**实 验 报 告**

1. **实验名称：**非线性结构及相关算法的设计与应用

**二、实验学时：**8学时

**三、实验目的：**掌握非线性结构的相关算法的设计与应用的方法

**四、实验原理：**

二叉树，链表，二叉树的序列化与反序列化，dijksra算法，floyd算法

**五、实验内容：**

第1部分：二叉树的链式存储、序列化和反序列化

第2部分：公园景点间的最短路径查询程序的设计

**六、实验器材（设备、元器件）：**

电脑。

**七、实验步骤：**

用链表来实现对二叉树的存储，其中运用到了对文件的读写、链表操作、递归算法、递归栈的模拟

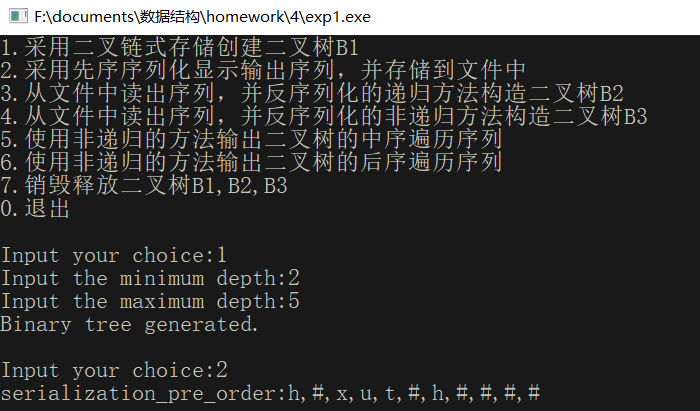
**八、实验结果与分析（含重要数据结果分析或核心代码流程分析）**

**（一）、第一部分**

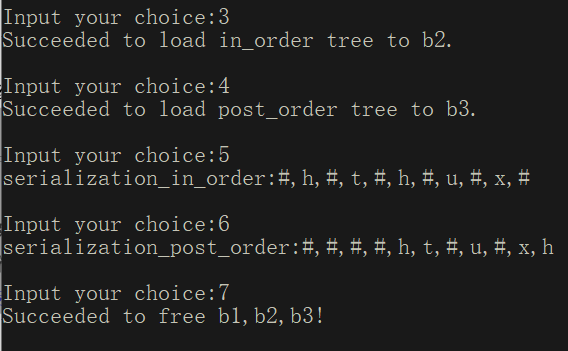
1. **程序运行展示**

随机生成了一个二叉树B1

并且将这个二叉树先序序列化



接下来从文件中分别用非递归和递归的方法读取刚刚所保存的那棵树，并且将它们分别以中序遍历和后序遍历的方法输出



1. **代码分析**

**利用随机函数，在一定的范围中自动生成一棵二叉树（利用递归生成）**

1. BT \*Generate\_BT(**int** now, **int** mind, **int** maxd){//随机生成一棵二叉树
2. BT \*rt = NULL;
3. **if**((rand()%10 > 3)&&(now <= maxd) || (now < mind) ){
4. rt = (BT\*) malloc(**sizeof**(BT));
5. rt->data = (rand()%26) + 'a';
6. rt->lch = Generate\_BT(now+1, mind, maxd);
7. rt->rch = Generate\_BT(now+1, mind, maxd);
8. }
9. **return** rt;
10. }

**先序遍历将二叉树序列化**

**其中运用了文件读写的相关知识**

1. **void** Pre\_Order(BT \*rt, **FILE** \*f){//先序遍历显示输出序列
2. Dot\_f(f);
3. **if**(rt == NULL){
4. printf("#");
5. fprintf(f, "#");
6. **return**;
7. }
8. printf("%c", rt->data);
9. fprintf(f, "%c", rt->data);
11. Pre\_Order(rt->lch, f);
12. Pre\_Order(rt->rch, f);
14. }

**利用递归的方法来对文件中的序列化的二叉树进行反序列化**

1. BT \*Deserialize\_RE(**FILE** \*f){//递归反序列化
2. BT \*rt = NULL;
3. **char** c = Read\_node(f);
4. **if**(c == '#') **return** rt;
5. rt = (BT\*) malloc(**sizeof**(BT));
6. rt->data = c;
7. rt->lch = Deserialize\_RE(f);
8. rt->rch = Deserialize\_RE(f);
9. **return** rt;
10. }

**采用非递归的方式进行反序列化二叉树**

**我并没有采用书上的实现方式**

**而是利用STL中的栈，来模拟程序中的递归栈，进而实现整个过程**

**其中process变量是用来储存当前结点进行到了哪一个步骤**

**所有步骤分为：**

1. **验证该结点是否为空，并读取这个结点的信息**
2. **遍历其左子树**
3. **遍历其右子树**
4. **返回**
5. BT \*Deserialize\_NOT(**FILE** \*f){//非递归反序列化
6. stack<BT\*> mystack;
7. BT \*rt = NULL;
8. BT \*ret;
9. **char** c = Read\_node(f);
10. **if**(c == '#') **return** rt;
11. rt = (BT\*) malloc(**sizeof**(BT));
12. rt->process = 1;
13. rt->data = c;
14. mystack.push(rt);
16. **while**(mystack.empty() == 0){
17. BT \*t = mystack.top();
18. **if**(t->process == 0){//验证该结点是否存在
19. t->data = Read\_node(f);
20. **if**(t->data == '#'){//结点不存在
21. free(t);
22. ret = NULL;
23. mystack.pop();
24. **continue**;
25. }**else**{//结点存在
26. t->process += 1;
27. **continue**;
28. }
30. }**else** **if**(t->process == 1){//左子树
31. t->lch = (BT\*) malloc(**sizeof**(BT));
32. mystack.push(t->lch);
33. t->lch->process = 0;
34. t->process += 1;
35. **continue**;
36. }**else** **if**(t->process == 2){//右子树
37. t->lch = ret;
38. t->rch = (BT\*) malloc(**sizeof**(BT));
39. mystack.push(t->rch);
40. t->rch->process = 0;
41. t->process += 1;
42. **continue**;
43. }**else**{//返回
44. t->rch = ret;
45. mystack.pop();
46. ret = t;
47. **continue**;
48. }
49. }
50. **return** rt;
51. }

**用书上的方法来实现的一个中序遍历**

**要对空节点进行特判，以免程序忽略掉要输出’#’的步骤**

1. **void** In\_Order(BT \*rt){
2. stack<BT\*> mystack;
3. BT \*p;
4. p = rt;
5. **while**(mystack.empty()==0 || p!=NULL){
6. **while**(p != NULL){//循环找到最左边的叶子
7. mystack.push(p);
8. p = p->lch;
9. }
11. Dot();
12. printf("#,");
14. **if**(mystack.empty() == 0){//输出该叶子并且回退，然后找到最邻近的右儿子
15. p = mystack.top();
16. mystack.pop();
18. printf("%c", p->data);
19. p = p->rch;
20. }
21. }
22. Dot();
23. printf("#");
24. }

**依然是使用自己的方法来后序遍历整个二叉树并输出**

1. **void** Post\_Order(BT \*rt){
2. stack<BT\*> mystack;
3. BT \*p;
4. **if**(rt == NULL){
5. printf("#");
6. **return**;
7. }
9. mystack.push(rt);
10. rt->process = 0;
11. **while**(mystack.empty() == 0){
12. p = mystack.top();
13. **if**(p->process == 0){
14. **if**(p == NULL){
15. Dot();
16. printf("#");
17. mystack.pop();
18. **continue**;
19. }**else**{
20. **if**(p->lch == NULL){
21. Dot();
22. printf("#");
23. }**else**{
24. mystack.push(p->lch);
25. p->lch->process = 0;
26. }
27. p->process += 1;
28. **continue**;
29. }
30. }**else** **if**(p->process == 1){
31. **if**(p->rch == NULL){
32. Dot();
33. printf("#");
34. }**else**{
35. mystack.push(p->rch);
36. p->rch->process = 0;
37. }
38. p->process += 1;
39. **continue**;
40. }**else**{
41. Dot();
42. printf("%c", p->data);
43. mystack.pop();
44. **continue**;
45. }
46. }
47. }

**程序的main函数：**

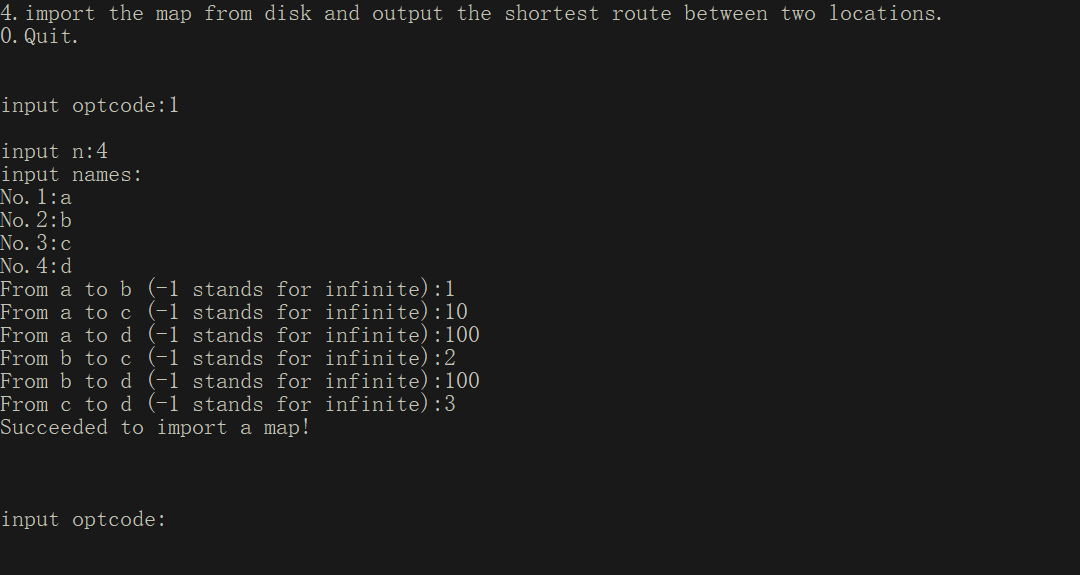
**写得比较规范、简洁，各种函数一目了然**

1. **int** main(){
2. srand(time(NULL));
4. BT \*b1 = NULL;
5. BT \*b2 = NULL;
6. BT \*b3 = NULL;
7. **int** Choice;
8. **FILE** \*f;
9. Print\_Manu();
10. **while**(1){
11. printf("Input your choice:");
12. scanf("%d", &Choice);
13. **if**(!Choice) **break**;
14. **switch**(Choice){
16. **case** 1:{
17. **int** mind, maxd;
18. printf("Input the minimum depth:");
19. scanf("%d", &mind);
20. printf("Input the maximum depth:");
21. scanf("%d", &maxd);
22. **if**(mind > maxd){
23. printf("Invalid!\n\n");
24. **continue**;
25. }
26. Free\_BT(b1); //把之前储存的树销毁
27. b1 = Generate\_BT(1, mind, maxd);
28. printf("Binary tree generated.\n\n");
29. **break**;
30. }
31. **case** 2:{
32. f = fopen("tree.txt", "w");
33. printf("serialization\_pre\_order:");
34. FLAG = 0;
35. Pre\_Order(b1, f);
36. fclose(f);
37. printf("\n\n");
39. **break**;
40. }
41. **case** 3:{
42. f = fopen("tree.txt", "r");
43. b2 = Deserialize\_RE(f);
44. fclose(f);
45. printf("Succeeded to load in\_order tree to b2.\n\n");
47. **break**;
48. }
50. **case** 4:{
51. f = fopen("tree.txt", "r");
52. b3 = Deserialize\_NOT(f);
53. fclose(f);
54. printf("Succeeded to load post\_order tree to b3.\n\n");
56. **break**;
57. }
59. **case** 5:{
60. printf("serialization\_in\_order:");
61. FLAG = 0;
62. In\_Order(b2);
63. printf("\n\n");
64. **break**;
65. }
67. **case** 6:{
68. printf("serialization\_post\_order:");
69. FLAG = 0;
70. Post\_Order(b3);
71. printf("\n\n");
72. **break**;
73. }
75. **case** 7:{
76. Free\_BT(b1);
77. Free\_BT(b2);
78. Free\_BT(b3);
79. b1 = b2 = b3 = NULL;
80. printf("Succeeded to free b1,b2,b3!\n\n");
82. **break**;
83. }
84. }
85. }



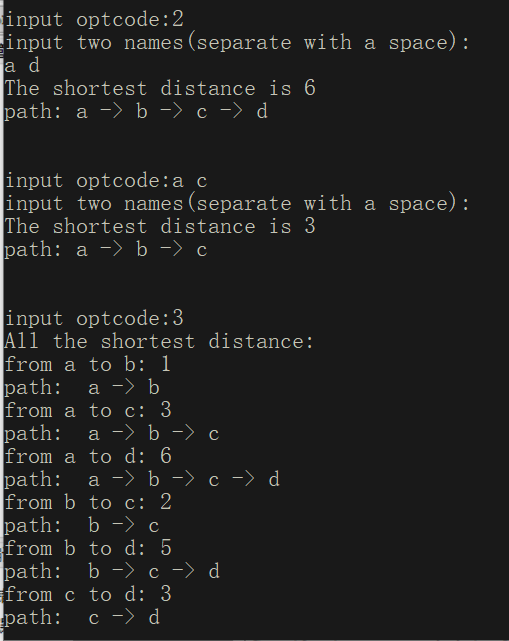
90. }
    1. **、第二部分**
91. **程序运行展示**

**读取一个地图的所有结点，以及各个结点之间的距离**



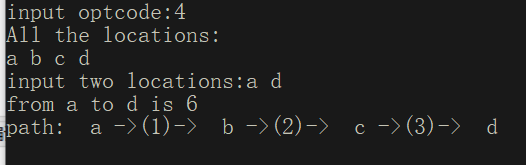
**对两个结点之间最短路径的计算、**

**以及对所有节点之间最短距离的计算(并且输出了整个最短路径)**



**从文件中读取了最短路径信息，并且要求用户输入两个结点，进行查询**

**输出了两个结点之间的距离，并且把整个最短路径（包括每条路的长度）输出了出来**



1. **代码分析**

**从屏幕中读取整个地图的信息，包括每个结点的名字，以及各个结点之间的距离**

1. **void** Input\_Map(**int** n){
2. printf("input names:\n");
3. **for**(**int** i = 1; i <= n; i++){
4. printf("No.%d:", i);
5. scanf("%s", name[i]);
6. }
8. **for**(**int** i = 1; i <= n; i++)
9. **for**(**int** j = 1; j <= n; j++)
10. map[i][j] = INF;
12. **for**(**int** i = 1; i <= n; i++){
13. **for**(**int** j = i+1; j <= n; j++){
14. **int** t;
15. printf("From %s to %s (-1 stands for infinite):", name[i], name[j]);
16. scanf("%d",&t);
17. **if**(t == -1) **continue**;//距离无限远
18. map[i][j] = map[j][i] = t;
19. }
20. map[i][i] = 0;
21. }
22. }

**根据结点的名称来查找这个结点的序号，方便后面对最短距离的计算**

1. **int** Search\_Id(**char** str[]){
2. **for**(**int** i = 1; i <= n; i++)
3. **if**(strcmp(name[i], str)==0)
4. **return** i;
5. **return** -1;
6. }

**最朴素的dijkstra算法，只能计算一个起点到其余所有节点的距离，所以适用于两两节点之间最短距离的查询**

**注意**fa[j] = u;**这个句子，我把某个最短路径的所有结点以树的形式逆向串起来了，在输出的时候只需要把这个序列倒序输出即可**

1. **int** Dijkstra(**int** s, **int** t){
2. **for**(**int** i = 1; i <= n; i++)
3. map[i][i] = 0;//要把所有的结点首先剔除S集合
4. map[s][s] = 1;

7. **for**(**int** i = 1; i <= n; i++){//将s的邻接点加入dist
8. dist[i] = INF;
9. fa[i] = i;
10. **if**(map[s][i] < INF){
11. dist[i] = map[s][i];
12. fa[i] = s;
13. }
14. }
16. **for**(**int** i = 1; i <= n-1; i++){//循环n-1次
17. **int** MIN = INF, u = 0;
18. **for**(**int** j = 1; j <= n; j++){//在剩下的点中找一个与S集合邻接的，距离s点最短的点
19. **if**(map[j][j] == 0 && dist[j]<MIN){
20. u = j;
21. MIN = dist[j];
22. }
23. }
24. map[u][u] = 1;//加入这个点到S集合中
25. **for**(**int** j = 1; j <= n; j++)
26. **if**(map[j][j] == 0)//找到了u点，接下来把与u点邻接的所有边更新一下
27. **if**(map[u][j] < INF && dist[u] + map[u][j] < dist[j]){
28. dist[j] = dist[u] + map[u][j];
29. fa[j] = u;//把j的父亲设为u
30. }
32. }
34. **return** dist[t];//返回距离
35. }

**Floyd算法，适用于所有结点之间最短路径的计算，当然时间复杂度也要高一些，有n^3之高，但是比dijkstra算法更为方便实现，且算法也比较稳定，不会出现特殊情况来影响算法复杂度的情况**

**注意** path[i][j] = path[i][k];**这个代码，我利用一个二维数组巧妙地把所有的最短路径完整地存了下来。意义如下：**

**Path[i][j]=k代表从i到j的最短路径上，离i最近的结点为k，这样利于输出**

1. **void** Floyd(){
2. **for**(**int** i = 1; i <= n; i++)
3. **for**(**int** j = 1; j <= n; j++)
4. path[i][j] = j;//初始化path数组
5. memcpy(shortest, map, **sizeof**(map));
6. **for**(**int** k = 1; k <= n; k++)
7. **for**(**int** i = 1; i <= n; i++)
8. **for**(**int** j = 1; j <= n; j++)
9. **if**(shortest[i][j] > shortest[i][k] + shortest[k][j]){
10. shortest[i][j] = shortest[i][k] + shortest[k][j];
11. path[i][j] = path[i][k];//代表i到j的路径上，i的下一个结点为k
12. //重要 这段代码值得好好理解
13. }
14. }

**Main函数展示，依旧是把每个要实现的功能分门别类地放在了switch的每个case里面，方便debug以及阅读，非常清爽，有利于代码审计**

**其中case2，我用STL开了一个栈，将dijkstra生成的最短路径用栈的方式实现了逆序变正序，正确且快速地读取并输出了两点之间的最短路径**

1. **int** main(){
2. Print\_Manu();
3. **int** Choice;
4. **FILE** \*f;
6. **do**{
7. printf("\n\ninput optcode:");
8. scanf("%d", &Choice);
9. **switch**(Choice){
10. **case** 1:{
11. printf("\ninput n:");
12. scanf("%d", &n);
13. Input\_Map(n);
14. printf("Succeeded to import a map!\n\n");
15. **break**;
16. }
17. **case** 2:{
18. printf("input two names(separate with a space):\n");
19. **char** name1[20], name2[20];
20. **int** s, t;
21. scanf("%s%s", name1, name2);
22. s = Search\_Id(name1);
23. t = Search\_Id(name2);
25. **if**(s < 0 || t < 0)
26. printf("Invalid name!\n");
27. **else**{
28. **int** res = Dijkstra(s, t);
29. printf("The shortest distance is %d\n", res);
30. **if**(res == INF)
31. printf("which means no path.\n");
32. **else**{
33. printf("path: ");
34. **int** k = t;
35. stack <**int**> mystack;
36. **while**(k != s){
37. mystack.push(k);
38. k = fa[k];
39. }
41. printf("%s ", name[s]);
42. **while**(mystack.empty() != 1){
43. printf("-> %s ", name[mystack.top()]);
44. mystack.pop();
45. }
46. printf("\n");
47. }
48. }

51. **break**;
52. }
54. **case** 3:{
55. f = fopen("AllPath.dat", "w");
56. Floyd();
57. printf("All the shortest distance:\n");
59. fprintf(f, "%d\n", n);
60. **for**(**int** i = 1; i <= n; i++)
61. fprintf(f, "%s ", name[i]);
62. fprintf(f, "\n");
64. **for**(**int** i = 1; i <= n; i++)
65. **for**(**int** j = i+1; j <= n; j++){
66. printf("from %s to %s: %d\n", name[i], name[j], shortest[i][j]);
67. fprintf(f, "%s %s %d\n", name[i], name[j], shortest[i][j]);
68. **if**(shortest[i][j] == INF)//没有路径
69. printf("which means no path.\n");
70. **else**{
71. printf("path: ");
72. **int** k = i;
73. **while**(k != j){
74. printf(" %s ->", name[k]);
75. k = path[k][j];
76. }
77. printf(" %s\n",name[j]);
78. }
79. }
81. //以下是记录两个节点的最短路径与之间的距离
82. **for**(**int** i = 1; i <= n; i++){
83. **for**(**int** j = 1; j <= n; j++)
84. fprintf(f, "%d ", map[i][j]);
85. fprintf(f, "\n");
86. }
88. **for**(**int** i = 1; i <= n; i++){
89. **for**(**int** j = 1; j <= n; j++)
90. fprintf(f, "%d ", path[i][j]);
91. fprintf(f, "\n");
92. }
94. fclose(f);
96. **break**;
97. }
99. **case** 4:{
100. **char** name1[20], name2[20];
101. **int** t;
102. f = fopen("AllPath.dat", "r");
103. fscanf(f, "%d", &n);
104. **for**(**int** i = 1; i <= n; i++)
105. fscanf(f, "%s", name[i]);
106. **for**(**int** i = 1; i <= n\*(n-1)/2; i++){
107. fscanf(f, "%s %s %d", name1, name2, &t);
108. shortest[Search\_Id(name1)][Search\_Id(name2)] = t;
109. shortest[Search\_Id(name2)][Search\_Id(name1)] = t;
110. }
112. //以下是读取两个节点的最短路径与之间的距离
113. **for**(**int** i = 1; i <= n; i++)
114. **for**(**int** j = 1; j <= n; j++)
115. fscanf(f, "%d ", &map[i][j]);
117. **for**(**int** i = 1; i <= n; i++)
118. **for**(**int** j = 1; j <= n; j++)
119. fscanf(f, "%d ", &path[i][j]);

122. printf("All the locations:\n");
123. **for**(**int** i = 1; i <= n; i++)
124. printf("%s ",name[i]);
125. printf("\ninput two locations:");
126. scanf("%s%s",name1,name2);
127. **int** u = Search\_Id(name1);
128. **int** v = Search\_Id(name2);
129. **if**(u==-1 || v==-1)
130. printf("Invalid name!\n");
131. **else** printf("from %s to %s is %d\n", name1, name2, shortest[u][v]);
133. **if**(shortest[u][v]==INF)
134. printf("which means no path.\n");
135. **else**{
136. printf("path: ");
137. **int** k = u;
138. **while**(k != v){
139. printf(" %s ->(%d)-> ", name[k], map[k][path[k][v]]);
140. k = path[k][v];
141. }
142. printf(" %s\n",name[v]);
143. }

146. fclose(f);
147. **break**;
148. }


152. }

155. }**while**(Choice);

158. }

**九、总结及心得体会：**

**二叉树的序列化是对二叉树进行格式化存储的一个解决方法，十分重要。**

**在实现二叉树的遍历时，有两种方法可以使用，一种是递归实现，一种是非递归实现，递归实现固然简单，但是在数据量过大时会发生爆栈的可能，这时候使用非递归方式显得更为高效和安全**

**dijkstra算法适合于两个点之间最短距离的计算**

**Floyd算法更适合于所有结点之间最短距离的计算**

**虽然这两个算法各有优劣，但是，他们都可以用来计算两个结点之间的最短路径（并且可以保存，不会丢失信息），而且对于每个算法，路径的保存也只是一段代码的问题，大大降低了我们的编程复杂度**

**十、对本实验过程及方法、手段的改进建议：**

**书上所编写的dijkstra算法并不完美，因为在寻找最短路的时候会遍历所有的附属结点来找一条最短路，如果将程序套上一层堆（优先队列，那么就能把算法复杂度降为nlogn）**

**十一、附完整代码**

1. 第一部分
2. #include<cstdio>
3. #include<cstdlib>
4. #include<ctime>
5. #include<stack>
6. **using** **namespace** std;
8. **int** FLAG;
9. **struct** BT{
10. **char** data;
11. BT \*lch, \*rch;
12. **int** process;//记录当前递归到左子树还是右子树
13. };
15. **void** Free\_BT(BT \*rt){
16. **if**(rt == NULL) **return**;
17. Free\_BT(rt->lch);
18. Free\_BT(rt->rch);
19. free(rt);
20. }
22. BT \*Generate\_BT(**int** now, **int** mind, **int** maxd){//随机生成一棵二叉树
23. BT \*rt = NULL;
24. **if**((rand()%10 > 3)&&(now <= maxd) || (now < mind) ){
25. rt = (BT\*) malloc(**sizeof**(BT));
26. rt->data = (rand()%26) + 'a';
27. rt->lch = Generate\_BT(now+1, mind, maxd);
28. rt->rch = Generate\_BT(now+1, mind, maxd);
29. }
30. **return** rt;
31. }
33. **void** Print\_Manu(){
34. printf("1.采用二叉链式存储创建二叉树B1\n");
35. printf("2.采用先序序列化显示输出序列，并存储到文件中\n");
36. printf("3.从文件中读出序列，并反序列化的递归方法构造二叉树B2\n");
37. printf("4.从文件中读出序列，并反序列化的非递归方法构造二叉树B3\n");
38. printf("5.使用非递归的方法输出二叉树的中序遍历序列\n");
39. printf("6.使用非递归的方法输出二叉树的后序遍历序列\n");
40. printf("7.销毁释放二叉树B1,B2,B3\n");
41. printf("0.退出\n\n");
42. }
44. **void** Dot\_f(**FILE** \*f){
45. **if**(FLAG){
46. printf(",");
47. fprintf(f, ",");
48. }**else** FLAG = 1;
49. }
51. **void** Dot(){
52. **if**(FLAG){
53. printf(",");
54. }**else** FLAG = 1;
55. }
57. **char** Read\_node(**FILE** \*f){
58. **char** c;
59. **do**{
60. fscanf(f, "%c", &c);
61. }**while**(c == ',');
62. **return** c;
63. }
65. **void** Pre\_Order(BT \*rt, **FILE** \*f){//先序遍历显示输出序列
66. Dot\_f(f);
67. **if**(rt == NULL){
68. printf("#");
69. fprintf(f, "#");
70. **return**;
71. }
72. printf("%c", rt->data);
73. fprintf(f, "%c", rt->data);
75. Pre\_Order(rt->lch, f);
76. Pre\_Order(rt->rch, f);
78. }
80. BT \*Deserialize\_RE(**FILE** \*f){//递归反序列化
81. BT \*rt = NULL;
82. **char** c = Read\_node(f);
83. **if**(c == '#') **return** rt;
84. rt = (BT\*) malloc(**sizeof**(BT));
85. rt->data = c;
86. rt->lch = Deserialize\_RE(f);
87. rt->rch = Deserialize\_RE(f);
88. **return** rt;
89. }


93. BT \*Deserialize\_NOT(**FILE** \*f){//非递归反序列化
94. stack<BT\*> mystack;
95. BT \*rt = NULL;
96. BT \*ret;
97. **char** c = Read\_node(f);
98. **if**(c == '#') **return** rt;
99. rt = (BT\*) malloc(**sizeof**(BT));
100. rt->process = 1;
101. rt->data = c;
102. mystack.push(rt);
104. **while**(mystack.empty() == 0){
105. BT \*t = mystack.top();
106. **if**(t->process == 0){//验证该结点是否存在
107. t->data = Read\_node(f);
108. **if**(t->data == '#'){//结点不存在
109. free(t);
110. ret = NULL;
111. mystack.pop();
112. **continue**;
113. }**else**{//结点存在
114. t->process += 1;
115. **continue**;
116. }
118. }**else** **if**(t->process == 1){//左子树
119. t->lch = (BT\*) malloc(**sizeof**(BT));
120. mystack.push(t->lch);
121. t->lch->process = 0;
122. t->process += 1;
123. **continue**;
124. }**else** **if**(t->process == 2){//右子树
125. t->lch = ret;
126. t->rch = (BT\*) malloc(**sizeof**(BT));
127. mystack.push(t->rch);
128. t->rch->process = 0;
129. t->process += 1;
130. **continue**;
131. }**else**{//返回
132. t->rch = ret;
133. mystack.pop();
134. ret = t;
135. **continue**;
136. }
137. }
138. **return** rt;
139. }
141. **void** In\_Order(BT \*rt){
142. stack<BT\*> mystack;
143. BT \*p;
144. p = rt;
145. **while**(mystack.empty()==0 || p!=NULL){
146. **while**(p != NULL){//循环找到最左边的叶子
147. mystack.push(p);
148. p = p->lch;
149. }
151. Dot();
152. printf("#,");
154. **if**(mystack.empty() == 0){//输出该叶子并且回退，然后找到最邻近的右儿子
155. p = mystack.top();
156. mystack.pop();
158. printf("%c", p->data);
159. p = p->rch;
160. }
161. }
162. Dot();
163. printf("#");
164. }
166. **void** Post\_Order(BT \*rt){
167. stack<BT\*> mystack;
168. BT \*p;
169. **if**(rt == NULL){
170. printf("#");
171. **return**;
172. }
174. mystack.push(rt);
175. rt->process = 0;
176. **while**(mystack.empty() == 0){
177. p = mystack.top();
178. **if**(p->process == 0){
179. **if**(p == NULL){
180. Dot();
181. printf("#");
182. mystack.pop();
183. **continue**;
184. }**else**{
185. **if**(p->lch == NULL){
186. Dot();
187. printf("#");
188. }**else**{
189. mystack.push(p->lch);
190. p->lch->process = 0;
191. }
192. p->process += 1;
193. **continue**;
194. }
195. }**else** **if**(p->process == 1){
196. **if**(p->rch == NULL){
197. Dot();
198. printf("#");
199. }**else**{
200. mystack.push(p->rch);
201. p->rch->process = 0;
202. }
203. p->process += 1;
204. **continue**;
205. }**else**{
206. Dot();
207. printf("%c", p->data);
208. mystack.pop();
209. **continue**;
210. }
211. }
212. }

215. **int** main(){
216. srand(time(NULL));
218. BT \*b1 = NULL;
219. BT \*b2 = NULL;
220. BT \*b3 = NULL;
221. **int** Choice;
222. **FILE** \*f;
223. Print\_Manu();
224. **while**(1){
225. printf("Input your choice:");
226. scanf("%d", &Choice);
227. **if**(!Choice) **break**;
228. **switch**(Choice){
230. **case** 1:{
231. **int** mind, maxd;
232. printf("Input the minimum depth:");
233. scanf("%d", &mind);
234. printf("Input the maximum depth:");
235. scanf("%d", &maxd);
236. **if**(mind > maxd){
237. printf("Invalid!\n\n");
238. **continue**;
239. }
240. Free\_BT(b1); //把之前储存的树销毁
241. b1 = Generate\_BT(1, mind, maxd);
242. printf("Binary tree generated.\n\n");
243. **break**;
244. }
245. **case** 2:{
246. f = fopen("tree.txt", "w");
247. printf("serialization\_pre\_order:");
248. FLAG = 0;
249. Pre\_Order(b1, f);
250. fclose(f);
251. printf("\n\n");
253. **break**;
254. }
255. **case** 3:{
256. f = fopen("tree.txt", "r");
257. b2 = Deserialize\_RE(f);
258. fclose(f);
259. printf("Succeeded to load in\_order tree to b2.\n\n");
261. **break**;
262. }
264. **case** 4:{
265. f = fopen("tree.txt", "r");
266. b3 = Deserialize\_NOT(f);
267. fclose(f);
268. printf("Succeeded to load post\_order tree to b3.\n\n");
270. **break**;
271. }
273. **case** 5:{
274. printf("serialization\_in\_order:");
275. FLAG = 0;
276. In\_Order(b2);
277. printf("\n\n");
278. **break**;
279. }
281. **case** 6:{
282. printf("serialization\_post\_order:");
283. FLAG = 0;
284. Post\_Order(b3);
285. printf("\n\n");
286. **break**;
287. }
289. **case** 7:{
290. Free\_BT(b1);
291. Free\_BT(b2);
292. Free\_BT(b3);
293. b1 = b2 = b3 = NULL;
294. printf("Succeeded to free b1,b2,b3!\n\n");
296. **break**;
297. }
298. }
299. }
301. }

2.第二部分

1. #include<cstdio>
2. #include<cstdlib>
3. #include<ctime>
4. #include<cstring>
5. #include<stack>
6. #include<iostream>
7. #define MAX 500
8. #define INF 1e9+7
9. **using** **namespace** std;
11. **int** map[MAX+1][MAX+1], n;
12. **int** shortest[MAX+1][MAX+1];
13. **int** dist[MAX+1],path[MAX+1][MAX+1]; //path数组负责记录从u到v，以u为起点的下一个结点的位置
14. **char** name[MAX+1][20];
15. **int** fa[MAX+1];//用于在dijkstra里面保存某个结点的前一个结点

18. **void** Print\_Manu(){
19. printf("1.Import a map.(and store it in the disk)\n");
20. printf("2.Calculate the shortest distance from a to b.\n");
21. printf("3.Calculate the shortest route of all.(and store it in the disk)\n");
22. printf("4.import the map from disk and output the shortest route between two locations.\n");
23. printf("0.Quit.\n");
24. }
26. **void** Input\_Map(**int** n){
27. printf("input names:\n");
28. **for**(**int** i = 1; i <= n; i++){
29. printf("No.%d:", i);
30. scanf("%s", name[i]);
31. }
33. **for**(**int** i = 1; i <= n; i++)
34. **for**(**int** j = 1; j <= n; j++)
35. map[i][j] = INF;
37. **for**(**int** i = 1; i <= n; i++){
38. **for**(**int** j = i+1; j <= n; j++){
39. **int** t;
40. printf("From %s to %s (-1 stands for infinite):", name[i], name[j]);
41. scanf("%d",&t);
42. **if**(t == -1) **continue**;//距离无限远
43. map[i][j] = map[j][i] = t;
44. }
45. map[i][i] = 0;
46. }
47. }
49. **int** Search\_Id(**char** str[]){
50. **for**(**int** i = 1; i <= n; i++)
51. **if**(strcmp(name[i], str)==0)
52. **return** i;
53. **return** -1;
54. }
56. **int** Dijkstra(**int** s, **int** t){
57. **for**(**int** i = 1; i <= n; i++)
58. map[i][i] = 0;//要把所有的结点首先剔除S集合
59. map[s][s] = 1;

62. **for**(**int** i = 1; i <= n; i++){//将s的邻接点加入dist
63. dist[i] = INF;
64. fa[i] = i;
65. **if**(map[s][i] < INF){
66. dist[i] = map[s][i];
67. fa[i] = s;
68. }
69. }
71. **for**(**int** i = 1; i <= n-1; i++){//循环n-1次
72. **int** MIN = INF, u = 0;
73. **for**(**int** j = 1; j <= n; j++){//在剩下的点中找一个与S集合邻接的，距离s点最短的点
74. **if**(map[j][j] == 0 && dist[j]<MIN){
75. u = j;
76. MIN = dist[j];
77. }
78. }
79. map[u][u] = 1;//加入这个点到S集合中
80. **for**(**int** j = 1; j <= n; j++)
81. **if**(map[j][j] == 0)//找到了u点，接下来把与u点邻接的所有边更新一下
82. **if**(map[u][j] < INF && dist[u] + map[u][j] < dist[j]){
83. dist[j] = dist[u] + map[u][j];
84. fa[j] = u;//把j的父亲设为u
85. }
87. }
89. **return** dist[t];//返回距离
90. }
92. **void** Floyd(){
93. **for**(**int** i = 1; i <= n; i++)
94. **for**(**int** j = 1; j <= n; j++)
95. path[i][j] = j;//初始化path数组
96. memcpy(shortest, map, **sizeof**(map));
97. **for**(**int** k = 1; k <= n; k++)
98. **for**(**int** i = 1; i <= n; i++)
99. **for**(**int** j = 1; j <= n; j++)
100. **if**(shortest[i][j] > shortest[i][k] + shortest[k][j]){
101. shortest[i][j] = shortest[i][k] + shortest[k][j];
102. path[i][j] = path[i][k];//代表i到j的路径上，i的下一个结点为k
103. //重要 这段代码值得好好理解
104. }
105. }
107. **int** main(){
108. Print\_Manu();
109. **int** Choice;
110. **FILE** \*f;
112. **do**{
113. printf("\n\ninput optcode:");
114. scanf("%d", &Choice);
115. **switch**(Choice){
116. **case** 1:{
117. printf("\ninput n:");
118. scanf("%d", &n);
119. Input\_Map(n);
120. printf("Succeeded to import a map!\n\n");
121. **break**;
122. }
123. **case** 2:{
124. printf("input two names(separate with a space):\n");
125. **char** name1[20], name2[20];
126. **int** s, t;
127. scanf("%s%s", name1, name2);
128. s = Search\_Id(name1);
129. t = Search\_Id(name2);
131. **if**(s < 0 || t < 0)
132. printf("Invalid name!\n");
133. **else**{
134. **int** res = Dijkstra(s, t);
135. printf("The shortest distance is %d\n", res);
136. **if**(res == INF)
137. printf("which means no path.\n");
138. **else**{
139. printf("path: ");
140. **int** k = t;
141. stack <**int**> mystack;
142. **while**(k != s){
143. mystack.push(k);
144. k = fa[k];
145. }
147. printf("%s ", name[s]);
148. **while**(mystack.empty() != 1){
149. printf("-> %s ", name[mystack.top()]);
150. mystack.pop();
151. }
152. printf("\n");
153. }
154. }

157. **break**;
158. }
160. **case** 3:{
161. f = fopen("AllPath.dat", "w");
162. Floyd();
163. printf("All the shortest distance:\n");
165. fprintf(f, "%d\n", n);
166. **for**(**int** i = 1; i <= n; i++)
167. fprintf(f, "%s ", name[i]);
168. fprintf(f, "\n");
170. **for**(**int** i = 1; i <= n; i++)
171. **for**(**int** j = i+1; j <= n; j++){
172. printf("from %s to %s: %d\n", name[i], name[j], shortest[i][j]);
173. fprintf(f, "%s %s %d\n", name[i], name[j], shortest[i][j]);
174. **if**(shortest[i][j] == INF)//没有路径
175. printf("which means no path.\n");
176. **else**{
177. printf("path: ");
178. **int** k = i;
179. **while**(k != j){
180. printf(" %s ->", name[k]);
181. k = path[k][j];
182. }
183. printf(" %s\n",name[j]);
184. }
185. }
187. //以下是记录两个节点的最短路径与之间的距离
188. **for**(**int** i = 1; i <= n; i++){
189. **for**(**int** j = 1; j <= n; j++)
190. fprintf(f, "%d ", map[i][j]);
191. fprintf(f, "\n");
192. }
194. **for**(**int** i = 1; i <= n; i++){
195. **for**(**int** j = 1; j <= n; j++)
196. fprintf(f, "%d ", path[i][j]);
197. fprintf(f, "\n");
198. }
200. fclose(f);
202. **break**;
203. }
205. **case** 4:{
206. **char** name1[20], name2[20];
207. **int** t;
208. f = fopen("AllPath.dat", "r");
209. fscanf(f, "%d", &n);
210. **for**(**int** i = 1; i <= n; i++)
211. fscanf(f, "%s", name[i]);
212. **for**(**int** i = 1; i <= n\*(n-1)/2; i++){
213. fscanf(f, "%s %s %d", name1, name2, &t);
214. shortest[Search\_Id(name1)][Search\_Id(name2)] = t;
215. shortest[Search\_Id(name2)][Search\_Id(name1)] = t;
216. }
218. //以下是读取两个节点的最短路径与之间的距离
219. **for**(**int** i = 1; i <= n; i++)
220. **for**(**int** j = 1; j <= n; j++)
221. fscanf(f, "%d ", &map[i][j]);
223. **for**(**int** i = 1; i <= n; i++)
224. **for**(**int** j = 1; j <= n; j++)
225. fscanf(f, "%d ", &path[i][j]);

228. printf("All the locations:\n");
229. **for**(**int** i = 1; i <= n; i++)
230. printf("%s ",name[i]);
231. printf("\ninput two locations:");
232. scanf("%s%s",name1,name2);
233. **int** u = Search\_Id(name1);
234. **int** v = Search\_Id(name2);
235. **if**(u==-1 || v==-1)
236. printf("Invalid name!\n");
237. **else** printf("from %s to %s is %d\n", name1, name2, shortest[u][v]);
239. **if**(shortest[u][v]==INF)
240. printf("which means no path.\n");
241. **else**{
242. printf("path: ");
243. **int** k = u;
244. **while**(k != v){
245. printf(" %s ->(%d)-> ", name[k], map[k][path[k][v]]);
246. k = path[k][v];
247. }
248. printf(" %s\n",name[v]);
249. }

252. fclose(f);
253. **break**;
254. }


258. }

261. }**while**(Choice);

264. }

**感谢您的阅读！**

**报告评分：**

**指导教师签字：**