start到end，权重为weight
        EdgeNode* newNode = new EdgeNode(end, weight, nullptr);
        newNode->next = vertices[start].firstEdge; // 将新节点插入到链表的头部
        vertices[start].firstEdge = newNode;
    }

    // 其他图相关的方法，比如遍历等
};
```

你的笔记涉及图的基础知识，包括图的表示方法、图的遍历算法（DFS和BFS）以及相应的C++代码实现。我将对代码进行适当的重构，主要是修改变量名以增强代码的可读性，同时保持代码逻辑不变。

图的遍历

深度优先搜索 (DFS)

邻接表形式的DFS

```
void DFS(vector<GNode>& graph, vector<bool>& isVisited, int currentVertex){
    isVisited[currentVertex] = true;
    cout << VertexData[currentVertex] << " ";
    for(AdjNode *edge = graph[currentVertex].firstEdge; edge != nullptr; edge =
edge->next){
        if(!isVisited[edge->adjV]){
            DFS(graph, isVisited, edge->adjV);
        }
    }
}
```

邻接矩阵形式的DFS

```
void DFS(vector<vector<int>>& graph, vector<bool>& isVisited, int currentVertex)
{
    isVisited[currentVertex] = true;
    cout << VertexData[currentVertex] << " ";
    for(int adjacent = 0; adjacent < graph.size(); adjacent++){
        if(graph[currentVertex][adjacent] != 0 && !isVisited[adjacent]){
            DFS(graph, isVisited, adjacent);
        }
    }
}
```

广度优先搜索 (BFS)

邻接表形式的BFS

```
void BFS(vector<GNode>& graph, vector<bool>& isVisited, int startVertex){
    queue<int> vertexQueue;
    vertexQueue.push(startVertex);
    isVisited[startVertex] = true;
    while(!vertexQueue.empty()){
        int currentVertex = vertexQueue.front();
        cout << VertexData[currentVertex] << " ";
        vertexQueue.pop();
        for(AdjNode *edge = graph[currentVertex].firstEdge; edge != nullptr;
edge = edge->next){
            if(!isVisited[edge->adjV]){
                isVisited[edge->adjV] = true;
                vertexQueue.push(edge->adjV);
            }
        }
    }
}
```

邻接矩阵形式的BFS

```
void BFS(vector<vector<int>>& graph, vector<bool>& isVisited, int startVertex){
    queue<int> vertexQueue;
    vertexQueue.push(startVertex);
    isVisited[startVertex] = true;
    while(!vertexQueue.empty()){
        int currentVertex = vertexQueue.front();
        cout << VertexData[currentVertex] << " ";
        vertexQueue.pop();
        for(int adjacent = 0; adjacent < graph.size(); adjacent++){
            if(graph[currentVertex][adjacent] != 0 && !isVisited[adjacent]){
                isVisited[adjacent] = true;
                vertexQueue.push(adjacent);
            }
        }
    }
}
```

最短路径Dijkstra算法

Dijkstra算法用于在加权图中找到一个顶点到其他所有顶点的最短路径。

```
/* 邻接矩阵的Dijkstra算法 */
vector<int> Dijkstra(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    const int INF = INT_MAX; // 用INT_MAX代表无穷大
    vector<bool> visited(n, false);
    vector<int> dist(n, INF); // 存储从起点到每个点的最短距离
    vector<int> path(n, -1); // 存储最短路径的前驱节点
    dist[start] = 0; // 起点到自身的最短距离是0

    for (int j = 0; j < n; ++j) {
        int u = -1;
        int minDist = INF;
        // 找到未访问的距离最小的节点
        for (int i = 0; i < n; ++i) {
            if (!visited[i] && dist[i] < minDist) {
                minDist = dist[i];
                u = i;
            }
        }
        // 如果找不到则退出循环
        if (u == -1) break;
        visited[u] = true;

        // 更新未访问节点的最短距离
        for (int v = 0; v < n; ++v) {
            if (!visited[v] && graph[u][v] != INF && u != v) {
                if (dist[v] > dist[u] + graph[u][v]) {
                    dist[v] = dist[u] + graph[u][v];
                    path[v] = u; // 记录最短路径的前驱节点
                }
            }
        }
    }
    return path;
}
```

最短路径Floyd算法

Floyd算法用于计算图中所有顶点对的最短路径。

```
/* 邻接矩阵的Floyd算法 */
vector<int> Floyd(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    const int INF = INT_MAX; // 用INT_MAX代表无穷大
    vector<vector<int>> dist = graph; // dist数组初始化为邻接矩阵的值

    // 初始化距离矩阵，设置自身到自身的距离为0
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) dist[i][j] = 0;
            else if (dist[i][j] == 0) dist[i][j] = INF;
        }
```

```
    }

    // Floyd算法核心，动态更新最短路径
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // 提取start到所有点的最短距离
    vector<int> shortestDistances(n, INF);
    for (int i = 0; i < n; ++i) {
        shortestDistances[i] = dist[start][i];
    }
    return shortestDistances;
}
```

最小生成树Prim算法

Prim算法是一种用来寻找图中的最小生成树的算法。

```
struct MST {
    vector<int> path;
    int weight;
};

// Prim算法: 返回一个最小生成树
MST* Prim(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    const int INF = INT_MAX; // 用INF表示无穷大
    vector<int> dist(n, INF); // 距离数组
    vector<int> path(n, -1); // 最小生成树的路径
    int weight = 0; // 最小生成树的总权重
    int cnt = 1; // 已经加入的节点数
    for (int i = 0; i < n; i++) {
        dist[i] = graph[start][i];
    }
    dist[start] = 0; // 起点到自己的距离为0

    for (int j = 0; j < n - 1; j++) { // 需要加入n-1个节点
        int u = -1;
        int minDist = INF;
        // 找出未加入生成树的，距离最小的节点
        for (int i = 0; i < n; i++) {
            if (dist[i] != 0 && dist[i] < minDist) {
                minDist = dist[i];
                u = i;
            }
        }
        if (u == -1) break; // 如果找不到则退出循环

        weight += dist[u]; // 累加权重
```

```
            dist[u] = 0; // 将节点u加入到生成树中
            cnt++; // 已加入节点数加1

            // 更新与节点u相连的节点到生成树的距离
            for (int i = 0; i < n; i++) {
                if (graph[u][i] != INF && dist[i] != 0 && dist[i] > graph[u][i]) {
                    dist[i] = graph[u][i];
                    path[i] = u; // 记录最小生成树的路径
                }
            }
        }

        // 检查所有节点是否都被加入
        if (cnt != n) {
            return nullptr; // 如果未全部加入，返回空
        } else {
            MST* mst = new MST;
            mst->path = path;
            mst->weight = weight;
            return mst;
        }
    }
```

最小生成树Kruskal算法

Kruskal算法是另一种寻找最小生成树的算法，适用于边的权重不一致的情况。

```
// 定义边的结构体
struct Edge {
    int start, end, weight;
    bool operator > (const Edge &other) const {
        return this->weight > other.weight;
    }
};

// 并查集查找函数
int Find(int x, vector<int>& parent) {
    return x == parent[x] ? x : parent[x] = Find(parent[x], parent);
}

// Kruskal算法
int Kruskal(priority_queue<Edge, vector<Edge>, greater<Edge>>& edges, int n) {
    vector<int> parent(n + 1); // 并查集数组
    iota(parent.begin(), parent.end(), 0); // 初始化并查集
    int totalWeight = 0; // 最小生成树的总权重
    int cnt = 0; // 记录加入的边数

    while (!edges.empty()) {
        Edge minEdge = edges.top();
        edges.pop();
        int root1 = Find(minEdge.start, parent);
        int root2 = Find(minEdge.end, parent);

        // 如果两个节点的根不相同，则没有形成环，可以加入该边
        if (root1 != root2) {
            parent[root1] = root2; // 合并两个集合
            totalWeight += minEdge.weight; // 累加权重
```

```
            cnt++;
        }
    }

    // 检查是否所有节点都在生成树中
    return cnt == n - 1 ? totalWeight : -1

; // 如果不是，返回-1
}

// 算法入口函数
int Solve(int n, int m) {
    priority_queue<Edge, vector<Edge>, greater<Edge>> edges; // 存储所有边的最小堆
    for (int i = 0; i < m; i++) {
        int start, end, weight;
        cin >> start >> end >> weight;
        edges.push({start, end, weight});
    }
    cout << Kruskal(edges, n);
}
```

拓扑排序

```
// 邻接矩阵的拓扑排序，返回拓扑序列
vector<int> topSort(vector<vector<int>>& graph) {
    int n = graph.size();
    const int INF = INT_MAX; // 用INT_MAX表示无穷大，即不存在的边
    vector<int> inDegree(n, 0); // 存储每个节点的入度
    vector<int> topoSequence; // 存储拓扑排序的序列
    queue<int> zeroInDegreeNodes; // 存储入度为0的节点
    int count = 0; // 已处理的节点数

    // 统计每个节点的入度
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (graph[i][j] != INF) {
                inDegree[j]++;
            }
        }
    }

    // 将所有入度为0的节点加入队列
    for (int i = 0; i < n; ++i) {
        if (inDegree[i] == 0) {
            zeroInDegreeNodes.push(i);
        }
    }

    // 开始处理队列中的节点
    while (!zeroInDegreeNodes.empty()) {
        int u = zeroInDegreeNodes.front();
        zeroInDegreeNodes.pop();
        topoSequence.push_back(u); // 将当前节点加入拓扑序列
        count++; // 处理的节点数加1

        // 减少所有由u出发的边所到达节点的入度
        for (int i = 0; i < n; ++i) {
```

```
            if (graph[u][i] != INF) {
                if (--inDegree[i] == 0) {
                    zeroInDegreeNodes.push(i); // 如果入度减为0，则加入队列
                }
            }
        }
    }

    // 检查是否所有节点都被处理，如果没有，则图中存在环，无法进行拓扑排序
    if (count != n) {
        cout << "不存在拓扑序列！" << endl;
        return {}; // 返回空数组表示无法进行拓扑排序
    }
    return topoSequence; // 返回拓扑排序的序列
}
```

排序

冒泡排序

```
void bubbleSort(vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (nums[j] > nums[j + 1]) {
                swap(nums[j], nums[j + 1]);
            }
        }
    }
}
```

插入排序

```
void insertionSort(vector<int>& nums) {
    int n = nums.size();
    for (int i = 1; i < n; i++) {
        int key = nums[i];
        int j = i - 1;
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = key;
    }
}
```

选择排序

```cpp
void selectionSort(vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (nums[j] < nums[min_index]) {
                min_index = j;
            }
        }
        swap(nums[min_index], nums[i]);
    }
}
```

快速排序

```cpp
void quickSortRecursive(vector<int>& nums, int start, int end) {
    if (start >= end) return;
    int pivot = nums[start];
    int left = start, right = end;
    while (left != right) {
        while (left < right && nums[right] >= pivot) right--;
        while (left < right && nums[left] <= pivot) left++;
        if (left < right) {
            swap(nums[left], nums[right]);
        }
    }
    swap(nums[start], nums[left]);
    quickSortRecursive(nums, start, left - 1);
    quickSortRecursive(nums, left + 1, end);
}
void quickSort(vector<int>& nums) {
    quickSortRecursive(nums, 0, nums.size() - 1);
}
```

堆排序

```cpp
// 将nums数组中以top为根的子堆调整为最大堆
void percDown(vector<int>& nums, int top, int n) {
    int parent, child;
    int tmp = nums[top];

    for (parent = top; parent * 2 + 1 < n; parent = child) {
        child = parent * 2 + 1; // 左孩子索引
        // 确定两个孩子中较大的一个
        if (child + 1 < n && nums[child] < nums[child + 1])
            child++;
        // 如果根节点已经是最大的，结束调整
        if (nums[child] <= tmp)
            break;
        // 否则，继续向下调整
        nums[parent] = nums[child];
    }
    nums[parent] = tmp;
}
```

```
// 堆排序函数
void heapSort(vector<int>& nums) {
    int n = nums.size();
    // 构建最大堆
    for (int i = n / 2 - 1; i >= 0; i--) {
        percDown(nums, i, n);
    }
    // 逐步减少堆的规模，并调整成最大堆
    for (int i = n - 1; i > 0; i--) {
        // 将堆顶元素与最后一个元素交换
        swap(nums[i], nums[0]);
        // 将剩余元素调整成最大堆
        percDown(nums, 0, i);
    }
}
```

归并排序

```
void merge(vector<int>& nums, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
    }
    while (i <= mid) {
        temp[k++] = nums[i++];
    }
    while (j <= right) {
        temp[k++] = nums[j++];
    }
    for (i = left, k = 0; i <= right; i++, k++) {
        nums[i] = temp[k];
    }
}

void mergeSortRecursive(vector<int>& nums, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortRecursive(nums, left, mid);
        mergeSortRecursive(nums, mid + 1, right);
        merge(nums, left, mid, right);
    }
}

void mergeSort(vector<int>& nums) {
    mergeSortRecursive(nums, 0, nums.size() - 1);
}
```

桶排序

```
void bucketSort(vector<int>& nums) {
    int maxVal = *max_element(nums.begin(), nums.end());
    int minVal = *min_element(nums.begin(), nums.end());
    int bucketNum = (maxVal - minVal) / nums.size() + 1;
    vector<vector<int>> bucket(bucketNum);

    for (int i = 0; i < nums.size(); i++) {
        int index = (nums[i] - minVal) / nums.size();
        bucket[index].push_back(nums[i]);
    }
    for (int i = 0; i < bucket.size(); i++) {
        sort(bucket[i].begin(), bucket[i].end()); // 使用标准排序算法
    }
    int index = 0;
    for (int i = 0; i < bucket.size(); i++) {
        for (int j = 0; j < bucket[i].size(); j++) {
            nums[index++] = bucket[i][j];
        }
    }
}
```

基数排序

```
void countingSortForRadix(vector<int>& nums, int exp) {
    vector<int> output(nums.size());
    vector<int> count(10, 0);

    for (int i = 0; i < nums.size(); i++) {
        count[(nums[i] / exp) % 10]++;
    }
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    for (int i = nums.size() - 1; i >= 0; i--) {
        output[count[(nums[i] / exp) % 10] - 1] = nums[i];
        count[(nums[i] / exp) % 10]--;
    }
    for (int i = 0; i < nums.size(); i++) {
        nums[i] = output[i];
    }
}

void radixSort(vector<int>& nums) {
    int m = *max_element(nums.begin(), nums.end());
    for (int exp = 1; m / exp > 0; exp *= 10) {
        countingSortForRadix(nums, exp);
    }
}
```

希尔排序

```
void shellSort(vector<int>& nums) {
    int n = nums.size();
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = nums[i];
            int j;
            for (j = i; j >= gap && nums[j - gap] > temp; j -= gap) {
                nums[j] = nums[j - gap];
            }
            nums[j] = temp;
        }
    }
}
```

计数排序

```
void countingSort(vector<int>& nums) {
    if (nums.empty()) return;

    int maxVal = *max_element(nums.begin(), nums.end());
    int minVal = *min_element(nums.begin(), nums.end());
    int range = maxVal - minVal + 1;

    vector<int> count(range, 0);
    vector<int> output(nums.size(), 0);

    // 计算每个元素的出现次数
    for (int num : nums) {
        count[num - minVal]++;
    }

    // 累计计数
    for (int i = 1; i < count.size(); i++) {
        count[i] += count[i - 1];
    }

    // 根据累计计数，将元素放置到正确位置
    for (int i = nums.size() - 1; i >= 0; i--) {
        output[count[nums[i] - minVal] - 1] = nums[i];
        count[nums[i] - minVal]--;
    }

    // 将排序后的数据复制回原数组
    for (int i = 0; i < nums.size(); i++) {
        nums[i] = output[i];
    }
}
```

总结

排序算法	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
冒泡排序	O(n^2)	O(n^2)	O(1)	稳定
选择排序	O(n^2)	O(n^2)	O(1)	数组不稳定、链表稳定
插入排序	O(n^2)	O(n^2)	O(1)	稳定
快速排序	O(n log n)	O(n^2)	O(log n)	不稳定
堆排序	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)	稳定
希尔排序	通常认为是O(n log n)，但依具体增量序列而异	O(n^2)	O(1)	不稳定
计数排序	O(n+m)	O(n+m)	O(n+m)	稳定
桶排序	O(n)	O(n)	O(m)	稳定
基数排序	O(kn)	O(n^2)	取决于内部使用的排序算法	稳定

说明:

- k：代表数值中的“数位”个数。
- n：代表数据规模。
- m：代表数据的最大值减最小值。

解释

1. **冒泡排序**
    - **时间复杂度**: O(n^2)。因为每个元素都需要与其他元素比较，比较次数接近 n*(n-1)/2。
    - **空间复杂度**: O(1)。只需要常数级别的额外空间用于交换元素。
    - **稳定性**: 稳定。相等的元素在排序过程中不会改变顺序。
2. **选择排序**
    - **时间复杂度**: O(n^2)。尽管减少了交换次数，但仍需要比较所有未排序的元素。
    - **空间复杂度**: O(1)。与冒泡排序相似，只需要常数级别的额外空间。
    - **稳定性**: 数组不稳定、链表稳定。数组中相等的元素可能由于选择过程被交换。
3. **插入排序**
    - **时间复杂度**: O(n^2)。在最坏的情况下，每个新元素都可能与已排序的所有元素比较。
    - **空间复杂度**: O(1)。像冒泡和选择排序一样，只需要常数级别的额外空间。
    - **稳定性**: 稳定。插入操作不会改变相等元素的相对顺序。

4. **快速排序**
    - **时间复杂度**: 平均 O(n log n)，但最坏情况下是 O(n^2)。选择的枢轴点会极大影响效率。
    - **空间复杂度**: O(log n)。递归调用堆栈消耗的空间。
    - **稳定性**: 不稳定。枢轴点的选择和元素交换可能会改变相等元素的原始顺序。
5. **堆排序**
    - **时间复杂度**: O(n log n)。建堆过程是 O(n)，之后每个元素的堆调整是 O(log n)。
    - **空间复杂度**: O(1)。排序过程在原数组上进行，不需要额外空间。
    - **稳定性**: 不稳定。元素被移动到堆的末端可能会改变相等元素的原始顺序。
6. **归并排序**
    - **时间复杂度**: O(n log n)。每个元素都会在每层递归中被处理。
    - **空间复杂度**: O(n)。需要与原数组相等大小的额外空间来存储合并后的数组。
    - **稳定性**: 稳定。合并操作能保持相等元素的原始顺序。
7. **希尔排序**
    - **时间复杂度**: 取决于增量序列。最好可以达到 O(n log n)，但某些增量序列可能导致最坏情况下为 O(n^2)。
    - **空间复杂度**: O(1)。排序在原地进行。
    - **稳定性**: 不稳定。较远距离的元素可以互换。
8. **计数排序**
    - **时间复杂度**: O(n+m)。n 是元素数量，m 是数据范围。
    - **空间复杂度**: O(n+m)。需要额外空间来存储计数和输出。
    - **稳定性**: 稳定。计数排序通过计算小于等于每个元素的元素数量来确定每个元素的位置。
9. **桶排序**
    - **时间复杂度**: O(n)。在数据均匀分布的情况下可以达到线性时间，但最坏情况下可能退化为 O(n^2)。
    - **空间复杂度**: O(m)。m 是桶的数量。
    - **稳定性**: 稳定。相同元素会被放在同一个桶中并保持原有顺序。
10. **基数排序**
    - **时间复杂度**: O(kn)。k 是数字的最大位数。
    - **空间复杂度**: 取决于内部使用的排序算法。
    - **稳定性**: 稳定。每一位的排序都保持了前一位排序的顺序。

总结:

- **稳定性**是考量排序算法的一个重要方面，特别是在排序的键是复合键的情况下。一个排序算法是稳定的，如果两个具有相等关键字的元素，在排序后的输出中保持它们在输入中的相对顺序。换句话说，如果在排序前的序列中，元素A出现在元素B之前，并且A和B的关键字相同（即它们是相等的），那么在排序后的序列中，A仍然会出现在B之前。
- 排序算法的选择不仅取决于时间和空间复杂度，还可能受到数据规模、数据的初始状态、内存使用限制等多种因素的影响。
- 对于小数据集，简单排序（如插入排序）可能比更复杂的排序（如快速排序、归并排序）更有效。
- 对于大数据集，通常更倾向于时间复杂度较低的算法，如快速排序、堆排序或归并排序。