第 6、7 章 集合与搜索

一、集合存储
二、集合操作的实现
三、查找的基本概念
四、二分查找
五、二叉排序树（二叉搜索树）
六、B-树
七、索引查找
八、哈希或散列(HASH)查找

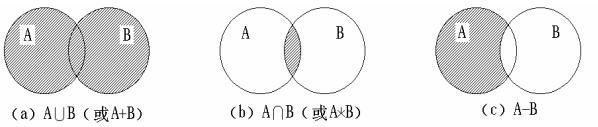
集合及其表示

集合基本概念

- 集合是成员(对象或元素)的一个群集。集合中的成员可以是原子(单元素)，也可以是集合。
- 集合的成员必须互不相同。
- 集合中的成员一般是无序的，没有先后次序关系。
- 表示方法：{1,2,3,4,5,6,7,8,9,10}

集合的操作

- 集合操作有求集合的并、交、差、判存在等。



(a) A∪B（或A+B)　(b) A∩B（或A×B)　(c) A-B

集合运算的文氏(Venn)图

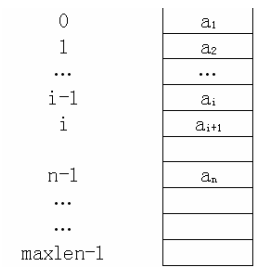
如何在计算中实现集合?

- (1) 逻辑结构的设定
- (2) 存储结构
- (3) 算法

线性结构

- 顺序存储——顺序表
- 链式存储——链表

顺序存储——顺序表



顺序表(SeqList)类的定义

```
const int defaultSize = 100;
template <class T>
class SeqList: public LinearList<T> {
protected:
    T *data;              //存放数组
    int maxSize;          //最大可容纳表项的项数
    int last;             //数组中最后一个元素的下标
    void reSize(int newSize);    //改变数组空间大小
public:
    SeqList(int sz = defaultSize);        //构造函数
    SeqList(SeqList<T>& L);               //复制构造函数
    ~SeqList() {delete[ ] data;}          //析构函数
    int Size() const {return maxSize;} //求表最大容量
    int Length() const {return last+1;}      //计算表长度
    int Search(T& x) const;
        //搜索x在表中位置，函数返回表项序号
    int Locate(int i) const;
        //定位第 i 个表项，函数返回表项序号
    bool getData(int i, T & x); //取第i个元素
    bool Insert(int i, T &x);                  //插入
    bool Remove(int i, T& x);          //删除
    ……
};
```

集合操作的实现

集合的“并”运算

P52

集合的“并”运算

```
void Union ( SeqList<int>& LA,
                    SeqList<int>& LB ) {
   int n1 = LA.Length ( ), n2 = LB.Length ( );
   int i, k, x;
   for ( i = 0; i < n2; i++ ) {
      x = LB.getData(i);          //在LB中取一元素
      k = LA.Search(x);           //在LA中搜索它
      if (k == 0)          //若在LA中未找到插入它
        { LA.Insert(n1, x); n1++; }
                          //插入到第n个表项位置}
}
```

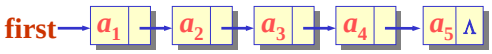
集合操作的实现

集合的“交”运算

集合的“交”运算

```
void Intersection ( SeqList<int> & LA,
                          SeqList<int> & LB ) {
   int n1 = LA.Length ( );
   int x, k, i = 0;
   while ( i < n1 ) {
      x = LA.getData(i);       //在LA中取一元素
      k = LB.Search(x);        //在LB中搜索它
      if (k == 0)              //若在LB中未找到
        { LA.Remove(i, x); n1--; } //在LA中删除它
      else i++;                //未找到在A中删除它
   }
}
```

链式结构



链表的结点类

```
template <class T>

struct LinkNode {      //链表结点类的定义
   T data;                    //数据域
   LinkNode<T> * link;      //链指针域
};
```

单链表类

```
template <class T>
class List {    //单链表类定义
protected:
   LinkNode<T> * first;  //表头指针
public:
   List() { first = new LinkNode<T>; }  //构造函数
   List(T x) { first = new LinkNode<T>(x); }
   List( List<T>& L);            //复制构造函数
   ~List(){ }                    //析构函数
   void makeEmpty();             //将链表置为空表
   int Length() const;           //计算链表的长度
   ……
```

集合操作的实现——链表

集合的逻辑结构——树形结构

输入关系	分离集合
初始状态	{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}
(2,4)	{1}{2,4}{3}{5}{6}{7}{8}{9}{10}
(5,7)	{1}{2,4}{3}{5,7}{6}{8}{9}{10}
(1,3)	{1,3}{2,4}{5,7}{6}{8}{9}{10}
(8,9)	{1,3}{2,4}{5,7}{6}{8,9}{10}
(1,2)	{1,2,3,4}{5,7}{6}{8,9}{10}
(5,6)	{1,2,3,4}{5,6,7}{8,9}{10}
(2,3)	{1,2,3,4}{5,6,7}{8,9}{10}

操作的分析

- 1－判断两个元素是否属于同一个集合(查找过程)
- 2－合并两个不相交集合(合并过程)

并查集(Disjoint Set)

- 英文：Disjoint Set，即“不相交集合”
- 即将编号为1...N的N个对象划分为不相交集合，在每个集合中，选择其中某个元素代表所在集合。
- 并查集是一种树型的数据结构，用于处理一些不相交集合的合并问题。
- 并查集的主要操作有
- 1－合并两个不相交集合(合并过程)
- 2－判断两个元素是否属于同一个集合(查找过程)
- 3－路径压缩

并查集——逻辑结构的组织

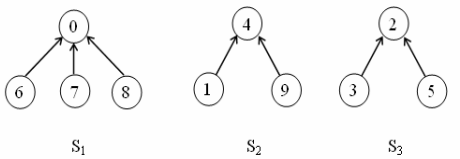
用根树来表示集合,树中的每个节点包含集合的一个成员,每棵树表示一个集合。

多个集合形成森林态,以每棵树的树根作为集合的代表,并且根结点的父结点指向其自身,树上的其他结点都用一个父指针表示它的附属关系。

用树表示集合,使得各种操作更加高效率。

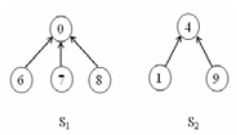
用树表示集合

S_1={0，6，7，8}
S_2={4，1，9}
S_3={2，3，5}
用树表示如下：

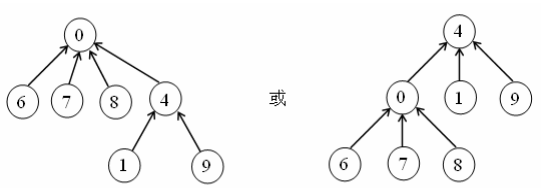


其中，每个集合用一棵树表示，且分枝由子女指向双亲。

实现并操作可直接使两棵树之一成为另一棵树的子树，这样$S_1 \cup S_2$可表示为下列两种形式之一：

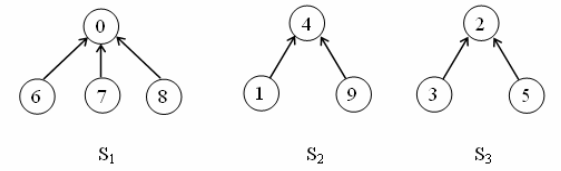


S1 ∪ S2



存储结构的实现

- 静态链式结构——反映双亲关系



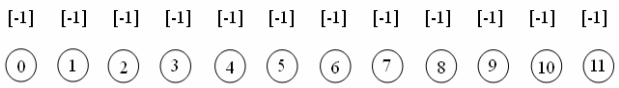
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

并查集操作过程的分析例子

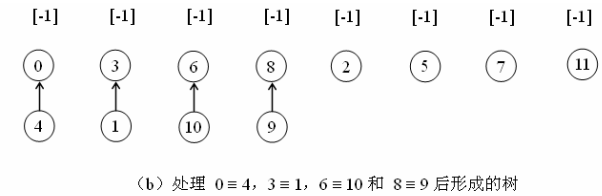
例 考虑S＝{0，1，2，3，4，5，6，7，8，9，10，11}，S上存在以下关系对：0≡4，3≡1，6≡10，8≡9，7≡4，6≡8，3≡5，2≡11，11≡0，那么利用并查集方法进行关系划分如何？

处理过程

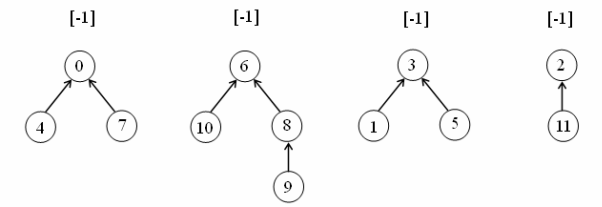
(1) 初始时有12棵树，每棵树包含一个元素。



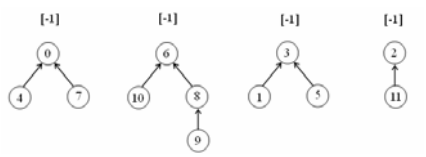
(a) 初始树

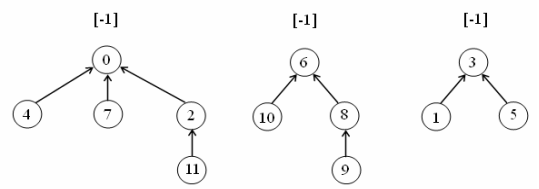


（b）处理 0≡4，3≡1，6≡10 和 8≡9 后形成的树



（c）处理 7≡4，6≡8，3≡5 和 2≡11 后形成的树





（d）处理 11≡0 后形成的树

存储结构的总结

由于集合元素为0, 1, 2, 3, …, n – 1，可以用数组parent[MaxElements]表示树结点。

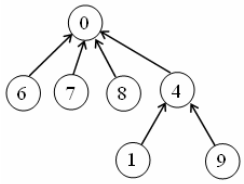
数组的第i个位置表示元素i对应的树结点。数组元素存放相应树结点的双亲指针。

并查集类的描述

```
class Sets {
public:
  // 集合操作
   …
private:
  int *parent;   // 动态数组
  int n;         // 集合元素个数
};
Sets::Sets (int sz = DefaultSize)
{    n = sz;
     parent = new int[sz];
     for (int i = 0; i < n; i++) parent[i] = -1;
}
```

操作的实现

- 查找
- 合并

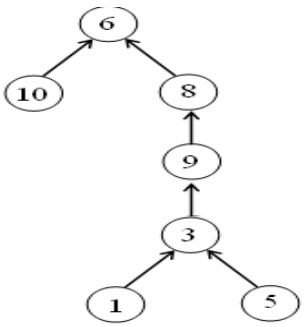


查找算法

实现find(i)只需简单地沿着结点i的parent向上移动，直至到达parent值为–1的结点：

```
int Sets::SimpleFind (int i) // 找到含元素i的树的根
 {
   while (parent[i] >= 0) i = parent[i];
   return i;
 }
```

问题的发现



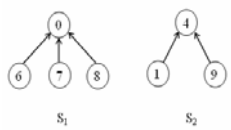
坍塌规则
(路径压缩)

改进算法——坍塌规则

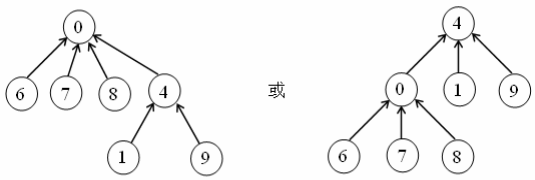
函数CollapsingFind采用了坍塌规则：

```
int Sets::CollapsingFind (int i )
{   // 找到含元素i的树的根，采用坍塌规则缩短
    //从i到根的所有结点到根的距离
  for (int r = i; parent[r] >= 0; r = parent[r]);   // 找到根
  while ( i != r )
  {
     int s = parent[i];
     parent[i] = r;
     i = s;
  }
  return r;
}
```

合并操作union(i, j)



S1 ∪ S2



合并操作union(i, j)

这里假设采用令第一棵树为第二棵的子树的策略：

// 用以i 为根和以j(i!=j)为根的两个不相交集合的并取代它们

```
void Sets::SimpleUnion (int i, int j)
{
    parent[i] = j;
}
```

问题的发现

执行以下操作序列：

union(0, 1),
union(1, 2),
union(2, 3),
…,
union(n–2, n–1)

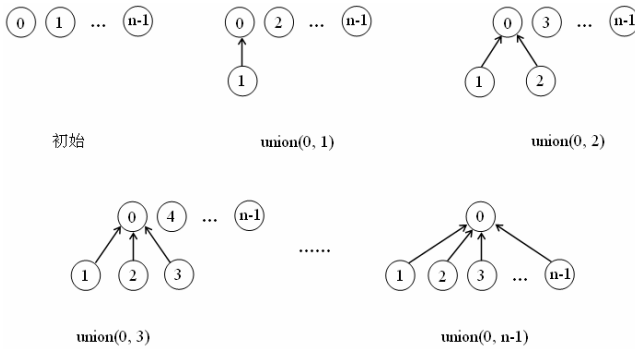
将生成下一页所示的退化树。

改进方法

(1) 以树高为依据

(2) 以结点个数为依据

分析结点个数的解决方案



分析结点个数的解决方案

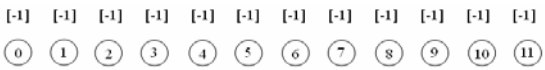
可以对union(i, j)操作应用以下权重规则：若以i为根的树中结点数小于以j为根的树中结点数，则令j成为i的双亲，否则令i成为j的双亲。

当应用权重规则时，执行前述union操作序列生成的树如下一页所示。其中，对union操作的参数作了适当修改，使其与树根对应。

如何保存树的结点个数？

```
class Sets
{
   int *parent;
   int n;
   ….?
};
```

分析结点个数的解决方案



节省空间

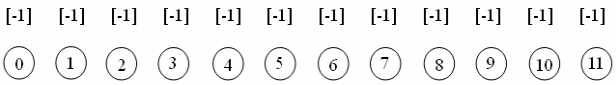
为了实现权重规则，可利用根结点的parent字段以负数的形式存储计数数据。

合并过程改进方案的分析例子

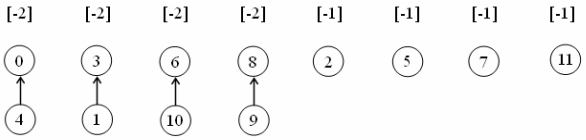
例 考虑S＝{0，1，2，3，4，5，6，7，8，9，10，11}，S上存在以下关系对：0≡4，3≡1，6≡10，8≡9，7≡4，6≡8，3≡5，2≡11，11≡0，那么利用并查集方法进行关系划分的过程如下：

(1) 初始时有12棵树，每棵树包含一个元素。

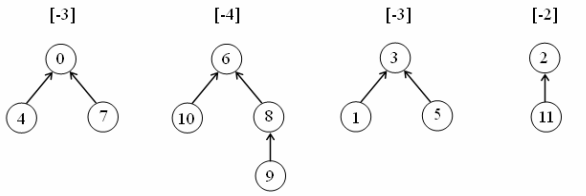
每处理一个等价对以后，树的结构变化如下：



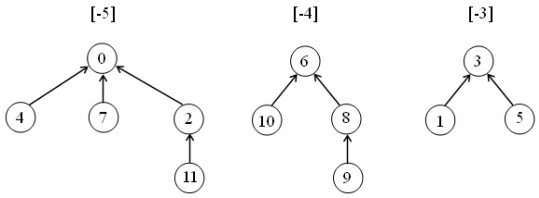
(a) 初始树



（b）处理 0≡4，3≡1，6≡10 和 8≡9 后形成的树



（c）处理 7≡4，6≡8，3≡5 和 2≡11 后形成的树



（d）处理 11≡0 后形成的树

合并的改进算法

为了实现权重规则，可利用根结点的parent字段以负数的形式存储计数数据。由此可得基于权重规则的union算法：

```
void Sets::WeightedUnion (int i, int j)
{// 基于权重规则构造以i和j为根的集合的并
  int temp = parent[i] + parent[j];
  if ( parent[j] < parent[i] ) // 树i 的结点少
   {  parent[i] = j; parent[j] = temp;  }
  else        // 树j 的结点少或与树i的同样多
   {  parent[j] = i; parent[i] = temp;    }
}
```

搜索——查找

- 英汉字典中查找某个英文单词的中文解释；
- 新华字典中查找某个汉字的读音、含义；
- 对数表、平方根表中查找某个数的对数、平方根；
- 邮递员送信件要按收件人的地址确定位置等等。
- 可以说查找是为了得到某个信息而常常进行的工作。

- 计算机、计算机网络使信息查询更快捷、方便、准确。
- 要从计算机、计算机网络中查找特定的信息，就需要在计算机中存储包含该特定信息的表。
- 查找是许多程序中最消耗时间的一部分。因而，一个好的查找方法会大大提高运行速度。
- 由于计算机的特性，象对数、平方根等是通过函数求解，无需存储相应的信息表。

查找的基本概念

(1)查找表的定义

（2）查找表可分为两类：

静态查找表

动态查找表

（3）对查找表经常进行的操作

- 1）查询某个“特定的”数据元素是否在查找表中；
- 2）检索某个“特定的”数据元素的各种属性；
- 3）在查找表中插入一个数据元素；
- 4）从查找表中删去某个数据元素。

关键字　主关键字　次关键字

查找

查找的结果　“查找成功”　“查找不成功”

（4）如何进行查找？

查找的方法取决于查找表的结构。

- （5）衡量查找算法的时间效率标准是：
- 1. 平均比较次数　平均查找长度

ASL_{succ}=(每个元素的比较次数之和)/总的元素个数

- 2.另外衡量一个查找算法还要考虑算法所需要的存储量和算法的复杂性等问题。

（6）常见的查找方法

- 顺序查找
- 二分查找
- 二叉排序树（树表查找）
- 索引查找
- 散列查找
- B-树

1.顺序查找

- 查找过程
- 适合于顺序查找的存储结构：顺序存储、链式存储
- 查找效率：ASL=n*(n+1)/2 (成功时)
  ASL=n （不成功时）

存储结构——顺序存储

顺序存储的结构类型定义如下：

```
struct Rec
{ Key key;
   ……;
};
Rec S[n+1];
```

注意：元素分别存放在S[1]至 S[n]。

实现算法：（采用顺序存储结构）

```
int  Search(Rec S[ ],Key x，  int n)
{
  int i = 1; //从头开始查找
  while ( S[i].key != x && i<=n)
      i++ ;  //往后查找
  if (i<=n) return i;
  else return 0;
}
```

该算法有何缺陷？

改进算法1：

```
int Search (Rec S[ ] ,Key x ,int n)
{//顺序查找关键码为x的数据对象，
 //使用第n+1号位置作为控制搜索自动结束的“监视哨”使用
  S[n+1].key = x; //将x送n号位置设置监视哨
  int i =1;
  while ( S[i].key != x )   i++ ;
    //从前向后顺序搜索
  if(i==n+1) return 0; else  return i;
}
```

改进算法2：

```
int Search (Rec S[ ] ,Key x ,int n)
{//顺序查找关键码为x的数据对象，
 //使用第0号位置作为控制搜索自动结束的“监视哨”使用
  S[0].key = x; //将x送0号位置设置监视哨
  int i =n;
  while ( S[i].key != x )   i-- ;
    //从后向前顺序搜索
  if(i==0) return 0; else  return i;
}
```

算法分析

- 时间复杂度

基于有序顺序表的顺序查找算法

若查找表是一个有序表，并采用顺序存储的方法进行存储，则查找过程如何？

例,对有序顺序表(10, 20, 30, 40, 50, 60)进行顺序查找。

0	
1	10
2	20
3	30
4	40
5	50
6	60
.	
n	

基于有序顺序表的顺序查找算法

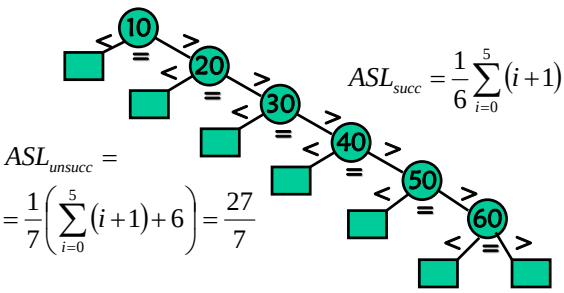
```
Int Search(Rec S[ ],Key x ,int n)
{ //顺序搜索关键码为x的数据对象

  for ( int i = 1; i <= n; i++ )
    if(S[i].key == x)
              return i; //成功
      else
         if ( S[i].key > x ) break;
  return 0; //顺序搜索失败，返回失败信息
}
```

该算法有何缺陷？如何改进？

0	
1	10
2	20
3	30
4	40
5	50
6	60
.	
n	

过程分析

- 查找算法分析工具：判定树



$$ASL_{succ} = \frac{1}{6}\sum_{i=0}^{5}(i+1) = \frac{7}{2}$$

$$ASL_{unsucc} = \frac{1}{7}\left(\sum_{i=0}^{5}(i+1)+6\right) = \frac{27}{7}$$

改进方案

- 思路：搜索范围？

对有序顺序表(10, 20, 30, 40, 50, 60)进行查找。

基于有序顺序表的二分查找算法

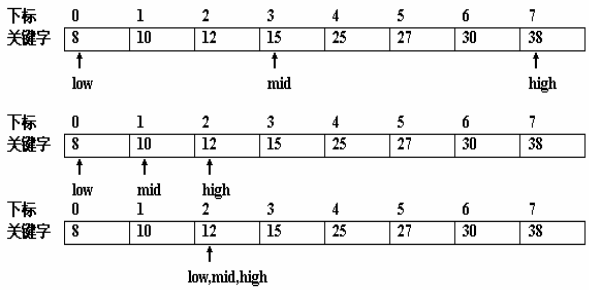
二分查找，也称折半查找

1.算法要求：有序且用顺序存储方法存储

2.查找方法：
先确定待查记录所在的范围（区间），然后逐步缩小范围直到找到或找不到该记录为止。

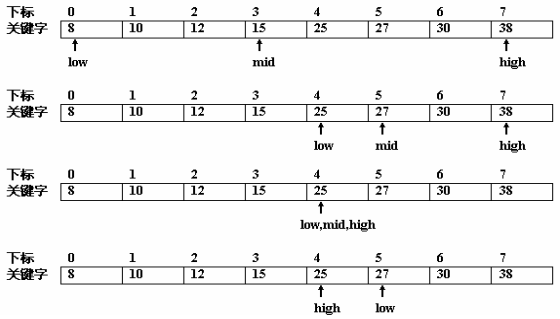
• 二分查找算法的运行实例

• 假设数组 S 的内容为{8，10，12，15，25，27，30，38},则查找元素12的过程为：



• 二分查找算法的运行实例

- 假设数组 S 的内容为{8，10，12，15，25，27，30，38},则查找元素26的过程为：



二分查找算法的实现（非递归算法）

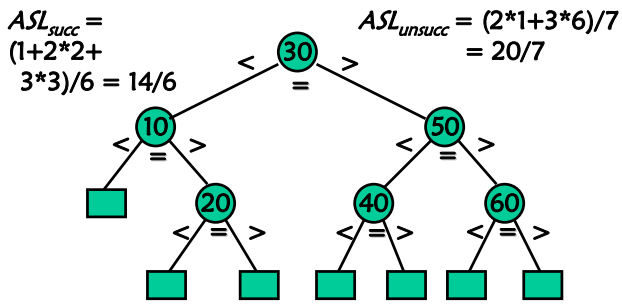
```
int Search_Bin(Rec S[ ]，Key x,int n)
{//二分查找的非递归算法 S[ ]为有序表
   low=1;high=n;
   while(low<=high)
    {
      mid=(low+high)/2;
      if(x==S[mid].key) return mid;
       else if(x<S[mid].key) high=mid -1;
           else low=mid +1 ;
    }
    return 0;
} //Search_Bin;
```

二分查找算法的实现（递归算法）

```
int Search_Bin(Rec S[],Key x,int low,int high)
{//在主函数main中调用该函数时实参为Search_bin(S,x,1,n)
   if (low>high) return 0;//查找不成功
   else
    {
      mid=(low+high)/2;
      if(x==S[mid].key) return mid;
       else if(x<S[mid].key)
              return Search_bin(S,x,low ,mid-1);
          else return Search_bin(S,x,mid+1,high);
   }
}//Search_Bin;
```

算法时间复杂度的分析

- 工具：二分查找判定树
- 例如，（10, 20, 30, 40, 50, 60 ）对应的判定树为：



结论

从上面的二分查找判定树可知：一般情况下，表长为 n 的二分查找的判定树的深度和含有 n 个结点的完全二叉树的深度相同。

假设 $n=2^h-1$ 并且查找概率相等，则有：

$$ASL_{bs}=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\left[\sum_{j=1}^{h}j\times 2^{j-1}\right]=\frac{n+1}{n}\log_2(n+1)-1$$

在 n>50 时，可得近似结果

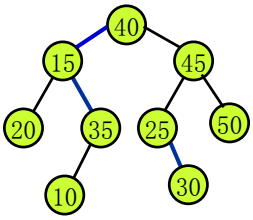
$$ASL_{bs}\approx\log_2(n+1)-1$$

二分查找的要求

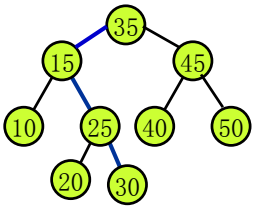
- （1）顺序存储
- （2）数据有序

如何改进？ 分类？

树形查找



树1



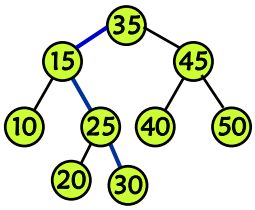
树2

二叉排序树（二叉搜索树或二叉检索树）

定义：

二叉搜索树或者是一棵空树，或者是具有下列性质的二叉树：

- 每个结点都有一个作为搜索依据的关键码(key)，所有结点的关键码互不相同。
- 左子树(如果存在)上所有结点的关键码都小于根结点的关键码。
- 右子树(如果存在)上所有结点的关键码都大于根结点的关键码。
- 左子树和右子树也是二叉搜索树。



- 二叉排序树→中序遍历
- 注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键码不一定小于路径上的结点的关键码。

二叉排序树的存储结构

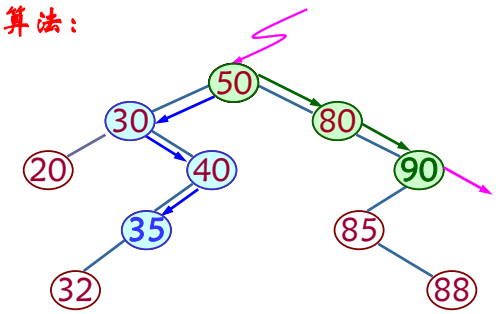
由于二叉排序树也是二叉树，因此，通常取二叉链表作为二叉排序树的存储结构。

```
template <class T>
struct BTreeNode // 结点结构
{
  T data;
  BTreeNode  *lchild, *rchild;// 左右孩子指针
};
```

二叉排序树类的定义

```
template <class T>
class BSTree :
{   BTreeNode<T>  *root;

BTreeNode<T> *searchBST ( T x, BTreeNode<T> *p); //递归查找函数
int  insert(T x, BTreeNode<T> *&bst); //递归插入函数

 public:
  BSTree(BTreeNode<T> *p=0) {  root=p; } // 构造函数
  BTreeNode<T> *searchBST( T x); //查找函数,调用递归查找,供main函数调用
  BTreeNode<T> *searchBST1( T x ); //非递归查找函数
  T min( ); // 获取最小值
  T max( );  // 获取最大值
  int insert(T x);        //插入函数,其调用递归插入，供main函数调用
  int insert1(T x);       //非递归插入函数
  void CreateBST(void); //建立二叉排序树
};
```

1.查找算法：



查找关键字

== 50 , 35 , 90 , 95 ,

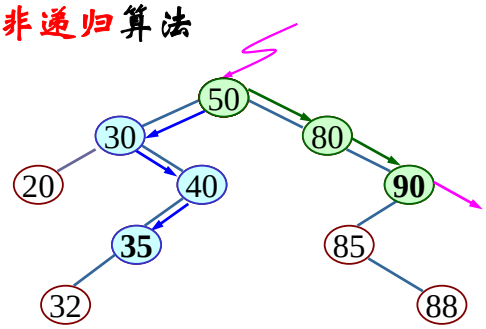
算法描述如下（递归算法）：

```
BTreeNode<T> * BSTree<T>:: SearchBST(T x,BTreeNode<T> *p)
{
 if((!p)||(x==p->data))    return p;
  else if(x<p->data)
        return SearchBST (x,p->lchild);
                    // 在左子树中继续查找
   else
         return SearchBST (x,p->rchild);
                    // 在右子树中继续查找
}//SearchBST
```

递归查找函数对应的公有函数

```
BTreeNode<T> *BSTree<T>::searchBST(T x)
//查找函数,调用递归查找,供main函数调用
{
  return searchBST(x， root);
}
```

查找非递归算法



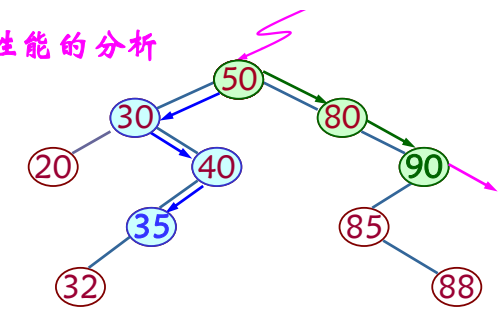
查找关键字

== 50，35，90，95，

算法描述如下（非递归算法）：

```
BTreeNode<T> * BSTree<T>:: SearchBST1(T x)
{ BTreeNode<T> *TT=root;
  while (TT!=0)
  {
    if(x==TT->data) return (TT);
      else if(x<TT->data)
        TT=TT->lchild;//在左子树中找
      else
        TT=TT->rchild;//在右子树中找
  }
  return 0;
}//SearchBST1
```

查找性能的分析



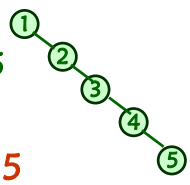
查找关键字 == 50 ,35 , 90 , 95

查找效率与什么因素有关？

与所建立的二叉排序树树形有关。

例如：

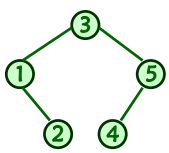
由关键字序列 1，2，3，4，5 构造而得的二叉排序树，



ASL = (1+2+3+4+5) / 5

= 3

由关键字序列 3，1，2，5，4 构造而得的二叉排序树



ASL = (1+2+3+2+3) / 5

= 2.2

下面讨论理想的情况:

在理想的情况下,即要求ASL是最小的.为了使ASL最小,该二叉排序树必与同样结点数的完全二叉树树高一样,即

$$h=\log^n$$

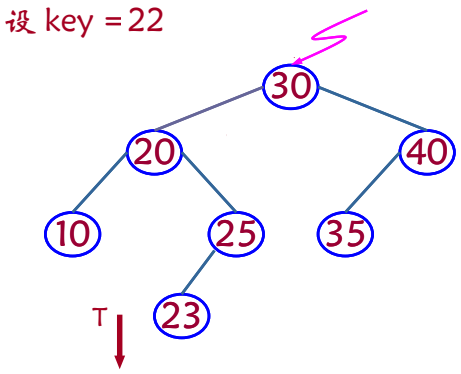
此时,

$$ASL= \log^n$$

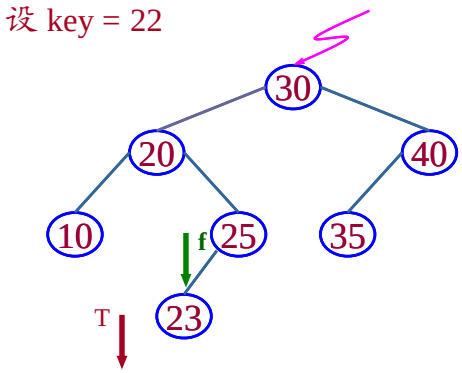
我们也可以进一步证明得到,在平均情况下,二叉排序树的ASL为:

$$ASL= \log^n$$

2.插入算法



2.插入算法



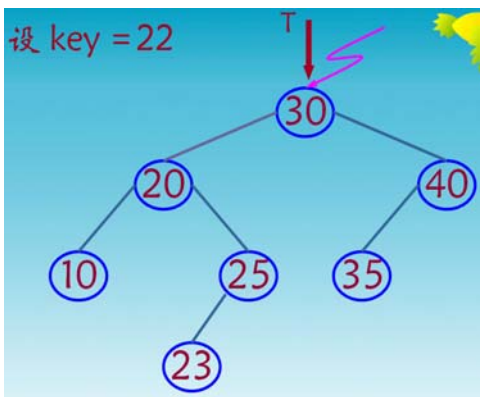
结 论

- 判断存在与否?
- 搜索成功: 不插入
- 搜索不成功
- 因此,为了能进行有效的插入,我们有必要找到插入点的双亲结点,方法是改写查找算法.
- 在查找过程中增加一个指向双亲的指针.

算法描述如下(非递归算法):

```
int BSTree<T>:: Insert1(T x)
{ BTreeNode <T> *TT,*f,*s;
  TT=root; f=0;
  while (TT!=0)
  { if( x==TT->data) return -1; //重复值,插入不成功
      else if( x<TT->data)
         { f=TT; TT=TT->lchild; } //在左子树中找
      else
         { f=TT; TT=TT->rchild; } //在右子树中找
  }
  p= new BTreeNode<T>;  p->data=x;  // 新结点生成
  p->lchild=p->rchild=0;
  if ( f== 0) root=p;  //根结点
    else if ( x>f->data ) f->rchild=p; //作为右子树
      else f->lchild=p; //作为左子树
  return 1;  //插入成功
}//Insert1
```

插入递归算法



算法描述如下（递归算法）：

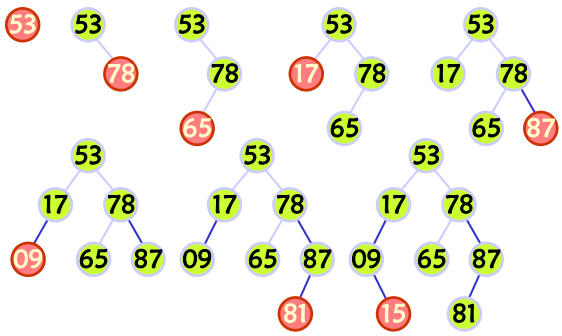
```
int BSTree<T>:: insert( T x, BTreeNode<T> *&bst)
{ BTreeNode<T> *p;
  if (bst==0)
  {
   p= new BTreeNode<T>;  p->data=x;
   p->lchild=p->rchild=NULL;
   bst=p;
   return 1; //插入成功
  }
  if( x<bst->data)
      return Insert( x,bst->lchild); //在左子树中插入
  else if( x>bst->data)
      return Insert( x,bst->rchild); //在右子树中插入
   else  return -1;//重复值,插入不成功
}//Insert
```

调用插入递归函数——插入操作公有函数

```
int BSTree<T>::insert( T x )
{
    int ret;

    ret=insert( x, root );

    if (ret==-1)   cout<<"HAVE FOUND ";
    else  cout<<" insert  ok";

}
```

3.二叉排序树建立算法

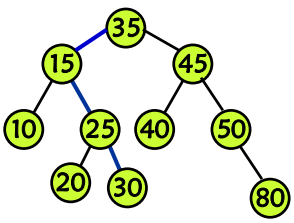
输入数据{ 53, 78, 65, 17, 87, 09, 81, 15 }



建立算法描述如下：

```
Void   BSTree<T>:: CreateBST(void)
{
   root = 0;
   cin >> x;
   while ( x != -1)
   {
     insert ( x );
     cin >> x;
   }
}//CreateBST
```

4、二叉排序树的删除



和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。

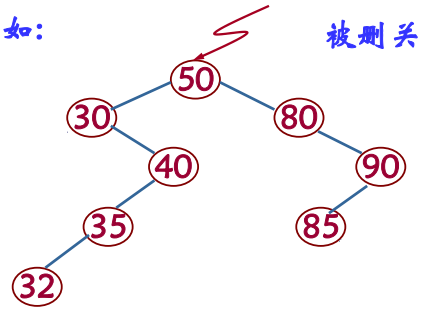
删除操作可分三种情况讨论：

- （1）被删除的结点是叶子；
- （2）被删除的结点只有左子树或者只有右子树；
- （3）被删除的结点既有左子树，也有右子树。

(1) 被删除的结点是叶子结点

例如:

被删关键字 = 88

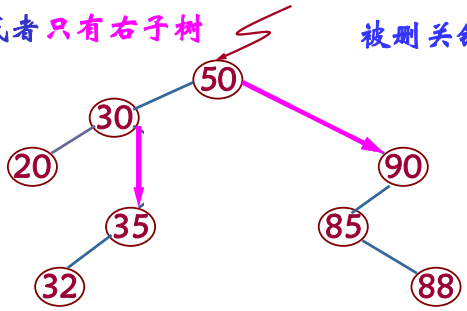


其双亲结点中相应指针域的值改为“空”

(2) 被删除的结点只有左子树

或者只有右子树

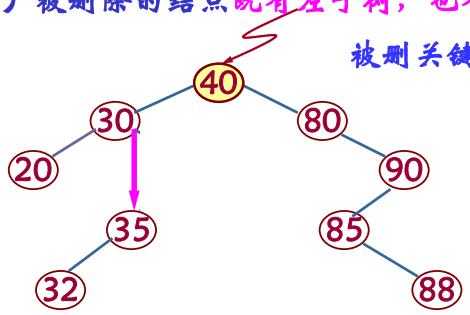
被删关键字 = 80

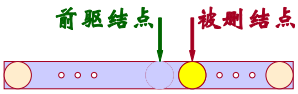


其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

(3) 被删除的结点既有左子树，也有右子树

被删关键字 = 50



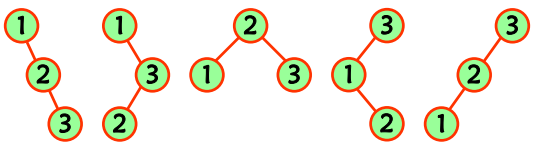


以其前驱替代之，然后再删除该前驱结点

5、讨论

1. 集合{1, 2, 3}所生成的二叉排序树树形有多少种？

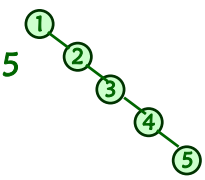
{1, 2, 3} {1, 3, 2} {2, 1, 3} {2, 3, 1} {3, 1, 2} {3, 2, 1}



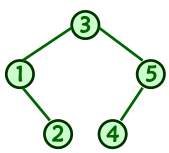
5、讨论

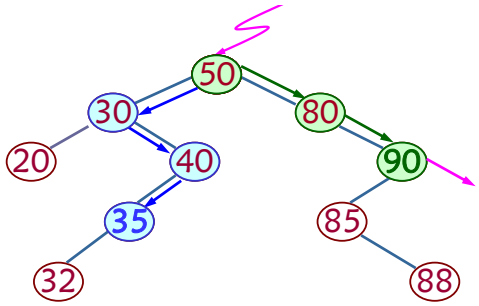
2. 当序列中出现重复元素时，如何构造二叉树排序？其查找效率如何？

由关键字序列 1，2，3，4，5



由关键字序列 3，1，2，5，4 构造而得的二叉排序树





查找关键字 == 50 ,35 , 90 , 95

查找效率与什么因素有关？

与所建立的二叉排序树树形有关。

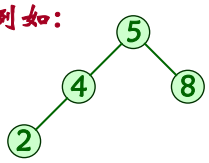
解决方案

平衡二叉树

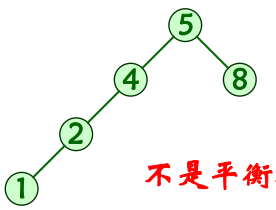
二叉平衡树是二叉查找树的另一种形式，其特点为：

二叉查找树中每个结点的左、右子树深度之差的绝对值不大于1，即 $|h_L - h_R| \le 1$。

例如：



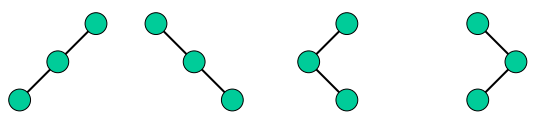
是平衡树



不是平衡树

结点的平衡因子

- 平衡因子.
- AVL树任一结点平衡因子只能取 -1, 0, 1
- 如果一个结点的平衡因子的绝对值大于1，则这棵二叉搜索树就失去了平衡，不再是AVL树。
- 如果一棵二叉搜索树是高度平衡的，且有 n 个结点，其高度可保持在 $O(\log_2 n)$，平均搜索长度也可保持在 $O(\log_2 n)$。

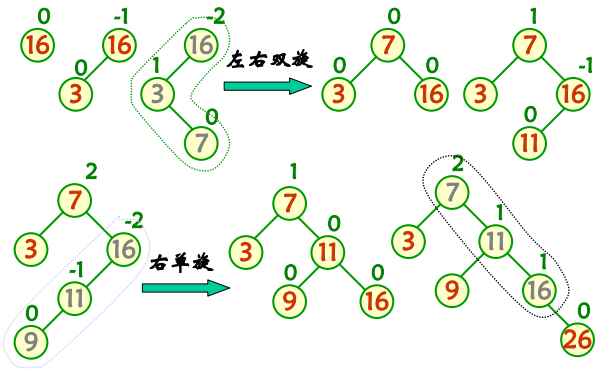


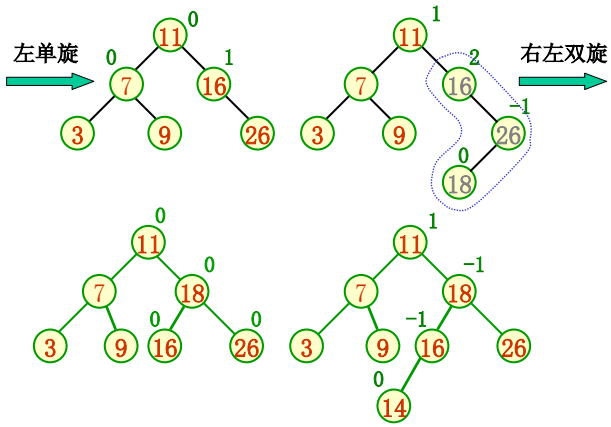
右单旋转 LL型　左单旋转 RR型　左右双旋转 LR型　右左双旋转 RL型

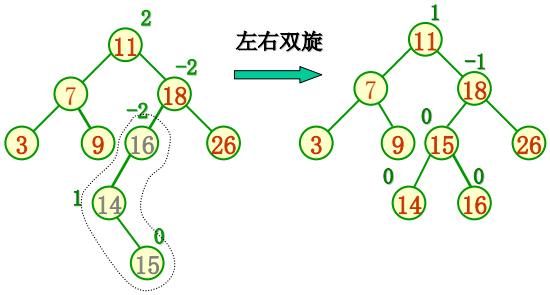
AVL树的建立

- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 |balance| > 1，则出现了不平衡，需要做平衡化处理。
- 因此，操作应从一棵空树开始，通过输入一系列对象关键码，逐步建立AVL树。在插入新结点时使用平衡旋转方法进行平衡化处理。

例，输入关键码序列为{16, 3, 7, 11, 9, 26, 18, 14, 15}，则插入和调整的过程如下：







新疑问

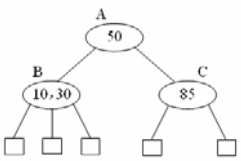
是否有其他方法也可以保持查找树在高度上的绝对平衡，这样就可以让同样结点个数的情况下，其对应的树高会相对最小，从而让其取得较好的查找性能。

思路：

（1）反向增长？

（2）绝对平衡意味着树结构的对称

2-3树



定义：一棵2-3树是一棵查找树，该树或者为空，或者满足下列性质：

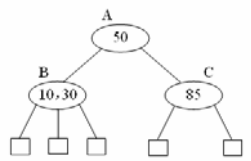
（1）每个内部结点是2-结点或3-结点。

（2）令LeftChild和MiddleChild为2-结点的子女，dataL为该结点中的元素。LeftChild子2-3树中的所有关键字小于dataL.key，MiddleChild子2-3树中的所有关键字大于dataL.key。

（3）令LeftChild，MiddleChild和RightChild为3-结点的子女，dataL和dataR为该结点中的两个元素，且dataL.key < dataR.key。LeftChild子2-3树中的所有关键字小于dataL.key；MiddleChild子2-3树

中的所有关键字大于dataL.key且小于dataR.key；RightChild子2-3树中的所有关键字大于dataR.key。

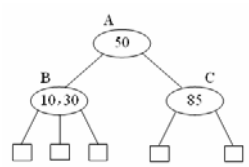
（4） 所有外部结点都处于同一个层次。

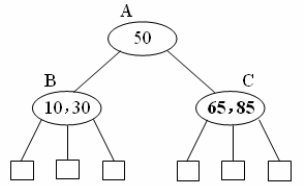


2-3树的插入操作

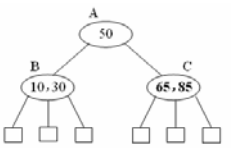
下面通过实例说明2-3树的插入过程。

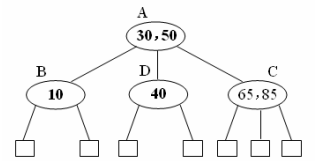
首先将65插入左图的2-3树。



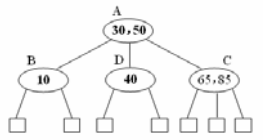


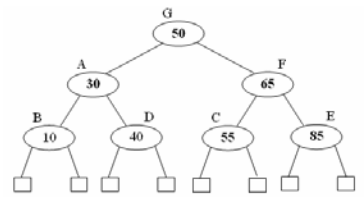
接着插入40。





最后插入55。



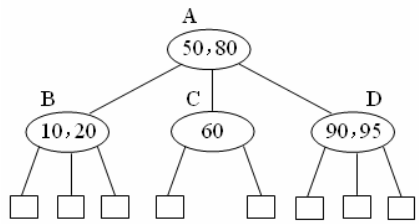


2-3树的删除操作

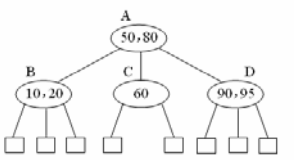
如果需删除的元素不在叶结点中，可用被删除元素左边子树中的最大元素或右边子树的最小元素取代需删除的元素，从而将问题转化为从叶结点中删除元素。

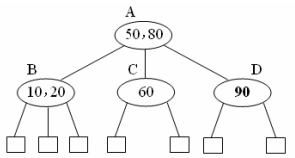
下面将只考虑从叶结点删除元素的情况。

假设从下图开始：

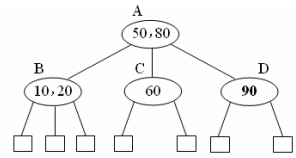


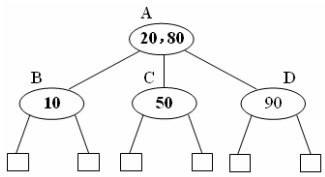
首先删除95。



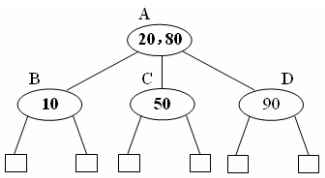


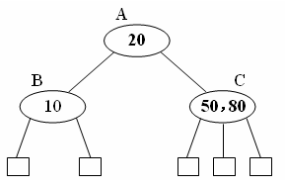
接着删除60。



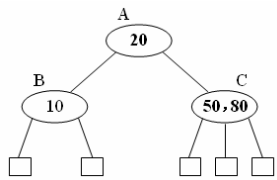


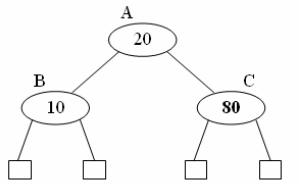
再删除90。



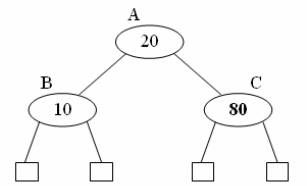


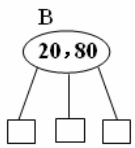
接着删除50。





最后删除10。





时间复杂度的分析

设2-3树的高度为h（这时，外部结点一定在h + 1层）。

在最稀疏情况下，所有内部结点都是2-结点，结点总数为$2^h - 1$，元素个数也为$2^h - 1$。

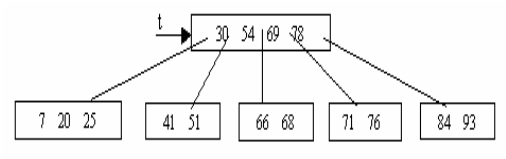
在最密集情况下，所有内部结点都是3-结点，结点总数为$(3^h - 1)/2$，元素个数则为$3^h - 1$。

所以，2-3树的元素个数在$2^h - 1$和$3^h - 1$之间，具有n个元素的2-3树的高度在$\lceil \log_3(n + 1) \rceil$和$\lceil \log_2(n + 1) \rceil$之间。

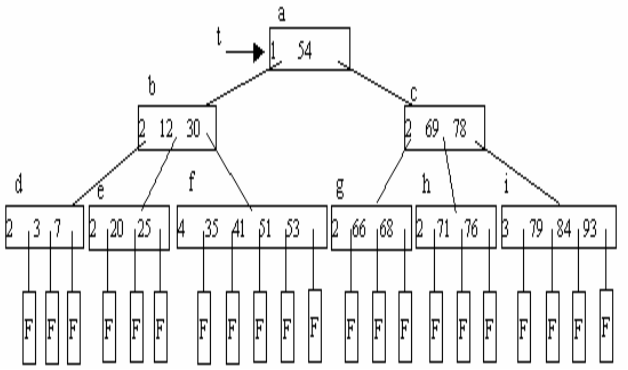
B-树

- B-树是一种平衡的多路查找树，它在文件系统中很有用。

例如，下图是一棵5阶的B-树，其深度为3。



例如，下图是一棵5阶的B-树，其深度为4。



B-树

- 定义：
- 一棵m阶的B-树，或者为空树，或为满足下列特性的m叉树：
- (1)树中每个结点至多有m棵子树；
- (2)若根结点不是叶子结点，则至少有两棵子树；
- (3)除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树；

- (4)所有的非终端结点中包含以下信息数据：
  （n，A0，K1，A1，K2，…，Kn，An）
- 其中：Ki（i=1,2,…,n）为关键码，且Ki<Ki+1，Ai为指向子树根结点的指针(i=0,1,…,n)，且指针Ai-1所指子树中所有结点的关键码均小于Ki(i=1,2,…,n)，An所指子树中所有结点的关键码均大于Kn，⌈m/2⌉-1≤n≤m -1，n为关键码的个数。
- (5)所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

B-树的查找

B-树的查找类似二叉排序树的查找，所不同的是B-树每个结点上是多关键码的有序表，在到达某个结点时，

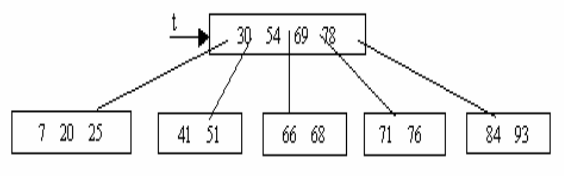
先在有序表中查找，若找到，则查找成功；

否则，到按照对应的指针信息指向的子树中去查找，

当到达叶子结点时，则说明树中没有对应的关键码，查找失败。

B-树的插入

- 在B-树上插入关键码与在二叉排序树上插入结点不同，关键码的插入不是在叶结点上进行的，而是在最底层的某个非终端结点中添加一个关键码，**若该结点上关键码个数不超过m-1个，则可直接插入到该结点上**；
- 否则，**该结点上关键码个数至少达到m个**，因而使该结点的子树超过了m棵，这与B-树定义不符。所以要进行调整，即结点的“**分裂**”。
- **方法为：关键码加入结点后，将结点中的关键码分成三部分，使得前后两部分关键码个数均大于等于⌈m/2⌉-1，而中间部分只有一个结点**。前后两部分成为两个结点，中间的一个结点将其插入到父结点中。若插入父结点而使父结点中关键码个数超过m-1，则父结点继续分裂，直到插入某个父结点，其关键码个数小于m。可见，B-树是从底向上生长的。

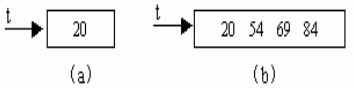


例如，以下列关键码，建立5阶B-树。

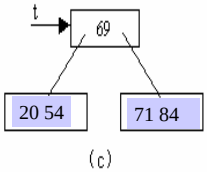
20, 54, 69, 84, 71, 30, 78, 25, 93, 41, 7, 76, 51, 66, 68, 53, 3, 79, 35, 12, 15, 65

(1)向空树中插入20，得图(a)。

(2)插入54，69，84，得图(b)。

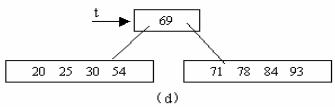


(3)插入71，索引项达到5，要分裂成三部分：{20，54}，{69}和{71，84},并将69上升到该结点的父结点中，如图(c)。

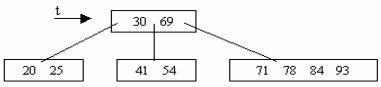


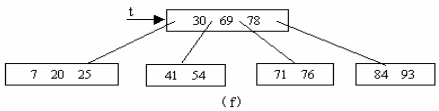
20, 54, 69, 84, 71, 30, 78, 25, 93, 41, 7, 76, 51, 66, 68, 53, 3, 79, 35, 12, 15, 65

(4)插入 30,78,25,93 得图(d)。



(5)插 41 又分裂得图(e)。



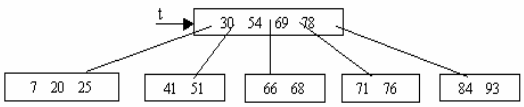


(6)7 直接插入。

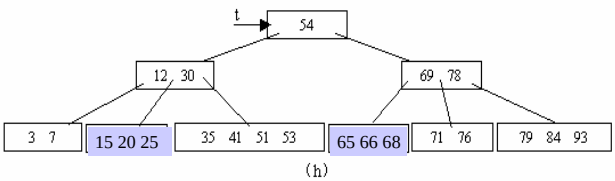
(7)76 插入,分裂得图(f)。

20, 54, 69, 84, 71, 30, 78, 25, 93, 41, 7, 76, 51, 66, 68, 53, 3, 79, 35, 12, 15, 65

(8)51，66直接插入,当插入68，需分裂，得图(g)。



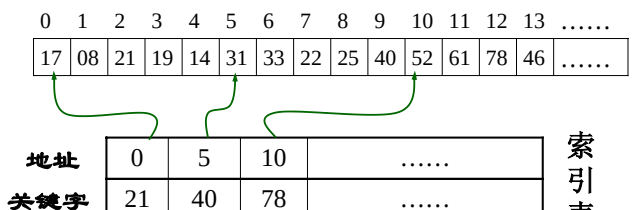
(9)53，3，79，35直接插入,12插入时，需分裂，但中间关键码12插入父结点时，又需要分裂，则54上升为新根。15，65直接插入得图(h)。



索引顺序表

在建立顺序表的同时，建立一个索引。

例如：　　顺序表



索引顺序表 = 索引表 + 顺序表

索引顺序表的存储结构

索引表的结构：

```
struct indexItem{// 索引项
    Key maxkey;
    int  stadr;
}IndexTable[MaxSize];    //索引表
```

顺序表的结构：

```
struct Rec{
  Key key;
  …
}Table[n]; //顺序表
```

索引顺序表的查找过程

1）在索引表确定记录所在区间(可用顺序查找或二分查找)；

2）在顺序表的某个区间内进行查找（用顺序查找）。

可见，索引顺序查找的过程也是一个“缩小区间”的查找过程。

注意： 索引可以根据查找表的特点来构造。

索引顺序查找的平均查找长度 = 查找“索引”的平均查找长度

+ 查找“顺序表”的平均查找长度

请计算：

如果顺序表元素个数为n，索引表用顺序查找,在顺序表的某个区间也是用顺序查找，则索引表为多大使查找速度最快？

散列表（哈希表）--HASH

一、哈希表的定义？

二、哈希函数的构造方法

三、处理冲突的方法

四、哈希表的查找

一、哈希表的定义？

- 前面所讨论的查找方法，由于数据元素的存储位置与关键码之间不存在确定的关系，
- 因此，查找时，需要进行一系列对关键码的查找比较，即“查找算法”是建立在比较的基础上的，查找效率由比较一次缩小的查找范围决定。
- 理想的情况是依据关键码直接得到其对应的数据元素位置，即要求关键码与数据元素间存在一一对应关系，通过这个关系，能很快地由关键码得到对应的数据元素位置。

【例】11个元素的关键码分别为 18，27，1，20，22，6，10，13，41，15，25。 如何存储才能使查找效率提高?

f（key）=key % 11

0	1	2	3	4	5	6	7	8	9	10
22	1	13	25	15	27	6	18	41	20	10

如何查找?

有何缺陷?

哈希表(杂凑表)

哈希方法(杂凑法)

哈希函数(杂凑函数)

冲突(Collision) 或 同义词。

- 哈希方法需要解决以下两个问题：
- 1. 构造好的哈希函数
- （1）所选函数尽可能简单，以便提高转换速度。
- （2）所选函数对关键码计算出的地址，应在哈希地址集中大致均匀分布，以减少空间浪费。
- 2. 制定解决冲突的方案。

二、哈希函数的构造方法

1. 直接定址法

Hash(key)=a·key+b　　(a、b为常数)

【例】关键码集合为{ 100，300，500，700，800，900 }，

Hash(key)=key/100

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

- 2. 除留余数法

Hash(key)=key % p　　(p是一个整数)

选取合适的p很重要

若哈希表表长为m，则要求p≤m，且接近m或等于m。

p一般选取质数，也可以是不包含小于20质因子的合数。

为什么要对 p 加限制？

例如：给定一组关键字为：12，39，18，24，33，21，若取 p=9，则他们对应的哈希函数值将为：3，3，0，6，6，3

可见，若 p 中含质因子 3，则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

3. 数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成（u_1，u_2，…，u_s），分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。[即设关键码集合中，每个关键码均由m位组成，每位上可能有r种不同的符号。

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。

- 【例】若关键码是4位十进制数，则每位上可能有十个不同的数符0～9，所以r=10。
- 【例】若关键码是仅由英文字母组成的字符串，不考虑大小写，则每位上可能有26种不同的字母，所以r=26。

【例】有一组关键码如下：

①	②	③	④	⑤	⑥	⑦
3	4	7	0	5	2	4
3	4	9	1	4	8	7
3	4	8	2	6	9	6
3	4	8	5	2	7	0
3	4	8	6	3	0	5
3	4	9	8	0	5	8
3	4	7	9	6	7	1
3	4	7	3	9	1	9

第1、2位均是“3和4”，第3位也只有“7、8、9”，因此，这几位不能用，余下四位分布较均匀，可作为哈希地址选用。若哈希地址是两位，则可取这四位中的任意两位组合成哈希地址，也可以取其中两位与其它两位叠加求和后，取低两位作哈希地址。

4. 折叠法

此方法将关键码自左到右分成位数相等的几部分，最后一部分位数可以短些，然后将这几部分叠加求和，并按哈希表表长，取后几位作为哈希地址。这种方法称为折叠法。

有两种叠加方法：

1）移 位 法 —— 将各部分的最后一位对齐相加。

2）间界叠加法 —— 从一端向另一端沿各部分分界来回折叠后，最后一位对齐相加。

此方法适合于：

关键字的数字位数特别多。

【例】关键码为 key=05326248725，设哈希表长为三位数，则可对关键码每三位一部分来分割。

关键码分割为如下四组： 253 463 587 05

用上述方法计算哈希地址：

对于位数很多的关键码，且每一位上符号分布较均匀时，可采用此方法求得哈希地址。

```
  253
  463
  587
+  05
------
 1308
```

Hash(key)=308

移位法

```
  253 ]
[ 364 ]
  587 ]
+  50
------
 1254
```

Hash(key)=254

间界叠加法

5. 平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值” 的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于：

关键字中的每一位都有某些数字重复出现频度很高的现象。

6.随机数法

设定哈希函数为：

H(key) = Random(key)

其中，Random 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

结 论：

实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是使产生冲突的可能性降到尽可能地小。

三、处理冲突的方法

“处理冲突” 的实际含义是:

为产生冲突的地址寻找下一个哈希地址。

1. 开放定址法（闭散列表法）

2. 链地址法（开散列表法）

1. 开放定址法

为产生冲突的地址 H(key) 求得一个地址序列： $H_0, H_1, H_2, \ldots, H_s$ $1 \leqslant s \leqslant m-1$

其中： $H_0 = H(key)$

$$H_i = (H(key) + d_i) \% m \ , \ i=1, 2, \ldots, s$$

对增量 di 有三种取法：

1) 线性探测再散列

$di = c \times i$ 最简单的情况 $c=1$

2) 平方探测再散列

$di = 1^2, -1^2, 2^2, -2^2, \ldots,$

3) 随机探测再散列(双散列函数探测)

d_i 是一组伪随机数列 或者

$d_i = i \times H_2(key)$

例如: 关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

H(key) = key % 11

若采用线性探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		

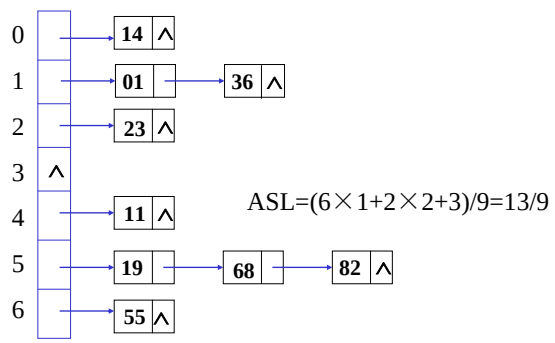
若采用二次探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	11	82	68	36	19		

2. 链地址法

将所有哈希地址相同的记录都链接在同一链表中。

例 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }　H(key)=key % 7



ASL=(6×1+2×2+3)/9=13/9

算法的实现

- 存储结构
- 链表的插入
- 头插入

存储结构

```
#define MaxSize 100
struct node
{
    LinkNode *head;
} HT[MaxSize];
```

算法的实现——插入

```
LinkNode *p;
int x;
cin>>x;
key=H(x); // 求出函数值
p=new LinkNode;
p->data=x;
p->link=HT[key]  // 头插入
HT[key]=p;
```

算法的实现——查找

```
LinkNode *p;
int x;
cin>>x;
key=H(x); // 求出函数值
p=HT[key] ;
while(p)
{
  if (p->data==x) return 1; // 表示查找成功
}
return -1; //表示查找失败
```

四、哈希表的查找

查找过程和造表过程一致。假设采用开放定址处理冲突，则查找过程为：

对于给定值 K, 计算哈希地址 i = H(K)

若 r[i] = NULL 则查找不成功

若 r[i].key = K 则查找成功

否则 “求下一地址 Hi” ，直至

r[Hi] = NULL (查找不成功)

或 r[Hi].key = K (查找成功) 为止。