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：202000130143	姓名： 郑凯饶	班级： 计科 20.1
实验题目：堆及其应用		
实验学时：2	实验日期： 1201	

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：

Windows 10 家庭中文版 64 位（10.0，版本 18363）

Dev-C++ IDE

1. 实验内容
2. 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。
3. 哈夫曼编码

4. 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>

```
template<class T>
class minHeap {
    public:
        minHeap(int ini = 10);
        ~minHeap() { delete [] heap; }
        bool empty() const { return Size == 0; }
        int size() const { return Size; }
        const T& top() {
            if (Size == 0) { }
            return heap[1];
        }

        void pop();
        void push(const T&);
        void initialize(T *, int);

        void deactivate() {
            heap = NULL; Length = Size = 0;
        }
        void output(ostream& out) const;
    private:
        int Size;
        int Length;
        T * heap;
};
```

Private:

Heap: 由于堆是完全二叉树，数组存储可以很好地利用空间，并且访问简单

Public:

Top(): 返回堆顶元素，即 heap[1]

Pop(): 弹出堆顶元素，具体实现如下：

```cpp
template<class T>
void minHeap<T>::pop() {
    if (Size == 0) { return; }
    heap[1].~T();

    T last = heap[Size--];
    int cur = 1, child = 2;

    // 向下遍历
    while (child <= Size) {
        if (child < Size && heap[child] > heap[child + 1]) child++;

        if (last <= heap[child]) break;
        heap[cur] = heap[child];
        cur = child;
        child *= 2; // cur = child / 2
    }
    heap[cur] = last;

}
```

将下标为 Size 的元素取代堆顶元素，并“下沉”，选择当前节点及其左右孩子的最小者换至当前元素位置，由于是完全二叉树节点和孩子的关系为$\mathrm{child}_1 = \mathrm{curNode} * 2$和$\mathrm{child}_2 = curNode * 2 + 1$, 以上为“下沉”细节。

Push()：往小根堆中加入元素，将节点添至数组最后位置，然后执行“上浮”操作，与 pop() 操作相似。

Initialize()：O(n)初始化小根堆，具体实现如下：

```cpp
template<class T>
void minHeap<T>::initialize(T *theheap, int theSize) {
    delete [] heap;
    heap = theheap;
    Size = theSize;
    // How about Length ?
    // it assums: theheap.Length == heap.Length

    // 堆化
    for (int root = Size / 2; root >= 1; root--) {
        T rEle = heap[root];

        int child = 2 * root ;
        while (child <= Size) {
            if (child < Size && heap[child] > heap[child + 1]) child++;
            if (rEle <= heap[child]) break;

            heap[child / 2] = heap[child];
            child *= 2;
        }
        heap[child / 2] = rEle;
    }
}
```

从最后一个有孩子的节点开始，“下沉”。

<u>复杂度具体计算：</u>

在树的第 i 层，最多有2^{j-1}个节点，它们的高度为$h_i = h - j + 1$.

于是初始化的时间为

$$\mathrm{O}\left(\sum_{j=1}^{h-1} 2^{j-1}(h-j+1)\right) = O(n)$$

2>

这里基本使用课程网站上的标程。

统计最终编码长度可以在霍夫曼树的建树过程中实现。

我们知道，最终编码长度等于 WEP，它的计算方式为

$$\mathrm{WEP} = \sum_{i=1}^{n} L(i) * F(i)$$

实际对应叶子节点的权重乘深度。

建树过程中的每次合并使得被合并节点的深度+1，最终长度应为每次合并节点的权重之和求和。

5. 测试结果（测试输入，测试输出）

在 OJ 平台上成功提交。

6. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

这次实验比较简单，我尝试使用局部的静态变量避免了全局变量的使用 hhhhh 另一方面，尝试计算了一下时间复杂度。

7. 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>
using namespace std;

template<class T>
class minHeap {
 public:
  minHeap(int ini = 10);
  ~minHeap() { delete [] heap; }
  bool empty() const { return Size == 0; }
  int size() const { return Size; }
  const T& top() {
   if (Size == 0) { }
   return heap[1];
  }

  void pop();
  void push(const T&);
  void initialize(T *, int);

  void deactivate() {
   heap = NULL; Length = Size = 0;
  }
  void output(ostream& out) const;
 private:
  int Size;
  int Length;
  T * heap;
};

template<class T>
minHeap<T>::minHeap(int iniCap) {
// if
 Length = iniCap + 1; // **
 heap = new T[Length];
 Size = 0;
}
```

```

template<class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
//   if (newLength < 0)
//      throw illegalParameterValue("new length must be >= 0");

   T* temp = new T[newLength];              // new array
   int number = min(oldLength, newLength);  // number to copy
   copy(a, a + number, temp);
   delete [] a;                             // deallocate old memory
   a = temp;
}

template<class T>
void minHeap<T>::push(const T& ele) {
 if (Size == Length - 1) {
  changeLength1D(heap, Length, 2 * Length);
  Length <<= 1;
 }

 int cur = ++Size;

 // 向上起泡
 while (cur != 1 && heap[cur / 2] > ele) {
  heap[cur] = heap[cur / 2];
  cur /= 2;
 }
 heap[cur] = ele;
}

template<class T>
void minHeap<T>::pop() {
 if (Size == 0) { return; }
 heap[1].~T();

 T last = heap[Size--];
 int cur = 1, child = 2;

 // 向下遍历
 while (child <= Size) {
  if (child < Size && heap[child] > heap[child + 1]) child++;

  if (last <= heap[child]) break;
  heap[cur] = heap[child];
  cur = child;
```

```
        child *= 2; // cur = child / 2
    }
    heap[cur] = last;

}

template<class T>
void minHeap<T>::initialize(T *theheap, int theSize) {
    delete [] heap;
    heap = theheap;
    Size = theSize;
    // How about Length ?
    // it assums: theheap.Length == heap.Length

    // 堆化
    for (int root = Size / 2; root >= 1; root--) {
        T rEle = heap[root];

        int child = 2 * root ;
        while (child <= Size) {
            if (child < Size && heap[child] > heap[child + 1]) child++;
            if (rEle <= heap[child]) break;

            heap[child / 2] = heap[child];
            child *= 2;
        }
        heap[child / 2] = rEle;
    }
}

template<class T>
void heapSort(T a[], int n) { // ** 原地重排  **
    minHeap<T> heap(n);
    heap.initialize(a, n);

    for (int i = n - 1; i >= 1; i--) {
        T x = heap.top(); heap.pop(); // heap.Size = i then we can put the min ele on the (i + 1) th.
        a[i + 1] = x;
    }
    heap.deactivate();
}

void solve() {
    int n; cin >> n;
    int *a = new int[n + 1];
```

```
    for (int i = 1; i <= n; i++) {
     cin >> a[i];
    }

    minHeap<int> hp(n);
    hp.initialize(a, n);
    cout << hp.top() << endl;

    int m; cin >> m;
    for (int i = 0, opt, num; i < m; i++) {
     cin >> opt;
     if (opt == 1) {
      cin >> num;
      hp.push(num);
      cout << hp.top() << endl;
     }
     else if (opt == 2) {
      hp.pop();
      cout << hp.top() << '\n';
     }
     else if (opt == 3) {
      int sz; cin >> sz;
      int *b = new int[sz + 1];
      for (int j = 1; j <= sz; j++) cin >> b[j];
      heapSort(b, sz);
      for (int j = sz; j >= 1; j--) cout << b[j] << " ";
      cout << endl;
     }
    }
   }

   int main() {
    solve();
    return 0;
   }





   #include<bits/stdc++.h>
   using namespace std;

   // huffman 树本质上是二叉树，其构造是特殊的二叉树建树过程
   template <class T>
   struct binaryTreeNode
```

```
{
   T element;
   binaryTreeNode<T> *leftChild,   // left subtree
                     *rightChild;  // right subtree

   binaryTreeNode() {leftChild = rightChild = NULL;}
   binaryTreeNode(const T& theElement):element(theElement)
   {
      leftChild = rightChild = NULL;
   }
   binaryTreeNode(const T& theElement,
                  binaryTreeNode *theLeftChild,
                  binaryTreeNode *theRightChild)
                  :element(theElement)
   {
      leftChild = theLeftChild;
      rightChild = theRightChild;
   }
};


template<class E>
class linkedBinaryTree // : public binaryTree<binaryTreeNode<E> >
{
   public:
      linkedBinaryTree() {root = NULL; treeSize = 0;}
      ~linkedBinaryTree(){erase();};
      bool empty() const {return treeSize == 0;}
      int size() const {return treeSize;}
      E* rootElement() const;
      void makeTree(const E& element,
           linkedBinaryTree<E>&, linkedBinaryTree<E>&);
      linkedBinaryTree<E>& removeLeftSubtree();
      linkedBinaryTree<E>& removeRightSubtree();
      void preOrder(void(*theVisit)(binaryTreeNode<E>*))
           {visit = theVisit; preOrder(root);}
      void inOrder(void(*theVisit)(binaryTreeNode<E>*))
           {visit = theVisit; inOrder(root);}
      void postOrder(void(*theVisit)(binaryTreeNode<E>*))
           {visit = theVisit; postOrder(root);}
      void levelOrder(void(*)(binaryTreeNode<E> *));
      void preOrderOutput() {preOrder(output); cout << endl;}
      void inOrderOutput() {inOrder(output); cout << endl;}
      void postOrderOutput() {postOrder(output); cout << endl;}
      void levelOrderOutput() {levelOrder(output); cout << endl;}
      void erase()
```

```
           {
              postOrder(dispose);
              root = NULL;
              treeSize = 0;
           }
      int height() const {return height(root);}
   protected:
      binaryTreeNode<E> *root;                // pointer to root
      int treeSize;                           // number of nodes in tree
      static void (*visit)(binaryTreeNode<E>*);      // visit function
      static int count;         // used to count nodes in a subtree
      static void preOrder(binaryTreeNode<E> *t);
      static void inOrder(binaryTreeNode<E> *t);
      static void postOrder(binaryTreeNode<E> *t);
      static void countNodes(binaryTreeNode<E> *t)
                  {
                     visit = addToCount;
                     count = 0;
                     preOrder(t);
                  }
      static void dispose(binaryTreeNode<E> *t) {delete t;}
      static void output(binaryTreeNode<E> *t)
                  {cout << t->element << ' ';}
      static void addToCount(binaryTreeNode<E> *t)
                  {count++;}
      static int height(binaryTreeNode<E> *t);
};
// the following should work but gives an internal compiler error
 template <class E> void (*linkedBinaryTree<E>::visit)(binaryTreeNode<E>*);
// so the explicit declarations that follow are used for our purpose instead
//void (*linkedBinaryTree<int>::visit)(binaryTreeNode<int>*);
//void (*linkedBinaryTree<booster>::visit)(binaryTreeNode<booster>*);
//void (*linkedBinaryTree<pair<int,int> >::visit)(binaryTreeNode<pair<int,int> >*);
//void (*linkedBinaryTree<pair<const int,char> >::visit)(binaryTreeNode<pair<const int,char> >*);
//void (*linkedBinaryTree<pair<const int,int> >::visit)(binaryTreeNode<pair<const int,int> >*);

template<class E>
E* linkedBinaryTree<E>::rootElement() const
{// Return NULL if no root. Otherwise, return pointer to root element.
   if (treeSize == 0)
      return NULL;  // no root
   else
      return &root->element;
```

```
}

template<class E>
void linkedBinaryTree<E>::makeTree(const E& element,
           linkedBinaryTree<E>& left, linkedBinaryTree<E>& right)
{// Combine left, right, and element to make new tree.
 // left, right, and this must be different trees.
   // create combined tree
   root = new binaryTreeNode<E> (element, left.root, right.root);
   treeSize = left.treeSize + right.treeSize + 1;

   // deny access from trees left and right
   left.root = right.root = NULL;
   left.treeSize = right.treeSize = 0;
}

template<class E>
linkedBinaryTree<E>& linkedBinaryTree<E>::removeLeftSubtree()
{// Remove and return the left subtree.
   // check if empty
//   if (treeSize == 0)
//      throw emptyTree();

   // detach left subtree and save in leftSubtree
   linkedBinaryTree<E> leftSubtree;
   leftSubtree.root = root->leftChild;
   count = 0;
   leftSubtree.treeSize = countNodes(leftSubtree.root);
   root->leftChild = NULL;
   treeSize -= leftSubtree.treeSize;

   return leftSubtree;
}

template<class E>
linkedBinaryTree<E>& linkedBinaryTree<E>::removeRightSubtree()
{// Remove and return the right subtree.
   // check if empty
//   if (treeSize == 0)
//      throw emptyTree();

   // detach right subtree and save in rightSubtree
   linkedBinaryTree<E> rightSubtree;
   rightSubtree.root = root->rightChild;
   count = 0;
   rightSubtree.treeSize = countNodes(rightSubtree.root);
```

```
        root->rightChild = NULL;
        treeSize -= rightSubtree.treeSize;

        return rightSubtree;
    }

    template<class E>
    void linkedBinaryTree<E>::preOrder(binaryTreeNode<E> *t)
    {// Preorder traversal.
        if (t != NULL)
        {
            linkedBinaryTree<E>::visit(t);
            preOrder(t->leftChild);
            preOrder(t->rightChild);
        }
    }

    template<class E>
    void linkedBinaryTree<E>::inOrder(binaryTreeNode<E> *t)
    {// Inorder traversal.
        if (t != NULL)
        {
            inOrder(t->leftChild);
            linkedBinaryTree<E>::visit(t);
            inOrder(t->rightChild);
        }
    }

    template<class E>
    void linkedBinaryTree<E>::postOrder(binaryTreeNode<E> *t)
    {// Postorder traversal.
        if (t != NULL)
        {
            postOrder(t->leftChild);
            postOrder(t->rightChild);
            linkedBinaryTree<E>::visit(t);
        }
    }

    template <class E>
    void linkedBinaryTree<E>::levelOrder(void(*theVisit)(binaryTreeNode<E> *))
    {// Level-order traversal.
        queue<binaryTreeNode<E>*> q;
        binaryTreeNode<E> *t = root;
        while (t != NULL)
        {
```

```
         theVisit(t);  // visit t

         // put t's children on queue
         if (t->leftChild != NULL)
            q.push(t->leftChild);
         if (t->rightChild != NULL)
            q.push(t->rightChild);

         // get next node to visit
//       try {t = q.front();}
//       catch (queueEmpty) {return;}

         if (!q.empty()) {
          t = q.front();
      }
      else return;

         q.pop();
      }
}

template <class E>
int linkedBinaryTree<E>::height(binaryTreeNode<E> *t)
{// Return height of tree rooted at *t.
   if (t == NULL)
      return 0;                    // empty tree
   int hl = height(t->leftChild);  // height of left
   int hr = height(t->rightChild); // height of right
   if (hl > hr)
      return ++hl;
   else
      return ++hr;
}


// 最小堆
template<class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
//   if (newLength < 0)
//      throw illegalParameterValue("new length must be >= 0");

   T* temp = new T[newLength];               // new array
   int number = min(oldLength, newLength);   // number to copy
   copy(a, a + number, temp);
   delete [] a;                              // deallocate old memory
```

```cpp
      a = temp;
   }

   template<class T>
   class minHeap // : public minPriorityQueue<T>
   {
      public:
         minHeap(int initialCapacity = 10);
         ~minHeap() {delete [] heap;}
         bool empty() const {return heapSize == 0;}
         int size() const
             {return heapSize;}
         const T& top()
            {// return min element
//             if (heapSize == 0)
//                throw queueEmpty();
               return heap[1];
            }
         void pop();
         void push(const T&);
         void initialize(T *, int);
         void deactivateArray()
            {heap = NULL; arrayLength = heapSize = 0;}
         void output(ostream& out) const;
      private:
         int heapSize;       // number of elements in queue
         int arrayLength;    // queue capacity + 1
         T *heap;            // element array
   };

   template<class T>
   minHeap<T>::minHeap(int initialCapacity)
   {// Constructor.
//    if (initialCapacity < 1)
//    {ostringstream s;
//     s << "Initial capacity = " << initialCapacity << " Must be > 0";
//     throw illegalParameterValue(s.str());
//    }
      arrayLength = initialCapacity + 1;
      heap = new T[arrayLength];
      heapSize = 0;
   }

   template<class T>
   void minHeap<T>::push(const T& theElement)
   {// Add theElement to heap.
```

```

    // increase array length if necessary
    if (heapSize == arrayLength - 1)
    {// double array length
       changeLength1D(heap, arrayLength, 2 * arrayLength);
       arrayLength *= 2;
    }

    // find place for theElement
    // currentNode starts at new leaf and moves up tree
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode / 2] > theElement)
    {
       // cannot put theElement in heap[currentNode]
       heap[currentNode] = heap[currentNode / 2]; // move element down
       currentNode /= 2;                          // move to parent
    }

    heap[currentNode] = theElement;
 }

 template<class T>
 void minHeap<T>::pop()
 {// Remove max element.
    // if heap is empty return null
 //   if (heapSize == 0)   // heap empty
 //      throw queueEmpty();

    // Delete min element
    heap[1].~T();

    // Remove last element and reheapify
    T lastElement = heap[heapSize--];

    // find place for lastElement starting at root
    int currentNode = 1,
        child = 2;     // child of currentNode
    while (child <= heapSize)
    {
       // heap[child] should be smaller child of currentNode
       if (child < heapSize && heap[child] > heap[child + 1])
          child++;

       // can we put lastElement in heap[currentNode]?
       if (lastElement <= heap[child])
          break;   // yes
```

```

         // no
         heap[currentNode] = heap[child]; // move child up
         currentNode = child;             // move down a level
         child *= 2;
      }
      heap[currentNode] = lastElement;
   }

   template<class T>
   void minHeap<T>::initialize(T *theHeap, int theSize)
   {// Initialize max heap to element array theHeap[1:theSize].
      delete [] heap;
      heap = theHeap;
      heapSize = theSize;

      // heapify
      for (int root = heapSize / 2; root >= 1; root--)
      {
         T rootElement = heap[root];

         // find place to put rootElement
         int child = 2 * root; // parent of child is target
                               // location for rootElement
         while (child <= heapSize)
         {
            // heap[child] should be smaller sibling
            if (child < heapSize && heap[child] > heap[child + 1])
               child++;

            // can we put rootElement in heap[child/2]?
            if (rootElement <= heap[child])
               break;  // yes

            // no
            heap[child / 2] = heap[child]; // move child up
            child *= 2;                    // move down a level
         }
         heap[child / 2] = rootElement;
      }
   }

   template<class T>
   void minHeap<T>::output(ostream& out) const
   {// Put the array into the stream out.
      copy(heap + 1, heap + heapSize + 1, ostream_iterator<T>(cout, "  "));
```

```
}

// overload <<
template <class T>
ostream& operator<<(ostream& out, const minHeap<T>& x)
   {x.output(out); return out;}


template<class T>
struct huffmanNode {
 linkedBinaryTree<int> *tree;
 T weight;

 operator T () const { return weight; }
};

//int Ans = 0;

template <class T>
linkedBinaryTree<int>* huffmanTree(T weight[], int n, bool op)
{
 static int Ans = 0;
 if (op) {
  cout << Ans << '\n';
  return NULL;
 }

// Generate Huffman tree with weights weight[1:n], n >= 1.
   // create an array of single node trees
   huffmanNode<T> *hNode = new huffmanNode<T> [n + 1];
   linkedBinaryTree<int> emptyTree;
   for (int i = 1; i <= n; i++)
   {
      hNode[i].weight = weight[i];
      hNode[i].tree = new linkedBinaryTree<int>;
      hNode[i].tree->makeTree(i, emptyTree, emptyTree);
   }

   // make node array into a min heap
   minHeap<huffmanNode<T> > heap(1);
   heap.initialize(hNode, n);

   // repeatedly combine trees from min heap
   // until only one tree remains
   huffmanNode<T> w, x, y;
   linkedBinaryTree<int> *z;
```

```
   for (int i = 1; i < n; i++)
   {
      // remove two lightest trees from the min heap
      x = heap.top(); heap.pop();
      y = heap.top(); heap.pop();

      // combine into a single tree
      z = new linkedBinaryTree<int>;
      z->makeTree(0, *x.tree, *y.tree);
      w.weight = x.weight + y.weight;
      Ans += w.weight;
//      cout << Ans << '\n';
      w.tree = z;
      heap.push(w);
      delete x.tree;
      delete y.tree;
   }

   // destructor for min heap deletes hNode
   return heap.top().tree;
}


void solve() {
 string s; cin >> s;
 int len = s.length();

// if (len == 1) {
//  cout << "1\n";
//  return;
// }

 int *cnt = new int[27];
 int *w = new int[27];

// memset(cnt, 0, cnt + 27); // 1 - 26
 for (int i = 1; i < 27; i++) cnt[i] = 0;

 for (int i = 0; i < len; i++) {
  cnt[s[i] - 'a' + 1] ++;
 }

 // 离散化
 int one = 1;
 for (int i = 1; i < 27; i++) {
  if (cnt[i] != 0) {
```

```
   w[one] = cnt[i];
   one ++;
  }
 }

 // 不发生合并
 if (one == 2) {
  cout << len << '\n';
  return;
 }

 // 建立含 26 个小写字母节点的霍夫曼树 kill

 huffmanTree(w, one - 1, 0);

 huffmanTree(w, one - 1, 1);
}


int main(){
 solve();
 return 0;
}
```