Document Name		Confidentiality Level
File Index		Only for Recipients' Reference
Project Code	Version	Document Code
BP	1.0	BP

File Index

—— File Index, Hash Table

Prepared by		Date	
Reviewed by		Date	
Approved by		Date	

Copyright Declaration

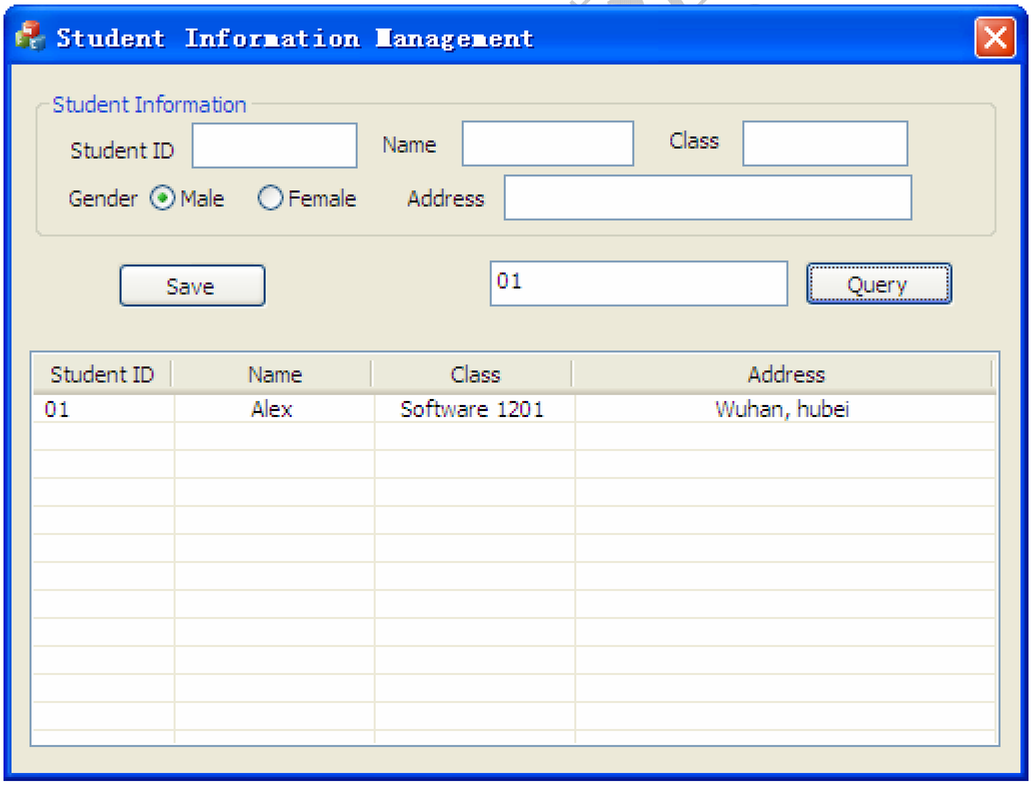
Contents

1 Teaching Tips

(1) Understand the concepts of file and file management
(2) Understand how to create and search the file index.
(3) Master the file operations.
(4) Understand how to create and search the hash table.
(5) Understand how to use the basic controls and the list control in MFC Dialog
(6) Finish the development of “File Index”program with above knowledges compreshensively.

2 Functional Requirements

Student Information Management is a MFC dialog program. Query the student information in Student.dat file through student ID. The student information includes: student ID, name, class, gender, home address. In Student.dat file, the student information is store in binary form
When we have a large amount of data in the file, the query efficiency is very low. In order to improve the query efficiency, we will use an index for searching student information.
The found student information will be displayed in the list control. Student IDs, names, classes and home addresses of the found students will be displayed in the list.



Student ID	Name	Class	Address
01	Alex	Software 1201	Wuhan, hubei

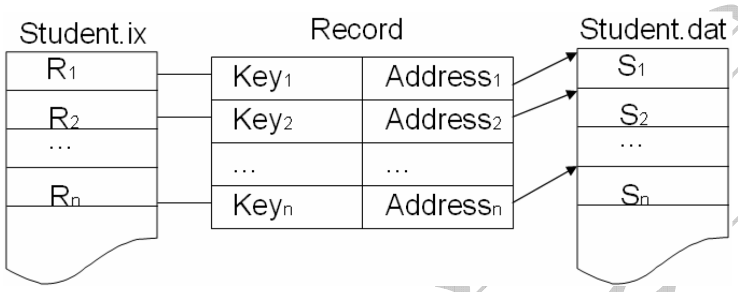
In order to maintain data consistency, when saving student information in Student.dat file, we will save the index information of the student information.

3 Design Ideas

With VS2010 development tool, create a MFC Dialog project. The solution name is FileManagerSln. The project name is FileManagerCPro. With "MFC Dialog + CFile +File+Index" technology, implement the functions of saving, reading and querying the student information.

When we query the student information, we usually read the student information in student information file Student.dat file item by item according to student ID. And compare whether student ID is the same as the student ID that needs to be queried. But, when we have a large amount of data, the query efficiency is very low.

In order to improve the query efficiency, we will create an index for each item of the student information in Student.dat file, query the location of the information in the file according to the index, and then, read this item of information from the file.



Create the index according to the student ID. Key is student ID. Value is the location of this item of the student information in Student.dat file.

3.1 Data Structure Design

1. Student Information Design

In the program, we will represent the student information with Student structure variable. Student structure includes student ID, name, class, gender, native place of the student.

```
struct Student
{
   char num[16];        // Student ID
   char name[16];       // Name
   char class_name[32];// Class
   int gender;          // Gender
   char address[128];   // Address
};
```

2. Student Information File Design

Student.dat file is used to save the data in student information Student structure. In this file, the student information is store in binary form. Student.dat file is saved under the project directory. The format of Student.dat file is as follows:

Student1	Student2	…
…	…	Studentn

3. Index Information Design

In this program, we will use student ID as the index information. The index information is stored with the hash table structure. Each item of the index information is represented with a Record structure.

(1) Record Structure

Save the index data information. Each index data includes the key word (student ID of the student) and the location of the record that corresponds to the index.

```
struct Record
{
   char key[16];    // Student ID
   ULONGLONG address; // Address
};
```

(2) IndexHead Structure

IndexHead structure is used to save the basic information of the index. It includes the index data capacity, actual data number and modified index number.

```
struct IndexHead{
   int capacity;    // Index data capacity
   int record_count;// Actual data number
   int mod_no;      // Modified index number
};
```

4. Index File Design

Student.ix file is used to save the index information. The index information is saved in this file with binary format. This file is saved under the project directory.

The format of Student.ix file is as follows:

IndexHead	Record1	Record2
…	…	Recordn

3.2 Hash Function Design

The file index is stored with the hash table. The index key is student ID. The value is the location of the student information in the file. The number of the index data in the hash table is the remainer that the int type data converted from student ID divides by the capacity of the hash table.

```
int Hash (const char* key)
{
   //Traversal key for each character, to get the sum of all the characters' ASCII
   // The sum is divided by the capacity of the hash table to get the remainder
}
```

Notice: When we save the index information with the hash table, if we get the same data through the hash function, we can't save the index information. In order to solve this conflict, we can fill

the conflicted data into the overflow table with the method of “Create a public overflow area”.
In this program, we will not solve the conflict problem first. First, we will not save the index that generates the conflict into the hash table. And then, we will optimize the index. We will introduce the concept of “Bucket” to solve the conflict problem.

3.3 Function Design

In order to maintain data consistency, when we save the student, we will save the index information into the hash table. When we search the student information, we will search the location of the corresponding student information from the hash table according to student ID.

1. Save Student Information

(1) Get the student information from the interface.
(2) Save the student information into Student.dat file, and return the location of this record in the file.
(3) Accrording to student ID and the location in the file, create an item of index information, and save this index information into the hash table.
(4) Determine whether it has been saved successfully or it has failed. If it has failed, prompt “Fail to save the index information”. If it has been saved successfully, save all the information into Student.ix file.

Algorithm of adding the index record into the hash table:
(1) Get the location of the record in the hash table according to hash function.
(2) Get the index record of this location from the hash table, and determine whether the address of this index record is null.
(3) If the address is null, save all records to this location.
(4) If the address is not null, it represents the index of this location has existed, and we can’t add the index record into the hash table.

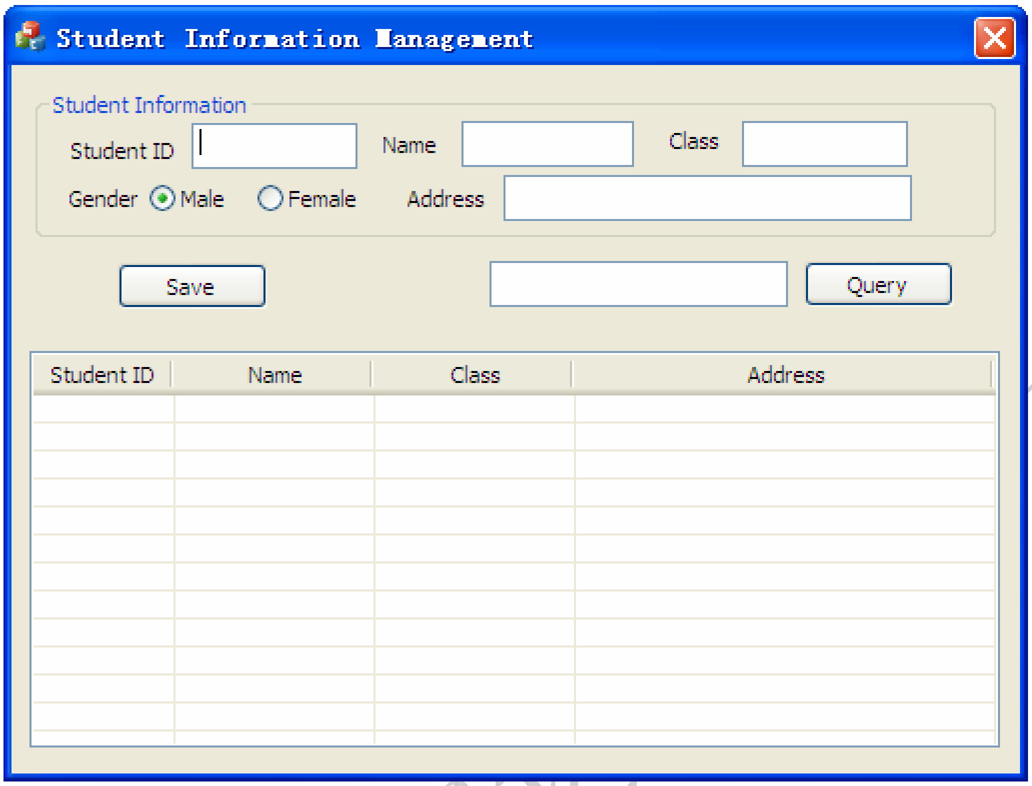
2. Search Student Information

(1) According to student ID input in the interface, query whether there is the same index record as this student ID in the hash table.
(2) If there is, return the address in the index record data.
(3) According to the address, read the student information of this location from Student.dat file.
(4) If there is none, query the student information that has the same student ID from the file.

Query Hash Table Algorithm:
(1) According to student ID and hash function, get the location of the index record in the hash table.
(2) Take out this index record from the hash table.
(3) Compare whether key value (student ID) of this index record is equal to the input student ID.
(4) If they are equal, get the address value in this index record.
(5) If they are not equal, it represents there is no index information of this student information.

3.4 Interface Design

The interface of this program is a dialog box. The interface effect is shown below:



1. With Edit control, receive the input student ID, name, class, home address, and student ID of the student that we want to find.
2. With the list control, display the found student information. Each column of the list is student ID, name, class and home address, respectively.
3. With the button, response the user's saving and query operations.

3.5 CIndex Class

1. Overview

Index class . operation of the index.

2. Member Variables

Type	Name	Description
int	m_nRecordCnt	Total number of the index record
int	m_nOverflowCnt	Overflow bucket number
int	m_nBucketCnt	Total capacity of the Hash table
Bucket*	m_pBuckets	The first address of the fixed bucket array
Bucket*	m_pOverflows	The first address of overflow bucket array
ModInfor	m_modInfor	Save the modified information

3. Mainly Methods

Prototype	Description
bool SetAt(char* pKey, ULONGLONG nAddress);	Insert index record
bool Lookup(char* pKey, ULONGLONG &nAddress)	Query index record
int Hash(const char* pKey);	Hash function

3.6 CFileDao Class

1. Overview

File operate class, read and write file.

2. Mainly Methods

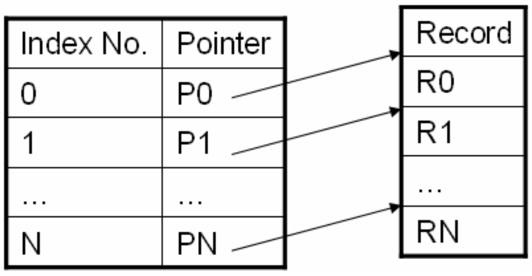
Prototype	Description
ULONGLONG Write(const Student stu);	Save student information
Student Search(ULONGLONG nAddress)	Query student information
BOOL InitIndex(CIndex* pIndex)	Initialize index file
BOOL UpdateIndex(CIndex* pIndex)	Update index information
BOOL CreateIndex(CIndex* pIndex)	Create index
BOOL ReadIndex(CIndex* pIndex)	Load index

4 Technical Analysis

This program mainly involves the technologies of file, MFC file operations, file index, and so on.

4.1 File Index

When we have a large amount of file contents, or the record length in the file is uncertain, it is very difficult to access multi files. In order to solve this problem, we can create an index table for the file. For each record in the main file, set a corresponding table entry in the index table to record the pointer that points to this record. Thus, we can implement direct access easily.



In it, the index number can be the keyword, and the pointer can be the logic address of the record in the file.

When we perform the file retrieval with the index, first, we will retrieve the index table according to the keyword and a search algorithm to find the logic address corresponding to the record, and then, search the main file to find the corresponding record in it.

The index file can be stored with many methods. The hash table is a common kind among them.

4.2 Hash Table

1. Hash Table

According to hash function H (key) and conflict handling methods that have been set, map a group of keywords to a limit continuous address set, and take “image” of the key word in the address set as the storage location of the record in the table. This kind of table is called the hash table.

2. Construction Methods of Hash Function

Common hash function construction methods include:

(1) Direct addressing method

(2) Digital analysis method

(3) Middle-square method

(4) Folding method

(5) Division method

Get the remainder that the keyword divides by a number p no larger than the hash table length m as the hash address, that is:

```
H(Key) = key MOD p;  // p<= m
```

Generally, select p as a prime number or composite number that does not contain the prime factors of 20.

(6) Random number method

In practice, we should use different hash functions according to different conditions. Usually, we will think about these factors: the time required to calculate the hash function, the keyword length, the hash table size, the key word distribution, and the record search frequency.

3. Conflict Handling Methods

The hash function can reduce conflicts, but it can not avoid conflicts. How to handle the conflicts is essential to the hash table.

(1) Open addressing method

(2) Double hashing method

(3) Chain address method

(4) Build a public overflow area

4. Search of Hash Table

The search process on the hash table is basically the same as the hash table creation process. Give k value, and get the hash address according to the hash function that has been set when the table is created. If there is no record at this location in the table, it fails to search, otherwise, compare the keywords. If the given values are equal, the search is successful, otherwise, find "next address" according to the conflict handling method that has been set when the table is created until a location in the hash table is "null" or the keyword of the record filled in the table is equal to the given value.

5 Implementation Idea

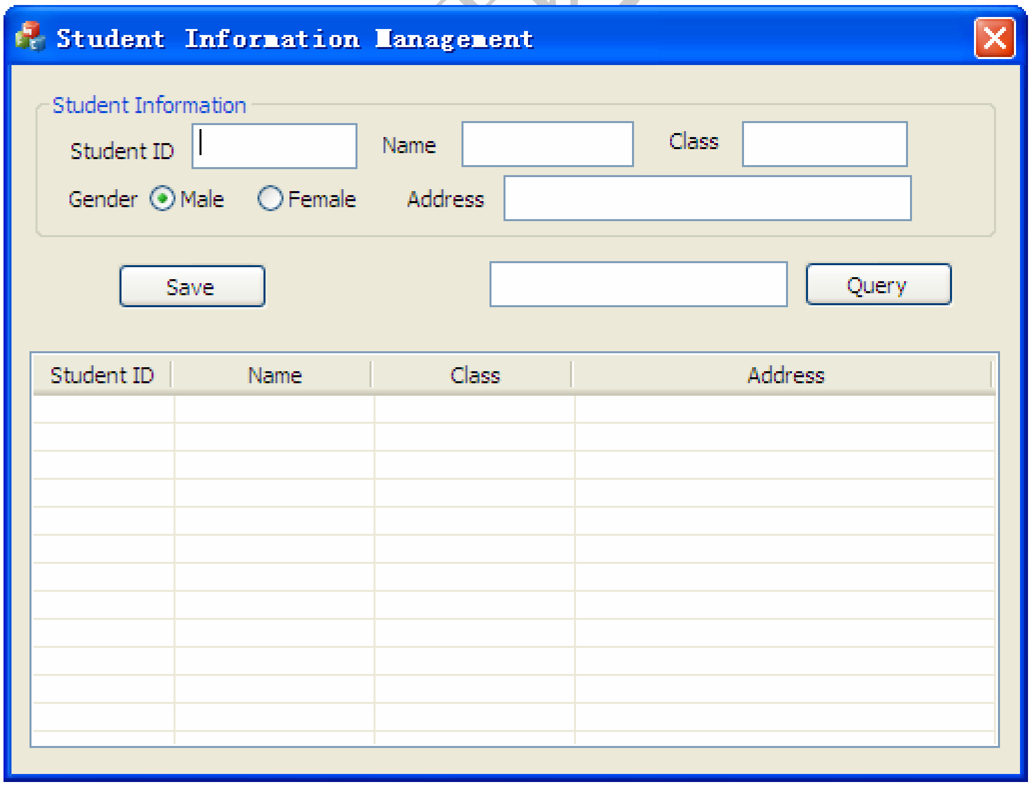
5.1 Create Project

1. Launch VS2010 development tool.
2. With VS2010 tool, create MFC dialog project. The solution name is FileManagerSln. The project name is FileManagerCPro.

5.2 Interface Layout

1. Add Controls
According to the interface design, with the tool box of VS2010, add the required controls into the dialog.

2. Interface Layout
According to the interface design, set the properties of the controls. The effect is shown below:



Student ID	Name	Class	Address

5.3 Save Student Information

1. Add the student information structure.
Add DataStructure.h file, and define student information structure Student in this file.

2. Add Save Student Information function CFileDao::Write().
(1) Define CFileDao class.
(2) Add CFileDao::Write() function.

```
class CFileDao
{
public:
    CFileDao(void);
    ~CFileDao(void);

public:
    BOOL Write(const CString strPath, const Student stu);
};
```

3. Write Save Student Information function CFileDao::Write().
In CFileDao::Write() function, save the student information into Student.dat under the project directory in binary form, and return the location of the student information in the file.

```
ULONGLONG CFileDao::Write(const Student stu)
{
    // Open file in write-only way
    // If fail to open file, create a new file
            // If fail to create file, return FALSE

     // Write file in binary way
     // Write buffer data into the file

    // Close file

    // Return address
}
```

4. Receive input student information.
(1) In "Save" button BN_CLICKED message response function CFileManagerCProDlg::OnBtnSave() function, get the input student information through control mapping.
Notice, the text input from the interface is CString. In VS2010, it is Unicode and a wide byte by default. But, in Student structure, char array is a multi-byte. So, we need to add a CCharHelp class, add member function CCharHelp::ToChar() function, through ::WideCharToMultiByte() function, convert the variables of CString type to char array of multi-byte from wide byte.
(2) In CFileManagerCProDlg::OnBtnSave() function, call CFileDao::Write() function to save the student information into Student.dat file.

5. Compile and run program.

5.4 Add Index

1. Define the index data structure.

In DateStructure.h file, define index file header data structure IndexHead and index data structure Record.

```
// The header of the index file
struct IndexHead
{
   int bucket_count;   // Total number of buckets
   int record_count;   // Record number
   int overflow_count; // Overflow bucket number
};

// Index record structure
struct Record
{
   char key[16];       // Key value
   ULONGLONG address;  // Address
};
```

2. Add CIndex class.

(1) Add CIndex class

(2) Add data members for CIndex class, the access permission is Protected.

(3) Add corresponding GetXxx() and SetXxx() methods for the members of CIndex class.

(4) Overload the constructor, through the record quantity in the hash table, initialize all information.

```
class CIndex
{
public:
   CIndex(int nCapacity);
   ~CIndex(void);

public:
   // SetXxx() and GetXxx()function

protected:
   int m_nRecordCnt;   // Total number of the record
   int m_nCapacity;    // Total capacity of the Hash table
```

```
    int m_nModNo;       // Recently updated record number
    Record* m_pRecords; // Record the first address of the array
};
```

(5) Add Hash Function.

Add int Hash(const char* pKey) function for CIndex class. Traverse each character in key, accumulate the corresponding ASCII code of the character, and take the remainder that divides the result after accumulation by the capacity as the index number.

```
int CIndex::Hash(const char* pKey)
{
    int nValue = 0;
    for (int i = 0; i < 16; i++)
    {
        int nVal = *(pKey + i);
        if (nVal != NULL)
        {
            nValue += nVal;
        }
        else
        {
            break;
        }
    }
    return (nValue % m_nBucketCnt);
}
```

(6) Add Add Record into Hash Table function

Add bool CIndex::SetAt(char* pKey, ULONGLONG nAddress) function for CIndex class.

```
bool CIndex::SetAt(char* pKey, ULONGLONG nAddress)
{
    // Get the bucket number saved in the record according to Hash
function
    // Get the corresponding record according to the number
    if (There is no record)
    {
        // Save record
    }
    return false;
}
```

3. Initialize the index information.

(1) Add member function IninIndex() for CFileDao, and get the index information in the index file.

```
BOOL CFileDao::InitIndex(CIndex* pIndex)
```

```
{
    if (No index file exists)
    {
        // Create index file and save index information into file
    }
    else
    {
        // If the file exists, read index information from file
    }
}
```

(2) Add a CIndex object m_pIndex as the data member of CFileManagerCProDlg, and initialize this data member in the constructor of CFileManagerCProDlg class.

(3) In CFileManagerCProDlg::OnInitDialog() function, call CFileDao::InitIndex() function to initialize the index information.

4. Save the index information.

(1) In CFileDao, add member function UpdateIndex() to save the index information into Student.ix file.

There are two ideas to update the index information. One is to update all records in the index. Another one will only update the modified records. In this program, we will update the modified records.

```
BOOL CFileDao::UpdateIndex(CIndex* pIndex)
{
    // Update the index file
}
```

(2) Modify CFileManagerCProDlg::OnBtnSave() function

In CFileManagerCProDlg::OnBtnSave() function, after the student information has been saved into the file successfully, according to student ID of the student and the location of this information in the file, create the corresponding index, and add the index into the hash table. If it has been added into the hash table successfully, update the index information in the file.

```
void CFileManagerCProDlg::OnBtnSave()
{
    .....;
    if (Success to save student information)
    {
        if (Success to insert index)
        {
            if (Success to update index information)
            {
                // Display student information on the list
            }
        }
    }
}
```

5. Compile and run program.

5.5 Search Student Information

1. Add Search Index Information function.
In CIndex class, add LookUp() function. According to student ID, in the index, query the address of the corresponding student information in the file.

```
bool CIndex::Lookup(char* pKey, ULONGLONG &nAddress)
{
    // Get the number according to Hash function
    // Get record according to the record number
    if (The recorded student ID is the same as the ID for querying)
    {
        nAddress = pRecord->nAddress;
        return true;
    }
    return false;
}
```

2. Search the student information according to the location.
In CFileDao, add Search() function. According the location in the file, get the student information in the location.

```
Student CFileDao::Search(ULONGLONG nAddress)
{
    // Open file
    // Move to the position specified
    // Get student information of this position
    // Close file
}
```

3. Search the student information.
In "Search" button BN_CLICKED message response function CFileManagerCProDlg::OnBtnSearch(), get the student ID that you want to query. First, call CIndex::LookUp() to query the location of this student information in the file from the hash table. And then, call CFileDao::Search() to get the student information in the file according to the location.

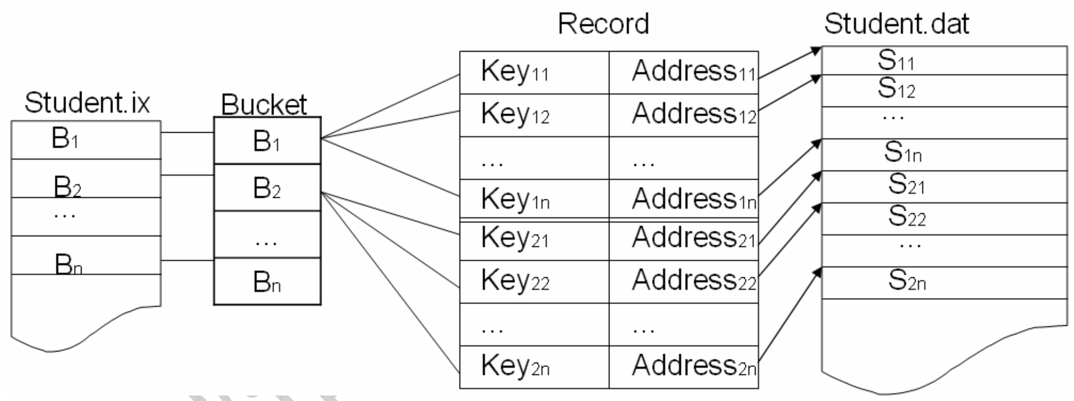
```
void CFileManagerCProDlg::OnBtnSearch()
{
    // Get student ID for querying
    UpdateData(TRUE);

    if (Success to query student information address)
    {
        // Query student information
        // Display student information on the list
    }
}
```

4. Compile and run program.

5.6 Optimize Index

When we store the index information with the hash table, generally, the index number will be less the student information record number. When multi student information correspond to an index record number, it is easy to cause conflicts. Thus, some student information indexes can not be saved. In order to save the student information indexes as many as possible, we will introduce the concept of "bucket". So, an index number can correspond to many indexes. If they still can be saved, add "overflow bucket" on this basis to save the index information.



1. Declare Bucket Structure

(1) In "DataStructur.h" file, declare bucket structure Bucket. The structure is declared as follows:

```
struct Bucket
{
    int code;              // Bucket number
    int count;             // Record number
    int overflow_no;       // Overflow bucket number
    Record records[RECORD_NUM];    // Record
};
```

(2) In "DataStructur.h" file, declare structure ModInfor that saves modified bucket information, and use it to save the modified bucket information. The structure is declared as follows:

```
struct ModInfor
{
    int mod_no;      // Bucket number
    int parent_no;   // Parent bucket number
    bool overflow;   // Whether overflow
};
```

2. Modify Data Member of CIndex Class

In CIndex class, add a pointer of Bucket type that points to the bucket and overflow bucket information in the hash table, and add the corresponding member function.

(1) Data Member

```
class CIndex
{
......;
protected:
    int m_nRecordCnt;       // Total number
    int m_nOverflowCnt;     // Overflow bucket number
    int m_nBucketCnt;        // Total capacity of the Hash table
    ModInfor m_modInfor;    // Save the modified information
    Bucket* m_pBuckets;     // The first address of the fixed bucket
array
    Bucket* m_pOverflows;   // The first address of overflow bucket
array };
```

(2) Member Function

```
Bucket CIndex::GetBucket( int nIndex )
{
    // Get bucket information from Hash table according to the bucket
number
}
void CIndex::SetBucket(int nIndex, Bucket b )
{
    // Save bucket information into Hash table
}
```

3. Modify CIndex::SetAt() Function

Save the index information into the bucket, and then, save the bucket information into the hash table. If the buck is full, save into the overflow bucket.

```
bool CIndex::SetAt(char* pKey, ULONGLONG nAddress)
{
    // Allocate space
    // Get the recorded bucker number according to hash function
    // Get bucker information according to the nunber
    // Traverse bucket
    while(pNextBucket != NULL)
    {
        pBucket = pNextBucket;
        pRecord = pBucket->records;
        for (Traverse records of the basic bucket)
        {
            if (There has a record with the same key value)
            {
                return false;
            }
            pRecord++;
        }

        if(There has a overflow bucket)
        {
            // Traverse overflow bucket
        }
    }
    if (Need to add overflow bucket)
    {
        if(Overflow space is full)
        {
            // Return false
        }
        // Initialize the overflow bucket
        // Set the number of the next overflow bucket
        // Add the overflow bucket number to 1
        // Save the modified record
        // Save the bucket of current stored record
    }
    // Save record
    // Set the modified bucket number
}
```

Notice: in this program, the overflow bucket quantity is fixed 50. The overflow bucket quantity also can be determined dynamically according to the actual situation.

4. Modify CIndex::LookUp() Function

In CIndex::LookUp() function, according to hash function, get the bucket number, and determine whether there is the student information of the same student ID. If there is an overflow bucket, we also need to traverse the overflow bucket to determine there is the student information of the same student ID. If there is, return the address in the index record.

```
bool CIndex::Lookup(char* pKey, ULONGLONG &nAddress)
{
    // Get bucket number according to Hash function
    // Get bucket
    // Traverse buckets, decide whether there has the same record
    while(pBucket != NULL)
    {
        pRecord = pBucket->records;
        for(Traverse records of the basic bucket)
        {
            if (There has the same record)
            {
                Get the record address
            }
        }
        // Movt to overflow bucket, traverse overflow buckets
    }

    return false;
}
```

5. Modify CFileDao Class

After CIndex class is modified, the index information will be saved into Student.ix file with the binary format of Bucket structure. So, we need to modify Index Information Initialization and Read/Write functions in CIndex class.

(1) Modify CFileDao::CreateIndex() function

In CFileDao::CreateIndex() function, initialize the content of Student.ix file according to the bucket data in CIndex.

```
BOOL CFileDao::CreateIndex(CIndex* pIndex)
{
    ......;
    // Initialize the content of Student.ix file according to the
bucket data in Cindex
    ......;
}
```

(2) Modify CFileDao::ReadIndex() function

In CFileDao::ReadIndex() function, save the content in Student.ix file into the bucket of CIndex.

```
BOOL CFileDao::ReadIndex(CIndex* pIndex)
{
    ......;
    // Read bucket information from Student.ix file and save it into
CIndex
    .....;
}
```

(3) Modify CFileDao::UpdateIndex() function

In CFileDao::UpdateIndex() function, get the modified information. Determine whether there is an overflow bucket in the modified information. If there is an overflow bucket, write the overflow bucket data into the file first, and then, update the parent bucket information. If there is none, directly update the bucket information.

```
BOOL CFileDao::UpdateIndex(CIndex* pIndex)
{
   ......;
   // Get the modified information
   if (There has overflow bucket)
   {
      // Write the data of overflow bucket into file

      // Update parent bucket information

   }
   else
   {
      // Update bucket information

   }
   .....;
}
```

6. Compile and run the program.