**进制转换**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_STACKSIZE 10000

int Stack[MAX\_STACKSIZE];

int top;

int n;

void Create(){

top = -1;

}

int IsFull(){

if (top >= MAX\_STACKSIZE - 1) { return 1; }

else { return 0; }

}

void Push(int x){

if (IsFull()) { return; }

else{Stack[++top] = x;}

}

int IsEmpty(){

if (top < 0) { return 1; }

else { return 0; }

}

void Pop(){

if (IsEmpty()) { return; }

else{top--;}

}

int main()

{

FILE \*pf, \*op;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

while (fscanf(pf, "%d", &n), n != -1) {

Create();

int num = n;

while (num){

int tmp = num % 2;

Push(tmp);

num /=2;

}

fprintf(op, "%d--->", n);

while (!IsEmpty()){

fprintf(op,"%d", Stack[top]);

Pop();

}

fprintf(op,"\n");

}

}

**数据归并**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int a[10000], b[10000], c[20000];

int main() {

int m1, m2;

int i, j, k;

int sum = 0;//合并数组数量

int number;//数组元素数量和

while (scanf("%d", &m1) != EOF, m1 != -1) {

//输入两个数组

for (i = 0; i<m1; i++) {

scanf("%d", &a[i]);

}

scanf("%d", &m2);

for (i = 0; i<m2; i++) {

scanf("%d", &b[i]);

}

number = m1 + m2;

sum++;

i = j = k = 0;

while (i<m1&&j<m2) {

if (a[i]<b[j]) {

c[k++] = a[i++];

}

else if (a[i] == b[j]) {

c[k++] = a[i++];

c[k++] = b[j++];

}

else {

c[k++] = b[j++];

}

}

while (i<m1) {

c[k++] = a[i++];

}

while (j<m2) {

c[k++] = b[j++];

}

//输出

printf("Case %d:%d\n", sum, number);

for (i = 0; i<number; i++) {

printf("%d ", c[i]);

}

printf("\n");

}

}

**快排**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_SIZE 10000

#define SWAP(i,j,tmp) temp=i,i=j,j=temp

typedef struct {

int key;

}element;

element a[MAX\_SIZE];

void quickSort(element list[],int left,int right) {

int pivot, i, j;

element temp;

if (left < right) {

i = left;

j = right + 1;

pivot = list[left].key;

do {

do { i++; } while (list[i].key < pivot);

do { j--; } while (list[j].key > pivot);

if (i < j) { SWAP(list[i], list[j], temp); }

} while (i < j);

SWAP(list[left], list[j], temp);

quickSort(list, left, j - 1);

quickSort(list, j + 1, right);

}

}

int main(){

int n;

int caseID=0;

FILE \*pf, \*op;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

while (fscanf(pf, "%d", &n), n != -1) {

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

fscanf(pf, "%d", &a[i].key);

}

quickSort(a, 0, n);

fprintf(op, "Case %d:%d\n", caseID, n);

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

fprintf(op, "%d ", a[i].key);

}

fprintf(op, "%d\n", a[n - 1].key);

}

fclose(pf);

fclose(op);

}

**堆排**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_SIZE 10000

#define SWAP(i,j,tmp) temp=i,i=j,j=temp

typedef struct {

int key;

}element;

element a[MAX\_SIZE];

void adjust(element list[], int root, int n) {

int child, rootkey;

element temp;

temp = list[root];

rootkey = list[root].key;

child = root \* 2;

while (child <= n){

if ((child < n) && (list[child].key < list[child + 1].key)) { child++; }

if (rootkey > list[child].key) { break; }

else{

list[child / 2] = list[child];

child \*=2;

}

}

list[child / 2] = temp;

}

void heapSort(element list[], int n){

int i, j;

element temp;

for (i = n / 2; i > 0; i--) { adjust(list, i, n); }

for (i = n - 1; i > 0; i--){

SWAP(list[1], list[i + 1], temp);

adjust(list, 1, i);

}

}

int main(){

int n;

FILE \*pf, \*op;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

while (fscanf(pf, "%d", &n), n != -1) {

a[i].key = 0;

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

fscanf(pf, "%d", &a[i].key);

}

heapSort(a, n);

fprintf(op, "Case %d:%d\n", caseID, n);

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

fprintf(op, "%d ", a[i].key);

}

fprintf(op, "%d\n", a[n - 1].key);

}

fclose(pf);

fclose(op);

}

**归并排序**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_SIZE 10000

#define SWAP(i,j,tmp) temp=i,i=j,j=temp

typedef struct {

int key;

}element;

element a[MAX\_SIZE];

void merge(element list [], element sorted[], int i,int m,int n ) {

int j, k, t;

j = m + 1;

k = i;

while (i <= m&&j <= n) {

if (list[i].key <= list[j].key) { sorted[k++] = list[i++]; }

else{ sorted[k++] = list[j++]; }

}

if (i > m) {

for (t = j; t <= n; t++) {

sorted[k + t - j] = list[t];

}

}

else {

for (t = i; t <= m; t++) {

sorted[k + t - i] = list[t];

}

}

}

void mergePass(element list[],element sorted[],int n,int length) {

int i, j;

for (i = 0; i <= n - 2 \* length; i += 2 \* length) {

merge(list, sorted, i, i + length - 1, i+2\*length-1);

}

if (i + length < n) { merge(list, sorted, i, i + length - 1, n - 1); }

else {

for (j = i; j < n; j++) {

sorted[j] = list[j];

}

}

}

void mergeSort(element list [],int n) {

int length = 1;

element extra[MAX\_SIZE];

while (length < n) {

mergePass(list,extra, n, length);

length \*= 2;

mergePass(extra,list, n, length);

length \*= 2;

}

}

int main(){

int n;

FILE \*pf, \*op;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

while (fscanf(pf, "%d", &n), n != -1) {

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

fscanf(pf, "%d", &a[i].key);

}

mergeSort(a, n);

fprintf(op, "Case %d:%d\n", caseID, n);

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

fprintf(op, "%d ", a[i].key);

}

fprintf(op, "%d\n", a[n - 1].key);

}

fclose(pf);

fclose(op);

}

**数树的叶节点**

**数树的高**

void countLeave(tree\* root) {

if (root) {

if (!(root->left) && !(root->right)) { sum++; }

countLeave(root->left);

countLeave(root->right);

}

}

int countHeight(tree\* root) {

if (!root) return 0;

int height = 0;

int front = 0, rear = 0, last = 1;

tree \*queue[MAX\_QUEUE\_SIZE];

tree \*p;

queue[rear++] = root;

while (front < rear) {

p = queue[front++];

if (p->left) {

queue[rear++] = p->left;

}

if (p->right) {

queue[rear++] = p->right;

}

if (front == last) {

height++;

last = rear;

}

}

return height;

}

**树的建立（直接）**

typedef struct node {

char data;

struct node \*left, \*right;

}tree;

tree \*biTree = NULL;

void create() {

tree \*B = (tree\*)malloc(sizeof(tree));

tree \*D = (tree\*)malloc(sizeof(tree));

tree \*G = (tree\*)malloc(sizeof(tree));

tree \*C = (tree\*)malloc(sizeof(tree));

tree \*E = (tree\*)malloc(sizeof(tree));

tree \*F = (tree\*)malloc(sizeof(tree));

tree \*H = (tree\*)malloc(sizeof(tree));

biTree->data = 'A';

B->data = 'B';

D->data = 'D';

G->data = 'G';

C->data = 'C';

E->data = 'E';

F->data = 'F';

H->data = 'H';

biTree->left = B;

biTree->right = C;

B->left = D;

B->right = NULL;

D->left = NULL;

D->right = G;

G->left = NULL;

G->right = NULL;

C->left = E;

C->right = F;

E->left = NULL;

E->right = NULL;

F->left = H;

F->right = NULL;

H->left = NULL;

H->right = NULL;

}

**树的建立（前序输入 无为#）**

typedef struct node \*treePointer;

typedef struct node

{

char data;

treePointer leftChild,rightChild;

};

treePointer create(){

treePointer bt;

char ch;

scanf("%c",&ch);

if(ch=='#'){

bt=NULL;

}

else{

bt = (treePointer)malloc(sizeof(node));

bt->data=ch;

bt->leftChild=create();

bt->rightChild=create();

}

return bt;

}

**前序遍历**

void preorder(tree\* root) {

if (root) {

printf("%c ", root->data);

preorder(root->left);

preorder(root->right);

}

}

**中序遍历**

void inorder(tree\* root) {

if (root) {

inorder(root->left);

printf("%c ", root->data);

inorder(root->right);

}

}

**后序遍历**

void postorder(tree\* root) {

if (root) {

postorder(root->left);

postorder(root->right);

printf("%c ", root->data);

}

}

**中序迭代遍历**

void iter\_inorder(tree \*root) {

int top = -1;

tree \*stack[MAX\_STACK\_SIZE];

while (1) {

while (root) {

stack[++top] = root;

root = root->left;

}

if (top<0) { break; }

root = stack[top--];

printf("%c ", root->data);

root = root->right;

}

}

**层次序遍历**

void level\_order(tree \*root) {

int front = 0, rear = 0;

tree \*queue[MAX\_QUEUE\_SIZE];

if (!root) return;

queue[rear++] = root;

while (front<rear) {

root = queue[front++];

if (root) {

printf("%c ", root->data);

if (root->left) {

queue[rear++] = root->left;

}

if (root->right) {

queue[rear++] = root->right;

}

}

else { break; }

}

}

**MAIN**

int main() {

biTree = (tree\*)malloc(sizeof(tree));

create();

printf("preorder: \n");

preorder(biTree);

printf("\n");

}

**图的最短路径**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_VERTICES 11

#define FALSE 0

#define TRUE 1

int visited[MAX\_VERTICES];

int cost[MAX\_VERTICES][MAX\_VERTICES];

int path[MAX\_VERTICES] = { 1000 };

FILE \*pf, \*op;

typedef struct node \*node\_pointer;

typedef struct node {

int vertex;

int distance;

node\_pointer link;

};

node\_pointer graph[MAX\_VERTICES];

void Create(int edgeNum, int vertexNum) {

int vertex1, vertex2, distance;

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

graph[i] = (node\_pointer)malloc(sizeof(node));

graph[i]->vertex = i;

graph[i]->distance = 1000;

graph[i]->link = NULL;

}

for (int i = 0; i < MAX\_VERTICES; i++) {//初始化路径

for (int j = 0; j < MAX\_VERTICES; j++) {

cost[i][j] = 1000;

if (i == j) { cost[i][j] = 0; }

}

}

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

node\_pointer temp1, temp2;//temp2用来存尾部地址

temp1 = (node\_pointer)malloc(sizeof(node));

fscanf(pf,"%d %d %d", &vertex1, &vertex2, &distance);

temp1->vertex = vertex2;

temp1->distance = distance;

cost[vertex1][vertex2] = distance;

temp1->link = NULL;

temp2 = graph[vertex1];

while (temp2->link != NULL) {

temp2 = temp2->link;

}

temp2->link = temp1;

}

}

int choose(int vertexNum) {

int min = 1000, minPos = -1;

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

if (path[i] < min && !visited[i]) {

min = path[i];

minPos = i;

}

}

return minPos;

}

void shortestpath(int v, int vertexNum) {

int minPos;

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

visited[i] = 0;

path[i] = cost[v][i];

}

visited[v] = TRUE;

path[v] = 0;

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

minPos = choose(vertexNum);

visited[minPos] = TRUE;

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

if (!visited[j]) {

if (path[minPos] + cost[minPos][j] < path[j])

{ path[j] = path[minPos] + cost[minPos][j]; }

}

}

}

}

int main() {

int vertexNum, edgeNum;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

fscanf(pf,"%d %d", &vertexNum, &edgeNum);

Create(edgeNum, vertexNum);

printf("Shortest paths from v0 to each vertex are:\n");

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

shortestpath(0, vertexNum);

fprintf(op,"v0 to v%d:%5d\n", i, path[i]);

}

}

**图的BFS（广度优先）**

#include<stdio.h>

#include<time.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_VERTICES 11

#define FALSE 0

#define TRUE 1

int visited[MAX\_VERTICES];

int cost[MAX\_VERTICES][MAX\_VERTICES];

int path[MAX\_VERTICES] = { 1000 };

FILE \*pf, \*op;

typedef struct node \*node\_pointer;

typedef struct node {

int vertex;

int distance;

node\_pointer link;

};

void Create(int edgeNum, int vertexNum) {

int vertex1, vertex2, distance;

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

graph[i] = (node\_pointer)malloc(sizeof(node));

graph[i]->vertex = i;

graph[i]->distance = 1000;

graph[i]->link = NULL;

}

for (int i = 0; i < MAX\_VERTICES; i++) {//初始化路径

for (int j = 0; j < MAX\_VERTICES; j++) {

cost[i][j] = 1000;

if (i == j) { cost[i][j] = 0; }

}

}

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

node\_pointer temp1, temp2;//temp2用来存尾部地址

temp1 = (node\_pointer)malloc(sizeof(node));

fscanf(pf,"%d %d %d", &vertex1, &vertex2, &distance);

temp1->vertex = vertex2;

temp1->distance = distance;

cost[vertex1][vertex2] = distance;

temp1->link = NULL;

temp2 = graph[vertex1];

while (temp2->link != NULL) {

temp2 = temp2->link;

}

temp2->link = temp1;

}

}

node\_pointer graph[MAX\_VERTICES];

typedef struct queue \*queue\_pointer;

typedef struct queue {

int vertex;

queue\_pointer link;

};

queue\_pointer front, rear;

void Clear(int n) {

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

visited[i] = 0;

while (graph[i]) {

node\_pointer temp = graph[i];

graph[i] = graph[i]->link;

free(temp);

}

}

}

void addq(int v) {

queue\_pointer temp1;

temp1 = (queue\_pointer)malloc(sizeof(queue));

temp1->vertex = v;

temp1->link = NULL;

if (!front) {

front = temp1;

}

else {

rear->link = temp1;

}

rear = temp1;

}

int deleteq() {

int v = front->vertex;

queue\_pointer temp1 = front;

front = temp1->link;

free(temp1);//减少内存的浪费

return v;

}

void bfs(int v) {

node\_pointer temp;

front = rear = NULL;

memset(visited, 0, sizeof(visited));

visited[v] = TRUE;

printf("%5d", v);

addq(v);

while (front) {

v = deleteq();

for (temp = graph[v]; temp; temp = temp->link) {

if (!visited[temp->vertex]) {

visited[temp->vertex] = TRUE;

printf("%5d", temp->vertex);

addq(temp->vertex);

}

}

}

}

int main() {

int vertexNum, edgeNum;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

fscanf(pf,"%d %d", &vertexNum, &edgeNum);

Create(edgeNum, vertexNum);

printf("The sequence of vertex names getting from Width-First Search:\n");

bfs(1);

for (int i = 0; i < vertexNum; i++) {// 不连通分支

if (visited[i] == 0) { bfs(i); }

}

fprintf(op, "\n");

}

**图的DFS（深度优先）**

#include<stdio.h>

#include<time.h>

#include<stdlib.h>

#include<string.h>

#define MAX\_VERTICES 11

#define FALSE 0

#define TRUE 1

int visited[MAX\_VERTICES];

int cost[MAX\_VERTICES][MAX\_VERTICES];

int path[MAX\_VERTICES] = { 1000 };

FILE \*pf, \*op;

typedef struct node \*node\_pointer;

typedef struct node {

int vertex;

int distance;

node\_pointer link;

};

node\_pointer graph[MAX\_VERTICES];

void Create(int edgeNum, int vertexNum) {

int vertex1, vertex2, distance;

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

graph[i] = (node\_pointer)malloc(sizeof(node));

graph[i]->vertex = i;

graph[i]->distance = 1000;

graph[i]->link = NULL;

}

for (int i = 0; i < MAX\_VERTICES; i++) {//初始化路径

for (int j = 0; j < MAX\_VERTICES; j++) {

cost[i][j] = 1000;

if (i == j) { cost[i][j] = 0; }

}

}

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

node\_pointer temp1, temp2;//temp2用来存尾部地址

temp1 = (node\_pointer)malloc(sizeof(node));

fscanf(pf,"%d %d %d", &vertex1, &vertex2, &distance);

temp1->vertex = vertex2;

temp1->distance = distance;

cost[vertex1][vertex2] = distance;

temp1->link = NULL;

temp2 = graph[vertex1];

while (temp2->link != NULL) {

temp2 = temp2->link;

}

temp2->link = temp1;

}

}

void printGraph(int vertexNum) {

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

fprintf(op,"%d:", graph[i]->vertex);

node\_pointer temp = graph[i]->link;

while (temp) {

fprintf(op,"%d ", temp->vertex);

temp = temp->link;

}

fprintf(op,"\n");

}

}

//DFS

void dfs(int v) {

node\_pointer temp;

visited[v] = TRUE;

printf("%5d", v);

for (temp = graph[v]; temp; temp = temp->link) {

if (!visited[temp->vertex]) {

dfs(temp->vertex);

}

}

}

int main() {

int vertexNum, edgeNum;

pf = fopen("input.txt", "r");

op = fopen("output.txt", "w");

fscanf(pf,"%d %d", &vertexNum, &edgeNum);

Create(edgeNum, vertexNum);

printf("The sequence of vertex names getting from Depth-First Search:\n");

dfs(1);

for (int i = 0; i < vertexNum; i++) {// 不连通分支

if (visited[i] == 0) { dfs(i); }

}

fprintf(op, "\n");

}