山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201700130009 | 姓名： 张愈博 | | 班级： 计科17.3 |
| 实验题目：堆及其应用 | | | |
| 实验学时：2 | | 实验日期： 2018.11.29 | |
| 实验目的：  1、掌握堆结构的定义、描述方法、操作定义及实现。  2、掌握堆结构的应用。 | | | |
| 软件环境：  Windows 10  Dev C++ 5.11 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求）   (1)创建最小堆类。最小堆的存储结构使用数组。提供操作:插入、删除、初始化。  (2)接收键盘录入的一系列整数，以文本形式输出其对应的最小堆。  (3)对建立好的最小堆，键盘输入插入元素，输出插入操作完成后的堆（可以文本形式表示）；键盘输入删除元素，输出删除操作完成后的堆。  (4)键盘输入n, 随机生成 n 个 0~1000 之间的整数；输出堆排序的排序过程。  (5)键盘输入字符个数 n, 以（c，w）形式依次字符和字符出现的频率，字符互不相同，输出 Huffman 树（可用文本形式）和每个字符的 Huffman 编码。   1. 数据结构与算法描述（整体思路描述，所需要的数据结构与算法）   (1)首先按题目要求创建一个最小堆类，提供插入、删除、初始化操作，并重载输出运算符来输出类中保存成员的heap数组。录入整数并输出用插入操作完成，输出最小堆用层次遍历，即直接用cout输出heap数组。  (2)继续插入和删除元素，重新输出显示操作后的小根堆。注意这里的按关键字删除，只需先在heap数组中搜索关键字，然后以该位置为根来进行重新堆化即可。  (3)使用随机数生成函数，按要求生成n个整数，再利用小根堆的插入和删除操作对这些数进行排序，堆排序的复杂度是O(nlogn)。  (4)本题的难点在于建立霍夫曼树和输出对应字符的霍夫曼编码。对于前者，可以先给所有的初始节点赋上权重（频率），在用他们初始化一个小根堆，之后不断去两颗最轻的树合并再插入小根堆直至堆中仅剩一个节点，把它的tree指针作为返回值。对于后者，现在我们已经有huffman树了，需要输出其对应的huffman编码。为此我们需要从根节点开始搜索叶节点，并把经过的路径保存下来以便下一次搜索操作。这里的实现思想类似于二叉树的层次遍历，在经过一个节点时将将它的左右孩子压入二叉树节点队列（如果不为空的话），同时将它的对应编码压入另一个code队列，当遇到叶节点时直接输出code中对应的代码。不断进行上述操作直到二叉树节点队列为空。   1. 测试结果（测试输入，测试输出，结果分析）     如图，经分析各项输出结果正常。   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   1)需要自定义huffmanNode的比较运算符，不然会在堆的初始化操作中出错，进而导致小根堆的push操作中出错。  2)注意在输出霍夫曼编码时，采用了2个队列，一个用来保存路过的节点，另一个用来保存已走路径所对应的编码。它们的插入和删除顺序都是有严格要求的，不能轻易改变，否则会造成不能正常读取节点的情况。  3）在实现小根堆的删除、初始化操作时，在循环中最好先判断是否小于在判断大于等于，否则currentNode的值会被提前改变，导致灾难性后果。   1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）   Main.cpp  #include<iostream>  #include"小根堆.h"  #include"二叉树.h"  #include"huffmanNode.h"  #include <time.h>  #include<stdlib.h>  using namespace std;  int\* generate\_random(int size)  {  srand((unsigned)time(0)); //这个随机数种子1秒会改变一次，利用了系统时钟  int \*p=new int[size+1];  for(int i=1;i<=size;i++)  {  p[i]=(rand()%1001);  cout<<"p["<<i<<"]="<<p[i]<<" ";  }  cout<<endl;  return p;  }  int main()  {  minHeap<int> heap;  int size,\*a;  cout<<"please input the size:"; cin>>size;  a=new int [size+1];  cout<<"please input the value of array：";  for(int i=1;i<=size;i++)  {  cin>>a[i];  }  heap.initialize(a,size);  cout<<"the minHeap is:"<<heap<<endl;    //以下是堆排序部分  cout<<"请输入生成随机数的数量n"<<endl; cin>>size;  int \*Array=generate\_random(size);  heapSort(Array,size);  cout<<"After heap sort:";  for(int i=1;i<size;i++)  cout<<Array[i]<<" ";  cout<<endl;    //以下是生成huffman树的部分  cout<<"please input the number of char:"; int num; cin>>num;  int \*weight=new int[num+1];  char \*str=new char[num+1];  cout<<"please input the data:"<<endl;  for(int i=1;i<=num;i++)  {  cin>>str[i];  cin>>weight[i];  }  linkedBinaryTree<int> \*Htree=huffman(weight,num);  cout<<"The huffman tree is as followed:"<<endl;  cout<<"levelOrder:"; Htree->levelOrder(); cout<<endl;  cout<<"preOrder:"; Htree->preOrder(); cout<<endl;  Htree->huffmanCode(str);  }  二叉树.h  #pragma once  #include<iostream>  #include"arrayQueue.h"  #include<string.h>  using namespace std;  template<class T>  class binaryTreeNode  {  public:  T element; //节点中的元素  binaryTreeNode<T> \*left,\*right; //左右子树    binaryTreeNode() //无参数  { left=right=NULL; }  binaryTreeNode(const T& ele)  {//只有数据参数  element=ele;  left=right=NULL;  }  binaryTreeNode(const T& ele,binaryTreeNode\* theLeft,binaryTreeNode\* theRight)  {//数据加指针  element=ele;  left=theLeft;  right=theRight;  }  };  template<class T>  class linkedBinaryTree  {  public:  linkedBinaryTree()  {  root=NULL;  treeSize=0;  }  ~linkedBinaryTree()  {  //erase();  }  bool empty() const  {  return treeSize;  }  void preOrder()  {  preOrder(root); //这里调用的是静态成员函数  }  void inOrder()  {  inOrder(root);  }  void postOrder()  {  postOrder(root);  }  void makeTree(const T& element,linkedBinaryTree<T>& left,linkedBinaryTree<T> &right)  {//将left,right和element合并成一颗新树  //left，right和this必须不同！！    root=new binaryTreeNode<T>(element,left.root,right.root);  treeSize=left.treeSize+right.treeSize+1;    //阻止访问left和right  left.root=right.root=NULL;  left.treeSize=right.treeSize=0;  }  binaryTreeNode<T> \*makeTree(char \*pre,char \*in,const int n)  {  root=Sub\_makeTree(pre,in,n);  // cout<<"this is "<<root<<endl;  }  int height()  {  return Sub\_height(root);  }  int size()  {  //return treeSize;  return size(root);  }  int size(binaryTreeNode<T> \*t)  {//返回根为\*t的树的节点数。对某根点求它左右子树的元素数，相加再加上它本身即为总节点数。  if(t==NULL) return 0;  else  {  int ls=size(t->left); //左子树元素数  int rs=size(t->right); //右子树元素数  return ls+rs+1; //加上根节点  }  //复杂度O(n)  }  void levelOrder()  {  Sub\_levelOrder(root);  }  int max\_width(binaryTreeNode<T> \*t)  {//求二叉树的每层最大节点数  arrayQueue< binaryTreeNode<T>\* > q;  binaryTreeNode<T> \*last=root; //last指向每层最右端的元素  int level=1; //层数  int width[20]; //保存每层节点数  while(t!=NULL)  {  if(t->left!=NULL)  q.push(t->left.element);  if(t->right!=NULL)  q.push(t->right.element);    //下一个t  if(q.empty()==true) break; //队列为空，结束循环  t=q.front();  q.pop();  if(t==last)  {//已经扫完当前层,此时队列里全是下一层的节点  width[++level]=q.size();  last=q.back(); //更新last  }  }  //从层数组中找出最大节点数  int max=0;  for(int i=1;i<level;i++)  {  if(width[i]>max)  max=width[i];  }    return max;  }  static bool compare(binaryTreeNode<T> \*root1,binaryTreeNode<T> \*root2)  {//先判断两棵树根节点是否有空，若均不为空就比较其内的元素值  //再判断左右子树 ------O(N)  if( (root1==NULL&&root2!=NULL) || (root1!=NULL&&root2==NULL) )  return false; //树的结构不一样  else  {  if(root1==NULL && root2==NULL) return true; //根节点均为空，递归到终点  if( \*(root1) != \*(root2) ) return false; //根节点均不为空且值不同  else  {//根节点不为空且值相同，比较左右子树  bool lt=compare(root1->left,root2->left);  bool rt=compare(root1->right,root2->right);  return lt&&rt;  }  }  }  static void changeIntoPost(T \*pre,T \*in,T \*post,const int n)  {  if(n<1) return;  T root=pre[0];  post[n-1]=root;  T \*p;  int mid; //mid是root在in中的索引    for(p=in;p<in+n;p++)  {  if( \*(p)==root )  break;  }  mid=p-in;  changeIntoPost(pre+1,in,post,mid);  changeIntoPost(pre+mid+1,p+1,post+mid,n-1-mid);  }  void huffmanCode(char \*ch)  {//ch是节点的关键字，本算法的思想类似于层次遍历  arrayQueue< binaryTreeNode<T>\* > q;  arrayQueue< string > code;  code.push("");  binaryTreeNode<T> \*t=root;  while(t!=NULL)  {  if(t->left!=NULL){ q.push(t->left); code.push(code.front()+"0"); } //由于是string类型可以用+连接  if(t->right!=NULL){ q.push(t->right); code.push(code.front()+"1"); }  if(t->left==NULL&&t->right==NULL)  {//遇到了叶节点  cout<<ch[t->element]<<": "<<code.front()<<endl;  //cout<<t->element<<code.front()<<endl;  }  if(q.empty()==true) return;  else t=q.front();  q.pop();  code.pop();  }  }  T root\_element()  {  if(root!=NULL)  return root->element;  }  private:  binaryTreeNode<T> \*root; //指向根节点的指针  int treeSize; //树的节点数  static void visit(binaryTreeNode<T> \*X)  {//访问函数  cout<< X->element<<' ';  }  static void preOrder(binaryTreeNode<T> \*t)  {//前序遍历  if(t!=NULL)  {  visit(t);  preOrder(t->left);  preOrder(t->right);  }  }  static void inOrder(binaryTreeNode<T> \*t)  {//中序遍历  if(t!=NULL)  {  inOrder(t->left);  visit(t);  inOrder(t->right);  }  }  static void postOrder(binaryTreeNode<T> \*t)  {//后序遍历  if(t!=NULL)  {  postOrder(t->left);  postOrder(t->right);  visit(t);  }  }  static void dispose(binaryTreeNode<T> \*t)  {  delete t;  }  int Sub\_height(binaryTreeNode<T> \*t)  {//返回根为\*t的树的高度。对根节点求它的左右子树的高度，取大者作为总树的高度  if(t==NULL)  return 0;  else  {  int hl=Sub\_height(t->left);  int hr=Sub\_height(t->right);  if(hl>hr)  return ++hl;  else  return ++hr;  }  //复杂度O(n)，每一个元素都比较了一遍  }  binaryTreeNode<T> \*Sub\_makeTree(char \*pre,char \*in,const int n)  {//由前序和中序序列构造一颗二叉树，n是二叉树的节点总数  if(n<1) return NULL;  char Root=pre[0]; cout<<"Root="<<Root<<"\t";  binaryTreeNode<T> \*b=new binaryTreeNode<T>(Root);  treeSize++;    char \*p; int mid=0; //根节点在in中的索引  for(p=in;p<in+n;p++)  {//在中序序列中找到根节点  if( (\*p)==Root )  break;  mid++;  }  cout<<"mid="<<mid<<endl;  b->left=Sub\_makeTree(pre+1,in,mid); //对左子树递归  b->right=Sub\_makeTree(pre+1+mid,in+mid+1,n-mid-1); //对右子树递归    return b;  }  void Sub\_levelOrder(binaryTreeNode<T> \*t)  {//层次遍历二叉树\*t  arrayQueue< binaryTreeNode<T>\* > q; //这里先进先出所以用队列  while(t!=NULL)  {  visit(t); //访问 t    //将t的孩子插入队列，先左后右  if(t->left!=NULL)  q.push(t->left);  if(t->right!=NULL)  q.push(t->right);    //访问下一个节点  if(q.empty()==true) return; //已访问完所有元素  t=q.front();  q.pop();  }  }    };  大根堆.h  #pragma once  #include<iostream>  using namespace std;  template<class T>  class maxHeap  {  public:  maxHeap(const int initialCapacity=10)  {  heap=new T [initialCapacity];  arrayLength=heapSize=0;  }  ~maxHeap()  {  delete [] heap;  }  bool empty() const  {  if(heapSize==0) return true;  else return false;  }  int size() const  {  return heapSize;  }  const T& top()  {//返回最大元素的引用  return heap[1];  }  void pop()  {//删除最大元素 ,这里只是把最后一个元素踢出heapSize的范围但是并没有删除，有隐患？  if(heapSize==0)  {  cerr<<"堆为空！"<<endl;  return;  }  heap[1].~T(); //删除最大元素    T lastElement=heap[heapSize-1]; //删除一个元素，然后重新建堆  //重新堆化：从根开始，为最后一个元素寻找位置  int currentNode=1,child=2;  while(child<=heapSize)  {//首先选出currentNode较大的孩子，然后看看能不能插入  if(child<heapSize && heap[child]<heap[child+1] )  child++;  if( heap[currentNode]<heap[child] )  {  heap[currentNode]=heap[child];  currentNode=child;  child\*=2;  }  if(heap[currentNode]>=heap[child])  break; //找到一个可以插入的位置，跳出循环  }  heap[currentNode]=lastElement;  heapSize--;  }  void push(const T& theElement)  {//把元素theElement加入堆  if(heapSize == arrayLength-1)  {//快满的时候倍长数组  T \*p=new T[2\*arrayLength];  for(int i=0;i<heapSize;i++)  {  p[i]=heap[i];  }  delete [] heap;  heap=p;  arrayLength\*=2;  }  //为元素寻找插入位置  int pos = heapSize+1;  while( pos!=1 && heap[pos/2]<theElement )  {//父节点较小且没有走到根  heap[pos]=heap[pos/2]; //把元素向下移动  pos/=2; //检查节点向上一层  }  heap[pos]=theElement;  heapSize++;  }  void initialize(T \*theHeap,int theSize)  {//在数组theHeap[1:theSize]中建大根堆  //首先收回原heap空间  delete [] heap;  heap=theHeap;  heapSize=theSize;    //从有叶节点的节点开始堆化  for(int root=heapSize/2;root>=1;root--)  {  T lastElement=heap[root];  int child=root\*2; //child是root的子节点  int currentNode=root;  while(child<=heapSize) //保证以root为根节点的树是大根堆  {  if(child<heapSize && heap[child] < heap[child+1] )  child++; //保证child是较大子节点  if(heap[child]>heap[currentNode])  {  heap[currentNode]=heap[child]; //把孩子向上移  currentNode=child;  child\*=2;  }  if(heap[child]<=heap[currentNode])  break;  }  heap[currentNode]=lastElement;  }  }  friend ostream& operator<<(ostream &out,maxHeap &x)  {  for(int i=0;i<x.heapSize;i++)  {  out<<x.heap[i];  }  return out;  }      private:  T \*heap; //保存堆中元素的数组，从1开始存放  int arrayLength; //数组heap的容量  int heapSize; //堆的元素个数  };  template<class T>  void heapSort(T a[],int n)  {//使用堆排序方法给a[1:n]排序  //在数组上建立大根堆  maxHeap<T> heap;  heap.initialize(a,n);    //逐个从大根堆中提取元素，从小到大排序  for(int i=n-1;i>=1;i--)  {  T x=heap.top();  heap.pop();  a[i+1]=x;  }  heap.deactiveArray(); //从堆的析构函数中保留数组a  }  huffmanNode.h  #pragma once  #include<iostream>  #include"小根堆.h"  #include"二叉树.h"  using namespace std;  template<class T>  class huffmanNode  {  public:  operator T()  {  return weight;  }  bool operator >(huffmanNode<T> A)  {  if(A.weight<this->weight) return true;  else return false;  }  bool operator >=(huffmanNode<T> A)  {  if(A.weight<=this->weight) return true;  else return false;  }  linkedBinaryTree<int> \* tree;  T weight;  };  template<class T>  linkedBinaryTree<int>\* huffman(T weight[],int n)  {//用权weight[1:n]生成huffman树，n>=1  //创建一组单节点  huffmanNode<int> \*hNode=new huffmanNode<T> [n+1];  linkedBinaryTree<int> ET; //空树  for(int i=1;i<=n;i++)  {  hNode[i].weight=weight[i];  hNode[i].tree=new linkedBinaryTree<int>;  hNode[i].tree->makeTree(i,ET,ET); //i是tree中的element值，相当于节点标号  }  //让单节点构成一个小根堆(根据权重weight来确定相对位置)  minHeap< huffmanNode<T> > heap;  heap.initialize(hNode,n);    //不断从小根堆中取出两个最轻的树合并后再插入堆，直到只剩下一颗树  huffmanNode<T> w,x,y;  linkedBinaryTree<int> \*t;  for(int i=1;i<n;i++)  {  x=heap.top(); heap.pop();  y=heap.top(); heap.pop();    t=new linkedBinaryTree<int>;  //cout<<"中间元素是："<<x.tree->root\_element()<<endl;  t->makeTree(x.weight+y.weight, \*(x.tree) , \*(y.tree) ); //让huffman节点中树的根元素值等于其子树的权重值，与书上不一样  w.weight=x.weight+y.weight;  w.tree=t;  heap.push(w);  delete x.tree; delete y.tree;  }  //跳出循环时heap中只剩一颗树  return heap.top().tree;  }  arrayQueue.h  //本队列使用映射公式：location(i)=(location(队列首元素)+i)%arrayLength 即环形数组表示法  #pragma once  #include<iostream>  #include<cstdlib>  #include<algorithm>  template<class T>  class arrayQueue  {  public:  arrayQueue(int initialCapacity = 10)  {//构造函数  if(initialCapacity<0) std::cerr<<"队列长度必须大于0！"<<std::endl;  else{  Queue=new T[initialCapacity];  arrayLength=initialCapacity;  qFront=qBack=0; //这里是从Queue[1]开始插入元素  }  }  ~arrayQueue() {delete [] Queue;}  bool empty() const  {  if(qFront==qBack) return true;  else return false;  }  int size() const  {  return (arrayLength+qBack-qFront)%arrayLength;  }  T& front()  {  if(empty()!=true)  return Queue[(qFront+1)%arrayLength];  else  { std::cerr<<"队列为空"<<std::endl; exit(1); }  }  T& back()  {  if(empty()!=true)  return Queue[qBack];  else  { std::cerr<<"队列为空"<<std::endl; exit(1); }  }  T pop()  {//从队首删除元素  T \*p=&front(); //这里已经判断了队列是否为空  T temp=(\*p);  qFront=(qFront+1)%arrayLength;  (\*p).~T(); //析构首元素好像不能表示int的删除...，就是无法恢复到初始化以前的状态  return temp;  }  void push(const T& ele)  {//从队尾添加元素  if( (qBack+1)%arrayLength==qFront )  {//队列将满，加倍数组长度  T \*newQueue=new T[2\*arrayLength];  int start=(qFront+1)%arrayLength;  if(start==0||start==1)  {//未形成环  std::copy(Queue+start,Queue+qBack+1,newQueue);  }  else  {//形成了环  std::copy(Queue+start,Queue+arrayLength,newQueue);  //复制第2段(start,队列末端，新队列起点）  std::copy(Queue,Queue+qBack+1,newQueue+(arrayLength-start));  //复制第1段（原队列首端，qback,新队列第arraylength-start个位置）  }  qFront=(arrayLength)\*2-1;  qBack=arrayLength-1-1; //重新设置首尾游标  arrayLength=arrayLength\*2;  delete [] Queue;  Queue=newQueue;  }  //把元素插入队列的尾部  qBack=(qBack+1)%arrayLength;  Queue[qBack]=ele;    }  void output()  {  for(int i=qFront;i<qBack;i++)  std::cout<<Queue[i];  std::cout<<std::endl;  }  private:  int qFront; //队列中第一个元素的前一个未知  int qBack; //队列最后一个元素的位置  int arrayLength; //队列的容量  T \*Queue; //队列元素  }; | | | |