数据结构与算法

DATA STRUCTURE

第十四讲 堆和Huffman树

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

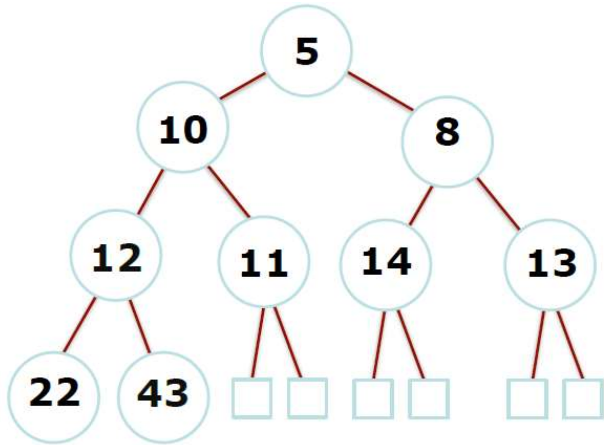
- 堆Heap
- 霍夫曼树Huffman Tree

堆

heap

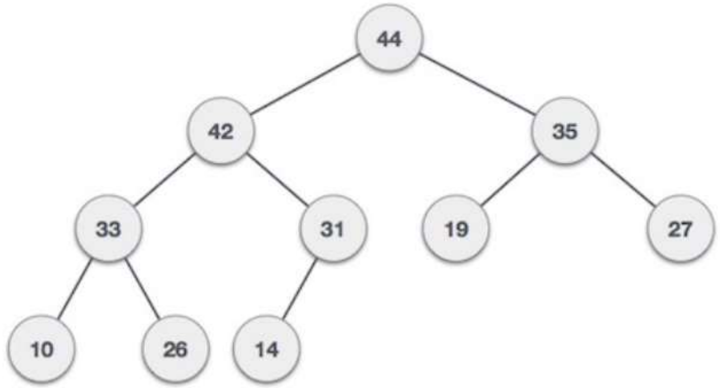
定义

- 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

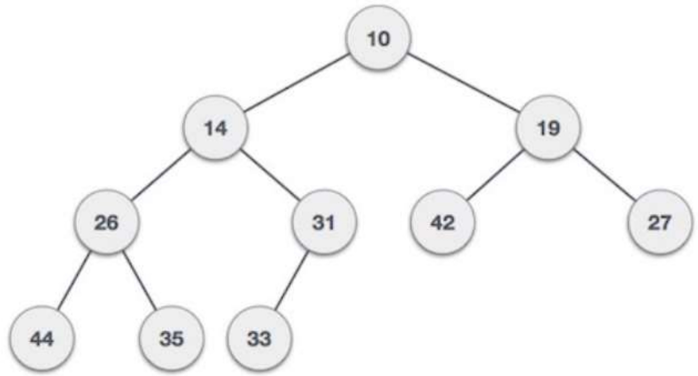


堆种类

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

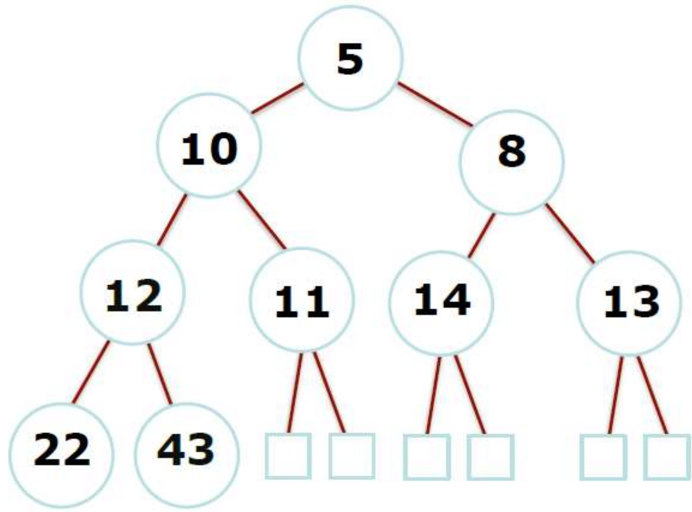


- 最小\堆$Key(\text{parent}) \leq Key(\text{child})$



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
  - 其左子节点是第$2i$个数组元素
  - 其右子节点是第$2i+1$个数组元素
- 数组里的元素是按层排列的

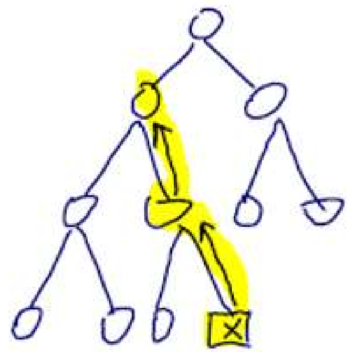


	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

最小堆插入算法

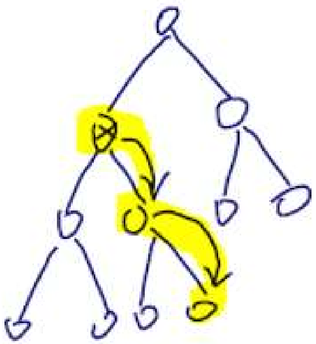
- 在堆的最后加入一个新节点，设置值
- 比较其父节点的值，如果比父节点大，就交换
- 循环直到满足最小堆的性质，或者到根节点
- 时间复杂度$log(n)$

```
// 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$log(n)$

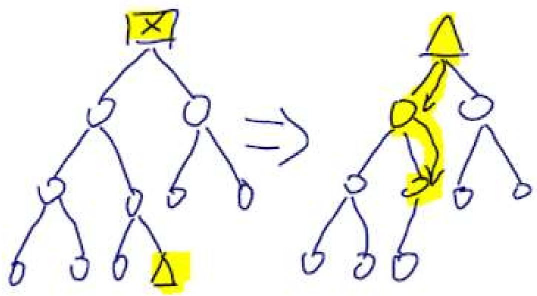


```
void MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int min = i;
    if (l < data.size() && data[l] < data[min])
    {
        min = l;
    }
    if (r < data.size() && data[r] < data[min])
    {
        min = r;
    }

    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$log(n)$

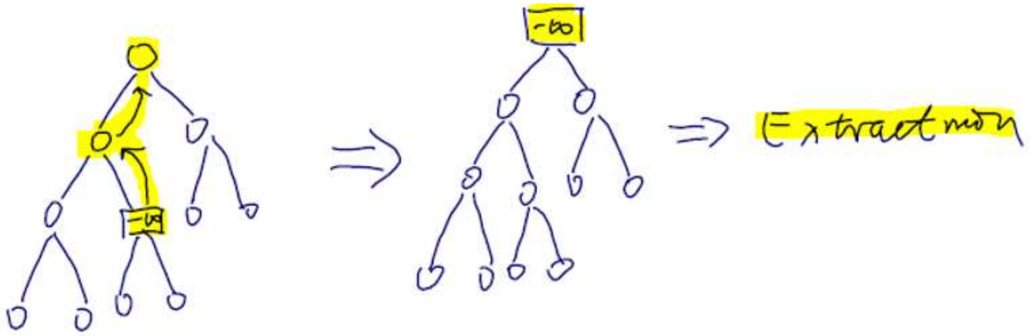
```
// to extract the root which is the minimum element
int extractMin()
{
    // data[0] is a place holder.
    if (data.size() <= 1)
    {
        return INT_MAX;
    }

    // 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$log(n)$

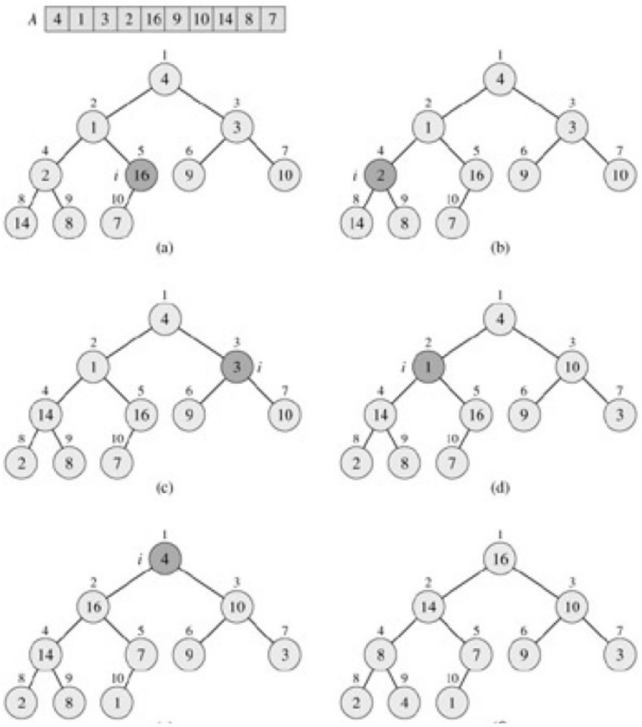
```
// Decreases key value of key at index i to new_val
void decreaseKey(int i, int new_val)
{
    while (i > 1 && data[parent(i)] > new_val)
    {
        data[i] = data[parent(i)];
        i = parent(i);
    }
    data[i] = new_val;
}

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



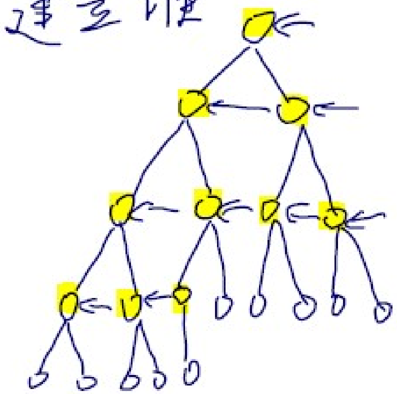
建立堆

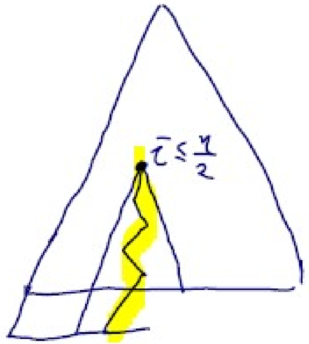
- 给定一组数，如何建立最大堆

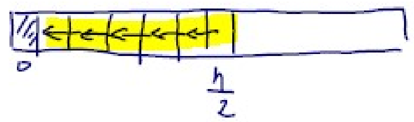
```
void BuildHeap(int arr[], int length)
{
    data.clear();
    data.push_back(0);

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

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



建立堆







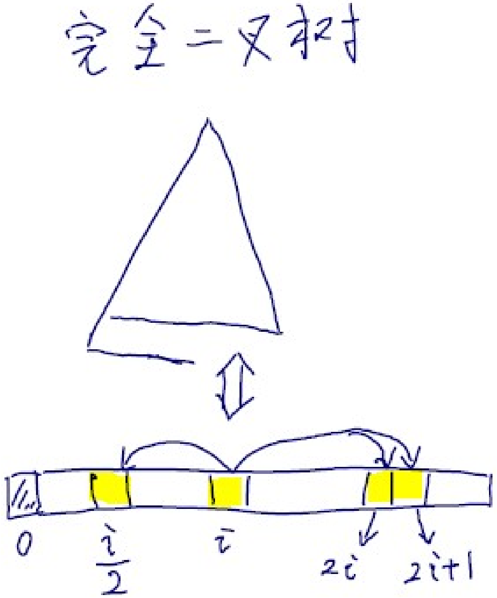
对每个i从$\frac{n}{2}$到0
需要执行一次
min_heapify

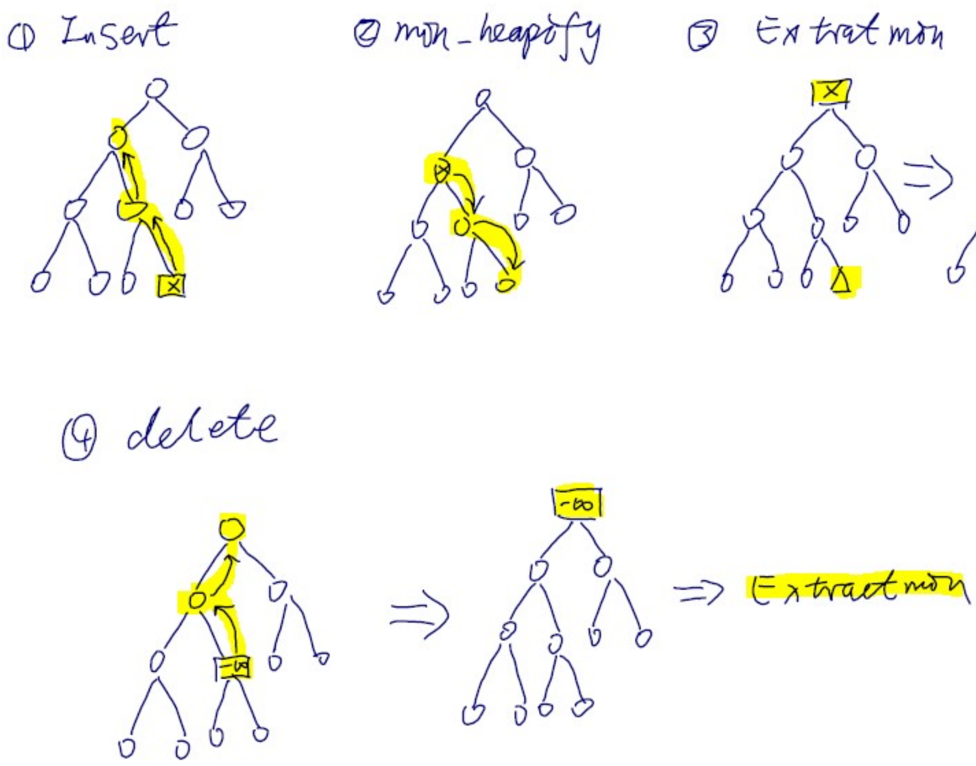
总费用 $\sum_{h=0}^{\infty} \frac{n}{2^{h+1}} \cdot h = O(n)$

堆的应用

1. Heapsort
2. 找出一组数中第k-th大的数
   - 排序，从大到小数到第k个
   - 用最大堆，连续取k次
   - 前k个数的最小堆，逐个添加后面n-k个数，如果比堆顶元素要大的话。每加一个，然后移去最小一个。
3. 合并K个有序数组
4. Online steam里找最大的k个数
5. 用来实现priority queue
   - O(1) 取到优先级最高元素
   - O(logn) 插入新元素
   - O(logn) 移去优先级最高元素
   - O(logn) 改变元素的优先级

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

霍夫曼树

Huffman Tree

问题

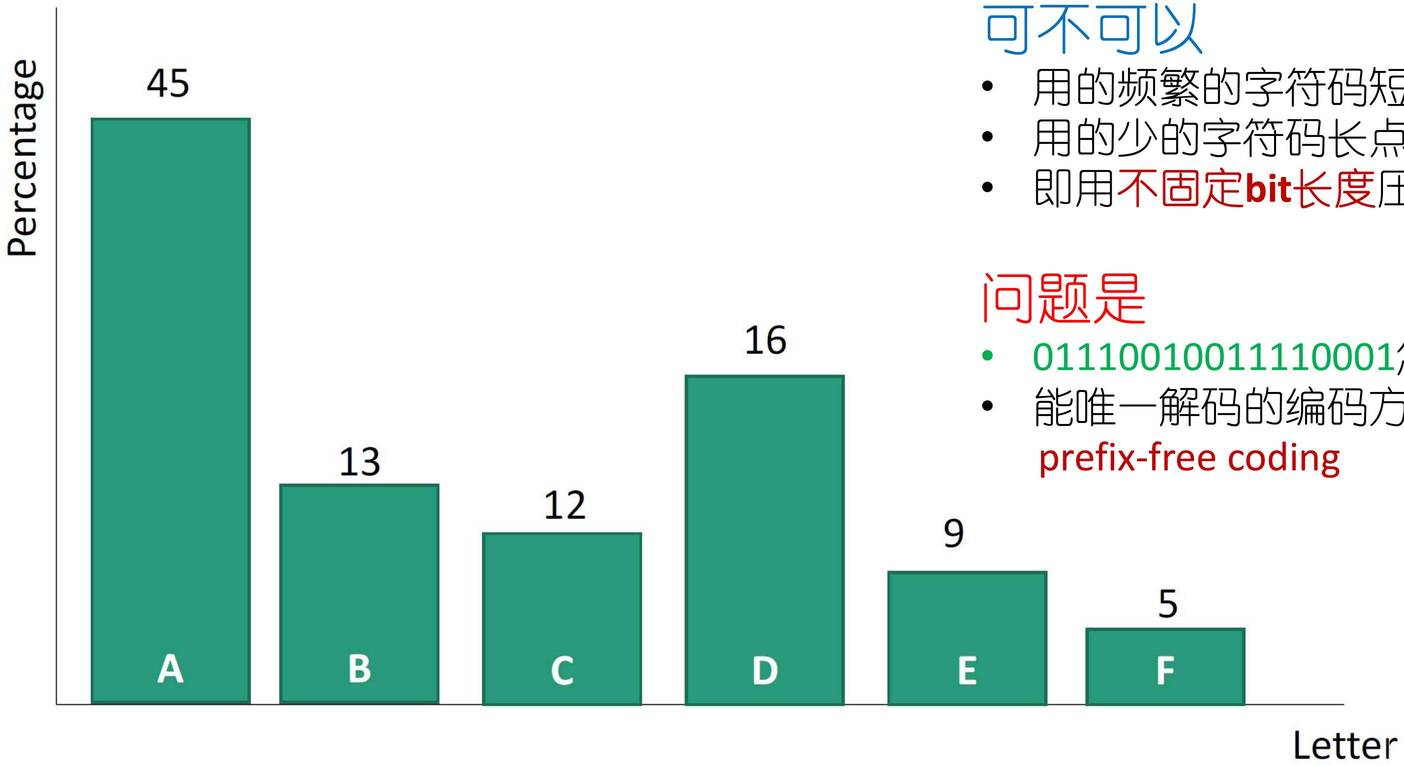
- 对文本文件怎么压缩保存，比如基因序列
- 特点：不同字符个数比较少（ACGT）

- ASCII字符一个字节，比较浪费空间

01100101 01110110 01100101 01110010 01111001 01100101 01100001 01111001 00100000 01100101 01101110 01100101 01101100 01100101 01110011 01101000 01100101 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101

- 解码比较简单
- 一种改进方法：用固定长度的3bit/4bit来代替字符，用的时候解码

假如我们知道用到的字符的分布



可不可以

- 用的频繁的字符码短点
- 用的少的字符码长点
- 即用不固定**bit**长度压缩

问题是

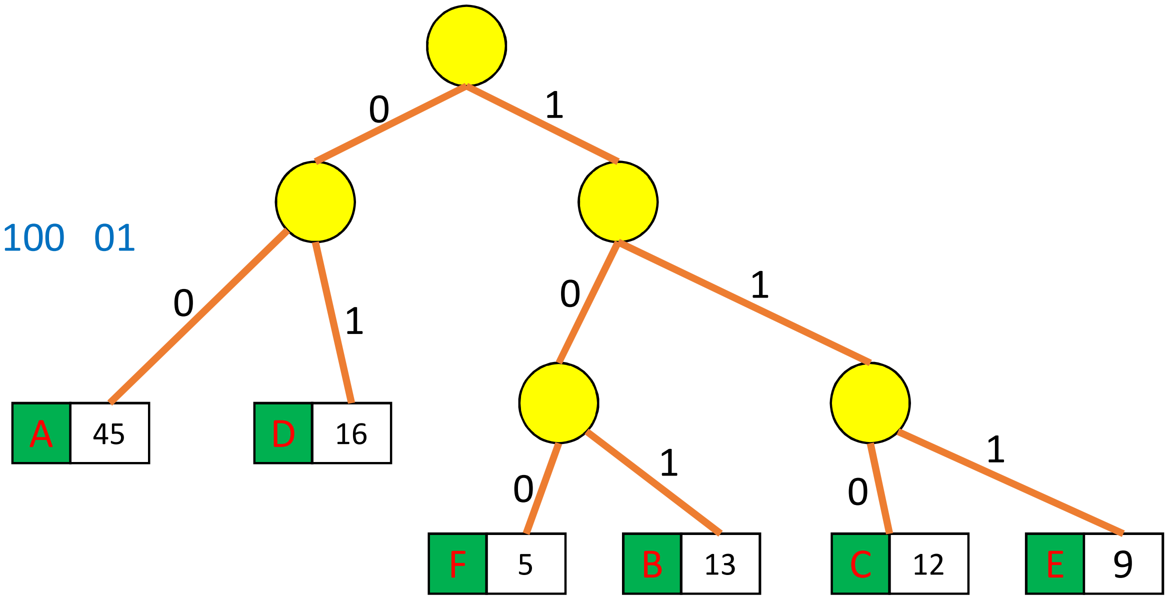
- 01110010011110001怎么解码？
- 能唯一解码的编码方式叫 prefix-free coding

Prefix-free coding

- 一个字符代表的码不是另一个字符代表的码的前缀
- 那么在所有字符码生成的Trie
- 只要字符是在Trie的叶节点，就是prefix-free coding
- 可以唯一解码

01110010011110001

01 110 01 00 111 100 01



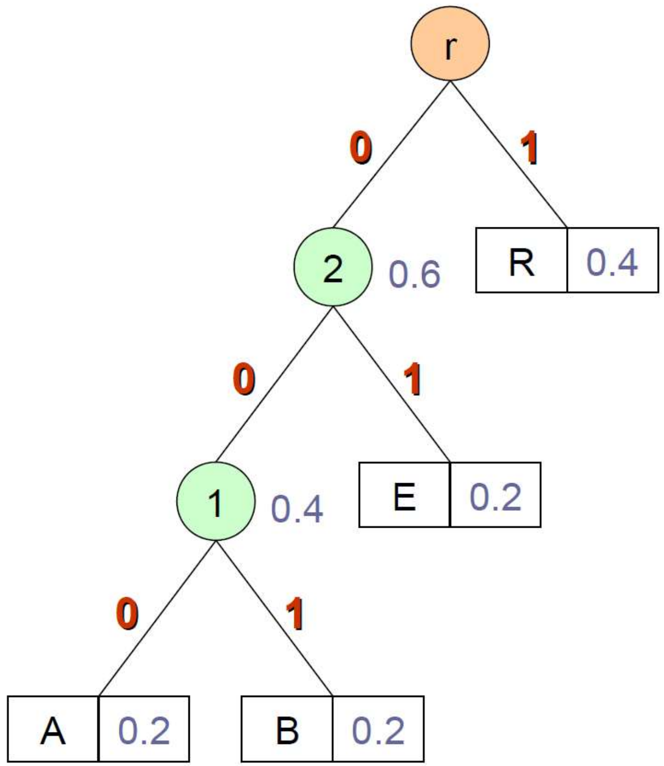
Prefix-free coding例子

比如，文本中只出现四种字符A B E R

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

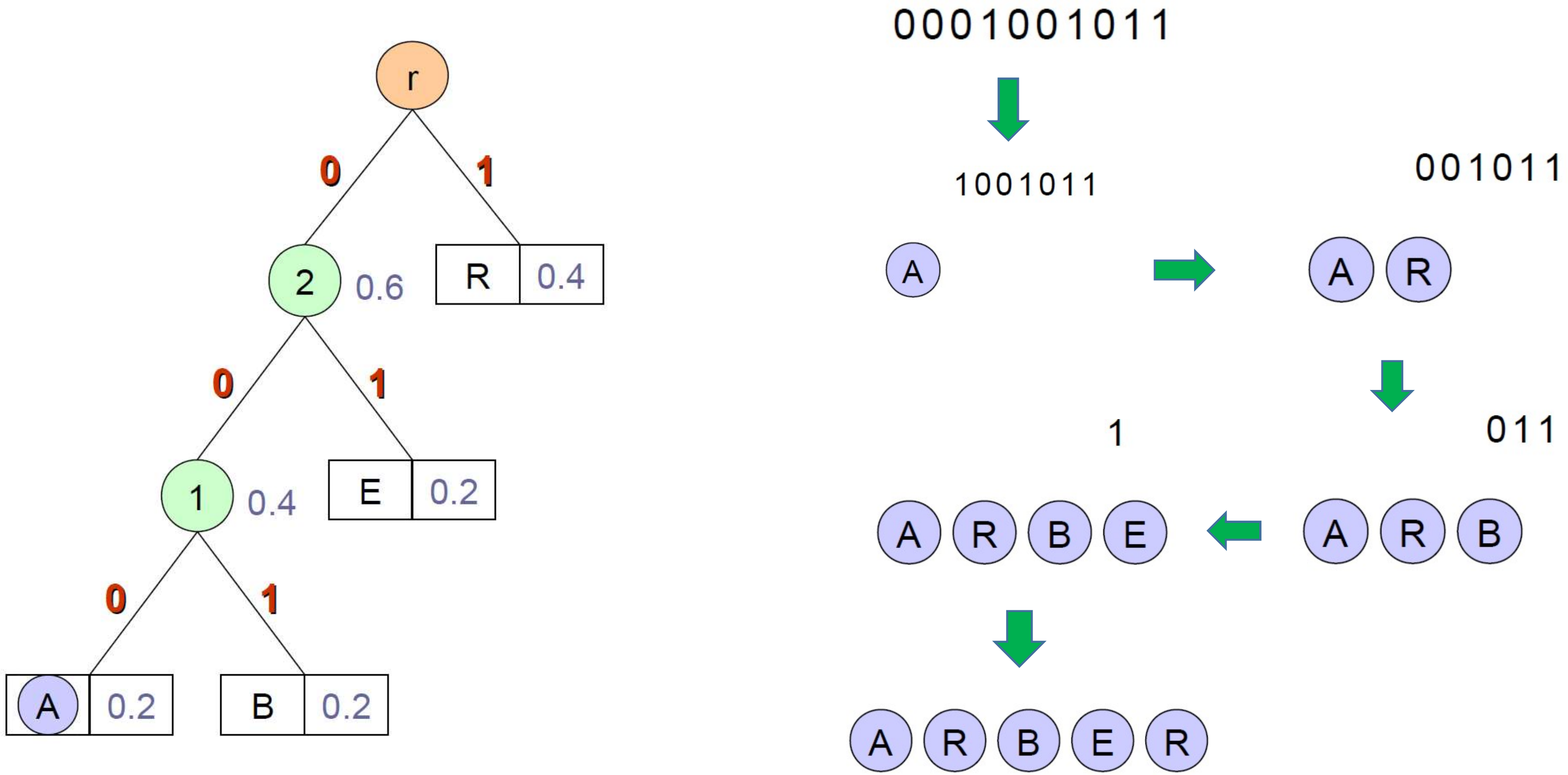
字符码对应表

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



满full二叉树

Huffman decoding例子



问题变成

- 找一种prefix-free coding，使得在给定字符使用率下，压缩率最高
- 对应的Trie，就是Huffman tree
- 可以证明Huffman tree一定是满的二叉树
- 也可以证明以下贪婪法生成的Huffman tree就是最优编码

Huffman tree算法 -- 贪婪法

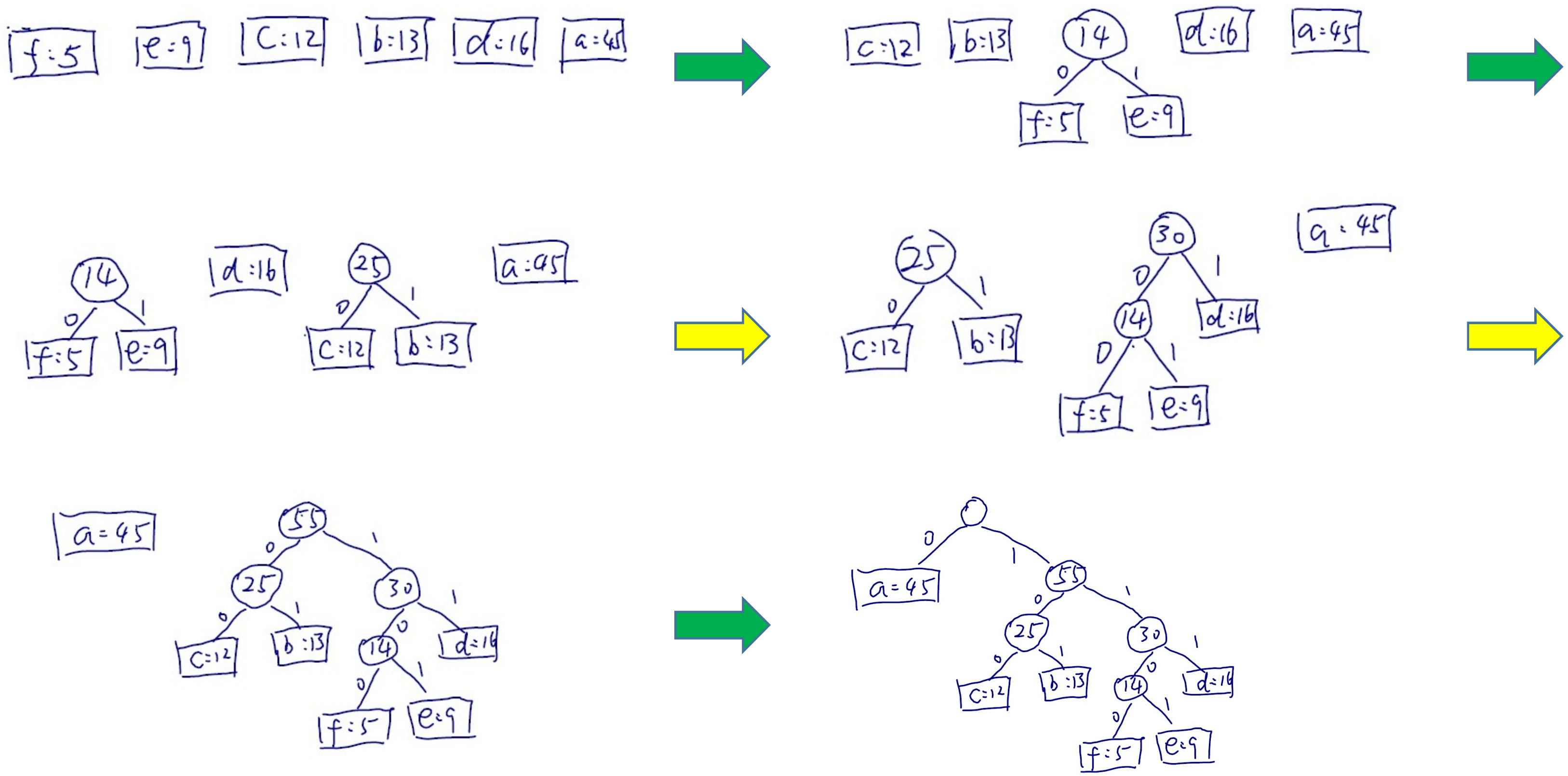
- 先对每个要编码字符建立叶节点
- 按这些节点节点建立最小堆，用的最少的是根节点
  1. 从堆里取出最小的两个A， B
  2. 建立一个新的内部节点C，使得A，B是C的两个子节点
  3. C的优先级是A和B的优先级的和，然后把C放入最小堆

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

实例

注意，每步只取出频率最小的两个节点组合。

所以使用最小堆$O(\log n)$，而不需要对所有节点遍历$O(n)$，甚至排序$O(n\log n)$。



作业

- 用最小堆实现Huffman树
- 测试：对二进制文本解码

- 要求：输入一组字符以及相应的使用频率，
  生成BinaryTree表示的Huffman tree，并能对二进制编码解码

提示：每个字符及其频率可以创建各自BinaryTree*，然后用最小堆，根据贪婪法
取出频率最小的树，合并成新的BinaryTree*后插入最小堆，继续这
样过程，直到只剩下一个BinaryTree*

注意： 最小堆需要是单独的类，实现的内部数据应该是动态数组，
这个数组每个元素可以是BinaryTree的指针（考虑用std::vector）
这样每次插入BinaryTree指针到最小堆，然后移去的两个树指针需要释放内存

BinaryTree类

```
class BinaryTree
{
public:
    // build a random binary tree, given size
    BinaryTree(int size);
    // Build a binary tree based on preorder and inorder array
    BinaryTree(const std::vector<int> &preorder, const std::vector<int> &inorder);

    // Release tree memory recursively
    ~BinaryTree();
    // Copy a tree structure, not pointer.
    BinaryTree(const BinaryTree & other);
    // Implement assigment overloading, the same as copy constructor
    BinaryTree & operator =(const BinaryTree & rhs);

    // implement iterative version of traversals.
    void LevelOrder_Backwards();
    void LevelOrder();
    void PreOrder_Iterative();
    void InOrder_Iterative();
    void PostOrder_Iterative();

    void PreOrder(const Node * root);
    void InOrder(const Node * root);
    void PostOrder(const Node * root);

private:
    void Release(Node *root);
    Node * CopyTree(const Node *root);
    Node * BuildTree_Iterative(const std::vector<int> &preorder, const std::vector<int> &inorder)
    Node * BuildTree(const std::vector<int> &preorder, int pre,
                     const std::vector<int> &inorder, int lh, int rh);

    Node * _pRoot;
};
```

```
// Specific method to build Huffman tree
// Create a new root with value of key,
// having itself as left child and input rChild as right chil
BinaryTree(char ch, int key) { _pRoot = new Node(key, ch); }
void Merge(int key, const BinaryTree & rChild);
int GetRootKey() { return _pRoot ? _pRoot->key : INT_MIN; }
```

成员函数用来帮助建立Huffman tree

```
struct Node
{
    int key;
    char alpha;
    Node * left;
    Node * right;

    Node(int val, char ch = '')
    {
        key = val;
        alpha = ch;
        left = right = nullptr;
    }
};
```

构造/析构函数

```
BinaryTree::BinaryTree(const BinaryTree &other)
    :
    _pRoot(nullptr)
{
    _pRoot = CopyTree(other._pRoot);
}
```

```
BinaryTree::~BinaryTree()
{
    DeleteTree();
}
```

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

    DeleteTree();
    _pRoot = CopyTree(rhs._pRoot);

    return *this;
}
```

```
Node * BinaryTree::CopyTree(const Node *root)
{
    if (root == nullptr)
    {
        return nullptr;
    }
    Node * copyRoot = new Node(root->key);
    copyRoot->left = CopyTree(root->left);
    copyRoot->right = CopyTree(root->right);

    return copyRoot;
}
```

```
void BinaryTree::DeleteTree()
{
    Release(_pRoot);
    _pRoot = nullptr;
}
```

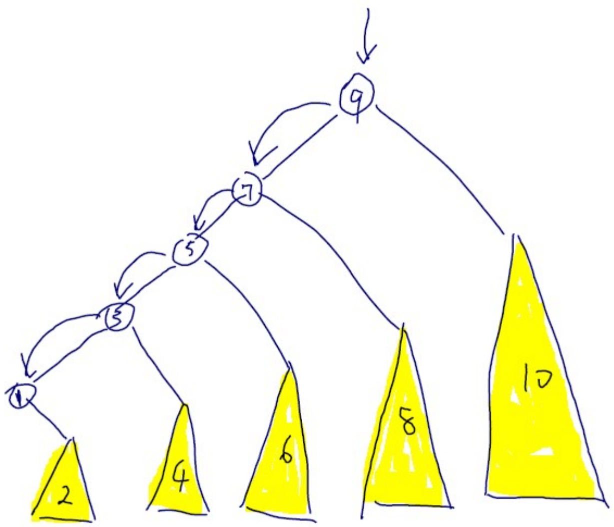
```
void BinaryTree::Release(Node * root)
{
    if (root == nullptr)
    {
        return;
    }
    Release(root->left);
    Release(root->right);
    root->left = nullptr;
    root->right = nullptr;
    delete root;
}
```

```
void BinaryTree::Merge(int key, const BinaryTree & rChild)
{
    Node * newRoot = new Node(key);
    newRoot->left = _pRoot;
    newRoot->right = CopyTree(rChild._pRoot);
    _pRoot = newRoot;
}
```

从前序/中序恢复

```
BinaryTree::BinaryTree(const vector<int> &preorder, const vector<int> &inorder)
{
    if (preorder.size() != inorder.size())
    {
        return;
    }

    _pRoot = BuildTree(preorder, 0, inorder, 0, inorder.size()-1);
}

Node* BinaryTree::BuildTree(const vector<int> &preorder, int pre,
                            const vector<int> &inorder, int lh, int rh)
{
    if (lh > rh)
    {
        return nullptr;
    }
    int val = preorder.at(pre);
    Node *root = new Node(val);

    int index = lh;
    for (; index <= rh; index++)
    {
        if (inorder.at(index) == val)
        {
            break;
        }
    }
    assert(index <= rh);

    root->left = BuildTree(preorder, pre+1, inorder, lh, index - 1);
    root->right = BuildTree(preorder, pre+index-lh+1, inorder, index + 1, rh);
    return root;
}
```

迭代实现




```
Node * BinaryTree::BuildTree_Iterative(const vector<int> &preorder, const vector<int> &inorder)
{
    assert(preorder.size() == inorder.size());

    // pre index always points to node will be created next
    // in index always points to leftmost child of right child of stack top when push, or stack top itself
    int pre = 0, ind = 0;
    Node *root = new Node(preorder.at(pre++));

    stack<Node *> stk;
    stk.push(root);

    while (true)
    {
        if (inorder[ind] == stk.top()->key)
        {
            Node * top = stk.top();
            stk.pop();
            ind++;

            // all done
            if (ind == inorder.size())
            {
                break;
            }
            // the case that right child is empty
            if (!stk.empty() && inorder[ind] == stk.top()->key)
            {
                continue;
            }
            // set right child and push to stk.
            // Then process right subtree like a new one
            top->right = new Node(preorder.at(pre++));
            stk.push(top->right);
        }
        else
        {
            // Keep pushing left child to stk until leftmost child.
            Node * nd = new Node(preorder.at(pre++));
            stk.top()->left = nd;
            stk.push(nd);
        }
    }

    return root;
}
```

Q&A

Thanks!