算法题

## 向量的实现

public class SeqList<T>: IList<T>

{

private int maxsize;

private T[] data;

private int last;

public T this[int index]

{

get{ return data[index]; }

set{ data[index] = value; }

}

public int Last

{

get{ return last; }

}

public int Maxsize

{

get{ return maxsize; }

set{ maxsize = value; }

}

public SeqList(int size)

{

data = new T[size];

maxsize = size;

last = -1;

}

public int GetLength()

{

return last+1;

}

public bool IsEmpty()

{

if(last==-1){ return true; }

else{ return false; }

}

}

## 栈实现

public class SeqStack<T>: IStack<T>

{

public int Maxsize{ get; set; }

private T[] data;

private int top;

public T this[int index]

{

get{ return data[index]; }

set{ data[index] = value; }

}

public int Top{ get{ return top; } }

public SeqStack(int size)

{

data = new T[size];

Maxsize = size;

top = -1;

}

public bool IsFull()

{

if(top == Maxsize - 1){ return true; }

return false;

}

public bool IsEmpty()

{

if(top == -1){ return true; }

return false;

}

public bool Push(T item)

{

if(IsFull()) { return false; }

data[++top] = item;

return true;

}

public T Pop()

{

if(IsEmpty()){ return null; }

T temp = data[top];

--top;

return temp;

}

}

## 队列实现

public class SeqQueue<T>: IQueue<T>

{

Public int Maxsize{ get; set; }

Private T[] data;

Public int Front{ get; set; }

Public int Rear{ get; set; }

Public T this[int index]{

get{ return data[index]; }

set{ data[index] = value; }

}

Public SeqQueue(int size){

data = new T[size];

Maxsize = size;

Front = Rear = -1;

}

Public int GetLength(){

return (Rear - Front + Maxsize) % Maxsize;

}

Public bool IsFull(){

if((Rear+1) % Maxsize == Front){ return true; }

return false;

}

Public bool IsEmpty(){

if(Front == Rear) { return true; }

return false;

}

Public bool In(T item){

if(IsFull()) { return false; }

data[++Rear] = item;

return true;

}

Public T Out(){

if(IsEmpty()){ return null; }

T temp = data[++front];

return temp;

}

}

## 链表的实现

public class Node<T>

{

public T Data {get; set;}

public Node<T> Next {get; set;}

public Node(T val, Node<T> p)

{

Data = val;

Next = p;

}

}

树

## 二叉树的实现

public class TreeNode<T>

{

public T Data{ get; set; }

public TreeNode<T> LChild{ get; set; }

public TreeNode<T> RChild{ get; set; }

public TreeNode(T val, TreeNode<T> lp, TreeNode<T> rp)

{

Data = val;

lp = LChild;

rp = RChild;

}

}

## 遍历二叉树（前中后）

public void PreOrder(TreeNode<T> root){

if(root == null){

return;

}

visit(root);

PreOrder(root.LChild);

PreOrder(root.RChild);

}

public void InOrder(TreeNode<T> root){

if(root == null){ return; }

MidOrder(root.LChild);

visit(root);

MidOrder(root.RChild);

}

public void PostOrder(TreeNode<T> root){

if(root == null){ return; }

PostOrder(root.LChild);

PostOrder(root.RChild);

visit(root);

}

## 求二叉树的深度

public int GetTreeDeep(BinTree BT){

int deep=0;

if(BT){

int lchilddeep=GetTreeDeep(BT->lchild);

int rchilddeep=GetTreeDeep(BT->rchild);

deep=lchilddeep>=rchilddeep?lchilddeep+1:rchilddeep+1;

}

return deep;

}

## 按层次遍历二叉树

public void TraverseByLayers(TreeNode<T> root){

if(root == null) { return; }

Queue< TreeNode<T>> queue = new Queue< TreeNode<T>>();

queue.In(root);

while(!queue.IsEmpty()) {

TreeNode<T> tNode = queue.Out();

visit(tNode);

if(tNode.LChild != null) {

queue.In(tNode.LChild);

}

if(tNode.RChild != null) {

queue.In(tNode.RChild);

}

}

}

## 求二叉树的宽度

public int GetTreeWidth(TreeNode<T> root){

if(root == null){ return; }

Queue< TreeNode<T>> queue = new Queue< TreeNode<T>>();

queue.In(root);

int width = 1;

int lastLevelWidth = 1;

while(!queue.IsEmpty()){

while(lastLevelWidth > 0){

TreeNode<T> tNode = queue.Out();

if(tNode.LChild != null){

queue.In(tNode.LChild);

}

if(tNode.RChild != null){

queue.In(tNode.RChild);

}

lastLevelWidth --;

}

lastLevelWidth = queue.length;

width = width > lastLevelWidth? lastLevelWidth: width;

}

return width;

}

## 非递归遍历二叉树

public void PreOrder(TreeNode<T> root){

if(root == null) { return; }

Stack<TreeNode<T>> stack = new Stack<TreeNode<T>>();

stack.Push(root);

if(!stack.IsEmpty()) {

TreeNode<T> tNode = stack.Pop();

visit(tNode);

if(tNode.LChild != null) {

stack.Push(tNode.RChild);

}

if(tNode.RChild != null) {

stack.Push(tNode.LChild);

}

}

}

public void InOrder(TreeNode<T> root)

{

Stack<TreeNode<T>> s = new Stack<TreeNode<T>>()

TreeNode<T> tNode = root;

while(tNode || !s.IsEmpty())

{

while(tNode)

{

s.Push(tNode);

tNode = tNode.LChild;

}

else

{

tNode = s.Pop();

visit(tNode);

tNode = tNode.RChild;

}

}

}

public void PostOrder(TreeNode<T> root){ //找出某点的所有祖先节点

Stack<TreeNode<T>> stack = new Stack<TreeNode<T>>();

Stack<int> flagStack = new Stack<int>();

TreeNode tNode = root;

int flag = 0;

while(tNode != null || !stack.IsEmpty()){

if(tNode != null){

stack.Push(tNode);

flagStack.Push(1); //第一次入栈

tNode = tNode.LChild;

}else{

tNode = stack.Pop();

flag = flagStack.Pop();

if(flag == 1){

stack.Push(tNode);

flagStack.Push(2); //第二次入栈

tNode = tNode.RChild;

}else {

visit(tNode);

tNode = null;

}

}

}

}

## 线索二叉树的实现

public class ThreadedTreeNode<T>

{

public T Data{ get; set; }

public int LTag{ get; set; }

public int RTag{ get; set; }

public ThreadedTreeNode<T> LChild{ get; set; }

public ThreadedTreeNode<T> RChild{ get; set; }

public ThreadedTreeNode(T val, ThreadedTreeNode<T> lp, ThreadedTreeNode<T> rp)

{

Data = val;

lp = LChild;

rp = RChild;

}

}

## 哈弗曼树的实现

public class Node

{

public int Weight{ get; set; }

public int LChild{ get; set; }

public int RChild{ get; set; }

public int Parent{ get; set; }

public Node(int w, int lc, int rc, int p)

{

Weight = w;

LChild = lc;

RChild = rc;

Parent = p;

}

}

public class Huffman Tree

{

private Node[] data;

Public int LeafNum;

public Node this[int index]

{

get{ return data[index]; }

set{ data[index] = value; }

}

public Huffman Tree(List<int> weightList)

{

data = new Node[2\*weightList.Count - 1];

LeafNum = weightList.Count;

for(int i = 0; i < LeafNum; ++i)

{

data[i].Weight = weightList[i];

}

}

public void Create()

{

for(int i = 0; i < LeafNum - 1; ++i) //处理n个叶子节点，建立哈弗曼树

{

min1 = min2 = Int32.MaxValue;

temp 1 = temp 2 = 0;

for(int j = 0; j < LeafNum + i; ++j) //在全部节点中找权值最小的两个点

{

if((data[i].Weight < min1) && (data[i].Parent == -1)

{ //找出第一个最小点

min2 = min1;

temp2 = temp1;

temp1 = j;

min1 = data[j].Weight;

}

else if((data[i].Weight < min2) && (data[i].Parent == -1))

{ //找出第二个最小点

min2 = data[j].Weight;

temp2 = j;

}

}

data[temp1].Parent = data[temp2].Parent = LeafNum + i;

data[LeafNum + i].Weight = data[temp1].Weight + data[temp2].Weight;

data[LeafNum + i].LChild = temp1;

data[LeafNum + i].RChild = temp2;

}

}

}

图

## 用邻接矩阵表示图、用邻接表表示图

public class Vertex<T>{

public T Data{ get; set; };

public bool IsVisited;

}

public class Graph{

private const int Number = 10; //图中所能包含的点上限

private Vertex[] vertiexes; //顶点数组

public int[,] adjmatrix; //邻接矩阵

int numVerts = 0; //统计当前图中有几个点

//初始化图

public Graph(){

//初始化邻接矩阵和顶点数组

adjmatrix = new Int32[Number, Number];

vertiexes = new Vertex[Number];

//将代表邻接矩阵的表全初始化为0

for (int i = 0; i < Number; i++){

for (int j = 0; j < Number; j++){

adjmatrix[i, j] = 0;

}

}

}

//向图中添加节点

public void AddVertex(String v){

vertiexes[numVerts] = new Vertex(v);

numVerts++;

}

//向图中添加有向边

public void AddEdge(int vertex1, int vertex2) {

adjmatrix[vertex1, vertex2] = 1;

//adjmatrix[vertex2, vertex1] = 1;

}

//显示点

public void DisplayVert(int vertexPosition) {

Console.WriteLine(vertiexes[vertexPosition]+" ");

}

}

//从邻接矩阵查找给定点第一个相邻且未被访问过的点

//参数v是给定点在邻接矩阵的行

private int GetAdjUnvisitedVertex(int v){

for (int j = 0; j < numVerts; j++){

if (adjmatrix[v,j]==1 && vertiexes[j].IsVisited == false){

return j;

}

}

return -1;

}

## 深度优先遍历

public void DepthFirstSearch(){

//声明一个存储临时结果的栈

Stack s = new Stack();

//先访问第一个节点

vertiexes[0].IsVisited = true;

DisplayVert(0);

s.Push(0);

int v;

while (!s.IsEmpty()) {

//获得和当前节点连接的未访问过节点的序号

v = GetAdjUnvisitedVertex((int)s.Peek());

if (v == -1){

s.Pop();

}else {

//标记为已经被访问过

vertiexes[v].IsVisited = true;

DisplayVert(v);

s.Push(v);

}

}

//重置所有节点为未访问过

for (int u = 0; u < numVerts; u++){

vertiexes[u].IsVisited = false;

}

}

## 广度优先遍历

//广度优先遍历

public void BreadthFirstSearch()

{

Queue q = new Queue();

/\*首先访问第一个节点\*/

vertiexes[0].IsVisited = true;

DisplayVert(0);

q.Enqueue(0);

/\*第一个节点访问结束\*/

int vert1, vert2;

while (!q.IsEmpty())

{

/\*首先访问同层级第一个节点\*/

vert1 = (int)q.Dequeue();

vert2 = GetAdjUnvisitedVertex(vert1);

/\*结束\*/

while (vert2 != -1)

{

/\*首先访问第二个节点\*/

vertiexes[vert2].IsVisited = true;

DisplayVert(vert2);

q.Enqueue(vert2);

//寻找邻接的

vert2 = GetAdjUnvisitedVertex(vert1);

}

}

//重置所有节点为未访问过

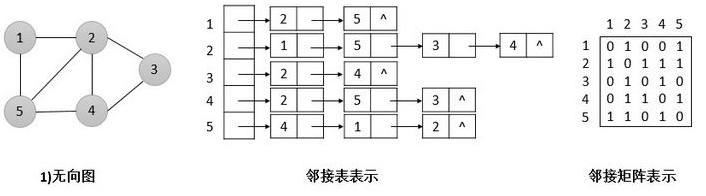
for (int u = 0; u < numVerts; u++)

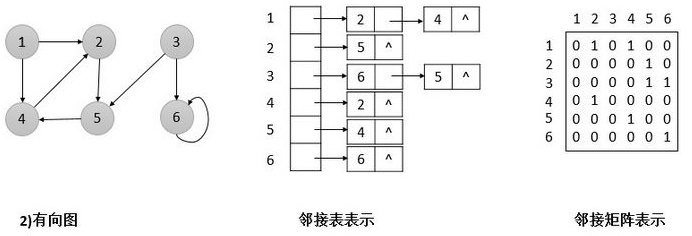
{

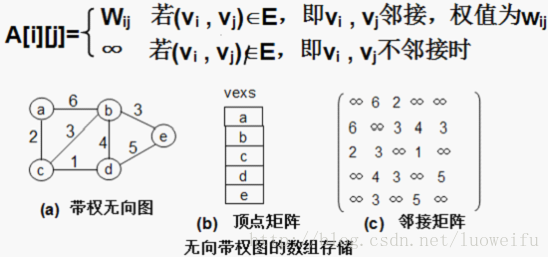
vertiexes[u].IsVisited = false;

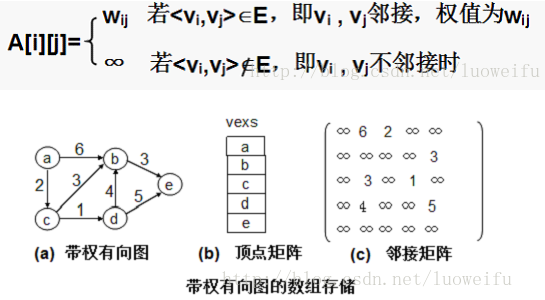
}

}









## 最小生成树（克鲁斯卡尔Kruskal算法，普里姆Prim算法）

###### Kruskal算法

克鲁斯卡尔算法的基本思想是以边为主导地位，**始终选择当前可用（所选的边不能构成回路）的最小权植边**。所以Kruskal算法的第一步是给所有的边按照从小到大的顺序排序。这一步可以直接使用库函数qsort或者sort。接下来从小到大依次考察每一条边（u，v）。

###### Prim算法

对于图G而言，V是所有顶点的集合；现在，设置两个新的集合U和T，其中U用于存放G的最小生成树中的顶点，T存放G的最小生成树中的边。

**从所有uЄU，vЄ(V-U) (V-U表示出去U的所有顶点)的边中选取权值最小的边(u, v)，将顶点v加入集合U中**，**将边(u, v)加入集合T中**。如此不断重复，**直到U=V为止**，最小生成树构造完毕，这时集合T中包含了最小生成树中的所有边。

## 拓扑排序

每次删除入度为0的顶点并输出

## 最短路径（迪杰斯特拉算法，弗洛伊德算法）

###### Dijkstra算法

Dijkstra(迪杰斯特拉)算法是典型的**单源最短路径**算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。注意该算法要求图中不存在负权边。

算法步骤：

a.初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则<u,v>正常有权值，若u不是v的出边邻接点，则<u,v>权值为∞。

b.从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。

c.以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。

d.重复步骤b和c直到所有顶点都包含在S中。

###### Floyd算法

Floyd-Warshall算法（Floyd-Warshall algorithm）是解决**任意两点间的最短路径**的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd-Warshall算法的时间复杂度为O(N3)，空间复杂度为O(N2)。

算法步骤：

a.从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。

b.对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比己知的路径更短。如果是更新它。

map[i, j] = min { map[i, k] + map[k, j], map[i, j] }；

查找

## 顺序查找

int Search(List<int> numList, int number)

{

foreach(int i in numList)

{

if(numList[i] == number)

{

return i;

}

}

return -1;

}

时间复杂度：O(n) 平均查找长度（ASL）：(n+1)/2

## 分块查找

将线性表分为若干块，用另一个索引表记录对应块的最大关键字，第一个元素位置，最后一个元素位置，关键字大小必须有序。

时间复杂度：O(log2n) 平均查找长度（ASL）：两次查找的平均查找长度之和

## 折半查找

int BinarySearch(List<int> numList, int number){

int low = 0, high = numList.Count - 1;

while(low <= high){

int mid = (low + high)/2;

if(numList[mid] == number){

return mid;

}

else if(number > numList[mid]){

low = mid + 1;

}else {

high = mid - 1;

}

}

return -1;

}

时间复杂度：O(log2n) 平均查找长度（ASL）：log2(n+1)-1

## 二叉排序树

左子树不空，则左子树上所有关键字的值均小于根关键字的值。

右子树不空，则右子树上所有关键字的值均大于根关键字的值。

Node<int> BstSearch(Node<int> root, int number){

if(root != null) {

if(root.Data == number) {

return root;

}

else if(number > root.Data){

BstSearch(root.RChild, number);

}else {

BstSearch(root.LChild, number);

}

}

return null;

}

## 平衡二叉树（查找，插入算法，平衡因子）

左右子树高度差不超过1。

分为RR，LL，RL，LR四种类型。

RR向左旋转，LL向右旋转。

RL，LR需要先分别转成RR，LL型，再对RR，LL做第二次旋转。

## B-树和B+树

B-树是二叉树的扩展，即多路查找的二叉树，插入操作只会使得B-树逐渐变高而不会改变叶子节点在同一层的特性。B+树是B-树的变形，即所有元素都存在叶子节点。

排序

## 直接插入排序

void InsertSort(List<int> numList)

{

int length = numList.Count;

for(int i = 1; i < length; i++)

{

if(numList[i] < numList[i-1])

{

int temp = numList[i];

int j = 0;

for(j = i-1; j >= 0 && temp < numList[j]; j--)

{

numList[j+1] = numList[j]; //如果比较数小于比较对象，对象后移 }

numList[j+1] = temp;

}

}

}

时间复杂度：O(n^2) 空间复杂度: O(1)

稳定性：是 特点：元素少或者基本有序时效率高

## 折半插入排序

void BinInsertSort(List<int> numList){

int length= numList.Count;

for(int i = 1; i < length; i++){

int low = 0;

int high = i-1;

while(low <= high){

int mid = (low + high)/2;

if(a[i] < a[mid]){

high = mid - 1;

}else {

low = mid + 1;

}

}

int temp = a[i];

for(int j = i-1; j>high; j--){

a[j+1] = a[j];

}

a[high+1] = temp;

}

}

时间复杂度：O(n^2) 空间复杂度: O(1)

稳定性：是 特点：元素少或者基本有序时效率高

## 基数排序

多关键字排序法，即以最低位（数字）为关键字，将相同数字的记录放到对应的容器里（队列或者链表），按容器的顺序收集记录，即完成了记录的最低位排序。再对每个记录的次低位进行相同的操作和收集，即完成了记录次低位的排序，同时他们的最低位也是有序的。如此反复，直到完成最高位的排序和收集，即完成整个序列排序。

时间复杂度：O(d(n+rd)) 空间复杂度: O(rd)

稳定性：是 特点：适合个数多关键字较小

（n为序列中元素的个数；d为元素的关键字位数，如980是三位；rd为关键字的取值范围，如980的每一位数字取值范围都是0-9，即rd=10）

## 希尔排序

void ShellSort(List<int> numList)

{

int length = numList.Count;

int gap = length/2;

int temp = 0;

while(gap >= 1)

{

for(int i = gap; i < length; i++)

{

for(int j = i-gap; j > 0 && numList[j] > numList[j+gap]; j = j-gap)

{

temp = numList[j];

numList[j] = numList[j+gap];

numList[j+gap] = temp;

}

}

}

}

时间复杂度：O(n^(3/2)) 空间复杂度: O(1)

稳定性：否

## 起泡排序

void BubbleSort(List<int> numList){

int length = numList.Count;

int temp;

while(length > 1){

for(int i = 0; i < length-1; i++) {

if(numList[i] > numList[i+1]){

temp = numList[i];

numList[i] = numList[i+1];

numList[i+1] = temp;

}

}

length--;

}

}

时间复杂度：O(n^2) 空间复杂度: O(1)

稳定性：是

## 快速排序

void QuickSort(List<int> numList, int low, int high){

if(low < high){

int pivot = numList[low];

int i = low;

int j = high;

while(i < j){

while(i < j && numList[j]>=pivot){

j--;

}

numList[i] = numList[j];

while(i < j && numList[i]<=pivot){

i++;

}

numList[j] = numList[i];

}

numList[i] = pivot;

QuickSort(numList, low, i-1);

QuickSort(numList, j+1, high);

}

}

时间复杂度：O(nlog2n) 空间复杂度: O(log2n)

稳定性：否 特点：基本有序时效率低，平均时间性能最好

## 简单选择排序

void SelectionSort(List<int> numList){

int length = numList.Count;

for(int i = 0; i < length; i++){

int k = i;

for(int j = i; j < length; j++){

if(numList[j] < numList[k]){

k = j;

}

}

numList[i] <--> numList[k];

}

}

时间复杂度：O(n^2) 空间复杂度: O(1)

稳定性：否 特点：比较次数最多

## 堆排序

void Sift(List<int> numList, int start, int end)

{

int i = start, j = 2\*i; //i为父节点位置，j为左孩子节点

int temp = numList[i]; //保存父节点的值

while(j <= end)

{

if(j < end && numList[j] < numList[j+1])

{

j++; //如果左孩子比右孩子小，j指向右孩子

}

if(numList[j] > temp)

{

numList[i] = numList[j]; //如果孩子比父节点大，将孩子节点的值放入父节点

i = j; //原来的父节点指针指向孩子节点的位置

j = i\*2; //孩子节点指针指向下一个左孩子节点的位置

}

else

break;

}

numList[i] = temp;

}

void HeapSort(List<int> numList)

{

int length = numList.Count;

int i;

int temp;

for(i = length/2; i>=1; --i)

{

Sift(numList, i, length); //建立初堆

}

for(i = n; i>=2; --i)

{

temp = numList[1]; //将第一个放入最后

numList[1] = numList[i];

numList[i] = temp;

Sift(numList, 1, i-1); //再次对调整后的堆进行排序

}

}

时间复杂度：O(nlog2n) 空间复杂度: O(1)

稳定性：否 特点：辅助空间少

## 归并排序

void Merge(List<int> numList, int low, int mid, int high)

{

int i = low, j = mid+1;

List<int> tempList = new List<int>();

while(i <= mid && j <= high)

{

if(numList[i] <= numList[j])

{

tempList.Add(numList[i]);

i++;

}

else

{

tempList.Add(numList[j]);

j++;

}

}

while(i <= mid)

{

tempList.Add(numList[i]);

}

while(j <= high)

{

tempList.Add(numList[j]);

}

numList.SetRange(tempList, low, high);

}

void MergeSort(List<int>numList, int low, int high)

{

if(low >= high){ return; }

else

{

int mid = (low + high)/2;

MergeSort(numList, low, mid);

MergeSort(numList, mid+1, high);

Merge(numList, low, mid, high);

}

}

时间复杂度：O(nlog2n) 空间复杂度: O(n)

稳定性：是 特点：各方面最稳定