山东大学 软件 学院

数据结构 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201400301034 | 姓名： 石兴帮 | | 班级： 软件八班 |
| 实验题目：堆和搜索树 | | | |
| 实验学时：4h | | 实验日期： 2015－12-13 | |
| 实验目的：  掌握堆和搜索树的基本概念，插入，删除方法 | | | |
| 硬件环境：  MacBook Pro  OS X Yosemite 10.10.3 | | | |
| 软件环境：  Xcode 6.4 | | | |
| 实验内容与设计：   1. 实验内容（题目内容，输入要求，输出要求）   1.创建最大堆类，最大堆的存储结构使用链表。  2.提供操作：堆的插入，堆的删除，堆的初始化。Huffman树的构造，二叉搜索树的构造。  3.根据输入的一系列整数，输出其对应的最大堆，Huffman编码，二叉搜索树。  4.堆排序。  2.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）  最大堆的插入：尝试向堆的后一个位置插入元素，其和父节点比较大小，不断往上比较，被比下来的父节点向下一级，新元素则上升，直到找到合适位置。  最大堆的删除：从顶点删除元素后，在其左右节点中选一个大的作为新的根节点，然后访问它产生的空缺位置，在它的左右节点中再找，依次向下，直至结束。  最大堆的初始化：接受一系列整数，依次插进去。    Huffman树的构造：将元素存入数组，每次从数组中选出两个最小元素，用它们MakeTree，然后把这个小tree的根节点元素用这两个元素的和表示，并把这个值再放入数组，进行n－1次这样的操作。  二叉搜索树的构造：插入元素时，从根节点位置开始，先匹配当前准备插进去的元素，小则向左，大则向右，直到不能再进下一层，将元素插入。  输出Huffman编码：遍历已经构造好了的Huffman树，对于每个元素的遍历过程，向左一次加个0，向右一次价格1，使用堆栈实现，将路径入栈，访问完出栈。  堆排序：将接受的一系列整数构造成二叉搜索树再中序输出。  3.测试结果（测试输入，测试输出）    **输入堆/树的大小:**  4  **输入堆/树中的各个元素:**  2 4 9 5  **初始化出的最大堆:**  **9 5 2 4**  **初始化出的二叉搜索树:**  **2 4 5 9**  **初始化出的霍夫曼编码:**  **2 4 9 5**  **9对应的编码是: 0**  **5对应的编码是: 1 0**  **2对应的编码是: 1 1 0**  **4对应的编码是: 1 1 1**  **输入最大堆中要删除的元素:**  5  **删除元素后的最大堆:**  **9 4 2**  **输入最大堆中要插入的元素:**  3  **插入元素后的最大堆:**  **9 4 2 3**  **堆排序。输入一串数字的长度:**  4    **输入这些数字:**  2 3 4 1  **堆排序后:**  1 2 3 4    4.实现源代码（程序风格清晰易理解，有充分的注释）  //  //  shiyan7.h  //  c++初体验  //  //  Created by SXB on 15/11/22.  //  Copyright (c) 2015年 apple. All rights reserved.  //  #ifndef c\_\_\_\_\_\_shiyan7\_h  #define c\_\_\_\_\_\_shiyan7\_h  #include<iostream>  #include "shiyan6.h"  //  因为我要用到实验六里的BinaryTree  #include "shiyan5.h"  //  因为我又想用堆栈了。。。  using namespace std;  //  MARK:  最大堆  template<class T>  class MaxHeap  {  public:      MaxHeap(int MaxHeapSize = 10);      ~MaxHeap() {delete [] heap;}      int Size() const { return CurrentSize;}      T Max() {          if (CurrentSize == 0) {  //            throw OutOfBounds()          }          return heap[1];      }      MaxHeap<T>& Insert(const T& x );      MaxHeap<T>& DeleteMax(T& x );      MaxHeap<T>& Delete(T x);      void Initialize(T a[], int size, int ArraySize);      void Output();        int CurrentSize, MaxSize;      T \*heap; //  元素数组  };  template<class T>  MaxHeap<T>::MaxHeap(int MaxHeapSize)  {  //    构造函数      MaxSize = MaxHeapSize;      heap = new T[MaxSize + 1];      CurrentSize = 0;  }  template<class T>  MaxHeap<T>& MaxHeap<T>::Insert(const T& x )  {  //    把 x插入到最大堆中      if (CurrentSize == MaxSize) {  //        throw NoMem()      }    //    为 x寻找应插入位置  //    i从新的叶节点开始， 并沿着树上升      int i = ++CurrentSize;      while (i != 1 && x > heap[i/2]) {  //        不能够把x 放入 heap[i]          heap[i] = heap[i/2];   //  将元素下移          i /= 2;                //  移向父节点      }      heap[i] = x;      return \*this;  }  template<class T>  MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x )  {  //    将最大元素放入x，并从堆中删除最大元素  //    检查堆是否为空      if (CurrentSize == 0) {  //        throw OutOfBounds();      }        x = heap[1];                //  最大元素    //    重构堆（将右下角元素放入由于上移缺省掉的位置，保持原形态）      T y = heap[CurrentSize--];  //  最后一个元素    //    从根开始，为y寻找合适的位置      int i = 1;      //  堆的当前节点      int ci = 2;     //  i的孩子      while (ci <= CurrentSize) {  //        heap[ci]应是i的较大的孩子          if (ci < CurrentSize && heap[ci] < heap[ci+1]) {              ci++;          }    //        能把y放入heap[i]吗?          if (y >= heap[ci]) {              break;      //  能          }    //        不能          heap[i] = heap[ci];   //  将孩子上移          i = ci;               //  下移一层          ci \*= 2;      }      heap[i] = y;        return \*this;  }  template<class T>  MaxHeap<T>& MaxHeap<T>::Delete(T x)  {      int i=1;      while (heap[i]) {          if (heap[i] == x) {              break;          }          i++;      }  //    重构堆      T y = heap[CurrentSize--];      //  最后一个元素      heap[CurrentSize+1] = 0;  //    从根开始，为y 寻找合适的位置  //    i为堆的当前节点      int ci = 2\*i;  // i的孩子      while (ci <= CurrentSize) {          if (ci < CurrentSize && heap[ci] < heap[ci+1]) {              ci++;          }            if (y >= heap[ci]) {              break;          }            heap[i] = heap[ci];          i = ci;          ci \*= 2;      }      heap[i] = y;        return \*this;  }  template<class T>  void MaxHeap<T>::Initialize(T a[], int size, int ArraySize)  {  //    把最大堆初始化为数组 a      delete [] heap;      heap = new T[ArraySize+1];      for (int i=1; i<size+1; i++) {          heap[i] = a[i-1];      }      CurrentSize = size;      MaxSize = ArraySize;    //    产生一个最大堆      for (int i = CurrentSize/2; i >= 1; i--) {          T y = heap[i];              //  子树的根    //        寻找放置y的位置          int c = 2\*i;                // c的父节点是y的目标位置          while (c <= CurrentSize) {  //            heap[c]应是较大的同胞节点              if (c < CurrentSize && heap[c] < heap[c+1]) {                  c++;              }  //            把 y 放入heap[c/2]              if (y >= heap[c]) {                  break;              //  能              }    //            不能              heap[c/2] = heap[c];    //  将孩子上移              c \*= 2;                 //  下移一层          }            heap[c/2] = y;      }  }  template<class T>  void MaxHeap<T>::Output()  {      int i=1;      while (heap[i] != 0) {          cout << heap[i] << " ";          i++;      }  }  //  MARK:  堆排序  template<class T>  void HeapSort(T a[], int n)  {      MaxHeap<T> H(1);      H.Initialize(a, n ,n);        T x;      for (int i=n; i>=1; i--) {          H.DeleteMax(x);          a[i-1] = x;      }        cout << "堆排序后: " << endl;      for (int i=0; i<n; i++) {          cout << a[i] << " ";      }  }  //  MARK: 最小堆  template<class T>  class MinHeapNode  {      MinHeapNode()      {          LeftChild = RightChild = 0;      }      MinHeapNode(const T& e)      {          data = e;          LeftChild = RightChild = 0;      }      MinHeapNode(const T& e,MinHeapNode \*l, MinHeapNode \*r){          data = e;          LeftChild = l;          RightChild = r;      }      T data;      MinHeapNode<T> \*LeftChild;      MinHeapNode<T> \*RightChild;  };  template<class T>  class MinHeap {  public:      MinHeap(int MinHeapSize = 10);      ~MinHeap() {delete [] heap;}      int Size() const {return CurrentSize;}      T Min() {          if (CurrentSize == 0) {              //            throw OutOfBounds();          }          return heap[1];      }      MinHeap<T>& Insert(const T& x);      MinHeap<T>& DeleteMin(T &x);      void Initialize(T a[], int size, int ArraySize);      void Output();    private:      int CurrentSize, MaxSize;      T \*heap;      MinHeapNode<T> \*root;  };  template<class T>  MinHeap<T>::MinHeap(int MinHeapSize)  {      MaxSize = MinHeapSize;      heap = new T[MaxSize + 1];  //    heap = new MinHeapNode<T>[MaxSize + 1];      CurrentSize = 0;      root = 0;  }  template<class T>  MinHeap<T>& MinHeap<T>::Insert(const T& x )  {  //    MinHeapNode<T> tmp\_x = new MinHeapNode<T>(x);  //  //    MinHeapNode<T> father = heap[(++CurrentSize)/2];  //    if (CurrentSize % 2 == 0) {  //        father =  MinHeapNode<T>(father->data, heap[CurrentSize-1], tmp\_x);  //    }else  //        father = MinHeapNode<T>(father->data, tmp\_x, NULL);  //  //    if (CurrentSize == MaxSize) {  //        //        throw NoMem();  //    }  //  //    int i = ++CurrentSize;  //    while (i != 1 && x < heap[i/2]->data) {  //        heap[i]->data = heap[i/2]->data;  //        i /= 2;  //    }  //  //    heap[i]->data = x;  //    return \*this;      if (CurrentSize == MaxSize) {  //        throw NoMem();      }      int i = ++CurrentSize;      while (i != 1 && x.weight < heap[i/2].weight) {          heap[i] = heap[i/2];          i /= 2;      }      heap[i] = x;      return \*this;  }  template<class T>  MinHeap<T>& MinHeap<T>::DeleteMin(T &x)  {      if (CurrentSize == 0) {          //        throw OutOfBounds();      }        x = heap[1];        T y = heap[CurrentSize--];        int i= 1, ci = 2;      while (ci <= CurrentSize) {          if (ci < CurrentSize && heap[ci].weight > heap[ci+1].weight) {              ci++;          }            if (y.weight <= heap[ci].weight) {              break;          }            heap[i] = heap[ci];          i = ci;          ci \*= 2;      }      heap[i] = y;        return \*this;  }  template<class T>  void MinHeap<T>::Initialize(T a[], int size, int ArraySize)  {      delete [] heap;      heap = new T[ArraySize+1];      for (int i=1; i<size+1; i++) {          heap[i] = a[i];      }      CurrentSize = size;      MaxSize = ArraySize;        for (int i = CurrentSize/2; i >= 1; i--) {          T y = heap[i];              //  子树的根            //        寻找放置y的位置          int c = 2\*i;                // c的父节点是y的目标位置          while (c <= CurrentSize) {              if (c < CurrentSize && heap[c].weight > heap[c+1].weight) {                  c++;              }              //            把 y 放入heap[c/2]              if (y.weight <= heap[c].weight) {                  break;              //  能              }                //            不能              heap[c/2] = heap[c];    //  将孩子上移              c \*= 2;                 //  下移一层          }            heap[c/2] = y;      }  }  template<class T>  void MinHeap<T>::Output()  {      for (int i=1; i<=CurrentSize; i++) {          cout << heap[i].weight << " ";      }  }  template<class T>  class Huffman {      friend BinaryTree<int> HuffmanTree(T [], int);    public:      operator T() const { return weight;}    //private:      BinaryTree<int> tree;      T weight;  };  template<class T>  BinaryTree<int> HuffmanTree(T a[], int n)  {  //    根据权重 a[1:n] 构造霍夫曼树  //    创建一个单节点树的数组      T aa[n+1];      for (int i=0; i<n; i++) {          aa[i+1] = a[i];      }        Huffman<T> \*w = new Huffman<T>[n+1];      BinaryTree<int> z,zero;      for (int i = 1; i <= n; i++) {          z.MakeTree(aa[i], zero, zero);          w[i].weight = aa[i];          w[i].tree = z;      }    //    把数组变成一个最小堆      MinHeap<Huffman<T>> H(1);      H.Initialize(w, n, n);      H.Output();      //  标记一下。这里的w的类型是Huffman，应该不能直接用w来互相比大小吧。  //    如果我用w的weight属性来建树呢。  //    如果Huffman可以直接比大小的话，w会变成一个最小堆。元素都是Huffman，H确实可以直接提元素出来合并了。    //    将堆中的树不断合并      Huffman<T> x,y;      for (int i = 1; i < n; i++) {          H.DeleteMin(x);          H.DeleteMin(y);          z.MakeTree(0, x.tree, y.tree);          x.weight += y.weight;          x.tree= z;          H.Insert(x);      }        H.DeleteMin(x);  //  最后的树      delete [] w;      return x.tree;  }  Stack<int> \*sta = new Stack<int>(100);  void OutputHuffman(BinaryTreeNode<int> \*bt)  {      int x;      if (bt) {          if (bt->data != 0) {              cout << bt->data << "对应的编码是: ";              sta->Output2();              cout << endl;          }            if (bt->LeftChild) {              sta->Add(0);              OutputHuffman(bt->LeftChild);              sta->Delete(x);          }            if (bt->RightChild) {              sta->Add(1);              OutputHuffman(bt->RightChild);              sta->Delete(x);          }      }  }  //  MARK:  二叉搜索树  template<class T>  class BSTree: BinaryTree<T> {  public:      bool Search(const T& t, T& x) const;      BSTree<T>& Insert(const T& t);      BSTree<T>& Delete(const T& t);        void InOrder(void(\*Visit) (BinaryTreeNode<T> \*u ))      {          InOrder(Visit, this->root );      }      void InOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t );    };  template<class T>  void BSTree<T>::InOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t)  {      //    中序遍历      if (t) {          InOrder(Visit, t->LeftChild);          Visit(t);          //        treeSize++;          InOrder(Visit, t->RightChild);      }  }  template<class T>  bool BSTree<T>::Search(const T& t, T& x) const  {  //    搜索与k匹配的元素  //    指针p从树根开始进行查找      BinaryTreeNode<T> \*p = this->root;      while (p) {          if (t < p->data) {              p = p->LeftChild;          }          else if (t > p->data)              p = p->RightChild;          else {              x = p->data;              return true;          }      }      return false;  }  template<class T>  BSTree<T>& BSTree<T>::Insert(const T& t)  {      BinaryTreeNode<T> \*p = this->root, \*pp=0;        while (p) {          pp = p;          if (t <= p->data) {              p = p->LeftChild;          }          else if (t > p->data)              p = p->RightChild;          else{  //            throw BadInput();          }      }        BinaryTreeNode<T> \*r = new BinaryTreeNode<T>(t);      if (this->root) {          if (t < pp->data) {              pp->LeftChild = r;          }          else              pp->RightChild = r;      }      else          this->root = r;        return \*this;  }  //  MARK:  实验6main方法  //#include "shiyan7.h"  //  //int main()  //{  //    //    int t[5] = {3,5,2,1,4};  //    int a;  //  //    int treesiz;  //    cout << "输入堆/树的大小: " << endl;  //    cin >> treesiz;  //    int \*treenodes = new int[treesiz];  //  //    cout << "输入堆/树中的各个元素: " << endl;  //    for (int i=0; i<treesiz; i++) {  //        cin >> treenodes[i];  //    }  //    MaxHeap<int> \*mh = new MaxHeap<int>(30);  //  //    mh->Initialize(treenodes , treesiz, 30);  //  //    cout << "初始化出的最大堆: " << endl;  //    mh->Output();  //    cout << endl;  //  //    cout << "初始化出的二叉搜索树: " << endl;  //    BSTree<int> \*bst = new BSTree<int>();  //    for (int i=0; i<treesiz; i++) {  //        bst->Insert(treenodes[i]);  //    }  //    bst->InOrder(ct );  //    cout << endl;  //  //    cout << "初始化出的霍夫曼编码:" << endl;  //    BinaryTree<int> test = HuffmanTree(treenodes , treesiz);  //    cout << endl;  //    OutputHuffman(test.root);  //  //    cout << "输入最大堆中要删除的元素: " << endl;  //    cin >> a;  //    mh->Delete(a);  //    cout << "删除元素后的最大堆:" << endl;  //    mh->Output();  //    cout << endl;  //  //    cout << "输入最大堆中要插入的元素: " << endl;  //    cin >> a;  //    mh->Insert(a );  //    cout << "插入元素后的最大堆: " << endl;  //    mh->Output();  //    cout << endl;  //  //  //    cout << "堆排序。输入一串数字的长度:" << endl;  //    cin >> a;  //    treenodes = new int[a];  //    cout << endl;  //    cout << "输入这些数字: " << endl;  //    for (int i=0; i<a; i++) {  //        cin >> treenodes[i];  //    }  //    HeapSort(treenodes, a);  //  //}  #endif | | | |
| 结论分析与体会：  我学到了很多 | | | |