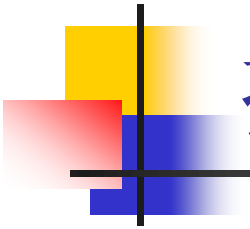


搜索/查找 (Search)

- Search基本概念
- Search算法
  - 静态搜索表
  - 二叉搜索树
  - AVL树
- B树
- B+树
- Hashing
- Search算法的分析

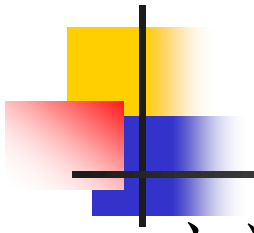
搜索(Search)的概念

- 搜索
  - 在数据集合中寻找满足某种条件的数据对象
- 搜索的结果成功
  - 找到满足条件的数据对象。
  - 作为结果，可给出该对象在一些信息
- 失败
  - 作为结果，应给出一些信息，如失败标志等

搜索结构

- 指用于搜索的数据集合
- 是由同一数据类型的对象(或记录)组成

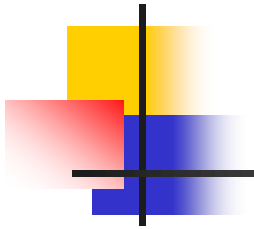
- 在每个对象中有若干属性，其中有一个属性，其值可唯一地标识这个对象，称为关键码
- 使用基于关键码的搜索，搜索结果唯一
- 在实际应用时，搜索条件是多方面的，可以使用基于属性的搜索方法，但搜索结果可能不唯一



- 实施搜索时，有两种不同的环境
  - 静态环境
    - 搜索结构在插入和删除等操作的前后不发生改变
    - 静态搜索表
  - 动态环境
    - 为保持较高的搜索效率，搜索结构在执行插入和删除等操作的前后将自动进行调整，结构可能发生变化
    - 动态搜索表

静态搜索表

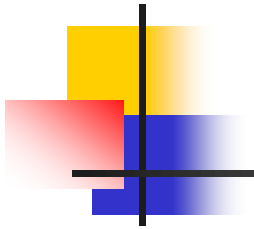
- 在静态搜索表中
  - 数据元素存放于数组中
  - 利用数组元素的下标作为数据元素的存放地址
  - 搜索算法
    - 根据给定值 k，在数组中进行搜索，直到找到 k 在数组中的存放位置或可确定在数组中找不到 k 为止



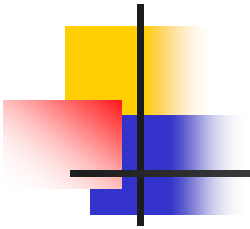
数据表与搜索表的类定义

数据表与搜索表的类定义

```
#include <iostream.h>
#include <assert.h>
const int defaultSize = 100;
template <class E, class K>

class dataList;

template <class E, class K >
class dataNode
{                                    //数据表中结点类的定义
friend class dataList<E, K>;
                                     //声明其友元类为dataList
```



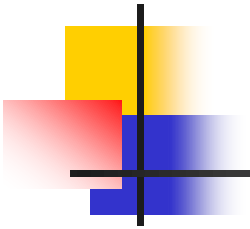
```
public:
    dataNode (const K x) : key(x) { }        //构造函数
     K getKey() const { return key; }        //读取关键码
     void setKey (K x) { key = x; }          //修改关键码
private:
     K key;                          //关键码域
     E other;
};
```



```
template <class E, class K >
class dataList {                        //数据表类定义
public:
    dataList (int sz = defaultSize) : ArraySize(sz), CuurentSize(0)
    {
        Element = new dataNode<E, K>[sz];
        assert (Element != NULL);
    }
    dataList (dataList<E, K>& R);   //复制构造函数
    virtual ～dataList() { delete []Element; }      //析构函数
    virtual int Length() { return CurrentSize; }  //求表的长度
```

```
virtual K getKey (int i) const
{   //提取第 i（1开始）元素值
        assert (i > 0 || i <= CurrentSize);
        return Element[i−1].key;
  }
   virtual void setKey (K x, int i)
 {   //修改第 i（1开始）元素值
        assert (i > 0 || i <= CurrentSize);
        Element[i−1].key = x;
  }
friend ostream& operator << (ostream& out, const
   dataList<E, K>& OutList); //输出
friend istream& operator >> (istream& in, dataList<E,
   K>& InList); //输入
```

```
 virtual int SeqSearch (const K x) const; //搜索
 virtual bool Insert (E& e1);                //插入
 virtual bool Remove (K x, E& e1);           //删除
protected:
    dataNode<E, K> *Element
                //数据表存储数组
    int ArraySize, CurrentSize;
                //数组最大长度和当前长度
};
```

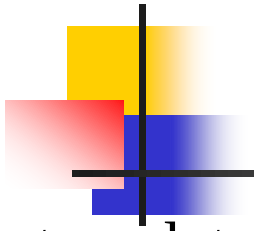


```
template <class E, class K >
bool dataList<E, K>::Insert (E& e1)
{
  //在dataList的尾部插入新元素, 若插入失败函数返
  //回false, 否则返回true.

    if (CurrentSize == ArraySize)
          return false;
    Element[CurrentSize] = e1;  //插入在尾端
    CurrentSize++;
    return true;
};
```

```
template <class E, class K>
bool dataList<E, K>::Remove (K x, E& e1)
{
  //在dataList中删除关键码为x的元素, 通过e1返回。
  //用尾元素填补被删除元素。
    if (CurrentSize == 0)
          return false;
    for (int i == 0; i < CurrentSize &&
          Element[i] != x; i++);              //在表中顺序寻找
     if (i == CurrentSize)
           return false;                      //未找到
     e1 = Element[i].other; //找到,保存被删元素的值
    Element[i] = Element[CurrentSize-1];    //填补
     CurrentSize--;
     return true;
};
```

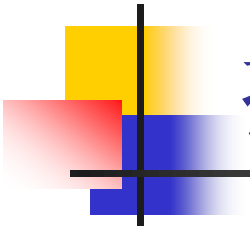
数据表类的友元函数

```
template <class E, class K>
ostream& operator << (ostream& out, const   dataList<E,K>&
    OutList)
{
    out << "存储数组大小"  << endl;

    if (i == CurrentSize) return false;          //未找到
    e1 = Element[i].other;          //找到,保存被删元素的值
    Element[i] = Element[CurrentSize-1];    //填补
    CurrentSize--;  return true;
};
```



```
template <class E, class K>
ostream& operator << (ostream& out, const   dataList<E,K>& OutList)
{
    out << "存储数组大小"  << endl;
    for (int i = 1; i <= OutList.CurrentSize; i++)
        out << OutList.Element[i-1] << ' ';
          out << endl;                  //输出表的所有表项到out
          out << "数组当前长度 : " <<
              OutList.CurrentSize << endl;
                                  //输出表的当前长度到out
          return out;
};
```

```
template <class E, class K>
istream& operator >> (istream& in, dataList<E, K>& InList)
 {
    cout << "输入存储数组当前长度 : ";
    in >> InList.CurrentSize;
                          //从in输入表的当前长度
    cout << "输入数组元素的值 : \n";
    for (int i = 1; i <= InList.CurrentSize; i++) {
                              //从in输入表的全部表项
        cout << "元素 "  << i << " : ";
        in >> InList.Element[i-1];
    }
    return in;
};
```



搜索/查找 (Search)

- Search基本概念
- Search算法
  - 静态搜索表
  - 二叉搜索树
  - AVL树
- Hash
- B树
- B+树
- Search算法的分析

顺序搜索（Sequential Search）

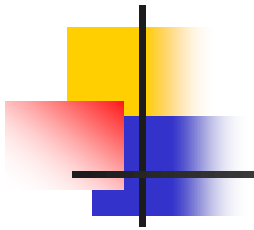
- 主要用于在线性表中搜索
  - 设表中有 CurrentSize 个元素，则顺序搜索从表的先端开始
  - 依序用各元素的关键码key与给定值 x 进行比较
    - 若找到与其值相等的元素，则搜索成功，给出该元素在表中的位置
    - 若整个表都已检测完，仍未找到与 x 相等的元素，则搜索失败，给出失败信息

使用“监视哨”的顺序搜索方法

- 设在数据表 dataList 中顺序搜索关键码key与给定值 x 相等的数据元素
- 要求：
  - 数据元素在表中从下标 0 开始存放
  - 下标为 CurrentSize 的元素作为控制搜索过程自动结束的“监视哨”使用
- 若搜索成功，则函数返回该元素在表中序号 Location（比下标大 1）
- 若搜索失败，则函数返回 CurrentSize+1

使用监视哨的顺序搜索算法

```
template <class E>
int dataList<E>::SeqSearch (const E x) const
{
    Element[CurrentSize].key = x;
    int i = 0;                                  //将X设置为监视哨
    while (Element[i].key != x) i++;
                                                //从前向后顺序搜索
    return i+1;
};
const int Size = 10;
```



```
main ()
{
    dataList<int> L1 (Size);    //定义int型搜索表L1
    int Target;  int Loc;
    cin >> L1;  cout << L1;     //输入L1
    cout << "Search for a integer : ";
    cin >> Target;                          //输入要搜索的数据
    if ( (Location = L1.Seqsearch(Target)) !=L1.Length() )
        cout << "找到待查元素位置在 : " << Loc+1 << endl;
                                            //搜索成功
    else cout << " 没有找到待查元素\n";
                                            //搜索不成功
};
```

顺序搜索的平均搜索长度

- 设数据表中有 n 个元素，搜索第 i 个元素的概率为 p_i
- 搜索到第 i 个元素所需比较次数为 c_i，则搜索成功的平均搜索长度:

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot c_i. \quad (\sum_{i=0}^{n-1} p_i = 1)$$

- 在顺序搜索并设置“监视哨”情形:

$$c_i = i + 1, \quad i = 0, 1, \ldots, n-1$$

- 因此: $$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot (i+1)$$

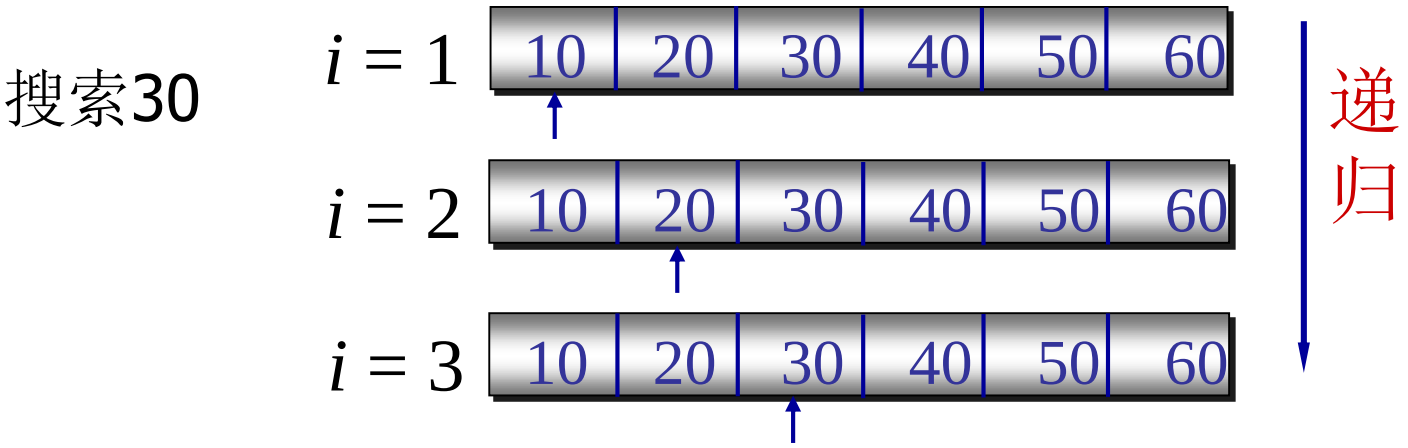
- 在表中各个元素搜索概率不同情形下，如果按搜索概率的高低排列表中的元素，从有序顺序表的情况可知，能够得到高的平均搜索长度
- 在等概率情形，$p_i = 1/n,\ i = 1, 2, \ldots, n$
  - 搜索成功的平均搜索程度为：

$$ASL_{succ} = \sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

  - 在搜索不成功情形，$ASL_{unsucc} = n+1$

顺序搜索的递归算法

- 采用递归方法，搜索值为 x 的元素，每递归一层就向待查元素逼近一个位置，直到到达该元素
- 假设待查元素在第 i（$1 \le i \le n$）个位置，则算法递归深度达 i（1～i）



顺序搜索的递归算法

```
template <class E>
int dataList<E>::
SeqSearch (const E x, int loc) const
 {
//在数据表 Element[1..n] 中搜索其关键码与给定值
//匹配的对象, 函数返回其表中位置。 参数 loc 是在
//表中开始搜索位置
    if (loc > CurrentSize) return 0;          //搜索失败
    else if (Element[loc−1].key == x) return loc;
                                              //搜索成功
    else return SeqSearch (x, loc+1);          //递归搜索
};
```