

**class Procedure Reality test report tell**

**Course Title: Data Structure Experiment**

**professional class: Computer Science and Technology (Inter-school Exchange) Class 1601**

**study No.: U201612696**

**surname Name: Chen Haorui**

**mentor: Wei Wei**

**Date of report: January 11 , 2018**

**School of Computer Science and Technology**

head record

[1 Realization of linear table based on sequential storage structure](#_Toc501754122)  [2](#_Toc501754122)

[1.1 Problem description](#_Toc501754123)  [2](#_Toc501754123)

[1.2 System Design](#_Toc501754127)  [4](#_Toc501754127)

[1.3 System Implementation](#_Toc501754133)  [10](#_Toc501754133)

[1.4 Experiment summary](#_Toc501754138)  [20](#_Toc501754138)

[2 Realization of linear table based on linked storage structure](#_Toc501754139)  [21](#_Toc501754139)

[2.1 Problem description](#_Toc501754140)  [21](#_Toc501754140)

[2.2 System Design](#_Toc501754144)  [23](#_Toc501754144)

[2.3 System Implementation](#_Toc501754150)  [30](#_Toc501754150)

[2.4 Experimental summary](#_Toc501754155)  [40](#_Toc501754155)

[3 Implementation of binary tree based on binary linked list](#_Toc501754156)  [41](#_Toc501754156)

[3.1 Problem description](#_Toc501754157)  [41](#_Toc501754157)

[3.2 System Design](#_Toc501754161)  [44](#_Toc501754161)

[3.3 System Implementation](#_Toc501754167)  [50](#_Toc501754167)

[3.4 Experimental summary](#_Toc501754172)  [62](#_Toc501754172)

[4 Graph Implementation Based on Adjacency List](#_Toc501754173)  [63](#_Toc501754173)

[4.1 Problem description](#_Toc501754174)  [63](#_Toc501754174)

[4.2 System Design](#_Toc501754178)  [65](#_Toc501754178)

[4.3 System Implementation](#_Toc501754184)  [72](#_Toc501754184)

[4.4 Experimental summary](#_Toc501754189)  [83](#_Toc501754189)

[References](#_Toc501754190)  [84](#_Toc501754190)

[Instructor Evaluation Comments](#_Toc501754191)  [85](#_Toc501754191)

[Appendix A Source Program Based on Sequential Storage Structure Linear Table](#_Toc501754192)  [87](#_Toc501754192)

[102 Realized Based on Linked Storage Structure Linear Table](#_Toc501754193)

[Appendix C Source Program Based on Binary Linked List Binary Tree](#_Toc501754194)  [117](#_Toc501754194)

[Appendix D Source Program Based on Adjacency List Diagram](#_Toc501754195)  [135](#_Toc501754195)

# 1 Linear table implementation based on sequential storage structure

## 1.1 Problem description

* + 1. Basic concept of linear table

A linear list is the most commonly used and simplest data structure, that is, a finite sequence of n data elements. The number n of elements in the linear list is defined as the length of the linear list, and when n=0, it becomes an empty list. Each data element in the non-empty table has a certain position, such as a1 is the first data element, an is the last data element, and ai is the i-th data element. The storage structure of the linear table is divided into linear storage and chain storage.

* + 1. Logical structure and basic operations

The data logic structure of the linear table is defined as follows:

ADT List{

Data object: D={ai|ai∈ElemSet, i=1, 2,……, n, n≥0}

Data relationship: R1={<ai-1, ai> | ai-1, ai∈D, i=2,……,n}

}

Based on the principle of combining minimum completeness and common use, the initialization table, loading table, saving table, destroying table, clearing table, judging empty table, finding the length of the table, obtaining elements, finding elements, and There are 14 basic operations of obtaining predecessor, obtaining successor, inserting element, deleting element, and traversing the table. It is required to define functions to realize the above functions. The specific functional operations are as follows:

1 . Initialization table: the function name is InitaList(L); the initial condition is that the linear list L does not exist and already exists; the operation result is to construct an empty linear list.

2 . Destroy the list: the function name is DestroyList(L); the initial condition is that the linear list L already exists; the operation result is to destroy the linear list L.

3 . Clear the list: the function name is ClearList(L); the initial condition is that the linear list L already exists; the operation result is to reset L to an empty list.

4 . Determine the empty list: the function name is ListEmpty(L); the initial condition is that the linear list L already exists; the operation result is to return TRUE if L is an empty list, otherwise return FALSE.

5 . Find the length of the table: the function name is ListLength(L); the initial condition is that the linear table already exists; the operation result is to return the number of data elements in L.

6 . Get elements: the function name is GetElem(L,i,e); the initial condition is that the linear list already exists, 1≤i≤ListLength(L); the operation result is to use e to return the value of the i-th data element in L.

7 . Find elements: the function name is LocateElem(L,e,compare()); the initial condition is that the linear table already exists; the operation result is to return the bit order of the first data element in L that satisfies the relationship compare() with e, if If no such data element exists, the return value is 0.

8 . Get the predecessor: the function name is PriorElem(L, cur\_e, pre\_e); the initial condition is that the linear table L already exists; the operation result is that if cur\_e is the data element of L and is not the first one, use pre\_e to return its predecessor, otherwise Operation failed, pre\_e is not defined.

9 . Get the successor: the function name is NextElem(L, cur\_e, next\_e); the initial condition is that the linear table L already exists; the operation result is that if cur\_e is the data element of L and not the last one, use next\_e to return its successor, otherwise the operation Failed, next\_e is not defined.

10 . Insert elements: the function name is ListInsert(L,i,e); the initial condition is that the linear list L already exists and is not empty, 1≤i≤ListLength(L)+1; the operation result is inserted before the i-th position of L The new data element e.

11 . Delete elements: the function name is ListDelete(L,i,e); the initial condition is that the linear list L already exists and is not empty, 1≤i≤ListLength(L); the operation result: delete the i-th data element of L, use e returns its value.

12 . Traversing the table: the function name is ListTraverse(L,visit()), the initial condition is that the linear table L already exists; the operation result is to call the function visit() on each data element of L in turn.

* + 1. Multilinear table abstract data type

Based on the previously implemented linear table data types, the data objects and data relationships of the multi-linear table are designed and the basic operations of the multi-linear table are defined. The specific data and operation functions are defined as follows.

A DT MulList{

Data object: D = { e i |e i ∈ SqList , i = 1,2, … , n, n ≥ 0 }

Data relationship: R1 = { <e i-1 ,e i >|e i-1 ,e i ∈ D , i = 2,...,n}

}

In the form of functions, two basic operations including linear multi-table change operation object and traversal are defined. It is required to define functions separately to realize the above functions. The specific function operations are as follows:

1. Chooselist ( vol , &L, List )

The initial condition is that the table MList is not empty; the result of the operation is to traverse and point L to the volth table in the List .

2. ChartTraverse(&MList, visit() )

The initial condition is that the table MList is not empty; the operation result is to traverse all the linear tables in MList, and call visit() on the element traversal.

## 1.2 System design

* + 1. data physical structure

The physical structure of the data related to the linear table is as follows:

typedef struct { //Definition of sequence table (sequence structure)

ElemType \*elem; //Define an integer pointer, which is the base address of the storage space

int length; //The length of the linear table

int listsize; //Currently allocated storage capacity (in sizeof(ElemType))

}SqList;

To manage multiple linear tables at the same time, you only need to define a structure array.

In this program, the data atomic type ElemType is defined as an int integer.

* + 1. demo system

The sequence table is used as the physical structure of the linear table to realize the basic operation of § 1.1.2 . Among them, ElemType is the type name of the data element, and the specific meaning can be defined by oneself. For the definition and reference of other related types and constants, please refer to p10 of the literature [1] .

It is required to construct a functional demonstration system with menus. Among them, in the main program, the preparation of the actual parameter value required for the function call and the display of the function execution result are completed, and an appropriate operation prompt is given.

The demonstration system can choose to save the file form of the linear table. Among them, ① it is necessary to design the file data record format to efficiently save the complete information of the linear table data logical structure (D, {R}) ; ② it is necessary to design a reasonable mode of linear table file saving and loading operations.

The demo system chooses to implement multiple linear table management.

in each cycle through the printf function , and op is initialized to 1 for entering the cycle for the first time. op=0 is the only exit condition of the loop, which is given in the menu . 1-14 respectively represent a basic operation of the linear table. After updating the value of op in each cycle , execute the corresponding function through the switch statement in the main function . After executing the function, break jumps out of the switch statement and starts the next while loop. Until the user enters 0 to select exit, exit the system. The system design structure is shown in Figure 1-1 .

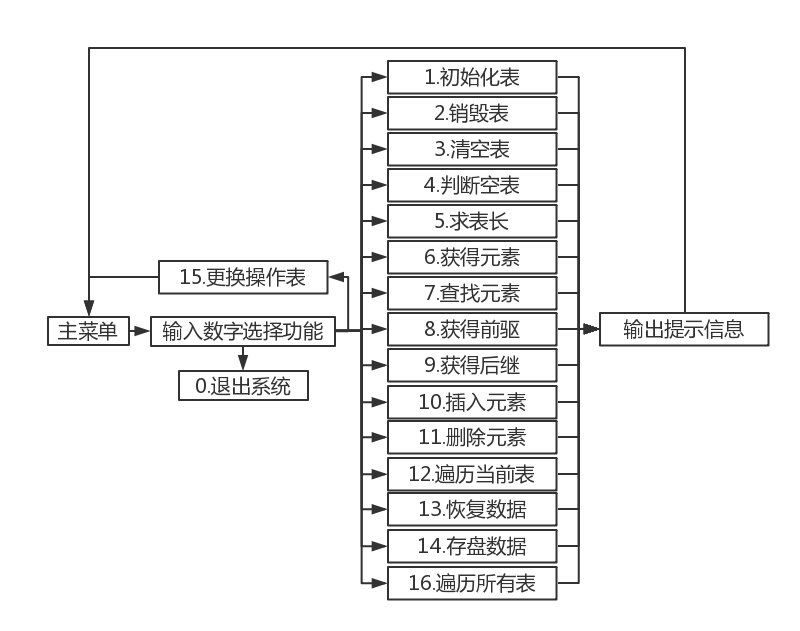


Figure 1-1 System design structure diagram

* + 1. Data file storage format

Considering the principle of simplicity and efficiency, the data file storage format is designed as follows.

File format: dat , which is convenient for programs to read data quickly .

The corresponding file space is allocated according to the corresponding data type in order for storage, and there is no gap between the data .

* + 1. Algorithm thinking and design

The idea and design of the linear table operation algorithm are as follows:

1. The idea of initializing the linear table: the parameter of the function is the reference of the structure type pointer variable L defined in the main function. In the function, first use the malloc function to allocate a continuous memory space of the size of LISTSIZE, and return the first address and assign it to L- >elem, since the length of the linear table is 0, initialize L->length to 0, that is, the initialization of the linear table is completed.

2. The idea of destroying the linear table: the parameter of the function is the structure type pointer variable L. Legality needs to be judged at the outset. Define a decision array of the same size as the number of tables and its dedicated subscripts. For each decision variable in the array, TRUE (1) means that the corresponding table has been initialized, otherwise it means that it has not been initialized ( the value of the element in the array that has not been modified is uncertain ) . Both array and subscript variables are global variables. Modify the subscript while modifying the operation table , and write the process of destroying the linear table as a function, where the incoming function is the structure type pointer variable L defined in the main function. In the function, first use the free function to release the continuous memory space headed by L->elem, reassign L->length, L->listsize to 0, and modify the corresponding decision variable to FALSE ( - 1 ) .

3. The idea of clearing the linear table: the parameter of the function is a reference to the structure type pointer variable L, and the legality needs to be judged at the beginning. In the function, since the clearing operation does not release the memory space, it is only necessary to set the length of the linear table to 0.

4. The idea of judging whether it is an empty table : the parameter of the function is a reference to the pointer variable L defined in the main function, and the legality needs to be judged at the beginning. When calling this function, directly judge whether L -> length is 0, if it is 0, it is empty, otherwise it is not empty .

5. The idea of finding the length of the linear table: the parameter of the function is a reference to the pointer variable L defined in the main function, and the legality needs to be judged at the beginning. In the function, directly returning L -> length is the length of the linear table to be obtained.

6. The algorithm idea of obtaining elements: the parameters of the function are the structural variable L and the serial number i of the data element, and the legality needs to be judged at the beginning. Since the linear storage structure is adopted, the elements are obtained directly by accessing the array, that is, L->elem[i-1]. Of course, the legality needs to be judged before this.

7. Algorithm idea of finding elements: The parameters of the function are the structure type pointer variable L defined in the main function and the value of the data element to be searched, and the legality needs to be judged at the beginning. Compare each element in the linear list with the given value to see if they are equal by looping, and return the order of the element if they are equal.

8. Obtain the idea of the predecessor algorithm: the parameters of the function are the pointer variable of the structure type and the value of the specific data element, and the variable of the predecessor is accepted as another parameter, and the legality needs to be judged at the beginning. First call the function to obtain the element to judge the bit sequence of the specific data element in the linear table, first judge that it is not 1, otherwise return FALSE, and then directly return the previous element that is L->elem[j-2]. Its flow chart is shown in Figure 1-2 .

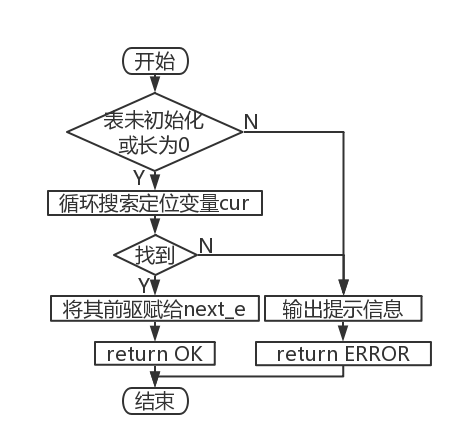
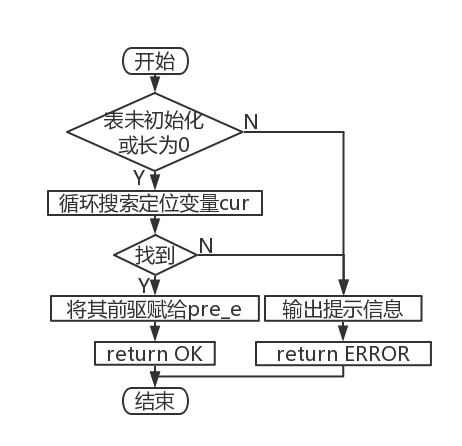


Figure 1-2 Obtaining the flow chart of predecessor implementation Figure 1-3 Obtaining the flow chart of successor implementation

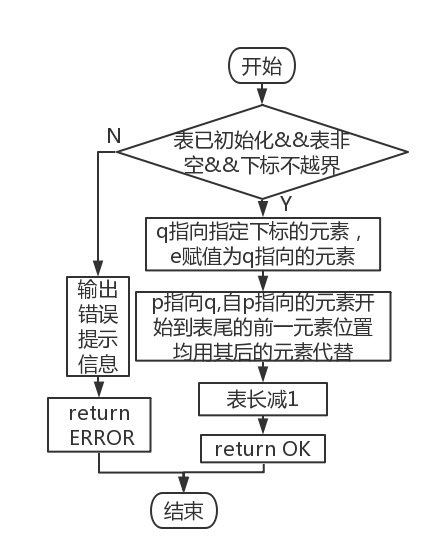
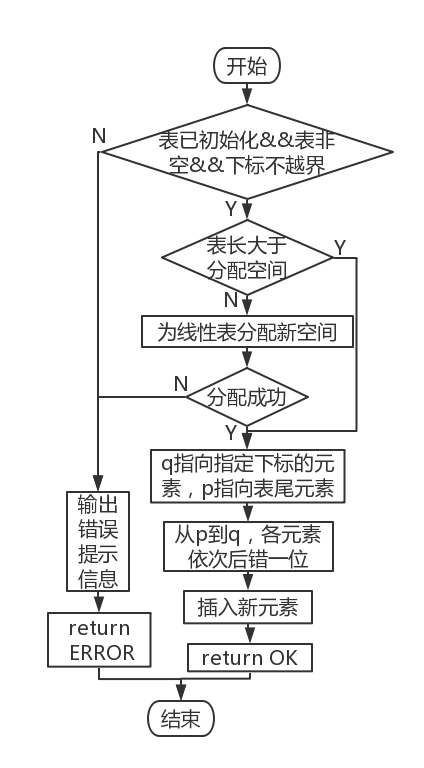


Figure 1-4 Insert element implementation flow chart Figure 1-5 Flow chart of deleting elements

9. Obtain the subsequent algorithm idea: the parameters of the function are the structure type pointer variable and the value of the specific data element. Legality needs to be judged at the outset. First judge whether it is the last element, if not, return the next element directly , otherwise return FALSE, and the algorithm also needs to call the function to obtain the element to determine the order of the specific element in the linear list. Its flow chart is shown in Figure 1-3 .

10. The algorithm idea of inserting elements: The parameters of the function are structure type pointer variables, the value size of the inserted element, and the insertion position. Legality needs to be judged at the outset. In the function, first judge the validity of the insertion position, that is, whether it is in a suitable position in the linear table, and then judge whether the current storage space is full. If it is full, the malloc function must be used to reallocate the space. When inserting elements, start from this position Go to the last element and move backward one unit from the beginning. Its flow chart is shown in Figure 1-4 .

11. Algorithm idea of deleting elements: The parameter of the function is a pointer variable of structure type. First, judge the legitimacy of the bit order, and then directly delete the element after the position of the element until the last element, so as to move one unit from front to back. Its flow chart is shown in Figure 1-5 .

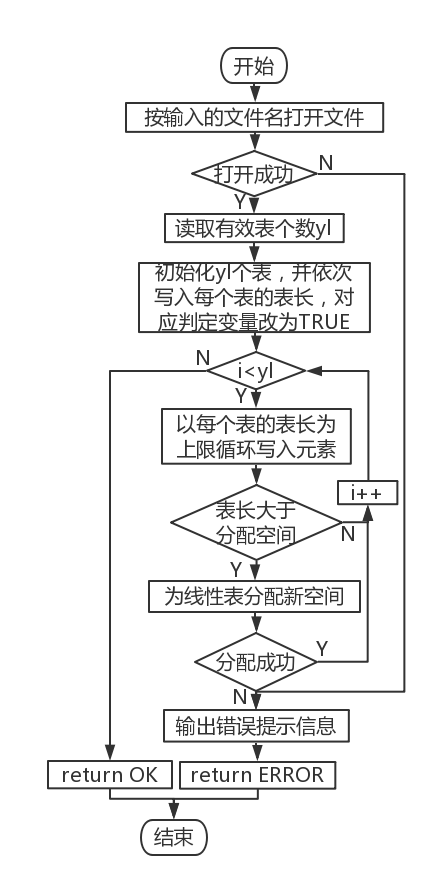
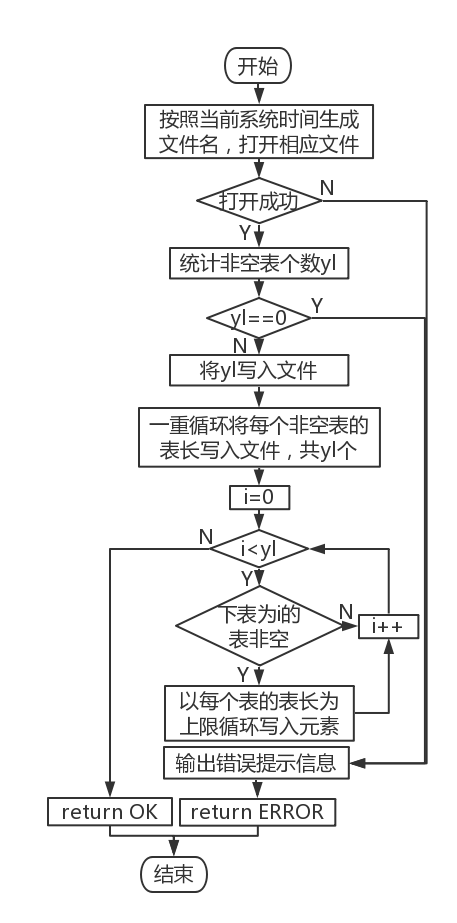


Figure 1-6 Data storage implementation flow chart Figure 1-7 Data reading implementation flow chart

12. Algorithm idea of traversing the linear table: The parameter of the function is a structure type pointer variable, and the legality needs to be judged at the beginning. Directly use a loop to operate on each element in the linear list.

13. Data storage algorithm idea : The parameters of the function are file pointers and structure array variables . The file name is automatically generated from the system time , and the corresponding binary file is generated. First count the number of initialized and non- empty tables , and return an error directly if there is no non-empty table. Save this data ; secondly, for each non-empty table , save its table length; finally, for each non-empty table, read each element in the table in a loop, and finally close the file. Its flow chart is shown in Figure 1-6 .

14. Data reading algorithm idea: The parameters of the function are file pointers and structure array variables . First read the number of valid tables , initialize the same number of tables according to this number, read the table lengths of the same number of tables and write them into different tables in order , and finally loop according to the table lengths that have been read Read the elements in each table , close the file. Its flowchart is shown in Figure 1-7 .

15. Algorithm idea of changing tables : The parameters of the function are the subscript of the table to be operated , the secondary pointer variable of the structure and the array variable of the structure . Legality needs to be judged at the outset. Assign the address of the table to be operated to the pointer of the same structure , and output a prompt message if the table is not initialized.

16. Cross- table traversal algorithm idea : The parameters of the function are structure array variables . Starting from the first table , if the current table is uninitialized or empty, output the corresponding prompt information, otherwise output all elements in the table.

* + 1. Algorithm Time and Space Complexity Analysis

This system is a demonstration system for the operation of the sequence table. Since the operation on the sequence table is relatively simple, and this program does not deliberately use a high-complexity algorithm, the highest space/time complexity involved in this program is only O(n 2 ) .

Table 1-1 Algorithm time and space complexity analysis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function name | time complexity | space complexity | Function name | time complexity | space complexity |
| IntiaList | O(1) | O(1) | NextElem | O(n) | O(1) |
| DestroyList | O(1) | O(1) | ListInsert | O(n) | O(1) |
| ClearList | O(1) | O(1) | ListDelete | O(n) | O(1) |
| ListEmpty | O(1) | O(1) | List Trabverse | O(n) | O(1) |
| ListLength | O(1) | O(1) | LoadList | O(n 2 ) | O(1) |
| GetElem | O(1) | O(1) | SaveList | O(n 2 ) | O(1) |
| LocateElem | O(n) | O(1) | ChooseList | O(1) | O(1) |
| PriorElem | O(n) | O(1) | Chart Traverse | O(n 2 ) | O(1) |

## 1.3 System implementation

* + 1. Programming environment , operating environment , project engineering description

This experiment is written with Codeblocks programming software and compiled and run with Codeblocks . The project name is Sequential Linear Chart (Cpp). The demo interface menu is shown in Figure 1-8 .



Figure 1-8 demonstrates the system menu interface

* + 1. header files and predefined constants

1. Header file

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

2. Predefined constants

#define OK 1

#define TRUE 1

#define FALSE -1

#define ERROR -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3. Type expressions

typedef int status;

typedef int ElemType;

typedef struct{

ElemType \*elem;

int length;

int listsize;

}SqList;

* + 1. Test Plan

The linear table operation function of the system is tested according to the following plan:

1. Detect the working conditions of the system under normal conditions;
2. Focus on checking the working conditions during illegal border operations.

Table 1-2 Demonstration system test plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function to be tested | test sequence | test input | expected output | Linear table status |
| 1. IntiaList | 3 | none | Initialize L successfully | table length 0 , linear table is created |
| 2. Destroy List | 11 | none | Destroyed successfully | does not exist |
| 3. ClearList | 10 | none | Table cleared successfully | empty table |
| 4. ListEmpty | called multiple times | none | L is displayed as uninitialized when it points to an uninitialized table , as an empty table when it points to an empty table, and as a non-empty table when it points to a non - empty table | as the previous step |
| 5. List Length | called multiple times | none | L points to an uninitialized table, it is displayed as uninitialized, and when it points to an initialized table, it displays the table length | as the previous step |
| 6. GetElem | 5 | Enter 1 and 8 respectively | The first input should output 7362, the second input subscript is out of bounds and cannot be found | same as step 4 |
| 7. LocateElem | 6 | Enter 6363 and 53463 respectively | The first input should output 2, the second cannot be found | same as step 5 |
| 8. PriorElem | 7 | Enter 8585, 7362 and 8614 respectively | The first input should output 6363 , the second input has no predecessor, the lookup fails, and the third input cannot be found | Same as step 6 |
| 9. NextElem | 8 | Enter 7362, 8585 and 7361 respectively | The first input should output 6363 , the second input has no successor, the lookup fails , and the third input cannot be found | same as step 7 |
| 10. ListInsert | 4 | Enter 1, 7362,; 2, 6363; 3, 8585 and -52,2 respectively | For the first three inputs, it shows that the insertion is successful; for the last input, it should prompt that the subscript is out of bounds, and the insertion fails | as the data corresponding to the subscript within a reasonable range is updated |
| 11. ListDelete | 9 | Enter 3 and 5 respectively | For the first input , it should output that the deletion is successful , and the deleted element is 8585 ; for the second input, it should prompt that the subscript is out of bounds, and the deletion failed | After deleting the third element 8585, the remaining elements are rearranged in order |
| 12. List Trabverse | called multiple times | none | If there are elements in the table , print all its elements in a loop; if the table is an empty table , the output prompts that the table is empty; if the table is not initialized , the output prompts that the table is not initialized | as the previous step |
| 13. LoadList | 1, 13 | binary file name | If the binary file is opened successfully , each table stored in the file should be output . If the corresponding file is not found or the file fails to be opened, an error message will be output. | (point to the first table by default ) When opening the binary file successfully, it should be the first table stored in the binary file , and remain uninitialized when opening the file fails |
| 14. SaveList | 12 | none | new binary data file under the source file directory , the file name is the time of the save operation, and the format is "week\_month abbreviation\_date\_hour\_minute\_second\_year " | as step 11 |
| 15. Choose List | 2 | 20 | L points to the last table in memory | Depending on the state of the last table itself, it may be uninitialized, empty, or non-empty |
| 16. Chart Traverse | called multiple times | none | For a table with elements , loop to print all its elements; for an empty table, the output prompts that the table is empty; for an uninitialized table, the output prompts that the table is not initialized | as the previous step |

* + 1. test

1. At the beginning, all 20 linear tables are empty, and the system operates the first table by default.



Figure 1-9 Before importing data

2. Import data by entering a filename.

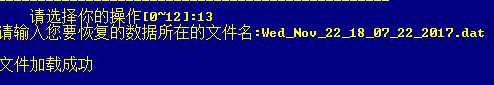


Figure 1-10 Enter the file name and import data

3. Traverse again and find that the data has been imported successfully.

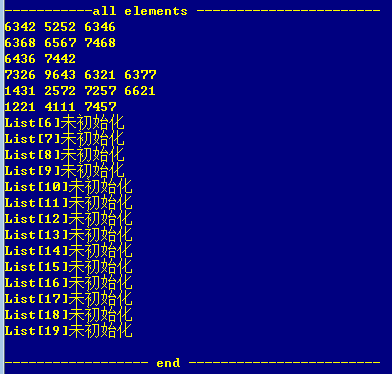


Figure 1-11 import data successfully

4. Operate on List [19] , the last list. It can be found that the output prompt matches the information when traversing the entire table .



Figure 1-12 Change operation table



Figure 1-13 Unified menu prompts for subsequent operations

5. At this time , any form of access to the table (function 2-12 ) will prompt that the table is not initialized . Examples are as follows :



Figure 1-14 \_ Error message when destroying linear table ( uninitialized )



Figure 1-15 \_ Error message for judging an empty table ( not initialized )

6. This table is initialized , and the table length is 0 at this time.



Figure 1-16 Initialize the current table



Figure 1-17 Judgment table length

7 . At this time , any form of access to the table (function 2-12 ) will prompt that the table is empty. Examples are as follows :

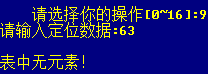
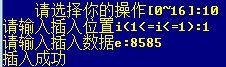


Figure 1-18 \_ Ask for subsequent error prompts ( empty list )



Figure 1-19 traversal error prompt ( empty table )

8 . Elements 7362, 6363, 8585 are inserted successively.



Figures 1-20 \_ element inserted successfully

9. Insertion fails when the subscript is out of bounds .

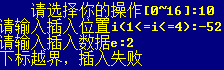


Figure 1-21 \_ \_ Failed to insert element

10. Traverse the current table. At this time , there are 3 elements in the current table .

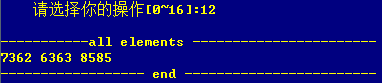


Figure 1-22 \_ \_ iterate over the current table



Figure 1 - 2 3 Judgment of empty table



Figure 1-24 \_ \_ Ask for table length

11. Get the first element .



Figure 1-25 \_ \_ get element success



Figure 1-26 \_ \_ Failed to get element

12. Find the elements in the current table .



Figure 1-27 find elements successfully



Figure 1-28 \_ \_ Failed to find element

13. Get the element precursor.



Figure 1-29 Get element precursor successfully



Figure 1-30 Get the head precursor



Figure 1-31 element not found

14. Get the element successor .



Figure 1-32 \_ \_ Get the successor of the element



Figure 1-33 Get tail element successor

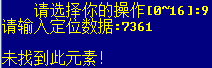


Figure 1-34 element not found

15. Delete elements .



Figure 1-35 \_ \_ Delete element successfully



Figure 1-3 6 Failed to delete element ( subscript out of bounds )

16. Traverse the current table again and find that 8585 disappears .



Figure 1-37 \_ \_ Traversing the current table to check for deleted elements

17. Clear the current table and check the clearing situation.



Figure 1-38 \_ \_ clear current table



Figure 1-39 \_ \_ ListEmpty checks for emptiness

18 . Delete this table.



Figure 1-40 \_ \_ delete current table

19. Review table deletions.



Figure 1-41 \_ \_ ListEmpty check for deletion



Figure 1-42 \_ \_ ListLength check for deletion

20. Initialize the 7th , 9th , 14th , and 20th tables , insert some elements into each of the 12th, 15th, and 18th tables, and traverse all the tables .



Figure 1-43 \_ \_ Iterate over all tables after some input

twenty one. Save the modified table as a binary file

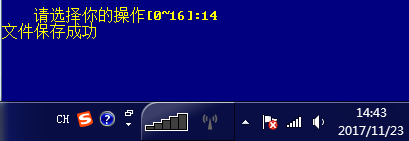


Figure 1-44 \_ \_ File saved successfully



Figure 1-45 Saved files

twenty two. Read the file just saved to the system.

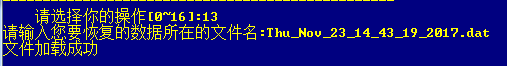


Figure 1-46 \_ \_ read file successfully

twenty three. Re-traversing all the tables shows that only initialization is performed , and tables with no elements and uninitialized tables are skipped when saving to disk, resulting in intensive table storage in the new form .

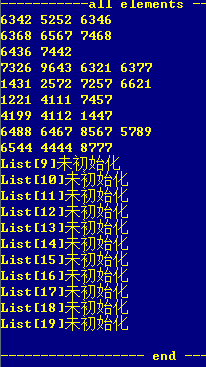


Figure 1-47 \_ \_ traverse all tables

twenty four. When opening a file with the wrong format , it will display that the file failed to open.



Figure 1-48 \_ \_ target file is not readable

25. Log out of the system .

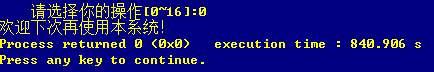


Figure 1-49 \_ Exit system

## 1.4 Experimental summary

After this experiment, I fully understood the linear physical structure of the linear table, and mastered the basic operations of the linear table through personal experience, and improved my ability to write codes related to the linear table, especially in the process of writing. I encountered many difficulties, and finally solved them with the help of my classmates many times. I also discovered my own weakness, that is, there were many flaws in the processing of the file, which resulted in the failure to read the file. And I also deepened the distinction between the existence and emptiness of linear tables. There is also an understanding of the "&" reference symbol, which strengthens its difference from "\*". "&e" allows the value "e" in the main function to be called in the function to change its value at the same time, such as calling in the GetElem function Change the value of "e" and then output it in the main function. If you want to change its value, you must use the "&" symbol.

# 2 Realization of linear table based on linked storage structure

## 2.1 Problem description

1. * 1. Basic concept of linear table

A linear list is the most commonly used and simplest data structure, that is, a finite sequence of n data elements. The number n of elements in the linear list is defined as the length of the linear list, and when n=0, it becomes an empty list. Each data element in the non-empty table has a certain position, such as a1 is the first data element, an is the last data element, and ai is the i-th data element. The storage structure of the linear table is divided into linear storage and chain storage.

* + 1. Logical structure and basic operations

The data logic structure of the linear table is defined as follows:

ADT List {

Data object: D={ai|ai∈ElemSet, i=1, 2,……, n, n≥0}

Data relationship: R1={<ai-1, ai> | ai-1, ai∈D, i=2,……,n}

}

Based on the principle of combining minimum completeness and common use, the initialization table, loading table, saving table, destroying table, clearing table, judging empty table, finding the length of the table, obtaining elements, finding elements, and There are 14 basic operations of obtaining predecessor, obtaining successor, inserting element, deleting element, and traversing the table. It is required to define functions to realize the above functions. The specific functional operations are as follows:

1 . Initialization table: the function name is InitaList(L); the initial condition is that the linear list L does not exist and already exists; the operation result is to construct an empty linear list.

2 . Destroy the list: the function name is DestroyList(L); the initial condition is that the linear list L already exists; the operation result is to destroy the linear list L.

3 . Clear the list: the function name is ClearList(L); the initial condition is that the linear list L already exists; the operation result is to reset L to an empty list.

4 . Determine the empty list: the function name is ListEmpty(L); the initial condition is that the linear list L already exists; the operation result is to return TRUE if L is an empty list, otherwise return FALSE.

5 . Find the length of the table: the function name is ListLength(L); the initial condition is that the linear table already exists; the operation result is to return the number of data elements in L.

6 . Get elements: the function name is GetElem(L,i,e); the initial condition is that the linear list already exists, 1≤i≤ListLength(L); the operation result is to use e to return the value of the i-th data element in L.

7 . Find elements: the function name is LocateElem(L,e,compare()); the initial condition is that the linear table already exists; the operation result is to return the bit order of the first data element in L that satisfies the relationship compare() with e, if If no such data element exists, the return value is 0.

8 . Get the predecessor: the function name is PriorElem(L, cur\_e, pre\_e); the initial condition is that the linear table L already exists; the operation result is that if cur\_e is the data element of L and is not the first one, use pre\_e to return its predecessor, otherwise Operation failed, pre\_e is not defined.

9 . Get the successor: the function name is NextElem(L, cur\_e, next\_e); the initial condition is that the linear table L already exists; the operation result is that if cur\_e is the data element of L and not the last one, use next\_e to return its successor, otherwise the operation Failed, next\_e is not defined.

10 . Insert elements: the function name is ListInsert(L,i,e); the initial condition is that the linear list L already exists and is not empty, 1≤i≤ListLength(L)+1; the operation result is inserted before the i-th position of L The new data element e.

11 . Delete elements: the function name is ListDelete(L,i,e); the initial condition is that the linear list L already exists and is not empty, 1≤i≤ListLength(L); the operation result: delete the i-th data element of L, use e returns its value.

12 . Traversing the table: the function name is ListTraverse(L,visit()), the initial condition is that the linear table L already exists; the operation result is to call the function visit() on each data element of L in turn.

* + 1. Multilinear table abstract data type

Based on the previously implemented linear table data types, the data objects and data relationships of the multi-linear table are designed and the basic operations of the multi-linear table are defined. The specific data and operation functions are defined as follows.

A DT MulList{

Data object: D = { e i | e i ∈ SqList , i = 1,2, … , n, n ≥ 0 }

Data relationship: R1 = { <e i-1 ,e i >|e i-1 ,e i ∈ D , i = 2,...,n}

}

In the form of functions, two basic operations including linear multi-table change operation object and traversal are defined. It is required to define functions separately to realize the above functions. The specific function operations are as follows:

1. Chooselist ( vol , &L, List )

The initial condition is that the table MList is not empty; the result of the operation is to traverse and point L to the volth table in the List .

2. ChartTraverse(&MList, visit() )

The initial condition is that the table MList is not empty; the operation result is to traverse all the linear tables in MList, and call visit() on the element traversal.

## 2.2 System design

* + 1. data physical structure

1. The data physical structure of the element node

typedef struct lnode{ //element node definition

ElemType data; // data field

struct lnode \*next; // pointer field

}LNode;

2 . The data physical structure of the head node

typedef struct llist{ //head node definition

int len; //len stores the length of this table

struct lnode \*head, \*tail; //head points to the first element in the table, tail points to the tail element

struct llist \*fore, \*aft; //aft points to the next table header node, and ore points to the previous table header node. This way of definition determines that there should be a blank header node before the first header node.

}Linklist;

In this program, the data atomic type ElemType is defined as an int integer.

* + 1. demo system

The linked table is used as the physical structure of the linear table to realize the basic operation of § 2.1.2 . Among them, ElemType is the type name of the data element, and the specific meaning can be defined by oneself. For the definition and reference of other related types and constants, please refer to p10 of the literature [1] .

It is required to construct a functional demonstration system with menus. Among them, in the main program, the preparation of the actual parameter value required for the function call and the display of the function execution result are completed, and an appropriate operation prompt is given.

The demonstration system can choose to save the file form of the linear table. Among them, ① it is necessary to design the file data record format to efficiently save the complete information of the linear table data logical structure (D, {R}) ; ② it is necessary to design a reasonable mode of linear table file saving and loading operations.

The demo system chooses to implement multiple linear table management.

in each cycle through the printf function , and op is initialized to 1 for entering the cycle for the first time. op=0 is the only exit condition of the loop, which is given in the menu . 1-14 respectively represent a basic operation of the linear table. After updating the value of op in each cycle , execute the corresponding function through the switch statement in the main function . After executing the function, break jumps out of the switch statement and starts the next while loop. Until the user enters 0 to select exit, exit the system. The system design structure is shown in Figure 2-1 .

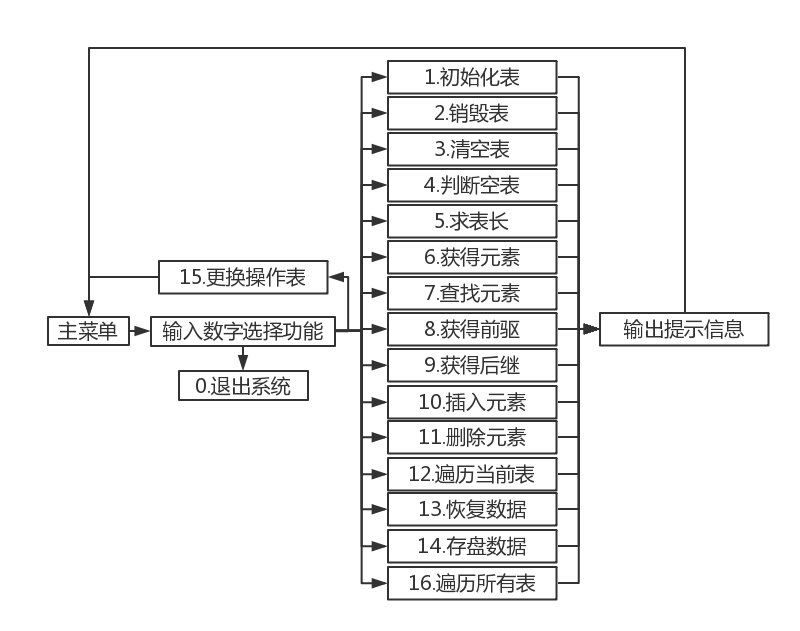


Figure 2-1 System design structure diagram

* + 1. Data file storage format

Considering the principle of simplicity and efficiency, the data file storage format is designed as follows.

File format: dat , which is convenient for programs to read data quickly .

The corresponding file space is allocated according to the corresponding data type in order for storage, and there is no gap between the data .

* + 1. Algorithm thinking and design

The idea and design of the linear table operation algorithm are as follows:

1. The idea of initializing the linear table: the parameter of the function is the address of the header type pointer variable L defined in the main function. In the function, first use the malloc function to allocate a memory space p as the header of the total table , and then connect an additional space as the small header of the first table . At this time, there is an empty table in the memory, that is, the next table of p The len member in the node is 0, the total table number length and the operation bit sequence vol are both changed to 1, and the initialization of the linear table is about to be completed.

2. The idea of destroying linear tables: At the beginning, it is necessary to judge the legality. The parameter of the function is the header type pointer variable L defined in the main function. If the current list has two or more elements, the elements are released in sequence from front to back until the header is directly connected to the only remaining element . After deleting this element , directly connect the predecessor of the header of the empty table to the successor. Note that since the header adopts a doubly linked list, the aft pointer of its predecessor and the fore pointer of the successor must be changed at the same time . When deleting the last table , the aft pointer of its predecessor is directly Point to NULL , and operate bit sequence vol minus 1 at the same time . The flowchart is shown in Figure 2-2 .

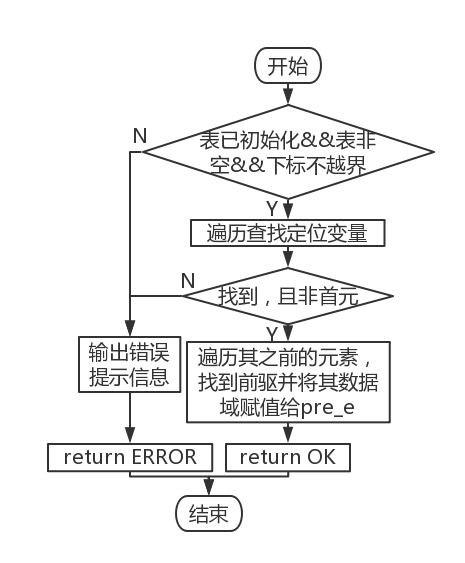
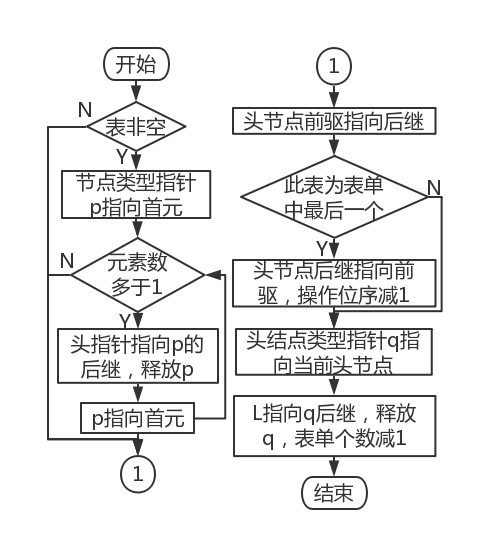


Figure 2-2 Implementation flow chart of destroying linear tables Figure 2-3 Implementation flow chart of obtaining precursors

3. The idea of clearing the linear table: At the beginning, it is necessary to judge the legality. The parameter of the function is the header type pointer variable L defined in the main function. When the table is not empty , clear the elements in the table in the same way as destroying the linear table .

4. The idea of judging whether it is an empty table : at the beginning, it is necessary to judge the legality. The parameter of the function is the header type pointer variable L defined in the main function. When calling this function, directly judge whether L- >head is NULL, if it is 0, it is empty, otherwise it is not empty .

5. The idea of finding the length of the linear table: at the beginning, it is necessary to judge the legality. The parameter of the function is the header type pointer variable L defined in the main function. In the function, directly return L -> len is the table length of the linear table to be obtained.

6. The algorithm idea of obtaining elements: at the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function, the serial number i of the data element and the variable e of the obtained element. Since the chain storage structure is adopted, it is necessary to find the node at the corresponding position through a loop and then access the data field of the node to obtain the target element .

7. Algorithmic thinking for finding elements: At the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function and the variable e for obtaining elements. Check whether the data field of each node in the linked list is equal to the given value by looping, and return the bit order of the element if they are equal.

8. Obtain the idea of the precursor algorithm: At the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function, the positioning variable cur and the variable p re\_e receiving the predecessor . First, use a loop to find the same element as the positioning variable cur in the linked list . If found , record the current position, and use the second loop to obtain the data field of its predecessor. The positioning variable is discussed separately when it is in the header. Assign it to pre\_e, otherwise return ERROR. The flowchart is shown in Figure 2-3 .

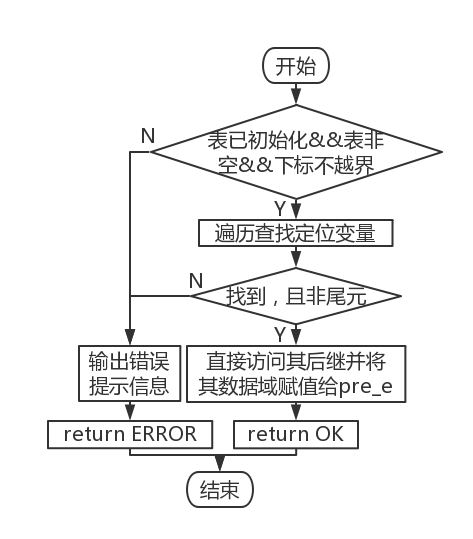


Figure 2-4 Get the follow-up implementation flow chart

9. Obtain the idea of the subsequent algorithm: at the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function, the positioning variable cur and the subsequent variable next \_e . First, use a loop to find the same element as the positioning variable cur in the linked list . If found , record the current position. At this time, you can directly access its subsequent data fields . Assign it to next\_e, and discuss it separately at the end of the table for positioning variables . Otherwise returns FALSE. The flowchart is shown in Figure 2-4 .

a

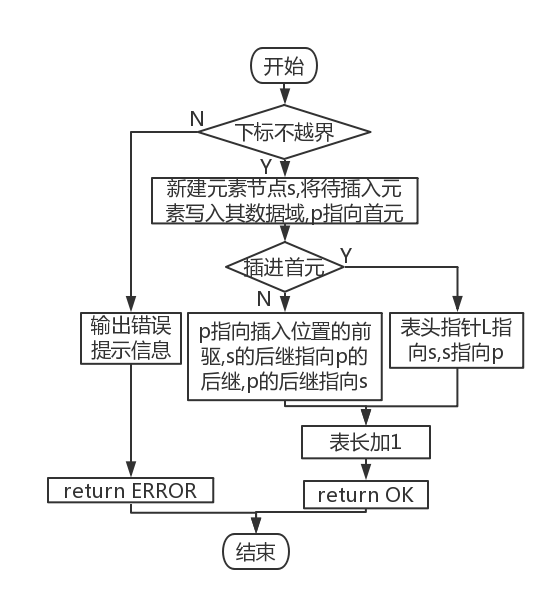


Figure 2-5 Insert element implementation flow chart

10. The algorithm idea of inserting elements: At the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function, the positioning variable i and the variable next \_e to be inserted . In the function, first judge whether the insertion position is in the linear table. When legal, create a new element node s, and insert variables as its data field. For the insertion of the head , directly let the table head point to the new node, and the new node points to the previous head. A loop makes the pointer point to the predecessor p of the target position, and inserts a new node by modifying the successor pointers of s and p . The flowchart is shown in Figure 2-5 .

11. Algorithm idea of deleting elements: At the beginning, it is necessary to judge the legality. The parameters of the function are the header type pointer variable L defined in the main function, the positioning variable i and the deleted variable next \_e . In the function, first judge whether the deletion position is in the linear table. When the position is legal, use a loop to locate its predecessor p , and direct the next pointer of the predecessor to the successor. For the deletion of the first element , directly let the table head point to the successor of the first element . Assign the data field of the deleted node to next\_e, and then directly delete the element after the element until the last element to move one unit from front to back. The flowchart is shown in Figure 2-6 .

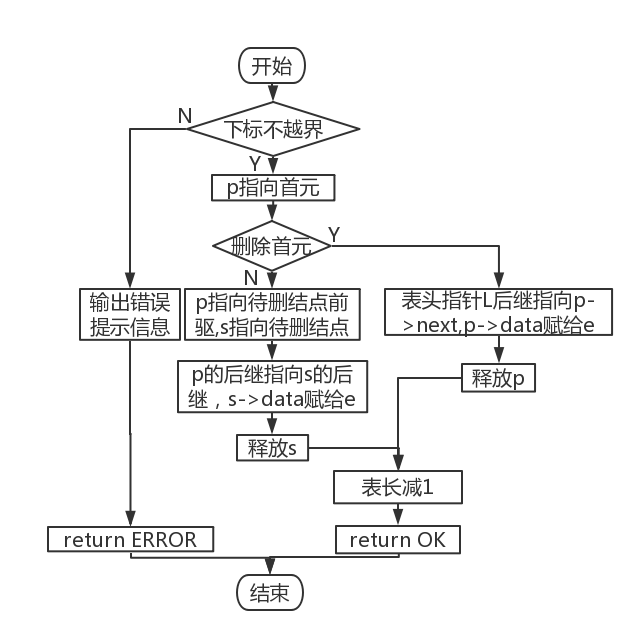


Figure 2-6 Flow chart of deleting elements

12. The idea of traversing the linear table algorithm: At the beginning, it is necessary to judge the legality. The parameter passed into the function is the header type pointer variable L defined in the main function. Starting from the head, use a loop to visit all nodes , and an error will be reported when the table is empty.

13. Data storage algorithm idea : At the beginning, it is necessary to judge the legality. The parameters of the function are the file pointer fp and the total header type pointer variable List . The file name is automatically generated from the system time , and the corresponding binary file is generated. First use a loop to count the number of non- empty tables , and return an error directly if there is no non-empty table. Save this data ; secondly, save the table length of each non-empty table in a cycle ; then, for each non-empty table, read each element in the table in a loop, and finally close the file. The flowchart is shown in Figure 2-7 .

14. Data recovery algorithm idea: At the beginning, it is necessary to judge the legality. The parameters of the function are the file pointer fp and the total header type pointer variable List . After successfully opening the file according to the file name, first read the number of valid tables, and initialize the same number of table header nodes with a cycle according to this number. For each node initialized, write the corresponding table length currently pointed by the file pointer into In the data field , according to the table length of each table , the same number of elements is read with a double loop , and the file is closed. The flowchart is shown in Figure 2-8 .

15. Algorithm idea of changing tables : At the beginning, it is necessary to judge the legality. The parameters of the function are the address & L of the header type pointer variable defined in the main function and the total header type pointer variable List . The provided operation range is from 1 to the table length plus 1, and you can choose to create a new table after the last table. If you choose this way , create a new table header directly after the table header and tail nodes, otherwise use a loop to locate the target table, and assign its location to the address of L.

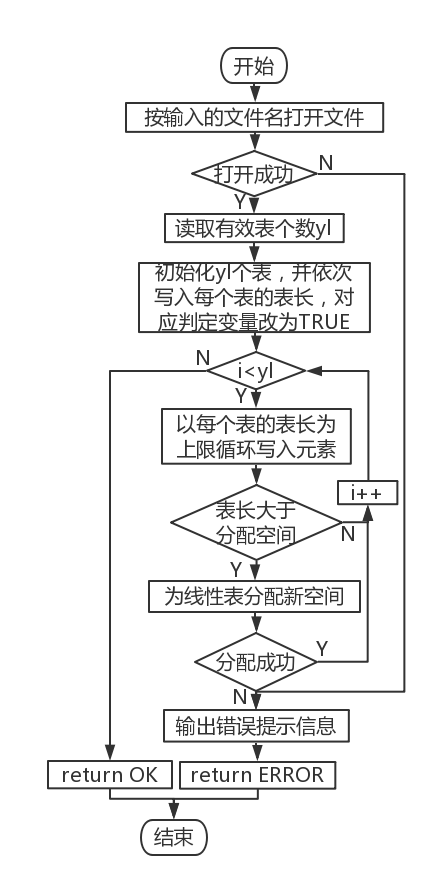
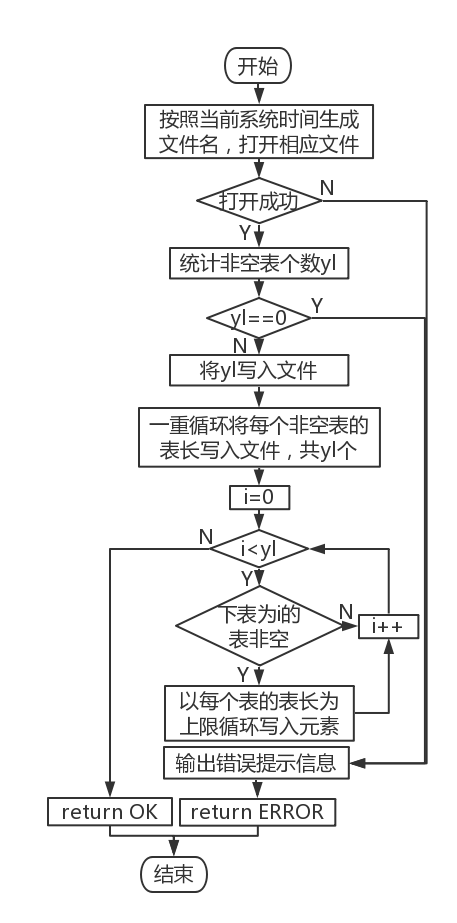


Figure 2-7 Data storage implementation flow chart Figure 2-8 Data Recovery Implementation Flowchart

16. Cross- table traversal algorithm idea : At the beginning, it is necessary to judge the legality. The parameter of the function is the total header type pointer variable List . Loop from the first table , if the current table is uninitialized or empty, then output the corresponding prompt information, otherwise the second loop outputs all the elements in the table.

* + 1. Algorithm Time and Space Complexity Analysis

each function is roughly the same as that in Experiment 1, but due to the physical structure of the linked list, it cannot be quickly located by element subscripts, and the time complexity of a small number of functions increases from O(1) to O( n ) . See table below .

Table 2-1 Algorithm time and space complexity analysis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function name | time complexity | space complexity | Function name | time complexity | space complexity |
| IntiaList | O(1) | O(1) | NextElem | O(n) | O(1) |
| DestroyList | O(n) | O(1) | ListInsert | O(n) | O(1) |
| ClearList | O(n) | O(1) | ListDelete | O(n) | O(1) |
| ListEmpty | O(1) | O(1) | List Trabverse | O(n) | O(1) |
| ListLength | O(1) | O(1) | LoadList | O(n 2 ) | O(1) |
| GetElem | O(n) | O(1) | SaveList | O(n 2 ) | O(1) |
| LocateElem | O(n) | O(1) | ChooseList | O(1) | O(1) |
| PriorElem | O(n 2 ) | O(1) | Chart Traverse | O(n 2 ) | O(1) |

## 2.3 System Implementation

1. * 1. Programming environment , operating environment , project engineering description

This experiment is written with Codeblocks programming software, and compiled and run with Codeblocks . The project name is Node-based Linear Chart (Cpp) . The demo system menu interface is shown in Figure 2-9



Figure 2-9 demonstrates the system menu interface

* + 1. header files and predefined constants

1. Header file

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

2. Predefined constants

#define OK 1

#define TRUE 1

#define FALSE -1

#define ERROR -2

3. Type expressions

typedef int status;

typedef int ElemType;

typedef struct lnode{

ElemType data;

struct lnode \*next;

}LNode;

typedef struct llist{

int len;

struct lnode \*head, \*tail;

struct llist \*fore, \*aft;

}Linklist;

* + 1. Test Plan

The linked list operation function of the system is tested according to the following plan:

1. Detect the working conditions of the system under normal conditions;
2. Focus on checking the working conditions during illegal border operations.

Table 2-2 Demonstration system test plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function to be tested | test sequence | test input | expected output | Linear table status |
| 1. IntiaList | 1 | none | Initialize L successfully | table length 0 , linear table is created |
| 2. Destroy List | 10 | none | Destroyed successfully | does not exist |
| 3. ClearList | 9 | none | Table cleared successfully | empty table |
| 4. ListEmpty | called multiple times | none | L is displayed as uninitialized when it points to an uninitialized table , as an empty table when it points to an empty table, and as a non-empty table when it points to a non - empty table | as the previous step |
| 5. List Length | called multiple times | none | L points to an uninitialized table, it is displayed as uninitialized, and when it points to an initialized table, it displays the table length | as the previous step |
| 6. GetElem | 4 | Enter 1 and 8 respectively | The first input should output 7362 , the second input subscript is out of bounds and cannot be found | same as step 4 |
| 7. LocateElem | 5 | Enter 6363 and 53463 respectively | The first input should output 2 , the second cannot be found | same as step 5 |
| 8. PriorElem | 6 | Enter 8585, 7362 and 8614 respectively | The first input should output 6363 , the second input has no predecessor, the lookup fails, and the third input cannot be found | Same as step 6 |
| 9. NextElem | 7 | Enter 7362, 8585 and 7361 respectively | The first input should output 6363 , the second input has no successor, the lookup fails , and the third input cannot be found | same as step 7 |
| 10. ListInsert | 3 | Enter 1, 6473,; 2, 7144; 3, 6369 and -52, 2 respectively | For the first three inputs, it shows that the insertion is successful; for the last input, it should prompt that the subscript is out of bounds, and the insertion fails | as the data corresponding to the subscript within a reasonable range is updated |
| 11. ListDelete | 8 | Enter 3 and 5 respectively | For the first input , it should output that the deletion is successful , and the deleted element is 8585 ; for the second input, it should prompt that the subscript is out of bounds, and the deletion failed | After deleting the third element 8585, the remaining elements are rearranged in order |
| 12. List Trabverse | called multiple times | none | If there are elements in the table , print all its elements in a loop; if the table is an empty table , the output prompts that the table is empty; if the table is not initialized , the output prompts that the table is not initialized | as the previous step |
| 13. LoadList | 2, 12 | binary file name | If the binary file is opened successfully , each table stored in the file should be output . If the corresponding file is not found or the file fails to be opened, an error message will be output. | (point to the first table by default ) When opening the binary file successfully, it should be the first table stored in the binary file , and remain uninitialized when opening the file fails |
| 14. SaveList | 11 | none | new binary data file under the source file directory , the file name is the time of the save operation, and the format is "week\_month abbreviation\_date\_hour\_minute\_second\_year " | Same as step 11 |
| 15. Choose List | 3 | 20 | L points to the last table in memory | Depending on the state of the last table itself, it may be uninitialized, empty, or non-empty |
| 16. Chart Traverse | called multiple times | none | For a table with elements , loop to print all its elements; for an empty table, the output prompts that the table is empty; for an uninitialized table, the output prompts that the table is not initialized | as the previous step |

* + 1. test

1. Initially there are no tables in the system .



Figure 2-10 Before importing data



Figure 2-11 There is no table in the current memory

2. At this time , any form of access to the table (function 2-12 ) will prompt that the table is not initialized . Examples are as follows :



Figure 2-12 DestroyList error message ( not initialized )



Figure 2-13 \_ ClearList error message ( not initialized )



Figure 2-14 \_ ClearList error message ( not initialized )

3. Initialize the form.



Figure 2-15 Enter a file name , import data



Figure 2-16 \_ There is 1 table in the current system

4 . At this time , any form of access to the table (function 2-12 ) will prompt that the table is empty. Examples are as follows :

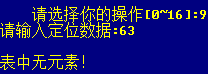


Figure 2-1 7 NextElem error message ( empty table )

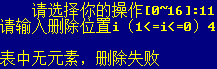


Figure 2-1 8 ListDelete error message ( empty list )



Figure 2-1 9 ListTrabverse error prompt ( empty table )

5. Import data by entering the file name . At this time, there are 4 tables in the system .

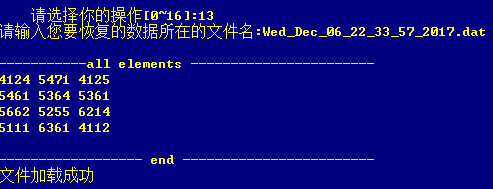


Figure 2-20 \_ Enter the file name , import data successfully



Figure 2-21 Import data successfully

6. Add a new table at the end of the form , namely List [5]. It can be found that the output prompt matches the information when traversing the entire table .



Figure 2-22 change action sheet



Figure 2-23 \_ Unified menu prompts for follow-up operations

7. At this point the table length is 0.



Figure 2-24 \_ Judgment table length

8. Elements 6473 , 7144 , 6369 are inserted successively.

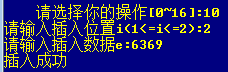


Figure 2-25 \_ element inserted successfully

9. Insertion fails when the subscript is out of bounds .

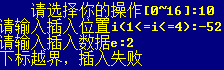


Figure 2-26 \_ \_ Failed to insert element

10. Traverse the current table. At this time , there are 3 elements in the current table .



Figure 2-27 \_ \_ iterate over the current table



Figure 2-28 \_ \_ Ask for table length

11. Get the first element .



Figure 2-29 \_ \_ get element success



Figure 2-30 \_ \_ Failed to get element

12. Find the elements in the current table .



Figure 2-31 \_ \_ Find element succeeded



Figure 2-32 \_ \_ Failed to find element

13. Get the elemental precursor.



Figure 2-33 \_ \_ Get element precursor success



Figure 2-34 \_ \_ Get the head precursor



Figure 2-35 \_ \_ element not found

14. Get the element successor .



Figure 2-36 Get the successor of the element



Figure 2-37 Get tail element successor

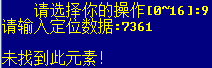


Figure 2-38 element not found

15. Delete elements .



Figure 2-39 \_ \_ Delete element successfully



Figure 2-40 \_ \_ Failed to delete element ( subscript out of bounds )

16. Traverse the current table again and find that 8585 disappears .



Figure 2-41 \_ \_ Traversing the current table to check for deleted elements

17. Clear the current table.



Figure 2-42 \_ \_ clear current table

18. Check for emptying.



Figure 2-43 \_ \_ The ListEmpty function checks for emptying

19. Delete this table.



Figure 2-44 \_ \_ delete current table

20. Review table deletions.



Figure 2-45 \_ \_ The ListEmpty function checks for deletions



Figure 2-46 \_ \_ ListLength function checks for deletions



Figure 2-4 7 LocateElem function checks deletion \_

21. Create the 5th , 7th , 9th , 10th, and 12th tables, insert some elements into the 6th, 8th, and 11th tables , and traverse all the tables.



Figure 2-48 \_ \_ Iterate over all tables after some input

twenty two. Save the modified table as a binary file

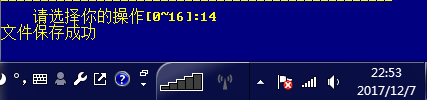


Figure 2-49 \_ \_ File saved successfully



Figure 2 -50 saved files

twenty three. Read the file just saved to the system.

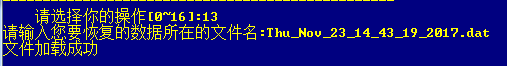


Figure 2-51 \_ \_ read file successfully

twenty four. Re-traversing all the tables shows that only initialization is performed , and tables with no elements and uninitialized tables are skipped when saving to disk, resulting in intensive table storage in the new form .

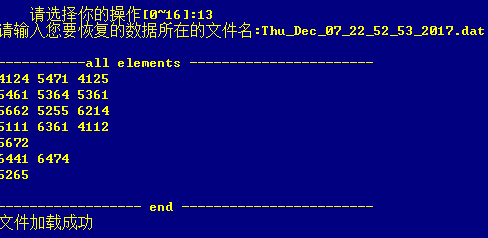


Figure 2-52 \_ \_ traverse all tables

25. When opening a file with the wrong format , it will display that the file failed to open.



Figure 2-53 \_ \_ target file is not readable

26. Log out of the system .

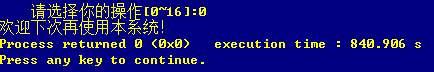


Figure 2-54 \_ \_ Exit system

## 2.4 Experimental summary

After this experiment, I fully understood the physical structure of the linked list, and mastered the basic operations of the linked list through personal experience, and improved my ability to write codes related to the linked list, especially in the process of writing I encountered many difficulties in the process, and finally solved it after seeking the help of my classmates many times. Through the multi-table operation part of this experiment , I reviewed the structure of the cross-linked list, and used the double-linked list for the first time in practice . Through the same long time of writing code and debugging , I understand that I still need to practice more after class, so that I will not feel unfamiliar with some basic sentences during the experiment .

# 3 Implementation of Binary Tree Based on Binary Linked List

## 3.1 Problem description

* + 1. Basic concept of binary tree

A binary tree is a tree structure in which each node has at most two subtrees. Usually subtrees are called "left subtree" and "right subtree". Binary trees are often used to implement binary search trees and binary heaps.

Each node of a binary tree has at most two subtrees (there is no node with a degree greater than 2). The subtrees of a binary tree are divided into left and right, and the order cannot be reversed. The i-th layer of the binary tree has at most 2^{i-1} nodes; the binary tree with a depth of k has at most 2^k-1 nodes; for any binary tree T, if the number of terminal nodes is n\_0, the degree The number of nodes is 2 is n\_2, then n\_0=n\_2+1 . The storage structure of the binary tree is divided into linear storage and chain storage.

* + 1. Logical structure and basic operations

The abstract data type binary tree is defined as follows:

ADT BinaryTree {

Data Object D: D is a collection of data elements with the same properties.

Data relation R:

If D=Φ, then R=Φ, BinaryTree is called an empty binary tree;

If D≠Φ, then R ={H}, H is the following binary relationship:

1. There is a unique root data element root in D, which has no predecessor in relation H;
2. If D- {root} ≠Φ, then there exists D- {root}={D1,Dr}, and D1∩Dr = Φ;
3. If D1≠Φ, then there is a unique element X1 in D1, <root,X1>∈ H , and the relationship H1 on D1 is included in H; if D r ≠Φ, then there is a unique element Xr in Dr, <root , Xr>∈H, and the relationship on D r belongs to H;
4. (D, {H1}) is a binary tree conforming to this definition, called the left subtree of the root, (D r , {H r}) is a binary tree conforming to this definition, called the right subtree of the root.

}

of combining minimum and minimum completeness and common use, 20 basic operations are defined in the form of functions, such as initialization of binary tree, destruction of binary tree, creation of binary tree, emptying of binary tree, determination of empty binary tree, and calculation of binary tree depth. The specific operation functions are defined as follows :

1. Initialize the binary tree: the function name is InitBiTree(T) ; the initial condition is that the binary tree T does not exist; the operation result is to construct an empty binary tree T.

2. Destroy the binary tree: the name of the tree function is DestroyBiTree(T) ; the initial condition is that the binary tree T already exists; the operation result is to destroy the binary tree T.

3. Create a binary tree: the function name is CreateBiTree(T, definition) ; the initial condition is that the definition gives the definition of the binary tree T ; the operation result is to construct the binary tree T according to the definition .

4. Clear the binary tree: the function name is ClearBiTree (T) ; the initial condition is that the binary tree T exists ; the operation result is to clear the binary tree T.

5. Determine the empty binary tree: the function name is BiTreeEmpty(T) ; the initial condition is that the binary tree T exists ; the operation result is to return TRUE if T is an empty binary tree , otherwise return FALSE .

6. Find the depth of the binary tree: the function name is BiTreeDepth(T) ; the initial condition is that the binary tree T exists ; the operation result is to return the depth of T.

7. Get the root node: the function name is Root (T) ; the initial condition is that the binary tree T already exists; the operation result is to return the root of T.

8. Get the node: the function name is Value(T , e) ; the initial condition is that the binary tree T already exists, and e is a certain node in T ; the operation result is to return the value of e .

9. Node assignment: the function name is Assign(T,&e , value) ; the initial condition is that the binary tree T already exists, and e is a node in T ; the operation result is that node e is assigned value .

10. Get the parent node: the function name is Parent(T , e) ; the initial condition is that the binary tree T already exists, and e is a node in T ; the operation result is if e is a non-root node of T , then return Its parent node pointer, otherwise NULL is returned .

1 1. Get the left child node: the function name is LeftChild(T , e) ; the initial condition is that the binary tree T exists, and e is a node in T ; the operation result is to return the left child node pointer of e . Returns NULL if e has no left child .

12. Get the right child node: the function name is RightChild(T ,e ) ; the initial condition is that the binary tree T already exists, and e is a node in T ; the operation result is to return the right child node pointer of e . Returns NULL if e has no right child .

13. Get the left sibling node: the function name is LeftSibling(T , e) ; the initial condition is that the binary tree T exists, and e is a node in T ; the operation result is to return the left sibling node pointer of e . Returns NULL if e is a left child of T or has no left sibling .

1 4. Get the right sibling node: the function name is RightSibling(T , e) ; the initial condition is that the binary tree T already exists, and e is a node in T ; the operation result is to return the right sibling node pointer of e . Returns NULL if e is the right child of T or has no siblings .

1 5. Insert subtree: the function name is InsertChild(T , p , LR , c) ; the initial condition is that the binary tree T exists, p points to a node in T , LR is 0 or 1 , , the non-empty binary tree c and T is disjoint and the right subtree is empty; the operation result is based on LR being 0 or 1 , insert c as the left or right subtree of the node pointed to by p in T , and the original left or right subtree of the node pointed to by p The subtree is the right subtree of c .

1 6. Delete subtree: the function name is DeleteChild(T . p . LR) ; the initial condition is that the binary tree T exists, p points to a certain node in T , and LR is 0 or 1 . The result of the operation is to delete c as the left or right subtree of the node pointed to by p in T according to LR being 0 or 1 .

1 7. Preorder traversal: the function name is PreOrderTraverse(T , Visit()) ; the initial condition is that the binary tree T exists, and Visit is an application function for node operations; the operation result: preorder traversal t , calling for each node The function Visit once and once, once the call fails, the operation fails.

1 8. Inorder traversal: the function name is InOrderTraverse(T , Visit)) ; the initial condition is that the binary tree T exists, and Visit is an application function for node operations; the operation result is inorder traversal t , and the function is called for each node Visit once and once, once the call fails, the operation fails.

1 9. Postorder traversal: the function name is PostOrderTraverse(T , Visit)) ; the initial condition is that the binary tree T exists, and Visit is an application function for node operations; the operation result is postorder traversal t , and the function is called for each node Visit once and once, once the call fails, the operation fails.

20. Traverse by level: the function name is LevelOrderTraverse(T , Visit)) ; the initial condition is that the binary tree T exists , and Visit is an application function for node operations; the operation result is level order traversal t , and the function is called for each node Visit once and once, once the call fails, the operation fails.

## 3.2 System design

* + 1. data physical structure

The data physical structure of the binary tree is as follows :

typedef struct BiTNode

{

ElemType key; // node number

char data; // node name

struct BiTNode \*lchild, \*rchild;//left and right subtree pointers

}BiTNode, \*BitTree;//typedef is to alias the structure type definition struct BiTNode to BiTNode

In this program, the data atomic type ElemType is defined as an int integer.

* + 1. demo system

The binary linked list is used as the physical structure of the binary tree to realize the basic operation of § 3.1.2 . Among them, T ElemType is the type name of the data element, and the specific meaning can be defined by oneself. For the definition and reference of other related types and constants, please refer to p127 of the literature [1] .

It is required to construct a functional demonstration system with menus. Among them, in the main program, the preparation of the actual parameter value required for the function call and the display of the function execution result are completed, and an appropriate operation prompt is given.

The demonstration system can choose to save the binary tree in file form. Among them, ① it is necessary to design the file data record format to efficiently save the complete information of the binary tree data logical structure (D, {R}) ; ② it is necessary to design a reasonable mode of binary tree file saving and loading operations. Appendix B provides methods for file access.

in each cycle through the printf function , and op is initialized to 1 for entering the cycle for the first time. op=0 is the only exit condition of the loop, which is given in the menu . 1-22 respectively represent a basic operation of the linear table. After updating the value of op in each round of loop , execute the corresponding function through the switch statement in the main function . After executing this function, break jumps out of the switch statement and starts the next round of while loop. Until the user enters 0 to select exit, exit the system. The system design structure is shown in Figure 3-1 .

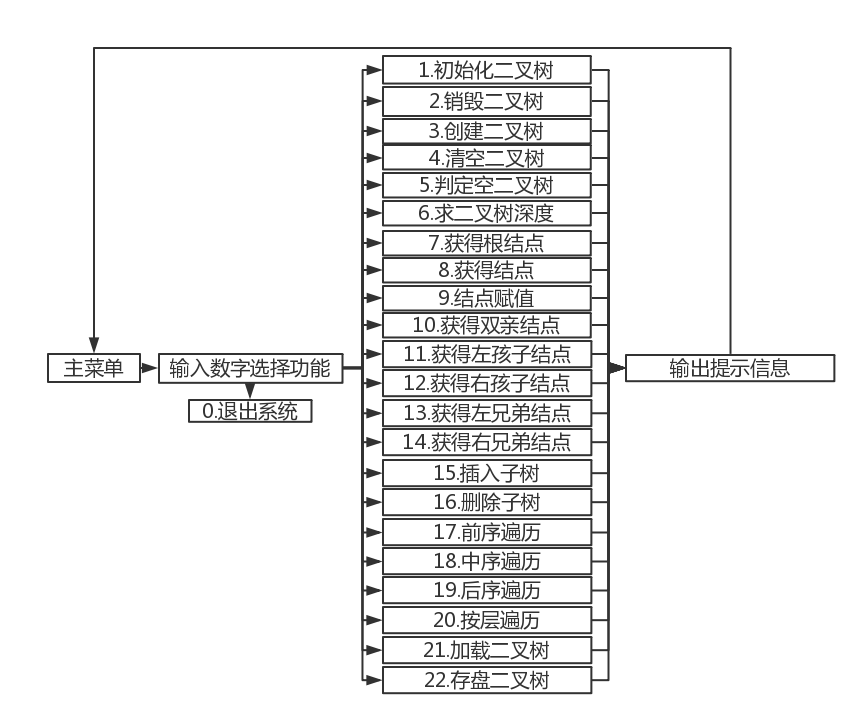


Figure 3-1 System design structure diagram

* + 1. Data file storage format

Considering the principle of simplicity and efficiency, the data file storage format is designed as follows.

File format: dat , which is convenient for programs to read data quickly .

The corresponding file space is allocated according to the corresponding data type in order for storage, and there is no gap between the data .

* + 1. Algorithm thinking and design

The idea and design of the binary tree operation algorithm are as follows:

1. The idea of initializing the binary tree algorithm: At the beginning, it is necessary to judge the legality (whether L is initialized). The parameter of the function is the pointer L of the tree root node . Allocate 1 node space for L, its key is set to -1 to mark an empty tree , and the left and right child pointers are set to NULL.

2. The idea of destroying the binary tree algorithm: At the beginning, it is necessary to judge the legality (whether L is initialized). The parameter of the function is the pointer L of the tree root node. The left subtree of the binary tree is destroyed first, and then the right subtree of the binary tree is destroyed in a recursive manner. Finally, the memory space corresponding to the current node is released by the free function.

3. The idea of creating a binary tree algorithm: At the beginning, it is necessary to judge the legality (whether L is initialized). The parameter of the function is the tree root node pointer L , input the value of the node in the binary tree according to the order of the first order, if the first character is #, then T is an empty binary tree. Otherwise, the malloc function allocates a unit of space as the root node of the tree, assigns a value to it, and continues to create the left subtree and right subtree of the root node in a recursive manner. The flowchart is shown in Figure 3-2 .

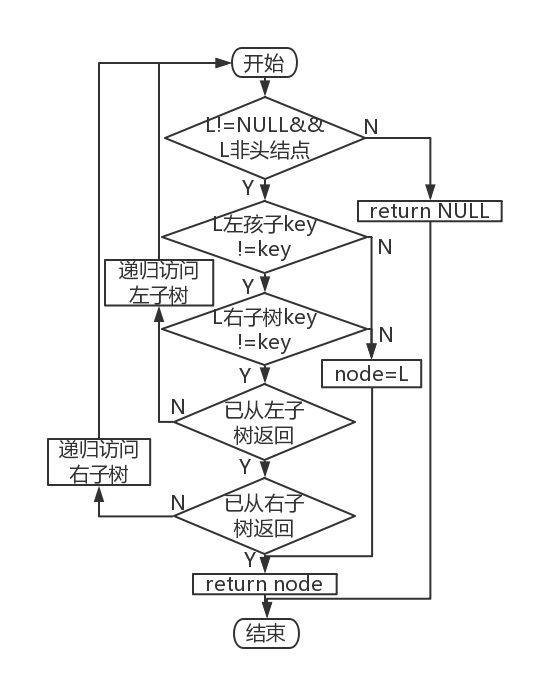
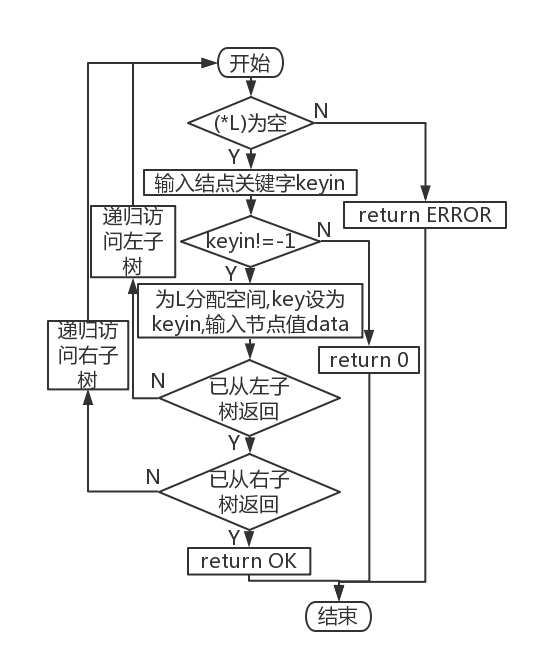


Figure 3-2 Create a binary tree flow chart Figure 3-3 Flowchart of Obtaining Parent Nodes

4. The idea of clearing the binary tree algorithm: the parameter of the function is the pointer L of the tree root node . Destroy the binary tree pointed to by L , and return ERROR if the destruction fails ; initialize a new empty binary tree with L as the root node , and return ERROR if the initialization fails.

5. The idea of determining the empty binary tree algorithm: the parameter of the function is the pointer L of the tree root node . Returns TRUE if the key of L is -1 , otherwise returns FALSE.

6. The idea of binary depth algorithm: the parameter of the function is the pointer L of the tree root node . Find the depth of the binary tree in a recursive way, first recursively visit its left subtree and return the depth of the left subtree , then recursively visit its right subtree and return the depth of the right subtree . The current node depth is the larger one of the left and right word depths plus 1. If the left and right subtrees are empty , the depth of the current node is 0. If the left and right subtrees have been visited or are empty, the upper level function returns the current depth.

7. Obtain the idea of binary tree root node algorithm: the parameter of the function is the pointer L of the tree root node . Return directly to the root node.

8. Get the idea of the node algorithm: the parameters of the function are the tree root node pointer L and the keyword key to be searched, pass the two parameters into the Search function and return the pointer M, if M is not empty , return the name of the node pointed to by M , otherwise return 0.

9. The idea of the node assignment algorithm: the parameter of the function is the address of the tree root node pointer L, the key word of the node to be modified and the new name data, pass L and key into the Search function and return the pointer M, if M is not empty then Change M to change its name to M , otherwise return 0.

10. Obtain the idea of the parent node algorithm: the parameter of the function is the tree root node pointer L, compare the keyword key and the address of the tree node node. If the keyword of L's left child or right child matches key , assign L to node and return , otherwise, first recursively visit its left child, and then recursively visit its right child . Returns NULL when visiting a leaf node . The flowchart is shown in Figure 3-3 .

11. The idea of obtaining the left child node algorithm: the parameter of the function is the tree root node pointer L, compare the keyword key and the address of the tree node node. If the keyword of L matches key , assign it to node and return if the left child of L exists , or return NULL if it does not exist , otherwise, first recursively visit its left child, and then recursively visit its right child . Returns NULL when visiting a leaf node . The flowchart is shown in Figure 3-4 .

12. The idea of obtaining the right child node algorithm is the same as above.

13. The idea of obtaining the algorithm of the left brother node: the parameter of the function is the tree root node pointer L, compare the keyword key and the address of the tree node node. If the keyword of the right child of L matches the key , assign it to node and return if the left child of L exists , and return NULL if it does not exist . Otherwise, first recursively visit its left child, and then recursively visit its right child . Returns NULL when visiting a leaf node .

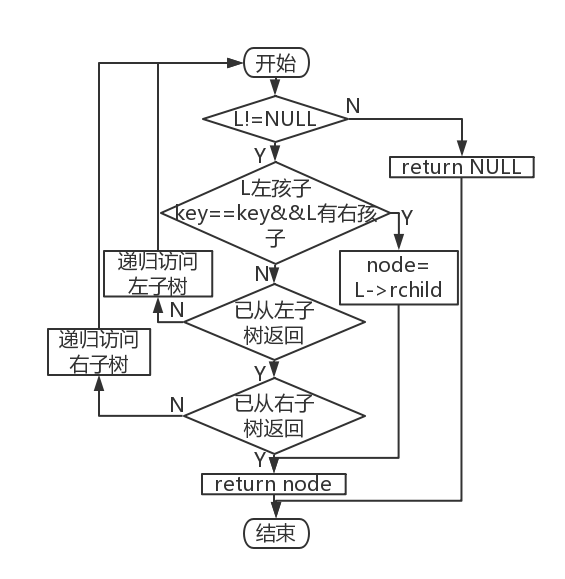
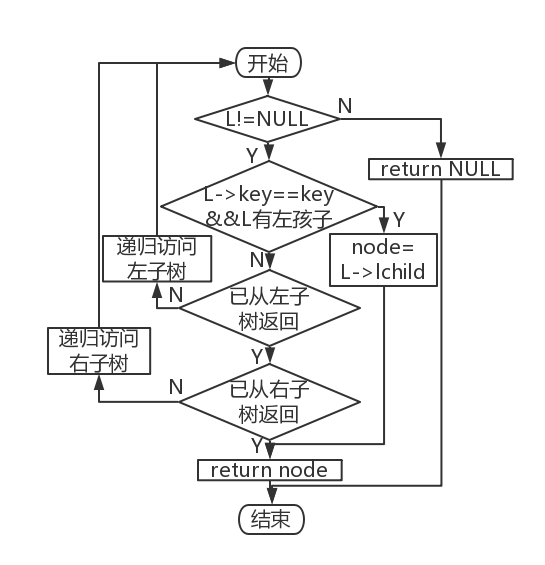


Figure 3-4 Get the left child flow chart Figure 3-5 Obtaining the Right Brother Flowchart

14 . The idea of obtaining the right brother node algorithm is the same as above. The flowchart is shown in Figure 3-5 .

15. The idea of inserting subtree algorithm: The parameters of the function are the address of the tree root node pointer L, the target binary tree node pointer p, the location variable LR and the new tree root node pointer c. When the left subtree is inserted , if the left subtree is not empty, return ERROR, otherwise, assign c to its left subtree; when inserting the right subtree , if the left subtree is not empty, return ERROR, otherwise, assign c to it right subtree .

16. Delete subtree algorithm idea: The parameters of the function are the address of the tree root node pointer L, the target binary tree node pointer p, and the positioning variable LR. When destroying the left subtree , if the left subtree is empty, return FALSE, otherwise call destroy The binary tree function destroys its left subtree and sets the left child pointer to NULL ; when destroying the right subtree , if the right subtree is empty, returns FALSE, otherwise calls the destroy binary tree function to destroy its right subtree and sets the right child pointer to NULL . The flowchart is shown in Figure 3-6 .

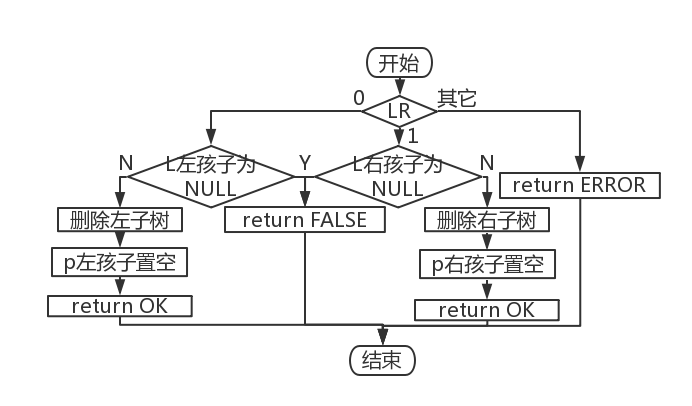


Figure 3-6 Flow chart of deleting subtrees

17. The idea of the preorder traversal algorithm: the parameter of the function is the tree root node pointer L. If L is empty , return 0 to the upper level function , otherwise print its node information . If its left child is not empty , recursively access its left child ; if its right child is not empty , recursively access its right child ;

18. Inorder traversal algorithm idea: The parameter of the function is the tree root node pointer L. If L is empty , return 0 to the upper level function . If its left child is not empty , recursively visit its left child . Print its node information . If its right child is non-null , recursively visit its right child .

19. Post-order traversal algorithm idea: The parameter of the function is the tree root node pointer L. If L is empty , return 0 to the upper level function . If its left child is not empty , recursively access its left child ; if its right child is not empty , recursively access its right child . Finally print its node information .

20. Thoughts of layer sequence traversal algorithm: The parameter of the function is the tree root node pointer L. Create a long enough tree node structure array , let the root node of the tree be the head. Define two subscripts front and rear . When q[front ] has a left child , it is assigned to q[rear ] , and rear is incremented; when q[front ] has a right child, it is assigned to q[rear ] , and rear is incremented. front increments automatically . Repeat the above process until no new nodes in the tree are added to the array. The flowchart is shown in Figure 3-7 .

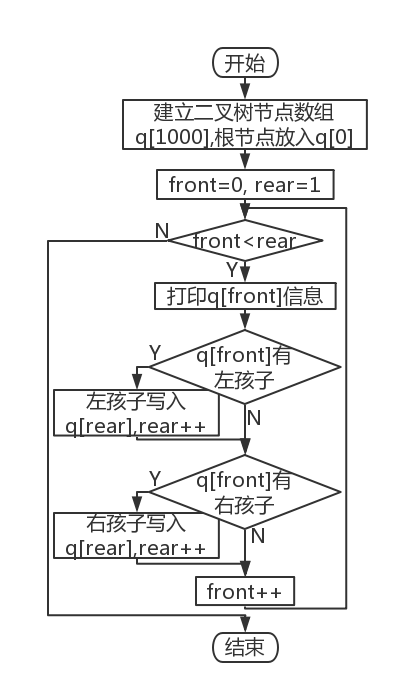


Figure 3-7 Layer order traversal flow chart

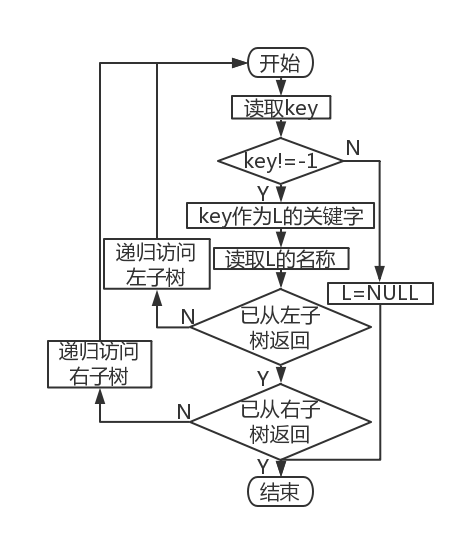
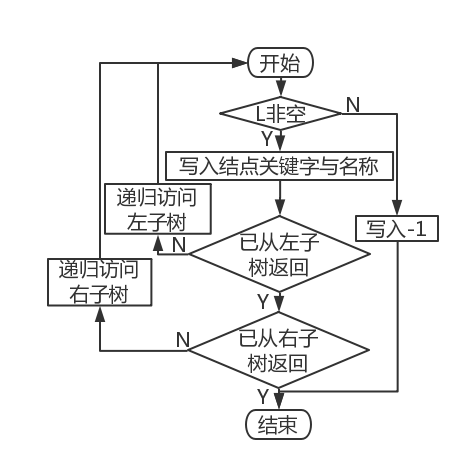


Figure 3-8 Data storage flow chart Figure 3-9 Flow chart of data reading

twenty one. Data storage algorithm idea : The parameters of the function are the tree root node pointer L and the file pointer fp. If L is empty, write -1 to the file , otherwise write the node key and name. Recursively visit its left child , then recursively visit its right child , and return OK to the upper level after all are completed . The flowchart is shown in Figure 3-8 .

twenty two. Data recovery algorithm idea: The parameters of the function are the tree root node pointer L and the file pointer fp. Read the key of the current node , if the key is -1, the current node will be empty , otherwise create the current node and write its key and name. Recursively visit its left child , then recursively visit its right child , and return OK to the upper level after all are completed . The flowchart is shown in Figure 3-9 .

* + 1. Algorithm Time and Space Complexity Analysis

Table 3-1 Algorithm time and space complexity analysis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function name | time complexity | space complexity | Function name | time complexity | space complexity |
| InitBiTree | O(1) | O(1) | Right Child | O(n) | O(1) |
| DestroyBiTree | O(n) | O(1) | Left Sibling | O(n) | O(1) |
| CreateBiTree | O(n) | O(1) | Right Sibling | O(n) | O(1) |
| ClearBiTree | O(n) | O(1) | Insert Child | O( 1 ) | O(1) |
| BiTreeEmpty | O(1) | O(1) | Delete Child | O(n) | O(1) |
| BiTreeDepth | O(n) | O(1) | PreOrderTraverse | O(n) | O(1) |
| root | O( 1 ) | O(1) | In Order Traverse | O( n ) | O(1) |
| value | O(n) | O(1) | PostOrderTraverse | O(n) | O(1) |
| assign | O(n) | O(1) | LevelOrderTraverse | O(n) | O(1) |
| parent | O(n) | O(1) | LoadBiTree | O(n) | O(1) |
| LeftChild | O(n) | O(1) | SaveBiTree | O(n) | O(1) |

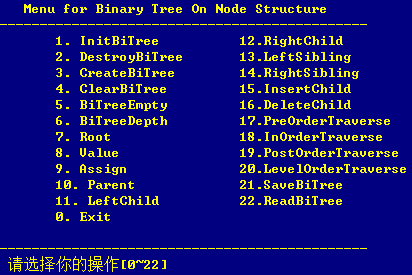


Figure 3-10 \_ \_ Demonstration system menu interface

## 3.3 System Realization

* + 1. Programming environment , operating environment , project engineering description

This experiment is written with Codeblocks programming software and compiled and run with Codeblocks . The project name is Binary-Tree-Sample . The demo system menu interface is shown in Figure 3-10 .

* + 1. header files and predefined constants

1. Header file

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

2. Predefined constants

#define OK 1

#define TRUE 1

#define FALSE -1

#define ERROR -2

3. Type expressions

typedef int Status; // is the type of the function, its value is the function result status code

typedef int ElemType; //Data element type definition

typedef struct BiTNode

{

ElemType key;

char data;

struct BiTNode \*lchild, \*rchild;//left and right subtree pointers

}BiTNode, \*BitTree;

* + 1. Test Plan

The binary tree operation function of the system is tested according to the following plan:

1. Detect the working conditions of the system under normal conditions;
2. Focus on checking the working conditions during illegal border operations

Table 3-2 Demonstration System Test Plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function to be tested | test sequence | test input | expected output | Binary tree state |
| 1. InitBiTree | 1 | none | Initialize L successfully | Tree depth 0, binary tree is created |
| 2. DestroyBiTree | twenty three | none | Tree destroyed successfully | does not exist |
| 3. CreateBiTree | 9 | none | Table cleared successfully | empty table |
| 4. ClearBiTree | twenty one | none | Tree cleared successfully | empty tree |
| 5. BiTreeEmpty | called multiple times | none | L is an uninitialized table, it is displayed as uninitialized, when it is empty , it is displayed as empty , and when it is not empty, it is displayed as non-empty | as the previous step |
| 6. Bi TreeDepth | 5 | none | Binary tree depth is 5 | as step 4 |
| 7. Root | 9 | none | of the root of the binary tree is 1 | as step 4 |
| 8. value | 10 | Enter 6 and 12 respectively | The first input should output data as f, and the second input and output search fails | as step 4 |
| 9. assign | 11 | Enter 2 , @ and 85,23 respectively | The first input should output assignment success, and the second input and output lookup failed | as step 4 |
| 10. Parent | 12 | Enter 7 , 1 and 76 respectively | The parent key of the first input and output is 5 , the second input and output has no parents , and the return fails; the target node is not found for the last input and output | as step 4 |
| 11. Left Child | 13 | Enter 2, 6 and 63 respectively | The first input and output left child key is 3 , the second input and output has no left child , and the return fails; the last input and output cannot find the target node | as step 4 |
| 12. RightChild | 1 4 | Enter 1, 4 and 51 respectively | the first input and output is 4 , the second input and output has no right child , and the return fails; the last input and output cannot find the target node | as step 4 |
| 13. Left Sibling | 15 | Enter 4, 1 and 73 respectively | the first input and output is 2 , the second input and output has no left sibling , and the return fails; the target node is not found for the last input and output | as step 4 |
| 14. Right Sibling | 16 | Enter 6, 8 and 6 9 respectively | the first input and output is 7 , the second input and output has no right sibling , and the return fails; the last input and output cannot find the target node | as step 4 |
| 15. Insert Child | 4, 17, 19 | Input one : 5-e;6-f;-1;-1;7-g;8-h;-1;-1;4;0  Input two: -1 ; 2; -1  Input three : 9 —1 ; -1 ; -1 ; 1; 1  Input four : 9 -); - 1 ; 10 -=; -1 ; -1 ; 2 ; 1 | For the first and fourth inputs , it should show that the insertion was successful; the last two inputs both show that the insertion failed | After inputting one, it is the binary tree in Figure 3- ; inputting two and three are the same as in step 4 ; after inputting four, it is the binary tree in Figure 3- |
| 16. Delete Child | 1 8 | Enter 7-0 , 2-8 and 9 respectively | For the first input and output, the deletion is successful ; the last two inputs show that the deletion failed | in Figure 3 - remove the state of node 8 -h |
| 17. PreOrder Traverse | 3 | none | The key and data of each node under the preorder | as step 2 |
| 18. In Order Traverse | 6 | none | The key and data of each node in the middle order | as step 4 |
| 19. PostOrderTraverse | 7 | none | The key and data of each node in the subsequent order | as step 4 |
| 20. Level Traverse | 8 | none | The key and data of each node under the sequence | as step 4 |
| twenty one. SaveBiTree | 20 | none | new binary data file under the source file directory , the file name is the time of the save operation, and the format is "week\_month abbreviation\_date\_hour\_minute\_second\_year " | Same as step 19 |
| twenty two. ReadBiTree | 2, 22 | binary file name | If the binary file is opened successfully , each table stored in the file should be output . If the corresponding file is not found or the file fails to be opened, an error message will be output. | If the import is successful , it will be the binary tree in Figure 3 -/3- , if it fails, only one node will be empty |

* + 1. test

1. Initially there are no tables in the system . At this time , any form of access to the table (function 2-22 ) will prompt that the table is not initialized . Examples are as follows :



Figure 3-11 DestroyBiTree error message ( not initialized )



Figure 3-12 CreateBiTree error message ( not initialized )



Figure 3-13 BiTreeDepth error message ( not initialized )



Figure 3-14 Parent error message ( not initialized )



Figure 3-15 ReadBiTree error message ( not initialized )

2. Initialize the form.



Figure 3-16 Initialize the binary tree

3 . At this time , any form of access to the table (function 4-21 ) will prompt that the table is empty. Examples are as follows :



Figure 3-17 BiTreeEmpty error message ( empty tree )



Figure 3-18 Value error message ( empty tree )



Figure 3-19 SaveBiTree error message ( empty tree )

4 . Import data by entering the file name . At this time, there are 4 nodes in the system .



Figure 3-20 Enter the file name , and the data is imported successfully

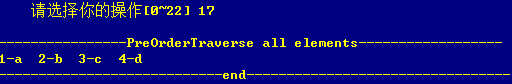


Figure 3-21 Preorder traversal

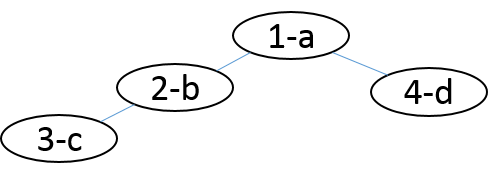


Figure 3-22 Binary tree structure diagram at this time

5. Add a new subtree under the left subtree of the node whose key is 4 .

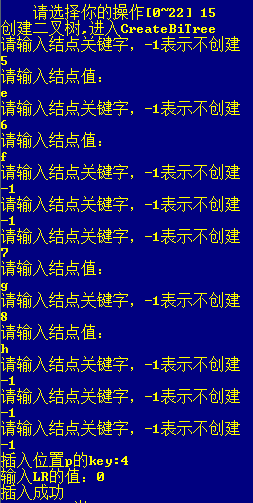


Figure 3-23 Change operation table

6. At this point the depth is 5.



Figure 3-24 Judgment table length

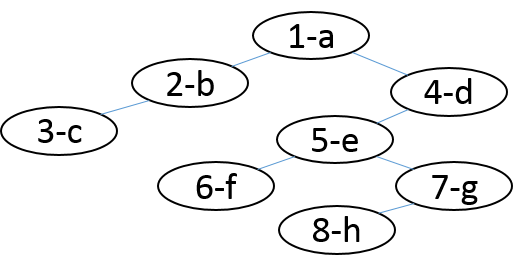


Figure 3-25 Binary tree structure diagram at this time

7. Inorder traversal.

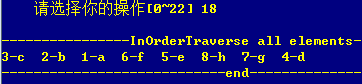


Figure 3-26 \_ Inorder traversal

8. Postorder traversal .

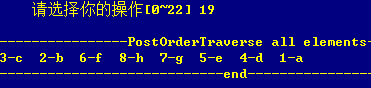


Figure 3-27 \_ \_ post order traversal

9. Layer order traversal.

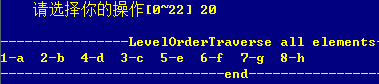


Figure 3-28 \_ \_ sequence traversal

Figure 1 - 24, 1 - 25, 1 - 26 Traversing the current table, judging the empty table, and finding the table length

10. Get the root node .



Figure 3-29 \_ \_ Get the root node

11. Find the node in the current table .

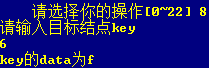


Figure 3-30 finds the node successfully

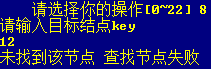


Figure 3-31 \_ \_ Failed to find node

12. Modify node data .

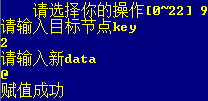


Figure 3-32 \_ \_ Modify node data successfully

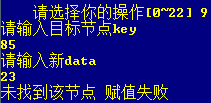


Figure 3-33 \_ \_ Failed to modify node data

13. Get the parent node .



Figure 3-34 \_ \_ Get the parent node successfully



Figure 3-35 \_ \_ Failed to get parent node



Figure 3-36 \_ \_ element not found

14. Get the left child .



Figure 3-37 \_ \_ Get the left child node successfully



Figure 3-38 \_ \_ Failed to get left child node



Figure 3-39 \_ \_ element not found

15. Get the right child .



Figure 3-40 \_ \_ Get the right child node successfully



Figure 3-41 \_ \_ Failed to get the right child node



Figure 3-42 \_ \_ element not found

16. Get the left brother .



Figure 3-43 \_ \_ Get the left sibling node successfully



Figure 3-44 \_ \_ Failed to get the left sibling node



Figure 3-45 \_ \_ element not found

17. Get the right sibling .



Figure 3-46 \_ \_ Get the left sibling node successfully



Figure 3-47 \_ \_ Failed to get the left sibling node



Figure 3-48 \_ \_ element not found

18. Insertion into subtree failed bounds check .

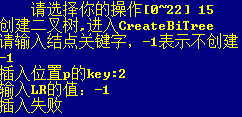


Figure 3-49 \_ \_ Failed to insert subtree (LR subscript out of bounds )

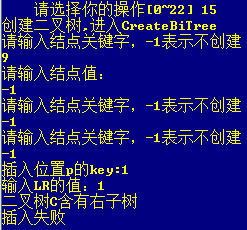


Figure 3-50 \_ \_ Failed to insert subtree (corresponding subtree is occupied )

1 9 . Delete subtree .

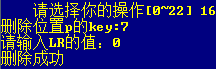


Figure 3-51 \_ \_ Subtree deleted successfully



Figure 3-52 \_ \_ Failed to delete subtree



Figure 3-53 \_ \_ element not found

20. Save the file after inserting the new binary tree.

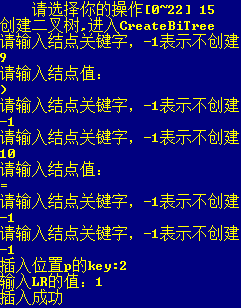


Figure 3-54 \_ insert binary tree

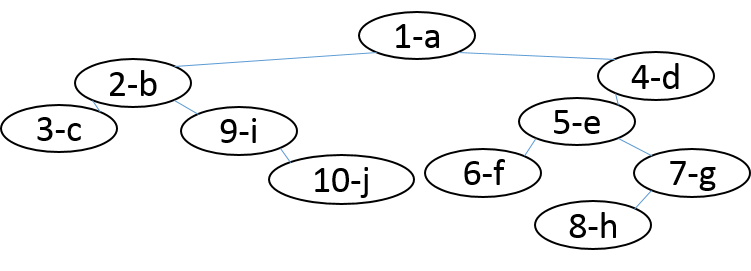


Figure 3-55 \_ The binary tree structure diagram at this time



Figure 3-56 \_ \_ file save



Figure 3-57 \_ \_ newly saved file

twenty one. Clear the binary tree.



Figure 3-58 \_ \_ empty binary tree



Figure 3-59 \_ \_ clear check

twenty two. Read the saved file to the system.



Figure 3-60 File read successfully

twenty three. Layer order traversal checks the read results.

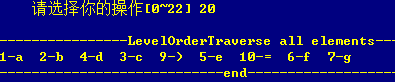


Figure 3-61 \_ \_ sequence traversal

twenty four. When opening a file with the wrong format , it will display that the file failed to open.



Figure 3-62 \_ \_ target file is not readable

25. Destroy the binary tree .



Figure 3-63 \_ \_ Destroy the binary tree successfully

26. Log out of the system .

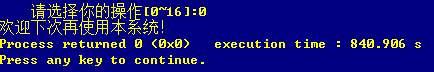


Figure 3-64 \_ \_ Exit system

## 3.4 Experimental summary

This experiment feels more difficult than the previous few times, so it takes more time. It is tangled to write the pre-order, in-order, and post-order traversal algorithms in the middle. After asking classmates, I found that it is very simple to use the recursive algorithm. It's almost done. In this experiment, I learned how to create a binary tree and various simple operations on the binary tree. I am more familiar with the idea of recursion, and I am more proficient in the use of pointers. Working hard to find a way to solve it is the only way to become a good programmer!

# 4 Graph Realization Based on Adjacency List

## 4.1 Problem Description

* + 1. Basic concept of graph

Graph G is an ordered two-tuple (V, E), where V is called the top set (Vertices Set), E is called the edge set (Edges set), and E and V are disjoint. They can also be written as V(G) and E(G).

The elements of E are all binary groups, represented by (x, y), where x, y ∈ V .

* + 1. Logical structure and basic operations

The abstract data type binary tree is defined as follows:

ADT BinaryTree {

Data Object D: D is a collection of data elements with the same properties.

Data relation R:

If D=Φ, then R=Φ, BinaryTree is called an empty binary tree;

If D≠Φ, then R ={H}, H is the following binary relationship:

1. There is a unique root data element root in D, which has no predecessor in relation H;
2. If D- {root} ≠Φ, then there exists D- {root}={D1,Dr}, and D1∩Dr = Φ;
3. If D1≠Φ, then there is a unique element X1 in D1, <root,X1>∈ H , and the relationship H1 on D1 is included in H; if D r ≠Φ, then there is a unique element Xr in Dr, <root , Xr>∈H, and the relationship on D r belongs to H;
4. (D, {H1}) is a binary tree conforming to this definition, called the left subtree of the root, (D r , {H r}) is a binary tree conforming to this definition, called the right subtree of the root.

}

According to the principle of combining minimum completeness and common use, 13 basic operations such as creating graph, destroying graph, finding vertex, obtaining vertex value and vertex value assignment are defined in the form of functions. The specific operation functions are defined as follows.

1. Create a graph: the function name is CreateCraph(&G,V,VR) ; the initial condition is that V is the vertex set of the graph , and VR is the relation set of the graph ; the operation result is to construct graph G according to the definitions of V and VR .

2. Destroy the graph: the name of the tree function is DestroyBiTree(T) ; the initial condition graph G already exists; the operation result is to destroy the graph G.

3. Find the vertex: the function name is LocateVex(G,u) ; the initial condition is that the graph G exists, and the vertices in u and G have the same characteristics; the operation result is that if u exists in the graph G , return the position information of the vertex u , Otherwise return other information.

4. Get the vertex value: the function name is GetVex (G,v) ; the initial condition is that the graph G exists, and v is a vertex in G ; the operation result is to return the value of v .

5. Vertex assignment: the function name is PutVex (G,v,value) ; the initial condition is that the graph G exists, and v is a vertex in G ; the operation result is to assign value to v .

6. Obtain the first adjacency point: the function name is FirstAdjVex(&G, v) ; the initial condition is that the graph G exists, v is a vertex of G ; the operation result is to return the first adjacency vertex of v , if v has no adjacency vertex, Returns "null" .

7. Obtain the next adjacent point: the function name is NextAdjVex(&G, v, w) ; the initial condition is that the graph G exists, v is a vertex of G , and w is an adjacent vertex of v ; the operation result is returned to v (relative to w ) the next adjacent vertex, or "null" if w is the last adjacent vertex.

8. Insert a vertex: the function name is InsertVex(&G,v) ; the initial condition is that the graph G exists, and the vertices in v and G have the same characteristics ; the operation result is to add a new vertex v in the graph G.

9. Delete a vertex: the function name is DeleteVex(&G,v) ; the initial condition is that the graph G exists, and v is a vertex of G ; the operation result is to delete the vertex v and the arcs related to v in the graph G.

10. Insert arc: the function name is InsertArc(&G,v,w) ; the initial condition is that graph G exists, and v and w are the vertices of G ; the operation result is to add arc <v,w> in graph G , if graph G It is an undirected graph, and <w,v> needs to be added .

11. Delete arc: the function name is DeleteArc(&G,v,w) ; the initial condition is that the graph G exists, v and w are the vertices of G ; the operation result is to delete the arc <v,w> in the graph G , if the graph G is an undirected graph, you also need to delete <w,v> .

12. Depth-first search traversal: the function name is DFSTraverse(G,visit()) ; the initial condition is that the graph G exists; the operation result is a depth-first search traversal of the graph G , and each vertex in the graph is visited once using the function visit , and only visited once.

13. Breadth-first search traversal: the function name is BFSTraverse(G,visit()) ; the initial condition is that the graph G exists; the operation result is a breadth-first search traversal of the graph G , and each vertex in the graph is visited once using the function visit , and Visit only once.

## 4.2 System Design

* + 1. data physical structure

1. Data physical structure of arc nodes

typedef struct ArcNode{

int adjvex; // The number of the vertex pointed by the arc

struct ArcNode \* nextarc; //pointer to the next arc

InfoType \*info; //The pointer of the arc-related information ( the arc length of the arc in the network )

}ArcNode;

2. The data physical structure of the vertex node

typedef struct VNode{

ArcNode \* firstarc; //Pointer to the first arc attached to this vertex

char data; //vertex name

}VNode, AdjList[MAX\_VERTEX\_NUM];

3. The data physical structure of the graph

typedef struct{

AdjList vertices;

int vexnum, arcnum; //The current number of vertices and arcs of the graph

int kind; //The kind flag of the graph

}ALGraph;

In this program, the data atomic type Vertex Type is defined as a char character type, and InfoType is defined as an int\* integer pointer.

* + 1. demo system

The adjacency list is used as the physical structure of the undirected network to realize the basic operation of § 4.1.2 . Among them, VertexType is the type name of the data element, and the specific meaning can be defined by oneself. For the definition and reference of other related types and constants, please refer to p163 of the literature [1] .

It is required to construct a functional demonstration system with menus. Among them, in the main program, the preparation of the actual parameter value required for the function call and the display of the function execution result are completed, and an appropriate operation prompt is given.

The demo system can choose to save the undirected network as a file. Among them, ① it is necessary to design the file data record format to efficiently save the complete information of the data logic structure (D,{R}) of the graph ; ② it is necessary to design a reasonable mode of file saving and loading operations in the undirected network. Appendix B provides methods for file access.

The demo system has the option to implement multiple undirected net management.

Write the menu presentation and user selection input into the while loop, and use op to get the user's selection. Op is initialized to 1 so that the loop can be entered for the first time. After entering the while loop, the system first displays the function menu, and then prompts the user to input a selection (0-20 ) , among which 1-20 corresponds to a basic operation of the binary tree , and each corresponds to a function , and the serial number input by the user is corresponding to the corresponding one through the switch statement. Function function, break jumps out of the switch statement after executing the statement, and executes the while loop until the user enters 0 to choose to exit and exit the system.

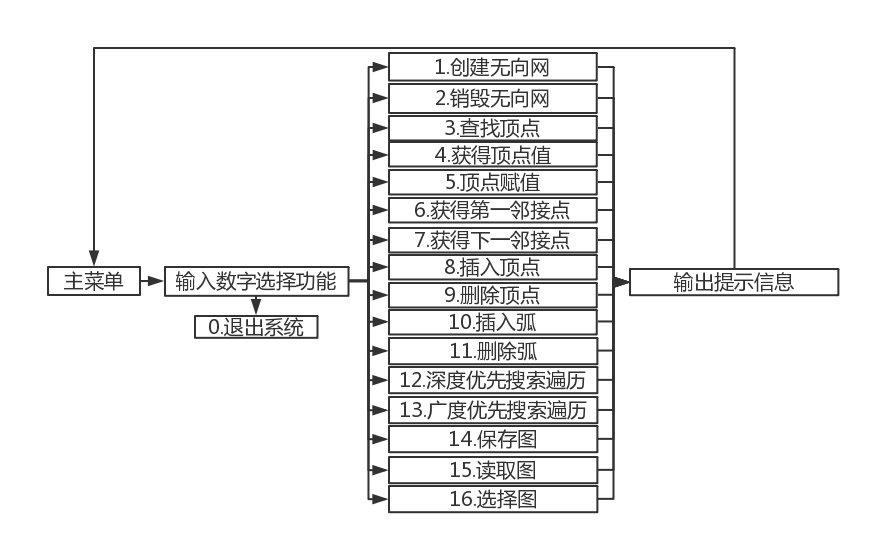


Figure 4-1 System design structure diagram

* + 1. Data file storage format

Considering the principle of simplicity and efficiency, the data file storage format is designed as follows.

File format: dat , which is convenient for programs to read data quickly .

The corresponding file space is allocated according to the corresponding data type in order for storage, and there is no gap between the data .

* + 1. Algorithm thinking and design

The idea and design of the undirected network computing algorithm are as follows:

1. The idea of creating an undirected network algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created). The parameter of the function is the graph structure pointer G defined in the main function . Enter the number of vertices , the number of edges and the corresponding number of vertices, arcs and corresponding information (including the arc length, the name of the arc head node and the arc tail node ). Note that under the condition of an undirected network , an inverse adjacency relationship should be established in the adjacency table at the same time. That is, the initialization of the undirected network is completed. The flowchart is shown in Figure 4-2 .

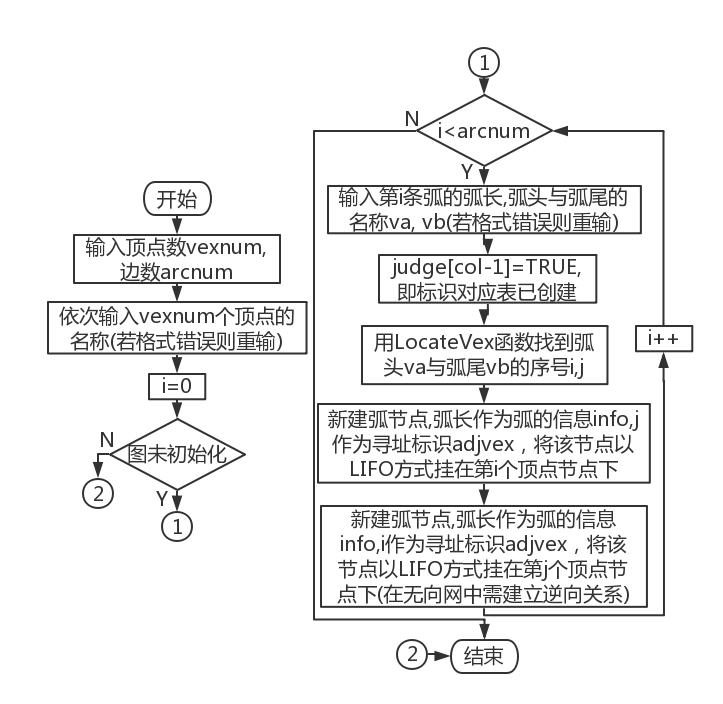


Figure 4-2 Flow chart of creating an undirected network

2. The idea of destroying the undirected network algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created). The parameter of the function is the graph structure pointer G. If there is a vertex node in the graph, the arc node under the vertex node should be deleted first , and the arc length should be deleted at the same time in the case of an undirected network . Reset the number of points and edges of G to 0, that is, complete the destruction of the graph. The flowchart is shown in Figure 4-3 .

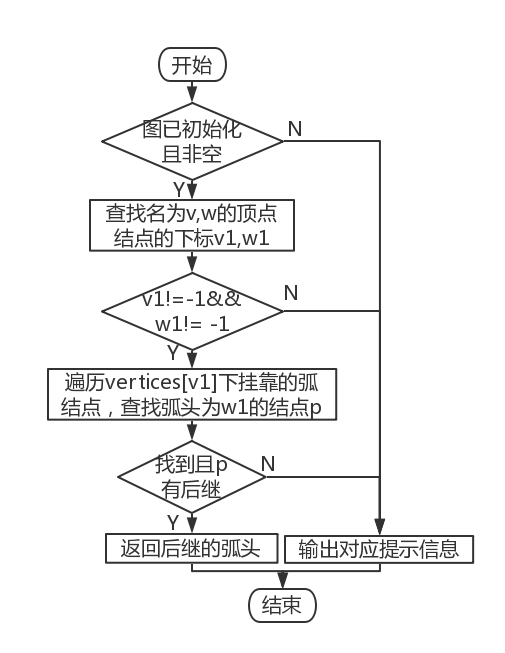
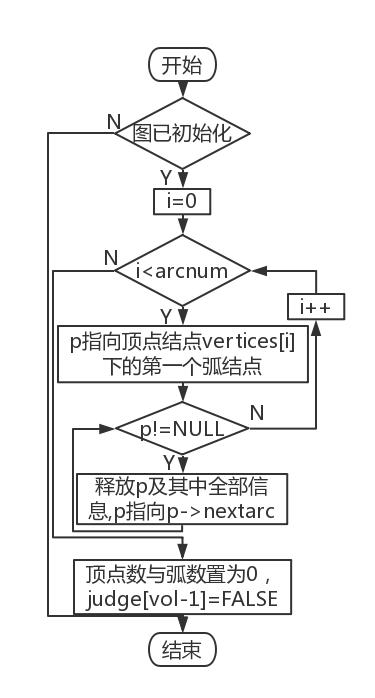


Figure 4-3 Flow chart of destroying undirected network Figure 4-4 Flowchart of obtaining the next adjacent point

3. The idea of finding vertices algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are the graph structure pointer G and the name u of the vertex to be searched, traverse the vertex nodes , find the node with the same name as u , return the subscript of the node if the search is successful , and return -1 if it fails .

4. The idea of obtaining the vertex value algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty + whether the subscript is out of bounds ). The parameters of the function are the graph structure pointer G and the subscript rec of the vertex to be checked, and directly return the name of the corresponding vertex in the table below .

5. Vertex assignment algorithm idea: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty + whether the subscript is out of bounds ). The parameters of the function are the graph structure pointer G , the subscript rec of the node and the new name v of the corresponding node. Assign the name of the vertex node corresponding to the subscript to v to complete .

6. Obtaining the idea of the first adjacent point algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are the graph structure pointer G and the node name v. First call the search vertex function to find the subscript of the vertex node named v, if the vertex does not exist, return ERROR; after finding it, judge whether it has an arc node attached to it, if so , return the arc head of the first arc node , otherwise returns ERROR.

7. Obtaining the idea of the next adjacent point algorithm: at the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are the graph structure pointer G , the reference node name v and the location node name w. First call the find vertex function to find the subscripts of the vertex nodes named v and w, and return ERROR if one of them is not found . After finding it, search for the subscript node whose arc head is w under the vertex node named v , if not found, return ERROR. If found, return the subscript of its successor node, or return ERROR if there is no successor. The flowchart is shown in Figure 4-4 .

8. The idea of inserting vertex algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty + whether the number of vertices has reached the maximum value ). The parameters of the function are the graph structure pointer G and the new node name v. Extract the number of existing vertices in the graph as the subscript of the new node , assign v to the data of the newly opened node and set the pointer to the arc node to empty ( the newly created nodes are all isolated nodes ) , and the completion is complete.

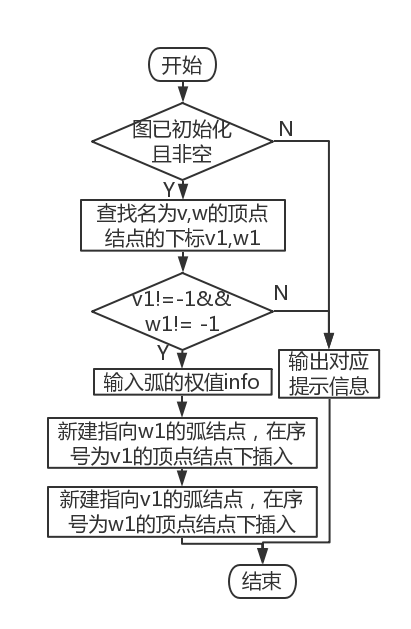
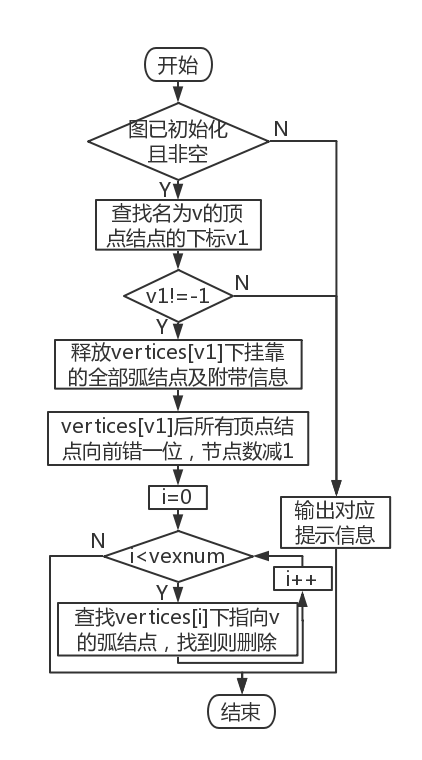


Figure 4-5 Flow chart of deleting vertices and nodes Figure 4-6 Insert Arc Node Flowchart

9. The idea of deleting vertices algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are the graph structure pointer G and the name v of the node to be deleted. First call the find vertex function to find the subscript of the vertex node named v, and return ERROR if the vertex does not exist. Delete all the arc nodes under the target node , and the subsequent vertex nodes are sequentially shifted one bit forward to cover the target vertex node , and finally traverse the adjacency list, delete all arc nodes whose arc head name is v, and it is completed. The flowchart is shown in Figure 4-5 .

10. The idea of arc insertion algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are graph structure pointer G , node names v and w. First call the find vertex function to find the subscripts of the vertex nodes named v and w, and return ERROR if at least one of them is not found. Input the weight of the arc, and set up the corresponding adjacency relationship in the table with v and w respectively as the arc head and arc tail , that is to say, it is completed. The flowchart is shown in Figure 4-6 .

11. The idea of deleting the arc algorithm: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameters of the function are graph structure pointer G , node names v and w. First call the find vertex function to find the subscripts i and j of the vertex nodes named v and w, and return ERROR if at least one of them is not found. Find and delete the arc node pointing to j under the vertex node with the subscript i ; similarly, find and delete the arc node pointing to i under the vertex node with the subscript j , which is completed.

12. Depth-first search traversal algorithm idea: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ) . The parameter of the function is the pointer G of the graph structure . Traverse the vertex nodes, use the auxiliary array to judge whether the current node has been visited, if not , call the deep search recursive function DFS. In the DFS function , first set the auxiliary array element corresponding to the current node to TRUE (visited), print the node information, and then find the nearest unvisited vertex node by attaching to the arc node , and recurse like this to complete .

13. Breadth-first search traversal algorithm idea: At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the graph is empty ). The parameter of the function is the pointer G of the graph structure . Initialize the queue first , and start accessing from the first vertex node. After the corresponding auxiliary array element is set to TRUE and the node information is printed, the element enters the queue. As long as the queue is not empty , the head element of the queue will be popped up continuously . For each queue head , traverse the arc node it is attached to. The arc head of is not accessed , then set the corresponding auxiliary array element to TRUE and print the node information , and so on until the auxiliary array elements corresponding to all vertices in the graph are TRUE, which is completed.

14 . Data storage algorithm idea : the parameter of the function is the graph structure array List . The file name is automatically generated from the system time , and the corresponding binary file is generated. Use a loop to count the number of non- empty tables , and return an error if there is no non-empty table. Save this data to disk . For each graph , first store the graph structure attributes ( type + number of vertices + number of edges) , and for each vertex node , call a function to find its out-degree (directed graph) / or degree ( undirected graph ) , and store it together with the name , and then traverse the arc nodes attached to it , store the arc head and arc length, and it is completed. The flowchart is shown in Figure 4-7 .

15 . Data recovery algorithm idea: the parameter of the function is the graph structure array L ist . After successfully opening the file according to the file name, first read the number of valid tables. For each graph , read the graph structure attributes ( type + number of vertices + number of edges) . For each vertex node, read its out-degree (directed graph ) / or degree ( undirected graph ) , successively establish the same number of arc nodes as the out degree, read their serial numbers and information and attach them to the vertex nodes in turn, and the completion is complete. The flowchart is shown in Figure 4-8 .

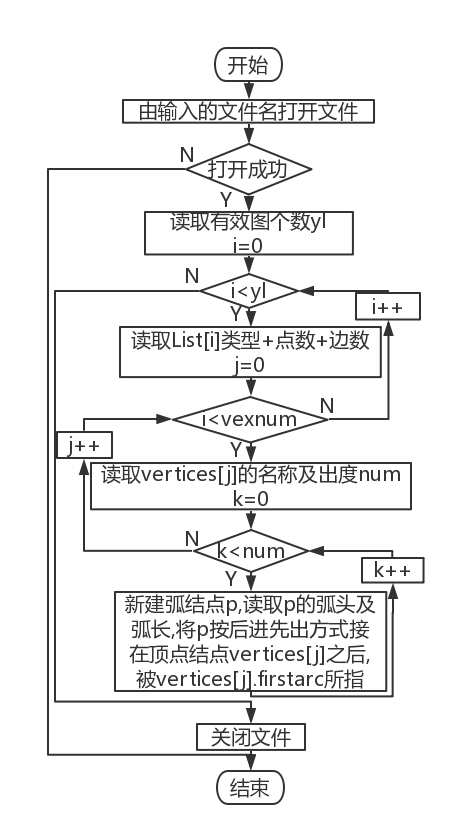
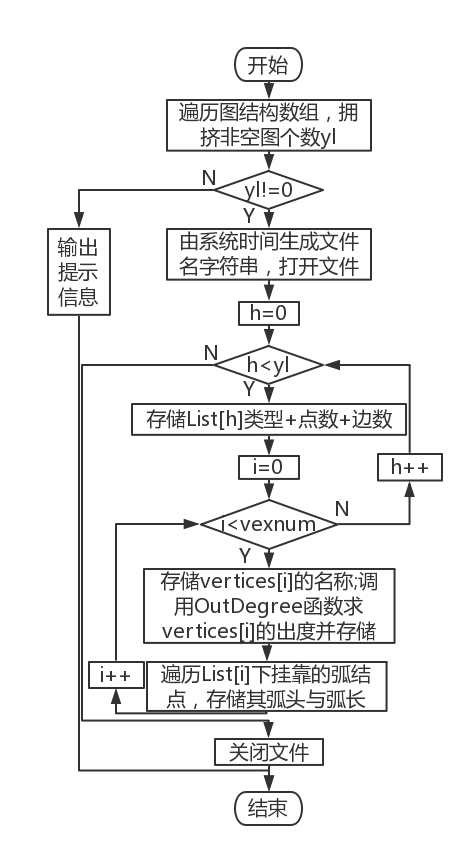


Figure 4-7 \_ Data storage flow chart Figure 4-8 \_ Data Recovery Flowchart

16. The idea of the table-changing algorithm: the parameters of the function are the bit sequence col of the target graph, the address of the graph structure pointer L and the graph structure array L ist . At the beginning, it is necessary to judge the legality (whether the graph has been created + whether the subscript is out of bounds). Assign the address of the table to be operated to the pointer of the same structure , and output a prompt message if the table is not initialized , and it is completed.

* + 1. Algorithm Time and Space Complexity Analysis

Table 4-1 Algorithm time and space complexity analysis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function name | time complexity | space complexity | Function name | time complexity | space complexity |
| CreateGraph | O(n) | O(1) | DeleteVex | O(n 2 ) | O(1) |
| Destroy Graph | O(n 2 ) | O(1) | InsertArc | O(n) | O(1) |
| LocateVex | O(n) | O(1) | DeleteArc | O(n) | O(1) |
| GetVex | O(1) | O(1) | DF Traverse | O(n n ) | O(1) |
| PutVex | O(1) | O(1) | BF Traverse | O(n 3 ) | O(n) |
| FirstAdjVex | O(n) | O(1) | SaveGraph | O(n 3 ) | O(1) |
| NextAdjVex | O(n) | O(1) | LoadGraph | O(n 3 ) | O(1) |
| InsertVex | O(1) | O(1) | ChooseList | O(1) | O(1) |

## 4.3 System Implementation

* + 1. Programming environment , operating environment , project engineering description

This experiment is written with Codeblocks programming software, and compiled and run with Codeblocks . The project name is Adjacent-Chart-Based-MultiGraph . The demo system menu interface is shown in Figure 4-9 .



Figure 4-9 \_ \_ Demonstration system menu interface

* + 1. header files and predefined constants

1. Header file

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

2. Predefined constants

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR -1

3. Type expressions

typedef int Status; // is the type of the function, its value is the function result status code

typedef int QElemType; //subscript corresponding to the vertex node

typedef struct QNode{

QElemType data; //node information

struct QNode \*next;

}QNode,\*QueuePtr; //queue node structure

typedef struct{

QueuePtr front, rear; //Queue head, queue tail pointer

}LinkQueue; //queue structure

typedef int Boolean;

typedef int InfoType; // arc length in the net

typedef char VertexType; //node name

typedef struct ArcNode{

int adjvex; //The position of the vertex pointed by the arc

struct ArcNode \* nextarc; //pointer to the next arc

InfoType \*info; //The pointer of the arc related information

}ArcNode; //arc node structure

typedef struct VNode{

ArcNode \* firstarc; //Pointer to the first arc attached to this vertex

char data; //vertex information

}VNode, AdjList[MAX\_VERTEX\_NUM]; //vertex node structure

typedef struct{

AdjList vertices;

int vexnum, arcnum; //The current number of vertices and arcs of the graph

int kind; //The kind flag of the graph

}ALGraph; //graph structure

* + 1. Test Plan

The undirected network operation function of the system is tested according to the following plan:

1. Detect the working conditions of the system under normal conditions;
2. Focus on checking the working conditions during illegal border operations.

Table 4-2 Demonstration System Test Plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function to be tested | test sequence | test input | expected output | Undirected state |
| 1. CreateGraph | 1,14 | Step 1:  The first set of inputs: 1;4;5;abcd;1 ab;2 ac;3 ad;2 bc;1 bd  Step 14 :  The second set of inputs: 1;0;@  The third set of inputs:  3;2;efg;2 eg;4 ef | Create List[0] List[1] List[2] successfully | Step 1: List[0] is created  Step 14:  List[1] and List[2] are created |
| 2. Destroy Graph | 19 | none | List[2] destroyed successfully | List[2] does not exist |
| 3. LocateVex | 2 | first input: a  Second input: \_ | The first input: the serial number is 0  Second input : lookup failed | same as step 1 |
| 4. GetVex | 3 | First input: 3  Second input: 9 | First input: name is d  Second input : lookup failed | same as step 1 |
| 5. PutVex | 13 | First input: 6 1  Second input : 3d | First input: Failed to modify  The second input : modified successfully | The node named e is renamed to d |
| 6. FirstAdjVex | 5,7 | Step 5 :  first input: e  Second input : q  Step 7:  Third input: e | The first + second input: failed to find  The third input: the first adjacent vertex number is 3 | Step 5 : Same as Step : 4  Step 7 : Same as Step 6 |
| 7. NextAdjVex | 8 | first input: de  Second input: ed  Third input: `/  Fourth input: ef  Fifth input: ae | The first input: the sequence number of the next adjacent point is 1  The second to fifth input : return failure and failure reason | Same as step 6 |
| 8. InsertVex | 4 | e | Inserted successfully | Add node e to List[0] |
| 9. DeleteVex | 11,18 | Step 11 :  First input: ?  Second input: d  Step 18:  Second input: @ | First input: no such vertex exists  The second input : deleted successfully  The third input: the deletion is successful , there are no vertices in the graph | Step 11 :  The node vertex named d in List[0] and all arc nodes pointing to d are deleted  Step 18 :  There are no nodes in List[1] |
| 10. InsertArc | 6 | First input: ag  First input: ed 4 | First input : vertex v or w does not exist  The second input: added successfully | Add arc ed with length 4 to List[0] |
| 11. Delete Arc | 9 | first input: df  First input: ed | First input : vertex does not exist  The second input: deleted successfully | same as step 4 |
| 12. DF Traverse | 10, 17 | none | The number and name of each vertex in the current List | Step 10 : Same as Step 4  Step 17 : Same as Step 14 |
| 13. BF Traverse | 12 | none | List[0] the number and name of each vertex | Same as step 11 |
| 14. SaveGraph | 15,20 | none | The information of the table is printed in the program , and the dat file named the system time at the time of archiving is in the root directory of the program | Step 15 :  Same as step 14  Step 20 :  Same as step 19 |
| 15. LoadGraph | 16 | filename string | The information of the table is printed in the program , and the reading is successful | Same as step 14 |
| 16. ChooseList | called multiple times | of the graph , that is, the subscript of the target graph minus 1 | If the table is not initialized, output a prompt message | same as previous step |

* + 1. test

1. There is no picture in the system at the beginning . At this time , any form of access to the graph (function 2-15 ) will prompt that the graph is not initialized . Examples are as follows :



Figure 4-10 DestroyGraph error message ( not initialized )



Figure 4-11 NextAdjVex error message ( not initialized )



Figure 4-12 InsertVex error message ( not initialized )



Figure 4-13 DFSTraverse error message ( not initialized )

2. Create graphs . In Figure 4-15 , the number of the vertex node is its subscript, and the letter is its name; the number on the left of the arc node is the subscript of the arc head , and the number on the right is the arc length, and the adjacency list behind it follows this representation.

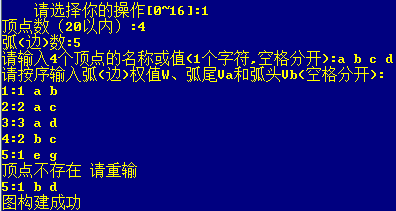


Figure 4-14 Initializing an undirected graph, which involves the special case of failing to find a vertex

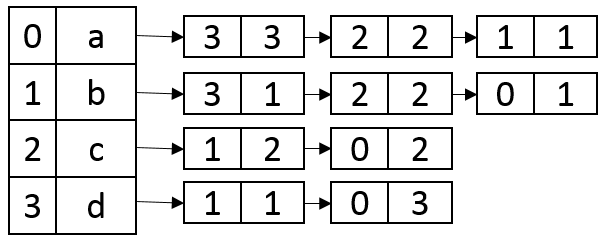


Figure 4-15 Adjacency list under List [0]

3 . At this time, if you create a picture again , you will be prompted that there is a picture under this location :



Figure 4-16 SaveBiTree error message ( empty tree )

4 . Find nodes .



Figure 4-17 find the node successfully



Figure 4-18 Failed to find nodes

5. Get the node name .



Figure 4-19 Get the node name successfully



Figure 4-20 Failed to get the node name

6. Add a node named e .



Figure 4-21 Adding nodes

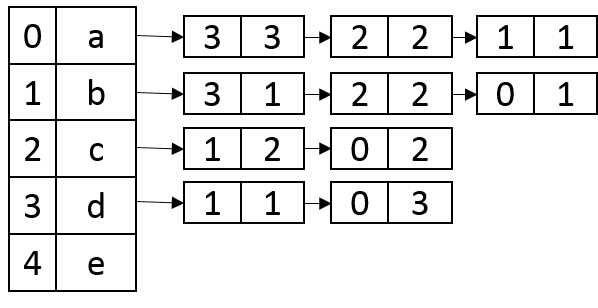


Figure 4-22 Adjacency list under List [0]

7. At this time e has no adjacent points.



Figure 4-23 \_ return first neighbor node



Figure 4-24 \_ \_ Failed to find reference node

8. Insert arc ed of length 4 .



Figure 4-25 \_ \_ insert arc failed

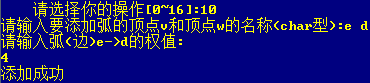


Figure 4-26 \_ \_ Insert arc ed of length 4

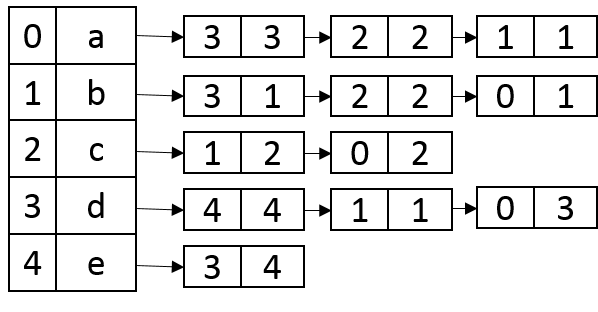


Figure 4-27 \_ \_ Adjacency list under List [0]

9. Check the insertion situation. At this time, e has only one adjacent node of d.



Figure 4-28 \_ \_ The first neighbor point of e is d



Figure 4-29 \_ \_ The next adjacent point of d relative to e is b

10. Returns the number of boundary conditions for the next neighbor .



Figure 4-30 \_ \_ The next adjacent point of e relative to d does not exist



Figure 4-31 \_ \_ Reference vertex v does not exist



Figure 4-32 \_ \_ positioning vertex w does not exist



Figure 4-33 \_ \_ two vertices are not adjacent

11. Delete the arc d- e, and the adjacency list returns to the state shown in Figure 4-22 .



Figure 4-34 Failed to delete arc



Figure 4-35 delete arc successfully

12. Depth-first search, e is visited at the end , which proves that arc deletion is successful .



Figure 4-36 Depth-first search

13. Delete node d .



Figure 4-37 \_ \_ Failed to delete node



Figure 4-38 \_ \_ Node deleted successfully

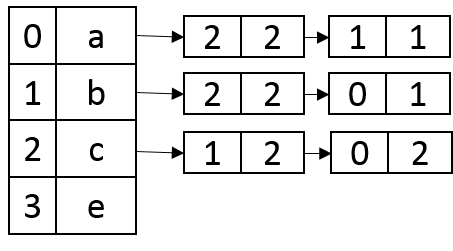


Figure 4-39 \_ \_ Adjacency list under List [0]

14. Breadth-first search proves that node d is successfully deleted .



Figure 4-40 \_ \_ breadth first search

15. Reassign e to d, then add the arc

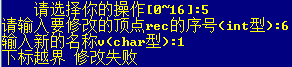


Figure 4-41 \_ \_ Vertex assignment failed

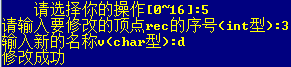


Figure 4-42 \_ \_ Vertex assignment succeeded

16. Switch the operation object , and add pictures successively in List[1] and List [2] .



Figure 4-43 \_ \_ Operation List [1]

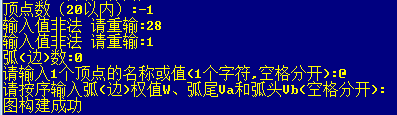


Figure 4-44 \_ \_ There is only one isolated vertex in List [1]



Figure 4-45 \_ \_ Operation List [2]

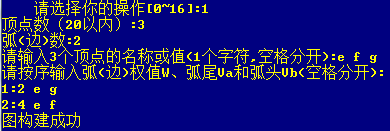


Figure 4-46 \_ \_ Create List [2]

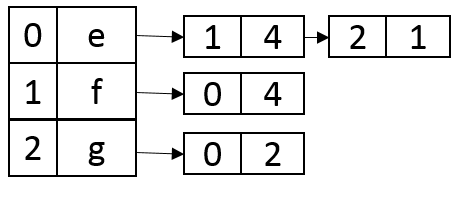


Figure 4 - Adjacency list under 47 List [2]

17. Save multiple graphs .

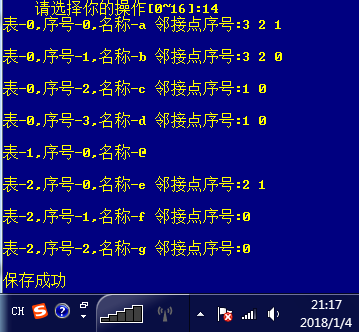


Figure 4 - 48 save multiple pictures



Figure 4-49 Archives in the root directory of the program

18. Close the program and reopen it to recover data from the file and examine it .



Figure 4-50 \_ \_ Failed to restore data

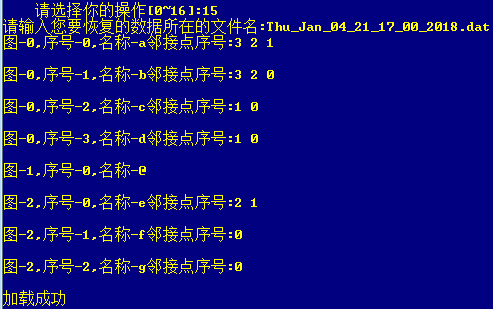


Figure 4-51 \_ \_ Successfully restored data



Figure 4-50 \_ Traverse List [0]



Figure 4-52 \_ traverse List [1]



Figure 4-53 \_ Traverse List [2]

19. Delete the last vertex in List [1] and check the deletion effect.



Figure 4-54 \_ Operation List [1]



Figure 4-55 \_ delete vertex



Figure 4-56 \_ Check delete result

20. Destroy the graph in List [2] .



Figure 4-57 \_ \_ Operation List [2]



Figure 4-58 \_ \_ Destroy List [2]

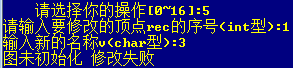


Figure 4-59 \_ Check the destruction result

twenty one. File save .

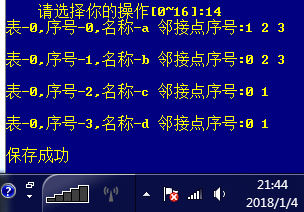


Figure 4-60 \_ \_ file save



Figure 4-61 \_ \_ Files in the root directory of the program

twenty two. Exit system.

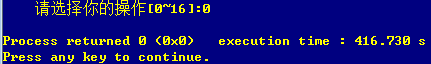


Figure 4-62 \_ insert binary tree

## 4.4 Experimental summary

This experiment can basically master the basic operations on the graph . I personally think that the core of this experiment lies in the structure of the graph stored in the adjacency list , and the mastery of the method of traversing the graph with depth and breadth first . Some small problems encountered during the experiment were solved one by one by asking the students . The data structure experiment is almost over here . I hope that I can systematically enhance my programming ability through the winter vacation courses , and practice what I have learned in the data structure class . Work harder !

# references

[1] Yan Weimin et al . Data Structure (C Language Edition ). Tsinghua University Press

[2] [Larry Nyhoff](http://www.calvin.edu/~nyhl/index.html) . [ADTs, Data Structures, and Problem Solving with C++.](http://vig.prenhall.com/catalog/academic/product/0,1144,0131409093,00.html) Second Edition, [Calvin College](http://cs.calvin.edu/) , 2005

[3] Yin Lifeng . Fundamentals of Qt C++ Cross-platform Graphical Interface Programming . Tsinghua University Press , 2014: 192-197

[4] Yan Weimin et al . Data Structure Problem Collection (C language version ). Tsinghua University Press

## Instructor's comments

1. Comments on the experiment report

|  |
| --- |
|  |

2. Scoring the experiment report

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Grading Items  ( point value ) | program content  (36.8 points ) | program specification  (9.2 points ) | Content of report  (36.8 points ) | Report Specification  (9.2 points ) | attendance  ( 8 points) | Overdue deductions | combine count  (100 points ) |
| Score |  |  |  |  |  |  |  |

# Appendix A Source Program Based on Linear Table of Sequential Storage Structure

using namespace std;

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define OK 1

#define TRUE 1

#define FALSE -1

#define ERROR -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef int status;

typedef int ElemType; //Data element type definition

typedef struct{ //Definition of sequence table (sequence structure)

ElemType \*elem;

int length;

int listsize;

}SqList;

//Arrays with a length of 20 determine whether the corresponding linear tables are initialized, TRUE means initialized; FALSE means uninitialized

//vol is used to locate the linear table of the current operation

int judge[20], vol=1;

ElemType \*newbase;

FILE \*fp;

char \*filename, \*pc, appe[5]=".dat", input[60]; //used to generate filename

status IntiaList(SqList \*L);

status DestroyList(SqList \*L);

status ClearList(SqList \*L);

status ListEmpty(SqList \*L);

int ListLength(SqList \*L);

status GetElem(SqList \*L,int i,ElemType &e);

status LocateElem(SqList \*L,ElemType e); //simplified

status PriorElem(SqList \*L,ElemType cur,ElemType &pre\_e);

status NextElem(SqList \*L,ElemType cur,ElemType &next\_e);

status ListInsert(SqList \*L,int i,ElemType e);

status ListDelete(SqList \*L,int i,ElemType &e);

status ListTrabverse(SqList \*L); // simplified

status LoadList(FILE \*fp,SqList List[]);

status SaveList(FILE \*fp,SqList List[]);

status ChooseList(int vol, SqList \*\*L, SqList List[]);

status ChartTraverse(SqList List[]);

int main(void){

SqList \*L=NULL, List[20];

int op=1,i,e,cur,pre\_e,next\_e;

L=List;

while(op){

system("cls");printf("\n\n");

printf(" There are 20 tables in the system, and the table %d is currently being operated\n", vol); //Prompt the order of the table currently being operated

printf("Menu for Linear Table On Sequence Structure\n");

printf("----------------------------------------------- --\n");

printf(" 1. IntiaList 7. LocateElem\n");

printf(" 2. DestroyList 8. PriorElem\n");

printf(" 3. ClearList 9. NextElem \n");

printf("4. ListEmpty 10. ListInsert\n");

printf("5. ListLength 11. ListDelete\n");

printf("6. GetElem 12. ListTrabverse\n");

printf("13.LoadList 14.SaveList\n");

printf(" 15.ChooseList 16.ChartTraverse\n");

printf(" 0. Exit\n");

printf("----------------------------------------------- --\n");

printf(" Please choose your operation [0~16]:");

scanf("%d",&op);

switch(op){

case 1:

if(IntiaList(L)==OK)

printf("Linear table created successfully!");

else printf(" The linear table creation failed!");

getchar();getchar();

break;

case 2:

DestroyList(L);

getchar();

break;

case 3:

ClearList(L);

getchar();

break;

case 4:

i = ListEmpty(L);

if(i==TRUE)

printf("The linear table is empty!");

else if(i==FALSE)

printf("The linear table is not empty!");

getchar();getchar();

break;

case 5:

if(ListLength(L)==ERROR){

getchar();getchar();

break;

}

printf(" Table length is %d", L->length);

getchar();getchar();

break;

case 6:

printf("Please enter element sequence i(1<=i<%d):",L->length+1);

scanf("%d",&i);

if(GetElem(L,i,e)==OK)

printf("The %dth element is %d", i, e);

getchar();getchar();

break;

case 7:

printf("Please enter the data to be checked:");

scanf("%d", &e);

i=LocateElem(L,e);

if(i>0)

printf("%d is the %dth element in the table", e, i);

if(!i)

printf("This element was not found");

getchar();getchar();

break;

case 8:

printf("Please enter location data:");

scanf("%d", &cur);

if(PriorElem(L,cur,pre\_e)==OK)

printf("%d's predecessor is %d", cur, pre\_e);

getchar();getchar();

break;

case 9:

printf("Please enter location data:");

scanf("%d", &cur);

if(NextElem(L,cur,next\_e)==OK)

printf("%d's successor is %d", cur, next\_e);

getchar();getchar();

break;

case 10:

printf("Please enter the insertion position i(1<=i<=%d):", L->length+1);

scanf("%d", &i);

printf("Please input insert data e:");

scanf("%d", &e);

if(ListInsert(L,i,e)==ERROR)

printf("Insert failed");

else

printf("insert successfully");

getchar();getchar();

break;

case 11:

printf(" Please enter the deletion position i (1<=i<=%d)", L->length);

scanf("%d", &i);

if(ListDelete(L,i,e)==ERROR)

printf("Delete failed");

else

printf("Deleted successfully, the deleted element is %d", e);

getchar();getchar();

break;

case 12:

if(ListTrabverse(L)==ERROR) printf(" Traversal failed!\n");

getchar();getchar();

break;

case 13:

if(LoadList(fp,List)==OK){

printf("File loaded successfully");

}

getchar();getchar();

break;

case 14:

if(SaveList(fp,List)==OK){

printf("File saved successfully");

}

getchar();getchar();

break;

case 15:

printf("Please enter the bit sequence vol of the table to be operated: (1<=vol<=20)");

scanf("%d", &vol);

ChooseList(vol, &L, List);

getchar();

break;

case 16:

if(ChartTraverse(List)==ERROR) printf("Form traversal failed!");

getchar();getchar();

case 0:

break;

}//end of switch

}//end of while

printf("Welcome to use this system next time!");

return 0;

}//end of main()

/\*\*

\* Function name: InitiaList

\* Function parameter: the address of the linear table pointer L or List[]

\* Function function: construct an empty linear table

\* Return value: return OK if successfully constructed, otherwise return ERROR

\*\*/

status IntiaList(SqList \*L){

L->elem = (ElemType \*)malloc( LIST\_INIT\_SIZE \* sizeof (ElemType));

if(!L->elem) return ERROR;

L->length=0;

L->listsize=LIST\_INIT\_SIZE; //Initial allocation space

judge[vol-1] = TRUE; //The current table has been initialized, modify the corresponding judgment variable

return OK;

}

/\*\*

\* Function name: DestroyList

\* Function parameters: linear table pointer L

\* Function function: delete linear table

\* Return value: return OK if destroyed successfully, otherwise return ERROR

\*\*/

status DestroyList(SqList \*L){

//For each element of the judgment array that has not been operated, its value is a random number, not necessarily FALSE, so the condition should be non-TRUE, and will not be explained later

if(judge[vol-1] !=TRUE){

printf("\nThis table is not initialized!");

getchar();

return ERROR;

}

free(L->elem);

L->length=L->listsize=0;

//Destroy table L and all its affiliated information

L=NULL;

judge[vol-1] = FALSE; //Modify the corresponding judgment variable

printf(" Linear table %d has been deleted!", vol);

getchar();

return OK;

}

/\*\*

\* Function name: ClearList

\* Function parameters: linear table pointer L

\* Function function: reset the linear table

\* Return value: return OK if it is empty successfully, otherwise return ERROR

\*\*/

status ClearList(SqList \*L){

if(judge[vol-1] !=TRUE){

printf("This table is not initialized!");

getchar();

return ERROR;

}

L->length=0; //Initial table length

L->listsize=LIST\_INIT\_SIZE; //Initial state space size

printf("The table has been cleared!");

getchar();

return OK;

}

/\*\*

\* Function name: ListEmpty

\* Function parameters: linear table pointer L

\* Function function: judge whether the linear table is empty

\* Return value: Return TRUE if L is an empty list, otherwise return FALSE

\*\*/

status ListEmpty(SqList \*L){

if(judge[vol-1] !=TRUE){

printf("This table is not initialized!");

getchar();

return ERROR;

}

int flag=TRUE;

if(L->length!=0)

flag = FALSE; // table length is not 0

return flag;

}

/\*\*

\* Function name: ListLength

\* Function parameters: linear table pointer L

\* Function function: calculate the number of data elements in the linear table L;

\* Return value: return the number of data elements in L if successful, return ERROR if failed

\*\*/

int ListLength(SqList \*L){

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized");

getchar();

return ERROR;

}

return L->length;

}

/\*\*

\* Function name: GetElem

\* Function parameters: linear table pointer L, e is the address of the i-th data;

\* Function function: find and display the value of the i-th element in L;

\* Return value: return the value of the i-th element if successful, otherwise return ERROR

\*\*/

status GetElem(SqList \*L,int i,ElemType &e){

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("\nThere is no element in the table!");

getchar();

return ERROR;

}

if(i<1 || i>L->length+1){

printf("Subscript out of bounds!");

getchar();

return ERROR;

}

e=L->elem[i-1];

return OK;

}

/\*\*

\* Function name: LocateElem

\* Function parameters: linear table pointer L, relative element value;

\* Function function: Find the bit sequence of the same data as e in L; the bit sequence starts from 1

\* Return value: return the bit sequence of the first data that is the same as e if successful, return 0 if it does not exist, and return ERROR in other cases

\*\*/

status LocateElem(SqList \*L,ElemType e){

int i;

if(judge[vol-1] !=TRUE){

printf(" This table has not been initialized");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("There is no element in the table!");

getchar();

return ERROR;

}

for(i=0;i<L->length;i++){

if(L->elem[i] == e)

return (i+1); //Subscript +1 is the bit sequence

}

return 0; // element not found

}

/\*\*

\* Function name: PriorElem

\* Function parameters: linear table pointer L, searched data cur\_e, precursor pre\_e;

\* Function function: Find the first data in L that is the same as cur\_e, and return its predecessor

\* Return value: If successful, return the first data same as cur, and return its first predecessor; if there is no predecessor and other conditions, return ERROR

\*\*/

status PriorElem(SqList \*L,ElemType cur,ElemType &pre\_e){

int i;

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("There is no element in the table!");

getchar();

return ERROR;

}

for(i=0;i<L->length;i++){

if(L->elem[i] == cur){ //Most loops only do this step

if(i==0){ //the first element in the table has no predecessor

printf("This element has no predecessor!");

getchar();

return ERROR;

}

else {

pre\_e = L->elem[i-1];

return OK;

}

}

}

printf("This element was not found!");

getchar();

return ERROR;

}

/\*\*

\* Function name: NextElem

\* Function parameters: linear table pointer L, search data cur, rear drive next\_e;

\* Function: Find the first data in L that is the same as cur, and return its successor

\* Return value: If successful, return the first data that is the same as cur, and return its successor; if there is no follower and other conditions, return ERROR

\*\*/

status NextElem(SqList \*L,ElemType cur,ElemType &next\_e){

int i;

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized!");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("There is no element in the table!");

getchar();

return ERROR;

}

for(i=0;i<L->length;i++){

if(L->elem[i] == cur){ //Most loops only do this step

if(i!=L->length-1){

next\_e = L->elem[i+1];

return OK;

}

else{ //the last element in the table has no successor

printf("This element has no successor!");

getchar();

return ERROR;

}

}

}

printf("This element was not found!");

return ERROR;

}

/\*\*

\* Function name: ListInsert

\* Function parameters: linear table pointer L, inserted position i, inserted data element e.

\* Function: Insert a new data element e before the i-th position of L; if it is full, allocate more space according to the allocation increment.

\* Return value: OK for successful insertion, otherwise ERROR

\*\*/

status ListInsert(SqList \*L,int i,ElemType e){

ElemType \*p, \*q;

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized,");

getchar(); return ERROR;

}

if(i<1 || i >L->length+1) {

printf("Subscript out of bounds,");

getchar(); return ERROR;

}

//Allocate additional space when the number of elements overflows

if(L->length >= L->listsize){

newbase = (ElemType \*)realloc(L->elem, (L->listsize+LISTINCREMENT)\* sizeof(ElemType));

if(!newbase){

printf("Memory allocation failed,");

getchar(); return ERROR;

}

L->elem = newbase;

L->listsize += LISTINCREMENT;

}

q = &(L->elem[i-1]);

for(p=&(L->elem[L->length-1]); p>=q; p--){

\*(p+1) = \*p; //The elements from the end of the list to the insertion place advance in sequence, leaving space for the element to be inserted

}

\*q = e; //insert element

L->length++;

return OK;

}

/\*\*

\* Function name: ListDelete

\* Function parameters: linear table pointer L, deleted bit sequence i, address of pointer e.

\* Function function: delete the i-th data element of L, and return its value with e.

\* Return value: return OK if deleted successfully, otherwise return ERROR

\*\*/

status ListDelete(SqList \*L,int i,ElemType &e){

ElemType \*p, \*q;

if(judge[vol-1] !=TRUE){

printf(" This table has not been initialized,");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("There is no element in the table,");

getchar();

return ERROR;

}

if(i<1 || i >L->length) {

printf("Subscript out of bounds,"); return ERROR;

}

q = &(L->elem[i-1]);

e = L->elem[i-1];

for(p=q;p<=&(L->elem[L->length-1]); p++){

\*(p) = \*(p+1); //The elements from the insertion point to the end of the table are reversed one by one

}

L->length--;

return OK;

}

/\*\*

\* Function name: ListTrabverse

\* Function parameters: linear table pointer L

\* Function function: display each element in the linear table in turn.

\* Return value: If successful traversal returns the length of the linear table, otherwise returns ERROR

\*\*/

status ListTrabverse(SqList \*L){

int i;

if(judge[vol-1] !=TRUE){

printf("This table has not been initialized,");

getchar();

return ERROR;

}

if(ListEmpty(L)==TRUE){

printf("There is no element in the table,");

getchar();

return ERROR;

}

printf("\n-----------all elements -----------------------\n");

for(i=0;i<L->length;i++) printf("%d ",L->elem[i]);

printf("\n------------------end ------------------------\n ");

return L->length;

}

/\*\*

\* Function name: LoadList

\* Function parameters: file pointer, linear form List[20]

\* Function function: data loading, input elements for the linear table.

\* Return value: return OK if loading is successful, return ERROR if loading fails

\*\*/

status LoadList(FILE \*fp,SqList List[20]){

int i, j, yl;

printf("Please enter the file name of the data you want to restore:");

scanf("%s", input);

filename=input;

if ((fp=fopen(filename,"r"))==NULL)

{

printf("Failed to open file");

return ERROR;

}

fread(&yl,sizeof(int),1,fp); //read the number of valid tables

for(i=0;i<yl;i++){

IntiaList(&List[i]); //Only initialize yl tables,

fread(&(List[i].length),sizeof(int),1,fp); //read the length of each table

judge[i]=TRUE; //Change the initialized table judgment variable to TRUE

}

for(i=yl;i<20;i++){

judge[i]=FALSE; //Uninitialized table judgment variables are uniformly FALSE

}

for(i=0;i<yl;i++){

for(j=0;j<List[i].length;j++){

fread(&(List[i].elem[j]),sizeof(ElemType),1,fp);

//Allocate additional space when the number of elements overflows

if(List[i].length >= List[i].listsize){

newbase = (ElemType \*)realloc(List[i].elem, (List[i].listsize+LISTINCREMENT)\* sizeof(ElemType));

if(!newbase){

printf("Memory allocation failed"); return ERROR;

}

List[i].elem = newbase;

List[i].listsize += LISTINCREMENT;

}

}

}

//Here, read the data elements from the file one by one to restore the sequence table. For different physical structures, you can restore the physical structure in the memory through the read data elements.

fclose(fp);

return OK;

}

/\*\*

\* Function name: SaveList

\* Function parameters: file pointer, linear form List[20]

\* Function function: save the data in L to the file pointed to by the fp pointer

\* Return value: OK

\*\*/

status SaveList(FILE \*fp,SqList List[20]){

int i,j,yl=0;

for(i=0;i<20;i++){ // count the number of tables with elements

if(judge[i]==TRUE && List[i].length!=0)

yl++;

}

if(yl ==0){ //return an error if there is no valid data

printf("No data to save");

getchar();

return ERROR;

}

time\_t lt;

lt=time(NULL);

filename=ctime(&lt);

filename[strlen(filename)-1]='\0';

strcat(filename,appe); //The file name string named at the above generation time

pc=&filename[0];

while(\*pc){ //The file name string is normalized, replace ':' and ' ' with '\_'

if((\*pc==':' ) || (\*pc==' ')){ \*pc='\_'; }

pc++;

}

if ((fp=fopen(filename,"w"))==NULL)

{

printf("Failed to open the file");

getchar();getchar();

return ERROR;

}

fwrite(&yl,sizeof(int),1,fp); //The number of valid tables written

for(i=0;i<20;i++){ //write the table length of each valid table

if(judge[i]==TRUE && List[i].length!=0)

fwrite(&(List[i].length),sizeof(int),1,fp);

}

for(i=0;i<20;i++){ // loop to write the elements of each valid table

if(judge[i]==TRUE && List[i].length!=0)

for(j=0;j<List[i].length;j++)

fwrite(&(List[i].elem[j]),sizeof(ElemType),1,fp);

}

// Access data elements one by one by traversing and write to the file

fclose(fp);

return OK;

}

/\*\*

\* Function name: ChooseList

\* Function parameters: the table position vol to be replaced, the file pointer, the linear form List[20]

\* Function function: point L to the table subscript vol-1 to be operated

\* Return value: ERROR on failure, OK on success

\*\*/

status ChooseList(int vol, SqList \*\*L, SqList List[20]){

if(vol<1 || vol>20){

printf("Subscript out of bounds, operation failed!");

getchar();

return ERROR;

}

\*L=&(List[vol-1]);

if(judge[vol-1]==FALSE){

printf("This table has not been initialized"); //Prevent misoperation

getchar(); getchar();

}

return OK;

}

/\*\*

\* Function name: ChartTraverse

\* Function parameters: the table position vol to be replaced, the file pointer, the linear form List[20]

\* Function function: point L to the table subscript vol-1 to be operated

\* Return value: ERROR on failure, OK on success

\*\*/

status ChartTraverse(SqList List[20]){

int i,j,flag=ERROR;

printf("\n-----------all elements -----------------------\n");

for(i=0; i<20; i++){

if(judge[i] !=TRUE){ //uninitialized

printf("List[%d] is not initialized\n", i);

continue;

}

if(List[i].length==0){ //Empty table

printf("There is no element in List[%d]\n", i);

continue;

}

for(j=0;j<List[i].length;j++) printf("%d ",List[i].elem[j]);

flag=OK;

putchar('\n'); //New line after each table traversal

}

printf("\n------------------end ------------------------\n ");

return flag;

}

# Appendix B The source program based on the linear table of the linked storage structure

using namespace std;

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define OK 1

#define TRUE 1

#define FALSE -1

#define ERROR -2

typedef int status;

typedef int ElemType; //Data element type definition

typedef struct lnode{ //element node definition

ElemType data; // data field

struct lnode \*next; // pointer field

}LNode;

typedef struct llist{ //head node definition

int len; //len stores the length of this table

struct lnode \*head, \*tail; //head points to the first element in the table, tail points to the tail element

struct llist \*fore, \*aft; //aft points to the next table header node, and ore points to the previous table header node. This way of definition determines that there should be a blank header node before the first header node.

}Linklist;

int length=0, vol=0; //length is the number of tables, vol is the bit sequence of the current operation table

FILE \*fp;

char \*filename, \*pc, appe[5]=".dat", appf[5]=".txt", input[60]; // are used to generate file names

status IntiaList(Linklist \*\*L);

status DestroyList(Linklist \*L);

status ClearList(Linklist \*L);

status ListEmpty(Linklist \*L);

int ListLength(Linklist \*L);

status GetElem(Linklist \*L,int i,ElemType &e);

status LocateElem(Linklist \*L,ElemType e); //simplified

status PriorElem(Linklist \*L,ElemType cur,ElemType &pre\_e);

status NextElem(Linklist \*L,ElemType cur,ElemType &next\_e);

status ListInsert(Linklist \*L,int i,ElemType e);

status ListDelete(Linklist \*L,int i,ElemType &e);

status ListTrabverse(Linklist \*L); //simplified

status LoadList(FILE \*fp,Linklist \*List);

status SaveList(FILE \*fp,Linklist \*List);

status ChooseList(int vol, Linklist \*\*L, Linklist \*List);

status ChartTraverse(Linklist \*List);

int main(void){

Linklist \*List=NULL, \*L=NULL;

int op=1,i,cur,pre\_e,next\_e;

while(op){

system("cls");printf("\n\n");

printf(" There are %d tables in the system, and the current operation is the %d table\n", length, vol); //Prompt the bit sequence of the currently operated table

printf("Menu for Linear Table On Sequence Structure\n");

printf("----------------------------------------------- --\n");

printf(" 1. IntiaList 7. LocateElem\n");

printf(" 2. DestroyList 8. PriorElem\n");

printf(" 3. ClearList 9. NextElem \n");

printf("4. ListEmpty 10. ListInsert\n");

printf("5. ListLength 11. ListDelete\n");

printf("6. GetElem 12. ListTrabverse\n");

printf("13.LoadList 14.SaveList\n");

printf(" 15.ChooseList 16.ChartTraverse\n");

printf(" 0. Exit\n");

printf("----------------------------------------------- --\n");

printf(" Please choose your operation [0~16]:");

scanf("%d",&op);

switch(op){

case 1:

if(IntiaList(&List)==OK){

printf("Linear table created successfully!");

L=List->aft; vol=1;

}

else printf(" The linear table creation failed!");

getchar();getchar();

break;

case 2:

DestroyList(L);

getchar(); break;

case 3:

ClearList(L);

getchar(); break;

case 4:

i = ListEmpty(L);

if(i==TRUE) printf("The linear table is empty!");

else if(i==FALSE) printf("The linear table is not empty!");

getchar();getchar();

break;

case 5:

if(ListLength(L)==ERROR){

getchar(); getchar(); break;

}

printf(" Table length is %d", L->len);

getchar(); getchar(); break;

case 6:

if(L==NULL){

printf("table not initialized");

getchar(); break;

}

printf("Please enter element sequence i(1<=i<%d):",L->len+1);

scanf("%d",&i);

if(GetElem(L,i,cur)==OK)

printf("The %dth element is %d", i, cur);

getchar();getchar();

break;

case 7:

printf("Please enter the data to be checked:");

scanf("%d", &cur);

i=LocateElem(L,cur);

if(i>0)

printf("%d is the %dth element in the table", cur, i);

if(!i)

printf("This element was not found");

getchar(); getchar(); break;

case 8:

printf("Please enter location data:");

scanf("%d", &cur);

if(PriorElem(L,cur,pre\_e)==OK)

printf("%d's predecessor is %d", cur, pre\_e);

getchar(); getchar(); break;

case 9:

printf("Please enter location data:");

scanf("%d", &cur);

if(NextElem(L,cur,next\_e)==OK)

printf("%d's successor is %d", cur, next\_e);

getchar(); getchar(); break;

case 10:

if(L==NULL){

printf("table not initialized");

getchar(); break;

}

printf(" Please enter the insertion position i(1<=i<=%d):", L->len+1);

scanf("%d", &i);

printf("Please input insert data e:");

scanf("%d", &cur);

if(ListInsert(L,i,cur)==ERROR) printf("Insert failed");

else printf("insert successfully");

getchar(); getchar(); break;

case 11:

if(L==NULL){

printf("table not initialized");

getchar(); break;

}

printf(" Please enter the deletion position i (1<=i<=%d)", L->len);

scanf("%d", &i);

if(ListDelete(L,i,cur)==ERROR)

printf("Delete failed");

else

printf("Deleted successfully, the deleted element is %d", cur);

getchar(); getchar(); break;

case 12:

if(ListTrabverse(L)==ERROR) printf(" Traversal failed!\n");

getchar(); getchar(); break;

case 13:

if(LoadList(fp,List)==OK){

printf("File loaded successfully");

}

getchar(); getchar(); break;

case 14:

if(SaveList(fp,List)==OK){

printf("File saved successfully");

}

getchar(); getchar(); break;

case 15:

printf("Please enter the bit sequence vol of the table to be operated: (1<=vol<=%d)", length+1);

scanf("%d", &vol);

ChooseList(vol, &L, List);

getchar(); break;

case 16:

if(ChartTraverse(List)==ERROR) printf("Form traversal failed!");

getchar();getchar();

case 0:

break;

}//end of switch

}//end of while

printf(" Welcome to use this system next time!");

return 0;

}//end of main()

/\*\*

\* Function name: InitiaList

\* Function parameter: the address of the linked list pointer L

\* Function function: construct an empty linked list

\* Return value: return OK if successfully constructed, otherwise return ERROR

\*\*/

status IntiaList(Linklist \*\*L){

/\*Initialize the big table header - start\*/

Linklist \*p;

p =(Linklist\*)malloc(sizeof(Linklist));

if(!p){

printf("Memory allocation failed"); return ERROR;

}

p->fore=NULL;

p->head = p->tail = NULL;

p->len = 0;

/\*Initialize the large table header-end\*//\*Initialize the first small table header-start\*/

p->aft =(Linklist\*)malloc(sizeof(Linklist));

if(!p->aft){

printf("Memory allocation failed"); return ERROR;

}

p->aft->aft = NULL;

p->aft->head = p->aft->tail = NULL;

p->aft->fore = p;

p->aft->len = 0;

/\* Initialize the first small table header - end \*/

length = vol = 1; \*L=p;

return OK;

}

/\*\*

\* Function name: DestroyList

\* Function parameter: linked list pointer L

\* Function function: delete the linked list

\* Return value: return OK if destroyed successfully, otherwise return ERROR

\*\*/

status DestroyList(Linklist \*L){

if(L==NULL){

printf("table not initialized");

getchar();

return ERROR;

}

LNode \*p;

if(L->head!=NULL){

p = L->head;

while(p->next != NULL){ //Delete all elements except the first element

L->head = p->next;

free(p); p = L->head;

}

free(p); //Delete the first element

L->head=L->tail=NULL;

}

L->fore->aft=L->aft;

if(L->aft!=NULL)

L->aft->fore=L->fore;

free(L); //Delete the head node, connect the front and back tables

L=NULL; length--;

if(vol>length) vol=length;

printf("Linked table %d has been deleted!", vol); getchar(); getchar();

return OK;

}

/\*\*

\* Function name: ClearList

\* Function parameter: linked list pointer L

\* Function function: chained linear table

\* Return value: return OK if it is empty successfully, otherwise return ERROR

\*\*/

status ClearList(Linklist \*L){

if(L==NULL){

printf("table not initialized");

getchar();

return ERROR;

}

LNode \*p;

if(L->head!=NULL){ //The table is not empty

p = L->head;

while(p->next != NULL){ //Delete all elements except the first element

L->head = p->next;

free(p); p = L->head;

}

free(p); //Delete the first element

L->head=L->tail=NULL;

L->len=0; //reset the table length record to zero

}

printf("The table has been cleared!"); getchar();

return OK;

}

/\*\*

\* Function name: ListEmpty

\* Function parameter: linked list pointer L

\* Function function: determine whether the linked list is empty

\* Return value: Return TRUE if L is an empty list, otherwise return FALSE

\*\*/

status ListEmpty(Linklist \*L){

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

int flag=TRUE;

if(L->head!=NULL)

flag = FALSE; // table length is not 0

return flag;

}

/\*\*

\* Function name: ListLength

\* Function parameter: linked list pointer L

\* Function function: calculate the number of data elements in the linked list L;

\* Return value: returns the number of data elements in L

\*\*/

int ListLength(Linklist \*L){

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

return L->len; //return 0 if the table is empty, and return length if it is not empty

}

/\*\*

\* Function name: GetElem

\* Function parameters: linked list pointer L, e is the address of the i-th data;

\* Function function: find and display the value of the i-th element in L;

\* Return value: return the value of the i-th element if successful, otherwise return ERROR

\*\*/

status GetElem(Linklist \*L,int i,ElemType &e){

LNode \*p; int j=1;

if(L->head==NULL){

printf("There is no element in the table!"); getchar();

return ERROR;

}

if(i<1 || i>L->len){

printf(" Subscript out of bounds!"); getchar();

return ERROR;

}

for(p=L->head;j<i;p=p->next,j++){}

e = p->data; return OK;

}

/\*\*

\* Function name: LocateElem

\* Function parameters: linked list pointer L, the element value to be compared;

\* Function function: Find the bit sequence of the same data as e in L; the bit sequence starts from 1

\* Return value: return the bit sequence of the first data that is the same as e if successful, return 0 if it does not exist, and return ERROR in other cases

\*\*/

status LocateElem(Linklist \*L,ElemType e){

LNode \*p;

char i=1;

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

if(L->head==NULL){

printf("There is no element in the table!"); getchar();

return ERROR;

}

for(p=L->head;p!=NULL;p=p->next,i++)

if(p->data==e) return i;

return 0; // element not found

}

/\*\*

\* Function name: PriorElem

\* Function parameters: linked list pointer L, searched data cur\_e, predecessor pre\_e;

\* Function function: Find the first data in L that is the same as cur\_e, and return its predecessor

\* Return value: If successful, return the first data same as cur, and return its first predecessor; if there is no predecessor and other conditions, return ERROR

\*\*/

status PriorElem(Linklist \*L,ElemType cur,ElemType &pre\_e){

LNode \*p, \*q;

char i=1, j=0;

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

if(L->head==NULL){

printf("There is no element in the table!"); getchar();

return ERROR;

}

for(p=L->head;p!=NULL;p=p->next,i++)

if(p->data==cur){

if(i==1){ //the first element in the table has no predecessor

printf("This element has no predecessor!"); getchar();

return ERROR;

}

else {

for(q=L->head;j<i-2;q=q->next,j++){}

pre\_e= q->data;

return OK;

}

}

printf("The element was not found!"); getchar();

return ERROR;

}

/\*\*

\* Function name: NextElem

\* Function parameters: linked list pointer L, searched data cur, rear drive next\_e;

\* Function: Find the first data in L that is the same as cur, and return its successor

\* Return value: If successful, return the first data that is the same as cur, and return its successor; if there is no follower and other conditions, return ERROR

\*\*/

status NextElem(Linklist \*L,ElemType cur,ElemType &next\_e){

LNode \*p;

char i=1;

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

if(L->head==NULL){

printf("There is no element in the table!"); getchar();

return ERROR;

}

for(p=L->head;p!=NULL;p=p->next,i++)

if(p->data==cur){

if(p->next==NULL){ //the first element in the table has no predecessor

printf("This element has no successor!"); getchar();

return ERROR;

}

else {

next\_e=p->next->data; return OK;

}

}

printf("The element was not found!"); getchar();

return ERROR;

}

/\*\*

\* Function name: ListInsert

\* Function parameters: linked list pointer L, inserted position i, inserted data element e.

\* Function: Insert a new data element e before the i-th position of L; if it is full, allocate more space according to the allocation increment.

\* Return value: OK for successful insertion, otherwise ERROR

\*\*/

status ListInsert(Linklist \*L,int i,ElemType e){

LNode \*p=L->head, \*s;

char j=1;

if(i<1 || i>L->len+1) { // There is a place to insert after the tail element

printf("Subscript out of bounds,");

getchar(); return ERROR;

}

s = (LNode\*)malloc(sizeof(LNode)); //Create a node

s->data= e;

if(i==1){ //When inserting the first element, directly let it be pointed by the head node

s->next=p; L->head=s;

}

else{ //insert element elsewhere

for(;j<i-1;p=p->next,j++);

s->next=p->next; //Let the new node point to the existing node at the current position

p->next=s; //The predecessor of the existing node points to the new node

if(i==L->len+1) //The end of the table points to NULL when inserting the tail element

s->next=NULL;

}

L->len++; return OK;

}

/\*\*

\* Function name: ListDelete

\* Function parameters: linked list pointer L, deleted bit sequence i, address of pointer e.

\* Function function: delete the i-th data element of L, and return its value with e.

\* Return value: return OK if deleted successfully, otherwise return ERROR

\*\*/

status ListDelete(Linklist \*L,int i,ElemType &e){

LNode \*p=L->head, \*s;

char j=1;

if(L->head==NULL){

printf("There is no element in the table!"); getchar();

return ERROR;

}

if(i<1 || i>L->len) {

printf("Subscript out of bounds,");

getchar(); return ERROR;

}

if(i==1){ //Let the head node point to its successor when deleting the head

L->head=p->next; e=p->data; free(p); p=NULL;

}

else{ //delete element elsewhere

for(;j<i-1;p=p->next,j++);

s=p->next; p->next=s->next; //The predecessor of the current node points to its successor

e=s->data; free(s); s=NULL;

}

L->len--; return OK;

}

/\*\*

\* Function name: ListTrabverse

\* Function parameter: linked list pointer L

\* Function function: display each element in the linear table in turn.

\* Return value: If successful traversal returns the length of the linear table, otherwise returns ERROR

\*\*/

status ListTrabverse(Linklist \*L){

LNode \*p=L->head;

if(L==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

if(L->head==NULL){

printf("There is no element in the table,"); getchar();

return ERROR;

}

printf("\n-----------all elements -----------------------\n");

for(;p!=NULL;p=p->next) printf("%d ",p->data);

printf("\n------------------end ------------------------\n ");

return L->len;

}

/\*\*

\* Function name: LoadList

\* Function parameters: file pointer, chained form List

\* Function function: data loading, input elements for the linear table.

\* Return value: return OK if loading is successful, return ERROR if loading fails

\*\*/

status LoadList(FILE \*fp,Linklist \*List){

int i,yl;

Linklist \*p;

LNode \*r;

if(List==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

printf("Please enter the file name of the data you want to restore:");

scanf("%s", input);

filename=input;

if ((fp=fopen(filename,"r"))==NULL)

{

printf("Failed to open file"); return ERROR;

}

fread(&yl,sizeof(int),1,fp); //read the number of valid tables

length=yl;

for(i=0,p=List; i<yl; i++,p=p->aft){

p->aft =(Linklist\*)malloc(sizeof(Linklist));

if(!p->aft){

printf("Memory allocation failed"); return ERROR;

}

p->aft->fore = p;

p->aft->aft = NULL;

p->aft->head = p->aft->tail = NULL;

fread(&(p->aft->len),sizeof(int),1,fp); //read the length of each table

}

for(p=List->aft;p!=NULL;p=p->aft){

//printf("%d\n",p->len);

p->head=(LNode \*)malloc(sizeof(LNode));

p->head->next=NULL;

fread(&(p->head->data), sizeof(ElemType), 1, fp);

for(i=1,r=p->head; i<p->len; i++,r=r->next){

r->next=(LNode \*)malloc(sizeof(LNode));

r->next->next=NULL;

fread(&(r->next->data), sizeof(ElemType), 1, fp);

}

}

ChartTraverse(List);

if(vol>length) vol=length;

//Here, read the data elements from the file one by one to restore the sequence table. For different physical structures, you can restore the physical structure in the memory through the read data elements.

fclose(fp); return OK;

}

/\*\*

\* Function name: SaveList

\* Function parameters: file pointer fp, chained form pointer List

\* Function function: save the data in L to the file pointed to by the fp pointer

\* Return value: OK

\*\*/

status SaveList(FILE \*fp,Linklist \*List){

int yl=0;

Linklist \*p;

LNode \*q;

if(List==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

for(p=List->aft;p!=NULL;p=p->aft){ // count the number of tables with elements

if(p->len!=0) yl++;

}

if(yl ==0){ //return an error if there is no valid data

printf("No data to save"); getchar();

return ERROR;

}

time\_t lt;

lt=time(NULL);

filename=ctime(&lt);

filename[strlen(filename)-1]='\0';

strcat(filename,appf); //The file name string named at the above generation time

pc=&filename[0];

while(\*pc){ //The file name string is normalized, replace ':' and ' ' with '\_'

if((\*pc==':' ) || (\*pc==' ')){ \*pc='\_'; }

pc++;

}

if ((fp=fopen(filename,"w"))==NULL)

{

printf("Failed to open the file"); getchar();getchar();

return ERROR;

}

fwrite(&yl,sizeof(int),1,fp); //The number of valid tables written

for(p=List->aft;p!=NULL;p=p->aft){ //Write the table length of each valid table

if(p->len!=0)

fwrite(&(p->len), sizeof(int), 1, fp);

}

for(p=List->aft;p!=NULL;p=p->aft){ // loop to write the elements of each valid list

if(p->len!=0)

for(q=p->head;q!=NULL;q=q->next){

fwrite(&(q->data), sizeof(ElemType), 1, fp);

printf("%d ", q->data);

}

}

//Access data elements one by one by traversing and write to the file

fclose(fp); return OK;

}

/\*\*

\* Function name: ChooseList

\* Function parameters: the table position vol to be replaced, the chained form pointer L, the chained form pointer List

\* Function function: point L to the table subscript vol-1 to be operated

\* Return value: ERROR on failure, OK on success

\*\*/

status ChooseList(int vol, Linklist \*\*L, Linklist \*List){

Linklist \*p;

if(List==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

if(vol<1 || vol>length+1){

printf("Subscript out of bounds, operation failed!"); getchar();

return ERROR;

}

if(vol==length+1){

for(p=List->aft;p->aft!=NULL;p=p->aft);

p->aft=(Linklist\*)malloc(sizeof(Linklist));

if(!p->aft){

printf("Memory allocation failed"); return ERROR;

}

p->aft->aft = NULL;

p->aft->head = p->aft->tail = NULL; // At this time, create a new node through the 'front' tail node operation

p->aft->fore = p;

p= p->aft; //L points to the tail node

p->len = 0; length++; vol = length;

}

else {

char i=1;

for(p=List->aft;i<vol;p=p->aft,i++){}

}

\*L=p; return OK;

}

/\*\*

\* Function name: ChartTraverse

\* Function parameter: Linked form pointer List

\* Function function: traverse the entire form, and explain separately when there is no element in the table

\* Return value: ERROR on failure, OK on success

\*\*/

status ChartTraverse(Linklist \*List){

char i=0, flag=ERROR;

Linklist \*p;

LNode \*q;

if(List==NULL){

printf("table not initialized"); getchar();

return ERROR;

}

printf("\n-----------all elements -----------------------\n");

for(p=List->aft;p!=NULL;p=p->aft,i++){

if(p->len==0){ //empty table

printf("There is no element in List[%d]\n", i+1);

continue;

}

for(q=p->head;q!=NULL;q=q->next) printf("%d ",q->data);

flag=OK;

putchar('\n'); //New line after each table traversal

}

printf("\n------------------end ------------------------\n ");

return flag;

}

# Appendix C Source Program Based on Binary Linked List Binary Tree Implementation

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <malloc.h>

#include <time.h>

/\*---------page 10 on textbook ---------\*/

#define TRUE 1

#define FALSE -1

#define OK 1

#define ERROR -2

typedef int Status; // is the type of the function, its value is the function result status code

typedef int ElemType; //Data element type definition

typedef struct BiTNode

{

ElemType key;

char data;

struct BiTNode \*lchild, \*rchild;//left and right subtree pointers

}BiTNode, \*BitTree;

Status InitBiTree(BitTree \*L);

Status DestroyBiTree(BitTree \*L);

Status CreateBiTree(BitTree \*L);

Status ClearBiTree(BitTree \*L);

Status BiTreeEmpty(BitTree L);

Status BiTreeDepth(BitTree L);

BitTree Root(BitTree L);

char Value(BitTree L, ElemType key);

Status Assign(BitTree \*L, ElemType key, char data);

BitTree\* Parent(BitTree L, ElemType key, BitTree \*node);

BitTree LeftChild(BitTree L, ElemType key, BitTree \*node);

BitTree RightChild(BitTree L, ElemType key, BitTree \*node);

BitTree LeftSibling(BitTree L, ElemType key, BitTree \*node);

BitTree RightSibling(BitTree L, ElemType key, BitTree \*node);

BitTree FindNode(BitTree L, ElemType key, BitTree \*node);

Status InsertChild(BitTree \*L, BitTree p, int LR, BitTree c);

Status DeleteChild(BitTree \*L, BitTree p, int LR);

Status PreOrderTraverse(BitTree L);

Status InOrderTraverse(BitTree L);

Status PostOrderTraverse(BitTree L);

Status LevelOrderTraverse(BitTree L);

Status SaveBiTree(FILE \*fp, BitTree L);

Status ReadBiTree(FILE \*fp, BitTree \*L);

BitTree Search(BitTree L, ElemType key);

char\* Filename(void);

FILE \*fp;

char \*filename, \*pc, appe[5]=".dat",input[60]; // are used to generate file names

int main(void)

{

BitTree L=NULL,\*p,c=NULL;

FILE \*fp;

char ch,\*filename,data;

int LR,n,op = 1;

ElemType key;

p = (BitTree \*)malloc(sizeof(BitTree));

InitBiTree(&c);

while (op) {

\*p = NULL;

system("cls");printf("\n\n");

printf("Menu for Linear Table On Sequence Structure\n");

printf("----------------------------------------------- --\n");

printf(" 1. InitBiTree 12. RightChild\n");

printf(" 2. DestroyBiTree 13. LeftSibling\n");

printf(" 3. CreateBiTree 14. RightSibling\n");

printf(" 4. ClearBiTree 15. InsertChild\n");

printf(" 5. BiTreeEmpty 16. DeleteChild\n");

printf("6. BiTreeDepth 17. PreOrderTraverse\n");

printf("7. Root 18. InOrderTraverse\n");

printf(" 8. Value 19. PostOrderTraverse\n");

printf(" 9. Assign 20. LevelOrderTraverse\n");

printf(" 10. Parent 21. SaveBiTree\n");

printf(" 11. LeftChild 22. ReadBiTree\n");

printf(" 0. Exit\n");

printf("\n");

printf("----------------------------------------------- --\n");

printf("Please choose your operation [0~22] ");

scanf("%d", &op);

switch(op) {

case 1:

if (InitBiTree(&L) == OK) printf("Initialize the binary tree successfully");

else printf("Failed to initialize binary tree");

getchar(); getchar(); break;

case 2:

if (!L) printf("Binary tree is not initialized");

else if (DestroyBiTree(&L) == OK)

{

printf("Binary tree destroyed successfully"); L = NULL;

}

else printf("Binary tree destruction failed");

getchar(); getchar(); break;

case 3:

if (!L) printf("Binary tree is not initialized");

else if (L->key != -1) printf("Binary tree already exists"); //It is more reliable to directly use the condition of judging the empty tree

else if (CreateBiTree(&L) == OK)printf("Create successfully");

else printf("Failed to create");

getchar(); getchar(); break;

case 4:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else if (ClearBiTree(&L) == OK) printf("Cleared successfully");

else printf("Empty failed");

getchar(); getchar(); break;

case 5:

if (!L) printf("Binary tree is not initialized");

else if (BiTreeEmpty(L) ==TRUE) printf("The binary tree is empty");

else printf("Binary tree is not empty");

getchar(); getchar(); break;

case 6:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty ");//Empty tree is discussed separately

else printf("Binary tree depth is %d", BiTreeDepth(L));

getchar(); getchar(); break;

case 7:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else if (Root(L)) printf("The root node key is %d", L->key);

getchar(); getchar(); break;

case 8:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Please enter the target node key\n");

scanf("%d", &key); getchar();

data=Value(L, key); //The assignment expression is used as a judgment condition to report warning in the current compilation environment, so it is proposed as an independent statement

if (data) printf("key's data is %c", data);

else printf("Failed to find node"); //The reason for the failure will be discussed in the function

}

getchar(); getchar(); break;

case 9:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Please enter the target node key\n");

scanf("%d", &key); getchar();

printf("Please input new data\n");

scanf("%c", &data); getchar();

if (Assign(&L, key, data) == OK) printf("Assignment succeeded");

else printf("Assignment failed");

}

getchar(); getchar(); break;

case 10:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Location node key:");

scanf("%d", &key); getchar();

if(!Search(L,key)) printf("No target node found");

else if (Parent(L, key ,p)!=NULL)

printf("parent key is %d", (\*p)->key);

else printf("Return failed");

}

getchar(); getchar(); break;

case 11:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Location node key:");

scanf("%d", &key); getchar();

if(!Search(L,key)) printf("No target node found");

else if (LeftChild(L, key, p)!=NULL)

printf("Left child key is %d",(\*p)->key);

else printf("The node has no left child and failed to return");

}

getchar(); getchar(); break;

case 12:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Location node key:");

scanf("%d", &key); getchar();

if(!Search(L,key)) printf("No target node found");

else if (RightChild(L, key, p))

printf("The right child key is %d",(\*p)->key);

else printf("The node has no right child and failed to return");

}

getchar(); getchar(); break;

case 13:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Location node key:");

scanf("%d", &key); getchar();

if(!Search(L,key)) printf("No target node found");

else if (LeftSibling(L, key, p))

printf("Left sibling key is %d",(\*p)->key);

else printf("The node has no left sibling and failed to return");

}

getchar(); getchar(); break;

case 14:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else{ //The recursive function is not convenient to write the input and error judgment conditions, so this part is moved out of the function before and after

while ((ch = getchar()) != EOF && ch != '\n');

printf("Location node key:");

scanf("%d", &key); getchar();

if(!Search(L,key)) printf("No target node found");

else if (RightSibling(L, key, p))

printf("The right sibling key is %d",(\*p)->key);

else

printf("The node has no right sibling and failed to return");

}

getchar(); getchar(); break;

case 15:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

ClearBiTree(&(c));

while ((ch = getchar()) != EOF && ch != '\n');

printf("Create a binary tree, enter CreateBiTree\n");

CreateBiTree(&(c));

printf("insert key at position p:");

scanf("%d", &key);

FindNode(L, key, p);//Find the node corresponding to the key and return its pointer

while ((ch = getchar()) != EOF && ch != '\n');

if (\*p == NULL) printf("No target node found");

else

{

printf("Enter the value of LR:");

scanf("%d", &LR); getchar();

if (InsertChild(&L, \*p, LR, c) == OK)

printf("insert successfully");

else printf("Insert failed");

}

}

getchar(); getchar(); break;

case 16:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

while ((ch = getchar()) != EOF && ch != '\n');

printf("Delete the key at position p:");

scanf("%d", &key); getchar();

FindNode(L, key, p);

if (\*p == NULL) printf("No target node found");

else

{

printf("Please enter the value of LR:");

scanf("%d", &LR); getchar();

n = DeleteChild(&L, \*p, LR);

if (n == OK) printf("deleted successfully");

else if (n == ERROR) printf("Delete failed");

else printf("cannot delete empty subtree");

}

}

getchar(); getchar(); break;

case 17:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

printf("\n----------------PreOrderTraverse all elements------------------\n");

PreOrderTraverse(L);

printf("\n----------------------------end---------------- -----------------\n");

}

getchar(); getchar(); break;

case 18:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

printf("\n----------------InOrderTraverse all elements------------------\n");

InOrderTraverse(L);

printf("\n----------------------------end---------------- -----------------\n");

}

getchar(); getchar(); break;

case 19:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

printf("\n----------------PostOrderTraverse all elements------------------\n");

PostOrderTraverse(L);

printf("\n----------------------------end---------------- -----------------\n");

}

getchar(); getchar(); break;

case 20:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

printf("\n----------------LevelOrderTraverse all elements------------------\n");

LevelOrderTraverse(L);

printf("\n----------------------------end---------------- -----------------\n");

}

getchar(); getchar(); break;

case 21:

if (!L) printf("Binary tree is not initialized");

else if (L->key==-1) printf("The binary tree is empty");

else {

filename = Filename(); //Automatically generate filename with system time

if ((fp = fopen(filename, "w")) == NULL){

printf("Failed to open the file"); return ERROR;

}

else if (SaveBiTree(fp,L) == OK){

fclose(fp); printf("Save the file successfully");

}

else printf("Failed to save file");

}

getchar(); getchar(); break;

case 22:

if (!L) printf("Binary tree is not initialized");

else {

if (BiTreeEmpty(L) != TRUE)

ClearBiTree(&L);

free(L);

L=NULL;

printf("Please enter the file name to read:");

while ((ch = getchar()) != EOF && ch != '\n');

scanf("%s", input); getchar();

filename=input; //Use input string to transition between buffer and filename string

if ((fp = fopen(filename, "r")) == NULL)

printf("Failed to open the file");

else if (ReadBiTree(fp, &L) == OK)

{

fclose(fp); printf("File read successfully");

}

else

printf("Failed to read file");

}

getchar(); getchar(); break;

case 0:

break;

}

}

printf("Welcome to use this system next time!\n");

getchar();getchar();

return 0;

}

/\*\*

\* Function name: InitBiTree

\* Function parameter: the address of the binary tree head pointer L

\* Function function: Construct a list of 20 empty binary tree headers

\* Return value: return OK if successfully constructed, otherwise return ERROR

\*\*/

Status InitBiTree(BitTree \*L){

if (\*L) return ERROR;

\*L = (BitTree)malloc(sizeof(BiTNode));

if (\*L)

{

(\*L)->key = -1; //initialize to an empty tree

(\*L)->lchild = NULL; (\*L)->rchild = NULL; //The child pointer is empty

return OK;

}

return ERROR;

}

/\*\*

\* Function name: DestroyBiTree

\* Function parameter: the address of the binary tree head pointer L

\* Function function: delete binary tree

\* Return value: return OK if destroyed successfully, otherwise return ERROR

\*\*/

Status DestroyBiTree(BitTree \*L){

if ((\*L)==NULL) return ERROR;

if ((\*L)->lchild != NULL)

DestroyBiTree(&(\*L)->lchild);

if ((\*L)->rchild != NULL)

DestroyBiTree(&(\*L)->rchild);

free(\*L); //Recursively release nodes

return OK;

}

/\*\*

\* Function name: CreateBiTree

\* Function parameter: the address of the binary tree head pointer L

\* Function function: build a binary tree

\* Return value: return OK if successfully established, otherwise return ERROR

\*\*/

Status CreateBiTree(BitTree \*L){

int keyin;

if(!(\*L)) return ERROR;

printf("Please enter the node keyword, -1 means not to create\n"); //-1 is used as a NULL mark, and it is also the beginning of a recursion to the root of the tree

scanf("%d", &keyin); getchar();

if (keyin==-1)

{ (\*L) = NULL; return 0;}

else

{

if (!((\*L = (BiTNode \*)malloc(sizeof(BiTNode))))){

printf("Space allocation failed"); return ERROR;

}

(\*L)->key = keyin;

printf("Please enter the node value:\n");

scanf("%c", &((\*L)->data)); getchar();

CreateBiTree(&((\*L)->lchild));

CreateBiTree(&((\*L)->rchild)); //recursively create nodes

}

return OK;

}

/\*\*

\* Function name: ClearBiTree

\* Function parameter: the address of the binary tree head pointer L

\* Function function: reset the binary tree

\* Return value: return OK if it is empty successfully, otherwise return ERROR

\*\*/

Status ClearBiTree(BitTree \*L){

if (!DestroyBiTree(&(\*L))) return ERROR;

\*L = NULL;

if (!InitBiTree(&(\*L))) return ERROR;

return OK;

}

/\*\*

\* Function name: BiTreeEmpty

\* Function parameter: Binary tree head pointer L

\* Function function: determine whether the binary tree is empty

\* Return value: empty TRUE, non-empty FALSE

\*\*/

Status BiTreeEmpty(BitTree L){

if (L->key==-1) return TRUE;

else return FALSE;

}

/\*\*

\* Function name: BiTreeDepth

\* Function parameter: Binary tree head pointer L

\* Function function: Find the depth of the binary tree

\* Return value: depth depth

\*\*/

int BiTreeDepth(BitTree L){

int depth = 0,lchilddepth,rchilddepth;

if (L)

{

lchilddepth=BiTreeDepth((L->lchild));

rchilddepth=BiTreeDepth((L->rchild));

depth = rchilddepth >= lchilddepth ? rchilddepth + 1 : lchilddepth + 1;

}

return depth;

}

/\*\*

\* Function name: Root

\* Function parameter: Binary tree head pointer L

\* Function function: find the root node of the binary tree

\* Return value: root node L

\*\*/

BitTree Root(BitTree L){

return L;

}

/\*\*

\* Function name: Value

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: Find the data of the node whose keyword is key in the binary tree

\* Return value: return data if successful, return 0 if failed

\*\*/

char Value(BitTree L, ElemType key){

BitTree M = Search(L, key);

if(M) return M->data;

printf("The node was not found"); return 0;

}

/\*\*

\* Function name: Assign

\* Function parameters: address of binary tree head pointer L, keyword key, data field data

\* Function function: assign the data of the node whose keyword is key in the binary tree to value

\* Return value: return OK if the assignment is successful, otherwise return ERROR

\*\*/

Status Assign(BitTree \*L, ElemType key, char data){

if (\*L==NULL) return 0;

BitTree M = Search(\*L, key);

if(M){ M->data=data; return OK; }

printf("The node was not found"); return 0;

}

/\*\*

\* Function name: Parent

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: Find the parents of the node whose keyword is key in the binary tree

\* Return value: return parent node pointer successfully, otherwise return NULL

\*\*/

BitTree\* Parent(BitTree L, ElemType key, BitTree \*node)

{

if(L==NULL) return NULL;

if (L->key == key){

printf("The target node is the head node and has no parents"); return NULL;

}

if ((L->lchild) != NULL)

if (L->lchild->key == key)

{\*node = L; return node;}

if ((L->rchild)!=NULL)

if (L->rchild->key == key)

{ \*node = L; return node;}

if (Parent(L->lchild, key ,node)) return node;

if (Parent(L->rchild, key ,node)) return node;

return NULL;

}

/\*\*

\* Function name: LeftChild

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: find the left child of the node whose keyword is key in the binary tree

\* Return value: successfully returns the left child node pointer, otherwise returns NULL

\*\*/

BitTree LeftChild(BitTree L, ElemType key, BitTree \*node)

{

if (L == NULL) return NULL;

if (L->key == key)

if (L->lchild != NULL)

{ \*node = L->lchild; return \*node; }

if (LeftChild(L->lchild, key, node)) return \*node;;

if (LeftChild(L->rchild, key, node)) return \*node;

return NULL;

}

/\*\*

\* Function name: RightChild

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: Find the right child of the node whose keyword is key in the binary tree

\* Return value: successfully returns the right child node pointer, otherwise returns NULL

\*\*/

BitTree RightChild(BitTree L, ElemType key, BitTree \*node)

{

if (L == NULL) return NULL;

if (L->key == key)

if (L->rchild != NULL)

{ \*node = L->rchild; return \*node; }

if (RightChild(L->lchild, key, node)) return \*node;

if (RightChild(L->rchild, key, node)) return \*node;

return NULL;

}

/\*\*

\* Function name: LeftSibling

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: find the left brother of the node whose keyword is key in the binary tree

\* Return value: successfully returns the left sibling node pointer, otherwise returns NULL

\*\*/

BitTree LeftSibling(BitTree L, ElemType key, BitTree \*node)

{

if (L == NULL) return NULL;

if (L->rchild != NULL)

if (L->rchild->key == key && L->lchild != NULL)

{\*node = L->lchild; return \*node; }

if (LeftSibling(L->lchild, key,node)) return \*node;

if (LeftSibling(L->rchild, key, node)) return \*node;

return NULL;

}

/\*\*

\* Function name: RightSibling

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: Find the right brother of the node whose keyword is key in the binary tree

\* Return value: return right sibling node pointer successfully, otherwise return NULL

\*\*/

BitTree RightSibling(BitTree L, ElemType key, BitTree \*node)

{

if (L == NULL) return NULL;

if (L->lchild != NULL)

if (L->lchild->key == key && L->rchild != NULL)

{ \*node = L->rchild; return \*node; }

if (RightSibling(L->lchild, key, node)) return \*node;

if (RightSibling(L->rchild, key, node)) return \*node;

return NULL;

}

/\*\*

\* Function name: InsertChild (with changes)

\* Function parameters: address of binary tree head pointer L, binary tree node pointer p, positioning variable LR, binary tree node pointer c

\* Function function: insert subtree c at node p

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status InsertChild(BitTree \*L, BitTree p, int LR, BitTree c)

{

if (LR == 0){

if (p->rchild != NULL){

printf("Binary tree C contains left subtree\n"); return ERROR;

}

p->lchild = c; return OK;

}

else if (LR == 1){

if (p->rchild != NULL){

printf("Binary tree C contains right subtree\n"); return ERROR;

}

p->rchild = c; return OK;

}

else return ERROR;

}

/\*\*

\* Function name: DeleteChild

\* Function parameters: address of binary tree head pointer L, binary tree node pointer p, positioning variable LR

\* Function function: delete left/right subtree c at node p

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status DeleteChild(BitTree \*L, BitTree p, int LR)

{

if (LR == 0)

{

if ((\*L)->lchild == NULL) return FALSE;

DestroyBiTree(&((\*L)->lchild));

p->lchild = NULL;

}

else if (LR == 1)

{

if ((\*L)->rchild == NULL) return FALSE;

DestroyBiTree(&((\*L)->rchild));

p->rchild = NULL;

}

else return ERROR;

return OK;

}

/\*\*

\* Function name: PreOrderTraverse

\* Function parameter: Binary tree head pointer L

\* Function function: preorder traversal of the binary tree

\* Return value: OK

\*\*/

Status PreOrderTraverse (BitTree L)

{

if (L == NULL)

return 0;

printf("%d-%c ", L->key,L->data);

if (L->lchild != NULL)

PreOrderTraverse(L->lchild);

if (L->rchild != NULL)

PreOrderTraverse(L->rchild);

return OK;

}

/\*\*

\* Function name: InOrderTraverse

\* Function parameter: Binary tree head pointer L

\* Function function: Inorder traversal of binary tree

\* Return value: OK

\*\*/

Status InOrderTraverse(BitTree L)

{

if (L == NULL)

return 0;

if (L->lchild != NULL)

InOrderTraverse(L->lchild);

printf("%d-%c ", L->key,L->data);

if (L->rchild != NULL)

InOrderTraverse(L->rchild);

return OK;

}

/\*\*

\* Function name: PostOrderTraverse

\* Function parameter: Binary tree head pointer L

\* Function function: post-order traversal of the binary tree

\* Return value: OK

\*\*/

Status PostOrderTraverse (BitTree L)

{

if (L == NULL)

return 0;

if (L->lchild != NULL)

PostOrderTraverse(L->lchild);

if (L->rchild != NULL)

PostOrderTraverse(L->rchild);

printf("%d-%c ", L->key,L->data);

return OK;

}

/\*\*

\* Function name: LevelOrderTraverse

\* Function parameter: Binary tree head pointer L

\* Function function: level order traversal of binary tree

\* Return value: OK

\*\*/

Status Level Order Traverse (BitTree T)

{

if (T == NULL) return 0;

BitTree q[1000];

q[0] = T;

int front = 0, rear = 1;

while (front < rear)

{

printf("%d-%c ", q[front]->key,q[front]->data);

if (q[front]->lchild) q[rear++] = q[front]->lchild;

if (q[front]->rchild) q[rear++] = q[front]->rchild;

++front;

}

return OK;

}

/\*\*

\* Function name: SaveBiTree

\* Function parameters: file pointer fp, binary tree head pointer L

\* Function function: save the data in L to the file pointed to by the fp pointer

\* Return value: OK

\*\*/

Status SaveBiTree(FILE \*fp, BitTree L)

{

ElemType c= -1;

if (L == NULL)

fwrite(&(c),sizeof(ElemType),1,fp);

else

{

fwrite(&(L->key), sizeof(ElemType), 1, fp);

fwrite(&(L->data), sizeof(char), 1, fp);

SaveBiTree(fp, L->lchild);

SaveBiTree(fp, L->rchild);

}

return OK;

}

/\*\*

\* Function name: LoadBiTree

\* Function parameters: file pointer fp, binary tree head pointer L

\* Function function: data loading, input elements for the binary tree.

\* Return value: OK

\*\*/

Status ReadBiTree(FILE \*fp, BitTree \*L)

{

ElemType key;

fread(&(key),sizeof(ElemType),1,fp);

if (key == -1) \*L = NULL;

else

{

\*L = (BitTree)malloc(sizeof(BiTNode));

(\*L)->key = key;

fread(&((\*L)->data), sizeof(char), 1, fp);

ReadBiTree(fp, &(\*L)->lchild);

ReadBiTree(fp, &(\*L)->rchild);

}

return OK;

}

/\*\*

\* Function name: FindNode

\* Function parameters: binary tree head pointer L, keyword key, address of binary tree node pointer node,

\* Function function: find the node whose keyword is key and return its pointer

\* Return value: successfully return the address of node, otherwise return NULL

\*\*/

BitTree FindNode(BitTree L, ElemType key, BitTree \*node)

{

if (L == NULL) return NULL;

if (L->key == key){ \*node = L; return \*node; }

if (L->lchild != NULL&& FindNode(L->lchild, key, node))

return \*node;

if (L->rchild != NULL&& FindNode(L->rchild, key, node))

return \*node;

return NULL;

}

/\*\*

\* Function name: Search

\* Function parameters: binary tree head pointer L, keyword key

\* Function function: find the node whose keyword is key and return its pointer

\* Return value: successfully return the address of M, otherwise return NULL

\*\*/

BitTree Search(BitTree L, ElemType key){

BitTree M=NULL;

if(L==NULL) return M;

if(L->key==key) M=L;

if(!M) M=Search(L->lchild,key);

if(!M) M=Search(L->rchild,key);

return M;

}

/\*\*

\* Function name: Filename

\* Function parameters: none

\* Function function: automatically generate and modify the file name according to the system time

\* Return value: return time string pointer

\*\*/

char\* Filename(void){

time\_t lt;

lt=time(NULL);

filename=ctime(&lt);

filename[strlen(filename)-1]='\0';

strcat(filename,appe); //The file name string named at the above generation time

pc=&filename[0];

while(\*pc){ //The file name string is normalized, replace ':' and ' ' with '\_'

if((\*pc==':' ) || (\*pc==' ')){ \*pc='\_'; }

pc++;

}

return filename;

# Appendix D Source Program Based on Adjacency List Graph

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

// function result status code

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR -1

typedef int Status; // is the type of the function, its value is the function result status code

typedef int QElemType; //subscript corresponding to the vertex node

typedef struct QNode{

QElemType data; // node information

struct QNode \*next;

}QNode,\*QueuePtr; //queue node structure

typedef struct{

QueuePtr front, rear; // Queue head, queue tail pointer

}LinkQueue; //queue structure

/\*\*

\* Function name: InitQueue

\* Function parameter: queue structure pointer Q

\* Function function: initialize an empty queue

\* Return value: ERROR on failure, OK on success

\*\*/

Status InitQueue(LinkQueue \*Q)

{

// Construct an empty queue Q

(\*Q).front = (\*Q).rear = (QueuePtr)malloc(sizeof(QNode));

if(!(\*Q).front) {

printf(" Failed to allocate space"); return ERROR;

}

(\*Q).front->next = NULL;

return OK;

}

/\*\*

\* Function name: QueueEmpty

\* Function parameters: queue structure Q

\* Function function: determine whether the queue is empty

\* Return value: return TRUE if yes, return FALSE if successful

\*\*/

Status QueueEmpty(LinkQueue Q)

{

//If Q is an empty queue, return TRUE, otherwise return FALSE

if(Q. front == Q. rear) return TRUE;

else return FALSE;

}

/\*\*

\* Function name: EnQueue

\* Function parameters: queue structure pointer Q, vertex subscript e

\* Function function: put the subscript into the queue

\* Return value: ERROR on failure, OK on success

\*\*/

Status EnQueue(LinkQueue \*Q, QElemType e)

{

//Insert element e as the new tail element of Q

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

if(!p) // storage allocation failed

return ERROR;

p->data = e;

p->next = NULL;

(\*Q).rear->next = p;

(\*Q).rear = p;

return OK;

}

/\*\*

\* Function name: DeQueue

\* Function parameters: queue structure pointer Q, vertex subscript pointer e

\* Function function: take the subscript out of the queue

\* Return value: ERROR on failure, OK on success

\*\*/

Status DeQueue(LinkQueue \*Q, QElemType \*e)

{

//If the queue is not empty, delete the head element of Q, return its value with e, and return OK, otherwise return ERROR

QueuePtr p;

if((\*Q).front == (\*Q).rear){

printf("The queue is empty"); return ERROR;

}

p = (\*Q).front->next;

\*e = p->data;

(\*Q).front->next = p->next;

if((\*Q). rear == p)

(\*Q).rear = (\*Q).front;

free(p);

return OK;

}

#define KIND 3 //The default type of graph: 0-directed graph, 1-directed network, 2-undirected graph, 3-undirected network

#define COLUMN 5 //The number of graphs

#define MAX\_VERTEX\_NUM 20 //Maximum number of vertices

#define MAX\_VERNAME\_LEN 10 //maximum length of vertex name

//Data element type definition

typedef int Boolean;

typedef int InfoType; // arc length in the net

typedef char VertexType; //node name

typedef struct ArcNode{

int adjvex; // The position of the vertex pointed by the arc

struct ArcNode \* nextarc; // pointer to the next arc

InfoType \*info; // The pointer of the arc related information

}ArcNode; //arc node structure

typedef struct VNode{

ArcNode \* firstarc; // Pointer to the first arc attached to this vertex

char data; // vertex information

}VNode, AdjList[MAX\_VERTEX\_NUM]; //vertex node structure

typedef struct{

AdjList vertices;

int vexnum, arcnum; // The current number of vertices and arcs of the graph

int kind; // The kind flag of the graph

}ALGraph; //graph structure

Boolean judge[COLUMN], flag=FALSE;

Boolean visited[COLUMN][MAX\_VERTEX\_NUM]; //visit flag array

int col=1; //The serial number of the table to be operated, initially 1

FILE \*fp;

char \*filename, \*pc, appe[5]=".dat", input[60]; //used to generate filename

//Status (\*VisitFunc)(VertexType a); //Function variable

Status CreateGraph(ALGraph \*G); //Construct graph G according to the definition of V and VR

Status DestroyGraph(ALGraph \*G); //destroy graph G

int LocateVex(ALGraph \*G, VertexType u); //If u exists in graph G, return the position number of vertex u, otherwise return -1

char GetVex(ALGraph \*G, int rec);

Status PutVex(ALGraph \*G, int rec, VertexType v);

int FirstAdjVex(ALGraph \*G, VertexType v); //Return the first adjacent vertex of v, if v has no adjacent vertex, return empty

int NextAdjVex(ALGraph \*G, VertexType v, VertexType w); // Return v's (relative to w) next adjacent vertex, if w is the last adjacent vertex, return empty.

Status InsertVex(ALGraph \*G, VertexType v); //Add a new vertex v in graph G

Status DeleteVex(ALGraph \*G, VertexType v); //Delete vertex v and arcs related to v in graph G

Status InsertArc(ALGraph \*G, VertexType v, VertexType w); //Add arc <v,w> in graph G, if graph G is an undirected graph, also need to add <w,v>

Status DeleteArc(ALGraph \*G, VertexType v, VertexType w); //Delete arc <v,w> in graph G, if graph G is an undirected graph, also need to delete <w,v>

Status DFSTraverse(ALGraph \*G);//Depth-first search traversal on graph G

Status BFSTraverse(ALGraph \*G); //Breadth-first search traversal on graph G

int OutDegree(AdjList i); //Calculate the outdegree of the ith fixed point in graph G

Status SaveGraph(ALGraph List[]); //Save

Status LoadGraph(ALGraph List[]); //Read

Status ChooseList(int col, ALGraph \*\*G, ALGraph List[]);

int main(){

int op, rec;

VertexType e,v,w;

ALGraph \*G=NULL, List[COLUMN];

G=List;

while(op){

system("cls"); printf("\n\n");

printf("This system operates the undirected network system, there are %d pictures in total, and the current operation is the %d picture\n", COLUMN, col);

printf("Menu for Linear Table On Node Structure\n");

printf("----------------------------------------------- --\n");

printf(" 1. CreateGraph 9. DeleteVex\n");

printf(" 2. DestroyGraph 10. InsertArc\n");

printf(" 3. LocateVex 11. DeleteArc\n");

printf(" 4. GetVex 12. DFSTraverse\n");

printf(" 5. PutVex 13. BFSTraverse\n");

printf(" 6. FirstAdjVex 14. SaveGraph\n");

printf(" 7. NextAdjVex 15. LoadGraph\n");

printf(" 8. InsertVex 16. ChooseGraph\n");

printf(" 0. Exit\n");

printf("----------------------------------------------- --\n");

printf("Please choose your operation [0~16]:");

scanf("%d",&op);

switch(op){

case 1:

if(CreateGraph(G)==OK) printf(" The graph was built successfully\n");

else printf(" Graph construction failed\n");

getchar(); getchar(); break;

case 2:

if(DestroyGraph(G)==OK) printf(" Graph destroyed successfully\n");

else printf(" Graph destruction failed\n");

getchar(); getchar(); break;

case 3:

printf(" Please enter the name of the vertex e you want to find (char type):");

fflush(stdin);

scanf("%c",&e); getchar();

rec = LocateVex(G,e);

if(rec==-1) printf(" This vertex does not exist\n");

else printf(" The serial number of the vertex in the graph or network is %d\n", rec);

getchar(); getchar(); break;

case 4:

printf("Please enter the subscript of the vertex rec you want to find (int type):");

fflush(stdin);

scanf("%d",&rec); getchar();

e = GetVex(G,rec);

if(e==ERROR) printf(" Find failed\n");

else printf("%d' s name is %c\n", rec, e);

getchar(); getchar(); break;

case 5:

printf("Please enter the serial number (int type) of the vertex rec to be modified:");

fflush(stdin);

scanf("%d",&rec); getchar();

printf(" Enter a new name v(char type):");

scanf("%c", &v); getchar();

if(PutVex(G,rec,v)==OK) printf(" Modified successfully\n");

else printf(" Modification failed\n");

getchar(); getchar(); break;

case 6:

flag = FALSE;

printf(" Please enter the name of the vertex e you want to find (char type):");

fflush(stdin);

scanf("%c",&e); getchar();

rec = FirstAdjVex(G,e);

if(rec == ERROR){

if(flag==TRUE){

printf("v has no adjacent vertices"); flag = FALSE;

}

printf("Return failed\n");

}

else

printf("The serial number of the first adjacent vertex of v is %d\n",rec);

getchar(); getchar(); break;

case 7:

flag = FALSE;

printf(" Please enter the name of the vertex v you want to find and its relative vertex w (char type):");

fflush(stdin);

scanf("%c", &v); getchar();

scanf("%c", &w); getchar();

rec = NextAdjVex(G,v,w);

if(rec==ERROR){

if(flag==TRUE){

printf("w is the last adjacent vertex of v"); flag = FALSE;

}

printf("Return failed\n");

}

else printf(" The serial number of v's next adjacent vertex relative to w is %d\n",rec);

getchar(); getchar(); break;

case 8:

printf(" Please enter the name of the vertex v to be added (char type):");

fflush(stdin);

scanf("%c", &v); getchar();

if(InsertVex(G,v)==OK) printf(" Added successfully\n");

else printf(" Add failed\n");

getchar(); getchar(); break;

case 9:

printf(" Please enter the name of the vertex v to be deleted (char type):");

fflush(stdin);

scanf("%c", &v); getchar();

if(DeleteVex(G,v)==ERROR) printf(" This vertex does not exist\n");

else printf(" delete successfully\n");

getchar(); getchar(); break;

case 10:

printf(" Please input the name of vertex v and vertex w to add arc (char type):");

fflush(stdin);

scanf("%c", &v); getchar();

scanf("%c", &w); getchar();

if(InsertArc(G,v,w)==ERROR) printf(" Vertex v or w does not exist");

else printf(" Added successfully\n");

getchar(); getchar(); break;

case 11:

printf(" Please input the names of vertex v and vertex w to delete the arc (char type):");

fflush(stdin);

scanf("%c", &v); getchar();

scanf("%c", &w); getchar();

if(DeleteArc(G,v,w)==ERROR) printf(" Vertex v or w does not exist");

else printf(" Arc deleted successfully");

getchar(); getchar(); break;

case 12:

DFSTraverse(G);

getchar(); getchar(); break;

case 13:

BF Traverse(G);

getchar(); getchar(); break;

case 14:

if(SaveGraph(List)==OK) printf("Save successfully\n");

else printf("Failed to save\n");

getchar(); getchar(); break;

case 15:

if(LoadGraph(List)==OK) printf("Loading successfully\n");

else printf("Failed to load\n");

getchar(); getchar(); break;

case 16:

printf("Please enter the bit sequence vol:(1<=col<=%d)", COLUMN);

scanf("%d", &col);

ChooseList(col, &G, List);

getchar(); getchar();

case 0:

break;

}//end of switch

}//end of while

return 0;

}

/\*\*

\* Function name: CreateGraph

\* Function parameters: graph structure pointer G

\* Function function: construct a graph G

\* Return value: return OK if successfully constructed, otherwise return ERROR

\*\*/

Status CreateGraph(ALGraph \*G){

int i,j,k;

int w; // weight W

VertexType va, vb;// Arc tail Va, arc head Vb

ArcNode \*p;

/\*

printf("Please enter 3 basic information of the graph\nType (0-directed graph, 1-directed network, 2-undirected graph, 3-undirected network):\n");

// Output image type

scanf("%d",&(\*G).kind); getchar();

while ((\*G). kind < 0 || (\*G). kind > 3)

{

printf(" The input value is invalid, please retype:\n");

scanf("%d",&(\*G).kind); getchar();

}

\*/

if(judge[col-1]==TRUE){

printf("Please destroy the existing map at the current location and then create a new one"); return ERROR;

}

G->kind= KIND; // The default is undirected network

// Enter the number of vertices

printf(" Number of vertices (within %d):", MAX\_VERTEX\_NUM);

scanf("%d", &(G->vexnum)); getchar();

while (G->vexnum < 0 || G->vexnum > MAX\_VERTEX\_NUM)

{

printf(" The input value is invalid, please re-enter:");

scanf("%d",&(G->vexnum)); getchar();

}

if(G->vexnum) judge[col-1] = TRUE;

// Number of sides

printf(" Number of arcs (edges):");

scanf("%d", &(G->arcnum)); getchar();

printf(" Please enter the name or value of %d vertices (1 character, separated by spaces):", (\*G).vexnum);

for (i = 0; i < G->vexnum; i++) // Input each vertex information

{

scanf("%c",&(G->vertices[i].data)); getchar();

G->vertices[i].firstarc = NULL;

}

if (G->kind % 2) //1 and 3 are nets

printf("Please enter the arc (edge) weight W, arc tail Va and arc head Vb in order (separated by spaces):\n");

else //0 and 2 are pictures

printf(" Please enter arc tail Va and arc head Vb of the arc (edge) in order (separated by spaces):\n");

for (k = 0; k < G->arcnum; k++) // construct adjacency list

{

printf("%d:", k+1);

if (G->kind % 2) // net

scanf("%d %c %c", &w, &va, &vb);

else // graph

scanf("%c %c", &va, &vb);

i = LocateVex(G, va); // arc tail

j = LocateVex(G, vb); // arc head

if (i == -1 || j == -1)

{// vertex does not exist

printf(" The vertex does not exist, please re-enter\n");

k--; continue;

}

p = (ArcNode \*)malloc(sizeof(ArcNode));

p->adjvex = j;

if ((\*G).kind % 2)// net

{

p->info = (int \*)malloc(sizeof(int));

\*(p->info) = w;

}

else // graph

p->info = NULL; //

p->nextarc = (\*G).vertices[i].firstarc; // Insert in the header, last in first out

G->vertices[i].firstarc = p;

// Undirected graph or undirected network needs reverse

if(G->kind >= 2)

{

p = (ArcNode\*)malloc(sizeof(ArcNode));

p->adjvex = i;

if ((\*G).kind == 3) // undirected network

{

p->info = (int\*)malloc(sizeof(int));

\*(p->info) = w;

}

else // undirected graph

p->info = NULL;

p->nextarc = (\*G).vertices[j].firstarc; // Insert in the header

G->vertices[j].firstarc = p;

}

}

return OK;

}

/\*\*

\* Function name: DestroyGraph

\* Function parameters: graph structure pointer G

\* Function function: destroy graph G

\* Return value: return OK if destroyed successfully, otherwise return ERROR

\*\*/

Status DestroyGraph(ALGraph \*G){

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

int i;

ArcNode \*p, \*q;

for(i = 0; i < G->vexnum; i++)

{

p=G->vertices[i].firstarc;

while(p)

{

q = p->nextarc;

if((\*G).kind % 2) // net

free(p->info);

free(p);

p = q;

}

}

G->vexnum = G->arcnum = 0;

judge[col-1]=FALSE;

return OK;

}

/\*\*

\* Function name: LocateVex

\* Function parameters: graph structure pointer G, vertex name u to be checked

\* Function function: find the vertex named u

\* Return value: find the subscript of the returned node, otherwise return ERROR (ie -1)

\*\*/

int LocateVex(ALGraph \*G, VertexType u){

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

int i;

for(i=0; i < G->vexnum; i++)

if(u == G->vertices[i].data)

return i;

return -1;// If not found, return -1

}

/\*\*

\* Function name: GetVex

\* Function parameters: graph structure pointer G, vertex subscript rec to be checked

\* Function function: get the name of the vertex whose subscript is rec

\* Return value: find the return node name, otherwise return ERROR

\*\*/

VertexType GetVex(ALGraph \*G, int rec){

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

if(rec>= G->vexnum){

printf("Subscript out of bounds"); return ERROR;

}

return G->vertices[rec].data;

}

/\*\*

\* Function name: PutVex

\* Function parameters: graph structure pointer G, subscript rec of the vertex to be modified, new name of the corresponding node v

\* Function function: Assign the name u to the vertex whose subscript is rec

\* Return value: return OK if found, otherwise return ERROR

\*\*/

Status PutVex(ALGraph \*G, int rec, VertexType v){

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

if(rec>= G->vexnum){

printf("Subscript out of bounds"); return ERROR;

}

G->vertices[rec].data = v;

return OK;

}

/\*\*

\* Function name: FirstAdjVex

\* Function parameters: graph structure pointer G, vertex name v to be checked

\* Function: Find the subscript of the first adjacent node of the node named v

\* Return value: find the subscript value, otherwise return ERROR

\*\*/

int FirstAdjVex(ALGraph \*G, VertexType v){

ArcNode \*p;

int v1;

v1 = LocateVex(G,v);// Get the serial number v1 of v in graph G

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

if (v1 == -1){

printf(" This vertex does not exist"); return ERROR;

}

p = G->vertices[v1].firstarc;

if (p) return p->adjvex;

else

{ flag = TRUE; return ERROR;}

}

/\*\*

\* Function name: NextAdjVex

\* Function parameters: graph structure pointer G, the name of the vertex to be checked v, the name of the adjacent vertex w of v

\* Function function: Find the next adjacent vertex named v (relative to the vertex named w)

\* Return value: the serial number of the next adjacent vertex of v (relative to w),

\* If w is the last adjacent point of v or the search fails, return ERROR

\*\*/

int NextAdjVex(ALGraph \*G,VertexType v,VertexType w){

ArcNode \*p;

int v1, w1;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

v1 = LocateVex(G,v); // Get the serial number v1 of v in graph G

w1 = LocateVex(G,w); // Get the serial number w1 of w in graph G

if (v1 == -1){// vertex does not exist

printf(" Vertex v does not exist"); return ERROR;

}

if (w1 == -1){// adjacent point does not exist

printf(" Vertex w does not exist"); return ERROR;

}

p = G->vertices[v1].firstarc;

while (p && p->adjvex != w1)// The pointer p is not empty and the pointed table node is not w

p = p->nextarc;

// Did not find w or w is the last adjacent point

if(!p){

printf("w is not an adjacent vertex of v"); return ERROR;

}

else if(!p->nextarc){

flag=TRUE; return ERROR; //flag is used to distinguish whether to call from the main function or from other functions to decide whether to output relevant prompt information

}

else return p->nextarc->adjvex;

// Return the serial number of the next adjacent vertex of v (relative to w)

}

/\*\*

\* Function name: InsertVex

\* Function parameters: graph structure pointer G, new vertex name v

\* Function function: add a new vertex named v in graph G

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status InsertVex(ALGraph \*G, VertexType v){

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if((\*G).vexnum == 0){

printf("The picture is empty"); return ERROR;

}

if((\*G).vexnum == MAX\_VERTEX\_NUM){// The number of vertices reaches the maximum

printf("The number of vertices reaches the maximum value"); return ERROR;

}

// construct a new vertex

(\*G).vertices[(\*G).vexnum].data = v;

(\*G).vertices[(\*G).vexnum].firstarc=NULL;

(\*G).vexnum++; // Add 1 to the number of vertices in graph G

return OK;

}

/\*\*

\* Function name: DeleteVex

\* Function parameters: graph structure pointer G, vertex name v to be deleted

\* Function function: delete the vertex named v and its related arc in G

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status DeleteVex(ALGraph \*G, VertexType v){

int i, j;

ArcNode \*p, \*q;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if((\*G).vexnum == 0){

printf("The picture is empty"); return ERROR;

}

j = LocateVex(G, v); // Get the serial number of vertex v

if(j == -1) return ERROR;//v is not a vertex of graph G{

// Delete the arc or edge with v as the out degree

p = (\*G).vertices[j].firstarc;

while(p)

{

q = p;

p = p->nextarc;

if((\*G).kind % 2) // net

free(q->info);

free(q);

(\*G).arcnum--; // The number of arcs or edges minus 1

}

// rearrange vertices

(\*G).vexnum--; // The number of vertices minus 1

for(i = j; i < (\*G).vexnum; i++) // The vertex behind vertex v moves forward

(\*G).vertices[i] = (\*G).vertices[i+1];

// Delete the arc or edge with v as the in-degree and modify the position value of the vertex pointed by the arc

for(i = 0; i < (\*G).vexnum; i++)

{

p = (\*G).vertices[i].firstarc; // point to the first arc or edge

while(p) // has arc or edge

{

if(p->adjvex == j)// The arc points to the deleted node

{

if(p == (\*G).vertices[i].firstarc) // The node to be deleted is the first node

{

(\*G).vertices[i].firstarc = p->nextarc;

if((\*G).kind%2) // net

free(p->info);

free(p);

p = (\*G).vertices[i].firstarc;

if((\*G).kind < 2)// The directed graph needs to be subtracted again

(\*G).arcnum--; // The number of arcs or edges minus 1

}

else

{

q->nextarc = p->nextarc;

if((\*G).kind % 2) // net

free(p->info);

free(p);

p = q->nextarc;

if((\*G).kind < 2)// direction needs to be reduced again

(\*G).arcnum--; // The number of arcs or edges minus 1

}

}

else// The arc does not point to the deleted node

{

if(p->adjvex > j)// The position number of the element pointed by the arc is after the vertex is deleted

p->adjvex--; // Change the position number of the vertex pointed by the arc

q = p;

p = p->nextarc;// analyze the next arc

}

}

}

if(G->vexnum==0) printf(" There is no vertex in the graph");

return OK;

}

/\*\*

\* Function name: InsertArc

\* Function parameters: graph structure pointer G, vertex name v, w

\* Function function: add arc <v,w> in graph G, if graph G is an undirected graph/network, also need to add <w,v>

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status InsertArc(ALGraph \*G,VertexType v,VertexType w){

ArcNode \*p;

int w1, i, j;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if((\*G).vexnum == 0){

printf("The picture is empty"); return ERROR;

}

i = LocateVex(G, v); // Get the serial number of v position

j = LocateVex(G, w); // Get the position number of w

if(i == -1 || j == -1) return ERROR;// Corresponding vertex not found

// create arc vw

(\*G).arcnum++;

if((\*G).kind % 2) // The network needs to lose the weight

{

printf(" Please input the weight of arc (edge)%c->%c:\n", v, w);

scanf("%d", &w1); getchar();

}

p=(ArcNode\*)malloc(sizeof(ArcNode));

p->adjvex = j;

if((\*G).kind % 2) // net

{

p->info = (int \*)malloc(sizeof(int));

\*(p->info) = w1;

}

else p->info=NULL;

p->nextarc = (\*G).vertices[i].firstarc;

(\*G).vertices[i].firstarc = p;

// The undirected graph needs to create an arc w->v

if((\*G). kind >= 2)

{

p = (ArcNode\*)malloc(sizeof(ArcNode));

p->adjvex = i;

if((\*G).kind == 3) // undirected network

{

p->info = (int\*)malloc(sizeof(int));

\*(p->info) = w1;

}

else p->info = NULL;

p->nextarc = (\*G).vertices[j].firstarc;

(\*G).vertices[j].firstarc = p;

}

return OK;

}

/\*\*

\* Function name: DeleteArc

\* Function parameters: graph structure pointer G, names of two vertices v and w to be deleted

\* Function function: delete the arc <v,w> in the graph G, if the graph G is an undirected graph/network, you need to add <w,v>

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status DeleteArc(ALGraph \*G,VertexType v,VertexType w){

ArcNode \*p, \*q;

int i, j;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if((\*G).vexnum == 0){

printf("The picture is empty"); return ERROR;

}

if((\*G).arcnum == 0){

printf("There is no arc in the graph"); return ERROR;

}

i = LocateVex(G, v); // Vertex v number

j = LocateVex(G, w); // The serial number of vertex w

if(i < 0 || j < 0) return ERROR;

// delete arc<v,w>

p = (\*G).vertices[i].firstarc;

while(p && p->adjvex != j) //p is not empty and the arc pointed to is not an arc to be deleted<v,w>

{ //p points to the next arc

q = p; p = p->nextarc;

}

if(p && p->adjvex == j) // find arc<v,w>

{

if(p == (\*G).vertices[i].firstarc) //p refers to the first arc

(\*G).vertices[i].firstarc = p->nextarc; // point to the next arc

elseq->nextarc = p->nextarc; // point to the next arc

if((\*G).kind % 2) free(p->info); // The network needs to release more weights

free(p); // release this node

(\*G).arcnum--; // The number of arcs or edges minus 1

}

// The undirected graph needs to delete the symmetric arc<w,v>

if((\*G). kind >= 2)

{

p = (\*G).vertices[j].firstarc;

while(p && p->adjvex != i) //p is not empty and the arc pointed to is not an arc to be deleted<w,v>

{ //p points to the next arc\*/

q = p; p = p->nextarc;

}

if(p && p->adjvex == i) // find arc<w,v>

{

if(p == (\*G).vertices[j].firstarc) //p refers to the first arc

(\*G).vertices[j].firstarc=p->nextarc; // point to the next arc

else q->nextarc=p->nextarc; // point to the next arc

if((\*G).kind % 2) free(p->info); // The network needs to release more weights

free(p); // release this node

}

}

if((\*G).arcnum == 0) printf(" There is no arc in the graph");

return OK;

}

/\*\*

\* Function name: DFS

\* Function parameters: graph structure pointer G, vertex subscript v

\* Function: starting from the vth vertex, recursively traverse the graph G in depth first

\* Return value: None

\*\*/

void DFS(ALGraph \*G,int v){

int w;

VertexType v1, w1;

v1 = G->vertices[v].data;

visited[col-1][v] = TRUE; // Set the visited flag to TRUE (visited)

printf(" Number: %d-Name: %c ", v, G->vertices[v].data); /\* Access the vth vertex\*/

for(w = FirstAdjVex(G,v1); w >= 0; w = NextAdjVex(G, v1, w1=G->vertices[w].data))

if(!visited[col-1][w])

DFS(G,w); // Call DFS recursively on v's unvisited adjacent point w

}

/\* Function name: DFSTraverse

\*Input parameters: Figure G, call the function visit

\*Return value: Status successfully returns OK;

\*Function function: Perform a depth-first search traversal on the graph G, and use the function on each vertex in the graph in turn

\* Number of visits to visit once, and only visit once

\*/

/\*\*

\* Function name: DFSTraverse

\* Function parameters: graph structure pointer G

\* Function function: perform a depth-first search traversal on the graph G, and use the function on each vertex in the graph in turn

\* Count visit once, and only visit once

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status DFSTraverse(ALGraph \*G){

int v;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

// VisitFunc = Visit; // Use the global variable VisitFunc, so that DFS does not need to set function pointer parameters

for(v = 0; v < G->vexnum; v++)

visited[col-1][v]=FALSE; // visit flag array initialization

for(v = 0; v < G->vexnum; v++)

if(!visited[col-1][v])

DFS(G,v); //Call DFS on vertices that have not been visited yet

return OK;

}

/\*\*

\* Function name: BFSTraverse

\* Function parameters: graph structure pointer G

\* Function function: Traverse graph G non-recursively according to breadth first

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status BF Traverse(ALGraph \*G){

int v, u, w;

VertexType u1, w1;

LinkQueue Q;

if(judge[col-1]!=TRUE){

printf("Graph not initialized"); return ERROR;

}

if(G->vexnum == 0){

printf("The picture is empty"); return ERROR;

}

for(v = 0; v < G->vexnum; ++v) visited[col-1][v]=FALSE;

InitQueue(&Q); //Empty auxiliary queue Q

for(v = 0; v < G->vexnum; ++v)

if(!visited[col-1][v]) //v has not been visited yet

{

visited[col-1][v] = TRUE;

printf(" Number: %d-Name: %c ", v, G->vertices[v].data);

EnQueue(&Q, v); //v into the queue

while(!QueueEmpty(Q))

{

DeQueue(&Q, &u); // The head element of the queue is dequeued and set to u

u1 = G->vertices[u].data;

for(w = FirstAdjVex(G,u1); w >= 0; w = NextAdjVex(G,u1,w1=G->vertices[w].data))

if(!visited[col-1][w]) //w is the unvisited adjacent vertex of u

{

visited[col-1][w] = TRUE;

printf(" Number: %d-Name: %c ", w, G->vertices[w].data);

EnQueue(&Q,w); //w enters the queue

}

}

}

return OK;

}

/\*\*

\* Function name: OutDegree

\* Function parameters: the i-th vertex address of graph G

\* Function: Calculate the out-degree of the i-th vertex

\* Return value: out-degree of the i-th vertex

\*\*/

int OutDegree(AdjList i){

int num = 0;

ArcNode \*p;

p = i->firstarc;

while (p != NULL)

{

num++;

p = p->nextarc;

}

return num;

}

/\*\*

\* Function name: LoadGraph

\* Function parameters: graph structure array List

\* Function function: read the information of multiple graphs from the file

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status LoadGraph(ALGraph List[COLUMN]){

/\*

if(judge!=TRUE){

printf("Graph not initialized"); return ERROR;

}

\*/

int i, j, k, num, yl; //k is the vertex subscript, num is the outgoing degree of each vertex

FILE \*fp;

ArcNode \*p;

printf("Please enter the file name of the data you want to restore:");

scanf("%s", input);

filename=input;

if ((fp = fopen(filename,"r")) == NULL){

printf(" Failed to open the file"); return ERROR;

}

fread(&yl,sizeof(int),1,fp); // read the number of valid tables

// Read the basic information of the graph

for(i=0;i<yl;i++){

fread(&(List[i].kind), sizeof(int), 1, fp); //category

fread(&(List[i].vexnum), sizeof(int), 1, fp); //number of points

fread(&(List[i].arcnum), sizeof(int), 1, fp); //Number of sides

// read the adjacency list

for(j=0; j<List[i].vexnum; j++){ // There are remaining vertices

fread(&(List[i].vertices[j].data), sizeof(char), 1, fp); //name

printf("Figure-%d, serial number-%d, name-%c", i, j, List[i].vertices[j].data);

fread(&num, sizeof(int), 1, fp); // current vertex out degree

if (num == 0) List[i].vertices[j].firstarc = NUL L; // out degree is 0

else

{

printf("adjacent point serial number:");

List[i].vertices[j].firstarc = NULL;

for (k = 0; k < num; k++)

{

p = (ArcNode \*)malloc(sizeof(ArcNode));

fread(&(p->adjvex), sizeof(int), 1, fp); //adjacent point number

printf("%d ",p->adjvex);

if (List[i].kind % 2)//network

{

p->info = (int \*)malloc(sizeof(int));

fread(p->info, sizeof(InfoType), 1, fp); //

}

else p->info = NULL; //graph

p->nextarc = List[i].vertices[j].firstarc; //Insert in the header, last in first out

List[i].vertices[j].firstarc = p;

}

}

printf("\n\n");

}

judge[i] = TRUE;

}

fclose(fp);

return OK;

}

/\*\*

\* Function name: SaveGraph

\* Function parameters: graph structure array List

\* Function function: save the information of multiple valid graphs into a file

\* Return value: return OK if successful, otherwise return ERROR

\*\*/

Status SaveGraph(ALGraph List[COLUMN]){

int i, h, num, yl=0;

FILE \*fp;

ArcNode \*p;

for(i=0;i<COLUMN;i++){ // count the number of tables with elements

if(judge[i]==TRUE && List[i].vexnum!=0)

yl++;

}

if(yl ==0){ //return an error if there is no valid data

printf("No data to save"); return ERROR;

}

/\*

printf(" Please enter the save file name:");

scanf("%s",filename); getchar();

\*/

time\_t lt;

lt=time(NULL);

filename=ctime(&lt);

filename[strlen(filename)-1]='\0';

strcat(filename,appe); //The file name string named at the above generation time

pc=&filename[0];

while(\*pc){ //The file name string is normalized, replace ':' and ' ' with '\_'

if((\*pc==':' ) || (\*pc==' ')){ \*pc='\_'; }

pc++;

}

//method to write file

if ((fp = fopen(filename, "w")) == NULL)

{

printf(" Failed to open the file"); return ERROR;

}

fwrite(&yl,sizeof(int),1,fp); // The number of valid tables written

// Save the basic information of the graph

for(h=0; h<COLUMN; h++){

if(judge[h]==TRUE && List[h].vexnum!=0){

fwrite(&(List[h].kind), sizeof(int), 1, fp); //kind

fwrite(&(List[h].vexnum), sizeof(int), 1, fp); //number of points

fwrite(&(List[h].arcnum), sizeof(int), 1, fp); //Number of sides

// save the adjacency list

for (i = 0; i < List[h].vexnum; i++)

{

fwrite(&(List[h].vertices[i].data), sizeof(char), 1, fp); //name

printf("Table-%d, serial number-%d, name-%c ",h,i,List[h].vertices[i].data);

if(List[h].vertices[i].firstarc==NULL) printf("\n\n");

num = OutDegree(&(List[h].vertices[i])); //current vertex outdegree

fwrite(&num, sizeof(int), 1, fp);

if ((p = List[h].vertices[i].firstarc) != NULL)

{

fwrite(&(p->adjvex), sizeof(int), 1, fp); //adjacent vertex number

printf("adjacent point number:%d ",p->adjvex);

if (List[h].kind % 2)// network

fwrite(p->info, sizeof(int), 1, fp); //arc length

while ((p = p->nextarc) != NULL)

{

fwrite(&(p->adjvex), sizeof(int), 1, fp); //adjacent vertex number

printf("%d ",p->adjvex);

if (List[h].kind % 2)//network

fwrite(p->info, sizeof(int), 1, fp);//arc length

}

printf("\n\n");

}

}

}

}

fclose(fp);

return OK;

}

/\*\*

\* Function name: ChooseList

\* Function parameters: table position col to be replaced, graph structure pointer G, graph structure array List

\* Function function: point G to the subscript col-1 of the graph to be operated

\* Return value: ERROR on failure, OK on success

\*\*/

Status ChooseList(int col, ALGraph \*\*G, ALGraph List[COLUMN]){

if(col<1 || col>COLUMN){

printf("Subscript out of bounds operation failed");

return ERROR;

}

\*G=&(List[col-1]);

if(judge[col-1]!=TRUE)

printf("This table has not been initialized"); //Prevent misoperation

return OK;

}