目录

前言

最近期末考试，整理了这个学期数据结构与算法所学到的所有代码模板，供自己复习，也供大家参考。文中所有代码均使用C++编写。

作者水平有限，若各位发现代码有误或者有可以改进的地方，欢迎在评论区留言告诉我，谢谢！

文中所有代码已经同步上传到我的GitHub上，大家也可以去这里看：

一、线性表

1、顺序表

```
#include<iostream>
#include<stdio.h>
using namespace std;
//顺序表
class SeqList
{
private:
        int* list;   //数组
        int maxsize;   //最大表长
        int size;      //当前表长
public:
        SeqList();
        ~SeqList();
        void list_int(int n);  //输入n个数，并依次存入顺序表中从0开始的位置
        int list_size();   //计算并返回当前顺序表的表长
        void list_insert(int i, int item);  //把item插入到顺序表的第i个位置
        void list_del(int i);  //删除顺序表中第i个数据
        void list_get(int i);  //打印顺序表中第i个数据
        void list_display();   //打印整个顺序表中的内容
        //插入多个数据的list_multiinsert函数，实现在第i个位置，连续插入来自数组item的n个数据。
        //即从位置i开始插入多个数据。
        void list_multiinsert(int i, int n, int* item);
        //删除多个数据的list_multidel函数，实现从第i个位置开始，连续删除n个数据。
        //即从位置i开始删除多个数据。
        void list_multidel(int i, int n);
        void list_sort();  //顺序表排序函数
        int* list_getlist();  //返回数组指针，与list_combine函数配合
        int list_getsize();   //返回数组长度，与list_combine函数配合
        SeqList list_combine(SeqList& a, SeqList& b);  //合并两个顺序表
        void list_move(int d, int n);  //顺序表循环移位函数
};
//构造函数
SeqList::SeqList()
{
        maxsize = 1000;
        size = 0;
        list = new int[maxsize];
}
//析构函数
```

```
SeqList::~SeqList()
{
	delete[]list;
}
//输入n个数，并依次存入顺序表中从0开始的位置
void SeqList::list_int(int n)
{
	for (int i = 0; i < n; i++)
	{
		cin >> list[i];
	}
	size = n;
}
//计算并返回当前顺序表的表长
int SeqList::list_size()
{
	for (int i = 0; i < 1000; i++)
	{
		if (list[i] == '\0')
			size = i + 1;
	}
	return size;
}
//把item插入到顺序表的第i个位置
void SeqList::list_insert(int i, int item)
{
	if ((i > 0) && (i <= (size + 1)))
	{
		for (int k = size; k >= i; k--)
		{
			list[k] = list[k - 1];
		}
		list[i - 1] = item;
		size++;
		list_display();
	}
	else
		cout << "error" << endl;
}
//删除顺序表中第i个数据
void SeqList::list_del(int i)
{
	if ((i > 0) && (i <= size))
```

```
	{
		size--;
		for (int k = i - 1; k < size; k++)
		{
			list[k] = list[k + 1];
		}
		list_display();
	}
	else
		cout << "error" << endl;
}
//打印顺序表中第i个数据
void SeqList::list_get(int i)
{
	if ((i > 0) && (i <= size))
		cout << list[i - 1] << endl;
	else
		cout << "error" << endl;
}
//打印整个顺序表中的内容
void SeqList::list_display()
{
	//先打印顺序表的表长
	cout << size << " ";
	for (int i = 0; i < size; i++)
	{
		cout << list[i] << " ";
	}
	cout << endl;
}
//插入多个数据的list_multiinsert函数，实现在第i个位置，连续插入来自数组item的n个数据。
//即从位置i开始插入多个数据。
void SeqList::list_multiinsert(int i, int n, int* item)
{
	if ((i > 0) && (i <= (size + 1)))
	{
		for (int k = size + n - 1; k > i; k--)
		{
			list[k] = list[k - n];
		}
		for (int k = i - 1; k < i - 1 + n; k++)
		{
			list[k] = item[k - i + 1];
```

```
                }
                size += n;
                list_display();
        }
        else
                cout << "error" << endl;
}
//删除多个数据的list_multidel函数，实现从第i个位置开始，连续删除n个数据。
//即从位置i开始删除多个数据。
void SeqList::list_multidel(int i, int n)
{
        if ((i > 0) && (i <= size))
        {
                size -= n;
                for (int k = i - 1; k < size; k++)
                {
                        list[k] = list[k + n];
                }
                list_display();
        }
        else
                cout << "error" << endl;
}
//顺序表排序函数
void SeqList::list_sort()
{
        for (int i = 0; i < size - 1; i++)
        {
                for (int i = 0; i < size - 1; i++)
                {
                        if (list[i] > list[i + 1])
                        {
                                int mid;
                                mid = list[i];
                                list[i] = list[i + 1];
                                list[i + 1] = mid;
                        }
                }
        }
}
//返回数组指针，与list_combine函数配合
int* SeqList::list_getlist()
{
```

```
	return list;
}
//返回数组长度，与list_combine函数配合
int SeqList::list_getsize()
{
	list_size();
	return size;
}
//合并两个顺序表
SeqList SeqList::list_combine(SeqList& a, SeqList& b)
{
	SeqList sum;
	sum.list_multiinsert(0, a.list_getsize(), a.list_getlist());
	sum.list_multiinsert(1, b.list_getsize(), b.list_getlist());
	sum.list_sort();
	sum.list_display();
	return sum;
}
//顺序表循环移位函数
void SeqList::list_move(int d, int n)
{
	int* nlist = new int[size];
	if (d == 0)
	{
		for (int i = 0; i < size; i++)
		{
			if (i - n < 0)
			{
				nlist[i - n + size] = list[i];
			}
			else
			{
				nlist[i - n] = list[i];
			}
		}
	}
	else if (d == 1)
	{
		for (int i = 0; i < size; i++)
		{
			if (i + n > size - 1)
			{
				nlist[i + n - size] = list[i];
```

```
                        }
                        else
                        {
                                nlist[i + n] = list[i];
                        }
                }
        }
        list = nlist;
}
```

2、单链表

```
#include<stdio.h>
#include<iostream>
using namespace std;
//线性单链表结点类
class ListNode
{
public:
	int data;   //结点数据
	ListNode* next;    //下一个结点指针
	//构造函数
	ListNode()
	{
		next = NULL;
		data = 0;
	}
};
//线性单链表类
class LinkList
{
public:
	ListNode* head;    //头结点指针
	int len;           //链表长度
	LinkList();        //构造函数
	~LinkList();       //析构函数
	void LL_add(int x);//在链表最后插入数据x
	ListNode* LL_index(int i);   //查找并返回第i个结点的指针
	void LL_get(int i);//打印输出第i个结点的数据
	void LL_insert(int i, int item);//在把数据item插入到链表的第i个结点处
	void LL_del(int i);//删除第i个结点
	void LL_display();//打印输出单链表所有结点
	void swap(int pa, int pb);//结点交换函数
	void sort();  //单链表排序函数
	LinkList LL_merge(LinkList& La, LinkList& Lb);//合并两个单链表生成一个新链表并返回
};
//构造函数
LinkList::LinkList()
{
	head = new ListNode();
	len = 0;
}
//析构函数
```

```
LinkList::~LinkList()
{
	head = NULL;
}
//在链表最后插入数据x
void LinkList::LL_add(int x)
{
	ListNode* l = new ListNode();
	l->data = x;
	if (head->next == NULL)
	{
		head->next = l;
	}
	else
	{
		ListNode* p;
		p = head;
		for (int k = 1; k <= len; k++)
		{
			p = p->next;
		}
		p->next = l;
	}
	len++;
}
//查找并返回第i个结点的指针
ListNode* LinkList::LL_index(int i)
{
	if ((i <= 0) || (i > len))
	{
		cout << "error" << endl;
	}
	else
	{
		ListNode* p;
		p = head;
		for (int k = 1; k <= i; k++)
		{
			p = p->next;
		}
		return p;
	}
}
```

```
//打印输出第i个结点的数据
void LinkList::LL_get(int i)
{
	if ((i <= 0) || (i > len))
	{
		cout << "error" << endl;
	}
	else
	{
		int x;
		x = LL_index(i)->data;
		cout << x << endl;
	}
}
//在把数据item插入到链表的第i个结点处
void LinkList::LL_insert(int i, int item)
{
	if ((i <= 0) || (i > len + 1))
	{
		cout << "error" << endl;
	}
	else
	{
		ListNode* l = new ListNode();
		l->data = item;
		l->next = LL_index(i);
		LL_index(i - 1)->next = l;
		len++;
		LL_display();
	}
}
//删除第i个结点
void LinkList::LL_del(int i)
{
	if ((i <= 0) || (i > len))
	{
		cout << "error" << endl;
	}
	else
	{
		if (i == 1)
		{
			head->next = LL_index(2);
```

```
                }
                else if (i == len)
                {
                        LL_index(i-1)->next = NULL;
                }
                else
                {
                        LL_index(i-1)->next = LL_index(i+1);
                }
                len--;
                LL_display();
        }
}
//打印输出单链表所有结点
void LinkList::LL_display()
{
        ListNode* p;
        p = head;
        while (p->next != NULL)
        {
                p = p->next;
                cout << p->data << " ";
        }
        cout << endl;
}
//结点交换函数
void LinkList::swap(int pa, int pb)
{
        if ((pa != pb) && (pa > 0) && (pa <= len) && (pb > 0) && (pb <= len))
        {
                ListNode* p;
                p = LL_index(pa);

                ListNode* q;
                q = LL_index(pb);

                ListNode* p_before;
                if (pa == 1)
                {
                        p_before = head;
                }
                else
                {
```

```
                p_before = LL_index(pa - 1);
        }

        ListNode* p_after;
        if (pa == len)
        {
                p_after = NULL;
        }
        else
        {
                p_after = LL_index(pa + 1);
        }

        ListNode* q_before;
        if (pb == 1)
        {
                q_before = head;
        }
        else
        {
                q_before = LL_index(pb - 1);
        }

        ListNode* q_after;
        if (pb == len)
        {
                q_after = NULL;
        }
        else
        {
                q_after = LL_index(pa + 1);
        }

        p_before->next = q;
        q->next = p_after;

        q_before->next = p;
        p->next = q_after;
        LL_display();
}
else
{
        cout << "error" << endl;
```

```
        }
}
//单链表排序函数
void LinkList::sort()
{
        for (int i = 1; i <= len; i++)
        {
                for (int k = i; k <= len; k++)
                {
                        if (LL_index(i)->data > LL_index(k)->data)
                        {
                                swap(i, k);
                        }
                }
        }
}
//合并两个单链表生成一个新链表并返回
LinkList LinkList::LL_merge(LinkList& La, LinkList& Lb)
{
        LinkList L;

        ListNode* p;
        p = La.head;
        for (int i = 1; i <= La.len; i++)
        {
                p = p->next;
                L.LL_insert(i, p->data);
        }

        p = Lb.head;
        for (int i = 1; i <= Lb.len; i++)
        {
                p = p->next;
                L.LL_insert(i, p->data);
        }
        //cout << L.len << endl;
        //L.LL_display();
        for (int i = 1; i <= L.len; i++)
        {
                for (int k = i; k <= L.len; k++)
                {
                        //cout << "i=" << i << endl;
                        //cout << "k=" << k << endl;
```

```
                        if (L.LL_index(i)->data > L.LL_index(k)->data)
                        {
                                L.swap(i, k);
                                //L.LL_display();
                        }
                }
        }
        return L;
}
```

3、循环链表

```
#include<iostream>
#include<stdio.h>
using namespace std;
//循环链表结点类
class ListNode
{
public:
	int data;  //结点数据
	ListNode* next;  //下一个结点的指针
	//构造函数
	ListNode()
	{
		next = NULL;
		data = 0;
	}
};
//循环链表类
class LinkList
{
public:
	ListNode* head;  //头结点指针
	int len;  //循环链表长度
	LinkList();
	void LL_add(int x);  //在循环链表尾部插入数据x
	ListNode* LL_index(int i);  //查找并返回第i个结点的指针
	void LL_del(int i);  //删除第i个结点
	void LL_display();  //打印输出循环链表所有结点
};
//构造函数
LinkList::LinkList()
{
	head = new ListNode();
	len = 0;
}
//在循环链表尾部插入数据x
void LinkList::LL_add(int x)
{
	ListNode* l = new ListNode();
	l->data = x;
	l->next = head;
	if (head->next == NULL)
```

```
        {
                head->next = l;
        }
        else
        {
                LL_index(len)->next = l;
        }
        len++;
}
//查找并返回第i个结点的指针
ListNode* LinkList::LL_index(int i)
{
        if (i > len)
        {
                i = i - len;
                ListNode* p;
                p = head;
                for (int k = 1; k <= i; k++)
                {
                        p = p->next;
                }
                return p;
        }
        else
        {
                ListNode* p;
                p = head;
                for (int k = 1; k <= i; k++)
                {
                        p = p->next;
                }
                return p;
        }
}
//删除第i个结点
void LinkList::LL_del(int i)
{
        if ((i <= 0) || (i > len))
        {
                cout << "error" << endl;
        }
        else
        {
```

```
                if (i == 1)
                {
                        head->next = LL_index(2);
                }
                else if (i == len)
                {
                        LL_index(i - 1)->next = head;
                }
                else
                {
                        LL_index(i - 1)->next = LL_index(i + 1);
                }
                len--;
        }
}
//打印输出循环链表所有结点
void LinkList::LL_display()
{
        ListNode* p;
        p = head;
        for (int i = 1; i <= len; i++)
        {
                p = p->next;
        }
        cout << endl;
}
```

4、双向链表

```
#include<iostream>
using namespace std;
//双向链表结点类
class ListNode
{
public:
	ListNode* front;  //前一个结点的指针
	char data;  //结点数据
	ListNode* next;  //下一个结点的指针
	//构造函数
	ListNode()
	{
		front = this;
		next = this;
		data = 0;
	}
};
//双向链表类
class LinkList
{
public:
	ListNode* head;  //头结点指针
	int len;  //双向链表长度
	LinkList();
	void LL_add(char x);  //在双向链表末尾添加一个结点
	ListNode* LL_index(int i);  //返回第i个结点
	void LL_insert(int i, char item);  //将item插入双向链表的第i个结点
	void LL_del(int i);  //删除双向链表中的第i个结点
	void LL_display();  //打印输出双向链表所有结点
};
//构造函数
LinkList::LinkList()
{
	head = new ListNode();
	len = 0;
}
//在双向链表末尾添加一个结点
void LinkList::LL_add(char x)
{
	ListNode* l = new ListNode();
	l->data = x;
```

```
        l->next = head;
        if (head->next == head)
        {
                l->front = head;
                head->next = l;
                head->front = l;
        }
        else
        {
                l->front = LL_index(len);
                LL_index(len)->next = l;
                head->front = l;
        }
        len++;
}
//返回第i个结点
ListNode* LinkList::LL_index(int i)
{
        if ((i <= 0) || (i > len))
        {
                cout << "index_error" << endl;
        }
        else
        {
                ListNode* p;
                p = head;
                for (int k = 1; k <= i; k++)
                {
                        p = p->next;
                }
                return p;
        }
}
//将item插入双向链表的第i个结点
void LinkList::LL_insert(int i, char item)
{
        if (i == len)
        {
                LL_add(item);
        }
        else if (i == 1)
        {
                if (len == 0)
```

```
                {
                        LL_add(item);
                }
                else
                {
                        ListNode* l = new ListNode();
                        l->data = item;
                        l->front = head;
                        l->next = LL_index(1);
                        LL_index(1)->front = l;
                        head->next = l;
                        len++;
                }
        }
        else
        {
                ListNode* l = new ListNode();
                l->data = item;
                l->next = LL_index(i);
                LL_index(i)->front = l;
                LL_index(i - 1)->next = l;
                l->front = LL_index(i - 1);
                len++;
        }
}
//删除双向链表中的第i个结点
void LinkList::LL_del(int i)
{
        if ((i <= 0) || (i > len))
        {
                cout << "del_error" << endl;
        }
        else
        {
                if (i == 1)
                {
                        if (len == 1)
                        {
                                head->next = head;
                                head->front = head;
                        }
                        else
                        {
```

```
                                        head->next = LL_index(2);
                                        LL_index(2)->front = head;
                                }
                        }
                        else if (i == len)
                        {
                                LL_index(i - 1)->next = head;
                                head->front = LL_index(i - 1);
                        }
                        else
                        {
                                LL_index(i - 1)->next = LL_index(i + 1);
                                LL_index(i)->front = LL_index(i - 1);
                        }
                        len--;
        }
}
//打印输出双向链表所有结点
void LinkList::LL_display()
{
        ListNode* p;
        p = head;
        if (p->next == head)
        {
                cout << "-";
        }
        while (p->next != head)
        {
                p = p->next;
                cout << p->data;
        }
        cout << endl;
}
```

二、堆栈

```
/*
stack类使用的参考代码
1、包含头文件<stack>：#include <stack>
2、创建一个堆栈对象s（注意stack是模板类）：stack <char>  s;   //堆栈的数据类型是字符型
3、把一个字符ct压入堆栈：  s.push(ct);
4、把栈顶元素弹出：s.pop();
5、获取栈顶元素，放入变量c2：  c2 = s.top();
6、判断堆栈是否空：  s.empty()，如果为空则函数返回true，如果不空则返回false
7、对堆栈进行操作时，一定要记得查看堆栈是否为空
*/
```

三、队列

```
/*
队列queue的用法如下：
1.包含头文件：#include <queue>
2.定义一个整数队列对象：queue<int>  myQe;
3.定义一个整数队列对象数组：queue<int>  myQA[10];
4.入队操作：myQe.push(itemp); //把整数itemp进入队列
5.出队操作：myQe.pop();  //把队头元素弹出队列，注意本操作不获取队头元素
6.获取队头元素：  itemp = myQe.front(); // 把队头元素放入itemp中，注意本操作不弹出元素
7.判断队列是否为空：myQe.empty();//队列空则返回true，不空则返回false
*/
```

四、KMP算法

```
#include<iostream>
using namespace std;
void get_next(string t, int* next)
{
	int i = 0, j = -1;
	next[i] = j;
	while (i < (int)t.length())
	{
		if (j == -1 || t[i] == t[j])
		{
			++i; ++j;
			next[i] = j;
		}
		else
		{
			j = next[j];
		}
	}
}
//kmp算法，查找成功则返回起始位置，查找失败则返回-1
int KMP(string str, string t)
{
	int i = 0, j = 0;
	int* next = new int[t.length() + 1];
	get_next(t, next);
	while (i < (int)str.length() && j < (int)t.length())
	{
		if (j == -1 || str[i] == t[j])
		{
			++i;
			++j;
		}
		else
			j = next[j];
	}
	if (j == (int)t.length())
	{
		return i - j;
	}
	else
```

```
                return -1;
}
```

五、二叉树

1、普通二叉树

```
#include<iostream>
#include<string>
#include<queue>
#include<stack>
using namespace std;
//二叉树结点类
class BiTreeNode
{
public:
          char data;  //结点数据
          BiTreeNode* LeftChild;   //左子树指针
          BiTreeNode* RightChild;  //右子树指针
          //构造函数
          BiTreeNode() :LeftChild(NULL), RightChild(NULL) {}
          //~BiTreeNode();
};

//二叉树类
class BiTree
{
private:
          BiTreeNode* Root;   //根结点指针
          int pos;
          string strTree;

          BiTreeNode* CreateBiTree();
          void PreOrder(BiTreeNode* t);
          void InOrder(BiTreeNode* t);
          void PostOrder(BiTreeNode* t);
          void PostOrder_stack(BiTreeNode* t);
          void LevelOrder(BiTreeNode* t);
public:
          BiTree();
          //~BiTree();
          void CreateTree(string TreeArray);    //利用先序遍历结果创建二叉树
          void PreOrder();    //前序遍历
          void InOrder();     //中序遍历
          void PostOrder();   //后序遍历
          void PostOrder_stack();  //后序遍历非递归算法
```

```
        void LevelOrder();  //层次遍历
};
BiTree::BiTree()
{
        pos = 0;
        strTree = "";
}

//定义先序遍历函数
void BiTree::PreOrder() //公有函数，对外接口
{
        PreOrder(Root);
}
void BiTree::PreOrder(BiTreeNode* t)
{
        if (t != NULL)
        {
                cout << t->data;
                PreOrder(t->LeftChild);
                PreOrder(t->RightChild);
        }
        else
                return;
}

//定义中序遍历函数
void BiTree::InOrder()  //公有函数，对外接口
{
        InOrder(Root);
}
void BiTree::InOrder(BiTreeNode* t)
{
        if (t != NULL)
        {
                InOrder(t->LeftChild);
                cout << t->data;
                InOrder(t->RightChild);
        }
        else
                return;
}

//定义后序遍历函数
```

```
void BiTree::PostOrder()  //公有函数，对外接口
{
	PostOrder(Root);
}
void BiTree::PostOrder(BiTreeNode* t)
{
	if (t != NULL)
	{
		PostOrder(t->LeftChild);
		PostOrder(t->RightChild);
		cout << t->data;
	}
	else
		return;
}

//构造二叉树，利用先序遍历结果建树
void BiTree::CreateTree(string TreeArray) //公有函数，对外接口
{
	pos = 0;
	strTree.assign(TreeArray);
	Root = CreateBiTree();
}
//递归建树，私有函数，类内实现
BiTreeNode* BiTree::CreateBiTree()
{
	BiTreeNode* T;
	char c;
	c = strTree[pos++];
	if (c == '0')
		T = NULL;
	else
	{
		T = new BiTreeNode();
		T->data = c;    //生成根结点
		T->LeftChild = CreateBiTree();    //构造左子树
		T->RightChild = CreateBiTree();   //构造右子树
	}
	return T;
}

//层次遍历函数
void BiTree::LevelOrder()    //公有函数，对外接口
```

```
{
	LevelOrder(Root);
}
void BiTree::LevelOrder(BiTreeNode* t)
{//用队列实现
	queue<BiTreeNode*> tq;
	BiTreeNode* p = t;
	if (p != NULL)
	{
		tq.push(p);
	}
	while (!tq.empty())
	{
		p = tq.front();
		tq.pop();
		if (p != NULL)
		{
			cout << p->data;
			tq.push(p->LeftChild);
			tq.push(p->RightChild);
		}
	}
}

//后序非递归遍历
void BiTree::PostOrder_stack()   //公有函数，对外接口
{
	PostOrder_stack(Root);
}
void BiTree::PostOrder_stack(BiTreeNode* t)
{
	stack<BiTreeNode*> s1;
	stack<int> s2;
	int tag;
	BiTreeNode* p = t;
	do
	{
		while (1)
		{
			if (p != NULL)
			{
				s1.push(p);
				tag = 0;
```

```
                                s2.push(tag);
                                p = p->LeftChild;
                        }
                        else
                                break;
                }

                if (s1.empty())
                        break;
                if (p == NULL)
                {
                        tag = s2.top();
                        if (tag == 0)
                        {
                                s2.top() = 1;
                                p = s1.top()->RightChild;
                        }
                        else
                        {
                                p = s1.top();
                                s1.pop();
                                s2.pop();
                                cout << p->data;
                                p = NULL;
                        }
                }
        } while (!s1.empty());
}
```

2、二叉树—三叉链表

```
#include<iostream>
using namespace std;
//二叉树的三叉链表结点类
class BiTreeNode
{
public:
          char data;  //结点数据
          int balance;  //结点平衡因子
          BiTreeNode* LeftChild;   //左子树指针
          BiTreeNode* RightChild;  //右子树指针
          BiTreeNode* Parent;      //双亲指针
          BiTreeNode() :LeftChild(NULL), RightChild(NULL), Parent(NULL) {}
          //~BiTreeNode();
};

//二叉树的三叉链表类
class BiTree
{
private:
          BiTreeNode* Root;   //根节点指针
          int pos;
          string strTree;

          BiTreeNode* CreateBiTree();
          void PreOrder(BiTreeNode* t);
          void InOrder(BiTreeNode* t);
          void PostOrder(BiTreeNode* t);
          void LevelOrder(BiTreeNode* t, int i);
          BiTreeNode* Create_PreOrder(int i, string str);
          void Height_Subtree(BiTreeNode* t_root, BiTreeNode* t, int& h);
          void Balance_calculate(BiTreeNode* t);
public:
          BiTree();
          //~BiTree();
          void CreateTree(string TreeArray);    //利用先序遍历结果创建二叉树
          void PreOrder();    //前序遍历
          void InOrder();     //中序遍历
          void PostOrder();   //后序遍历
          void LevelOrder();  //层次遍历不使用队列的递归算法
          void CreatePreOrder(int i, string str);  //递归先序遍历顺序数组存储的二叉树
          int LeafDepth(BiTreeNode* t);  //计算叶子结点相对于根结点的深度
```

```
        int LeafDepth(BiTreeNode* t, BiTreeNode* t_root);  //计算叶子结点t相对于t_root结点的深度
        int HeightSubtree(BiTreeNode* t);   //计算子树高度
        void Balance_calculate();  //计算结点平衡因子
};
BiTree::BiTree()
{
        pos = 0;
        strTree = "";
}

//定义先序遍历函数
void BiTree::PreOrder() //公有函数，对外接口
{
        PreOrder(Root);
}
void BiTree::PreOrder(BiTreeNode* t)
{
        if (t != NULL)
        {
                cout << t->data;
                PreOrder(t->LeftChild);
                PreOrder(t->RightChild);
        }
        else
                return;
}

//定义中序遍历函数
void BiTree::InOrder()  //公有函数，对外接口
{
        InOrder(Root);
}
void BiTree::InOrder(BiTreeNode* t)
{
        if (t != NULL)
        {
                InOrder(t->LeftChild);
                cout << t->data;
                InOrder(t->RightChild);
        }
        else
                return;
}
```

```
//定义后序遍历函数
void BiTree::PostOrder()  //公有函数，对外接口
{
	PostOrder(Root);
}
void BiTree::PostOrder(BiTreeNode* t)
{
	if (t != NULL)
	{
		PostOrder(t->LeftChild);
		PostOrder(t->RightChild);
		cout << t->data;
	}
	else
		return;
}

//构造二叉树，利用先序遍历结果建树
void BiTree::CreateTree(string TreeArray) //公有函数，对外接口
{
	pos = 0;
	strTree.assign(TreeArray);
	Root = CreateBiTree();
}
//递归建树，私有函数，类内实现
BiTreeNode* BiTree::CreateBiTree()
{
	BiTreeNode* T;
	char c;
	c = strTree[pos++];
	if (c == '0')
		T = NULL;
	else
	{
		T = new BiTreeNode();
		T->data = c;    //生成根结点
		T->LeftChild = CreateBiTree();    //构造左子树
		if (T->LeftChild != NULL)
			T->LeftChild->Parent = T;
		T->RightChild = CreateBiTree();   //构造右子树
		if (T->RightChild != NULL)
			T->RightChild->Parent = T;
```

```
        }
        return T;
}
//计算叶子结点相对于根结点的深度
int BiTree::LeafDepth(BiTreeNode* t)
{
        int depth = 0;
        BiTreeNode* T = t;
        while (1)
        {
                if (T == Root)
                {
                        return depth;
                }
                else
                {
                        if (T->Parent != NULL)
                        {
                                T = T->Parent;
                                depth++;
                        }
                        else
                        {
                                return depth;
                        }
                }
        }
}
//计算叶子结点t相对于t_root结点的深度
int BiTree::LeafDepth(BiTreeNode* t, BiTreeNode* t_root)
{
        int depth = 0;
        BiTreeNode* T = t;
        while (1)
        {
                if (T->Parent != t_root->Parent)
                {
                        T = T->Parent;
                        depth++;
                }
                else
                {
                        depth++;
```

```
                        return depth;
                }
        }
}
//层次遍历函数
void BiTree::LevelOrder()    //公有函数，对外接口
{
        for (int i = 0; i < 10; i++)
        {
                LevelOrder(Root, i);
        }
}
void BiTree::LevelOrder(BiTreeNode* t, int i)
{
        if (t != NULL)
        {
                if (LeafDepth(t) == i)
                {
                        cout << t->data << " ";
                }
                LevelOrder(t->LeftChild, i);
                LevelOrder(t->RightChild, i);
        }
        else
                return;
}
//递归先序遍历顺序数组存储的二叉树
void BiTree::CreatePreOrder(int i, string str)
{
        Root = Create_PreOrder(i, str);
}
BiTreeNode* BiTree::Create_PreOrder(int i, string str)
{
        BiTreeNode* T;
        if (i < str.length())
        {
                if (str[i] != '0')
                {
                        T = new BiTreeNode;
                        T->data = str[i];
                        T->LeftChild = Create_PreOrder(2 * i + 1, str);
                        if (T->LeftChild != NULL)
                                T->LeftChild->Parent = T;
```

```
                        T->RightChild = Create_PreOrder(2 * i + 2, str);
                        if (T->RightChild != NULL)
                                T->RightChild->Parent = T;
                }
                else
                {
                        T = NULL;
                }
        }
        else
        {
                T = NULL;
        }
        return T;
}
void BiTree::Height_Subtree(BiTreeNode* t_root, BiTreeNode* t, int& h)
{
        if (t != NULL)
        {
                Height_Subtree(t_root, t->LeftChild, h);
                Height_Subtree(t_root, t->RightChild, h);
                if (LeafDepth(t, t_root) >= h)
                        h = LeafDepth(t, t_root);
        }
        else
                return;
}
//计算结点平衡因子
void BiTree::Balance_calculate()  //公有函数，对外接口
{
        Balance_calculate(Root);
}
void BiTree::Balance_calculate(BiTreeNode* t)
{
        if (t != NULL)
        {
                Balance_calculate(t->LeftChild);
                Balance_calculate(t->RightChild);
                t->balance = HeightSubtree(t->LeftChild) - HeightSubtree(t->RightChild);
        }
        else
                return;
}
```

3、赫夫曼树

```
#include<iostream>
using namespace std;
const int MaxW = 9999999;  // 假设结点权值不超过9999999
// 定义huffman树结点类
class HuffNode
{
public:
    int weight;     // 权值
    char ch;        //结点字符
    int parent;     // 父结点下标
    int leftchild;  // 左孩子下标
    int rightchild; // 右孩子下标
};
// 定义赫夫曼树类
class HuffMan
{
private:
    void MakeTree();    // 建树
    // 从 1 到 pos 的位置找出权值最小的两个结点，把结点下标存在 s1 和 s2 中
    void SelectMin(int pos, int* s1, int* s2);
public:
    int len;    // 结点数量
    int lnum;   // 叶子数量
    int f;
    HuffNode* huffTree; // 赫夫曼树，用数组表示
    string* huffCode;   // 每个字符对应的赫夫曼编码
    void MakeTree(int n, int wt[]); // 构建huffman树
    void Coding();  // 赫夫曼编码函数
    void Destroy(); // 销毁赫夫曼树
    void Decode(const string codestr);  //赫夫曼解码
};
// 构建huffman树
void HuffMan::MakeTree(int n, int wt[])
{
    // 参数是叶子结点数量和叶子权值
    // 公有函数，对外接口
    int i;
    lnum = n;
    len = 2 * n - 1;
    huffTree = new HuffNode[2 * n];
    huffCode = new string[lnum + 1];    // 位置从 1 开始计算
```

```
    // huffCode实质是个二维字符数组，第 i 行表示第 i 个字符对应的编码
    // 赫夫曼树huffTree初始化
    for (i = 1; i <= n; i++)
        huffTree[i].weight = wt[i - 1]; // 第0号不用，从1开始编号
    for (i = 1; i <= len; i++)
    {
        if (i > n) huffTree[i].weight = 0;  // 前n个结点是叶子，已经设置
        huffTree[i].parent = 0;
        huffTree[i].leftchild = 0;
        huffTree[i].rightchild = 0;
    }
    MakeTree();  // 调用私有函数建树
    f = 0;
    for (int i = 0; i < n; i++)
    {
        f += wt[i];
    }
}
// 从 1 到 pos 的位置找出权值最小的两个结点，把结点下标存在 s1 和 s2 中
void HuffMan::SelectMin(int pos, int* s1, int* s2)
{
    // 找出最小的两个权值的下标
    // 函数采用地址传递的方法，找出两个下标保存在 s1 和 s2 中
    int w1, w2, i;
    w1 = w2 = MaxW;  // 初始化w1和w2为最大值，在比较中会被实际的权值替换
    *s1 = *s2 = 0;
    for (i = 1; i <= pos; i++)
    {
        // 比较过程如下：
        // 如果第 i 个结点的权值小于 w1，且第 i 个结点是未选择的结点，提示：如果第 i 结点未选择，它父
        // 把第 w1 和 s1 保存到 w2 和 s2，即原来的第一最小值变成第二最小值
        // 把第 i 结点的权值和下标保存到 w1 和 s1，作为第一最小值
        // 否则，如果第 i 结点的权值小于 w2，且第 i 结点是未选择的结点
        // 把第 i 结点的权值和下标保存到 w2 和 s2，作为第二最小值
        if ((huffTree[i].weight < w1) && (huffTree[i].parent == 0))
        {
            w2 = w1;
            *s2 = *s1;
            w1 = huffTree[i].weight;
            *s1 = i;
        }
        else if ((huffTree[i].weight < w2) && (huffTree[i].parent == 0))
        {
```

```
            w2 = huffTree[i].weight;
            *s2 = i;
        }
    }
}
// 建树
void HuffMan::MakeTree()
{
    // 私有函数，被公有函数调用
    int i, s1, s2;
    for (i = lnum + 1; i <= len; i++)
    {
        SelectMin(i - 1, &s1, &s2);  // 找出两个最小权值的下标放入 s1 和 s2 中
        // 将找出的两棵权值最小的子树合并为一棵子树，过程包括
        // 结点 s1 和结点 s2 的父亲设为 i
        // 结点 i 的左右孩子分别设为 s1 和 s2
        // 结点 i 的权值等于 s1 和 s2 的权值和
        huffTree[s1].parent = i;
        huffTree[s2].parent = i;
        huffTree[i].leftchild = s1;
        huffTree[i].rightchild = s2;
        huffTree[i].weight = huffTree[s1].weight + huffTree[s2].weight;
    }
}
// 销毁赫夫曼树
void HuffMan::Destroy()
{
    len = 0;
    lnum = 0;
    delete[]huffTree;
    delete[]huffCode;
}
// 赫夫曼编码
void HuffMan::Coding()
{
    string code; // 存储符号的不定长二进制编码
    int i, j, k, parent;
    for (i = 1; i <= lnum; i++)
    { // 从叶子结点出发
        j = i;
        code = ""; // 初始化为空
        while (huffTree[j].parent != 0) { // 往上找到根结点
            parent = huffTree[j].parent; // 父结点
```

```
            if (j == huffTree[parent].leftchild) // 如果是左孩子，则记为0
                code += "0";
            else // 右孩子，记为1
                code += "1";
            j = parent; // 上移到父结点
        }
        // 编码要倒过来：因为是从叶子往上走到根，而编码是要从根走到叶子结点
        for (k = (int)code.size() - 1; k >= 0; k--)
            huffCode[i] += code[k]; // 保存编码
    }
}
//赫夫曼解码
void HuffMan::Decode(const string codestr)
{
    string txtstr;
    int c = len;
    char error_recode = '0';
    txtstr = "";
    for (int i = 0; i < codestr.length(); i++)
    {
        if (codestr[i] == '0')
        {
            c = huffTree[c].leftchild;
        }
        else if (codestr[i] == '1')
        {
            c = huffTree[c].rightchild;
        }
        else
        {
            cout << "error" << endl;
            return;
        }
        if (huffTree[c].ch == '0')
        {
            error_recode = '0';
        }
        else
        {
            txtstr += huffTree[c].ch;
            error_recode = huffTree[c].ch;
            c = len;
        }
```

```
    }
    if (error_recode == '0')
        cout << "error" << endl;
    else
        cout << txtstr << endl;
    return;
}
```

4、二叉排序树

```
#include<iostream>
using namespace std;
//二叉排序树结点类
class BSTNode
{
public:
	int key;		//结点关键字
	BSTNode* Parent;	//父结点
	BSTNode* Lchild;	//左孩子
	BSTNode* Rchild;	//右孩子
	BSTNode() :key(0), Parent(NULL), Lchild(NULL), Rchild(NULL) {}
};
//二叉排序树类
class BSTree
{
private:
	BSTNode* Root;  //根结点指针

	void InOrder(BSTNode* t);
	BSTNode* Search_BST(BSTNode* T, int key);
	BSTNode* Insert_BST(BSTNode* T, int key);
public:
	int search_times;   //每次查找次数
	BSTree();  //构造函数
	void InOrder();     //中序遍历
	BSTNode* Search_BST(int key);		//二叉排序树的递归查找
	void Insert_BST(int key);		//二叉排序树的递归插入
	void Delete_BST(int key);		//二叉排序树的递归删除
	void Create_BST(int* item, int n);	//构建二叉排序树
};
BSTree::BSTree()
{
	Root = NULL;
	search_times = 0;
}
//定义中序遍历函数
void BSTree::InOrder()  //公有函数，对外接口
{
	InOrder(Root);
}
void BSTree::InOrder(BSTNode* t)
```

```
{
	if (t != NULL)
	{
		InOrder(t->Lchild);
		cout << t->key << " ";
		InOrder(t->Rchild);
	}
	else
		return;
}
//定义二叉排序树的递归查找函数
BSTNode* BSTree::Search_BST(int key)  //公有函数，对外接口
{
	search_times = 0;   //查找次数清零
	return Search_BST(Root, key);
}
BSTNode* BSTree::Search_BST(BSTNode* T, int key)
{
	if (T == NULL)
		return NULL;  //搜索失败则返回NULL
	else
	{
		if (T->key == key)
		{
			search_times++;
			return T;          //搜索成功则返回结点指针
		}
		else if (T->key > key)
		{
			search_times++;
			Search_BST(T->Lchild, key);
		}
		else
		{
			search_times++;
			Search_BST(T->Rchild, key);
		}
	}
}
//定义二叉排序树的递归插入函数
void BSTree::Insert_BST(int key)  //公有函数，对外接口
{
	Root = Insert_BST(Root, key);
```

```
}
BSTNode* BSTree::Insert_BST(BSTNode* T, int key)
{
	if (T == NULL)
	{
		T = new BSTNode;
		T->key = key;
		return T;
	}
	else
	{
		if (T->key == key)
			return NULL;
		else if (T->key > key)
			T->Lchild = Insert_BST(T->Lchild, key);
		else
			T->Rchild = Insert_BST(T->Rchild, key);
		return T;
	}
}
//定义二叉排序树的递归删除函数
void BSTree::Delete_BST(int key)
{
	BSTNode* T = Search_BST(key);
	if (T == NULL)
	{
		return;   //若要删除的数据不在二叉树中，则不执行操作
	}
	if ((T->Lchild == NULL) && (T->Rchild == NULL))
	{//情况一:该结点是叶子结点
		if (T->Parent->Lchild == T)
			T->Parent->Lchild = NULL;
		else if (T->Parent->Rchild == T)
			T->Parent->Rchild = NULL;
	}
	else if ((T->Lchild != NULL) && (T->Rchild != NULL))
	{//情况二:该结点同时有左、右子树
		BSTNode* t = T->Rchild;
		while (t->Lchild != NULL)
			t = t->Lchild;
		T->key = t->key;

		if (t->Rchild != NULL)
```

```
                        *t = *t->Rchild;
                else
                {
                        if (t->Parent->Lchild == t)
                                t->Parent->Lchild = NULL;
                        else if (t->Parent->Rchild == t)
                                t->Parent->Rchild = NULL;
                }
        }
        else
        {//情况三:该结点只有左子树或右子树
                if (T->Lchild != NULL)
                {
                        *T = *T->Lchild;
                }
                else if (T->Rchild != NULL)
                {
                        *T = *T->Rchild;
                }
        }
}
//构建二叉排序树
void BSTree::Create_BST(int* item, int n)
{
        for (int i = 0; i < n; i++)
        {
                Insert_BST(item[i]);
        }
}
```

六、静态查找

1、顺序查找(带哨兵)

```
#include<iostream>
using namespace std;
//带哨兵的顺序查找函数
//查找成功则返回元素在数组中的位置
//查找失败则返回0
int Sequential_Search(int* item, int n, int key)
{
	int i = n;
	item[0] = key;

	while (item[i] != key)
		i--;

	return i;
}
```

2、顺序索引查找

```
#include<iostream>
using namespace std;
//顺序索引查找类
class Sequential_Index_Search
{
private:
	int n;	//主表长度
	int k;	//主表划分出的块数
	int* item;//主表
	int* maxkey_index;//最大关键字索引表
	int* StartPosition_index;//最大关键字起始位置索引表
public:
	//构造函数
	Sequential_Index_Search(int n1, int k1, int* item1, int* a)
	{
		n = n1;
		k = k1;
		item = item1;
		maxkey_index = a;
		StartPosition_index = new int[k];
		int p = 1;
		for (int i = 0; i < k; i++)
		{
			StartPosition_index[i] = p + i * (n / k);
		}
	}
	//顺序索引查找
	int Search(int key)
	{
		for (int i = 0; i < k; i++)
		{
			if (maxkey_index[i] >= key)
			{
				int spi = StartPosition_index[i];
				for (int j = spi; j < spi + (n / k); j++)
				{
					if (item[j] == key)
						return j;
				}
				return 0;
			}
```

```
                }
                return 0;
        }
        //输出带查找次数的顺序索引查找
        void Search_times(int key)
        {
                for (int i = 0; i < k; i++)
                {
                        if (maxkey_index[i] >= key)
                        {
                                int spi = StartPosition_index[i];
                                for (int j = spi; j < spi + (n / k); j++)
                                {
                                        if (item[j] == key)
                                        {
                                                //(j - spi + 1)表示在块内查找的次数
                                                //(i + 1)表示在块间查找的次数
                                                cout << j << "-" << (j - spi + 1) + (i
                                                return;
                                        }
                                }
                                cout << "error" << endl;
                                return;
                        }
                }
                cout << "error" << endl;
        }
};
```

3、折半查找

```
#include<iostream>
using namespace std;
//折半查找函数
//查找成功则返回元素在数组中的位置
//查找失败则返回0
int Binary_Search(int* item, int n, int key)
{
	int low = 1;
	int high = n;
	int mid = (n + 1) / 2;
	while (low <= high)
	{
		if (key < item[mid])
		{
			high = mid - 1;
			mid = (low + high) / 2;
		}
		else if (key > item[mid])
		{
			low = mid + 1;
			mid = (low + high) / 2;
		}
		else if (key == item[mid])
		{
			return mid;
		}
	}
	return 0;
}
```

七、哈希表

1、哈希查找-链地址法(表头插入)

```
#include<iostream>
using namespace std;
class hash_node
{
public:
	int data;   //结点数据
	int order;
	hash_node* next;
	hash_node()
	{
		data = -1;
		order = 1;
		next = NULL;
	}
};
//以求余法为哈希函数
//哈希冲突用链地址法(表头插入)
class hash_map
{
private:
	hash_node* hashmap;   //哈希结点数组
public:
	hash_map(int n)
	{
		hashmap = new hash_node[n];
		int a;
		int key;
		while (n--)
		{
			cin >> a;
			key = a % 11;
			if (hashmap[key].data != -1)
			{
				hash_insert(&hashmap[key], a);
			}
			else
				hashmap[key].data = a;
		}
	}
```

```
//递归查找哈希冲突
int find_next(hash_node* hash, int target)
{
	if (hash != NULL)
	{
		if (hash->data != target)
			return find_next(hash->next, target);
		else
			return hash->order;
	}
	else
		return 0;
}
//哈希查找函数
void hash_find(int target)
{
	int key;
	key = target % 11;
	if (hashmap[key].data != target)
	{
		if (find_next(hashmap[key].next, target) != 0)
		{
			cout << key << " " << find_next(hashmap[key].next, target)
		}
		else
		{
			//查找失败，将数据插入到哈希表中
			hash_insert(&hashmap[key], target);
			cout << "error" << endl;
		}
	}
	else
	{
		cout << key << " " << hashmap[key].order << endl;
	}
}
//哈希插入函数
//将数据target插人到哈希表中，哈希冲突用链地址法(表头插入)
void hash_insert(hash_node* hash, int target)
{
	if (hash->data == -1)
	{
		hash->data = target;
```

```
                    return;
                }
                else
                {
                    if (hash->next != NULL)
                    {
                        hash_insert(hash->next, target);
                        hash->order++;
                    }
                    else
                    {
                        hash_node* h = new hash_node();
                        h->data = target;
                        hash->order++;
                        hash->next = h;
                    }
                }
            }
};
```

2、哈希查找-链地址法(表尾插入)

```
#include<iostream>
using namespace std;
class hash_node
{
public:
	int data;   //结点数据
	int order;
	hash_node* next;
	hash_node()
	{
		data = 0;
		order = 1;
		next = NULL;
	}
};
//以求余法为哈希函数
//哈希冲突用链地址法(表尾插入)
class hash_map
{
private:
	hash_node* hashmap;   //哈希结点数组
public:
	hash_map(hash_node* item)
	{
		hashmap = item;
	}
	//递归查找哈希冲突
	int find_next(hash_node* hash, int target)
	{
		if (hash != NULL)
		{
			if (hash->data != target)
				return find_next(hash->next, target);
			else
				return hash->order;
		}
		else
			return 0;
	}
	//哈希查找函数
	void hash_find(int target)
```

```
{
        int key;
        key = target % 11;
        if (hashmap[key].data != target)
        {
                if (find_next(hashmap[key].next, target) != 0)
                {
                        cout << key << " " << find_next(hashmap[key].next, target)
                }
                else
                {
                        //查找失败，将数据插入到哈希表中
                        hash_insert(&hashmap[key], target);
                        cout << "error" << endl;
                }
        }
        else
        {
                cout << key << " " << hashmap[key].order << endl;
        }
}
//哈希插入函数
//将数据target插人到哈希表中，哈希冲突用链地址法(表尾插入)
void hash_insert(hash_node* hash, int target)
{
        if (hash->data == 0)
        {
                hash->data = target;
                return;
        }
        else
        {
                if (hash->next != NULL)
                        hash_insert(hash->next, target);
                else
                {
                        hash_node* h = new hash_node();
                        h->data = target;
                        h->order = hash->order + 1;
                        hash->next = h;
                }
        }
```

```
        }
};
```

3、哈希查找-线性探测再散列

```
#include<iostream>
using namespace std;
class hash_map
{
private:
	int* hashmap;
	int len;

	//哈希插入冲突时递归散列
	//私有函数，类内调用
	void next_insert(int& key, int p)
	{
		key++;
		if (key >= len)
			key = 0;
		if (hashmap[key] < 0)
			hashmap[key] = p;
		else
			next_insert(key, p);
	}
	//哈希查找冲突时递归散列
	//私有函数，类内调用
	int next_find(int& key, int target, int& num)
	{
		num++;
		key++;
		if (key >= len)
			key = 0;
		if (hashmap[key] < 0)
			return 0;
		else if (hashmap[key] == target)
			return key + 1;
		else
			next_find(key, target, num);
	}
public:
	hash_map(int m)
	{
		len = m;
		hashmap = new int[m];
	}
```

```
//哈希遍历展示函数
void hash_display()
{
	for (int i = 0; i < len; i++)
	{
		if (hashmap[i] >= 0)
			cout << hashmap[i] << " ";
		else
		{
			cout << "NULL ";
			//cout << hashmap[i] << " ";
		}
	}
	cout << endl;
}
//哈希插入函数
void hash_insert(int p)
{
	int key;
	key = p % 11;
	if (hashmap[key] < 0)
		hashmap[key] = p;
	else
		next_insert(key, p);
}

//哈希查找函数
void hash_find(int target)
{
	int key;
	key = target % 11;
	int num;  //查找次数
	num = 1;
	if (hashmap[key] < 0)
	{
		cout << 0 << " ";
		cout << num << endl;
	}
	else if (hashmap[key] == target)
	{
		cout << 1 << " ";
		cout << num << " ";
		cout << key + 1 << endl;
```

```
                }
                else
                {
                        int result = next_find(key, target, num);
                        if (result != 0)
                        {
                                cout << 1 << " ";
                                cout << num << " ";
                                cout << result << endl;
                        }
                        else
                        {
                                cout << 0 << " ";
                                cout << num << endl;
                        }
                }
        }
};
```

4、哈希查找-Trie树

```
#include<iostream>
#include<string>
#include<queue>
using namespace std;
//Trie树结点类
class TrieTreeNode
{
public:
	string word;  //结点字符
	int num;    // 有多少单词通过这个节点,即由根至该节点组成的字符串模式出现的次数
	bool isEnd; // 是不是最后一个节点
	TrieTreeNode** Nextword = new TrieTreeNode * [26];
	TrieTreeNode()
	{
		num = 0;
		isEnd = false;
		for (int i = 0; i < 26; i++)
		{
			Nextword[i] = NULL;
		}
	}
	//~TrieTreeNode();
};

//Trie树类
class TrieTree
{
private:
	TrieTreeNode* Root;   //根节点指针
public:
	TrieTree();
	//~TrieTree();
	void insert(string str); //在字典树中插入一个单词
	int find(string str);   //在字典树中查找以该字符串为公共前缀的单词数
	void show();            //层次遍历字典树
};
TrieTree::TrieTree()
{
	Root = new TrieTreeNode();
}
//在字典树中插入一个单词
```

```
void TrieTree::insert(string str)
{
	if (str == "" || str.length() == 0)
	{
		return;
	}
	TrieTreeNode* T = Root;
	int len = str.length();
	for (int i = 0; i < len; i++)
	{
		int pos = str[i] - 'a';
		//如果当前节点的儿子节点中没有该字符，则构建一个TrieTrieTreeNode并复值该字符
		if (T->Nextword[pos] == NULL)
		{
			T->Nextword[pos] = new TrieTreeNode();
			T->Nextword[pos]->word = str[i];
			T->Nextword[pos]->num++;
		}
		//如果已经存在，则将由根至该儿子节点组成的字符串模式出现的次数+1
		else
		{
			T->Nextword[pos]->num++;
		}
		T = T->Nextword[pos];
	}
	T->isEnd = true;
}
//在字典树中查找以该字符串为公共前缀的单词数
int TrieTree::find(string str)
{
	if (str == "" || str.length() == 0)
	{
		return 0;
	}
	TrieTreeNode* T = Root;
	int len = str.length();
	for (int i = 0; i < len; i++)
	{
		int pos = str[i] - 'a';
		//如果当前节点的儿子节点中没有该字符，则构建一个TrieTrieTreeNode并复值该字符
		if (T->Nextword[pos] != NULL)
		{
			T = T->Nextword[pos];
```

```
                }
                //如果已经存在，则将由根至该儿子节点组成的字符串模式出现的次数+1
                else
                {
                        return 0;
                }
        }
        return T->num;
}
//层次遍历字典树
void TrieTree::show()
{
        if (!Root) return;
        deque<TrieTreeNode*> q;
        q.push_back(Root);
        while (!q.empty())
        {
                int qs = q.size();
                for (int i = 0; i < qs; i++)
                {
                        TrieTreeNode* tp = q.front(); q.pop_front();
                        cout << tp->word;
                        for (int i = 0; i < 26; i++)
                                if (tp->Nextword[i]) q.push_back(tp->Nextword[i]);
                }
        }
        cout << endl;
}
```

5、哈希查找-二次线性探测再散列

```
#include<iostream>
using namespace std;
class hash_map
{
private:
	int* hashmap;
	int len;
	int pos;

	//哈希插入冲突时递归散列
	//私有函数，类内调用
	void next_insert(int& key, int p)
	{
		int original_key = key;
		key += pos * pos;
		if (key >= len)
			key = key - len;
		if (hashmap[key] < 0)
			hashmap[key] = p;
		else
		{
			key = key - pos * pos * 2;
			if (key < 0)
				key = key + len;
			if (hashmap[key] < 0)
				hashmap[key] = p;
			else
			{
				pos++;
				next_insert(original_key, p);
			}
		}
	}
	//哈希查找冲突时递归散列
	//私有函数，类内调用
	int next_find(int& key, int target, int& num)
	{
		num++;
		int original_key = key;
		key += pos * pos;
		if (key >= len)
```

```
                key = key - len;
            if (hashmap[key] == target)
                return key + 1;
            else if (hashmap[key] < 0)
                return 0;
            else
            {
                num++;
                key = key - pos * pos * 2;
                if (key < 0)
                    key = key + len;
                if (hashmap[key] == target)
                    return key + 1;
                else if (hashmap[key] < 0)
                    return 0;
                else
                {
                    pos++;
                    next_find(original_key, target, num);
                }
            }
        }
public:
        hash_map(int m)
        {
            len = m;
            hashmap = new int[m];
            pos = 1;
        }
        //哈希遍历展示函数
        void hash_display()
        {
            for (int i = 0; i < len; i++)
            {
                if (hashmap[i] >= 0)
                    cout << hashmap[i] << " ";
                else
                {
                    cout << "NULL ";
                    //cout << hashmap[i] << " ";
                }
            }
            cout << endl;
```

```
}
//哈希插入函数
void hash_insert(int p)
{
	int key;
	key = p % 11;
	if (hashmap[key] < 0)
		hashmap[key] = p;
	else
	{
		pos = 1;
		next_insert(key, p);
	}
}

//哈希查找函数
void hash_find(int target)
{
	int key;
	key = target % 11;
	int num;  //查找次数
	num = 1;
	if (hashmap[key] < 0)
	{
		cout << 0 << " ";
		cout << num << endl;
	}
	else if (hashmap[key] == target)
	{
		cout << 1 << " ";
		cout << num << " ";
		cout << key + 1 << endl;
	}
	else
	{
		pos = 1;
		int result = next_find(key, target, num);
		if (result != 0)
		{
			cout << 1 << " ";
			cout << num << " ";
			cout << result << endl;
		}
```

```
                        else
                        {
                                cout << 0 << " ";
                                cout << num << endl;
                        }
                }
        }
};
```

八、排序

1、插入排序

```
#include<iostream>
using namespace std;
//直接插入排序
//稳定排序
void StraightInsertSort(int* item, int n)
{
        for (int i = 1; i < n; i++)
        {
                for (int k = i; k > 0; k--)
                {
                        if (item[k] <= item[k - 1])
                        {
                                int m = item[k];
                                item[k] = item[k - 1];
                                item[k - 1] = m;
                        }
                        else
                                break;
                }

                //每趟排序后输出一次结果
                for (int i = 0; i < n; i++)
                {
                        cout << item[i] << " ";
                }
                cout << endl;
        }
}
```

2、折半插入排序

```
#include<iostream>
using namespace std;
//折半插入排序
//稳定排序
void BinaryInsertSort(int* item, int n)
{
	int temp;
	for (int i = 1; i < n; i++)
	{
		temp = item[i];
		int low = 0;
		int high = i - 1;
		int mid;
		while (low <= high)
		{
			mid = (low + high) / 2;
			if (temp > item[mid])
				high = mid - 1;
			else
				low = mid + 1;
		}
		int j;
		for (j = i - 1; j >= high + 1; j--)
			item[j + 1] = item[j];
		item[j + 1] = temp;

		for (int k = 0; k < n; k++)
		{
			cout << item[k] << " ";
		}
		cout << endl;
	}
}
```

3、希尔排序

```
#include<iostream>
using namespace std;
//希尔排序
//不稳定排序
void ShellSort(int* item, int n)
{
	int gap = n / 2;
	while (gap >= 1)
	{
		int t = gap;
		int pos = 0;
		while (t--)
		{
			for (int i = pos; i < n; i += gap)
			{
				for (int k = pos; k < n - gap; k += gap)
				{
					if (item[k] <= item[k + gap])
					{
						int m = item[k];
						item[k] = item[k + gap];
						item[k + gap] = m;
					}
				}
			}
			pos++;
		}

		//每趟排序完输出一次结果
		for (int i = 0; i < n; i++)
		{
			cout << item[i] << " ";
		}
		cout << endl;

		gap = gap / 2;
	}
}
```

4、快速排序

```
#include<iostream>
using namespace std;
int getPivotkey(int* item, int low, int high)
{
	int key = item[low];
	while (low < high)
	{
		while (low < high && item[high] >= key)
		{
			high--;
		}
		if (low < high)
		{
			item[low] = item[high];
		}

		while (low < high && item[low] <= key)
		{
			low++;
		}
		if (low < high)
		{
			item[high] = item[low];
		}
	}
	item[low] = key;
	return low;
}
//快速排序
//不稳定排序
void QuickSort(int* item, int low, int high)
{
	if (low <= high)
	{
		int pivotkey = getPivotkey(item, low, high);
		cout << item[pivotkey] << " " << pivotkey + 1 << endl;
		QuickSort(item, low, pivotkey - 1);
		QuickSort(item, pivotkey + 1, high);
	}
}
```

5、选择排序

```
#include<iostream>
using namespace std;
//选择排序
//不稳定排序
void SelectSort(int* item, int n)
{
	int min;
	for (int i = 0; i < n; i++)
	{
		min = i;
		for (int k = i + 1; k < n; k++)
		{
			if (item[k] < item[min])
			{
				min = k;
			}
		}

		int m = item[i];
		item[i] = item[min];
		item[min] = m;

		for (int i = 0; i < n; i++)
		{
			cout << item[i] << " ";
		}
		cout << endl;
	}
}
```

6、堆排序

```
#include<iostream>
using namespace std;
//堆排序
//不稳定排序
class HeapSort
{
private:
	int* item;  //数组指针
	int len;    //数组长度
public:
	HeapSort(int n)
	{
		len = n;
		item = new int[len];
		for (int i = 0; i < len; i++)
		{
			cin >> item[i];
		}
	}
	void sort()
	{
		//构建小堆顶
		for (int i = len / 2 - 1; i >= 0; i--)
		{
			//从最后一个非叶子结点开始，从下至上，从左至右调整结构
			adjustHeap(i, len);
		}
		display();
		//将堆顶元素和末尾元素进行交换，然后再重新对堆进行调整
		for (int j = len - 1; j > 0; j--)
		{
			swap(0, j);      //将堆顶元素和末尾元素进行交换
			adjustHeap(0, j);  //重新对堆进行调整
			display();
		}
	}
	void adjustHeap(int i, int length)
	{
		int temp = item[i];  //先取出当前元素
		for (int k = i * 2 + 1; k < length; k = k * 2 + 1)
		{
```

```
                        if (k + 1 < length && item[k] > item[k + 1])//从i结点的左子结点开始，
                        {
                                //如果右结点存在，且左结点大于右结点，那么k指向右结点
                                k++;
                        }
                        if (item[k] < temp)
                        {
                                //如果子节点小于父节点，将子节点值赋给父节点
                                item[i] = item[k];
                                i = k;
                        }
                }
                item[i] = temp;
        }
        void swap(int a, int b)
        {
                int m = item[a];
                item[a] = item[b];
                item[b] = m;
        }
        void display()
        {
                cout << len << " ";
                for (int i = 0; i < len; i++)
                {
                        cout << item[i] << " ";
                }
                cout << endl;
        }
};
```

7、2路归并排序

```
#include<iostream>
#include<cmath>
using namespace std;
//2路归并排序，递归做法
//稳定排序
class MergeSort
{
private:
	int* item;  //需要排序的数组
	int* temp;  //临时数组
	int* item1;  //合并函数所使用的临时数组
	int len;  //需要排序的数组长度
	int k;
	int p;  //2路归并排序次数
	//2路归并排序函数
	void sort(int left, int right)
	{
		if (left < right)
		{
			int mid = (left + right) / 2;
			sort(left, mid);//左边归并排序，使得左子序列有序
			sort(mid + 1, right);//右边归并排序，使得右子序列有序
			merge(left, mid, right);
		}
		p++;
	}
	//归并函数
	void merge(int left, int mid, int right)
	{
		int i = left;//左序列指针
		int j = mid + 1;//右序列指针
		int t = 0;//临时数组指针
		while (i <= mid && j <= right)
		{
			if (item[i] < item[j])
			{
				temp[t] = item[i];
				t++;
				i++;
			}
			else
```

```
                {
                        temp[t] = item[j];
                        t++;
                        j++;
                }
        }
        while (i <= mid)
        {
                //将左子序列剩余元素填入temp中
                temp[t] = item[i];
                t++;
                i++;
        }
        while (j <= right)
        {
                //将右子序列剩余元素填入temp中
                temp[t] = item[j];
                t++;
                j++;
        }
        //将temp中的元素全部拷贝到原数组中
        t = 0;
        while (left <= right)
        {
                item[left] = temp[t];
                t++;
                left++;
        }
}
//判断a是否为整数
bool IsInteger(double a)
{
        if (a - (int)a == 0)
                return true;
        else
                return false;
}
//选择排序函数
void SelectSort(int* arr, int n)
{
        int min;
        for (int i = 0; i < n; i++)
        {
```

```
                        min = i;
                        for (int k = i + 1; k < n; k++)
                        {
                                if (arr[k] < arr[min])
                                {
                                        min = k;
                                }
                        }

                        int m = arr[i];
                        arr[i] = arr[min];
                        arr[min] = m;
                }
        }
public:
        //合并函数
        MergeSort(int* str, int n)
        {
                p = 0;
                item = str;
                len = n;
                item1 = new int[len];
                for (int i = 0; i < len; i++)
                {
                        item1[i] = str[i];
                }
                temp = new int[len];
                if (IsInteger(log2(len)))
                        k = (int)log2(len) + 1;
                else
                        k = (int)log2(len) + 2;
                sort(0, len - 1);
                cout << p << endl;
        }
        void display()
        {
                for (int i = 0; i < len; i++)
                {
                        cout << item[i] << " ";
                }
                cout << endl;
        }
};
```

8、基数排序

```
#include<iostream>
using namespace std;
//基数排序
//稳定排序
class RadixSort
{
private:
	int* data;  //需要排序的数组
	int len;  //需要排序的数组长度
	//求数据的最大位数
	int maxbit()
	{
		int maxData = data[0];
		for (int i = 1; i < len; i++)
		{
			if (maxData < data[i])
				maxData = data[i];
		}
		int d = 1;
		while (maxData >= 10)
		{
			maxData /= 10;
			d++;
		}
		return d;
	}
	void Sort()
	{
		int d = maxbit();
		int* temp = new int[len];
		int* count = new int[10];//计数器
		int** arr = new int* [10];
		int radix = 1;
		int j, k, h;
		for (int i = 1; i <= d; i++)
		{
			for (j = 0; j < 10; j++)
				count[j] = 0;//每次分配前清空计数器

			for (j = 0; j < 10; j++)
			{
```

```
                    arr[j] = NULL;
                }

                for (j = 0; j < len; j++)
                {
                    k = (data[j] / radix) % 10;    //统计每个桶中的记录数
                    if (arr[k] == NULL)
                        arr[k] = new int[len];
                    arr[k][count[k]] = data[j];
                    count[k]++;
                }

                for (j = 0; j < 10; j++)
                {
                    cout << j << ":";
                    if (arr[j] == NULL)
                        cout << "NULL" << endl;
                    else
                    {
                        cout << "->";
                        for (h = 0; h < count[j]; h++)
                            cout << arr[j][h] << "->";
                        cout << "^" << endl;
                    }
                }

                for (j = 1; j < 10; j++)
                    count[j] = count[j - 1] + count[j];
                for (j = len - 1; j >= 0; j--)
                {
                    k = (data[j] / radix) % 10;
                    temp[count[k] - 1] = data[j];
                    count[k]--;
                }
                for (j = 0; j < len; j++)
                    data[j] = temp[j];
                radix *= 10;
                display();
            }
            delete[]temp;
            delete[]count;
        }
public:
```

```
        //构造函数
        RadixSort(int* item, int n)
        {
                data = item;
                len = n;
                Sort();
        }
        void display()
        {
                for (int i = 0; i < len; i++)
                {
                        cout << data[i] << " ";
                }
                cout << endl;
        }
};
```

文章知识点与官方知识档案匹配，可进一步学习相关知识

算法技能树首页概览56906 人正在系统学习中