数据结构与算法

DATA STRUCTURE

第十六讲 堆和Huffman树

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

课堂内容

- 串作业回顾
- 堆Heap

String私有成员

```
private:
    char* _pszData;
    int _capacity;
    int _size;

    // Clean everything and release memory
    void Reset();
    // Besides Reset(), it reserves a buffer.
    void GetBuffer(int capacity);
```

```
// [Private] Clean everything and release memory
void MyString::Reset()
{
    if (_pszData)
    {
        delete[] _pszData;
        _pszData = nullptr;
        _size = 0;
        _capacity = 0;
    }
}

// [Private] Clean everything and allocate a new buffer if necessary
void MyString::GetBuffer(int capacity)
{
    // Invalid paramter
    if (capacity <= 0)
    {
        return;
    }

    // No need to allocate a new buffer.
    // Clean everything
    if (capacity <= _capacity)
    {
        Resize(0);
        return;
    }

    // Release current buffer.
    Reset();

    // Allocate a new buffer and reset capacity value
    char* szBuffer = new (std::nothrow) char[capacity]{};
    if (szBuffer != nullptr)
    {
        _pszData = szBuffer;
        _capacity = capacity;
    }
}
```

String构造函数

```
MyString::MyString(const char * pszValue)
  :
  _pszData(nullptr),
  _capacity(0),
  _size(0)
{
    if (pszValue == nullptr)
    {
        return;
    }

    int leng = strlen(pszValue);
    if (leng <= 0)
    {
        return;
    }

    GetBuffer(2 * leng + 16);

    if (_capacity > 0)
    {
        _size = leng + 1;
        strncpy(this->_pszData, pszValue, leng);
    }
}
```

```
MyString::~MyString(void)
{
    Reset();
}

MyString::MyString(const MyString& other)
  :
  _pszData(nullptr),
  _capacity(0),
  _size(0)
{
    GetBuffer(other._capacity);

    if (_capacity > 0)
    {
        _size = other._size;
        strncpy(this->_pszData, other._pszData, other.Length())
    }
}
```

String运算符重载

```
MyString& MyString::operator+= (const MyString& rhs)
{
    Append(rhs);

    // return the existing object so we can chain this operator
    return *this;
}

int MyString::operator== (const MyString& str) const
{
    return strcmp(this->_pszData, str._pszData);
}

char & MyString::operator[] (int index)
{
    assert (index >= 0 && index < Length());
    return _pszData[index];
}

const char & MyString::operator[] (int index) const
{
    assert (index >= 0 && index < Length());
    return _pszData[index];
}
```

```
MyString& MyString::operator= (const MyString& rhs)
{
    if(this == &rhs)
    {
       return *this;
    }

    if (_capacity <= rhs._size)
    {
        GetBuffer(rhs._capacity);
    }

    if (_capacity > 0)
    {
        _size = rhs._size;
        strncpy(this->_pszData, rhs._pszData, rhs.Length());
    }

    // return the existing object so we can chain this operator
    return *this;
}

MyString MyString::operator+(const MyString& rhs)
{
    MyString newString(nullptr);

    if (Length() > 0 || rhs.Length() > 0)
    {
        newString.GetBuffer(_capacity + rhs.Length());
        if (newString._capacity > 0)
        {
            strncpy(newString._pszData, _pszData, Length());
            strcat(newString._pszData, rhs._pszData);

            newString._size = strlen(newString._pszData) + 1;
            newString._capacity = _capacity + rhs.Length();
        }
    }

    return newString;
}
```

String公开内存操作

```
// Append a new string at the end.
MyString & MyString::Append(const MyString & str)
{
    if (str.Length() <= 0)
    {
        return *this;
    }

    // Allocate new buffer, only when string buffer is not large enough
    if (_size + str.Length() > _capacity)
    {
        Reserve(_capacity + str.Length());
    }

    strncat(_pszData, str._pszData, str.Length());
    _size = strlen(_pszData) + 1;
    return *this;
}
```

```
// Allocate a large buffer, while keeping old data.
void MyString::Reserve(int capacity)
{
    if (capacity <= _capacity)
    {
        return;
    }

    // Allocate a new buffer. Replace old string buffer with new one.
    char* szBuffer = new (std::nothrow) char[capacity]{};
    if (szBuffer != nullptr)
    {
        int size = 0;
        if (Length() > 0)
        {
            strncpy(szBuffer, _pszData, Length());
            size = _size;
        }

        Reset();
        _pszData = szBuffer;
        _capacity = capacity;
        _size = size;
    }
}

// Resize to smaller string.
void MyString::Resize(int size)
{
    if (size < 0 || size > _size)
    {
        return;
    }

    _size = size;
    *(_pszData + _size) = '\0';
}
```

String动态操作

```
// Insert a substring after index
void MyString::Insert(int index, const char * pszStr)
{
    if (pszStr == nullptr)
    {
        return;
    }

    int leng = strlen(pszStr);

    // Validate input parameters
    if (index >= Length() || index < 0 || leng <= 0)
    {
        return;
    }

    // If buffer is not enough, allocate a new one while keeping old data.
    if (leng + _size > _capacity)
    {
        Reserve(_capacity + leng);
    }

    // Now save the substring after index, concatenate new string after index,
    // and concatenate the saved subtring (temp) after new string.
    MyString temp(_pszData + index);
    Resize(index);
    strncat(_pszData, pszStr, leng);
    strncat(_pszData, temp._pszData, temp.Length());
    _size = strlen(_pszData) + 1;
}
```

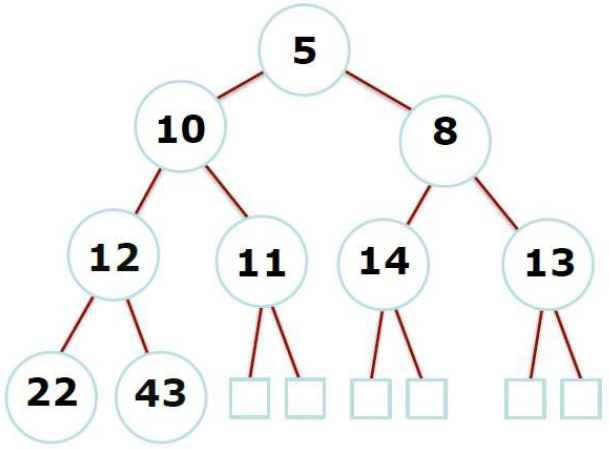
```
// Delete substring of given length, starting at index
void MyString::Remove(int index, int length)
{
    // Validate input parameters
    if (index >= Length() || index < 0 || length <= 0)
    {
        return;
    }

    // This is easier case by string resize
    if (index + length >= Length())
    {
        Resize(index);
        return;
    }

    // Otherwise, copy the rest of string after index
    char * szDst = _pszData + index;
    char * szSrc = _pszData + index + length;
    while (szSrc < _pszData + _size)
    {
        *szDst++ = *szSrc++;
    }
    *szDst = '\0';
    _size = strlen(_pszData) + 1;
}
```

堆

heap

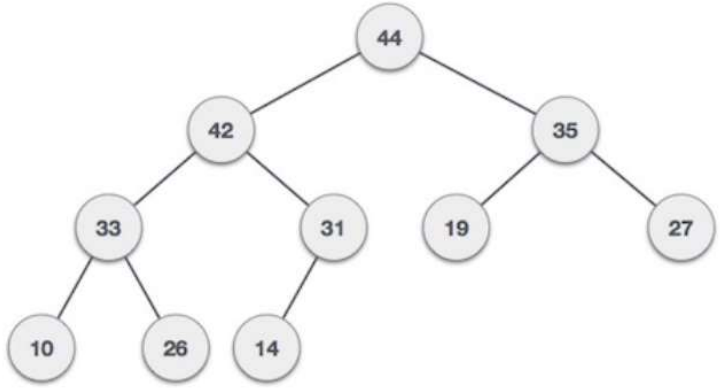
定义

- 是个完全树，即除了最后一层，前面都没有空节点
- 并且满足性质，如果父节点A有子节点B，那么$Key(A) \leq Key(B)$
- 注意兄弟节点之间没有直接关系
- 比较适合按数组存

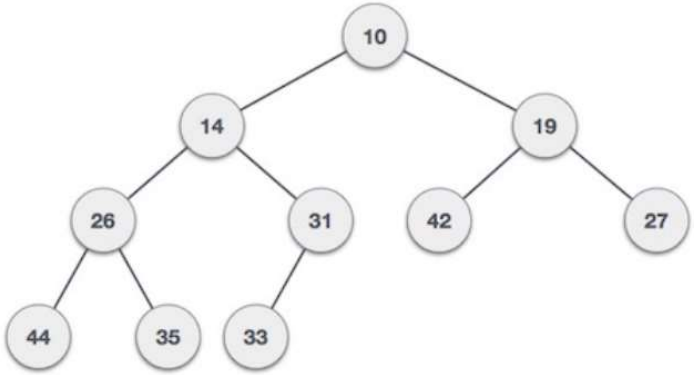


堆种类

- 最大堆$Key(\text{parent}) \geq Key(\text{child})$

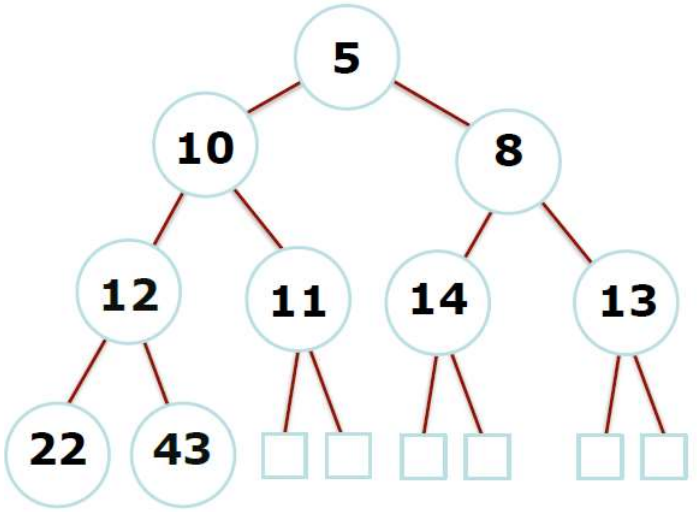


- 最小堆$Key(\text{parent}) \leq Key(\text{child})$



如何用数组表达堆

- 根节点是第一个数组元素
- 对于第i个数组元素，
  - 其父节点是第$i/2$个数组元素
  - 其左子节点是$2i$个数组元素
  - 其右子节点是$2i+1$个数组元素
- 数组里的元素是按层排列的



	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

最小堆插入算法

- 在堆的最后加入一个新节点，设置值
- 比较其父节点的值，如果比父节点大，就交换
- 循环直到满足最小堆的性质，或者到根节点
- 时间复杂度$log(n)$

```
// Inserts a new key 'k'
void insertKey(int k)
{
    // First insert the new key at the end
    data.push_back(k);
    int i = data.size() - 1;

    // Fix the min heap property if it is violated
    while (i > 1 && data[parent(i)] > data[i])
    {
       swap(data[i], data[parent(i)]);
       i = parent(i);
    }
}
```

最小堆维护算法

- 用递归调整根节点i所在的子树
- 这个函数作为子程序被删除算法调用
- 时间复杂度$log(n)$

```
void MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int min = i;
    if (l < data.size() && data[l] < data[min])
    {
        min = l;
    }
    if (r < data.size() && data[r] < data[min])
    {
        min = r;
    }

    if (min != i)
    {
        swap(data[i], data[min]);
        MinHeapify(min);
    }
}
```

最小堆移去最小值

- 最小值就是根节点
- 把最后一个元素替代第一个元素
- 然后调用维护算法，把新根结点修正
- 时间复杂度$log(n)$

```
// to extract the root which is the minimum element
int extractMin()
{
    // data[0] is a place holder.
    if (data.size() <= 1)
    {
        return INT_MAX;
    }

    // Store the minimum value, and remove it from heap
    int root = data[1];
    data[1] = data.back();
    data.pop_back();

    if (data.size() > 2)
    {
        MinHeapify(1);
    }

    return root;
}
```

最小堆删除算法

- 先把要删除的元素设为最小值
- 然后根据最小堆性质，把它交换到根节点
- 最后调用移去最小值算法
- 时间复杂度$log(n)$

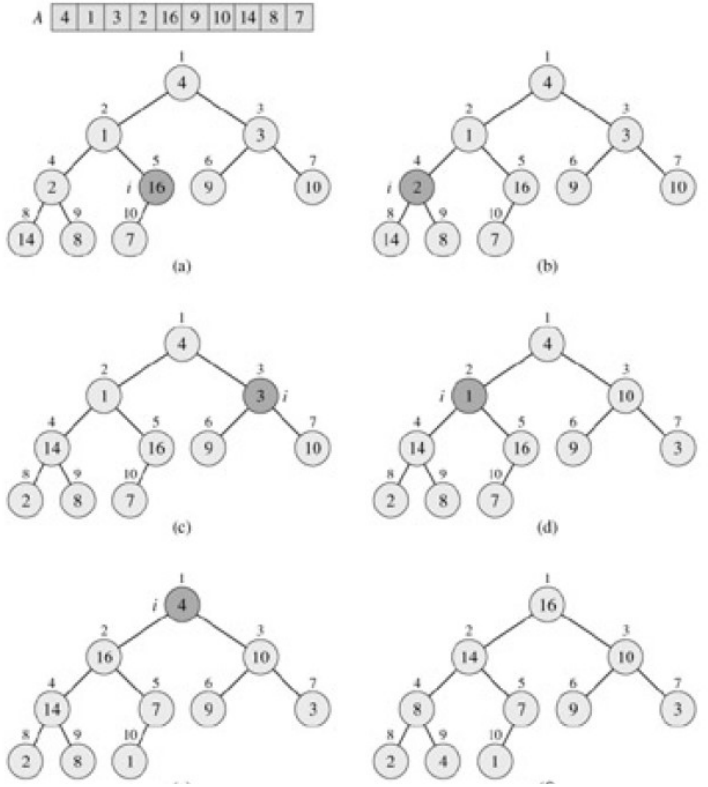
```
// Decreases key value of key at index i to new_val
void decreaseKey(int i, int new_val)
{
    while (i > 1 && data[parent(i)] > new_val)
    {
        data[i] = data[parent(i)];
        i = parent(i);
    }
    data[i] = new_val;
}

// Deletes a key stored at index i
void deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}
```

堆的应用

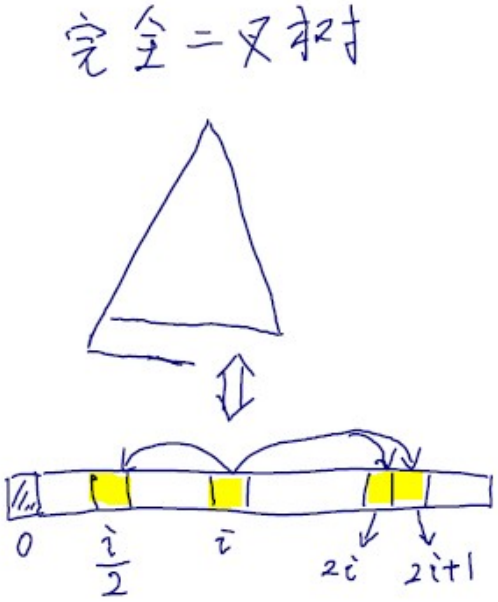
1. Heapsort
   - 给定一组数，如何建立堆

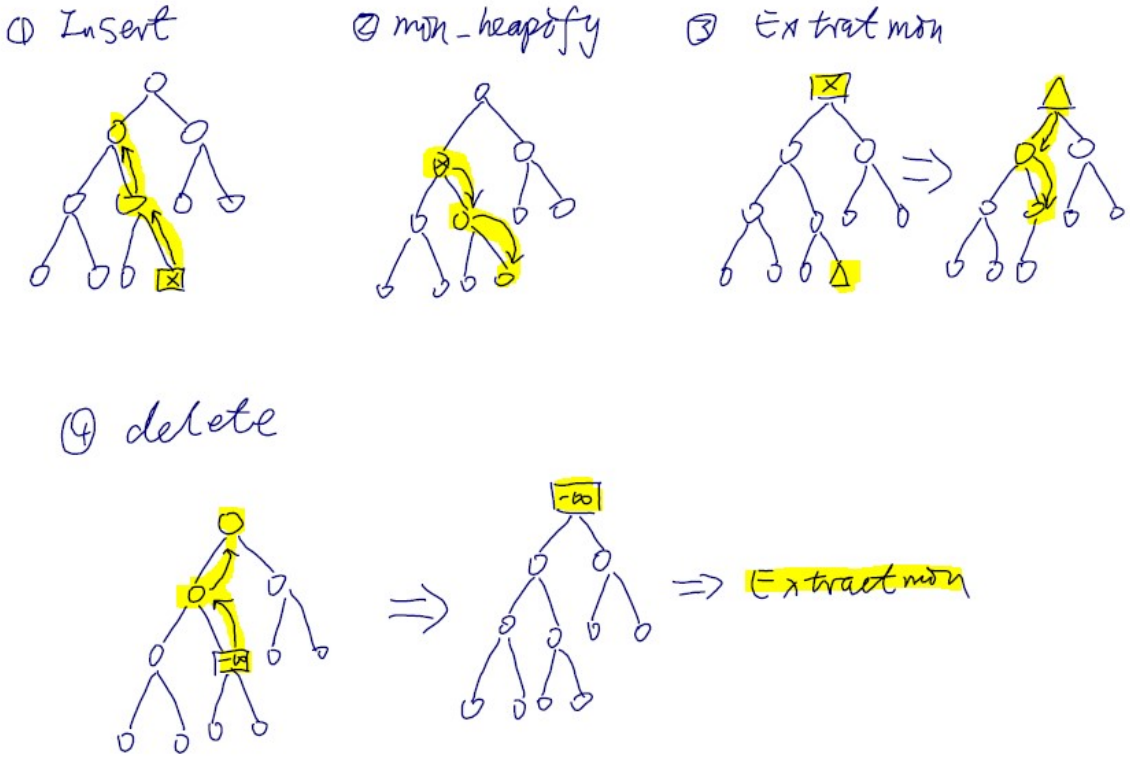
```
void BuildHeap(int arr[], int length)
{
    data.clear();
    data.push_back(0);

    for (int i = 0; i < length; i++)
    {
        data.push_back(arr[i]);
    }

    for (int i = length / 2; i > 0; i--)
    {
        MinHeapify(i);
    }
}
```

2. 找出一组数中前k个最小数
3. 合并K个有序数组
4. 用来实现priority queue

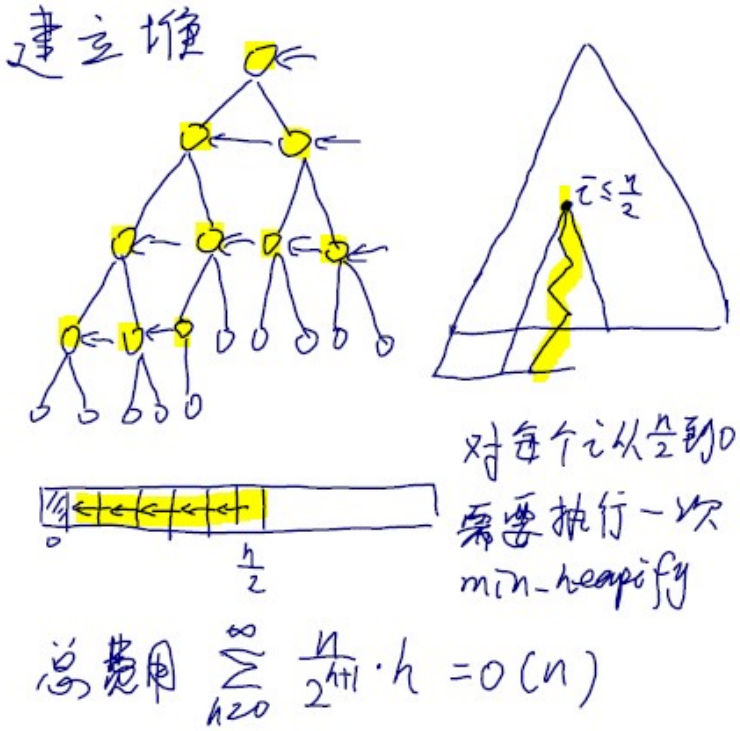


最小堆总结


完全二叉树
0
i/2
i
2i
2i+1


① Insert
② min_heapify
③ Extract min
④ delete
-∞
-∞
Extract min

最小堆总结


建立堆
$i \le \frac{n}{2}$
对每个i从$\frac{n}{2}$到0
需要执行一次
min_heapify
$\frac{n}{2}$
总费用 $\sum_{h\ge 0}^{\infty} \frac{n}{2^{h+1}} \cdot h = O(n)$

霍夫曼树

Huffman Tree

问题

- 对文本文件怎么压缩保存，比如基因序列
- 特点：不同字符个数比较少（ACGT）
- 正常的字符一个字节，比较浪费空间
- 一种方法是用固定长度的3bit/4bit来代替字符
- 用的时候解码
- 再进一步？

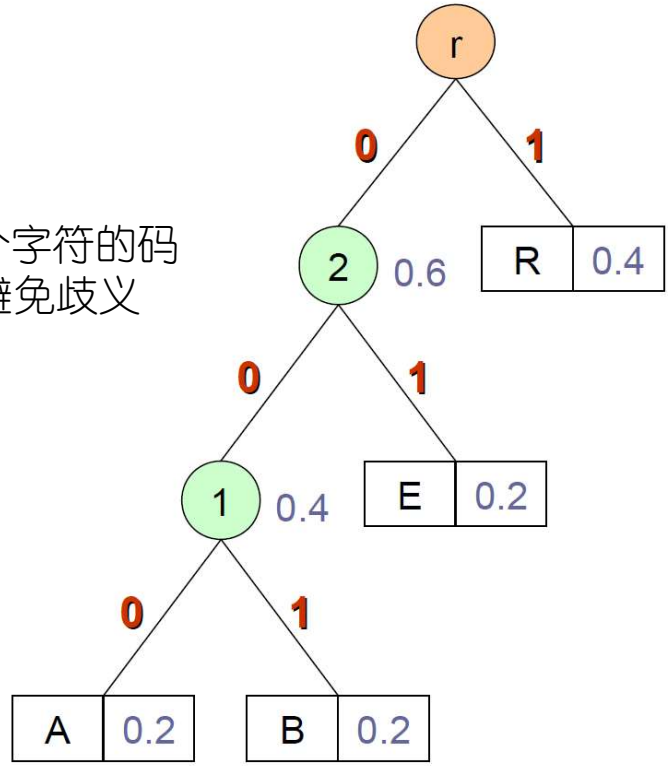
Huffman coding

- 对文本一种不损失的压缩方法
- 用的是满二叉树
- 用不固定bit长度压缩
- 希望用的频繁的字符码短点，用的少的字符码长点
- 比如 文本中只出现四种字符ABER

Symbol	A	B	E	R
Frequency	1	1	1	2
Probability	20%	20%	20%	40%

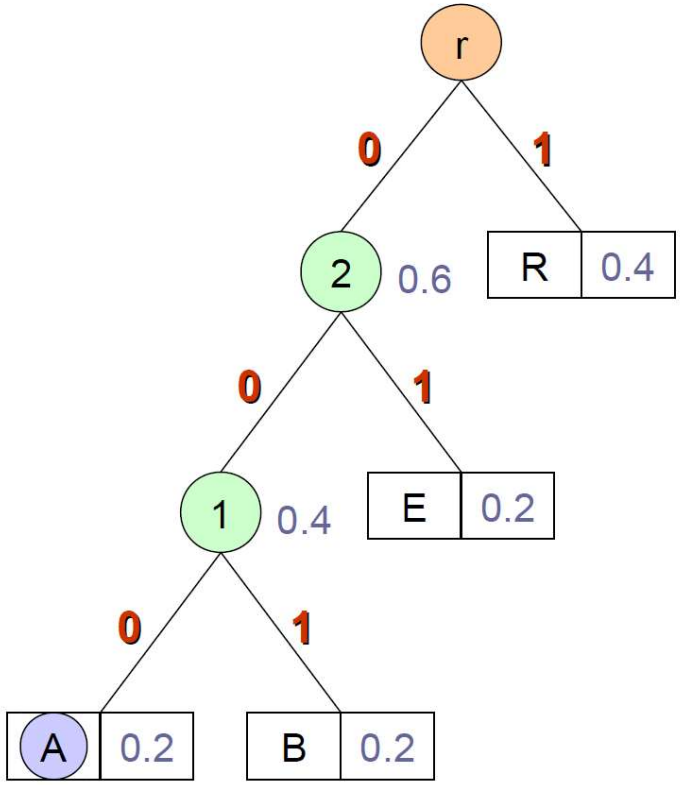
Huffman codeding例子

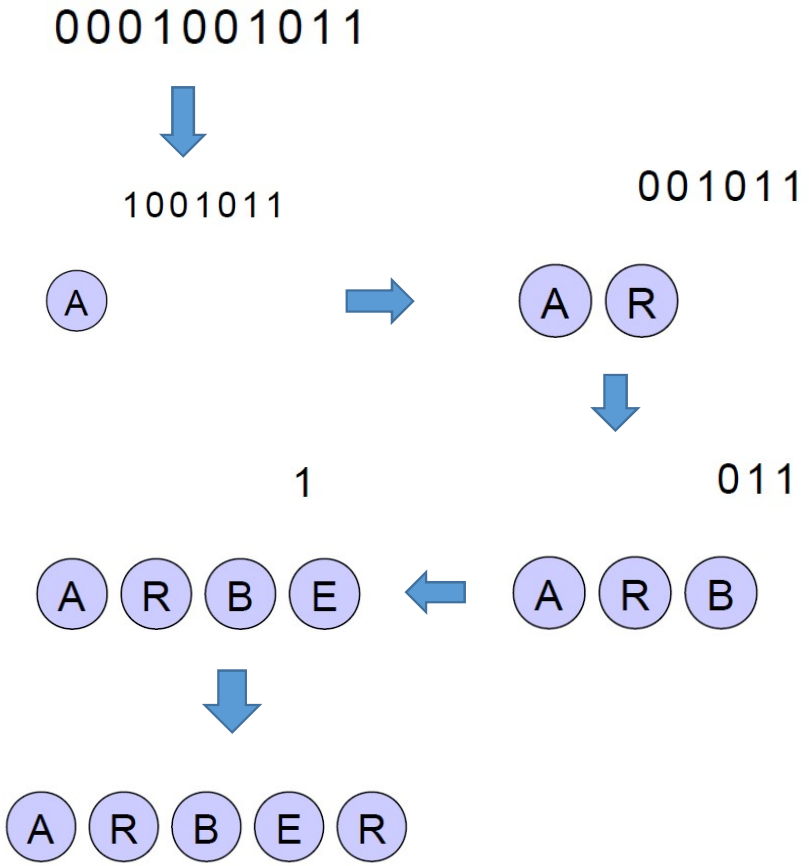
满二叉树
Prefix codes：意思是一个字符的码
不能是另一个的前缀，避免歧义



A	0 0 0
B	0 0 1
E	0 1
R	1

Huffman decodeding例子





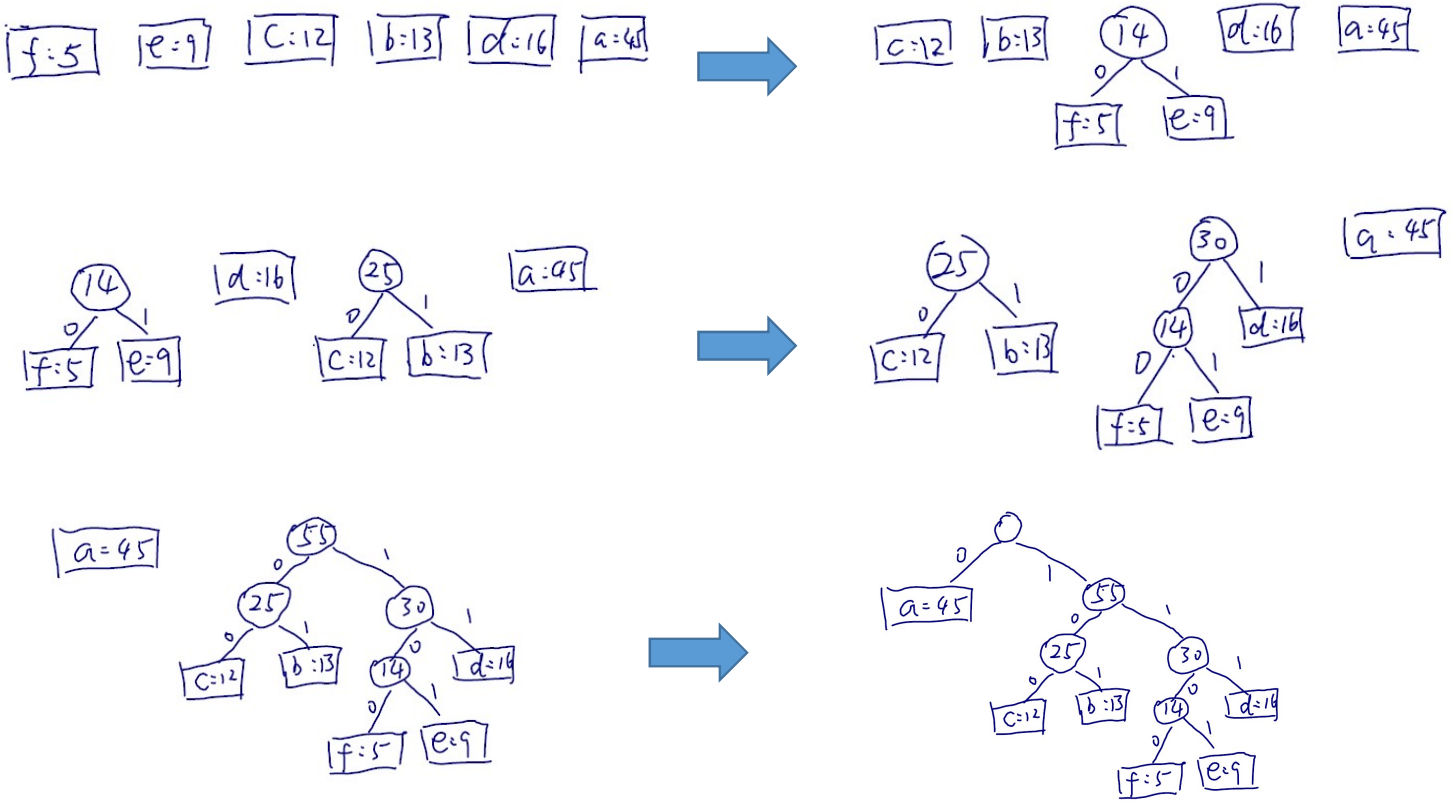
Huffman tree算法—贪婪法

- 先对每个要编码字符建立叶节点
- 按这些节点节点建立最小堆，用的最少的是根节点
  1. 从堆里取出最小的两个A， B
  2. 建立一个新的内部节点C，使得A，B是C的两个子节点
  3. C的优先级是A和B的优先级的和，然后把C放入最小堆

循环1，2，3，直到最小堆只有一个元素，它也是huffman树的根节点

可以证明贪婪法就是最优解

实例



Q&A

Thanks!