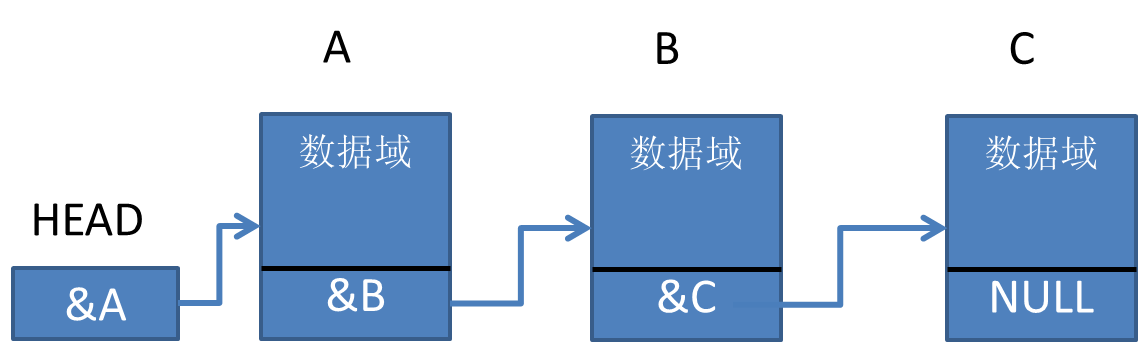
1.链表：元素不一定需要连续的内存空间，只要在需要存储数据时，再申请存储空间（动态申请内存空间或栈分配）即可，并采用指针将数据一个一个链接起来。



2.同质链表：当数据域为相同类型的对象时，为同质链表，否则称为异质链表。

//同质链表节点描述

struct Node{

int val;

struct Node \* pNext;

};

Or:

strcut Student{

//..

};

struct Node{

struct Student nData;

struct Node \* pNext;

};

Or:

struct Node{

struct Student \*pData;

struct Node \* pNext;

};

//异质链表节点描述

struct Node{

void \* pData;

struct Node \* pNext;

};

3. 链表的操作：创建链表

遍历链表输出

链表的插入（链头插入、链尾插入、按某种顺序插入）

链表的删除（链头节点删除、链尾节点删除、满足某种条件的节点删除）

链表的逆置

链表的销毁

两个链表的合并

等等

4. 按照结构化编程思想，遵循《本学期作业代码自检规范》，用C编写完整的异质链表的源程序，并检验效果。

【source code】

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ERROR 1e8

typedef enum { push, pop, inject, eject, end, rev, merge, create, output, insert, take} Operation; //定义所有可以采用的操作；

typedef enum { \_int, \_long, \_short, \_float, \_double, \_char, \_string, \_pointer} Data\_type; //定义数据类型，但输入的数据类型只支持int,double,char,string

struct Etype

{

void \*Content;

Data\_type DataCategory;

};//一个单位元素存储空间，记载内容所在地址和数据类型；

typedef struct Etype \*ElementType;

typedef struct Node \*PtrToNode;

struct Node {

ElementType Element;

PtrToNode Next, Last;

};//一个双向链表节点空间，记载链表内容，

typedef struct DequeRecord \*Deque;

struct DequeRecord {

PtrToNode Front, Rear;

};//一个链表头空间，记录这个链表的头尾；

Deque \*DeqRecRecorder; //链表头空间的存储空间；

void PrintX(ElementType tp);

Deque CreateDeque();

int Push( ElementType X, Deque D );

ElementType Pop( Deque D );

int Inject( ElementType X, Deque D );

ElementType Eject( Deque D );

int Insert (ElementType X, Deque D, int (\*Compare)(ElementType, ElementType));

ElementType Take(Deque D, int(\*Compare)(ElementType));

int Merge(Deque A, Deque B);

int Reverse(Deque D);

int DestructDeque(Deque D);

ElementType ScanX();

int ERRflag=0; //标注错误的标识符

Operation GetOp(); /\* details omitted \*/

void PrintDeque( Deque D ); /\* details omitted \*/

int d\_count=0;

void PrintInstruction() //初始的打印文字

{

char \*t="Type \042Create\042 to begin\n Available functions: \nPush (Entry)(List No)\nPop (List No)\nInject (Entry)(List No)\nEject (List No)\nInsert (Entry)(List No)\nTake (List No)\nReverse (List No)\nMerge (List a, List b)\nOutput (List No)\nEnd\n";

printf("%s", t);

}

Deque FindDeque(int k) //弹回所在的列表的地址，如果不存在此列表则返回0为错误值

{

if (k>d\_count) return 0;

return DeqRecRecorder[k];

}

int Compare(ElementType a, ElementType b) //比较函数1

{

if (a->DataCategory!=b->DataCategory) return 1;

return a->Content>b->Content;

}

int Compare2(ElementType a) //比较函数2

{

return (int)(a->Content)>10;

}

int main()

{

ElementType X;

Deque D;

int done = 0, Current\_Deque, Deque\_Temp;

DeqRecRecorder=(Deque \*)malloc(100\*sizeof(Deque));

PrintInstruction();

while (!done) {

ERRflag=0;

switch(GetOp())

{

case create:

D=CreateDeque();

printf("Linked List %d initialized\n", d\_count);

break;

//创建链表过程

case push:

X=ScanX();

scanf("%d",&Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

if (!Push(X, D)) printf("Memory of List %d is Full!\n", Current\_Deque);

}

break;

//从链表头推入元素

case pop:

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

X = Pop(D);

if (ERRflag) printf("List %d is Empty!\n", Current\_Deque);

else PrintX(X);

}

break;

//从链表头弹出元素

case inject:

X=ScanX();

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

if (!Inject(X, D)) printf("Memory of List %d is Full!\n", Current\_Deque);

}

break;

//从链表尾插入元素

case eject:

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

X = Eject(D);

if (ERRflag) printf("List %d is Empty!\n", Current\_Deque);

else PrintX(X);

}

break;

//从链表尾弹出元素

case insert:

X=ScanX();

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

if (!Insert(X, D, Compare)) printf("Memory of List %d is Full!\n", Current\_Deque);

}

break;

//当满足Compare函数中的条件时，插入元素于现有第一个满足条件的元素后，否则放在表尾

case take:

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

X = Take(D,Compare2);

if (ERRflag) printf("Cannot Find Any at List %d!\n", Current\_Deque);

else PrintX(X);

}

break;

//当满足Compare2函数中的条件时，弹出第一个满足条件的元素，否则返回错误

case rev:

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

if (!Reverse(D)) printf("Cannot reverse List %d!\n", Current\_Deque);

}

break;

//翻转链表过程；

case merge:

scanf("%d %d", &Current\_Deque, &Deque\_Temp);

Deque Dtemp=FindDeque(Deque\_Temp);

D=FindDeque(Current\_Deque);

if (!D || !Dtemp) printf("No Such List!\n");

else

{

if (!Merge(D,Dtemp)) printf("Cannot merge Lists %d and %d!\n", Current\_Deque, Deque\_Temp);

}

break;

//将前一链表和后一链表合并过程

case output:

scanf("%d", &Current\_Deque);

D=FindDeque(Current\_Deque);

if (!D) printf("No Such List!\n");

else

{

PrintDeque(D);

}

break;

//输出该链表过程

case end:

Ende();

done=1;

break;

//结束过程

}

}

return 0;

}

Operation GetOp() //监测输入何种命令

{

char \*t=(char \*)malloc(10\*sizeof(char));

scanf("%s", t);

if (!strcmp(t,"Push"))

return push;

else if (!strcmp(t,"Inject"))

return inject;

else if (!strcmp(t,"Eject"))

return eject;

else if (!strcmp(t,"Pop"))

return pop;

else if (!strcmp(t,"Output"))

return end;

else if (!strcmp(t,"Merge"))

return merge;

else if (!strcmp(t,"Reverse"))

return rev;

else if (!strcmp(t,"Create"))

return create;

}

Deque CreateDeque() //创建链表，并将地址登记在DeqRecRecorder表中，返回该链表地址（无用）

{

Deque S=(Deque)malloc(sizeof(struct DequeRecord));

PtrToNode temp=(PtrToNode)malloc(sizeof(struct Node));

S->Front=temp;

S->Rear=temp;

temp->Element=NULL;

temp->Last=temp;

temp->Next=temp;

d\_count++;

DeqRecRecorder[d\_count]=S;

return S;

}

ElementType ScanX() //扫描元素，并按照元素类型调整

{

char \*t=(char \*)malloc(50\*sizeof(char));

ElementType d=(ElementType)malloc(sizeof(struct Etype));

scanf("%s", t);

int tleng = 0;

char \*tstt = t;

while (\*t)

{

tleng++;

t++;

}

t = tstt;

t[tleng] = 0;

int isint=0, isfloat=0;

double tdouble=0,dot=0;

int tint=0;

char tchar=0;

while (t)

{

switch(\*t) //分流小数、整数、单字符、字符串

{

case '1': case '2': case '3': case '4': case '5':

case '6': case '7': case '8': case '9': case '0':

if (dot!=0)

{

tdouble+=(\*t-48)\*dot;

dot\*=0.1;

}

else

{

tint\*=10;

tint+=\*t-48;

}

break;

case 0:

if (d->DataCategory==\_char) break;

if (dot!=0)

{

d->DataCategory=\_double;

double \*content=(double \*)malloc(sizeof(double));

\*content = tdouble;

d->Content=content;

}

else

{

d->DataCategory=\_int;

int \*content = (int \*)malloc(sizeof(int));

\*content = tint;

d->Content=content;

}

return d;

break;

case '.': //对于小数点的处理，如果第一次出现小数点，则设置小数点分位，否则视作字符串

if(dot==0 && \*(t+1))

{

dot=0.1;

tdouble=tint;

break;

}

default:

if (!\*(t+1))

{

d->DataCategory=\_char;

char \*content = (char \*)malloc(sizeof(char));

d->Content=content;

break;

}

d->DataCategory=\_string;

char \*content = (char \*)malloc(strlen(tstt)\*sizeof(char));

strcpy(content, tstt);

d->Content = content;

return d;

break;

}

t++;

}

return d;

}

void PrintX(ElementType tp)

{

switch (tp->DataCategory)

{

case \_int: case \_long: case \_short:

printf("%d ", \*(int\*)(tp->Content));

break;

case \_float:

printf("%f ", \*(float\*)(tp->Content));

break;

case \_double:

printf("%lf ", \*(double\*)(tp->Content));

break;

case \_char:

printf("%c ", \*(char\*)(tp->Content));

break;

case \_string:

printf("%s ", (char\*)(tp->Content));

break;

default:

break;

}

printf("\n");

return;

}

void PrintDeque(Deque D) //遍历链表，逐个输出

{

PtrToNode temp=(D->Front)->Next;

while (temp->Next!=D->Front)

{

ElementType tp=temp->Element;

PrintX(tp);

temp=temp->Next;

}

return;

}

int Push( ElementType X, Deque D ) //从前插入元素，插入位置在空链表头后

{

struct Node \*temp=(PtrToNode)malloc(sizeof(struct Node));

if (!temp) return 0;

temp->Element=X;

temp->Next=(D->Front)->Next;

((D->Front)->Next)->Last=temp;

(D->Front)->Next=temp;

temp->Last=D->Front;

if (D->Front==D->Rear) D->Rear=temp;

return 1;

}

ElementType Pop( Deque D ) //从前弹出元素，空链表头后第一个为首个元素

{

if (D->Front==D->Rear)

{

ERRflag=1;

return 0;

}

PtrToNode temp=(D->Front)->Next;

ElementType temp\_int=temp->Element;

(D->Front)->Next=temp->Next;

if (temp->Next==D->Front) D->Rear=D->Front;

free(temp);

return temp\_int;

}

int Inject( ElementType X, Deque D ) //从尾插入元素，尾元素下一节点为头节点，保障当链表弹空时首尾为同，便于检验

{

struct Node \*temp=(PtrToNode)malloc(sizeof(struct Node));

if (!temp) return 0;

temp->Element=X;

temp->Last=D->Rear;

(D->Rear)->Next=temp;

D->Rear=temp;

temp->Next=D->Front;

return 1;

}

ElementType Eject( Deque D ) //从尾弹出元素，尾元素下一节点为头节点，保障当链表弹空时首尾为同，便于检验

{

if (D->Front==D->Rear)

{

ERRflag=1;

return 0;

}

PtrToNode temp=D->Rear;

ElementType temp\_int=temp->Element;

D->Rear=temp->Last;

free(temp);

return temp\_int;

}

int Insert (ElementType X, Deque D, int (\*Compare)(ElementType, ElementType))

{

struct Node \*temp=(PtrToNode)malloc(sizeof(struct Node));

if (!temp) return 0;

temp->Element=X;

struct Node \*temp2=(D->Front)->Next;

while (temp2->Next!=D->Front && !Compare(temp2->Element, temp->Element)) temp2=temp2->Next;

temp->Next=temp2->Next;

temp2->Next=temp;

(temp2->Next)->Last=temp;

temp->Last=temp2->Last;

if (temp->Next==D->Front || D->Front==D->Rear) D->Rear=temp;

return 1;

} //从中间插入元素，插入到第一个满足条件的元素后面，但是如果链表为空需要更改链表头尾记录

ElementType Take(Deque D, int(\*Compare)(ElementType))

{

if (D->Front==D->Rear)

{

ERRflag=1;

return 0;

}

PtrToNode temp=(D->Front)->Next;

while (temp!=D->Front && !Compare(temp->Element)) temp=temp->Next;

if (temp==D->Front)

{

ERRflag=1;

return 0;

}

ElementType temp\_int;

temp\_int=temp->Element;

if (temp->Next==D->Front) D->Rear=temp->Last;

(temp->Last)->Next=temp->Next;

(temp->Next)->Last=temp->Last;

free(temp);

return temp\_int;

} //从中间弹出元素，弹出第一个满足条件的元素，但是如果不存在满足要求的则不弹出。有需要的时候，更改链表尾。

int Reverse (Deque D)

{

PtrToNode temp=(D->Front)->Next, temp2, tlast=D->Front, tempstart=temp;

if (temp==D->Front) return 0;

while (temp!=D->Front)

{

temp2=temp->Next;

temp->Next=temp->Last;

temp->Last=temp2;

temp=temp2;

}

(D->Front)->Next=D->Rear;

(D->Rear)->Last=D->Front;

D->Rear=tempstart;

tempstart->Next=D->Front;

return 1;

} //翻转链表。

int Merge(Deque A, Deque B)

{

PtrToNode temp=(B->Front)->Next;

(A->Rear)->Next=temp;

temp->Last=A->Rear;

A->Rear=B->Rear;

(A->Rear)->Next=A->Front;

return 1;

} //将B链表归并到A链表中。B链表仍然可用。

int Ende()

{

int i;

for (i=0; i<d\_count; i++)

free(DeqRecRecorder[i]);

return -1;

}

【运行结果截图】

