# 软件设计与开发实践I
# 结课答辩

杨志飞
1336101班·1130310217
计算机科学与技术学院  计算机科学试验班

# 我要讲什么

# 我要讲什么

- 总结：数据结构和算法基础部分

# 我要讲什么

- 总结：数据结构和算法基础部分
  - ✳ 线性、树型、图结构，查找、散列、排序各章的亮点

# 我要讲什么

- 总结：数据结构和算法基础部分
  - ✳ 线性、树型、图结构，查找、散列、排序各章的亮点
  - ✳ 一个惊喜——点睛之笔

# 我要讲什么

- 总结：数据结构和算法基础部分
  - ✳ 线性、树型、图结构，查找、散列、排序各章的亮点
  - ✳ 一个惊喜——点睛之笔
- 展示：综合应用提高部分

# 我要讲什么

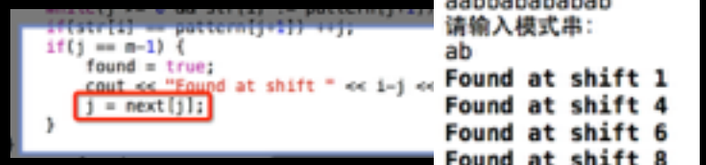
- 总结：数据结构和算法基础部分
  - ✳ 线性、树型、图结构，查找、散列、排序各章的亮点
  - ✳ 一个惊喜——点睛之笔
- 展示：综合应用提高部分
  - ✳ 从二叉树到栈——超复杂多冗余表达式求值

# 我要讲什么

- 总结：数据结构和算法基础部分
  - ✳ 线性、树型、图结构，查找、散列、排序各章的亮点
  - ✳ 一个惊喜——点睛之笔
- 展示：综合应用提高部分
  - ✳ 从二叉树到栈——超复杂多冗余表达式求值
  - ✳ 动态Huffman的正确使用姿势——实时多媒体压缩传输

# 基础部分——线性结构

- KMP
  - ✳ 可输出全部匹配



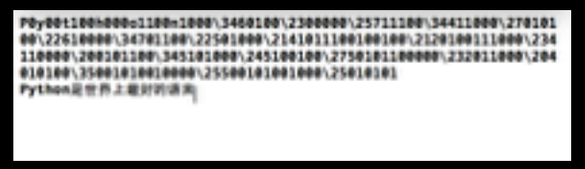

```
请输入主串:
aabbabababab
请输入模式串:
ab
Found at shift 1
Found at shift 4
Found at shift 6
Found at shift 8
Found at shift 10
```

- 优先队列、跳表、稀疏矩阵
  - ✳ 全部面向对象，方法丰富，便于调试和演示
  - ✳ 优先队列使用模板类，支持任意类型元素和优先级

稀疏矩阵：优化——统计每列非零元个数、转置时用数组存每列起始位置等

# 基础部分——树型结构

- 动态Huffman树
  - ✳ 静态使用内存（原理：字符个数有上界）
  - ✳ 插入时O(1)定位已有字符
  - ✳ 文本实时转换（GUI）


- 森林与二叉树互转
  - ✳ 使用模板类
  - ✳ 提供标准化互转函数

```
template<class T> Forest<T>* BinTree<T>::toForest()
template<class T> BinTree<T>* Tree<T>::toBinTree()
```

# 基础部分——树型结构

- 前序、后序线索树
  - 使用模板类
  - 支持任意类型键值
  - 下拉列表式GUI演示
  - 简明清晰

# 基础部分——图结构

- 最短路
  - 使用优先队列的Bellman-Ford：支持负权图和负环检测
  - 支持负权回路检测的Floyed
  - 基于拓扑排序的DAG最快算法：O(n)

floyed：（sp[i][i]=?0）

# 基础部分——查找和散列

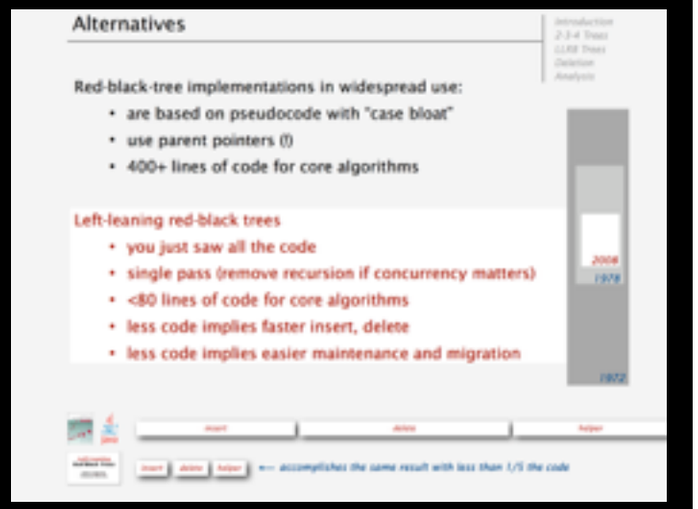
- AVL树
  - ✳ 非递归化插入、删除
  - ✳ 动态更新平衡因子
  - ✳ 可图形化输出AVL树
  - ✳ 模板类

```
1. Insert 2. Delete 3. Update 4. Search 5. Print 0. Exit
1
Input data:
65
1. Insert 2. Delete 3. Update 4. Search 5. Print 0. Exit
1
Input data:
-23
1. Insert 2. Delete 3. Update 4. Search 5. Print 0. Exit
1
Input data:
22
1. Insert 2. Delete 3. Update 4. Search 5. Print 0. Exit
1
Input data:
98
1. Insert 2. Delete 3. Update 4. Search 5. Print 0. Exit
5
The AVL Tree is:
                98*1[0]
        65*1[0]
                53*1[0]
34*1[0]
                23*2[0]
        22*1[0]
                -23*1[0]
```

没有存高度

# 基础部分——查找和散列

- 红黑树
  - ✳ LLRB
  - ✳ 核心代码<80行
  - ✳ 与2-3-4树或2-3树对应
- 基于红黑树的字典
  - ✳ 模仿Python中的dict
- 当然，全是模板类




```
Dict<string, People> Family;
Family["Father"] = People("Yang Dejun",
                          "Shandong",
                          45,
                          People::MALE);
```

完整实现只有100多行

# 基础部分——查找和散列

- 字符串Hash
  - ✴ 开放域寻址法
  - ✴ 拉链法
  - ✴ 模板类——任意Value
  - ✴ 四个测试（包括NodeInfo）
    - ✤ 相似字符串（abc）
    - ✤ 随机串
    - ✤ 通讯录
    - ✤ 给定200MB数据

```
const char Surname[24][8] = {"An", "Bai", "Cai", "Deng", "Fan", "Guo",
    "Han", "Ji", "Kuang", "Li", "Mu", "Nan", "Ou", "Piao", "Qian", "Sun"
    , "Wu", "Wang", "Yang", "Yu", "Zhang", "Zhao", "Zheng", "Zhou"};
const char Given[24][15] = {"Alice", "Bob", "Cindy", "Dan", "Einstein",
    "Faham", "God", "Helen", "Ivan", "Jack", "Keven", "Linda", "Mary",
    "Nancy", "Oliver", "Percy", "Qiqi", "Ran", "Sandy", "Tank", "Urban",
    "Vanon", "Wilfred", "Zack"};
```

- 字符串Ha
  - ✴ 开放域寻
  - ✴ 拉链法
  - ✴ 模板类—
  - ✴ 四个测试
    - ❖ 相似字
    - ❖ 随机串
    - ❖ 通讯录
    - ❖ 给定20

```
AVLTree<string, bool> used;
cout << "Generating data:" << endl;
for (int i = 1; i <= n; ) {
    string name;
    name += Given[rand()%24];
    name += " ";
    name += Surname[rand()%24];
    if (used.hasKey(name)) {
        continue;
    }
    else {
        char TEL[13] = "13000000000";
        TEL[1] += rand() % 6;
        for(int j = 2; j < 11; ++j)
            TEL[j] += rand() % 10;
        v[i].name = name;
        v[i++].tel += TEL;
        fprintf(fp, "%s\n%s\n", name.c_str(), TEL);
        used.insert(name, true);
    }
}
cout << "Generation finished." << endl;
```

# 基础部分——查找和散列

- 字符串Hash
  - ✴ 开放域寻址法
  - ✴ 拉链法
  - ✴ 模板类——任意Value
  - ✴ 四个测试（包括NodeInfo）
    - ✤ 相似字符串（abc）
    - ✤ 随机串
    - ✤ 通讯录
    - ✤ 给定200MB数据

# 基

- 字符串H
  - 开放域
  - 拉链法
  - 模板类
  - 四个测
    - 相似字
    - 随机串
    - 通讯录
    - 给定2

```
Input dataSize:
100000
Input load factor:
0.8
dataSize = 100000, loadFactor = 0.8
Distribution 1: Similar strings
Generating data:
Generation finished.
Inserting all (key = string, value = length) pairs to
OpenStrHashTable...
Crashes: 280708, Time used: 0.013236
Searching all keys in OpenStrHashTable...
Crashes: 280708, Time used: 0.010574
Inserting all (key = string, value = length) pairs to
ListStrHashTable...
Crashes: 40201, Time used: 0.020413
Searching all keys in ListStrHashTable...
Crashes: 40201, Time used: 0.01132
Distribution 2: Random strings
Generating data:
Generation finished.
Inserting all (key = string, value = length) pairs to
OpenStrHashTable...
Crashes: 199249, Time used: 0.017992
Searching all keys in OpenStrHashTable...
Crashes: 199249, Time used: 0.012289
Inserting all (key = string, value = length) pairs to
ListStrHashTable...
Crashes: 40038, Time used: 0.019399
Searching all keys in ListStrHashTable...
Crashes: 40038, Time used: 0.016485
Distribution 3: Randomly combined contactbook
Generating data:
Generation finished.
Inserting all (key = string, value = People*) pairs to
```

# 基

- 字符串H
  - ✴ 开放域
  - ✴ 拉链法
  - ✴ 模板类
  - ✴ 四个测
    - ✤ 相似
    - ✤ 随机
    - ✤ 通讯
    - ✤ 给定2

```
Inserting all (key = string, value = length) pairs to
OpenStrHashTable...
Crashes: 199249, Time used: 0.017992
Searching all keys in OpenStrHashTable...
Crashes: 199249, Time used: 0.012289
Inserting all (key = string, value = length) pairs to
ListStrHashTable...
Crashes: 40038, Time used: 0.019399
Searching all keys in ListStrHashTable...
Crashes: 40038, Time used: 0.016485
Distribution 3: Randomly combined contactbook
Generating data:
Generation finished.
Inserting all (key = string, value = People*) pairs to
OpenStrHashTable...
Crashes: 346, Time used: 1.6e-05
Searching all keys in OpenStrHashTable...
Crashes: 346, Time used: 1.5e-05
Inserting all (key = string, value = People*) pairs to
ListStrHashTable...
Crashes: 52, Time used: 1.8e-05
Searching all keys in ListStrHashTable...
Crashes: 52, Time used: 1.1e-05
Distribution 4: Given 200MB data
Inserting all (key = string, value = int) pairs to
OpenStrHashTable...
Crashes: 4717170, Time used: 1.01378
Searching all keys in OpenStrHashTable...
Crashes: 4717170, Time used: 1.03259
Inserting all (key = string, value = int) pairs to
ListStrHashTable...
Crashes: 940175, Time used: 1.24108
Searching all keys in ListStrHashTable...
Crashes: 940175, Time used: 0.97101
```

# 基础部分——排序

- 快速排序
  - ✳ 6种优化，6种测试数据
  - ✳ 模板函数—任意类型排序
  - ✳ 自动绘制统计比较图
  - ✳ 自动测定性能曲线

```
1. 不同优化横向评测 2. 性能曲线测定
1
1. 从中间向两边减小的数据（卡取中）
2. 每个二分区间的中点为最大值的数据（卡取首和三者取中）
3. 顺序数据（卡取首）
4. 逆序数据（卡取首）
5. int范围内随机数据
6. 所有数字全相同的数据
```

```
1. 不同优化横向评测 2. 性能曲线测定
2
1. 从中间向两边减小的数据（卡取中）
2. 每个二分区间的中点为最大值的数据（卡取首和三者取中）
3. 顺序数据（卡取首）
4. 逆序数据（卡取首）
5. int范围内随机数据
6. 所有数字全相同的数据
3
1. 取首快排
2. 取中快排
3. 随机快排
4. 0.618+尾递归快排
5. 三者取中快排
6. 取中+去重+小区间插排
```

# 基础部分——排序

- 线性排序
  - ✳ 优化：支持负数
  - ✳ 设计了6种测试数据
  - ✳ 模板函数—任意类型排序
  - ✳ 自动绘制统计比较图
  - ✳ 自动测定性能曲线

```
1. 三种算法横向评测 2. 性能曲线测定
1
1. 重复数据
2. 负数数据
3. 顺序数据
4. 逆序数据
5. 一般随机数据
6. 长整数数据
7. 自己输入数据
4
数据规模： 2000
radix_sort time used: 2.12333ms
counting_sort time used: 0.0223333ms
bucket_sort time used: 0.419667ms
```

```
1. 三种算法横向评测 2. 性能曲线测定
2
1. 重复数据
2. 负数数据
3. 顺序数据
4. 逆序数据
5. 一般随机数据
6. 长整数数据
7. 自己输入数据
5
1. 计数排序
2. 基数排序
3. 桶排序
1
数据规模区间端点和步长： 100 1000 10
100 0.0526667ms
110 0.0416667ms
120 0.042ms
```

计数排序：计算极差
基数排序：用19个队列来排就可以了

“Talk is cheap, show me the demo.”


“What Surprise?”

一个惊喜 点睛之笔
其实是告别软设之作

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先
- 跳表
- 稀疏
- 动态
- 前

```
PriorityQueue<int, int> queue(100);
queue.push(30, 12);
queue.push(20, 8);
queue.push(12, 9);
queue.push(80, -4);
while (!queue.isempty()) {
    cout << queue.peek() << endl;
    queue.pop();
}
```

叉树

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳
- 森
- 云
- 前序、后序线索树
- 森林、普通树、二叉树
- 哈希表

```
SkipList<string> skiplist(100, 1<<30);
skiplist.insert(10, "asdf");
skiplist.insert(11, "abcdefg");
skiplist.insert(12, "abddcdefg");
skiplist.insert(13, "abewscdefg");
skiplist.print();
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 森林、普通树、二叉树
- 
- 
- 
- 前序、后序线索树
- 哈希表

```
SparseMatrix sparsematrix(5, 5, 7);
sparsematrix.insert(1, 2, 888);
sparsematrix.insert(2, 3, 666);
SparseMatrix rev(sparsematrix.reverse());
rev.printmatrix();
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 森林、普通树、二叉树

```
cout << HuffmanTree::encode("Python大法好") << endl;
cout <<
    HuffmanTree::decode(HuffmanTree::encode("Python
    大法好").c_str());
```

- 动态哈夫曼树
- 字典
- 前序、后序线索树
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

二叉树

```
PreThreadBT<int> tree;
BTNode<int> *root = new BTNode<int>(1);
tree.initialize(root);
tree.insertLeft(root, new BTNode<int>(2));
tree.insertRight(root, new BTNode<int>(3));
tree.insertLeft(root->lc, new BTNode<int>(4));
tree.insertRight(root->lc->lc, new BTNode<int>(9));
tree.insertLeft(root->lc->lc, new BTNode<int>(8));
tree.insertRight
tree.insertRight
tree.insertLeft(
tree.PreOrderTra
```

```
PostThreadBT<int> tree;
BTNode<int> *root = new BTNode<int>(1);
tree.initialize(root);
tree.insertLeft(root, new BTNode<int>(2));
tree.insertRight(root, new BTNode<int>(3));
tree.insertLeft(root->lc, new BTNode<int>(4));
tree.insertRight(root->lc->lc, new BTNode<int>(9));
tree.insertLeft(root->lc->lc, new BTNode<int>(8));
tree.insertRight(root->lc, new BTNode<int>(5));
tree.insertRight(root->rc, new BTNode<int>(7));
tree.insertLeft(root->rc, new BTNode<int>(6));
tree.PostOrderTraversal();
```

- 动态暗关
- 前序、后序

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

```
BinTree<string> bintree(10); //会要求输入10个节点
Forest<string> *forest = bintree.toForest();
forest->Print();
cout << "Convert back to binary tree:" << endl;
bintree = *forest->toBinTree();
bintree.Print();


Forest<string> forest(10); //会要求输入10个节点
forest.toBinTree()->Print();
cout << "Convert back to forest:" << endl;
forest.toBinTree()->toForest()->Print();
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳
- 稀
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- 字典
- 哈希表

```
AVLTree<int, int> tree; //key, value
tree.insert(3, 10);
tree.insert(9, 20);
tree.search(3);
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- 字典
- 哈希表

```
RBTree<int, int> tree; //key, value
tree.insert(3, 10);
tree.insert(9, 20);
cout << tree.get(3) << endl;
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

```
Dict<string, People> Family;
bool MALE = true, FEMALE = false;
Family["Father"] = People("Yang Dejun", "Shandong",
    45, MALE);
Family["Mother"] = People("Zhang Shuying",
    "Shandong", 44, FEMALE);
Family["Son"] = People("Yang Zhifei", "Shandong", 19
    , MALE);
Family["Daughter"] = People("Yang Yifei", "Shandong"
    , 3, FEMALE);
cout << "The daughter " << Family["Daughter"].
    getName() << " is from " << Family["Daughter"].
    getProvince() << " province." << endl;
```

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 告别软设之作——数据结构标准库

```
string s[] = {"abc", "def", "ghi"};
{
    ListStrHashTable<int> listhash(100);
    listhash.insert(s[0], 0);
    listhash.insert(s[1], 1);
    listhash.insert(s[2], 2);
    cout << listhash.get(s[1]) << endl;
}
{
    OpenStrHashTable<int> openhash(100);
    openhash.insert(s[0], 0);
    openhash.insert(s[1], 1);
    openhash.insert(s[2], 2);
    cout << openhash.get(s[1]) << endl;
}
```

- 对

没有存高度

# 告别软设之作——数据结构标准库

**DataStructure.h**

- 优先队列
- 跳表
- 稀疏矩阵
- 动态哈夫曼树
- 前序、后序线索树
- 森林、普通树、二叉树
- **AVL树**
- 红黑树
- 字典
- 哈希表

没有存高度

# 提高部分——复杂表达式求值

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树
  - 给出一个具有冗余括号的表达式

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树
  - 给出一个具有冗余括号的表达式

```
1+2*(((3-(4+5*2))))-6/8
1+2*(3-4-5*2)-6/8 = 18.25
```

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树

✳ 给出一个具有冗余括号的表达式

- 1. 将其化为最简形式

```
1+2*(((3-(4+5*2))))-6/8
1+2*(3-4-5*2)-6/8 = 18.25
```

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树
  - 给出一个具有冗余括号的表达式
    - 1. 将其化为最简形式
    - 2. 对其求值

```
1+2*(((3-(4+5*2))))-6/8
1+2*(3-4-5*2)-6/8 = 18.25
```

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树

✳ 给出一个具有冗余括号的表达式

- 1. 将其化为最简形式

```
1+2*(((3-(4+5*2))))-6/8
1+2*(3-4-5*2)-6/8 = 18.25
```

- 2. 对其求值

Is that ALL?

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 表达式树

✳ 给出一个具有冗余括号的表达式

- 1. 将其化为最简形式

```
1+2*(((3-(4+5*2))))-6/8
1+2*(3-4-5*2)-6/8 = 18.25
```

- 2. 对其求值

Is that ALL?

**Of course NOT!**

出现两条要求以后，讲递归做法

# 提高部分——复杂表达式求值

- 基于栈的任意浮点四则表达式求值

✳ +-(1+2)=?

✳ 1./-(2.3*(2.+-.6666))=?

✳ .57025*(3/+-(.6+.8)-4.2*-(3—(.5/.5))+-8.0/(6+8)—+6)=?

```
Result:
.57025*(3/+-(.6+.8)-4.2*-(3--(.5/.5))+-8.0/(6+8)--+6) =
11.45387857142857
```

```
[15:21:00] Xivid:~ $ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on
Type "help", "copyright", "credits" or "license" for more
>>> .57025*(3/+-(.6+.8)-4.2*-(3--(.5/.5))+-8.0/(6+8)--+6)
11.453878571428573
```

# 提高部分——复杂表达式求值

- 基于栈的任意浮点四则表达式求值
  - ✳ 先将输入的中缀表达式转化为后缀表达式（栈1）
  - ✳ 再对后缀表达式求值（栈2）

```
The infix expression you input is saved as:
1+2+3

========================================
Postfix expression construct process:
("P+" means "Priority +")

Postfix Expression(under construction):
1
[Operator Stack] +[P1] pushed.
Stack State: +[P1]
Postfix Expression(under construction):
1
Postfix Expression(under construction):
1 2
[Operator Stack] +[P2] pushed.
Stack State: +[P1] +[P2]
Postfix Expression(under construction):
1 2
Postfix Expression(under construction):
1 2 3
[Operator Stack] +[P2] popped.
Stack State: +[P1]
[Operator Stack] +[P1] popped.
Stack State: Empty!

========================================
The corresponding postfix expression is:
1 2 3 + +
```

```
Evaluation:
[Operand Stack] 1 pushed.
Stack State: 1
[Operand Stack] 2 pushed.
Stack State: 1 2
[Operand Stack] 3 pushed.
Stack State: 1 2 3
Run into operator +
[Operand Stack] 3 popped.
Stack State: 1 2
[Operand Stack] 2 popped.
Stack State: 1
2+3=6
[Operand Stack] 6 pushed.
Stack State: 1 6
Run into operator +
[Operand Stack] 6 popped.
Stack State: 1
[Operand Stack] 1 popped.
Stack State: Empty!
1+6=7
[Operand Stack] 7 pushed.
Stack State: 7
[Operand Stack] 7 popped.
Stack State: Empty!
```

# 提高部分 式求值

- 基于 式求值

  ✳ 先将输 （栈1）

  ✳ 再对后

```
The infix expression you input is saved as:
1+2*3

============================================
Postfix expression construct process:
("P*" means "Priority *")

Postfix Expression(under construction):
1
[Operator Stack] +[P1] pushed.
Stack State: +[P1]
Postfix Expression(under construction):
1
Postfix Expression(under construction):
1 2
[Operator Stack] *[P2] pushed.
Stack State: +[P1] *[P2]
Postfix Expression(under construction):
1 2
Postfix Expression(under construction):
1 2 3
[Operator Stack] *[P2] popped.
Stack State: +[P1]
[Operator Stack] +[P1] popped.
Stack State: Empty!

============================================
The corresponding postfix expression is:
1 2 3 * +
```

# 提高部分——复杂表达式求值

- 基于栈的任意浮点四则表达式求值
  - ✳ 先将输入的中缀表达式转化为后缀表达式（栈1）
  - ✳ 再对后缀表达式求值（栈2）

```
Evaluation:
[Operand Stack] 1 pushed.
Stack State: 1
[Operand Stack] 2 pushed.
Stack State: 1 2
[Operand Stack] 3 pushed.
Stack State: 1 2 3
Run into operator *
[Operand Stack] 3 popped.
Stack State: 1 2
[Operand Stack] 2 popped.
Stack State: 1
2*3=6
[Operand Stack] 6 pushed.
Stack State: 1 6
Run into operator +
[Operand Stack] 6 popped.
Stack State: 1
[Operand Stack] 1 popped.
Stack State: Empty!
1+6=7
[Operand Stack] 7 pushed.
Stack State: 7
[Operand Stack] 7 popped.
Stack State: Empty!
```

# 提高 …值

- 基于 …值
  - 先将 …
  - 再对 …

```
Evaluation:
[Operand Stack] 1 pushed.
Stack State: 1
[Operand Stack] 2 pushed.
Stack State: 1 2
[Operand Stack] 3 pushed.
Stack State: 1 2 3
Run into operator *
[Operand Stack] 3 popped.
Stack State: 1 2
[Operand Stack] 2 popped.
Stack State: 1
2*3=6
[Operand Stack] 6 pushed.
Stack State: 1 6
Run into operator +
[Operand Stack] 6 popped.
Stack State: 1
[Operand Stack] 1 popped.
Stack State: Empty!
1+6=7
[Operand Stack] 7 pushed.
Stack State: 7
[Operand Stack] 7 popped.
Stack State: Empty!
```

# 提高部分——复杂表达式求值

- 基于栈的任意浮点四则表达式求值
  - ✳ 先将输入的中缀表达式转化为后缀表达式（栈1）
  - ✳ 再对后缀表达式求值（栈2）

## 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码

  ✳ 动态哈夫曼编码算法：在线压缩算法

  ✳ 可以在未知完整数据前压缩和解压信息

```
HuffmanTree::encode("abcabcabababababababababababc"):
1111111100000000000111111110111010110110110110110110110110110101101110
```

```
>>> len("abcabcabababababababababababc")
27
```

# 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码

  ✳ 动态哈夫曼编码算法：在线压缩算法

  ✳ 可以在未知完整数据前压缩和解压信息

```
HuffmanTree::encode("abcabcabababababababababababc"):
11111111000000000001111111101110101101101101101101101101101101110
```

```
>>> len("abcabcabababababababababababc")
27
```

```
>>> len("11111111000000000001111111101110101101101101101101101101101101110")/8
8
```

## 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码

  ✳ 动态哈夫曼编码算法：在线压缩算法

  ✳ 可以在未知完整数据前压缩和解压信息

```
HuffmanTree::encode("abcabcabababababababababababc"):
11111111000000000001111111101110101101101101101101101101101101110
```

```
>>> len("abcabcabababababababababababc")
27
```

```
>>> len("11111111000
8
```

## 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码

# 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码
  - ✳ 然而由于压缩率远低于离线压缩算法，并不适合用来编写压缩软件。

# 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码
  - ✳ 然而由于压缩率远低于离线压缩算法，并不适合用来编写压缩软件。
  - ✳ 那么正确的使用姿势是什么样的?

## 提高部分——动态Huffman实时压缩传输

- 动态哈夫曼编码
  - ✳ 然而由于压缩率远低于离线压缩算法，并不适合用来编写压缩软件。
  - ✳ 那么正确的使用姿势是什么样的?

在线压缩传输!

## 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟

只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

## 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟
  - ✳ 使用文件输入输出来模拟

只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

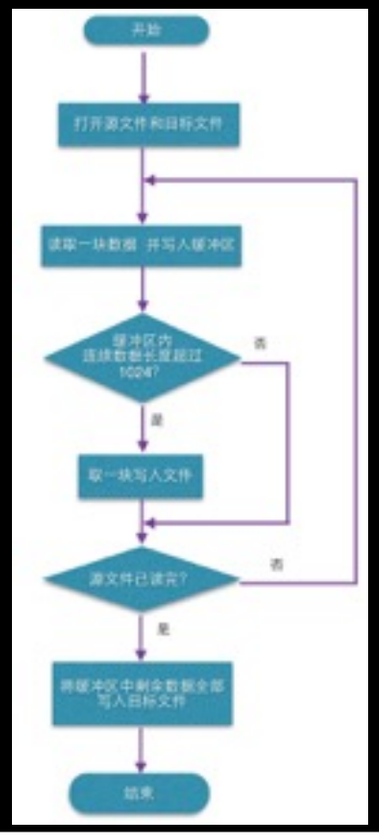
## 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟
  - ✳ 使用文件输入输出来模拟
  - ✳ 按数据包发送到缓冲区

只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

# 提高部分——动态Huffman实时压缩传输

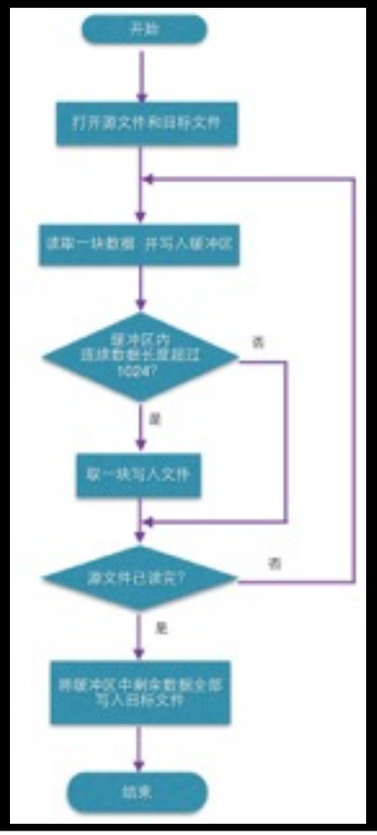
- 在线传输的模拟
  - ✳ 使用文件输入输出来模拟
  - ✳ 按数据包发送到缓冲区
  - ✳ 缓冲区中连续数据长度>阈值则输出




只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

# 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟
  - ✳ 使用文件输入输出来模拟
  - ✳ 按数据包发送到缓冲区
  - ✳ 缓冲区中连续数据长度>阈值则输出
  - ✳ 数据包乱序怎么办?

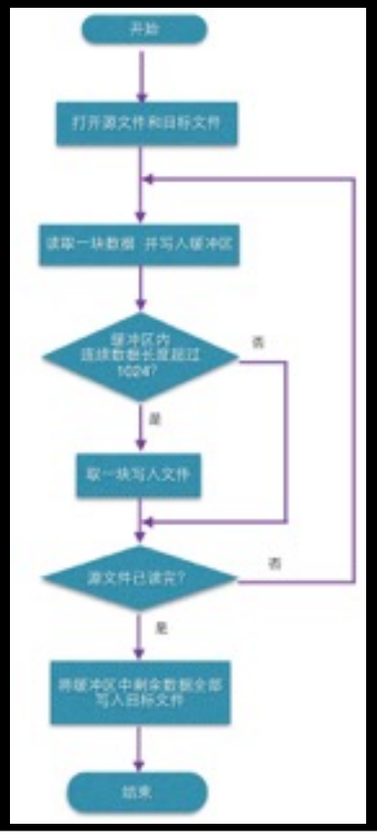



只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

## 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟
  - ✳ 使用文件输入输出来模拟
  - ✳ 按数据包发送到缓冲区
  - ✳ 缓冲区中连续数据长度>阈值则输出
  - ✳ 数据包乱序怎么办?

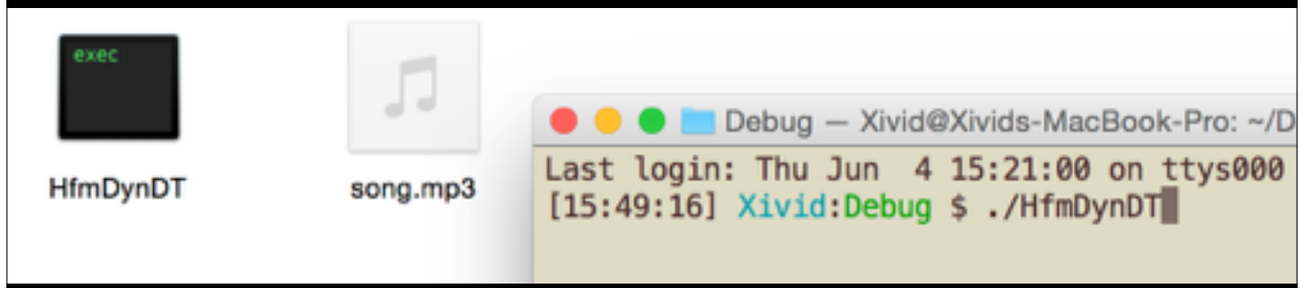
使用优先队列！




只解压最前面的数据包，与前端接不上的直接存在缓冲区，暂不解压（接收时要传实际长度和压缩长度）

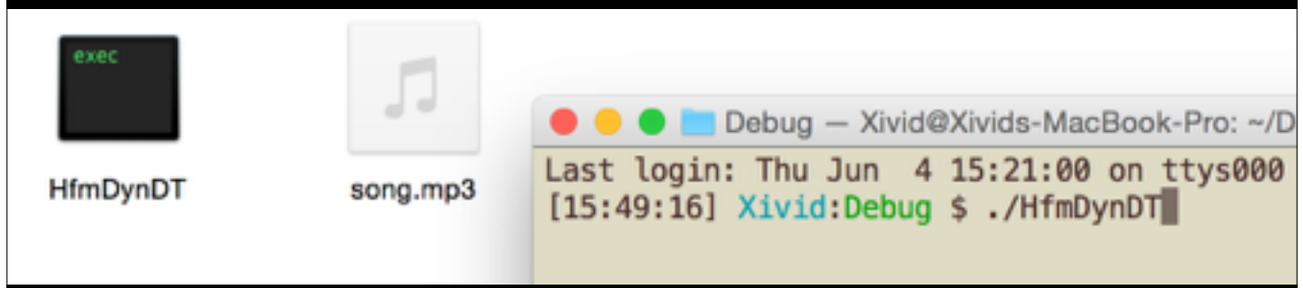
# 提高部分——动态Huffman实时压缩传输

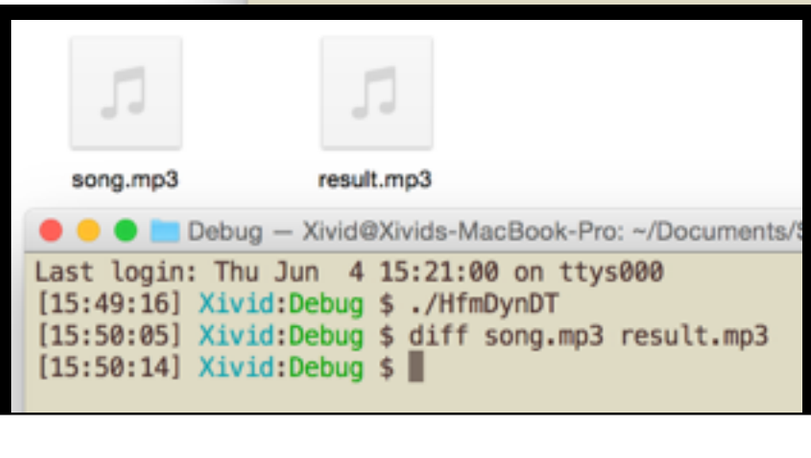
- 在线传输的模拟



# 提高部分——动态Huffman实时压缩传输

- 在线传输的模拟



## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输

## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输
  - ✳ 服务端发送数据包

## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输
  - ✳ 服务端发送数据包
  - ✳ 改为经动态Huffman编码后的数据包

## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输
  - ✳ 服务端发送数据包
  - ✳ 改为经动态Huffman编码后的数据包
  - ✳ 客户端接收数据包

## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输
  - **服务端发送数据包**
  - 改为经动态Huffman编码后的数据包
  - **客户端接收数据包**
  - 先用本地Huffman树解码再存入缓冲区

## 提高部分——动态Huffman实时压缩传输

- 实时压缩传输
  - ✳ **服务端发送数据包**
  - ✳ 改为经动态Huffman编码后的数据包
  - ✳ **客户端接收数据包**
  - ✳ 先用本地Huffman树解码再存入缓冲区

**节省网络流量！**

谢谢！