

**实 验（实训）报 告**

**学 院**  信息管理与工程学院

**班 级**  18软件工程2班

**学 号**  180110910603

**姓 名**  陈彬捷

**指导教师**  周志光

浙江财经大学教务处制

**目录**

[顺序表基础操作及复杂操作 2](#_Toc27999219)

[单链表基本操作和复杂操作 10](#_Toc27999220)

[静态链表基本操作和复杂操作 20](#_Toc27999221)

[栈的基本操作和四则混合运算 30](#_Toc27999222)

[循环队列应用 - 迷宫游戏 37](#_Toc27999223)

[树的孩子表现法 42](#_Toc27999224)

[二叉树的四种遍历方式 53](#_Toc27999225)

[哈夫曼编码 60](#_Toc27999227)

[图的深度和广度优先遍历分析(1) 65](#_Toc27999228)

[图的深度和广度优先遍历分析(2) 75](#_Toc27999229)

[Prim算法（最小生成树） 82](#_Toc27999230)

[Kruskal算法（最小生成树） 96](#_Toc27999231)

[Dijkstra算法（单源最短路） 110](#_Toc27999232)

[排序二叉树 120](#_Toc27999233)

[二叉排序树平衡化 128](#_Toc27999234)

[基于顺序表的二分查找实现 134](#_Toc27999235)

[索引查找的构建和实现 139](#_Toc27999236)

[二叉排序树平衡化 144](#_Toc27999237)

[八大排序结构算法 151](#_Toc27999238)

# 顺序表基础操作及复杂操作

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求用顺序表数据结构对数据进行存储和操作，包括创建顺序表、插入和删除顺序表中数据以及计算顺序表的表长等基础操作和计算得出顺序表的交集并集并输出，对顺序表中的数据进行排序等复杂操作。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 顺序表是在计算机内存中以数组的形式保存的线性表，线性表的顺序存储是指用一组地址连续的存储单元依次存储线性表中的各个元素、使得线性表中在逻辑结构上相邻的数据元素存储在相邻的物理存储单元中，即通过数据元素物理存储的相邻关系来反映数据元素之间逻辑上的相邻关系，采用顺序存储结构的线性表通常称为顺序表。顺序表是将表中的结点依次存放在计算机内存中一组地址连续的存储单元中。  将表中元素一个接一个的存入一组连续的存储单元中，这种存储结构是顺序结构。    采用顺序存储结构的线性表简称为“ 顺序表”。顺序表的存储特点是：只要确定了起始位置，表中任一元素的地址都通过下列公式得到：LOC（ai）=LOC（a1）+（i-1）\*L 　1≤i≤n 其中，L是元素占用存储单元的长度。  顺序表的结构定义：  #define maxlen 50 //定义顺序表中元素个数最多有几个  typedef struct  {  elementtype data[maxlen]; //elementtype是元素的类型 依具体情况而定  int listlen; //便于时刻了解顺序表里元素的个数  }seqlist; //顺序表的名称 不妨为seqlist  声明顺序表类型变量：  seqlist L,L1;  如顺序表的每个结点占用len个内存单元，用location (ki)表示顺序表中第i个结点ki所占内存空间的第1个单元的地址。则有如下的关系:  location (ki+1) = location (ki) +len  location (ki) = location(k1) + (i-1)len  存储结构要体现数据的逻辑结构，顺序表的存储结构中，内存中物理地址相邻的结点一定具有顺序表中的逻辑关系。  顺序存储的线性表的特点：  ◆ 线性表的逻辑顺序与物理顺序一致;  ◆ 数据元素之间的关系是以元素在计算机内“物理位置相邻”来体现  本次实验的完整代码如下：   1. #include<cstdio> 2. #include<cstdlib> 3. #include<cstring> 4. #include<cmath> 5. #include<algorithm> 6. #include<iostream> 7. #define MaxSize 100 8. #define ElemType int 9. #define Status int 10. **using** **namespace** std; 11. //顺序表数据结构 12. **typedef** **struct** 13. { 14. ElemType data[MaxSize];//顺序表元素 15. **int** length;            //顺序表当前长度 16. }SqList; 17. //初始化顺序表函数，构造一个空的顺序表 18. Status InitList(SqList &L) 19. { 20. memset(L.data, 0, **sizeof**(L));//初始化数据为0 21. L.length = 0;                //初始化长度为0 22. **return** 0; 23. } 24. //创建顺序表函数 初始化前n个数据 25. **bool** CreatList(SqList &L, **int** n) 26. { 27. **if** (n<0 || n>MaxSize)**false**;//n非法 28. **for** (**int** i = 0; i<n; i++) 29. { 30. **int** temp; 31. scanf("%d", &L.data[i]); 32. L.length++; 33. } 34. **return** **true**; 35. } 36. //插入函数 位置i插入数据 i及之后元素后移  1=<i<=length+1 37. **bool** InsertList(SqList &L, **int** i, ElemType e) 38. { 39. **if** (i<1 || i>L.length + 1) //判断位置是否有效 40. { 41. printf("ERROR！\n"); 42. **return** **false**; 43. } 44. **if** (L.length >= MaxSize)//判断存储空间是否已满 45. { 46. printf("The current storage space is full！\n"); 47. **return** **false**; 48. } 49. **for** (**int** j = L.length; j >= i; j--)//位置i及之后元素后移 50. { 51. L.data[j] = L.data[j - 1]; 52. } 53. L.data[i - 1] = e; 54. L.length++; 55. **return** **true**; 56. } 57. //删除函数 删除位置i的元素 i之后的元素依次前移 58. **bool**  ListDelete(SqList &L, **int** i) 59. { 60. **if** (i<1 || i>L.length) 61. { 62. printf("ERROR！\n"); 63. **return** **false**; 64. } 65. **for** (**int** j = i; j <= L.length - 1; j++)//位置i之后元素依次前移覆盖 66. { 67. L.data[j - 1] = L.data[j]; 68. } 69. L.length--; 70. **return** **true**; 71. } 72. //查找函数 按位置从小到大查找第一个值等于e的元素 并返回位置 73. **int** LocateElem(SqList L, ElemType e) 74. { 75. **for** (**int** i = 0; i<L.length; i++)//从低位置查找 76. { 77. **if** (L.data[i] == e) 78. **return** i + 1; 79. } 80. **return** 0; 81. } 82. //输出功能函数 按位置从小到大输出顺序表所有元素 83. **void** PrintList(SqList L) 84. { 85. printf("Now all numbers : "); 86. **for** (**int** i = 0; i<L.length; i++) 87. { 88. printf("%d ", L.data[i]); 89. } 90. printf("\n"); 91. } 92. //创建顺序表函数 93. **void** Create(SqList &L) 94. { 95. **int** n; **bool** flag; 96. L.length = 0; 97. printf("Please enter the length of the order table to create (> 1) : "); 98. scanf("%d", &n); 99. printf("Please enter %d number(s) : ", n); 100. flag = CreatList(L, n); 101. **if** (flag) { 102. printf("OKay！\n"); 103. PrintList(L); 104. } 105. **else** printf("ERROR！\n"); 107. } 108. //插入功能函数 调用InsertList完成顺序表元素插入 调用PrintList函数显示插入成功后的结果 109. **void** Insert(SqList &L) 110. { 111. **int** place; ElemType e; **bool** flag; 112. printf("Please enter the location (starting from 1) and element to insert :\n"); 113. scanf("%d %d", &place, &e); 114. flag = InsertList(L, place, e); 115. **if** (flag) 116. { 117. printf("OKay！\n"); 118. PrintList(L); 119. } 120. } 121. //删除功能函数 调用ListDelete函数完成顺序表的删除 调用PrintList函数显示插入成功后的结果 122. **void** Delete(SqList &L) 123. { 124. **int** place; **bool** flag; 125. printf("Please enter the location to delete (starting from 1) : "); 126. scanf("%d", &place); 127. flag = ListDelete(L, place); 128. **if** (flag) 129. { 130. printf("OKay！\n"); 131. PrintList(L); 132. } 133. } 134. //查找功能函数 调用LocateElem查找元素 135. **void** Search(SqList L) 136. { 137. ElemType e; **int** flag; 138. printf("Please enter the value to find : "); 139. scanf("%d", &e); 140. flag = LocateElem(L, e); 141. **if** (flag) 142. { 143. printf("The index of the element is : %d\n", flag); 144. } 145. **else** 146. printf("NOT FOUND！\n"); 147. } 149. **int** main() 150. { 151. SqList L; **int** choice; 152. InitList(L); 153. Create(L); 154. Insert(L); 155. Delete(L); 156. Search(L); 157. **return** 0; 158. }   本次实验使用的测试数据如下：   1. //4 2. //1 5 3 2 3. //2 6 4. //1 5. //2   本次实验测试样例得到结果如下图所示： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 单链表基本操作和复杂操作

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 要求实现静态链表的基本操作及复杂操作，实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 单链表就是指一个结构体里面存着一个数据域和一个指针域，一个数据指向另一个数据，另一个数据再指向下一个数据，即数据和数据之间由指针决定链的走向。   1. typedef **struct** LNode 2. { 3. **int** data; 4. **struct** LNode \*next; 5. }LNode;   第一步，让单链表存在，即定义(或者将初始化/创建)单链表，在这里我把它定义成createlist函数：   1. LNode \*CreateList(**int** len) 2. { 3. LNode \*L = (LNode\*)malloc(**sizeof**(LNode)); 4. LNode \*temp = L; 5. temp->next = NULL; 6. **for**(**int** i=1;i<=len;i++) 7. { 8. LNode \*p = (LNode\*)malloc(**sizeof**(LNode)); 9. scanf("%d",&p->data); 10. temp->next = p; 11. temp = p; 12. } 13. temp->next = NULL; 15. **return** (LNode\*)L; 16. }   然后，实现打印表中数据的功能，即printlist函数：   1. **void** PrintfList(LNode \*L) 2. { 3. LNode \*temp = L; 4. **int** count = 0; 5. printf("List:\n"); 6. **while**(temp->next) 7. { 8. temp = temp->next; 9. printf("%d ,",temp->data); 10. count++; 11. } 12. printf("\n"); 13. }   接着，实现数据的查找/添加/删除/替换功能：   1. **int** Serch(LNode \*L, **int** elem) 2. { 3. LNode \*temp = L; 4. **int** pos = 0; 5. **int** i = 1; 6. **while**(temp->next) 7. { 8. temp = temp->next; 9. **if**(elem==temp->data) 10. { 11. pos = i; 12. printf("The index of this number is %d\n",pos); 13. **return** pos; 14. } 15. i++; 16. } 17. printf("Serch error!\n"); 19. **return** ERROR; 20. } 21. LNode \*Insert(LNode \*L, **int** pos, **int** elem) 22. { 23. LNode \*temp = L; 24. **int** i = 0; 25. **while**( (temp!=NULL)&&(i<pos-1) ) 26. { 27. temp = temp->next; 28. ++i; 29. } 30. **if**( (temp==NULL)||(i>pos-1) ) 31. { 32. printf("ERROR!\n"); 33. **return** (LNode\*)temp; 34. } 35. LNode \*newl = (LNode\*)malloc(**sizeof**(LNode)); 36. newl->data = elem; 37. newl->next = temp->next; 38. temp->next = newl; 40. **return** (LNode\*)L; 41. } 43. LNode \*Delete(LNode \*L, **int** pos, **int** \*elem) 44. { 45. LNode \*temp = L; 46. **int** i = 0; 47. **while**( (temp!=NULL)&&(i<pos-1) ) 48. { 49. temp = temp->next; 50. ++i; 51. } 52. **if**( (temp==NULL)||(i>pos-1) ) 53. { 54. printf("ERROR!\n"); 55. **return** (LNode\*)temp; 56. } 57. LNode \*del = temp->next; 58. \*elem = del->data; 59. temp->next = del->next; 60. free(del); 61. del = NULL; 63. **return** (LNode\*)L; 64. } 66. LNode \*Replace(LNode \*L, **int** pos, **int** elem) 67. { 68. LNode \*temp = L; 69. temp = temp->next; 70. **for**(**int** i=1;i<pos;i++) 71. { 72. temp = temp->next; 73. } 74. temp->data = elem; 75. **return** (LNode\*)L; 76. }   最后，我写了一个检测程序来检验各个函数的功能，具体实现的效果即代码如下：  1.查找--serch函数    2.代替--replace函数    3.插入--insert函数    4.删除--delete函数     1. #include <stdio.h> 2. #include <stdlib.h> 4. #define ERROR 1 5. #define OK    0 6. typedef **struct** LNode 7. { 8. **int** data; 9. **struct** LNode \*next; 10. }LNode; 12. LNode \*CreateList(**int** len) 13. { 14. LNode \*L = (LNode\*)malloc(**sizeof**(LNode)); 15. LNode \*temp = L; 16. temp->next = NULL; 17. **for**(**int** i=1;i<=len;i++) 18. { 19. LNode \*p = (LNode\*)malloc(**sizeof**(LNode)); 20. scanf("%d",&p->data); 21. temp->next = p; 22. temp = p; 23. } 24. temp->next = NULL; 26. **return** (LNode\*)L; 27. } 29. **int** Serch(LNode \*L, **int** elem) 30. { 31. LNode \*temp = L; 32. **int** pos = 0; 33. **int** i = 1; 34. **while**(temp->next) 35. { 36. temp = temp->next; 37. **if**(elem==temp->data) 38. { 39. pos = i; 40. printf("The index of this number is %d\n",pos); 41. **return** pos; 42. } 43. i++; 44. } 45. printf("Serch error!\n"); 47. **return** ERROR; 48. } 50. LNode \*Replace(LNode \*L, **int** pos, **int** elem) 51. { 52. LNode \*temp = L; 53. temp = temp->next; 54. **for**(**int** i=1;i<pos;i++) 55. { 56. temp = temp->next; 57. } 58. temp->data = elem; 59. **return** (LNode\*)L; 60. } 62. LNode \*Insert(LNode \*L, **int** pos, **int** elem) 63. { 64. LNode \*temp = L; 65. **int** i = 0; 66. **while**( (temp!=NULL)&&(i<pos-1) ) 67. { 68. temp = temp->next; 69. ++i; 70. } 71. **if**( (temp==NULL)||(i>pos-1) ) 72. { 73. printf("ERROR!\n"); 74. **return** (LNode\*)temp; 75. } 76. LNode \*newl = (LNode\*)malloc(**sizeof**(LNode)); 77. newl->data = elem; 78. newl->next = temp->next; 79. temp->next = newl; 81. **return** (LNode\*)L; 82. } 84. LNode \*Delete(LNode \*L, **int** pos, **int** \*elem) 85. { 86. LNode \*temp = L; 87. **int** i = 0; 88. **while**( (temp!=NULL)&&(i<pos-1) ) 89. { 90. temp = temp->next; 91. ++i; 92. } 93. **if**( (temp==NULL)||(i>pos-1) ) 94. { 95. printf("ERROR!\n"); 96. **return** (LNode\*)temp; 97. } 98. LNode \*del = temp->next; 99. \*elem = del->data; 100. temp->next = del->next; 101. free(del); 102. del = NULL; 104. **return** (LNode\*)L; 105. } 107. **void** PrintfList(LNode \*L) 108. { 109. LNode \*temp = L; 110. **int** count = 0; 111. printf("List:\n"); 112. **while**(temp->next) 113. { 114. temp = temp->next; 115. printf("%d ,",temp->data); 116. count++; 117. } 118. printf("\n"); 119. } 121. **int** MenuSelect(**void**) 122. { 123. **int** cmd; 124. printf("1.Serch test\n"); 125. printf("2.Replace test\n"); 126. printf("3.Insert test\n"); 127. printf("4.Delete test\n"); 128. **do** 129. { 130. printf("Enter your choice: "); 131. scanf("%d",&cmd); 132. }**while**(cmd<0||cmd>4); 133. **return** cmd; 134. } 136. **void** Test1(LNode \*L) 137. { 138. **int** serchElem = 0; 139. printf("--start--\n"); 140. PrintfList(L); 141. printf("Input the element you want to serch:"); 142. scanf("%d",&serchElem); 143. Serch(L,serchElem); 144. printf("--end--\n"); 145. printf("\n"); 146. } 148. **void** Test2(LNode \*L) 149. { 150. **int** replacePos = 0, replaceElem = 0; 151. printf("--start--\n"); 152. PrintfList(L); 153. printf("Input the position and the element you want to replace(example:10,33):"); 154. scanf("%d,%d",&replacePos,&replaceElem); 155. L = Replace(L,replacePos,replaceElem); 156. printf("After replace,list is:\n"); 157. PrintfList(L); 158. printf("--end--\n"); 159. printf("\n"); 160. } 162. **void** Test3(LNode \*L) 163. { 164. **int** insertPos = 0, insertElem = 0; 166. printf("--start--\n"); 167. PrintfList(L); 168. printf("Input the position and the element you want insert(example:10,33):"); 169. scanf("%d,%d",&insertPos,&insertElem); 170. L = Insert(L,insertPos,insertElem); 171. printf("After insert,list is:\n"); 172. PrintfList(L); 173. printf("--end--\n"); 174. printf("\n"); 175. } 177. **void** Test4(LNode \*L) 178. { 179. **int** deletePos = 0; 180. **int** elem = NULL; 182. printf("--start--\n"); 183. PrintfList(L); 184. printf("Input the position of the element you want to delete(example:10):"); 185. scanf("%d",&deletePos); 186. L = Delete(L,deletePos,&elem); 187. printf("After delete,list is:\n"); 188. PrintfList(L); 189. printf("--end--\n"); 190. printf("\n"); 191. } 193. **int** main(**void**) 194. { 195. **int** len = 0; 196. **int** cmd; 197. LNode \*L; 199. printf("Please input list length:"); 200. scanf("%d",&len); 201. L = CreateList(len); 202. PrintfList(L); 203. **while**(1) 204. { 205. cmd = MenuSelect(); 206. **switch**(cmd) 207. { 208. **case** 1: Test1(L);       **break**; 209. **case** 2: Test2(L);       **break**; 210. **case** 3: Test3(L);       **break**; 211. **case** 4: Test4(L);       **break**; 212. } 213. } 214. **return** 0; 215. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 静态链表基本操作和复杂操作

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 要求实现静态链表的基本操作及复杂操作，实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 静态链表也是一种用线性形式来存储数据的一种容器，但是和顺序表、单链表不一样的是，静态链表是用数组来代替指针，来描述单链表的，即将数组元素分成两个数据域：data和cur。其中，data用来存放数据元素，cur存放该元素的后继在数组中的下标。所以又叫单链表的游标实现法。   1. typedef int ElemType; 2. typedef struct \_staticList 3. { 4. ElemType data; 5. int cur； 6. }StaticList[MAXSIZE];   这个形式就是静态链表，每一个结构体都要存他的数据，还有放置下一个数据的下标位置，一个数组表示一共有 maxsize个数据。  然后给出静态链表要求实现的基本操作和复杂操作，如下：   1. //将结构体数组中所有分量链接到备用链表中 2. void reserveArr(component \*array); 3. //初始化静态链表 4. int initArr(component \*array); 5. //向链表中插入数据，body表示链表的头结点在数组中的位置，add表示插入元素的位置，a表示要插入的数据 6. void insertArr(component \* array,int body,int add,int a); 7. //删除链表中含有字符a的结点 8. void deletArr(component \* array,int body,int a); 9. //查找存储有字符elem的结点在数组的位置 10. int selectElem(component \* array,int body,int elem); 11. //将链表中的字符oldElem改为newElem 12. void amendElem(component \* array,int body,int oldElem,int newElem); 13. //输出函数 14. void displayArr(component \* array,int body); 15. //从备用链表中摘除空闲节点的实现函数 16. int mallocArr(component \* array); 17. //将摘除下来的节点链接到备用链表上 18. void freeArr(component \* array,int k);   实现要求的效果和具体代码如下：     1. #include <stdio.h> 2. #define maxSize 7 3. typedef struct { 4. int data; 5. int cur; 6. }component; 7. void reserveArr(component \*array); 8. int initArr(component \*array); 9. void insertArr(component \* array,int body,int add,int a); 10. void deletArr(component \* array,int body,int a); 11. int selectElem(component \* array,int body,int elem); 12. void amendElem(component \* array,int body,int oldElem,int newElem); 13. void displayArr(component \* array,int body); 14. int mallocArr(component \* array); 15. void freeArr(component \* array,int k); 16. int main() { 17. component array[maxSize]; 18. int body=initArr(array); 19. printf("Now list：\n"); 20. displayArr(array, body); 21. int insert\_index,insert\_num; 22. printf("Input the index & number tou want to insert:"); 23. scanf("%d %d",&insert\_index,&insert\_num); 24. printf("Insert %d to %d:",insert\_num,insert\_index); 25. insertArr(array,body,insert\_index,insert\_num); 26. displayArr(array,body); 27. printf("Input the number you want to delete:"); 28. int delete\_num; 29. scanf("%d",&delete\_num); 30. deletArr(array, body, delete\_num); 31. displayArr(array, body); 32. int select\_num; 33. printf("Input thie num you want to find the index:"); 34. scanf("%d",&select\_num); 35. printf("The index of this num is:"); 36. int selectAdd=selectElem(array,body,select\_num); 37. printf("%d\n",selectAdd); 38. printf("Input the num you want to change:"); 39. int change\_num; 40. scanf("%d",&change\_num); 41. amendElem(array,body,select\_num,change\_num); 42. displayArr(array, body); 43. return 0; 44. } 45. void reserveArr(component \*array){ 46. for (int i = 0;i < maxSize;i++) { 47. array[i].cur=i+1; 48. } 49. array[maxSize-1].cur=0; 50. } 51. int initArr(component \*array,int nn){ 52. reserveArr(array); 53. int body = mallocArr(array); 54. int tempBody = body; 55. for (int i = 1;i < nn;i++) { 56. int j=mallocArr(array); 57. array[tempBody].cur=j; 58. array[j].data=i-1; 59. tempBody=j; 60. } 61. array[tempBody].cur=0; 62. return body; 63. } 64. void insertArr(component \* array,int body,int add,int a){ 65. int tempBody=body; 66. for (int i = 1;i < add;i++) { 67. tempBody=array[tempBody].cur; 68. } 69. int insert=mallocArr(array); 70. array[insert].cur=array[tempBody].cur; 71. array[insert].data=a; 72. array[tempBody].cur=insert; 73. } 74. void deletArr(component \* array,int body,int a){ 75. int tempBody = body; 76. while (array[tempBody].data != a) { 77. tempBody = array[tempBody].cur; 78. if (tempBody == 0) { 79. printf("NOT FOUND!\n"); 80. return; 81. } 82. } 83. int del=tempBody; 84. tempBody=body; 85. while (array[tempBody].cur!=del) { 86. tempBody=array[tempBody].cur; 87. } 88. array[tempBody].cur=array[del].cur; 89. freeArr(array, del); 90. } 91. int selectElem(component \* array,int body,int elem){ 92. int tempBody=body; 93. while (array[tempBody].cur!=0) { 94. if (array[tempBody].data==elem) { 95. return tempBody; 96. } 97. tempBody=array[tempBody].cur; 98. } 99. return -1; 100. } 101. void amendElem(component \* array,int body,int oldElem,int newElem){ 102. int add=selectElem(array, body, oldElem); 103. if (add==-1) { 104. printf("WITHOUT THIS NUMBER!\n"); 105. return; 106. } 107. array[add].data=newElem; 108. } 109. void displayArr(component \* array,int body){ 110. int tempBody=array[body].cur; 111. while (array[tempBody].cur) { 112. //      printf("%d,%d  |",array[tempBody].data,array[tempBody].cur); 113. printf("%d,",array[tempBody].data); 114. tempBody=array[tempBody].cur; 115. } 116. printf("%d\n",array[tempBody].data); 117. //    printf("%d,%d\n",array[tempBody].data,array[tempBody].cur); 118. } 119. int mallocArr(component \* array){ 120. int i=array[0].cur; 121. if (array[0].cur) { 122. array[0].cur=array[i].cur; 123. } 124. return i; 125. } 126. void freeArr(component \* array,int k){ 127. array[k].cur=array[0].cur; 128. array[0].cur=k; 129. }   另外，老师上课要求实现两个链表存在绝对包含关系的判断功能。具体代码如下：   1. #include <stdio.h> 2. #define maxSize 7 3. typedef **struct** { 4. **int** data; 5. **int** cur; 6. }component; 8. **int** a[100],b[100]; 10. **void** reserveArr(component \*array); 11. **int** initArr(component \*array,**int** nn); 12. **void** insertArr(component \* array,**int** body,**int** add,**int** a); 13. **void** deletArr(component \* array,**int** body,**int** a); 14. **int** selectElem(component \* array,**int** body,**int** elem); 15. **void** amendElem(component \* array,**int** body,**int** oldElem,**int** newElem); 16. **void** displayArr(component \* array,**int** body); 17. **int** mallocArr(component \* array); 18. **void** freeArr(component \* array,**int** k); 19. **int** flagArr(component \*a1,component \*a2,**int** body1,**int** body2); 20. **int** main() { 21. component a1[maxSize],a2[10]; 22. **int** b1=initArr(a1,maxSize),b2=initArr(a2,10); 23. displayArr(a1,b1); 24. displayArr(a2,b2); 26. **int** x = flagArr(a1,a2,b1,b2); 27. printf("%d\n",x); 28. **if**(x==-1) printf("Non-conformity!\n"); 29. **else** printf("The qualified array subscript is:%d\n",x); 31. **return** 0; 32. } 34. **void** reserveArr(component \*array){ 35. **for** (**int** i = 0;i < maxSize;i++) { 36. array[i].cur=i+1; 37. } 38. array[maxSize-1].cur=0; 39. } 41. **int** initArr(component \*array,**int** nn){ 42. reserveArr(array); 43. **int** body = mallocArr(array); 44. **int** tempBody = body; 45. **for** (**int** i = 1;i < nn;i++) { 46. **int** j=mallocArr(array); 47. array[tempBody].cur=j; 48. array[j].data=i-1; 49. tempBody=j; 50. } 51. array[tempBody].cur=0; 52. **return** body; 53. } 55. **void** insertArr(component \* array,**int** body,**int** add,**int** a){ 56. **int** tempBody=body; 57. **for** (**int** i = 1;i < add;i++) { 58. tempBody=array[tempBody].cur; 59. } 60. **int** insert=mallocArr(array); 61. array[insert].cur=array[tempBody].cur; 62. array[insert].data=a; 63. array[tempBody].cur=insert; 64. } 66. **void** deletArr(component \* array,**int** body,**int** a){ 67. **int** tempBody = body; 68. **while** (array[tempBody].data != a) { 69. tempBody = array[tempBody].cur; 70. **if** (tempBody == 0) { 71. printf("NOT FOUND!\n"); 72. **return**; 73. } 74. } 75. **int** del=tempBody; 76. tempBody=body; 77. **while** (array[tempBody].cur!=del) { 78. tempBody=array[tempBody].cur; 79. } 80. array[tempBody].cur=array[del].cur; 82. freeArr(array, del); 83. } 85. **int** selectElem(component \* array,**int** body,**int** elem){ 86. **int** tempBody=body; 87. **while** (array[tempBody].cur!=0) { 88. **if** (array[tempBody].data==elem) { 89. **return** tempBody; 90. } 91. tempBody=array[tempBody].cur; 92. } 93. **return** -1; 94. } 96. **void** amendElem(component \* array,**int** body,**int** oldElem,**int** newElem){ 97. **int** add=selectElem(array, body, oldElem); 98. **if** (add==-1) { 99. printf("WITHOUT THIS NUMBER!\n"); 100. **return**; 101. } 102. array[add].data=newElem; 103. } 105. **void** displayArr(component \* array,**int** body){ 106. **int** tempBody=array[body].cur; 107. **while** (array[tempBody].cur) { 108. printf("%d,",array[tempBody].data); 109. tempBody=array[tempBody].cur; 110. } 111. printf("%d\n",array[tempBody].data); 112. } 114. **int** mallocArr(component \* array){ 115. **int** i=array[0].cur; 116. **if** (array[0].cur) { 117. array[0].cur=array[i].cur; 118. } 119. **return** i; 120. } 122. **void** freeArr(component \* array,**int** k){ 123. array[k].cur=array[0].cur; 124. array[0].cur=k; 125. } 127. **int** flagArr(component \*a1,component \*a2,**int** body1,**int** body2) 128. { 129. **int** i1 = 0,i2 = 0; 130. **int** t1=a1[body1].cur,t2=a2[body2].cur; 131. **while**(a1[t1].cur) 132. { 133. a[i1++]=a1[t1].data; 134. t1=a1[t1].cur; 135. } 136. a[i1++]=a1[t1].data; 137. **while**(a2[t2].cur) 138. { 139. b[i2++]=a2[t2].data; 140. t2=a2[t2].cur; 141. } 142. b[i2++]=a2[t2].data; 144. **int** flag=1,findex=-1; 146. **if**(i1>i2) 147. { 148. **for**(**int** i = 0;i < i1;i++) 149. { 150. **int** j; 151. **for**(j = 0;j < i2;j++) 152. { 153. **if**(b[j] != a[i]) flag=0;**break**; 154. } 155. **if**(j == i2) flag=1; 156. **if**(flag != 0) findex = i;**break**; 157. } 158. } 159. **else** 160. { 161. **for**(**int** i = 0;i < i2;i++) 162. { 163. **int** j; 164. **for**(j = 0;j < i1;j++) 165. { 166. **if**(a[j] != b[i]) flag=0;**break**; 167. } 168. **if**(j == i1)flag=1; 169. **if**(flag != 0) findex = i;**break**; 170. } 171. } 172. **return** findex; 173. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 栈的基本操作和四则混合运算

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 这次实验要求实现栈的基本操作及四则混合运算操作，实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 栈是特殊的线性表，栈中数据具有前驱后继的关系，栈的一个重要规则是后进先出，即只能在表尾（栈顶）进行插入和删除数据的操作，和线性表一样，实现栈这个数据结构也有两种方法，顺序栈和链栈。这里主要使用链栈实现实验的要求功能。栈内每个数据都有它的data域和next域即数据域和指针域。    栈的主要操作包括定义和初始化栈、数据的进栈和出栈等，在本次实验中，我们利用栈的主要操作实现四则运算功能。首先，，我们需要把原来的中缀表达式进行转化，将其变为后缀表达式，然后再利用栈对其进行运算得出结果。从左到右遍历中缀表达式的每个数字和符号，若是数字就输出，即成为后缀表达式的一部分；若是符号，则判断其与栈顶符号的优先级，是右括号或优先级低于找顶符号（乘除优先加减）则栈顶元素依次出找并输出，并将当前符号进栈，一直到最终输出后缀表达式为止。在得到后缀表达式之后，我们需要。从左到右扫描后缀表达式，如果遇到操作数，将其压入栈中，如果遇到操作符，则从栈中弹出两个操作数，计算结果，然后把结果入栈，直到遍历完后缀表达式，则计算完成，此时的栈顶元素即为计算结果。  在具体的实现过程中，还需要对用户输入的表达式进行异常判断，实现要求的效果的具体代码如下：       1. #include <stdio.h> 2. #include <stdlib.h> 4. #define OK     1 5. #define ERROR  -1 6. **struct** node 7. { 8. **int** data; 9. **struct** node \*next; 10. }; 11. **typedef** **struct** node Node; 13. **struct** stack 14. { 15. Node \*top; 16. **int** count; 17. }; 18. **typedef** **struct** stack Stack; 20. **int** InitStack(Stack \*l) 21. { 22. l->top = NULL; 23. l->count = 0; 24. **return** OK; 25. } 27. **int** EmptyStack(Stack \*s) 28. { 29. **return** (s->count == 0) ? OK : ERROR; 30. } 32. **int** Push(Stack \*s, **int** e) 33. { 34. Node \*p = (Node \*)malloc(**sizeof**(Node)); 35. **if**(p == NULL) 36. { 37. **return** ERROR; 38. } 39. p->data = e; 40. p->next = s->top; 41. s->top = p; 42. s->count++; 43. **return** OK; 44. } 46. **int** GetTop(Stack \*s) 47. { 48. **if**(NULL == s->top) 49. { 50. **return** ERROR; 51. } 52. **return** (s->top->data); 53. } 55. **int** Priority(**char** s) 56. { 57. **switch**(s) 58. { 59. **case** '(': 60. **return** 3; 61. **case** '\*': 62. **case** '/': 63. **return** 2; 64. **case** '+': 65. **case** '-': 66. **return** 1; 67. **default**: 68. **return** 0; 69. } 70. } 72. **int** Pop(Stack \*s) 73. { 74. **int** e; 75. **if**(NULL == s->top) 76. { 77. **return** ERROR; 78. } 79. Node \*p = s->top; 80. e = p->data; 81. s->top = p->next; 82. free(p); 83. s->count--; 85. **return** e; 86. } 88. **int** main() 89. { 90. Stack num, opt; 91. **char** str[100]={0}; 92. **int** i = 0, tmp = 0, j; 94. **if**(InitStack(&num) != OK || InitStack(&opt) != OK) 95. { 96. printf("Init Failure!\n"); 97. **return** 0; 98. } 99. printf("Please Input Operator :\n"); 100. scanf("%s",str); 101. **while**(str[i] != '\0' || EmptyStack(&opt) != OK) 102. { 103. **if**(str[i] >= '0' && str[i] <= '9') 104. { 105. tmp = tmp \* 10 + str[i] - '0'; 106. i++; 107. **if**(str[i] < '0' || str[i] > '9') 108. { 109. Push(&num,tmp); 110. tmp = 0; 111. } 112. } 113. **else** 114. { 115. **if**((EmptyStack(&opt) == OK) || (GetTop(&opt) == '(' && str[i] != ')') || Priority(str[i]) > Priority(GetTop(&opt)))//进栈不参与运算 116. { 117. Push(&opt,str[i]); 118. i++; 119. **continue**; 120. } 121. **if**(GetTop(&opt) == '(' && str[i] == ')') 122. { 123. Pop(&opt); 124. i++; 125. **continue**; 126. } 127. **if**((str[i] == '\0' && EmptyStack(&opt) != OK) || (str[i] == ')' && GetTop(&opt) != '(') || Priority(str[i]) <= Priority(GetTop(&opt)))//出栈并参与运算 128. { 129. **switch**(Pop(&opt)) 130. { 131. **case** '+': 132. Push(&num, Pop(&num) + Pop(&num)); 133. **break**; 134. **case** '-': 135. j = Pop(&num); 136. Push(&num, Pop(&num) - j); 137. **break**; 138. **case** '\*': 139. Push(&num, Pop(&num) \* Pop(&num)); 140. **break**; 141. **case** '/': 142. j = Pop(&num); 143. Push(&num, Pop(&num) / j); 144. } 145. **continue**; 146. } 147. } 148. } 149. printf("%d\n",Pop(&num)); 150. **return** 0; 151. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 循环队列应用 - 迷宫游戏

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用循环队列的数据结构以及创建队列、数据入队、数据出队等基本操作完成迷宫游戏的应用场景。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 为充分利用向量空间，克服"假溢出"现象的方法是：将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列（Circular Queue）。循环队列是把顺序队列首尾相连，把存储队列元素的表从逻辑上看成一个环，成为循环队列。  循环队列就是将队列存储空间的最后一个位置绕到第一个位置，形成逻辑上的环状空间，供队列循环使用。在循环队列结构中，当存储空间的最后一个位置已被使用而再要进入队运算时，只需要存储空间的第一个位置空闲，便可将元素加入到第一个位置，即将存储空间的第一个位置作为队尾。循环队列可以更简单防止伪溢出的发生，但队列大小是固定的。  在循环队列中，当队列为空时，有front=rear，而当所有队列空间全占满时，也有front=rear。为了区别这两种情况，规定循环队列最多只能有MaxSize-1个队列元素，当循环队列中只剩下一个空存储单元时，队列就已经满了。因此，队列判空的条件是front=rear，而队列判满的条件是front=（rear+1)%MaxSize。  本次实验中，我将利用循环队列的基本操作实现迷宫游戏应用。  本次实验的全部代码如下：   1. #include <stdio.h> 2. #include<stdlib.h> 3. #include<time.h> 4. #define SIZE 65535 5. **typedef** **struct** qstru{ 6. **int** x; 7. **int** y; 8. **int** pre; 9. }QElemType; 11. **int** flag=0; 13. **typedef** **int** Status; 15. **typedef** **struct** queue{ 16. QElemType \*base; 17. **int** front; //指示队头位置 18. **int** rear;  //指示队尾位置 19. }SqQueue; 21. **void** initQueue(SqQueue \*q){ 22. q->base = (QElemType\*)malloc(SIZE\***sizeof**(QElemType)); 23. q->front=q->rear=0; 24. } 26. **void** enQueue(SqQueue \*q,QElemType e){ 27. q->base[q->rear]=e; 28. q->rear++; 29. } 31. **void** deQueue(SqQueue \*q,QElemType \*e){ 32. \*e=q->base[q->front]; 33. q->front++; 34. } 36. **void** printq(SqQueue q,**int** pos){//输出迷宫路径 37. **if**(q.base[pos].pre!=-1) 38. { 39. flag=1; 40. printq(q,q.base[pos].pre); 41. flag=0; 42. } 44. **if**(flag==1)printf("( %d, %d)",q.base[pos].x,q.base[pos].y); 45. **else** printf("-->( %d, %d)",q.base[pos].x,q.base[pos].y); 46. }  49. **int** \*\*randomMap(**int** m,**int** n){//生成随机迷宫 50. srand((**int**)time(NULL)); 51. **int** \*\*a,i,j; 52. a=(**int** \*\*)malloc(m\***sizeof**(**int** \*)); 53. **for**( i=0;i<m;i++){ 54. a[i]=(**int** \*)malloc(n\***sizeof**(**int**)); 55. } 57. **for**( i=0;i<m;i++){ 58. **for**( j=0;j<n;j++){ 59. **if**(i==0||i==m-1||j==0||j==n-1) 60. a[i][j]=1; 61. **else** 62. a[i][j]=rand()%2; 63. } 65. } 66. a[1][1]=0,a[m-2][n-2]=0; 67. **return** a; 68. } 69. **void** printMap(**int** \*\*m,**int** a,**int** b){//输出迷宫地图 70. **for**(**int** i=0;i<a;i++){ 71. **for**(**int** j=0;j<b;j++){ 72. printf("%d ",m[i][j]); 73. } 74. printf("\n"); 75. } 76. } 77. **int** main(){ 78. SqQueue Q; 79. **int** a,b,i,j; 80. printf("输入迷宫的长和宽:"); 81. scanf("%d %d",&a,&b); 82. **int** \*\*m=randomMap(a+2,b+2); 83. printMap(m,a+2,b+2);  86. **int** delta[8][2]={{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1}, {-1,0},{-1,1}};  89. **int** end\_x=a,end\_y=b; 90. initQueue(&Q); 91. QElemType e,e1; 92. e.pre=-1;   e.x=1;  e.y=1; 94. enQueue(&Q,e); 96. **while**(!(Q.front==Q.rear)){ 97. **int** i,x,y; 98. deQueue(&Q,&e); 99. **for**(i=0;i<8;i++){ 100. x=e.x+delta[i][0]; 101. y=e.y+delta[i][1]; 103. **if**(m[x][y]==0){//如果可以走 104. //形成新元素入队，标记走过-->2 105. e1.x=x; e1.y=y; e1.pre=Q.front-1; 106. enQueue(&Q,e1); 107. m[x][y]=2; 108. **if**(x==end\_x&&y==end\_y){ 109. printf("路径：\n"); 110. printq(Q,Q.rear-1);**return** 0; 111. } 112. } 114. } 115. } 116. printf("无法通过!"); 117. **return** 0; 118. }   本次实验采用的测试样例如下：   1. //8 9   （注：本次实验随机生成迷宫节点，所以实验结果可能有所不同）  本次实验的测试结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 树的孩子表现法

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用散列的除留余数法和链接地址法处理散列冲突。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 树的存储结构，孩子表示法：把每个结点的孩子结点排列起来，以单链表作存储结构，则n个结点有n个孩子链表，如果是叶子结点则此单链表为空。然后n个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中。  孩子表示法存储普通树采用的是 "顺序表+链表" 的组合结构，其存储过程是：从树的根节点开始，使用顺序表依次存储树中各个节点，需要注意的是，与双亲表示法不同，孩子表示法会给各个节点配备一个链表，用于存储各节点的孩子节点位于顺序表中的位置。  为此，设计两种结点结构，一个是孩子链表的孩子结点    ||child | next|| 另一个是表头数组的表头节点 ||data | firstchild||  #define MAXSIZE 100  typedef struct CTNode  /\*孩子结点\*/  {      int child;      struct CTNode \*next;  }ChildPtr;  typedef struct    /\*表头结构\*/  {      int data;      ChildPtr firstchild;  }CTBox;  typedef struct    /\*树结构\*/  {      CTBox nodes[MAXSIZE];      int r,n;  }CTree;  本次实验的全部代码如下：   1. #include<iostream> 2. #include <cstdio> 3. #include <cstdlib> 4. #include <cstring> 5. #define OK 1 6. #define ERROR 0 7. #define FALSE 0 8. #define TRUE 1 9. #define OVERFLOW -2 10. **using** **namespace** std; 11. **typedef** **int** Status; 12. **typedef** **char** ElemType; //结点的值设置为字符 13. **typedef** **struct** CSNode 14. { 15. ElemType data; 16. **struct** CSNode \*firstChild;  //第一个孩子 17. **struct** CSNode \*nextsbling;   //该孩子的第一个兄弟 18. }CSNode, \*CSTree; 20. **typedef** CSTree QElemType; 21. **typedef** **struct** QNode 22. { 23. QElemType data; 24. **struct** QNode \*next; 25. }QNode,\*QueuePtr; 27. **typedef** **struct** 28. { 29. QueuePtr front;  //队头指针 30. QueuePtr rear;  //队尾指针 31. }LinkQueue; 33. Status InitQueue(LinkQueue &Q)//构造一个空队列 34. { 35. Q.front = Q.rear = (QueuePtr)malloc(**sizeof**(QNode));//队头结点 36. **if**(!Q.front) 37. exit(OVERFLOW); 38. Q.front ->next = NULL; 39. **return** OK; 40. } 42. Status QueueEmpty(**const** LinkQueue &Q)//若队列为空，则返回TRUE，否则返回FALSE 43. { 44. **if**(Q.rear == Q.front) 45. **return** TRUE; 46. **return** FALSE; 47. } 49. Status EnQueue(LinkQueue &Q, QElemType e) //插入元素e为Q的新队尾元素 50. { 51. QueuePtr p = (QueuePtr)malloc(**sizeof**(QNode)); 52. **if**(!p) 53. exit(OVERFLOW); 54. p->data = e; 55. p->next = NULL; 56. Q.rear->next = p; 57. Q.rear = p; 58. **return** OK; 59. } 61. Status DeQueue(LinkQueue &Q,QElemType &e) //若队列不空，则删除Q的队头元素，用e返回其值，并返回OK; 62. { 63. **if**(Q.front == Q.rear) 64. { 65. **return** ERROR; //队空 66. } 67. QueuePtr p = Q.front->next; 68. e = p->data; 69. Q.front->next = p->next; 70. **if**(Q.rear == p) 71. Q.rear = Q.front; 72. free(p); 73. **return** OK; 74. } 76. Status CreateTree(CSTree &T)  //创建一棵树 77. { 78. LinkQueue Q; 79. InitQueue(Q);//构造一个空队列 80. **char** buffChild[20];  //用于存放孩子们的缓存 81. memset(buffChild,0,20); //初始化缓存数组，置为NULL 82. printf("请输入树的根结点（字符,以#代表空）:\n"); 83. scanf("%c",&buffChild[0]); 84. **if**(buffChild[0] != '#') 85. { 86. T = (CSTree)malloc(**sizeof**(CSNode));//为根结点开辟一个空间 87. **if**(!T) 88. exit(OVERFLOW);  //开辟失败，终止程序 89. T->data = buffChild[0]; 90. T->nextsbling = NULL;  //根结点无兄弟 91. EnQueue(Q,T);  //根结点入队 92. **while**(!QueueEmpty(Q)) 93. { 94. QElemType e; 95. DeQueue(Q,e); //结点出队 96. //CSTree p = e; //用以指向队头结点 97. printf("请按长幼顺序输入结点%c的孩子(输入的字符串以#结束):\n",e->data); 98. scanf("%s",buffChild); 99. **if**(buffChild[0] != '#')//有孩子 100. { 101. CSTree q; 102. q = (CSTree)malloc(**sizeof**(CSNode));  //开辟孩子结点空间 103. **if**(!q) 104. exit(OVERFLOW); 105. q->data = buffChild[0];  // 106. e->firstChild = q;  //指向第一个孩子 107. EnQueue(Q,q);  //第一个孩子入队 108. CSTree p = q;  //指向刚入队的孩子 109. **for**(**size\_t** i = 1; i < strlen(buffChild)-1; ++i) //孩子存在兄弟 110. { 111. q = (CSTree)malloc(**sizeof**(CSNode));  //开辟孩子结点空间 112. **if**(!q) 113. exit(OVERFLOW); 114. q->data = buffChild[i]; 115. p->nextsbling = q; 116. EnQueue(Q,q); 117. p = q;                     //指向刚入队的孩子 118. } 119. p->nextsbling = NULL; 120. } 121. **else**//无孩子 122. { 123. e->firstChild = NULL; 124. } 125. } 126. } 127. **else** 128. { 129. T = NULL;//空树 130. } 131. **return** OK; 132. } 134. **void** DestroyTree(CSTree &T) 135. { 136. **if**(T) 137. { 138. **if**(T->firstChild)    //左子树存在，即销毁以长子为结点的子树 139. DestroyTree(T->firstChild); 140. **if**(T->nextsbling)    //右子树存在，即销毁以兄弟为结点的子树 141. DestroyTree(T->nextsbling); 142. free(T); 143. T = NULL; 144. } 145. } 147. **void** ClearTree(CSTree &T) 148. { 149. DestroyTree(T); //树T存在，将树T清为空树, 150. } 152. Status TreeEmpty(**const** CSTree &T) 153. { 154. **if**(T)   //树T存在，空树返回TRUE，否则返回FALSE 155. **return** TRUE; 156. **else** 157. **return** FALSE; 158. } 160. **int** TreeDepth(**const** CSTree &T) 161. { 162. //树T存在，返回树的深度 163. **if**(!T)//树空 164. { 165. **return** 0; 166. } 167. **if**(!T->firstChild)//无长子 168. { 169. **return** 1; 170. } 171. CSTree p; 172. **int** depth,max = 0; 173. **for**(p = T->firstChild; p; ) 174. { 175. depth = TreeDepth(p); 176. **if**(depth > max) 177. max = depth; 178. p = p->nextsbling; 179. } 180. **return** max+1;//当前层的下一层 181. } 183. ElemType Root(**const** CSTree &T) 184. { 185. //树T存在，返回树的根 186. **if**(T) 187. **return** T->data; 188. **return** 0; 189. } 191. CSNode \*FindNode(**const** CSTree &T,ElemType cur\_e) 192. { 193. //树T存在，返回值为cur\_e的结点的指针 194. LinkQueue Q; 195. InitQueue(Q);  //构造一个空队列 196. **if**(T) 197. { 198. EnQueue(Q,T);//树根入队 199. **while**(!QueueEmpty(Q)) 200. { 201. QElemType e; 202. DeQueue(Q,e); 203. **if**(e->data == cur\_e) 204. **return** e; 205. **if**(e->firstChild) //当前结点有长子，则该长子入队 206. { 207. EnQueue(Q,e->firstChild); 208. } 209. **if**(e->nextsbling)//当前结点有兄弟,则该兄弟入队 210. { 211. EnQueue(Q,e->nextsbling); 212. } 213. } 214. } 215. **return** NULL; 216. } 218. CSNode \*Parent(CSTree &T,ElemType cur\_e) 219. { 220. //初始条件：树T存在，cur\_e是T中某个结点 221. //操作结果：若cur\_e是T的非根结点，则返回它的双亲，否则返回空 222. LinkQueue Q; 223. InitQueue(Q); 224. **if**(T) 225. { 226. **if**(T->data == cur\_e) 227. **return** NULL;//根结点无双亲,结束，返回NULL 228. EnQueue(Q,T);//根结点入队 229. **while**(!QueueEmpty(Q)) 230. { 231. QElemType e; 232. DeQueue(Q,e); 233. QElemType p = e;//提示刚出队的元素； 234. **if**(e->firstChild)//该结点有孩子 235. { 236. **if**(e->firstChild->data == cur\_e)//或该孩子是所求的结点，则返回双亲 237. { 238. **return** p; 239. } 240. EnQueue(Q,e->firstChild); 241. QElemType brotherPtr = e->firstChild->nextsbling;//指向孩子的兄弟结点 242. **while**(brotherPtr) //该孩子有兄弟 243. { 244. **if**(brotherPtr->data == cur\_e)//兄弟是所求的结点，则返回双亲 245. { 246. **return** p; 247. } 248. EnQueue(Q,brotherPtr);//兄弟入队 249. brotherPtr = brotherPtr->nextsbling; 250. } 251. } 252. } 253. } 254. **return** NULL; 255. } 257. ElemType LeftChild(CSTree &T, ElemType cur\_e) 258. { 259. //初始条件：树T存在，cur\_e是T中某个结点 260. //操作结果：若cur\_e是T的非叶子结点，则返回它的最左孩子，否则返回空 261. CSNode \*node; 262. node = FindNode(T,cur\_e); 263. **if**(node) 264. { 265. **if**(node->firstChild)//非叶子结点 266. { 267. **return** node->firstChild->data; //返回结点的值 268. } 269. } 270. **return** NULL; 271. } 273. ElemType RightSibling(CSTree &T, ElemType cur\_e) 274. { 275. //初始条件：树T存在，cur\_e是T中的某个结点。 276. //操作结果：若cur\_e有右兄弟，则返回它的右兄弟，否则返回空 277. CSNode \*node; 278. node = FindNode(T,cur\_e); 279. **if**(node) 280. { 281. **if**(node->nextsbling)//有右兄弟 282. { 283. **return** node->nextsbling->data;//返回右兄弟的值 284. } 285. } 286. **return** NULL; 287. } 289. Status LevelOrderTraverse(**const** CSTree &T) 290. { 291. //层序遍历树 292. LinkQueue Q; 293. InitQueue(Q); 294. **if**(T) 295. { 296. printf("%c ",T->data);     //访问结点 297. EnQueue(Q,T);             //根结点排队 298. **while**(!QueueEmpty(Q)) 299. { 300. QElemType e,p; 301. DeQueue(Q,e); 302. p = e->firstChild; 303. **while**(p) 304. { 305. printf("%c ",p->data); 306. EnQueue(Q,p); 307. p = p->nextsbling; 308. } 309. } 310. **return** OK; 311. } 312. **return** ERROR; 313. } 315. **int** main() 316. { 317. CSTree T; 318. CreateTree(T); 320. printf("按层序遍历该树："); 321. LevelOrderTraverse(T); 322. printf("\n"); 324. printf("树的根为:  %c\n",Root(T)); 325. printf("输入要查询的节点:\n"); 326. ElemType e ; 327. cin>>e; 328. CSNode \*node = FindNode(T,e); 329. **if**(node){ 330. printf("存在结点:  %c\n",node->data); 331. } 332. printf("树的深度为： %d\n",TreeDepth(T)); 333. node = Parent(T,e); 334. **if**(node) 335. { 336. printf("结点%c的双亲是:  %c\n",e,node->data); 337. } 338. **else**{ 339. printf("查询的节点不存在,或节点为根节点\n"); 340. } 341. DestroyTree(T); 342. **return** 0; 343. }   本次实验使用的测试样例如下：   1. //E 2. //A C D 3. //B F 4. //H 5. //# 6. //F   本次实验的测试结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 二叉树的四种遍历方式

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求实现二叉树的四种遍历方式。四种遍历方式分别是层序遍历、前序遍历、中序遍历和后序遍历。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 在计算机科学中，二叉树是每个结点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。  一棵深度为k，且有2^k-1个结点的二叉树，称为满二叉树。这种树的特点是每一层上的结点数都是最大结点数。而在一棵二叉树中，除最后一层外，若其余层都是满的，并且或者最后一层是满的，或者是在右边缺少连续若干结点，则此二叉树为完全二叉树。具有n个结点的完全二叉树的深度为floor(log2n)+1。深度为k的完全二叉树，至少有2k-1个叶子结点，至多有2k-1个结点。  遍历是对树的一种最基本的运算，所谓遍历二叉树，就是按一定的规则和顺序走遍二叉树的所有结点，使每一个结点都被访问一次，而且只被访问一次。由于二叉树是非线性结构，因此，[树的遍历](https://baike.baidu.com/item/%E6%A0%91%E7%9A%84%E9%81%8D%E5%8E%86)实质上是将二叉树的各个结点转换成为一个线性序列来表示。  设L、D、R分别表示遍历左子树、访问根结点和遍历右子树， 则对一棵二叉树的遍历有三种情况：DLR（称为先根次序遍历），LDR（称为中根次序遍历），LRD （称为后根次序遍历）。  a.先序遍历  首先访问根，再先序遍历左（右）子树，最后先序遍历右（左）子树，C语言代码如下：   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | void XXBL(tree \*root){      //DoSomethingwithroot      if(root->lchild!=NULL)          XXBL(root->lchild);      if(root->rchild!=NULL)          XXBL(root->rchild);  } |   b.中序遍历  首先中序遍历左（右）子树，再访问根，最后中序遍历右（左）子树，C语言代码如下   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | void ZXBL(tree \*root)  {      if(root->lchild!=NULL)          ZXBL(root->lchild);          //Do something with root      if(root->rchild!=NULL)          ZXBL(root->rchild);  } |   c.后序遍历  首先后序遍历左（右）子树，再后序遍历右（左）子树，最后访问根，C语言代码如下   |  |  | | --- | --- | | 1  2  3  4  5  6  7 | void HXBL(tree \*root){      if(root->lchild!=NULL)          HXBL(root->lchild);      if(root->rchild!=NULL)          HXBL(root->rchild);          //Do something with root  } |   d.层次遍历  即按照层次访问，通常用[队列](https://baike.baidu.com/item/%E9%98%9F%E5%88%97" \t "_blank)来做。访问根，访问子女，再访问子女的子女（越往后的层次越低）（两个子女的级别相同）  本次实验的全部代码如下：   1. #include <iostream> 2. #include<stack> 3. #include<queue> 5. **using** **namespace** std; 7. **typedef** **struct** Node 8. { 9. **char** data; 10. **struct** Node \*lchild, \*rchild; 11. }Node; 13. //通过先序的方式创建树，#表示空节点 14. /\* 15. A 16. B      C 17. D   E   F  # 18. #  # # # #  # 19. 创建上面的树应输入应为  ABD##E##CF### 20. 前序遍历：ABDECF 21. 中序遍历：DBEAFC 22. 后序遍历：DEBFCA 23. 层次遍历：ABCDEF 24. \*/ 25. **void** creatTree(Node\* &root) 26. { 27. **char** data; 28. cin >> data; 29. **if** (data == '#') 30. root = NULL; 31. **else** 32. { 33. root = **new** Node; 34. root->data = data; 35. creatTree(root->lchild); 36. creatTree(root->rchild); 37. } 39. }  42. //打印一个节点的数据 43. **void** visit(Node\* node) 44. { 45. **if**(node!=NULL) 46. cout << node->data; 47. } 49. //递归-前序遍历，先访问跟节点，然后访问左节点，最后访问右节点，每一个节点都要准守这样的规则 50. **void** preTraversal(Node\* root) 51. { 52. //访问跟节点 53. **if** (root != NULL) 54. { 55. visit(root); 56. preTraversal(root->lchild); 57. preTraversal(root->rchild); 58. } 60. } 62. //递归-中序遍历，先访问跟左节点，然后访问中节点，最后访问右节点，每一个节点都要准守这样的规则 63. **void** midTraversal(Node\* root) 64. { 65. **if** (root != NULL) 66. { 67. midTraversal(root->lchild); 68. visit(root); 69. midTraversal(root->rchild); 70. } 71. } 73. //递归-后序遍历，先访问左节点，然后访问右节点，最后访问根节点，每一个节点都要准守这样的规则 74. **void** postTraversal(Node\* root) 75. { 76. **if** (root != NULL) 77. { 78. postTraversal(root->lchild); 79. postTraversal(root->rchild); 80. visit(root); 81. } 82. } 84. //树的层次遍历 85. //思想：使用队列queue。先将根节点入队列，循环判断当前队列不为空时，将头元素出队列并访问头元素，然后在将它的左节点和右节点入队列 86. **void** levelTraversal(Node\* root) 87. { 88. queue<Node\*> q; 89. Node\* p = root; 90. q.push(p); 91. **while** (q.empty() == **false**) 92. { 93. p = q.front(); 94. q.pop(); 95. visit(p); 96. **if** (p->lchild != NULL) 97. q.push(p->lchild); 98. **if** (p->rchild != NULL) 99. q.push(p->rchild); 100. } 101. } 103. **int** main() 104. { 105. //创建上面的树应输入应为  ABD##E##CF### 107. Node\* root; 108. creatTree(root); 109. cout << "前序遍历:"; 110. preTraversal(root); 111. cout << endl; 113. cout << "中序遍历:"; 114. midTraversal(root); 115. cout << endl; 117. cout << "后序遍历:"; 118. postTraversal(root); 119. cout << endl; 121. cout << "层次遍历:"; 122. levelTraversal(root); 123. cout << endl; 124. }   本次实验采用的测试样例如下：   1. //ABD##E##CF###     本次实验的测试结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 哈夫曼编码

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 要求对用户输入的数据信息进行哈夫曼编码，实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 哈夫曼编码(Huffman Coding)，又称霍夫曼编码，是Huffman于1952年提出一种编码方法。哈夫曼编码是可变字长编码(VLC)的一种，该方法完全依据字符出现概率来构造异字头的平均长度最短的码字。哈夫曼编码基于二叉树来实现的，根据字符出现的次数（或者说频率），来改变字符的编码长度，从而使字符的二进制编码达到最优的效果。  从各个节点中找出最小的两个节点，给它们建一个父节点，值为这两个节点之和。从节点序列中去除这两个节点，把他们的父节点加到序列中。重复上述步骤直至节点序列中只剩下唯一一个节点。这样得到的树又称作哈夫曼树。  对树来讲叶子节点的权值是对叶子节点赋予的一个有意义的数值量，这里就是字符出现的次数。设二叉树具有 n 个带权值的叶子结点，从根节点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和叫做二叉树的带权路径长度。给定一组具有确定权值的叶子结点，可以构造处不同的二叉树，将其中带权路径长度最小的二叉树称为哈夫曼树。  实现要求的效果和具体代码如下：     1. #include <iostream> 2. #include <string> 4. **using** **namespace** std; 5. **struct** Node 6. { 7. **int** lChild, rChild, parent; 8. string code; 9. **int** weight; 10. string c; 11. }; 13. //从当前叶节点和已经构造的节点中选出最小的两个数 14. **void** Select(Node nd[], **int** n, **int**& a, **int** &b) 15. { 16. **double** wt = 0; 17. **for** (**int** i = 0; i < n; ++i) 18. { 19. **if** (nd[i].parent != -1) 20. **continue**; 21. **if** (wt == 0 && nd[i].parent == -1) 22. { 23. wt = nd[i].weight; 24. a = i; 25. } 26. **else** **if** (nd[i].parent == -1 && nd[i].weight < wt) 27. { 28. wt = nd[i].weight; 29. a = i; 30. } 31. } 32. wt = 0; 33. **for** (**int** i = 0; i < n; ++i) 34. { 35. **if** (nd[i].parent != -1 || i == a) 36. **continue**; 37. **if** (wt == 0 && nd[i].parent == -1) 38. { 39. wt = nd[i].weight; 40. b = i; 41. } 42. **else** **if** (nd[i].parent == -1 && nd[i].weight < wt) 43. { 44. wt = nd[i].weight; 45. b = i; 46. } 47. } 48. } 49. //构造huffman树 50. **void** HuffTree(**int** w[], string ch[], **int** N, **int** M, Node nd[]) 51. { 52. //初始化Node数组 53. **for** (**int** i = 0; i < M; ++i) 54. { 55. //-1代表没有 56. nd[i].lChild = -1; 57. nd[i].rChild = -1; 58. nd[i].parent = -1; 59. nd[i].code = ""; 60. } 61. **for** (**int** i = 0; i < N; ++i) 62. { 63. nd[i].weight = w[i]; 64. nd[i].c = ch[i]; 65. } 66. **for** (**int** i = N; i < M; ++i) 67. { 68. **int** a = 0, b = 0; 69. Select(nd, i, a, b); 70. nd[a].parent = i; 71. nd[b].parent = i; 72. nd[i].lChild = a; 73. nd[i].rChild = b; 74. nd[i].weight = nd[a].weight + nd[b].weight; 75. } 76. } 77. **void** HuffEnCode(Node nd[], **int** N) 78. { 79. **int** i, j, k; 80. **for** (**int** i = 0; i < N; ++i) 81. { 82. string s = ""; 83. j = i; 84. //从叶节点一直找到根节点 85. **while** (nd[j].parent != -1) 86. { 87. k = nd[j].parent; 88. **if** (nd[k].lChild == j) 89. { 90. s = s + "0"; 91. } 92. **else** 93. { 94. s = s + "1"; 95. } 96. j = k; 97. } 98. **for** (**int** x = s.size(); x >= 0; --x) 99. nd[i].code += s[x]; 100. cout << nd[i].c << " の code: " << nd[i].code << endl; 101. } 102. } 103. **void** HuffmanDecode(Node nd[], string str, **int** N, **int** M) 104. { 105. string res = ""; 106. **int** i = 0, j = 0; 107. **int** p = M - 1; 108. **while** (i < str.size()) 109. { 110. **if** (str[i] == '0') 111. p = nd[p].lChild; 112. **if** (str[i] == '1') 113. p = nd[p].rChild; 114. //找到了叶节点 115. j = p; 116. **if** (p < N) 117. { 118. res += nd[p].c; 119. //重新回到叶节点 120. p = M - 1; 121. } 122. ++i; 123. } 124. **if** (j >= N) 125. cout << "ERROR"; 126. **else** 127. cout << res << endl; 128. } 129. **int** main() 130. { 131. **int** wt[1000]; 132. string ch[1000]; 133. **int** n; 134. cout<<"Please input character information (quantity) like 8:"; 135. cin>>n; 136. **for**(**int** i=0;i<n;i++) 137. { 138. cout<<"Please input the number & content of string as 10 A:"; 139. **int** w; 140. string s; 141. cin>>wt[i]>>ch[i]; 142. } 143. Node nd[2000]; //n个叶节点构造的哈夫曼树的结点个数为2n - 1 144. HuffTree(wt, ch, n, 2\*n-1, nd); 145. HuffEnCode(nd, n); 146. //解码 147. //  string str; 148. //  cin >> str; 149. //  HuffmanDecode(nd, str, n, 2\*n-1); 150. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 图的深度和广度优先遍历分析(1)

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 要求对邻接矩阵建图的图数据进行两种不同的遍历输出方式：广度优先遍历和深度优先遍历。实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 邻接矩阵是表示顶点之间相邻关系的矩阵。设G=(V,E)是一个图，其中V={v1,v2,…,vn} 。G的邻接矩阵是一个具有下列性质的n阶方阵：  ①对无向图而言，邻接矩阵一定是对称的，而且主对角线一定为零（在此仅讨论无向简单图），副对角线不一定为0，有向图则不一定如此。  ②在无向图中，任一顶点i的度为第i列（或第i行）所有非零元素的个数，在有向图中顶点i的出度为第i行所有非零元素的个数，而入度为第i列所有非零元素的个数。  ③用邻接矩阵法表示图共需要n^2个空间，由于无向图的邻接矩阵一定具有对称关系，所以扣除对角线为零外，仅需要存储上三角形或下三角形的数据即可，因此仅需要n（n-1）/2个空间。  无向图的邻接矩阵一定是对称的，而有向图的邻接矩阵不一定对称。因此，用邻接矩阵来表示一个具有n个顶点的有向图时需要n^2个单元来存储邻接矩阵；对有n个顶点的无向图则只存入上（下）三角阵中剔除了左上右下对角线上的0元素后剩余的元素，故只需1+2+...+(n-1)=n(n-1)/2个单元。  无向图邻接矩阵的第i行（或第i列）非零元素的个数正好是第i个顶点的度。有向图邻接矩阵中第i行非零元素的个数为第i个顶点的出度，第i列非零元素的个数为第i个顶点的入度，第i个顶点的度为第i行与第i列非零元素个数之和。  可见，用邻接矩阵表示图，很容易确定图中任意两个顶点是否有边相连。  再介绍一下深度优先遍历和广度优先遍历的遍历方法。  深度优先遍历：类似与树的前序遍历。从图中的某个顶点v出发，访问此顶点，然后从v的未被访问到的邻接点进行遍历，直到图中所有和v有路径相通的顶点都被访问到（注：优先访问外层节点，访问到无新顶点时，会进行回退，访问未被访问过的分支顶点）。  广度优先遍历：类似于树的层序遍历。从图中的某个顶点w出发，让顶点w入队，然后顶点w再出队，并让所有和顶点w相连的顶点入队，然后再出队一个顶点t，并让所有和t相连但未被访问过的顶点入队……由此循环，指定图中所有元素都出队。  本实验构建的无向图可表示为：    广度优先遍历的过程可表示为：    深度优先遍历的过程可表示为：    实现要求的效果和具体代码如下：     1. // 邻接矩阵的深度和广度优先遍历 2. #include <stdio.h> 4. #define OK 1      // 执行成功 5. #define ERROR 0   // 执行失败 6. #define TRUE 1    // 返回值为真 7. #define FALSE 0   // 返回值为假 9. **typedef** **int** Status; // 执行状态（OK、ERROR） 10. **typedef** **int** Boolean; // 布尔值（TRUE、FALSE） 12. **typedef** **char** VertexType; // 顶点元素类型 13. **typedef** **int** EdgeType; // 边上权值的类型 15. #define MAXSIZE 9 // 队列储存空间初始分配量 16. #define MAXVEX 100 // 最大顶点数 18. // 邻接矩阵结构（无向图） 19. **typedef** **struct** { 20. VertexType vexs[MAXVEX]; // 顶点表 21. EdgeType arc[MAXVEX][MAXVEX]; // 边表 22. **int** numNodes, numEdges; // 图的顶点数、边数 23. } MGraph; 25. // 循环队列顺序存储结构 26. **typedef** **struct** { 27. **int** data[MAXSIZE]; // 用于存值的数组 28. **int** front; // 头指针 29. **int** rear; // 尾指针，若队列不空，指向队尾元素的下一个位置 30. } Queue; 32. /\*\* 33. \* 初始化一个空队列 34. \* @param Q 队列 35. \* @return 执行状态 36. \*/ 37. Status InitQueue(Queue \*Q) { 38. Q->front = Q->rear=  0; // 队头和队尾指针都指向0 39. **return** OK; 40. } 42. /\*\* 43. \* 判断队列是否为空 44. \* @param Q 队列 45. \* @return 队列是否为空 46. \*/ 47. Boolean QueueEmpty(Queue Q) { 48. **if** (Q.front == Q.rear) { // 队头等于队尾指针，队列为空 49. **return** TRUE; 50. } **else** { 51. **return** FALSE; 52. } 53. } 55. /\*\* 56. \* 将元素e插入队列Q的队尾 57. \* @param Q 队列 58. \* @param e 插入的元素 59. \* @return 执行状态 60. \*/ 61. Status EnQueue(Queue \*Q, **int** e) { 62. // 队列已满，插入失败 63. **if** ((Q->rear + 1) % MAXSIZE == Q->front) { 64. **return** ERROR; 65. } 67. // 将元素e插入队尾 68. Q->data[Q->rear] = e; 70. // 设置队尾指针指向下一个位置，若到最后则转向头部 71. Q->rear = (Q->rear + 1) % MAXSIZE; 72. **return** OK; 73. } 75. /\*\* 76. \* 队头元素出队，用e返回其值 77. \* @param Q 队列 78. \* @param e 队头元素的值 79. \* @return 执行状态 80. \*/ 81. Status DeQueue(Queue \*Q, **int** \*e) { 82. // 对头指针等于对尾指针，此时队列为空，出队失败 83. **if** (Q->front == Q->rear) { 84. **return** ERROR; 85. } 87. // 将队头元素的值赋给元素e 88. \*e = Q->data[Q->front]; 90. // 设置队头指针指向下一个位置，若到最后则转向头部 91. Q->front = (Q->front + 1) % MAXSIZE; 92. **return** OK; 93. } 95. /\*\* 96. \* 生成邻接矩阵 97. \* @param G 邻接矩阵 98. \*/ 99. **void** CreateMGraph(MGraph \*G) { 100. **int** i, j; // 用于遍历元素 102. G->numEdges = 15; // 设置有15条边 103. G->numNodes = 9; // 设置有9个顶点 105. // 读入顶点信息，建立顶点表 106. G->vexs[0] = 'A'; 107. G->vexs[1] = 'B'; 108. G->vexs[2] = 'C'; 109. G->vexs[3] = 'D'; 110. G->vexs[4] = 'E'; 111. G->vexs[5] = 'F'; 112. G->vexs[6] = 'G'; 113. G->vexs[7] = 'H'; 114. G->vexs[8] = 'I'; 116. // 初始化图的边 117. **for** (i = 0; i < G->numNodes; i++) { 118. **for** (j = 0; j < G->numNodes; j++) { 119. G->arc[i][j] = 0; // 设置所有边的值都为0 120. } 121. } 123. // 设置特定边（如果arc[i][j] = 1，代表顶点i到顶点j有边相连） 124. G->arc[0][1] = 1; 125. G->arc[0][5] = 1; 127. G->arc[1][2] = 1; 128. G->arc[1][8] = 1; 129. G->arc[1][6] = 1; 131. G->arc[2][3] = 1; 132. G->arc[2][8] = 1; 134. G->arc[3][4] = 1; 135. G->arc[3][7] = 1; 136. G->arc[3][6] = 1; 137. G->arc[3][8] = 1; 139. G->arc[4][5] = 1; 140. G->arc[4][7] = 1; 142. G->arc[5][6] = 1; 144. G->arc[6][7] = 1; 146. // 设置对称边 147. **for** (i = 0; i < G->numNodes; i++) { 148. **for** (j = i; j < G->numNodes; j++) { 149. G->arc[j][i] = G->arc[i][j]; 150. } 151. } 152. } 154. // 访问标志的数组 155. Boolean visited[MAXVEX]; 157. /\*\* 158. \* 邻接矩阵的深度优先递归算法 159. \* @param G 邻接矩阵 160. \* @param i 顶点下标 161. \*/ 162. **void** DFS(MGraph G, **int** i) { 163. **int** j; // 用于遍历元素 165. visited[i] = TRUE; // 记录该下标的元素已被访问 167. printf("%c ", G.vexs[i]); // 打印该位置的顶点值 169. // 遍历图中的顶点 170. **for** (j = 0; j < G.numNodes; j++) { 171. // 顶点i到顶点j有边相连，并且顶点j未被访问过 172. **if** (G.arc[i][j] == 1 && !visited[j]) { 173. DFS(G, j); // 对顶点j进行访问 174. } 175. } 176. } 178. /\*\* 179. \* 邻接矩阵的深度遍历 180. \* @param G 邻接矩阵 181. \*/ 182. **void** DFSTraverse(MGraph G) { 183. **int** i; // 用于遍历元素 185. // 初始化设置所有顶点都没被访问过 186. **for** (i = 0; i < G.numNodes; i++) { 187. visited[i] = FALSE; 188. } 190. // 遍历顶点i 191. **for** (i = 0; i < G.numNodes; i++) { 192. // 如果顶点i未被访问过 193. **if** (!visited[i]) { 194. DFS(G, i); // 访问顶点i 195. } 196. } 197. } 199. /\*\* 200. \* 邻接矩阵的广度遍历算法 201. \* @param G 邻接矩阵 202. \*/ 203. **void** BFSTraverse(MGraph G) { 204. **int** i, j; // 用于遍历元素 205. Queue Q; // 队列 207. // 初始设置图的所有顶点都没被访问过 208. **for** (i = 0; i < G.numNodes; i++) { 209. visited[i] = FALSE; 210. } 212. InitQueue(&Q); // 初始化队列 214. // 对每一个顶点做循环 215. **for** (i = 0; i < G.numNodes; i++) { 216. **if** (!visited[i]) { // 该顶点未被访问过，进行处理 217. visited[i] = TRUE; // 设置该顶点i已被访问 219. printf("%c ", G.vexs[i]); // 打印该顶点i的值 221. EnQueue(&Q, i); // 将该顶点i入队 223. // 当队列非空时，进行循环 224. **while** (!QueueEmpty(Q)) { 225. DeQueue(&Q, &i); // 将队头元素出队，赋值给i 227. // 遍历当前节点以外的节点j 228. **for** (j = 0; j < G.numNodes; j++) { 229. // 若顶点j与当前节点存在边，并且未被访问过 230. **if** (G.arc[i][j] == 1 && !visited[j]) { 231. visited[j] = TRUE; // 设置顶点j已被访问 232. printf("%c ", G.vexs[j]); // 打印顶点j的值 234. EnQueue(&Q, j); // 将顶点j入队 235. } 236. } 237. } 238. } 239. } 240. } 242. **int** main() { 243. MGraph G; // 邻接矩阵 244. CreateMGraph(&G); // 创建邻接矩阵 245. printf("深度遍历："); 246. DFSTraverse(G); // 深度遍历邻接矩阵 247. printf("\n"); 248. printf("广度遍历："); 249. BFSTraverse(G); // 广度遍历邻接矩阵 251. **return** 0; 252. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 图的深度和广度优先遍历分析(2)

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 要求对邻接矩阵建图的图数据进行两种不同的遍历输出方式：广度优先遍历和深度优先遍历。实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 邻接表，存储方法跟树的孩子链表示法相类似，是一种顺序分配和链式分配相结合的存储结构。如这个表头结点所对应的顶点存在相邻顶点，则把相邻顶点依次存放于表头结点所指向的单向链表中。  图的邻接表存储方法跟树的孩子链表示法相类似，是一种顺序分配和链式分配相结合的存储结构。如这个表头结点所对应的顶点存在相邻顶点，则把相邻顶点依次存放于表头结点所指向的单向链表中。如词条概念图所示，表结点存放的是邻接顶点在数组中的索引。对于无向图来说，使用邻接表进行存储也会出现数据冗余，表头结点A所指链表中存在一个指向C的表结点的同时，表头结点C所指链表也会存在一个指向A的表结点。  邻接表是图的一种最主要存储结构,用来描述图上的每一个点。对图的每个顶点建立一个容器（n个顶点建立n个容器），第i个容器中的结点包含顶点Vi的所有邻接顶点。实际上我们常用的邻接矩阵就是一种未离散化每个点的边集的邻接表。  在有向图中，描述每个点向别的节点连的边（点a->点b这种情况）。在无向图中，描述每个点所有的边(点a-点b这种情况)。对于无向图来说，使用邻接表进行存储也会出现数据冗余，表头结点A所指链表中存在一个指向C的表结点的同时，表头结点C所指链表也会存在一个指向A的表结点。  同样的，讲完邻接表之后再介绍一下深度优先和广度优先遍历的遍历方法。  深度优先遍历：类似与树的前序遍历。从图中的某个顶点v出发，访问此顶点，然后从v的未被访问到的邻接点进行遍历，直到图中所有和v有路径相通的顶点都被访问到（注：优先访问外层节点，访问到无新顶点时，会进行回退，访问未被访问过的分支顶点）。  广度优先遍历：类似于树的层序遍历。从图中的某个顶点w出发，让顶点w入队，然后顶点w再出队，并让所有和顶点w相连的顶点入队，然后再出队一个顶点t，并让所有和t相连但未被访问过的顶点入队……由此循环，指定图中所有元素都出队。  本实验使用邻接表建图，使用示例如图所示：    代码测试结果如图:    使用的测试样例为：  //4 5  //A B C D  //2 3  //2 1  //0 3  //0 2  //0 1  具体代码如下：   1. #include <iostream> 2. #include <queue> 4. **using** **namespace** std; 6. //访问标志的数组,为1表示访问过，为0表示未被访问 7. **int** visted[100]; 8. //边表结点 9. **typedef** **struct** EdgeNode { 10. //顶点对应的下标 11. **int** adjvex; 12. //指向下一个邻接点 13. **struct** EdgeNode \*next; 14. } edgeNode; 16. //顶点表结点 17. **typedef** **struct** VertexNode { 18. //顶点数据 19. **char** data; 20. //边表头指针 21. edgeNode \*firstedge; 22. } VertexNode, AdjList[100]; 24. //集合 25. **typedef** **struct** { 26. AdjList adjList; 27. //顶点数和边数 28. **int** numVertexes, numEdges; 29. } GraphAdjList; 31. **void** CreateALGraph(GraphAdjList \*G) { 32. **int** i, j, k; 33. edgeNode \*e; 34. cout << "输入顶点数和边数" << endl; 35. //输入顶点数和边数 36. cin >> G->numVertexes >> G->numEdges; 37. //读入顶点信息，建立顶点表 38. **for** (i = 0; i < G->numVertexes; i++) 39. { 40. //输入顶点信息 41. cin >> G->adjList[i].data; 42. //将边表置为空表 43. G->adjList[i].firstedge = NULL; 44. } 45. //建立边表（头插法） 46. **for** (k = 0; k < G->numEdges; k++) 47. { 48. cout << "输入边（vi,vj）上的顶点序号 :" ; 49. cin >> i >> j; 50. e = **new** EdgeNode; 51. e->adjvex = j; 52. e->next = G->adjList[i].firstedge; 53. G->adjList[i].firstedge = e; 55. e = **new** EdgeNode; 57. e->adjvex = i; 58. e->next = G->adjList[j].firstedge; 59. G->adjList[j].firstedge = e; 60. } 61. } 63. **void** Test() { 64. cout << "ALL IS OK." << endl; 65. } 67. **void** howALGraph(GraphAdjList \*G) { 68. **for** (**int** i = 0; i < G->numVertexes; i++) 69. { 70. cout << "顶点" << i << ": " << G->adjList[i].data << "--firstedge--"; 71. edgeNode \*p = **new** edgeNode; 72. p = G->adjList[i].firstedge; 73. **while** (p) 74. { 75. cout << p->adjvex << "-->"; 76. p = p->next; 77. } 78. cout << "NULL" << endl; 79. } 81. } 83. **void** DFS(GraphAdjList \*G, **int** i) { 84. EdgeNode \*p; 85. visted[i] = 1; 86. cout << G->adjList[i].data << "-->"; 87. p = G->adjList[i].firstedge; 88. **while** (p) 89. { 90. **if** (!visted[p->adjvex]) 91. { 92. //递归访问 93. DFS(G, p->adjvex); 94. } 95. p = p->next; 96. } 98. } 100. **void** DFSTraverse(GraphAdjList \*G) { 101. //初始化所有顶点都没有访问过 102. cout << "深度优先遍历结果为：" ; 103. **for** (**int** i = 0; i < G->numVertexes; i++) 104. { 105. visted[i] = 0; 106. } 107. **for** (**int** i = 0; i < G->numVertexes; i++) 108. { 109. **if** (visted[i] == 0) 110. { 111. DFS(G, i); 112. } 113. } 114. cout << "NULL" << endl; 115. } 117. **void** BFSTraverse(GraphAdjList \*G) { 119. EdgeNode \*p; 120. queue<**int**> Q; 121. **for** (**int** i = 0; i < G->numVertexes; i++) 122. { 123. visted[i] = 0; 124. } 125. cout << "广度优先遍历结果为："; 126. **for** (**int** i = 0; i < G->numVertexes; i++) 127. { 128. **if** (visted[i] == 0) 129. { 130. visted[i] = 1; 131. cout << G->adjList[i].data << "-->"; 132. Q.push(i); 133. **while** (!Q.empty()) 134. { 135. Q.front()=i; 136. Q.pop(); 137. p = G->adjList[i].firstedge; 138. **while** (p) 139. { 140. **if** (visted[p->adjvex] == 0) 141. { 142. visted[p->adjvex] = 1; 143. cout << G->adjList[p->adjvex].data << "-->"; 144. Q.push(p->adjvex); 145. } 146. p = p->next; 147. } 148. } 149. } 150. } 151. cout << "NULL" << endl; 152. }  155. **int** main() { 157. GraphAdjList \*GA = **new** GraphAdjList; 158. Test(); 159. CreateALGraph(GA); 160. BFSTraverse(GA); 161. DFSTraverse(GA); 163. **return** 0; 164. } |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# Prim算法（最小生成树）

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用Prim算法基于图的两种存储形式（邻接表和邻接矩阵）实现最小生成树的权值计算。  本次实验主要由两个程序组成。  **1.无向图获取最小生成树（使用邻接矩阵存储有向图结构）**  **2. 有向图获取最小生成树（使用邻接表存储有向图结构）**  本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| Prim算法是一种用来查找最小生成树的算法。  而最小生成树是指在一给定的无向图G = (V, E) 中，(u, v) 代表连接顶点 u 与顶点 v 的边（即），而 w(u, v) 代表此边的权重，若存在 T 为 E 的子集（即）且为无循环图，使得的 w(T) 最小，则此 T 为 G 的最小生成树。    Prim算法的本质是贪心算法，首先输入一个顶点作为起点，提取这个顶点的序号，然后有一个closedge数组来初始化，遍历之后将每个顶点的前一个访问的结点初始化为u,并且将他们之间的边的权重直接当成二者之间的权重。然后首先输出一个u.Minmum这个函数主要是用来求出所有的边中和u相连的边的权重最小的顶点值(第一个循环的时候),然后输出这个最小的权值以及顶点。并且将这个顶点的标记为lowcast=0；在后面的访问中就不会用到这个顶点。之后由于这个时候已知的顶点是两个，因此对于所有和第二个结点中连接边长最小的顶点，需要改变边的权值以及前一个顶点的位置。然后第二轮循环进来的时候，同样在Minmum这个函数中实现找出所有没有被访问过的权值最小的顶点，这个时候不需要担心会访问非前面两个邻接点，因为如果不是邻接点，它们的边的权重将是无穷大，因此肯定会访问权值最小的结点。然后循环将每一个结点都输出。  1. 无向图获取最小生成树  无向图采用的测试样例如下图所示：    完成实验要求的具体代码如下所示：   1. #include<stdio.h> 2. #include<stdlib.h> 3. #define OK 1 4. #define NO 0 5. #define FALSE 0 6. #define TRUE 1 7. #define ERROR -1 9. #define INFINITY 200000 10. #define MAX\_VERTEX\_NUM 20 12. **typedef** **int** \*SetType; 13. **typedef** **int** Status; 14. **typedef** **int** VRType;//图、表顶点关系类型 15. **typedef** **struct** ArcCell 16. { 17. VRType adj;  //权值 18. }ArcCell; 20. **typedef** ArcCell AdjMaxtrix[MAX\_VERTEX\_NUM+1][MAX\_VERTEX\_NUM+1];//邻接矩阵 21. **typedef** **char** VertexType\_M;//图的顶点类型  24. **typedef** **struct** 25. { 26. VertexType\_M vexs[MAX\_VERTEX\_NUM+1];//顶点向量 27. AdjMaxtrix arcs; 28. **int** vexnum,arcnum; 29. }MGraph; 31. //作用非常的重要,主要是为了实现记录每一个结点的前一个结点以及它们之间的权重 32. **typedef** **struct** 33. { 34. VertexType\_M adjvex;//较早加入当前边的端点 35. VRType lowcost;//当前边的权值 37. }Edge; 39. **typedef** **struct**  //定义一个边集用来存储图的所有边信息 40. { 41. **int** begin,end; 42. **int** weight; 43. }gEdge; 45. Status visited[MAX\_VERTEX\_NUM+1]; 46. Status CreateUDN\_M(MGraph \*G); 47. **void** OutputMGraph(MGraph G); 48. **int** LocateVex\_M(MGraph G,VertexType\_M u); 49. **int** MinSpanTree\_PRIM(MGraph G,VertexType\_M u); 50. **int** Minimum(Edge closedge[],**int** n); 51. gEdge \*CreateEdges(MGraph G);//生成图的排序过的边集数组 53. **int** main(**int** argc,**char**\*\* argv){ 54. MGraph G; 55. printf("ALL IS RIGHT~\n"); 56. CreateUDN\_M(&G); 57. printf("Adjacency matrix information you entered :\n"); 58. OutputMGraph(G); 59. printf("\n"); 60. **int** total; 61. printf("Each node added to the tree and its corresponding weights are as follows :\n"); 62. total=MinSpanTree\_PRIM(G,'A'); 63. printf("\n"); 64. printf("The value of minimum spanning tree is %d\n",total); 65. printf("\n"); 66. **return** 0; 67. } 68. Status CreateUDN\_M(MGraph \*G){ 69. **int** i,j,k; 70. VertexType\_M v1,v2; 71. VRType w; 72. printf("Enter the number of vertices :"); 73. scanf("%d",&(G->vexnum)); 74. printf("Enter the number of edges :"); 75. scanf("%d",&(G->arcnum)); 76. printf("Enter values for each vertex & information of all edges:\n"); 77. getchar(); 78. **for**(i=1;i<=G->vexnum;i++) 79. { 80. scanf("%c ",&(G->vexs[i])); 81. } 82. **for**(i=1;i<=G->vexnum;i++) 83. { 84. **for**(j=1;j<=G->vexnum;j++) 85. G->arcs[i][j].adj=INFINITY; 86. } 87. **for**(k=1;k<=G->arcnum;k++) 88. { 89. getchar(); 90. scanf("%c %c %d",&v1,&v2,&w); 91. i=LocateVex\_M(\*G,v1); 92. j=LocateVex\_M(\*G,v2); 94. G->arcs[i][j].adj=w; 95. G->arcs[j][i]=G->arcs[i][j]; 96. } 97. **return** OK; 98. } 100. **int** LocateVex\_M(MGraph G,VertexType\_M u) 101. { 102. **int** i; 103. **for**(i=1;i<=G.vexnum;i++) 104. { 105. **if**(G.vexs[i]==u) 106. **return** i; 107. } 108. **return** 0; 109. } 111. **void** OutputMGraph(MGraph G){ 112. **int** i,j; 113. **if**(!G.vexnum&&!G.arcnum) 114. printf("NULL\n"); 115. **else** 116. { 117. printf(" "); 118. **for**(i=1;i<=G.vexnum;i++) 119. { 120. printf("%3c",G.vexs[i]); 121. } 122. printf("\n"); 123. **for**(i=1;i<=G.vexnum;i++) 124. { 125. printf("%c",G.vexs[i]); 126. **for**(j=1;j<=G.vexnum;j++) 127. { 128. **if**(G.arcs[i][j].adj==INFINITY) 129. printf(" ∞"); 130. **else** 131. printf("%3d",G.arcs[i][j]); 132. } 133. printf("\n"); 134. } 135. } 136. } 138. **int** MinSpanTree\_PRIM(MGraph G,VertexType\_M u){ 139. **int** i,j,k; 140. **int** total=0; 141. Edge closedge[G.vexnum+1]; 142. k=LocateVex\_M(G,u); 143. /\*将辅助矩阵初始化使每一个结点都标记为前一个结点是u,并且它们之间的距离是无穷大，若是邻接点那么就是adj的值\*/ 144. **for**(j=1;j<=G.vexnum;j++) 145. { 146. **if**(j!=k) 147. { 148. closedge[j].adjvex=u;  //标记j这个顶点的前一个顶点的信息 149. closedge[j].lowcost=G.arcs[k][j].adj; 150. } 151. } 152. closedge[k].lowcost=0; //自己到自己的cost标记为0 153. printf("    %c\n", u); 154. **for**(i=1;i<=G.vexnum-1;i++)//总共需要这么多此找出最小边 155. { 156. k=Minimum(closedge,G.vexnum); 157. total+=closedge[k].lowcost; 158. printf("%2d,%c\n",closedge[k].lowcost,G.vexs[k]); 159. closedge[k].lowcost=0; 160. **for**(j=1;j<=G.vexnum;j++) 161. { 162. **if**(G.arcs[k][j].adj<closedge[j].lowcost) 163. { 164. closedge[j].adjvex=G.vexs[k]; 165. closedge[j].lowcost=G.arcs[k][j].adj; 166. } 167. } 168. } 169. **return** total; 170. } 172. //找到所有和结点邻接的点中权值最小的并且返回j（用来标记最小的值所在的下标） 173. **int** Minimum(Edge closedge[],**int** n){ 174. **int** i,j; 175. **int** min=INFINITY; 176. i=1; 177. j=0; 178. **for**(;i<=n;i++){ 179. **if**(closedge[i].lowcost)//从权值不为0的边中选择拥有最小权值的边 180. { 181. **if**(closedge[i].lowcost<=min) 182. { 183. min=closedge[i].lowcost; 184. j=i; 185. } 186. } 187. } 188. **return** j; 189. } 191. gEdge \*CreateEdges(MGraph G){ 192. **int** i,j; 193. **int** k=1; 194. **int** EdgeNum=G.arcnum; 195. **int** VertexNum=G.vexnum; 196. gEdge temp; 197. gEdge \*p=(gEdge\*)malloc(**sizeof**(gEdge)\*VertexNum\*VertexNum);   //之前程序报错 是因为申请的空间不够大，越界了 199. **for**(i=1;i<=VertexNum;i++) //边集数组初始化 200. **for**(j=i;j<=VertexNum;j++) //为了避免无向图的每条边被记录两次，只考虑上三角矩阵 201. **if**(G.arcs[i][j].adj!=0&&G.arcs[i][j].adj!=INFINITY){ 202. p[k].begin=i; 203. p[k].end=j; 204. p[k].weight=G.arcs[i][j].adj; 205. k++; 206. } 207. //首个p[i]与后面i+1……最后个依次比较，每一次都将小的放在第i个 208. **for**(i=1;i<EdgeNum;i++)//对边集数组进行选择排序 209. **for**(j=i+1;j<=EdgeNum;j++) 210. **if**(p[i].weight>p[j].weight){ 211. temp=p[i]; 212. p[i]=p[j]; 213. p[j]=temp; 214. } 215. **return** p; //这个时候返回的p是已经从小到大排好序的，并且拥有一共EdgeNumt个大小的数组 216. }   代码运行的结果如下图所示：    2. 有向图获取最小生成树  有向图采用的测试样例如下图所示：    完成实验要求的具体代码如下所示：   1. #include <iostream> 2. #include <cstdio> 3. #include <vector> 4. #include <queue> 5. **using** **namespace** std; 7. #define maxn 110  //最大顶点个数 8. **int** n, m;       //顶点数，边数 10. **struct** arcnode  //边结点 11. { 12. **int** vertex;     //与表头结点相邻的顶点编号 13. **int** weight;     //连接两顶点的边的权值 14. arcnode \* next; //指向下一相邻接点 15. arcnode() {} 16. arcnode(**int** v,**int** w):vertex(v),weight(w),next(NULL) {} 17. }; 19. **struct** vernode      //顶点结点，为每一条邻接表的表头结点 20. { 21. **int** vex;    //当前定点编号 22. arcnode \* firarc;   //与该顶点相连的第一个顶点组成的边 23. }Ver[maxn]; 25. **void** Init()  //建立图的邻接表需要先初始化，建立顶点结点 26. { 27. **for**(**int** i = 1; i <= n; i++) 28. { 29. Ver[i].vex = i; 30. Ver[i].firarc = NULL; 31. } 32. } 34. **void** Insert(**int** a, **int** b, **int** w)  //尾插法，插入以a为起点，b为终点，权为w的边，效率不如头插，但是可以去重边 35. { 36. arcnode \* q = **new** arcnode(b, w); 37. **if**(Ver[a].firarc == NULL) 38. Ver[a].firarc = q; 39. **else** 40. { 41. arcnode \* p = Ver[a].firarc; 42. **if**(p->vertex == b) 43. { 44. **if**(p->weight > w) 45. p->weight = w; 46. **return** ; 47. } 48. **while**(p->next != NULL) 49. { 50. **if**(p->next->vertex == b) 51. { 52. **if**(p->next->weight > w); 53. p->next->weight = w; 54. **return** ; 55. } 56. p = p->next; 57. } 58. p->next = q; 59. } 60. } 61. **void** Insert2(**int** a, **int** b, **int** w)   //头插法，效率更高，但不能去重边 62. { 63. arcnode \* q = **new** arcnode(b, w); 64. **if**(Ver[a].firarc == NULL) 65. Ver[a].firarc = q; 66. **else** 67. { 68. arcnode \* p = Ver[a].firarc; 69. q->next = p; 70. Ver[a].firarc = q; 71. } 72. } 73. **struct** node     //保存key值的结点 74. { 75. **int** v; 76. **int** key; 77. **friend** **bool** operator<(node a, node b)   //自定义优先级，key小的优先 78. { 79. **return** a.key > b.key; 80. } 81. }; 83. #define INF 0xfffff    //权值上限 84. **int** parent[maxn];   //每个结点的父节点 85. **bool** visited[maxn]; //是否已经加入树种 86. node vx[maxn];      //保存每个结点与其父节点连接边的权值 87. priority\_queue<node> q; //优先队列stl实现 88. **void** Prim()    //s表示根结点 89. { 90. **for**(**int** i = 1; i <= n; i++) //初始化 91. { 92. vx[i].v = i; 93. vx[i].key = INF; 94. parent[i] = -1; 95. visited[i] = **false**; 96. } 97. vx[1].key = 0; 98. q.push(vx[1]); 99. **while**(!q.empty()) 100. { 101. node nd = q.top();  //取队首，记得赶紧pop掉 102. q.pop(); 103. **if**(visited[nd.v])   //注意这一句的深意，避免很多不必要的操作 104. **continue**; 105. visited[nd.v] = **true**; 106. arcnode \* p = Ver[nd.v].firarc; 107. **while**(p != NULL)    //找到所有相邻结点，若未访问，则入队列 108. { 109. **if**(!visited[p->vertex] && p->weight < vx[p->vertex].key) 110. { 111. parent[p->vertex] = nd.v; 112. vx[p->vertex].key = p->weight; 113. vx[p->vertex].v = p->vertex; 114. q.push(vx[p->vertex]); 115. } 116. p = p->next; 117. } 118. } 119. } 121. **int** main() 122. { 123. **int** a, b ,w; 124. cout << "Enter the number of nodes & edges: "; 125. cin >> n >> m; 126. Init(); 127. cout << "Enter all edge information :" << endl; 128. **while**(m--) 129. { 130. cin >> a >> b >> w; 131. Insert2(a, b, w); 132. } 133. Prim(); 134. cout << "Output the parent nodes of all nodes :" << endl; 135. **for**(**int** i = 1; i <= n; i++) 136. cout << parent[i] << " "; 137. cout << endl; 138. cout << "The weight of the minimum spanning tree is ："; 139. **int** cnt = 0; 140. **for**(**int** i = 1; i <= n; i++) 141. cnt += vx[i].key; 142. cout << cnt << endl; 143. **return** 0; 144. }   代码运行的结果如下图所示： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# Kruskal算法（最小生成树）

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用Kruskal算法基于图的两种存储形式（邻接表和邻接矩阵）实现最小生成树的权值计算。  本次实验主要由两个程序组成。  **1.无向图获取最小生成树（使用邻接矩阵存储有向图结构）**  **2. 有向图获取最小生成树（使用邻接表存储有向图结构）**  本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| Kruskal算法是一种用来查找最小生成树的算法。  而最小生成树是指在一给定的无向图G = (V, E) 中，(u, v) 代表连接顶点 u 与顶点 v 的边（即），而 w(u, v) 代表此边的权重，若存在 T 为 E 的子集（即）且为无循环图，使得的 w(T) 最小，则此 T 为 G 的最小生成树。    Kruskal算法的基本思想是先构造一个只含 n 个顶点、而边集为空的子图，把子图中各个顶点看成各棵树上的根结点，之后，从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图，即把两棵树合成一棵树，反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直到森林中只有一棵树，也即子图中含有 n-1 条边为止。  Kruskal算法的实现步骤可概括为以下四点。  1.新建图G，G中拥有原图中相同的节点，但没有边；  2.将原图中所有的边按权值从小到大排序；  3.从权值最小的边开始，如果这条边连接的两个节点于图G中不在同一个连通分量中，则添加这条边到图G中；  4.重复3，直至图G中所有的节点都在同一个连通分量中。  在了解完Kruskal算法的基本思想和实现步骤之后，我开始用代码解释和执行，希望利用Kruskal算法解决有关图的最小生成树问题。  1. 无向图获取最小生成树  无向图采用的测试样例如下图所示：    完成实验要求的具体代码如下图所示：   1. #include<stdio.h> 2. #include<stdlib.h> 3. #define OK 1 4. #define NO 0 5. #define FALSE 0 6. #define TRUE 1 7. #define ERROR -1 8. #define INFINITY 200000 9. #define MAX\_VERTEX\_NUM 20 11. **typedef** **int** \*SetType; 12. **typedef** **int** Status; 13. **typedef** **int** VRType;//图、表顶点关系类型 14. **typedef** **struct** ArcCell 15. { 16. VRType adj;  //权值 17. }ArcCell; 19. **typedef** ArcCell AdjMaxtrix[MAX\_VERTEX\_NUM+1][MAX\_VERTEX\_NUM+1];//邻接矩阵 20. **typedef** **char** VertexType\_M;//图的顶点类型 22. **typedef** **struct** 23. { 24. VertexType\_M vexs[MAX\_VERTEX\_NUM+1];//顶点向量 25. AdjMaxtrix arcs; 26. **int** vexnum,arcnum; 27. }MGraph; 28. //作用非常的重要,主要是为了实现记录每一个结点的前一个结点以及它们之间的权重 29. **typedef** **struct** 30. { 31. VertexType\_M adjvex;//较早加入当前边的端点 32. VRType lowcost;//当前边的权值 33. }Edge; 35. **typedef** **struct**  //定义一个边集用来存储图的所有边信息 36. { 37. **int** begin,end; 38. **int** weight; 39. }gEdge; 41. Status visited[MAX\_VERTEX\_NUM+1]; 42. Status CreateUDN\_M(MGraph \*G); 43. **void** OutputMGraph(MGraph G); 44. **int** LocateVex\_M(MGraph G,VertexType\_M u); 45. **int** Minimum(Edge closedge[],**int** n); 46. **void** MinSpanTree\_KRUSKAL(MGraph G); 47. gEdge \*CreateEdges(MGraph G);//生成图的排序过的边集数组 49. **int** main(**int** argc,**char**\*\* argv){ 50. MGraph G; 51. printf("ALL IS RIGHT~\n"); 52. CreateUDN\_M(&G); 53. printf("Adjacency matrix information you entered :\n"); 54. OutputMGraph(G); 55. printf("\n"); 56. MinSpanTree\_KRUSKAL(G); 57. printf("\n"); 58. **return** 0; 59. } 60. Status CreateUDN\_M(MGraph \*G){ 61. **int** i,j,k; 62. VertexType\_M v1,v2; 63. VRType w; 64. printf("Enter the number of vertices :"); 65. scanf("%d",&(G->vexnum)); 66. printf("Enter the number of edges :"); 67. scanf("%d",&(G->arcnum)); 68. printf("Enter values for each vertex & information of all edges:\n"); 69. getchar(); 70. **for**(i=1;i<=G->vexnum;i++) 71. { 72. scanf("%c ",&(G->vexs[i])); 73. } 74. **for**(i=1;i<=G->vexnum;i++) 75. { 76. **for**(j=1;j<=G->vexnum;j++) 77. G->arcs[i][j].adj=INFINITY; 78. } 79. **for**(k=1;k<=G->arcnum;k++) 80. { 81. getchar(); 82. scanf("%c %c %d",&v1,&v2,&w); 83. i=LocateVex\_M(\*G,v1); 84. j=LocateVex\_M(\*G,v2); 85. G->arcs[i][j].adj=w; 86. G->arcs[j][i]=G->arcs[i][j]; 87. } 88. **return** OK; 89. } 91. **int** LocateVex\_M(MGraph G,VertexType\_M u) 92. { 93. **int** i; 94. **for**(i=1;i<=G.vexnum;i++) 95. { 96. **if**(G.vexs[i]==u) 97. **return** i; 98. } 99. **return** 0; 100. } 102. **void** OutputMGraph(MGraph G){ 103. **int** i,j; 104. **if**(!G.vexnum&&!G.arcnum) 105. printf("NULL\n"); 106. **else** 107. { 108. printf(" "); 109. **for**(i=1;i<=G.vexnum;i++) 110. { 111. printf("%3c",G.vexs[i]); 112. } 113. printf("\n"); 114. **for**(i=1;i<=G.vexnum;i++) 115. { 116. printf("%c",G.vexs[i]); 117. **for**(j=1;j<=G.vexnum;j++) 118. { 119. **if**(G.arcs[i][j].adj==INFINITY) 120. printf(" ∞"); 121. **else** 122. printf("%3d",G.arcs[i][j]); 123. } 124. printf("\n"); 125. } 126. } 127. } 129. //找到所有和结点邻接的点中权值最小的并且返回j（用来标记最小的值所在的下标） 130. **int** Minimum(Edge closedge[],**int** n){ 131. **int** i,j; 132. **int** min=INFINITY; 133. i=1; 134. j=0; 135. **for**(;i<=n;i++){ 136. **if**(closedge[i].lowcost)//从权值不为0的边中选择拥有最小权值的边 137. { 138. **if**(closedge[i].lowcost<=min) 139. { 140. min=closedge[i].lowcost; 141. j=i; 142. } 143. } 144. } 145. **return** j; 146. } 148. gEdge \*CreateEdges(MGraph G){ 149. **int** i,j; 150. **int** k=1; 151. **int** EdgeNum=G.arcnum; 152. **int** VertexNum=G.vexnum; 153. gEdge temp; 154. gEdge \*p=(gEdge\*)malloc(**sizeof**(gEdge)\*VertexNum\*VertexNum);   //之前程序报错 是因为申请的空间不够大，越界了 156. **for**(i=1;i<=VertexNum;i++) //边集数组初始化 157. **for**(j=i;j<=VertexNum;j++) //为了避免无向图的每条边被记录两次，只考虑上三角矩阵 158. **if**(G.arcs[i][j].adj!=0&&G.arcs[i][j].adj!=INFINITY){ 159. p[k].begin=i; 160. p[k].end=j; 161. p[k].weight=G.arcs[i][j].adj; 162. k++; 163. } 164. //首个p[i]与后面i+1……最后个依次比较，每一次都将小的放在第i个 165. **for**(i=1;i<EdgeNum;i++)//对边集数组进行选择排序 166. **for**(j=i+1;j<=EdgeNum;j++) 167. **if**(p[i].weight>p[j].weight){ 168. temp=p[i]; 169. p[i]=p[j]; 170. p[j]=temp; 171. } 172. **return** p; //这个时候返回的p是已经从小到大排好序的，并且拥有一共EdgeNumt个大小的数组 173. } 175. **void** MinSpanTree\_KRUSKAL(MGraph G){ 176. **int** VertexNum=G.vexnum; 177. **int** EdgeNum=G.arcnum; 178. gEdge \*p=CreateEdges(G);//边数组 179. **int** \*index=(**int**\*)malloc(**sizeof**(**int**)\*VertexNum); 180. //index 数组，其元素为连通分量的编号，index[i]=index[j]表示编号为i,j的顶点在一个连通分量里 181. **int** \*MSTEdge = (**int** \*)malloc(**sizeof**(**int**)\*VertexNum); //用来存储已经确定的MST的边的编号共VertexNum-1条边 182. **int** k=1; 183. **int** WeightSum=0; 184. **int** IndexBegin,IndexEnd; 185. **int** i,j,n; 186. **for**(i=1;i<=VertexNum;i++) //初始化所有的index=-1; 187. index[i]=-1; 188. **for**(i=1;i<VertexNum;i++) 189. { 190. **for**(j=1;j<=EdgeNum;j++) 191. { 192. **if**(!(index[p[j].begin]>=0&&index[p[j].end]>=0&&index[p[j].begin]==index[p[j].end])){//找到当前还没有加入到同一个连通分量权值最小的边 193. MSTEdge[i]=j;   //将每一个不同的弧复制给MSTEdge 一共有VertexNum条边 194. **if**((index[p[j].begin]==-1)&&(index[p[j].end]==-1)) 195. index[p[j].begin]=index[p[j].end]=i;//将两个点之间直接连通 196. **else** **if**((index[p[j].begin]==-1)&&(index[p[j].end]>=0)){ 197. index[p[j].begin]=i; 198. IndexEnd=index[p[j].end]; 199. **for**(n=1;n<=VertexNum;n++) 200. **if**(index[n]==IndexEnd) 201. index[n]=i; 202. //如果j这条弧中，有一个结点已经被连通，但是另一个结点还没有被连通，那么这个时候因为这个时候 203. //要加入这一段弧，因此相当于将这两个结点连接起来，因此我们就直接把，和另一个结点连接的所有的相同的结点都幅值为i 204. } 205. **else**  **if**((index[p[j].end]==-1)&&(index[p[j].begin]>=0)) 206. { 207. index[p[j].end]=i; 208. IndexBegin=index[p[j].begin]; 209. **for**(n=1;n<=VertexNum;n++) 210. { 211. **if**(index[n]==IndexBegin) 212. index[n]=i; 213. } 214. } 215. **else**   //也就是两个结点都被包含进去了 但是还没有连通 216. { 217. IndexEnd=index[p[j].end]; 218. IndexBegin=index[p[j].begin]; 219. **for**(n=1;n<=VertexNum;n++) 220. { 221. **if**(index[n]==IndexBegin||index[n]==IndexEnd) 222. index[n]=i; 223. } 224. } 225. **break**; 226. //里面执行完一次之后直接跳出循环，i进入下一条边，而j仍然从0开始，而之前幅值过的边不会再进去 227. //因此能够保证进去的边第一次，而对于新的边中如果两个结点已经是同一个集合中的元素，也不会再进去 228. } 229. } 230. } 232. printf("MTS's edges：\n"); 233. **for**(i=1;i<VertexNum;i++){ 234. printf("%c--%c\n",G.vexs[p[MSTEdge[i]].begin],G.vexs[p[MSTEdge[i]].end]); 235. WeightSum+=p[MSTEdge[i]].weight; 236. } 237. printf("MST's value：%d\n",WeightSum); 238. }   代码运行的结果（样例测试结果）如下图所示：    2.有向图获取最小生成树  有向图采用的测试样例如下图所示：    完成实验要求的具体代码如下所示：   1. #include<iostream> 2. #define Vnum 10 3. #define MAX 10000 4. #include<string.h> 5. #include<algorithm> 6. **using** **namespace** std; 7. **typedef** **char** Datatype; 8. **typedef** **struct** arcnode 9. { 10. **int** adjvex; 11. **int** weight; 12. **struct** arcnode \*nextarc; 13. } arcnode; 14. **typedef** **struct** 15. { 16. Datatype vexdata; 17. arcnode \*firstarc; 18. } Adjlist; 19. **typedef** **struct** 20. { 21. **int** arcnum,vexnum; 22. Adjlist adjlist[Vnum]; 23. } GraphTp; 24. **int** visit[Vnum]; 25. **int** quan[Vnum][Vnum]; 26. **void** creat(GraphTp \*g) 27. { 28. **int** i,j,k,ii,quanz; 29. **char** tailarc,headarc; 30. arcnode \*p; 31. cout<<"enter vexnum and arcnum:"; 32. cin>>g->vexnum>>g->arcnum; 33. cout<<"enter the vexdata:"; 34. **for**(i=0; i<g->vexnum; i++) 35. { 36. cin>>g->adjlist[i].vexdata; 37. g->adjlist[i].firstarc=NULL; 38. } 39. cout<<"enter the arcdata:"<<endl; 40. **for**(ii=0; ii<g->arcnum; ii++) 41. { 42. cin>>tailarc>>headarc>>quanz; 43. **for**(k=0; k<g->vexnum; k++) 44. **if**(g->adjlist[k].vexdata==tailarc) 45. { 46. i=k; 47. **break**; 48. } 49. **for**(k=0; k<g->vexnum; k++) 50. **if**(g->adjlist[k].vexdata==headarc) 51. { 52. j=k; 53. **break**; 54. } 55. p=**new** arcnode; 56. p->adjvex=j; 57. p->weight=quanz; 58. quan[i][j]=quanz; 59. quan[j][i]=quanz; 60. //        cout<<"the data is:"<<i<<' '<<j<<' '<<quanz<<endl; 61. p->nextarc=g->adjlist[i].firstarc; 62. g->adjlist[i].firstarc=p; 63. } 64. } 65. **struct** edge 66. { 67. **int** weight; 68. **int** u,v; 69. } bian[Vnum]; 70. **int** cmp(edge A,edge B) 71. { 72. **return** A.weight<B.weight; 73. } 74. **void** Kruskal(**int** n,**int** C[Vnum][Vnum])     //Kuruskral算法，顶点数n，带权临接矩阵V[n][n] 75. { 76. **int** i,j; 77. **int** nodeset[n];                       //顶点所属集合，之后用于判断顶点是否在同一暂时的树中 78. **int** count=0; 79. **bool** flag[n];                         //该数组用来判断顶点是否已经被选进生成树中 80. **for**(i=0; i<n; i++)                    //将图中所有边存入数组边中 81. { 82. nodeset[i]=i; 83. flag[i]=**false**; 84. **for**(j=0; j<n; j++) 85. **if**(C[i][j]<MAX) 86. { 87. bian[count].u=i; 88. bian[count].v=j; 89. bian[count].weight=C[i][j]; 90. count++; 91. } 92. } 93. sort(bian,bian+count,cmp);           /\*将图中各边权值升序排列\*/ 94. **int** edgeset=0; 95. **int** w=0;                             /\*用来存放生成树总权值\*/ 96. count=0; 97. **while**(edgeset<n-1)                   /\*核心算法\*/ 98. { 99. **if**(!flag[bian[count].u] && flag[bian[count].v])   /\*u未加入某一集合，v已经加入某个集合， 这个集合由已经找到的生成树顶点组成，开始可能形成多个\*/ 100. { 101. w+=bian[count].weight; 102. flag[bian[count].u]=**true**; 103. nodeset[bian[count].u]=nodeset[bian[count].v]; 104. edgeset++; 105. } 106. **else** **if**(flag[bian[count].u] && !flag[bian[count].v])/\*v未加入，u已经加入\*/ 107. { 108. w+=bian[count].weight; 109. flag[bian[count].v]=**true**; 110. nodeset[bian[count].v]=nodeset[bian[count].u]; 111. edgeset++; 112. } 113. **else** **if**(!flag[bian[count].u] && !flag[bian[count].v])            /\*两个都未加入\*/ 114. { 115. w+=bian[count].weight; 116. flag[bian[count].u]=**true**; 117. flag[bian[count].v]=**true**; 118. nodeset[bian[count].u]=nodeset[bian[count].v]; 119. edgeset++; 120. } 121. **else** **if**(nodeset[bian[count].u]!=nodeset[bian[count].v])/\*如果两个加入集合，判断加入集合是否相同，相同说明形成环，跳出此次循环，转向下一次。\*/ 122. {                                                      /\*否则将两个集合合并\*/ 123. w+=bian[count].weight; 124. edgeset++; 125. **int** tmp=nodeset[bian[count].v]; 126. **for**(i=0;i<n;i++) 127. **if**(nodeset[i]=tmp) 128. nodeset[i]=nodeset[bian[count].u]; 129. } 130. count++;                          /\*转向次小边，进行下一次判断\*/ 131. } 132. cout<<"The weight of the minimum spanning tree is："<<w;              /\*输出权值\*/ 133. } 134. **int** main() 135. { 136. **int** i,j; 137. **for**(i=0; i<Vnum; i++) 138. **for**(j=0; j<Vnum; j++) 139. quan[i][j]=MAX; 140. GraphTp g; 141. creat(&g); 142. cout<<endl; 143. Kruskal(g.vexnum,quan); 144. **return** 0; 145. }   代码运行的结果如下图所示： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# Dijkstra算法（单源最短路）

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用Dijkstra算法基于图的两种存储形式（邻接表和邻接矩阵）实现单源最短路径权值计算。  本次实验主要由以下两个程序组成：  **1.无向图计算最短路（使用邻接矩阵存储有向图结构）**  **2. 有向图计算最短路（使用邻接表存储有向图结构）**  本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 最短路径问题是图论研究中的一个经典算法问题， 旨在寻找图（由结点和路径组成的）中两结点之间的最短路径。 算法具体的形式包括：1）确定起点的最短路径问题 - 即已知起始结点，求最短路径的问题；2）确定终点的最短路径问题 - 与确定起点的问题相反，该问题是已知终结结点，求最短路径的问题。在无向图中该问题与确定起点的问题完全等同，在有向图中该问题等同于把所有路径方向反转的确定起点的问题；3）确定起点终点的最短路径问题 - 即已知起点和终点，求两结点之间的最短路径；4）全局最短路径问题 - 求图中所有的最短路径。  DIjkstra算法是指从一个顶点到其余各顶点的最短路径算法，解决的是有权图中最短路径问题。其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。  其算法思想可表示为以下三点：1）在任意时刻，我们都要得到从源点到所有顶点的估算距离，并维持一个顶点集合S，若顶点v在S中，则说明从源点到v的最短路径已知；2）在每一次将不在S中的顶点v加到S中去时，总是选择从源点到v的估算距离最小的；3）顶点v加入S中之后，对于所有与v相邻的顶点（不属于S），更新它们的估算距离。  伪代码如下：   1. Dijkstra(G, W, s)　　　　　　//G表示图，W表示权值函数，s表示源顶点 2. d[s] ←0　　　　　　　　　　//源点到源点最短路为0 3. **for** each v ∈ V - {s}　　//3-8行均为初始化操作 4. **do** d[v]←∞ 5. parent[v]←NIL 6. S←∅ 7. Q←V　　　　　　　　//此处Q为优先队列，存储未进入S的各顶点以及从源点到这些顶点的估算距离，采用二叉堆（最小堆）实现，越小越优先 8. **while** Q≠∅ 9. **do** u←Extract-Min(Q)　　//提取估算距离最小的顶点，在优先队列中位于顶部，出队列，放入集合S中 10. S←S∪{u} 11. **for** each v ∈ Adj(u)　　//松弛操作，对与u相邻的每个顶点v，进行维持三角不等式成立的松弛操作。 12. **do** **if** d[v] > d[u] + w(u, v) 13. then d[v] = d[u] + w(u, v)　　//这一步隐含了更新优先队列中的值，DECREASE。 14. parent[v]←u　　　　　　//置v的前驱结点为u   1.无向图计算最短路  无向图采用的测试样例如下图所示：    完成实验要求的具体代码如下所示：   1. #include <iostream> 2. #include <cstdio> 3. #include <queue> 4. **using** **namespace** std; 6. #define maxn 110        //最大顶点个数 7. #define INF 0xffffff    //权值上限 8. **int** w[maxn][maxn];      //邻接矩阵，存储权值 9. **int** n;                  //顶点个数 11. **struct** node             //顶点节点，保存id和到源顶点的估算距离，优先队列需要的类型 12. { 13. **int** id, weight;     //源顶点id和估算距离 14. **friend** **bool** operator<(node a, node b)   //因要实现最小堆，按升序排列，因而需要重载运算符，重定义优先级，以小为先 15. { 16. **return** a.weight > b.weight; 17. } 18. }; 19. priority\_queue<node> q;     //优先队列，最小堆，实现Dijkstra的重要数据结构，用stl实现 20. **int** parent[maxn];           //每个顶点的父亲节点，可以用于还原最短路径树 21. **bool** visited[maxn];         //用于判断顶点是否已经在最短路径树中，或者说是否已找到最短路径 22. node d[maxn];               //源点到每个顶点估算距离，最后结果为源点到所有顶点的最短路。 23. **void** Dijkstra(**int** s)        //Dijkstra算法，传入源顶点 24. { 25. **for**(**int** i = 1; i <= n; i++)     //初始化 26. { 27. d[i].id = i; 28. d[i].weight = INF;          //估算距离置INF 29. parent[i] = -1;             //每个顶点都无父亲节点 30. visited[i] = **false**; 31. } 32. d[s].weight = 0;                //源点到源点最短路权值为0 33. q.push(d[s]);                   //压入队列中 34. **while**(!q.empty())               //算法的核心，队列空说明完成了操作 35. { 36. node cd = q.top();          //取最小估算距离顶点 37. q.pop(); 38. **int** u = cd.id; 39. **if**(visited[u]) 40. **continue**; 41. visited[u] = **true**; 42. //松弛操作 43. **for**(**int** v = 1; v <= n; v++) //找所有与他相邻的顶点，进行松弛操作，更新估算距离，压入队列。 44. { 45. **if**(v != u && !visited[v] && d[v].weight > d[u].weight+w[u][v]) 46. { 47. d[v].weight = d[u].weight+w[u][v]; 48. parent[v] = u; 49. q.push(d[v]); 50. } 51. } 52. } 53. } 54. **int** main() 55. { 56. **int** m, a, b, c, st, ed; 57. printf("ALL IS RIGHT ~\n"); 58. printf("Please enter the number of nodes & edges: "); 59. scanf("%d %d", &n, &m); 60. printf("Please enter the edge and weight: \n"); 61. **for**(**int** i = 1; i <= n; i++)     //邻接矩阵存储前需要初始化 62. **for**(**int** j = i; j <= n; j++) 63. w[i][j] = w[j][i] = INF; 64. **while**(m--) 65. { 66. scanf("%d %d %d", &a, &b, &c); 67. **if**(w[a][b] > c) 68. w[a][b]= w[b][a] = c; 69. } 70. printf("Please enter start and end points:\n"); 71. scanf("%d %d", &st, &ed); 72. Dijkstra(st); 73. **if**(d[ed].weight != INF) 74. printf("The shortest path weight is: %d\n", d[ed].weight); 75. **else** 76. printf("There is no shortest path from %d to %d.\n", st, ed); 77. **return** 0; 78. }   代码运行的结果如下图所示：    2.有向图计算最短路  有向图采用的测试样例如下图所示：    完成实验要求的具体代码如下所示：   1. #include <iostream> 2. #include <cstdio> 3. #include <vector> 4. #include <queue> 5. **using** **namespace** std; 7. #define maxn 110  //最大顶点个数 8. **int** n;       //顶点个数 10. **struct** arcnode  //边结点 11. { 12. **int** vertex;     //与表头结点相邻的顶点编号 13. **int** weight;     //连接两顶点的边的权值 14. arcnode \* next; //指向下一相邻接点 15. arcnode() {} 16. arcnode(**int** v,**int** w):vertex(v),weight(w),next(NULL) {} 17. }; 19. **struct** vernode      //顶点结点，为每一条邻接表的表头结点 20. { 21. **int** vex;    //当前定点编号 22. arcnode \* firarc;   //与该顶点相连的第一个顶点组成的边 23. }Ver[maxn]; 25. **void** Init()  //建立图的邻接表需要先初始化，建立顶点结点 26. { 27. **for**(**int** i = 1; i <= n; i++) 28. { 29. Ver[i].vex = i; 30. Ver[i].firarc = NULL; 31. } 32. } 34. **void** Insert(**int** a, **int** b, **int** w)  //尾插法，插入以a为起点，b为终点，权为w的边，效率不如头插，但是可以去重边 35. { 36. arcnode \* q = **new** arcnode(b, w); 37. **if**(Ver[a].firarc == NULL) 38. Ver[a].firarc = q; 39. **else** 40. { 41. arcnode \* p = Ver[a].firarc; 42. **if**(p->vertex == b) 43. { 44. **if**(p->weight > w) 45. p->weight = w; 46. **return** ; 47. } 48. **while**(p->next != NULL) 49. { 50. **if**(p->next->vertex == b) 51. { 52. **if**(p->next->weight > w) 53. p->next->weight = w; 54. **return** ; 55. } 56. p = p->next; 57. } 58. p->next = q; 59. } 60. } 61. **void** Insert2(**int** a, **int** b, **int** w)   //头插法，效率更高，但不能去重边 62. { 63. arcnode \* q = **new** arcnode(b, w); 64. **if**(Ver[a].firarc == NULL) 65. Ver[a].firarc = q; 66. **else** 67. { 68. arcnode \* p = Ver[a].firarc; 69. q->next = p; 70. Ver[a].firarc = q; 71. } 72. } 73. **struct** node     //顶点节点，保存id和到源顶点的估算距离，优先队列需要的类型 74. { 75. **int** id;     //源顶点id和估算距离 76. **int** w; 77. **friend** **bool** operator<(node a, node b)   //因要实现最小堆，按升序排列，因而需要重载运算符，重定义优先级，以小为先 78. { 79. **return** a.w > b.w; 80. } 81. }; 83. #define INF 0xfffff    //权值上限 84. **int** parent[maxn];   //每个顶点的父亲节点，可以用于还原最短路径树 85. **bool** visited[maxn]; //用于判断顶点是否已经在最短路径树中，或者说是否已找到最短路径 86. node d[maxn];      //源点到每个顶点估算距离，最后结果为源点到所有顶点的最短路。 87. priority\_queue<node> q; //优先队列stl实现 88. **void** Dijkstra(**int** s)    //Dijkstra算法，传入源顶点 89. { 90. **for**(**int** i = 1; i <= n; i++) //初始化 91. { 92. d[i].id = i; 93. d[i].w = INF;           //估算距离置INF 94. parent[i] = -1;         //每个顶点都无父亲节点 95. visited[i] = **false**;     //都未找到最短路 96. } 97. d[s].w = 0;                 //源点到源点最短路权值为0 98. q.push(d[s]);               //压入队列中 99. **while**(!q.empty())           //算法的核心，队列空说明完成了操作 100. { 101. node cd = q.top();      //取最小估算距离顶点 102. q.pop(); 103. **int** u = cd.id; 104. **if**(visited[u])   //注意这一句的深意，避免很多不必要的操作 105. **continue**; 106. visited[u] = **true**; 107. arcnode \* p = Ver[u].firarc; 108. //松弛操作 109. **while**(p != NULL)    //找所有与他相邻的顶点，进行松弛操作，更新估算距离，压入队列。 110. { 111. **int** v = p->vertex; 112. **if**(!visited[v] && d[v].w > d[u].w+p->weight) 113. { 114. d[v].w = d[u].w+p->weight; 115. parent[v] = u; 116. q.push(d[v]); 117. } 118. p = p->next; 119. } 120. } 121. } 123. **int** main() 124. { 125. **int** m, a, b, c, st, ed; 126. printf("ALL IS RIGHT ~\n"); 127. printf("Please enter the number of nodes & edges: "); 128. scanf("%d %d", &n, &m); 129. printf("Please enter the edge and weight: \n"); 130. Init();     //计算前必须初始化 131. **while**(m--) 132. { 133. scanf("%d %d %d", &a, &b, &c); 134. Insert2(a, b, c); 135. } 136. printf("Please enter start and end points:\n"); 137. scanf("%d%d", &st, &ed); 138. Dijkstra(st); 139. **if**(d[ed].w != INF) 140. printf("The shortest path weight is: %d\n", d[ed].w); 141. **else** 142. printf("There is no shortest path from %d to %d.\n", st, ed); 143. **return** 0; 144. }   代码运行的结果如下图所示： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 排序二叉树

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求实现对二叉排序树进行基本操作，包括构建、查找、插入、删除等。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 二叉排序树（Binary Sort Tree），又称二叉查找树（Binary Search Tree），亦称二叉搜索树。  二叉排序树的定义如下：  一棵空树，或者是具有下列性质的二叉树：  （1）若左子树不空，则左子树上所有结点的值均小于它的根结点的值；  （2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值；  （3）左、右子树也分别为二叉排序树；  （4）没有键值相等的结点。  二次排序树查找的步骤如下：  若根结点的关键字值等于查找的关键字，成功。  否则，若小于根结点的关键字值，递归查左子树。  若大于根结点的关键字值，递归查右子树。  若子树为空，查找不成功。  平均情况分析（在成功查找两种的情况下）：    在一般情况下，设 P（n，i）为它的左子树的结点个数为 i 时的平均查找长度。如图的结点个数为 n = 6 且 i = 3; 则 P（n,i）= P（6, 3） = [ 1+ ( P(3) + 1) \* 3 + ( P(2) + 1) \* 2 ] / 6= [ 1+ ( 5/3 + 1) \* 3 + ( 3/2 + 1) \* 2 ] / 6  注意：这里 P(3)、P(2) 是具有 3 个结点、2 个结点的二叉分类树的平均查找长度。 在一般情况，P（i）为具有 i 个结点二叉分类树的平均查找长度。平均查找长度= 每个结点的深度的总和 / 总结点数  P(3) = （1+2+2）/ 3 = 5/3  P(2) = （1+2）/ 2 = 3/2∴ P（n,i）= [ 1+ ( P(i) + 1) \* i + ( P(n-i-1) + 1) \* (n-i-1) ] / n    因为 2(1+I/n)lnn≈1.38logn 故P(n)=O(logn)  二叉排序树的插入与删除操作。  与次优二叉树相对，二叉排序树是一种动态树表。其特点是：树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。新插入的结点一定是一个新添加的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。  完成本次实验操作的具体代码如下：   1. #include <stdio.h> 2. #include <stdlib.h> 4. #define TRUE     1 5. #define FALSE    0 7. **typedef** **struct** BiNode 8. { 9. **int** data; 10. **struct** BiNode \*lchild, \*rchild; 11. }BiNode, \*BiTree; 13. **int** SearchBST (BiTree T, **int** key, BiTree f, BiTree \*p) 14. { 15. **if**(!T) 16. { 17. \*p = f; 18. **return** FALSE; 19. } 20. **else** **if**(T->data == key) 21. { 22. \*p = T; 23. **return** TRUE; 24. } 25. **else** **if** (key < T->data) 26. { 27. SearchBST(T->lchild, key, T, p);      //递归寻找，根据二叉排序树的性质 28. } 29. **else** 30. { 31. SearchBST(T->rchild, key, T, p); 32. } 33. } 35. **void** InsertBST (**int** key, BiTree \*T) 36. { 37. BiTree p, s; 38. **if**(!SearchBST(\*T, key, NULL, &s))                  //查找二叉排序树中是否已经存在key值 39. { 40. p = (BiTree)malloc(**sizeof**(BiNode)); 41. p->data = key; 42. p->lchild = p->rchild = NULL; 43. **if**(!s)                                         //若二叉排序树为空，则插入根节点 44. { 45. \*T = p; 46. } 47. **else** **if**(key < s->data) 48. { 49. s->lchild = p; 50. } 51. **else** 52. { 53. s->rchild = p; 54. } 55. } 56. } 58. **void** CreatBST (BiTree \*T, **int** \*arry) 59. { 60. **int** i; 61. **for**(i = 0; i < 10; i++) 62. { 63. InsertBST(arry[i], T); 64. } 65. } 67. **void** PreOrderTree (BiTree T) 68. { 69. **if**(!T) 70. { 71. **return**; 72. } 73. printf("%d ", T->data); 74. PreOrderTree(T->lchild); 75. PreOrderTree(T->rchild); 76. } 78. **void** Delete (**int** key, BiTree \*p)                      //删除查找的结点 79. { 80. BiTree q, s; 81. **if**(!(\*p)->rchild) 82. { 83. q = \*p; 84. \*p = (\*p)->lchild; 85. free(q); 86. } 87. **else** **if**(!(\*p)->lchild) 88. { 89. q = \*p; 90. \*p = (\*p)->rchild; 91. free(q); 92. } 93. **else** 94. { 95. q = \*p; 96. s = q->lchild; 97. **while**(s->rchild) 98. { 99. q = s; 100. s = s->rchild; 101. } 102. (\*p)->data = s->data; 103. **if**(q != \*p) 104. { 105. q->rchild = s->lchild; 106. } 107. **else** 108. { 109. (\*p)->lchild = s->lchild; 110. } 111. free(s); 112. } 113. } 115. **void** DeletrBST (BiTree \*T, **int** key) 116. { 117. **if**(!\*T) 118. { 119. **return**; 120. } 121. **else** **if**(key == (\*T)->data) 122. { 123. Delete(key, T); 124. } 125. **else** **if**(key < (\*T)->data) 126. { 127. DeletrBST(&((\*T)->lchild), key); 128. } 129. **else** 130. { 131. DeletrBST(&((\*T)->rchild), key); 132. } 133. } 135. **int** main () 136. { 137. BiTree T = NULL; 138. **int** arry[100], i, key, n; 139. printf("Please enter the number of nodes :"); 140. scanf("%d",&n); 141. printf("Please enter the node number in turn :\n"); 142. **for**(i = 0; i < n; i++) 143. { 144. scanf("%d", &(arry[i])); 145. } 146. CreatBST(&T, arry); 148. printf("Preorder traversal results ："); 149. PreOrderTree(T); 150. printf("\nEnter delete num:\n"); 151. scanf("%d", &key); 152. DeletrBST(&T, key); 154. printf("Preorder traversal results ："); 155. PreOrderTree(T); 156. **return** 0; 157. }   本次实验使用的测试样例如下：   1. //9 2. //9 5 10 0 6 11 -1 1 2 3. //10   本次实验预计生成的二叉树如下表示：    本次实验测试样例的结果如下: |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 二叉排序树平衡化

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求将二叉排序树进行平衡化操作，将其转换成平衡二叉排序树并利用前序遍历打印输出。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 平衡二叉搜索树（英语：Balanced Binary Tree）是一种结构平衡的二叉搜索树，即叶节点高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。它能在O()内完成插入、查找和删除操作，最早被发明的平衡二叉搜索树为AVL树。  一棵AVL树有如下必要条件：  1.它必须是二叉查找树。  2.每个节点的左子树和右子树的高度差至多为1。  平衡因子（Balance Factor）指 某个节点 左右子树的高度差。  添加节点导致的失衡以及解决办法：  向AVL树中添加节点时（只会在叶子节点添加），可能会导致所有祖先节点都失衡，但是父节点、非祖父节点都是不可能失衡的。  其操作可具体分为：  LL ----- 右旋转    RR ----- 左旋转    LR ----- LR 旋转    RL ----- RL旋转: 与LR 旋转相似，先将p右旋转，再将g左旋转……  完成本次实验的具体代码如下：   1. #include<stdio.h> 2. #include<stdlib.h> 3. #include<bits/stdc++.h> 4. #include<iostream> 5. **using** **namespace** std; 6. **typedef** **struct** Node 7. { 8. **int** key; 9. **struct** Node \*left; 10. **struct** Node \*right; 11. **int** height; 12. }BTNode; 14. **int** height(**struct** Node \*N) 15. { 16. **if** (N == NULL) 17. **return** 0; 18. **return** N->height; 19. } 21. **int** max(**int** a, **int** b) 22. { 23. **return** (a > b) ? a : b; 24. } 26. BTNode\* newNode(**int** key) 27. { 28. **struct** Node\* node = (BTNode\*)malloc(**sizeof**(**struct** Node)); 29. node->key = key; 30. node->left = NULL; 31. node->right = NULL; 32. node->height = 1; 33. **return**(node); 34. } 36. BTNode\* ll\_rotate(BTNode\* y) 37. { 38. BTNode \*x = y->left; 39. y->left = x->right; 40. x->right = y; 41. y->height = max(height(y->left), height(y->right)) + 1; 42. x->height = max(height(x->left), height(x->right)) + 1; 43. **return** x; 44. } 46. BTNode\* rr\_rotate(BTNode\* y) 47. { 48. BTNode \*x = y->right; 49. y->right = x->left; 50. x->left = y; 51. y->height = max(height(y->left), height(y->right)) + 1; 52. x->height = max(height(x->left), height(x->right)) + 1; 53. **return** x; 54. } 56. **int** getBalance(BTNode\* N) 57. { 58. **if** (N == NULL) 59. **return** 0; 60. **return** height(N->left) - height(N->right); 61. } 63. BTNode\* insert(BTNode\* node, **int** key) 64. { 66. **if** (node == NULL) 67. **return** newNode(key); 68. **if** (key < node->key) 69. node->left = insert(node->left, key); 70. **else** **if** (key > node->key) 71. node->right = insert(node->right, key); 72. **else** 73. **return** node; 74. node->height = 1 + max(height(node->left), height(node->right)); 75. **int** balance = getBalance(node); 76. **if** (balance > 1 && key < node->left->key) //LL型 77. **return** ll\_rotate(node); 78. **if** (balance < -1 && key > node->right->key)     //RR型 79. **return** rr\_rotate(node); 80. **if** (balance > 1 && key > node->left->key)     //LR型 81. { 82. node->left = rr\_rotate(node->left); 83. **return** ll\_rotate(node); 84. } 85. **if** (balance < -1 && key < node->right->key)     //RL型 86. { 87. node->right = ll\_rotate(node->right); 88. **return** rr\_rotate(node); 89. } 90. **return** node; 91. }  94. **void** preOrder(**struct** Node \*root) 95. { 96. **if** (root != NULL) 97. { 98. printf("%d ", root->key); 99. preOrder(root->left); 100. preOrder(root->right); 101. } 102. } 104. **int** xx[100]={0}; 106. **int** main() 107. { 108. BTNode \*root = NULL; 109. **int** ax[100]={0}; 110. //  9 5 10 0 6 11 -1 1 2 111. **int** n; 112. printf("Please enter the number of nodes :"); 113. scanf("%d",&n); 114. printf("Please enter the node number in turn :\n"); 115. **for**(**int** i = 0; i < n; i++) 116. { 117. scanf("%d", &(ax[i])); 118. root = insert(root,ax[i]); 119. } 121. printf("Preorder traversal results ："); 122. //    9 1 0 -1 5 2 6 10 11 123. preOrder(root); 124. **return** 0; 125. }   本次实验使用的测试样例如下:   1. //9 2. //9 1 0 -1 5 2 6 10 11   本次实验生成的二叉排序树如下：    本次实验测试所得的结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 基于顺序表的二分查找实现

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用顺序表对数据进行二分查找并输出查找结果（找得到输出数据下标，到不到输出提示信息），本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。  二分查找的查找过程：首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。  二分查找的算法要求有如下两点。  1.必须采用顺序存储结构。  2.必须按关键字大小有序排列。  本次实验使用顺序表的存储结构，利用了二分查找的算法思想对用户输入的数据进行查找并反馈相关信息。  完成本次实验所写的具体代码如下：   1. #include<cstdio> 2. #include<cstdlib> 3. #include<cstring> 4. #include<cmath> 5. #include<algorithm> 6. #include<iostream> 7. #define MaxSize 100 8. #define ElemType int 9. #define Status int 10. **using** **namespace** std; 11. //顺序表数据结构 12. **typedef** **struct** 13. { 14. ElemType data[MaxSize];//顺序表元素 15. **int** length;            //顺序表当前长度 16. }SqList; 17. //初始化顺序表函数，构造一个空的顺序表 18. Status InitList(SqList &L) 19. { 20. memset(L.data,0,**sizeof**(L));//初始化数据为0 21. L.length=0;                //初始化长度为0 22. **return** 0; 23. } 24. //创建顺序表函数 初始化前n个数据 25. **bool** CreatList(SqList &L,**int** n) 26. { 27. **if**(n<0||n>MaxSize)**false**;//n非法 28. **for**(**int** i=0;i<n;i++) 29. { 30. scanf("%d",&L.data[i]); 31. L.length++; 32. } 33. **return** **true**; 34. } 35. //插入函数 位置i插入数据 i及之后元素后移  1=<i<=length+1 37. //二分查找函数 38. **int** Binary\_search(SqList L,ElemType key) 39. { 40. **int** low = 0;**int** mid = 0;**int** high = L.length-1; 41. **while**(low<=high) 42. { 43. mid = (low+high)/2; 44. **if**(key==L.data[mid]) 45. { 46. **return** mid; 47. } 48. **else** **if**(key>L.data[mid]) 49. { 50. low = mid+1; 51. } 52. **else** 53. { 54. high = mid-1; 55. } 56. } 57. **return** -1; 58. } 59. //输出功能函数 按位置从小到大输出顺序表所有元素 60. **void** PrintList(SqList L) 61. { 62. printf("All elements of the current sequence table :"); 63. **for**(**int** i=0;i<L.length;i++) 64. { 65. printf("%d ",L.data[i]); 66. } 67. printf("\n"); 68. } 69. //创建顺序表函数 70. **void** Create(SqList &L) 71. { 72. **int** n;**bool** flag; 73. printf("Please enter the length of the order table to create(>1) :"); 74. scanf("%d",&n); 75. printf("Please enter %d number (space separated):",n); 76. flag=CreatList(L,n); 77. **if**(flag){ 78. printf("Created successfully!\n"); 79. PrintList(L); 80. } 81. **else** printf("ILLEGAL INPUT!\n"); 83. } 84. //简单选择排序功能函数 为二分做准备 85. **void** SelectSort(SqList &L) 86. { 87. **int** min;**int** temp; 88. **for**(**int** i=0;i<L.length;i++) 89. { 90. min=i; 91. **for**(**int** j=i+1;j<L.length;j++) 92. { 93. **if**(L.data[j]<L.data[min])min=j; 94. } 95. **if**(min!=i) 96. { 97. temp=L.data[min]; 98. L.data[min]=L.data[i]; 99. L.data[i]=temp; 100. } 101. } 102. } 103. //二分查找功能函数 调用Binary\_search 104. **void** Binary(SqList L) 105. { 106. **int** key;**int** place; 107. SelectSort(L);       //二分查找前先排序 108. PrintList(L); 109. printf("Please enter the value to find ：\n"); 110. scanf("%d",&key); 111. place=Binary\_search(L,key); 112. **if**(place>=0)printf("Index (from 0) is ：%d\n",place); 113. **else** printf("NOT FOUND!\n"); 114. } 115. **int** main() 116. { 117. SqList L;**int** choice; 118. InitList(L); 119. Create(L); 120. printf("START BINARY-FOUND  ↓↓\n"); 121. Binary(L); 123. **return** 0; 124. }   本次实验使用的具体样例如下：   1. //10 2. //18 5 27 13 57 36 9 6 62 77 3. //36   本次实验样例测试完成的结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 索引查找的构建和实现

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用单链表构建和实现对数据的索引查找，本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 数据结构的最终目的就是提高数据的处理速度，索引是为了加快查找速度而设计的一种数据结构。索引就是把一个关键字与他对应的记录相关联的过程  一个索引由若干个索引项构成，每个索引项至少应包含关键字和其对应的记录在存储器中的位置等信息。  索引技术是组织大型数据库以及磁盘文件的一种重要技术。  索引按照结构可以分为线性索引，树形索引和多级索引。  所谓线性索引就是将索引项集合组织为线性结构，也称为索引表。  线性索引有三种索引方式，稠密索引，分块索引和倒排索引。本次实验使用的是分块索引。在这次实验中，我对单链表的数据集进行分块，，使其分块有序，然后再对每一块建立一个索引项（类似于图书馆的分块）。分块有序是把数据集的记录分成了若干块，并且这些块需要满足两个条件：  （1）块内无序  每一块内的记录不要求有序  （2）块间有序  比如要求第二块所以记录的关键字均要大于第一块中所有记录的关键字，第三块要大于第二块。  只有块间有序才有可能在查找时带来效率。对于分块有序的数据集，将每块对应一个索引项，这种索引方法叫做分块索引。  分块索引的索引项结构分为三个数据项：  a: 最大关键码--存储每一块中的最大关键字。  b: 存储每一块中记录的个数以便于循环时使用。  c: 用于指向块首数据元素的指针，便于开始对这一块中记录进行遍历。  分块索引的具体索引项见下图所示：    完成本次实验的具体代码如下：   1. #include <stdio.h> 2. #include <stdlib.h> 4. #define MAXSIZE 100 6. **typedef** **struct** \_blockIndex 7. { 8. **int** key; 9. **int** len; 10. **int** \*first; 11. }blockInfo,BlockIndex[MAXSIZE]; 13. **int** Binary\_Search(BlockIndex B, **int** key, **int** n,**int** \*loc) 14. { 15. **int** low, high, mid; 16. low = 0; 17. high = n-1; 18. **while** (low<=high) 19. { 20. mid = (high + low) / 2; 21. **if** (B[mid].key == key) 22. **return** mid; 23. **else** **if** (B[mid].key < key) 24. low = mid + 1; 25. **else** 26. high = mid - 1; 27. } 28. **if** (B[mid].key < key&&key < B[mid + 1].key) 29. \*loc = mid + 1; 30. **else** 31. \*loc = mid; 32. **return** -1; 33. } 35. **int** main() 36. { 37. **int** aa[1010]={0}; 38. **int** num; 39. printf("Please input the number of numbers to be found :"); 40. scanf("%d",&num); 41. printf("Please input numbers in turn :"); 42. **for**(**int** i=0;i<num;i++) 43. { 44. scanf("%d",&aa[i]); 45. } 46. BlockIndex BI; 47. **int** n,loc,ret; 48. BI[0].key = aa[num/3]; 49. BI[0].len = num/3+1; 50. BI[0].first = aa; 52. BI[1].key = aa[num/3+2]; 53. BI[1].len = num/3-1; 54. BI[1].first = aa+(num/3+1); 56. BI[2].key = aa[num/3\*2+1]; 57. BI[2].len = num/3; 58. BI[2].first = aa + (num/3\*2); 59. printf("input number to search :\n"); 60. scanf("%d", &n); 61. ret = Binary\_Search(BI, n, 3, &loc); 62. **if** (ret != -1) 63. loc = ret; 64. **for** (**int** i = BI[loc].first - aa; i < BI[loc].first - aa + BI[loc].len; i++) 65. **if** (aa[i] == n) 66. { 67. printf("find index : %d", i); 68. **break**; 69. } 70. **return** 0; 71. }   本次实验使用的测试数据如下：   1. //9 2. //18 5 27 13 57 36 96 62 77 3. //36   本实验测试完成的结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 二叉排序树平衡化

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求利用散列的除留余数法和链接地址法处理散列冲突。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 已知的几种查找方法有：  （1）顺序查找：O（n）；  （2）二分查找（静态查找）：O（logn）；  （3）二叉搜索树：O（h）（h为二叉查找树高度）；  （4）平衡二叉树：O（logn）。  查找的本质：已知对象找位置：  （1）通过有序的安排对象：全序（顺序），半序（查找树，左子树小于根结点，右子树大于根结点）；  （2）直接算出对象位置：散列表。  散列查找的两项基本工作：  （1）计算位置：构造散列函数确定关键词存储位置；  （2）解决冲突：应用某种策略解决多个关键词位置相同的问题。  散列表（哈希表）：（根据关键字直接进行访问的数据结构）  （1）类型名称：符号表（SymbolTable）；  （2）数据对象集：符号表是“名字（Name）——属性（Attribute）”对的集合。  装填因子（Loading Factor）：设散列表空间大小为m，填入表中的元素个数是n，则n/m为散列表的装填因子。  散列查找的时间复杂度几乎是O（1），查找规模与问题的规模n无关。  散列（Hashing）的基本思想是：  （1）以关键字key为自变量，通过一个确定的函数h（散列函数），计算出对应的函数值h（key），作为数据对象的存储地址；  （2）可能不同的关键字会映射到同一个散列地址上，即h（key i）= h（key j），但两个关键字并不相等，称为冲突（collision）（不可避免但要减少），把具有相同函数值的关键字称为同义词。  散列函数：建立表中关键字到散列地址的映射关系。即hash（key） = address。  散列函数的构造方法：  （1）散列函数应尽可能计算简单，以便提高转换速度；  （2）关键字对应的地址空间应均匀分布，而减少冲突的发生；  （3）散列函数的定义域应包含所有要存储的的关键字。  本次实验采用散列函数构造方法是除留余数法：h(key) = key mod p（p一般取素数，p不大于散列表的长度TableSize，一般选大于等于TableSize的最小的素数）。  字符串为关键字的散列函数构造：  （1）ASCII码加和法，容易产生冲突；  （2）前三个字符移位法，仍然有冲突，且空间效率不高；  （3）好的散列函数是移位法中涉及所有n个字符并且分布的很好。  影响散列查找时间的因素：  （1）散列函数执行速度（即计算散列值所需的时间）；  （2）关键字的长度；  （3）散列表的大小；  （4）关键字的分布情况；  （5）不同关键字的查找频率。  本次实验采用的冲突处理方法是链地址法。链地址法（分离链接法）（Separate Chaining）：把相同位置上冲突的所有关键字存储在同一个单链表中。  散列表性能分析：  平均查找长度（ASL）用来衡量散列表查找效率：  （1）ASL s：成功平均查找长度；  （2）ASL u：不成功平均查找长度。  关键字的比较次数，取决于产生冲突的多少。  影响产生冲突多少的因素：  （1）散列函数是否平均；  （2）处理冲突的方法；  （3）散列表的装填因子。  当装填因子小于0.5时，各种探测法的探测次数都不大，也比较接近，合理的最大装填因子应不超过0.85.  （3）链地址法期望探测次数：  链地址法的装填因子定义为所有地址链表的平均长度（可能会超过1）。  对于链地址法来说，散列表是顺序存储和链式存储的结合，链表部分的存储效率和查找效率都比较低。而且关键字的删除不需要懒惰删除，所以不会存储垃圾。填充因子太小会导致空间浪费，填充因子太大会导致付出更多的时间代价，不均匀的链表长度导致时间效率的严重下降。  本次实验所写的完整代码如下：   1. #include<bits/stdc++.h> 2. **using** **namespace** std; 4. **typedef** **struct** node 5. { 6. **int** data; 7. **struct** node \*next; 8. }Node; 9. //12 10. //12 24 48 13 26 14 15 16 17 19 20 21 11. //24 12. **int** a[1010] = {0}; 13. Node b[1010]; 14. **int** n; 16. **int** f(**int** x) 17. { 18. **return** x%n; 19. } 21. **void** Build() 22. { 23. cout<<"Input n : "; 24. cin>>n; 25. cout<<"Input numbers in turn:"; 26. **for**(**int** i = 0;i < n;i++) 27. { 28. cin>>a[i]; 29. } 30. **for**(**int** i = 0;i < n;i++) 31. { 32. b[i].next = NULL; 33. } 34. **for**(**int** i = 0;i < n;i++) 35. { 36. **int** x = f(a[i]); 37. Node \* newnode = (Node\*)malloc(**sizeof**(Node)); 38. newnode->data = a[i]; 39. newnode->next = b[x].next; 40. b[x].next = newnode; 41. } 42. } 44. **void** Display() 45. { 46. cout<<"HASH TABLE ："<<endl; 47. **for**(**int** i = 0;i < 12;i++) 48. { 49. cout<<i<<":  "; 50. Node \*s = b[i].next; 51. **while**(s!=NULL) 52. { 53. cout<<s->data<<" "; 54. s = s->next; 55. } 56. cout<<endl; 57. } 58. } 60. **void** Find() 61. { 62. **int** x; 63. cout<<"Please enter the number to find : "; 64. cin>>x; 65. **int** y = f(x); 66. Node \* s = b[y].next; 67. **int** num = 1; 68. **while**(s) 69. { 70. **if**(s->data == x) 71. { 72. cout<<"EXISTENT ~"<<endl; 73. cout<<"On line "<<y<<"，no."<<num<<endl; 74. **return**; 75. } 76. num++; 77. s = s->next; 78. } 79. cout<<"NON-EXISTENT !"; 80. } 82. **int** main() 83. { 84. Build(); 85. Display(); 86. Find(); 87. }   本次实验使用的测试样例如下：   1. //12 2. //12 24 48 13 26 14 15 16 17 19 20 21 3. //24   本次实验的测试结果如下图： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |

# 八大排序结构算法

|  |
| --- |
| **一、实验（实训）概述（目的及要求，基本原理，实施环境、使用的材料、设备、软件等）（可加页）** |
| 实验要求学习八大排序算法，分析其对应的时间复杂度和空间复杂度，并结合样例验证排序算法的正确性。本次实验所用的编译器为DEV-C++，使用语言为C++(14)，使用设备为DELL笔记本。 |
| **二、实验（实训）内容（项目内容，方案设计，实验（实训）过程步骤、记录、数据、程序，结果与分析等）（可加页）** |
| 本文主要介绍数据结构中常见的八大排序算法，冒泡排序、快速排序、直接插入排序、希尔排序、简单选择排序、堆排序、归并排序和基数排序。  排序相描述  排序分类：若排序过程中，所有的文件都是放在内存中处理的，不涉及数据的内外存交换，则称该排序算法是内部排序算法; 若排序过程中涉及内外存交换，则是外部排序。内部排序适合小文间，外部排序适用于不能一次性把所有记录放入内存的大文件。常见的分类算法还可以根据排序方式分为两大类：比较排序和非比较排序。  排序算法的稳定性： 若排序对象中存在多个关键字相同的记录，经过排序后，相同关键字的记录之间的相对次序保持不变，则该排序方法是稳定的，若次序发生变化(哪怕只有两条记录之间)，则该排序方法是不稳定的。关于哪些算法是稳定的，哪些不稳定，下面会详细介绍。  时间复杂度：一般情况下，算法中基本操作重复执行的次数是问题规模nnn的某个函数，用T(n)表示，若有某个辅助函数f(n)，使得T(n)/f(n))的极限值（当n趋近于无穷大时）为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n)，称O(f(n))为算法的渐进时间复杂度，简称时间复杂度。  空间复杂度：空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度，它是问题规模n的函数，记做S(n)=O(f(n))。比如直接插入排序的时间复杂度是O(n^2),空间复杂度是O(1)。而一般的递归算法就要有O(n)的空间复杂度了，因为每次递归都要存储返回信息，需要辅助空间的大小随着n的增大线性增大。  就地排序：若一个排序算法所需的辅助空间并不依赖于问题的规模n，即时间复杂度是O(1),则称该排序算法为就地排序。非就地排序算法的时间复杂度一般为O(n)。  注：算法是否稳定并不能衡量一个算法的优劣，排序算法主要包含2种操作：比较和移动，但并不是所有的排序都基于比较操作，比如基数排序。  排序算法的体系结构图如下：    各大排序算法的时间复杂度、空间复杂度及稳定性见下表：    本次实验的全部代码如下：   1. #include<stdio.h> 2. #include<stdlib.h> 3. #include<string.h> 5. **void** InsertSort(**int** a[], **int** n) 6. { 7. **int** i, j; 8. **for**(i=1; i<n; i++) 9. **if**(a[i] < a[i-1]) 10. { 11. **int** temp = a[i];                       //保存要比较的值 12. **for**(j=i-1; j>=0 && a[j]>temp; j--)    //从后向前查找待插入位置 13. a[j+1] = a[j];                    //挪位 14. a[j+1] = temp;                       //复制到插入位置 15. } 16. } 18. **void** Swap(**int** a,**int** b){ 19. **int** temp; 20. temp=a; 21. a=b; 22. b=temp; 23. } 25. **void** shellSort(**int** a[],**int** n){ 26. **int** i,j,gap,temp; 27. **for**(gap = n/2;gap>0;gap/=2){ 28. **for**(i=gap;i<n;i+=gap){ 29. temp = a[i]; 30. **for**(j = i-gap;j>=0&&a[j]>temp;j-=gap){ 31. a[j+gap] = a[j]; 32. } 33. a[j+gap] = temp; 34. } 35. } 36. } 38. **void** BubbleSort(**int** a[], **int** n) 39. { 40. **int** i, j; 41. **for**(i=0; i<n; i++){ 42. **bool** flag=**false**;              //表示本趟冒泡是否发生交换的标志 43. **for**(j=1; j<n-i; j++){         //j的起始位置为1，终止位置为n-i 44. **if**(a[j]<a[j-1]){ 45. Swap(a[j-1], a[j]); 46. flag=**true**; 47. } 48. } 49. **if**(flag==**false**)             //未交换，说明已经有序，停止排序 50. **return**; 51. } 52. } 54. **void** QuickSort(**int** a[],**int** left,**int** right){ 55. **if**(left<right){ 56. **int** i=left,j=right; 57. **int** base=a[left];             //基准 58. **while**(i<j){ 59. **while**(i<j&&a[j]>=base)    //从右往左找小于base的元素 60. j--; 61. **if**(i<j) 62. a[i++]=a[j]; 63. **while**(i<j&&a[i]<base)    //从左往右找大于base的值 64. i++; 65. **if**(i<j) 66. a[j--]=a[i]; 67. } 68. a[i]=base;                  //将基准base填入最后的坑中 69. QuickSort(a,left,i-1);      //递归调用，分治 70. QuickSort(a,i+1,right); 71. } 72. } 74. **void** SelectSort(**int** a[],**int** n){ 75. **int** i,j,min; 76. **for**(i=0;i<n-1;i++){ 77. min=i; 78. **for**(j=i+1;j<n;j++){ 79. **if**(a[j]<a[min]) 80. min=j; 81. } 82. **if**(min!=i) 83. Swap(a[i],a[min]); 84. } 85. } 87. **void** Merge(**int** A[], **int** Temp[], **int** L, **int** R, **int** RightEnd){ 88. **int** LeftEnd = R-1; 89. **int** p=L,i; 90. **int** num=RightEnd-L+1; 92. //先合并最短序列的长度的个数个元素 93. **while**(L<=LeftEnd&&R<=RightEnd){ 95. **if**(A[L]<=A[R]) 96. Temp[p++]=A[L++]; 97. **else** 98. Temp[p++]=A[R++]; 99. } 101. //判断如果是左侧序列还有剩余 102. **while**(L<=LeftEnd) 103. Temp[p++]=A[L++]; 104. //判断如果是右侧序列还有剩余 105. **while**(R<=RightEnd) 106. Temp[p++]=A[R++]; 107. // 将辅助空间中的值拷贝到原列表中，完成排序 108. **for**(i=0;i<num;i++,RightEnd--) 109. A[RightEnd]=Temp[RightEnd]; 110. } 112. //递归拆分，递归归并 113. **void** m\_sort(**int**\* arr, **int**\* temp, **int** L, **int** right\_end){ 114. **int** center; 115. **if**(L<right\_end){ 116. center = (L+right\_end)/2; 117. m\_sort(arr,temp,L,center); 118. m\_sort(arr,temp,center+1,right\_end); 119. Merge(arr,temp,L,center+1,right\_end); 120. } 121. } 123. //归并排序 124. **void** merge\_Sort(**int**\* arr,**int** length){ 125. **int** \*temp=(**int**\* )malloc(length\***sizeof**(**int**));  //申请辅助空间 126. **if**(temp==NULL){ 127. **return**; 128. } 129. m\_sort(arr,temp,0,length-1); 130. free(temp); 131. temp=NULL; 132. } 134. //打印元素 135. **void** printArr(**int** \*arr,**int** length){ 136. **for**(**int** i=0;i<length;i++){ 137. printf("%d  ",arr[i]); 138. } 139. printf("\n"); 140. } 142. **int** main(){ 143. **int** arr[10]={32,53,24,106,82,93,76,84,79,91}; 144. **int** length = **sizeof**(arr)/**sizeof**(**int**); 145. length = 10; 146. printf("BEFORE SORT : "); 147. printArr(arr,length); 149. merge\_Sort(arr,length); 150. printf("MERGE SORT  : "); 151. printArr(arr,length); 153. SelectSort(arr,length); 154. printf("SELECT SORT : "); 155. printArr(arr,length); 157. BubbleSort(arr,length); 158. printf("BUBBLE SORT : "); 159. printArr(arr,length); 161. QuickSort(arr,0,length-1); 162. printf("QUICK SORT  : "); 163. printArr(arr,length); 165. shellSort(arr,length); 166. printf("SHELL SORT  : "); 167. printArr(arr,length); 169. InsertSort(arr,length); 170. printf("INSERT SORT : "); 171. printArr(arr,length); 172. **return** 0; 173. }   本次实验采用的测试样例如下：   1. **int** arr[10]={32,53,24,106,82,93,76,84,79,91};   本次实验的测试结果如下： |
| **三、指导教师评语及成绩** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |