5.6 Heaps(堆)

Also called Priority Queue（优先队列）

88

Priority Queue/优先队列

优先级高的先出队

When a collection of objects is organized by importance or priority, we call this a Priority Queue

基本操作：

Insert（Enqueue），插入一个新任务后依然需保持优先队列的特点

removeFirst（Dequeue)， 完成（删除）优先级最高任务后依然需保持优先队列的特点

实现：

一些简单的实现： list，BST

Heap（堆）： 普遍应用，和优先队列几乎被认为是同一个概念

89

Defination of Heaps

Complete binary tree whose node with property:

1) value of any node less than or equal to that of its children (Min-heap，小堆) or

2) value of any node larger than or equal to that of its children (Max-heap 大堆)

逻辑定义

90

Defination of Heap

因为堆是CBT，所以堆通常用基于数组的方式来实现，即将BT中的结点按层由低到高，层内由左到右进行编号并存放于1维数组中，其结点下标满足下列关系式：

- PARENT(i) = (i-1)/2; /*父结点*/
- LEFT(i) = 2i+1; /* 左子结点 */
- RIGHT(i) = 2i+2; /* 右子结点 */
- n_0 =(int) ((n+1)/2) ; /* 叶子结点的个数 */

物理定义

Defination：n个元素组成的序列$\{k_0,k_1,k_2,\ldots,k_{n-1}\}$,当且仅当满足下列关系之一时，称之为堆

1） $k_i \le k_{2i+1}$,且 $k_i \le k_{2i+2}$, i =0,1,…, n/2-1 小堆

2) $k_i \ge k_{2i+1}$,且 $k_i \ge k_{2i+2}$, i =0,1,…, n/2-1 大堆

91

Heap 与BST的区别

- **BST ：**
  - 左与右的关系
  - 不一定是**CBT**
  - 一般用基于指针的方式存储/实现
- **heap：**
  - 前辈与后辈的关系
  - 一定是**CBT**
  - 一般用基于数组的方式存储/实现

92

92

Heap

- **1个数组 +2个整型变量就可描述一个堆**
  - 1个数组存放heap中各结点的值
  - 1个整型变量maxSize存放数组的尺寸
  - 1个整型变量size存放 堆中的结点数
- **heap所涉及的基本操作**
  - Insert
  - remove
  - removeFirst
  - buildHeap

93

93

maxHeap class(1)---Array based implement

```
template<class Elem> class maxHeap {
private:
  Elem* Heap;   // Pointer to the heap array
  int maxSize;    // Maximum size of the heap
  int size;       // Number of elems now in heap
  void siftDown(int); // Put element in place
public:
  maxHeap(Elem* h, int num, int max) {
    size=num;   maxSize=max;   Heap = new int[max];  }
  int heapSize() const  {return size ; }
  bool isLeaf(int pos) const {
      return (pos >= size/2) && (pos < size);      }
  int leftChild(int pos) const  { return  2*pos+1; }
  int rightChild(int pos) const { return  2*pos+2; }
  int parent(int pos) const   {return  (pos-1)/2; }
  void print( ) const   { …    }
  void clear( )   { … }
  int find (const Elem&)  { … }
```

94

94

maxHeap class(2)---Array based implement

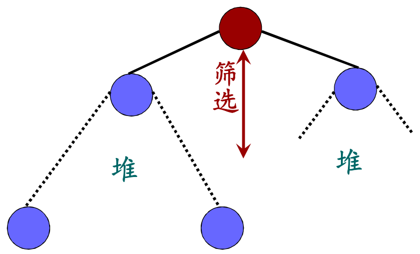
```
  void buildHeap();

  void insert(const Elem&);

  Elem removeFirst( );

  Elem remove(int);

};
```

95

95

Siftdown /筛选

所谓“siftdowm”指的是，对一棵左/右子树均为堆的完全二叉树，“调整”根结点使整个二叉树也成为一个堆。



96

SiftDown

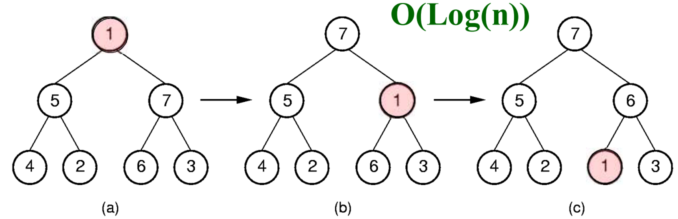
Siftdown/筛选的方法

S1：将根结点作为当前结点

S2：将当前结点值与其左、右子树的根结点值比较，并与三者中最大者进行交换；更新当前结点

S3：重复S2，直至叶子结点或无交换发生，所得结果即为堆。

O(Log(n))



97

maxHeap class(3)--- SiftDown

```
template <class Elem>
void maxHeap<Elem>::siftDown(int pos) {
  while  (!isLeaf(pos)) {
    int j = leftChild(pos);
    int rc = rightChild(pos);
    if ((rc<size) && (Heap[j]< Heap[rc]))
      j = rc;
    if ( Heap[pos] >= Heap[j] ) return;
    swap(Heap, pos, j); // 请自行写出该函数的代码
    pos = j;
  }
}
```

O(Log(n))

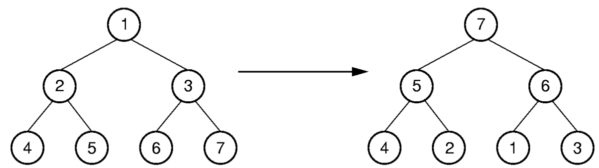
98

Building the MaxHeap

For fast heap construction:

•Call siftdown for each item from high end(尾端） of array to low end（前端） 从下往上/从后往前

•Don’t need to call siftDown on leaf nodes.



(7-3), (5-2), (7-1), (6-1) **共交换4次**

99

maxHeap class(4)--- BuildingHeap

```
template <class Elem>
void maxHeap<Elem>:: buildHeap( ) {
   for(i = size/2-1; i >= 0; i--)
        siftDown(i);
}
```

O(n)

f(n)的具体计算公式见课本p184

100

100

Insert a value in the MaxHeap

思路:

- 在堆末尾添加一取值为待插入值的叶子结点，作为当前结点，并size加1
- 将当前结点值与其双亲结点值比较，若大于则进行交换，并将其双亲作为当前节点;
- 重复上述操作，直至当前结点值小于等于其双亲结点值 或到达根结点。

101

101

maxHeap class(5)--- Insert

```
template <class Elem>
void maxHeap<Elem>:: insert(const Elem& e)
{
   Assert( size < maxSize, "Heap is full");
   int curr = size;
   Heap[curr] = e; size++;
   while(curr!=0 && Heap[curr]>Heap[parent(curr)]) {
       swap(Heap, curr,parent(curr)); curr=parent(curr);
   }
   return true;
}
```

O(Log(n))

102

102

Remove First value in the Maxheap

思路:

- 将根结点值与最末叶子结点值进行交换,并size减1
- 对根结点 做 siftDown 操作

103

103

maxHeap class (6)--Remove First Value

O(Log(n))

```
template <class Elem>
Elem maxHeap<Elem>::
removeFirst() {
  Assert ( size > 0, "Heap is empty");
  swap(Heap, 0, --size);    // Swap First with end
  if (size != 0) siftDown(0);
  return Heap[size];  // Return First value
}
```

104

104

Remove 给定下标位置的值从maxHeap

思路：

- 将待删除结点作为当前结点
- 将当前结点与最末叶子结点进行值交换，并size减1。
- 将当前结点值与其双亲结点值比较，若大于则进行交换，同时将其双亲作为当前节点；
- 重复上述操作，直至当前结点值小于其双亲结点值 或到达根结点。
- 对当前结点调用 siftDown

105

105

maxHeap class(7) --Remove

O(Log(n))

```
template <class Elem>
Elem maxHeap<Elem>::remove(int pos) {
  Assert((pos>=0)&&(pos<size),"Bad position");
  if ( pos == size-1 ) size--;
  else {
    swap(Heap, pos, --size);
    while ((pos != 0) &&
           (Heap[pos] > Heap[parent(pos)])){
      swap(Heap, pos, parent(pos));
      pos = parent(pos);
    }
    if (size != 0) siftDown(pos);  }
  return Heap[size];
}
```

106

106

An application example of heap

- 写一个程序，输入下列序列构建**maxHeap**，并测试插入，删除，查找，清空，打印等功能
- **1 2 3 4 5 6 7**
- **5 2 3 7 6 4 1**

输入序列顺序不同，构建的heap可能不同；但是，重复removeFirst 直到堆为空 得到的结果却是绝对相同的。

107

107

```
......
#include “heap.h“  // maxHead class--课件中PP91,92,95,97,101,103
using namespace std;
void main() {
     int i;   double a[100], temp;
     cout<<"please input 7 data:"<<endl;
     for(i=0;i<7;i++)   cin>>a[i];
     maxHeap<double>  h1(a,7,100);
     cout<<"after buildHeap the heap is:"<<endl;
     h1.buildHeap();  h1.print();     cout<<endl;
     cout<<"insert function test........"<<endl;
     cout<<"please input the insert data:";
     cin>>temp;   h1.insert(temp);
     cout<<"after insert "<<temp<<" the heap is:"<<endl;
     h1.print();  cout<<endl;
     cout<<"removeFirst function test........"<<endl;
     while(h1.heapSize()) {
        temp=h1.removeFirst( );  cout<<temp<<"  ";    }
     cout<<endl;
     ......
}
```

108

5.7 Huffman Coding Trees

5.7.1 Huffman Tree definition (哈夫曼树定义)

5.7.2 Huffman Tree Construction(哈夫曼树构造)

5.7.3 Huffman Coding (哈夫曼编码)

109

问题引入:

设给出一段报文 CAST CAST SAT AT A TASA

字符集合是 { C, A, S, T }，各个字符出现的频度(次数)是 W={ 2, 7, 4, 5 }

若给每个字符以等长编码

A : 00　T : 10　C : 01　S : 11

则报文总编码长度为 (2+7+4+5) * 2 = 36 bits

字符平均长度= 36/18 = 2 bits

若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度

A : 0　T : 10　C : 110　S : 111

它的总编码长度为

7*1+5*2+(2+4)*3 = 35 bits 。

字符平均长度= 35/18 = 1.944 bits　比等长编码的情形要短

哈夫曼编码

110

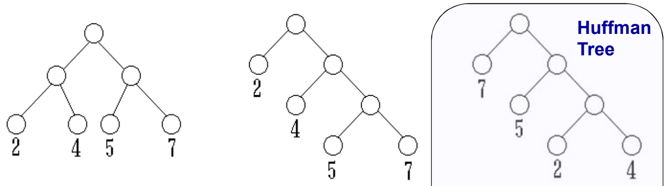
- 结点间路径长度(Path Length)
  连接两结点的路径上的分支数
- 结点的路径长度(又称结点的深度)
  从根结点到该结点的路径上分支的数目
- 叶子的加权路径长度（weighted path length of a leaf)
  叶子的深度与叶子的权值之积
- 树的外部路径权值 (External path weight)，也称树的加权路径长度(Weighted Path Length of a tree, WPL)
  树的所有叶结点的加权路径长度之和

$$WPL = \sum_{i=0}^{n_0-1} (w_i * l_i)$$

111

5.7.1 哈夫曼树定义 (Huffman Tree)

在所有 含n_0个带确定权值叶子结点的 二叉树中，必存在一棵加权路径长度最小的树，称该树为“最优树”,或“哈夫曼树”(Huffman Tree)



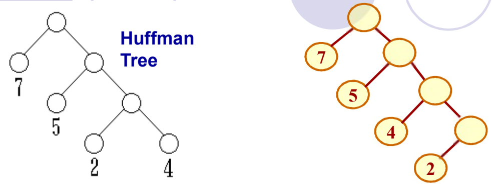
(a) $WPL = 36$ (b) $WPL = 46$ (c) $WPL = 35$

1）叶结点的权值越小，离根越远
2）叶结点的权值越大，离根越近

哈夫曼树的特点之一

112

哈夫曼树 (Huffman Tree)



(c) $WPL = 35$ WPL=37

Huffman Tree 特点一：
1）叶结点的权值越小，离根越远
2）叶结点的权值越大，离根越近

特点二：
满二叉树，FBT

113

5.7.2 Huffman Tree Construction

1. 根据给定的 n 个符号权值对 { $\{s_1, w_1\}, \{s_2, w_2\},, \ldots, \{s_n, w_n\}$, }，造 n 棵二叉树的集合 $F = \{T_1, T_2, \ldots, T_n\}$，其中每棵二叉树中均只含一个符号权 值对为 (s_i, w_i)的根结点，其左、右子树为 空树；
2. 在 F 中选取根结点权值最 小的两棵二叉树，分别作为左(最小)、右(次小)子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值 为其左、右子树根结点的权值之 和；
3. 从F中删去这两棵树，同时加入刚生成的新树；
4. 重复 (2) 和 (3) 两步，直至 F 中只含一棵树为止。

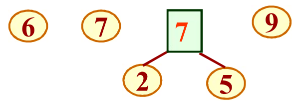
114

Huffman Tree Construction (2)

例1：已知字符集 { A, C,V, E, K }的出现频数为 { 5, 6, 2, 9, 7 }，以频数为权值构造哈夫曼树

1） 

F中有5棵树

2） 

F中有4棵树

3） 

F中有3棵树

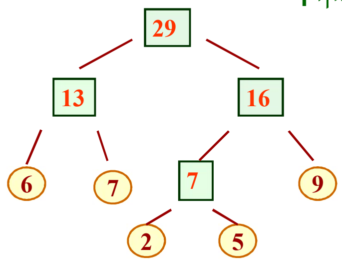
115

Huffman Tree Construction (3)

例1：已知字符集 { A, C,V, E, K }的出现频数为 { 5, 6, 2, 9, 7 }，以频数为权值构造哈夫曼树

4）

F中有1棵树



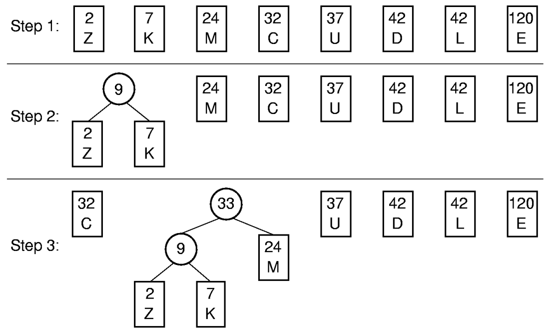
F中有2棵树

该树的加权路径长度WPL = ？

116

Huffman Tree Construction (4)

example2:

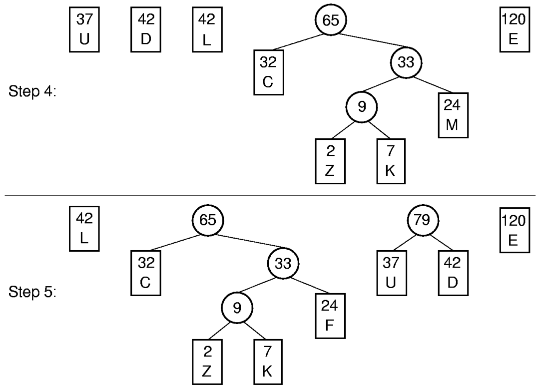


117

117

Huffman Tree Construction (5)

example2:

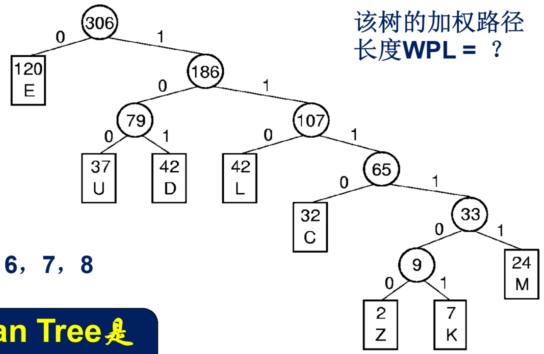


118

118

Huffman Tree Construction (6)

example2:



该树的加权路径长度WPL = ？

step 6，7，8

Huffman Tree是full(满)二叉树

119

119

哈夫曼树 (Huffman Tree)

若所有叶子权值相同，在huffman树中是否深度也相同呢？

字符集合: { A, B, C, D, E }
权值*W*: { 1, 1, 1, 1 , 1}

字符集合: { A, B, C, D, E, F, H, I }
权值*W*: { 1, 1, 1, 1 , 1, 1 , 1, 1 }

若所有叶节点权值相同且叶结点个数为2的整数次方，则对应的Huffman Tree是一个叶子节点均在最底层(即深度相同)的完全满二叉树

120

HuffmanTree class implement (pointer based)

需要定义3个class： FreqPair，HuffNode，HuffTree

FreqPair class

```
template <class Elem>
class FreqPair {
private:
        Elem symbol;
        int  freq;
public:
        FreqPair( Elem s, int f) {  symbol = s;  freq = f;}
        int weight() { return freq;  }
        Elem  val() { return symbol;  }
};
```

121

121

HuffNode class(1)

```
template <class Elem>
class HuffNode {      // Abstract base class,  可变类型结点
public:
        virtual  int weight()=0;
        virtual bool isLeaf() = 0;
};
template <class Elem>      // Leaf  of huffman tree
class LeafNode : public HuffNode<Elem> {     //<Elem>
private:
         FreqPair <Elem>  * it;           // freq pair
public:
        LeafNode(  Elem val, int freq)
        {  it = new FreqPair<Elem>(val, freq);  }    // Constructor
         int weight() {return it->weight();}
        bool isLeaf() { return true; }
        FreqPair<Elem>*  val() { return it; }
};
```

122

122

HuffNode class(2)

```
template <class Elem>   // Internal node of huffman tree
class IntlNode : public HuffNode<Elem> {
private:
        HuffNode<Elem>* lc;     // Left child
        HuffNode<Elem>* rc;    // Right child
        int  wgt;        // weight value
public:
         IntlNode( HuffNode<Elem>* l, HuffNode<Elem>* r)
        { wgt = l->weight()+ r->weight(); lc = l; rc = r; }
         int weight() {return wgt; }
         bool isLeaf() { return false;  }
        HuffNode<Elem>* left() { return lc; }
        void setLeft(HuffNode<Elem>* l)  {  lc = (HuffNode*)l;  }
        HuffNode<Elem>* right()  { return rc;   }
        void setRight(HuffNode<Elem>* r)   {  rc = (HuffNode*)r;   }
};
```

123

123

HuffTree class(1)

```
template <class Elem>
class HuffTree {
private:
   HuffNode<Elem>* myroot;
   void printhelp(HuffNode<Elem>* subroot, int level) const {  //相当于中序遍历
         FreqPair<Elem>* s1;
         if (subroot==NULL)  return;
        if (subroot->isLeaf()) {       // Do leaf node
          for(int i=0; i<level; i++)    cout << "*";
          cout << "Leaf: ";
          s1=((LeafNode<Elem> *)subroot)->val ();
          cout<<s1->val()<<"  "<<s1->weight()<< endl;   }
       else {
          printhelp(((IntlNode<Elem> *)subroot)->left(),level+1);  //打印左树
          for(int i=0; i<level; i++)    cout << "*";          //打印根
           cout << "Internal: ";
          cout<< ((IntlNode<Elem> *)subroot)->weight()<<endl;
          printhelp(((IntlNode<Elem> *)subroot)->right(),level+1); //打印左树
       }
    }
```

124

124

HuffTree class(2)

```
public:
    HuffTree(Elem val, int freq)  {
          myroot = new LeafNode<Elem>(val,freq);
    }
     HuffTree(HuffTree<Elem>* l, HuffTree<Elem>* r) {
          myroot = new IntlNode<Elem>(l->root(),r->root());
     }
     ~HuffTree( ) { ……   }    //可参考BST class 补写
      clear()  {  ……  }          //可参考BST class 补写

      HuffNode<Elem>*  root()  { return myroot;   }
      int weight() {  return myroot->weight();  }
      void print( ) const {     //相当于中序遍括历
           if (myroot == NULL)  cout << "The huffTree is empty.\n";
           else   printhelp(myroot, 0);
       }
};
```

125

125

本章作业二

- 5.26 (b)
- 5.32

126

126