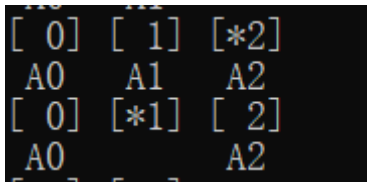
假设对于以长度为26的线性表：

用于存放26个大写字母开头加数字的元素，如A1，A2等，并且以此大写字母计算key（A记为0，B记为1，……），哈希函数为key%26，则可能出现如下情况：



原策略：在2号索引位置插入完A2后，删除A1，则A1处已经为空（原定策略将其置为空），按照线性探测法，在查找遇到空的元素时即会停止，记为找不到元素，按照置空策略，A2是不能再被正常访问到了，无论查找删除。

修改策略：约定一个key值用于表示已经删除，删除时将元素key值设为此值，并且保持此处为“被占据”的状态，即不为空。

```
bool hashDelete(string key)
{
    int ori = Hash(key);
    int cur = ori; //元素以pair<string,int>表示，first作key值，int作Element值
    while (hashTag[cur] && hashTable[cur].first != key) //hashTag[i]表示i处是否为空,1为非空,若不为所需key，继续寻找
    {
        cur = (cur + 1) % mod;
        if (cur == ori) return false;
    }
    hashTable[cur].first = ""; //删除

    printOut(cur); //打印操作，用于调试
    return true;
}
```

完整测试代码（插入 删除 查找）：

```
#include<iostream>
#include<algorithm>
#include<string>
#include<vector>
#include<utility>
#include<cstddef>
using namespace std;
constexpr int mod = 26; //用于测试循环
constexpr int Size = 26 + 4;
pair<string, int>* hashTable = new pair<string, int>[Size];
bool hashTag[Size];
void printOut(int index)
{
    for (int i = 0; i != 26; i++)
    {
        if (!hashTag[i]) continue;
        if (i != index)
        {
            if (index == -1 && hashTable[i].first == "")
```

```
                continue;
            printf("[ %2d] ", i);
        }
        else
            printf("[*%2d] ", i);

    }
    cout << endl;
    for (int i = 0; i != 26; i++)
    {
        if (!hashTag[i]) continue;
        if (index == -1 && hashTable[i].first == "")
            continue;
        printf(" %3s  ", hashTable[i].first.c_str());
    }
    cout << endl;
}
int Hash(string key)
{
    return (key[0] - 'A') % mod;
}
bool hashDelete(string key)
{
    int ori = Hash(key);
    int cur = ori;
    while (hashTag[cur] && hashTable[cur].first != key)
    {
        cur = (cur + 1) % mod;
        if (cur == ori) return false;
    }
    hashTable[cur].first = "";

    printOut(cur);
    return true;
}
bool hashInsert(string key, int tar)
{
    int ori = Hash(key);
    int cur = ori;
    while (hashTag[cur] && hashTable[cur].first != "")
    {
        if (hashTable[cur].first == key) return false; //key不可重复
        cur = (cur + 1) % mod;
        if (cur == ori) return false;
    }
    hashTable[cur].first = key;
    hashTable[cur].second = tar;
    hashTag[cur] = 1;
    printOut(cur);
    return true;
}
int hashSearch(string key)
{
    int ori = Hash(key);
    int cur = ori;
    while (hashTag[cur] && hashTable[cur].first != key)
    {
        cur = (cur + 1) % mod;
```

```
        if (cur == ori) return INT_MAX; //错误数据，相当于异常
    }
    printOut(cur);
    return hashTable[cur].second;
}
string getA(int i)
{
    return "A" + to_string(i);
}
int main()
{
    //失败的操作不会打印
    int i = 0;
    while (i != 11)
    {
        printf("------------Insert: %s-----------\n", getA(i).c_str());
        hashInsert(getA(i), i);
        i++;
    }
    printf("------------Insert: %s-----------\n", string("B0").c_str());
    hashInsert("B0", i);
    printf("------------Insert: %s-----------\n", string("Z0").c_str());
    hashInsert("Z0", i);
    i = 5;
    while (i != 11)
    {
        printf("------------Delete: %s-----------\n", getA(i).c_str());
        hashDelete(getA(i));
        i += 2;
    }
    printf("------------Insert: %s-----------\n", string("G0").c_str());
    hashInsert("G0", 6);
    printf("------------Insert: %s-----------\n", string("E0").c_str());
    hashInsert("E0", 5);
    printf("------------Search: %s-----------\n", string("E0").c_str());
    cout << "with value: " << hashSearch("E0") << endl;
    printf("------------Search: %s-----------\n", string("G0").c_str());
    cout << "with value: " << hashSearch("G0") << endl;
    i = 0;
    while(i != 100)
    {
        printf("------------Insert: %s-----------\n", getA(i).c_str());
        hashInsert(getA(i), i);
        i++;
    }
}
```