**附件：**

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：202000130138 | 姓名：宋璎航 | | 班级： 20.3 |
| 实验题目：堆及其应用 | | | |
| 实验学时：2 | | 实验日期： 2021/12/2 | |
| 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。 | | | |
| 软件开发环境：  VSC | | | |
| 1. 实验内容   **1、题目描述**：  创建最小堆类，使用数组作为存储结构，提供操作：插入、删除、初始化、排序。保证第一个操作是建堆操作，接下来是对堆的插入和删除操作，删除和插入都在建好的堆上进行操作。  **输入输出格式**：  **输入**：  第一行一个数n（n<=5000)，代表堆的大小；第二行n个数，代表堆的各个元素；第三行一个数m (m<=1000)，代表接下来共m个操作。接下来m行，分别代表各个操作。下面是各个操作的格式：  插入操作：1 num；  删除操作：2；  排序操作：第一行两个数3和n，3代表是排序操作，n代表待排序的数的数目，接下来一行n个数是待排序数。  保证排序操作只出现一次且一定是最后一个操作。  **输出**：  第一行建堆操作输出建好堆后的堆顶的元素。接下来m个操作，若是插入和删除操作。每行输出执行操作后堆顶的元素的值；若是排序操作，输出一行按升序排序好的结果，每个元素间用空格分隔。  **2、题目描述**：哈夫曼编码。  **输入输出格式**：  **输入**：  一串小写字母组成的字符串（不超过1000000)。  **输出**：  输出这个字符串通过Huffman编码后的长度。   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）   最小堆类即根元素最小，插入即从叶节点开始逐级向上比较直到父节点小于插入元素，删除即将最小元素（即堆顶元素）删除，再从子树中找出最小元素替换之。   1. 测试结果（测试输入，测试输出）  输入 10  -225580 113195 -257251 384948 -83524 331745 179545 293165 125998 376875  10  1 -232502  1 -359833  1 95123  2  2  2  1 223971  1 -118735  1 -278843  3 10  -96567 37188 -142422 166589 -169599 245575 -369710 423015 -243107 -108789  **输出**  -257251  -257251  -359833  -359833  -257251  -232502  -225580  -225580  -225580  -278843  -369710 -243107 -169599 -142422 -108789 -96567 37188 166589 245575 423015 输入 abcdabcaba 输出 19   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   测试结果均正确   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 "bits/stdc++.h"  template <class T>  class minHeap  {  T \*heap;  int heapSize, arrayLength;  public:  minHeap()  {  heapSize = 0;  arrayLength = 10000;  heap = new T[arrayLength];  }  minHeap(T \*theHeap, int theSize)  {  arrayLength = 10000;  heap = new T[arrayLength];  std::copy(theHeap, theHeap + theSize + 1, heap);  heapSize = theSize;  for (int root = heapSize / 2; root >= 1; --root)  {  T rootElement = heap[root];  int child = 2 \* root;  while (child <= heapSize)  {  if (child < heapSize and heap[child] > heap[child + 1])  child++;  if (rootElement <= heap[child])  break;  heap[child / 2] = heap[child];  child \*= 2;  }  heap[child / 2] = rootElement;  }  }  ~minHeap() { delete[] heap; }  bool empty()  {  return heapSize == 0;  }  void push(const T theElement)  {  if (heapSize == arrayLength - 1)  {  T \*temp = new T[2 \* arrayLength];  arrayLength \*= 2;  std::copy(heap, heap + arrayLength, temp);  delete[] heap;  heap = temp;  }  int currentNode = ++heapSize;  while (currentNode != 1 and heap[currentNode / 2] > theElement)  {  heap[currentNode] = heap[currentNode / 2];  currentNode /= 2;  }  heap[currentNode] = theElement;  }  void pop()  {  heap[1].~T();  T lastElement = heap[heapSize--];  int currentNode = 1, child = 2;  while (child <= heapSize)  {  if (child < heapSize and heap[child] > heap[child + 1])  ++child;  if (lastElement <= heap[child])  break;  heap[currentNode] = heap[child];  currentNode = child;  child \*= 2;  }  heap[currentNode] = lastElement;  }  T top() const  {  return heap[1];  }  };  int main()  {  int n, m;  std::cin >> n;  int \*theHeap = new int[n + 1];  for (int i = 1; i <= n; ++i)  std::cin >> theHeap[i];  minHeap<int> H(theHeap, n);  std::cout << H.top() << std::endl;  std::cin >> m;  while (m--)  {  int opt;  std::cin >> opt;  switch (opt)  {  case 1:  int num;  std::cin >> num;  H.push(num);  std::cout << H.top() << std::endl;  break;  case 2:  H.pop();  std::cout << H.top() << std::endl;  break;  case 3:  {  std::cin >> n;  int \*theHeap1 = new int[n + 1];  for (int i = 1; i <= n; ++i)  std::cin >> theHeap1[i];  minHeap<int> H1(theHeap1, n);  while (!H1.empty())  {  std::cout << H1.top() << ' ';  H1.pop();  }  std::cout << std::endl;  break;  }  default:  break;  }  }  }  #include <iostream>  #include <string>  #include <queue>  using namespace std;  template <class T>  struct binaryTreeNode  {  T element;  binaryTreeNode \*leftchild;  binaryTreeNode \*rightchild;  binaryTreeNode()  {  element = 0;  leftchild = NULL;  rightchild = NULL;  }  binaryTreeNode(const T &theelement)  {  element = theelement;  leftchild = NULL;  rightchild = NULL;  }  binaryTreeNode(const T &theelement, binaryTreeNode \*l, binaryTreeNode \*r)  {  element = theelement;  leftchild = l;  rightchild = r;  }  };  template <class T>  class minheap  {  private:  int size;  T \*heap;  public:  minheap()  {  heap = new T[7001];  size = 0;  }  int fix(int index)  {  if (index \* 2 > size)  return -1;  if (index \* 2 + 1 > size)  {  if (heap[index \* 2]->element < heap[index]->element)  {  T temp = heap[index];  heap[index] = heap[index \* 2];  heap[index \* 2] = temp;  return index \* 2;  }  else  {  return -1;  }  }  else  {  if (heap[index]->element < heap[index \* 2]->element && heap[index]->element < heap[index \* 2 + 1]->element)  {  return -1;  }  else  {  if (heap[index \* 2 + 1]->element < heap[index \* 2]->element)  {  T temp = heap[index];  heap[index] = heap[index \* 2 + 1];  heap[index \* 2 + 1] = temp;  return index \* 2 + 1;  }  else  {  T temp = heap[index];  heap[index] = heap[index \* 2];  heap[index \* 2] = temp;  return index \* 2;  }  }  }  }  T &top()  {  return heap[1];  }  void push(const T &value)  {  heap[++size] = value;  int p = size;  while (p != 1)  {  int x = fix(p / 2);  if (x == -1)  break;  p = p / 2;  }  }  void pop()  {  if (size == 0)  return;  heap[1] = heap[size--];  int p = 1;  while (1)  {  p = fix(p);  if (p == -1)  break;  }  }  };  template <class T>  class binaryTree  {  private:  int num;  public:  binaryTreeNode<T> \*root;  binaryTree(binaryTreeNode<T> \*r)  {  root = r;  }  int computehuffman()  {  if (root == NULL)  return 0;  int ans = 0;  precompute(root, 0, ans);  return ans;  }  void precompute(binaryTreeNode<T> \*t, int h, int &ans)  {  if (t->leftchild == NULL && t->rightchild == NULL)  {  ans += h \* (t->element);  return;  }  if (t->leftchild != NULL)  {  precompute(t->leftchild, h + 1, ans);  }  if (t->rightchild != NULL)  {  precompute(t->rightchild, h + 1, ans);  }  }  };  int main()  {  int f[26] = {0};  minheap<binaryTreeNode<int> \*> p;  string s;  cin >> s;  for (int i = 0; i < s.length(); i++)  {  f[s[i] - 97]++;  }  binaryTreeNode<int> \*\*node = new binaryTreeNode<int> \*[26];  for (int i = 0; i < 26; i++)  {  node[i] = new binaryTreeNode<int>;  }  int num = 0;  for (int i = 0; i < 26; i++)  {  node[i]->element = f[i];  if (node[i]->element != 0)  {  p.push(node[i]);  num++;  }  }  for (int i = 1; i <= num - 1; i++)  {  binaryTreeNode<int> \*x = p.top();  int xe = p.top()->element;  p.pop();  binaryTreeNode<int> \*y = p.top();  int ye = p.top()->element;  p.pop();  binaryTreeNode<int> \*t = new binaryTreeNode<int>(xe + ye, x, y);  p.push(t);  }  binaryTree<int> tree(p.top());  cout << tree.computehuffman();  } | | | |