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：201805130155 | 姓名：赵雨晗 | | 班级：计科18.3 |
| 实验题目：堆及其应用 | | | |
| 实验学时：2 | | 实验日期：2019.11.20 | |
| 实验目的：   1. 掌握堆结构的定义、描述方法、操作定义的实现 2. 掌握堆结构的应用 | | | |
| 软件环境：  Visual Studio Community 2017 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求） 2. 创建最小堆类，最小堆的储存结构用数组。提供操作：插入、删除、初始化 3. 接受键盘录入的一系列整数，以文本的方式输出其对应的最小堆。 4. 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。 5. 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 6. 键盘输入字符个数n，以(c, w)的形式依次表示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。 7. 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） 8. 堆是一种完全二叉树，可以用数组描述。数组中第k个元素的左右儿子分别为2\*k, 2\*k+1，他的父亲为[k/2]。 9. 堆的插入：将元素放到数组末尾并不断向上调整，直至满足堆的性质。 10. 堆的删除：找到删除元素后将其删除，并将数组末尾的元素放置该位置，并对这棵子树进行调整，直至满足堆的性质。 11. 堆排序：建立最大堆，每次取top后将其放在数组的第i位上，i从n减到1。 12. Huffman树：初始有n棵树，将其放到堆中，每次取两个top后将其合成一棵树并再次放入堆中，直至堆中只剩一个元素。 13. 测试结果（测试输入，测试输出，结果分析）      1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   Huffman树的结果与手工模拟的结果一致  堆排序结果为递增数列  插入元素和删除元素后的结果与手工模拟的结果一致   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）   #include<cstdio>  #include<iostream>  #include<stdexcept>  #include<cstdlib>  #include<algorithm>  #include<cstring>  #include<functional>  #include<queue>  using namespace std;  template<typename T>  struct binaryTreeNode{  T element;  int height, size;  binaryTreeNode<T>\* leftChild;  binaryTreeNode<T>\* rightChild;  binaryTreeNode<T>\* father;  binaryTreeNode() {height = 0, size = 0;}  binaryTreeNode(const T& theElement) : element(theElement) {  leftChild = rightChild = father = nullptr;  }  bool operator<(const binaryTreeNode<T>& x) const {  return element < x.element;  }  };  template<typename T>  class linkedBinaryTree{  public:  linkedBinaryTree(binaryTreeNode<T>\* node = nullptr) {root = node;}  virtual ~linkedBinaryTree() {}  bool operator <(const linkedBinaryTree<T>& x) const{  return root ->element < x.root->element;  }    void size() const {  visit = linkedBinaryTree<T>::size;  postOrder(root);  }  //求树根  binaryTreeNode<T>\*& getRoot(){  return root;  }  //先序遍历  void preOrder(void(\*\_visit)(binaryTreeNode<T>\*) = linkedBinaryTree<T>::output) {  visit = \_visit;  preOrder(root);  }  //中序遍历  void inOrder(void(\*\_visit)(binaryTreeNode<T>\*) = linkedBinaryTree<T>::output) {  visit = \_visit;  inOrder(root);  }  //后序遍历  void postOrder(void(\*\_visit)(binaryTreeNode<T>\*) = linkedBinaryTree<T>::output) {  visit = \_visit;  postOrder(root);  }  //层次遍历  void levelOrder(void(\*\_visit)(binaryTreeNode<T>\*) = linkedBinaryTree<T>::output) {  visit = \_visit;  levelOrder(root);  }  //清空树  void erase(){  postOrder(linkedBinaryTree<T>::dispose);  root = nullptr;  }  //求树高  void height() const {  visit = linkedBinaryTree<T>::height;  preOrder(root);  }  int calc() {  return tmp2;  }  protected:  int tmp;  int tmpArray[100010];  T \*tmp1;  binaryTreeNode<T> \*root;  static int tmp2;  static void (\*visit)(binaryTreeNode<T>\*);  static void preOrder(binaryTreeNode<T>\*);  static void inOrder(binaryTreeNode<T>\*);  static void postOrder(binaryTreeNode<T>\*);  static void levelOrder(binaryTreeNode<T>\*);  static void dispose(binaryTreeNode<T>\* t) {delete t;}  static void size(binaryTreeNode<T>\*);  static void height(binaryTreeNode<T>\*);  static void output(binaryTreeNode<T>\* t) {cout << t ->element << t ->height<<endl;}  };  template<typename T>  int linkedBinaryTree<T>::tmp2 = 0;  template<typename T>  void (\*linkedBinaryTree<T>::visit)(binaryTreeNode<T>\*) = linkedBinaryTree<T>::output;  template<typename T>  void linkedBinaryTree<T>::preOrder(binaryTreeNode<T>\* t){  if (t == nullptr) return;  linkedBinaryTree<T>::visit(t);  preOrder(t ->leftChild);  preOrder(t ->rightChild);  }  template<typename T>  void linkedBinaryTree<T>::inOrder(binaryTreeNode<T>\* t){  if (t == nullptr) return;  inOrder(t ->leftChild);  linkedBinaryTree<T>::visit(t);  inOrder(t ->rightChild);  }  template<typename T>  void linkedBinaryTree<T>::postOrder(binaryTreeNode<T>\* t) {  if (t == nullptr) return;  postOrder(t ->leftChild);  postOrder(t ->rightChild);  linkedBinaryTree<T>::visit(t);  }  template<typename T>  void linkedBinaryTree<T>::levelOrder(binaryTreeNode<T>\* t) {  queue<binaryTreeNode<T>\* >q;  q.push(t);  while(!q.empty()){  binaryTreeNode<T>\* p = q.front(); q.pop();  if (p == nullptr) continue;  linkedBinaryTree<T>::visit(p);  q.push(p ->leftChild);  q.push(p ->rightChild);  }  }  template<typename T>  void linkedBinaryTree<T>::height(binaryTreeNode<T>\* t){  if (t -> father != nullptr) t ->height = t ->father ->height + 1;  else t ->height = 0;  if (t ->leftChild == nullptr && t ->rightChild == nullptr)  tmp2 += t ->height \* t ->element;  if (t ->leftChild != nullptr)t ->leftChild ->father = t;  if (t ->rightChild != nullptr) t ->rightChild ->father = t;  }  template<typename T>  void linkedBinaryTree<T>::size(binaryTreeNode<T>\* t){  t ->size = 1;  if (t ->leftChild) t ->size += t ->leftChild ->size;  if (t ->rightChild) t ->size += t ->rightChild ->size;  }  template<typename T>  binaryTreeNode<T>\* makeTree(binaryTreeNode<T> \*l, binaryTreeNode<T> \*r){    binaryTreeNode<T> \*node = new binaryTreeNode<T>;  node ->leftChild = l;  node ->rightChild = r;  node ->element = l -> element + r -> element;  if (node ->leftChild != nullptr) node ->leftChild ->father = node;  if (node ->rightChild != nullptr) node ->rightChild ->father = node;  return node;  }  template<typename T, typename Pr = std::greater<T> >  class Heap {  public:  Heap(int = 10);  Heap(const Heap<T, Pr>&);  ~Heap() {  delete[] heap;  }  bool empty() const {return heapSize == 0;}  int size() const {return heapSize;}  const T& top() {  if (heapSize < 1) throw out\_of\_range("heap is empty");  return heap[1];  }  void pop();  void push(const T&);  void init(T\*, T\*);  void erase(const T&);  void output(std::ostream&, char = '\n');  protected:  int heapSize;  int arrayLength;  T\* heap;  Pr cmp;  };  template<typename T, typename cmp>  Heap<T, cmp>::Heap(int initialCapacity) : cmp() {  if (initialCapacity < 1) throw out\_of\_range("the initial capacity of heap must > 0");  arrayLength = initialCapacity + 1;  heap = new T[arrayLength];  heapSize = 0;  }  template<typename T, typename Pr>  Heap<T, Pr>::Heap(const Heap<T, Pr>& theHeap) {  arrayLength = theHeap.arrayLength;  heapSize = theHeap.heapSize;  heap = new T[arrayLength];  copy(theHeap.heap, theHeap.heap + heapSize + 1, heap);  }  template<typename T, typename Pr>  void Heap<T, Pr>::pop(){  heap[1] = heap[heapSize--];  int k = 1, c = 2;  while(c <= heapSize){  if (c < heapSize && !cmp(heap[c], heap[c | 1])) c |= 1;  if (cmp(heap[k], heap[c])) break;  swap(heap[k], heap[c]);  k = c;  c <<= 1;  }  }  template<typename T, typename Pr>  void Heap<T, Pr>::push(const T& element) {  if (heapSize >= arrayLength - 1) {  T\* temp = new T[arrayLength \* 2];  for (int i = 1; i <= heapSize; ++i) temp[i] = heap[i];  delete[] heap;  heap = temp;  arrayLength \*= 2;  }  heap[++heapSize] = element;  int k = heapSize;  while(k != 1) {  if (cmp(heap[k], heap[k >> 1])) swap(heap[k], heap[k >> 1]);  k >>= 1;  }  }  template<typename T, typename Pr>  void Heap<T, Pr>::init(T\* begin, T\* end) {  delete[] heap;  heapSize = end - begin;  arrayLength = heapSize;  heap = new T[arrayLength + 1];  for (int i = 1; i <= heapSize; ++i) heap[i] = \*(begin + i - 1);  for (int rt = heapSize / 2; rt; --rt) {  int k = rt;  int c = k << 1;  while(c <= heapSize) {  if (c < heapSize && !cmp(heap[c], heap[c | 1])) c |= 1;  if (cmp(heap[k], heap[c])) break;  swap(heap[k], heap[c]);  k = c;  c <<= 1;  }  }  }  template<typename T, typename Pr>  void Heap<T, Pr>::erase(const T& element) {  int k = heapSize;  for (int i = 1; i <= heapSize; ++i) {  if (heap[i] == element) {  heap[i] = heap[heapSize];  heap[heapSize--].~T();  k = i;  break;  }  }  int c = k << 1;  while(c <= heapSize) {  if (c < heapSize && !cmp(heap[c], heap[c << 1])) c |= 1;  if (cmp(heap[k], heap[c])) break;  swap(heap[k], heap[c]);  k = c;  c <<= 1;  }  }  template<typename T, typename Pr>  void Heap<T, Pr>::output(ostream& out, char ends) {  for (int i = 1; i <= heapSize; ++i) out << heap[i] << ' ';  out<<ends;  }  binaryTreeNode<int> a[50];  char s[1000010];  int main(){  Heap<binaryTreeNode<int>, less<binaryTreeNode<int> > >H;  scanf("%s", s + 1);  int n = strlen(s + 1);  for (int i = 1; i <= n; ++i) a[s[i] - 'a' + 1].element++;  int cnt = 0;  for (int i = 1; i <= 26; ++i) if (a[i].element){  H.push(a[i]);  cnt++;  }  while(cnt > 1){  binaryTreeNode<int> \*l = new binaryTreeNode<int>(H.top());  H.pop();  binaryTreeNode<int> \*r = new binaryTreeNode<int>(H.top());  H.pop();  binaryTreeNode<int> \*k = makeTree(l, r);  H.push(\*k);  cnt--;  }  binaryTreeNode<int> root = H.top();  linkedBinaryTree<int> Tr(&root);  Tr.height();  // Tr.preOrder();  printf("%d\n", Tr.calc());  } | | | |