**软件工程一班 马启凡 202430551135**

**实验名称：链表**

**实验内容：**

1．建立单链表，并在单链表上实现插入、删除和查找操作（验证性内容）。

2．建立双向链表，并在双向链表上实现插入、删除和查找操作（设计性内容）。

3．计算已知一个单链表中数据域值为一个指定值x的结点个数（应用性设计内容）。

**任务一：**

1．实验要求

编程实现如下功能

（1）根据输入的一系列整数，以0标志结束，用***头插法***建立单链表，并输出单链表中各元素值，观察输入的内容与输出的内容是否一致。

（2）根据输入的一系列整数，以0标志结束，用***尾插法***建立单链表，并输出单链表中各元素值，观察输入的内容与输出的内容是否一致。

（3）在单链表的第i个元素之后插入一个值为x的元素，并输出插入后的单链表中各元素值。

（4）删除单链表中第i个元素，并输出删除后的单链表中各元素值。

（5）在单链表中查找第i个元素，如果查找成功，则显示该元素的值，否则显示该元素不存在。

代码实现：

//（1）根据输入的一系列整数，以0标志结束，用头插法建立单链表，并输出单链表中各元素值，观察输入的内容与输出的内容是否一致。

#include<iostream>

using namespace std;

struct Node

{

int num;

Node \*next;

} \* head;

int main()

{

head = NULL;

int nodenum;

cin >> nodenum;

while (nodenum!=0)

{

Node \* p = new Node;

p->num = nodenum;

p->next = head;

head = p;

cin >> nodenum;

}

Node \* p = head;

while (p!=NULL)

{

cout << p->num << endl;

p = p->next;

}

system("pause");

return 0;

}

运行示例

2

3

6

5

8

7

0//输入结束

7

8

5

6

3

2

请按任意键继续. . .

分析：

成功构建了由头插法实现的链表，然而，输出的内容与输入的顺序相反，反映了头插法的特性。有时候这种特性可以被用来倒置一组数据。

//（2）根据输入的一系列整数，以0标志结束，用尾插法建立单链表，并输出单链表中各元素值，

观察输入的内容与输出的内容是否一致。

#include<iostream>

using namespace std;

struct Node

{

Node\* next;

int data;

};

int main()

{

Node\* head=new Node;

head->next = NULL;

Node\* r = head;

int Nodenum;

cin >> Nodenum;

while (Nodenum != 0)

{

Node\* p = new Node;

p->data = Nodenum;

r->next = p;

p->next = NULL;

r = p;

cin >> Nodenum;

}

head = head->next;

while (head != NULL)

{

cout << head->data << endl;;

head = head->next;

}

system("pause");

return 0;

}

代码运行：

2

3

6

5

8

4

0//输入结束

2

3

6

5

8

4

请按任意键继续. . .

分析：

由尾插法获得的链表优点在于可以按顺序输出输入的数据，缺点在于需要耗费额外的空间来存储尾指针r。

将第三个任务与第五个任务放在一起来完成，主要是因为第五个任务也是第三个任务的一部分。

//（3）在单链表的第i个元素之后插入一个值为x的元素，并输出插入后的单链表中各元素值。

//（5）在单链表中查找第i个元素，如果查找成功，则显示该元素的值，否则显示该元素不存在。

代码展示：

#include<iostream>

using namespace std;

struct Node

{

Node\* next;

int data;

};

enum status { OK, ERROR };//定义枚举类型

status list\_search(Node\* L, int i, int& e)//在带头结点的单链表L中查找第i个元素，如果查找成功则用e返回其值

{

Node\* p = L->next; /\*使指针p指向第1个元素结点\*/

int j = 1; /\*计数器赋初值为1\*/

while (p && j < i) /\*顺着后继指针查找第i个元素结点\*/

{

p = p->next;

j++;

}

if ( j < i)

return ERROR; /\*如果i值不合法，即i值小于1或大于表长，则出错\*/

e = p->data; /\*如果第i个元素存在，则将该元素值赋给e\*/

return OK;

}

status list\_insert(Node\*& L, int i, int x)//在带头结点的单链表L中第i个位置之后插入新元素x

{

Node\* p = L;

int j = 0;

while (p != NULL && j < i ) /\*寻找插入位置，并使p指向插入位置的结点，即L中的第i个位置\*/

{

p = p->next;

++j;

}

if (p == NULL || j > i ) return ERROR; /\*若位置不正确,即i小于1或大于表的长度加1，则出错\*/

Node\* s = new Node; /\*分配一个新结点的空间\*/

s->data = x; /\*为新结点数据域赋值\*/

/\*下面两条语句就是完成修改链，将新结点s插入到p结点之后\*/

s->next = p->next; /\*新结点指针域指向p的后继结点\*/

p->next = s; /\*新结点成为p的后继结点\*/

return OK;

}

int main()

{

Node\* head = new Node;

head->next = NULL;

Node\* r = head;

int Nodenum;

cin >> Nodenum;

while (Nodenum != 0)

{

Node\* p = new Node;

p->data = Nodenum;

r->next = p;

p->next = NULL;

r = p;

cin >> Nodenum;

}

int i;

cin >> i;

int e = 0;

list\_search(head, i,e);

cout << e << endl;

int j;

cin >> j;

int x;

cin >> x;

list\_insert(head, j, x);

head = head->next;

while (head != NULL)

{

cout << head->data << endl;;

head = head->next;

}

system("pause");

return 0;

}

运行实例：

（1）//任务5的运行

3

4

6

3

8

6

4

7

0//链表输入完成

3//输入想要查找的元素的位置

6//输出对应元素

请按任意键继续. . .

（2）//任务5的运行

2

3

6

5

4

8

0//链表输入完成

10//输入的位置不合法，超出了链表长度

0//程序结束并报错

请按任意键继续. . .

（3）//任务3的运行

2

3

7

5

4

8

9

0//链表输入完成

3//输入想要插入的位置

4//输入想要插入的元素

2

3

7

4

5

4

8

9

请按任意键继续. . .

分析：

综合了老师的代码提示，将任务3和5合并完成，实现了对于链表数据的查找与插入，难点在于要避免链表的“断裂”，还要清楚地了解指针的不同作用，实现对于链表的调动。

//（4）删除单链表中第i个元素，并输出删除后的单链表中各元素值。

1.代码展示

#include<iostream>

using namespace std;

struct Node

{

Node\* next;

int data;

};

void list\_search(Node\* L, int i)//在带头结点的单链表L中查找第i个元素，如果查找成功则删除该元素所在结点

{

Node\* q = L;

Node\* p = L->next; /\*使指针p指向第1个元素结点\*/

int j = 1; /\*计数器赋初值为1\*/

while (p && j < i) /\*顺着后继指针查找第i个元素结点\*/

{

q = p;

p = p->next;

j++;

}

if (!p || j > i)

{

cout << "ERROR" << endl;

return; /\*如果i值不合法，即i值小于1或大于表长，则出错\*/

}

//如果第i个元素存在，则将该元素所在结点删除

q->next= p->next;

delete p;

cout << "OK" << endl;

return;

}

int main()

{

Node\* head = new Node;

head->next = NULL;

Node\* r = head;

int Nodenum;

cin >> Nodenum;

while (Nodenum != 0)

{

Node\* p = new Node;

p->data = Nodenum;

r->next = p;

p->next = NULL;

r = p;

cin >> Nodenum;

}

int i;

cin >> i;

list\_search(head, i);

head = head->next;

while (head != NULL)

{

cout << head->data << endl;

head = head->next;

}

system("pause");

return 0;

}

运行示例：

（1）输入位置合法时

2

3

7

5

4

8

9

0//链表输入完成

3//输入想要删除的结点位置

OK//提示删除成功

2

3

5

4

8

9//输出删除后的链表

请按任意键继续. . .

（2）输入位置非法时

2

3

7

5

4

8

9

0//链表输入完成

9//输入的位置超出链表长度

ERROR//报错，并终止函数

2

3

7

5

4

8

9//输出原链表

请按任意键继续. . .

1. 分析

这个删除元素的方法沿用了一部分查找某个位置元素的方法，然后多创建了一个“滞后”指针，用来赋值前指针进行delete前的链接工作，最后刷出链表，与要求吻合。

//任务二：

//在实验任务一的基础上，实现插入排序

1.代码展示

#include<iostream>

using namespace std;

struct Node

{

Node\* next;

int data;

};

bool list\_search(Node\* L, int i, int& e)//在带头结点的单链表L中查找第i个元素，如果查找成功则用e返回其值

{

Node\* p = L->next; /\*使指针p指向第1个元素结点\*/

int j = 1; /\*计数器赋初值为1\*/

while (p && j < i) /\*顺着后继指针查找第i个元素结点\*/

{

p = p->next;

j++;

}

if (!p && j > i)

return 0; /\*如果i值不合法，即i值小于1或大于表长，则出错\*/

e = p->data; /\*如果第i个元素存在，则将该元素值赋给e\*/

return 1;

}

bool list\_insert(Node\*& L, int i, int x)//在带头结点的单链表L中第i个位置之前插入新元素x

{

Node\* p = L;

int j = 0;

while (p != NULL && j < i - 1) /\*寻找插入位置，并使p指向插入位置的前驱结点，即L中的第i-1个位置\*/

{

p = p->next;

++j;

}

if (p == NULL || j > i - 1)

{

cout << "ERROR" << endl;

return 0; /\*若位置不正确,即i小于1或大于表的长度加1，则出错\*/

}

Node\* s = new Node; /\*分配一个新结点的空间\*/

s->data = x; /\*为新结点数据域赋值\*/

/\*下面两条语句就是完成修改链，将新结点s插入到p结点之后\*/

s->next = p->next; /\*新结点指针域指向p的后继结点\*/

p->next = s; /\*新结点成为p的后继结点\*/

return 1;

}

void list\_delete(Node\* L, int i)//在带头结点的单链表L中查找第i个元素，如果查找成功则删除该元素所在结点

{

Node\* q = L;

Node\* p = L->next; /\*使指针p指向第1个元素结点\*/

int j = 1; /\*计数器赋初值为1\*/

while (p && j < i) /\*顺着后继指针查找第i个元素结点\*/

{

q = p;

p = p->next;

j++;

}

if (!p && j > i)

{

cout << "ERROR" << endl;

return; /\*如果i值不合法，即i值小于1或大于表长，则出错\*/

}

//如果第i个元素存在，则将该元素所在结点删除

q->next = p->next;

delete p;

return;

}

void insertsort(Node\* L, int n)

{

for (int i = 1; i <= n - 1; i++)

{

int max = 0;

int maxp = 1;

for (int j=i;j<=n;j++)

{

int e = 0;

list\_search(L, j, e);

if (e > max)

{

max = e;

maxp = j;

}

}

list\_delete(L, maxp);

list\_insert(L, i, max);

}

}

int main()

{

Node\* head = new Node;

head->next = NULL;

Node\* r = head;

int Nodenum;

cin >> Nodenum;

while (Nodenum != 0)

{

Node\* p = new Node;

p->data = Nodenum;

r->next = p;

p->next = NULL;

r = p;

cin >> Nodenum;

}

int nums;

cin >> nums;

insertsort(head, nums);

cout << "OK" << endl;

head = head->next;

while (head != NULL)

{

cout << head->data << endl;;

head = head->next;

}

system("pause");

return 0;

}

运行示例

2

3

7

5

4

8

9

0//链表输入完成

7//输入链表的长度

OK//提示排序完成

9

8

7

5

4

3

2//输出排序后的链表

请按任意键继续. . .

2.分析

这里的插入排序综合运用了之前的链表的创建、插入、查找与删除，实现了较为高效的排序，利用了链表的特点，完成了对于数组来说十分低效的任务。

**任务三：**

1.实验要求

（1）输入链表的长度和各元素的值，用尾插法建立双向循环链表，并输出链表中各元素值，观察输入的内容与输出的内容是否一致。

（2）在双向循环链表的第i个元素之前/后插入一个值为x的元素，并输出插入后的链表中各元素值。

（3）删除双向循环链表中第i个元素，并输出删除后的链表中各元素值。

（4）在双向循环链表中查找值为x元素，如果查找成功，则显示该元素在链表中的位置，否则显示该元素不存在。

2, 对比单链表核心算法提示

双向循环链表采用的存储结构描述如下：

struct DULNODE

{ Elemtype data; /\*数据域\*/

struct DULNODE \*prior; /\*指向前驱结点的指针域\*/

struct DULNODE \*next;/\*指向后继结点的指针域\*/

} DulNODE,\*DuLinklist;

不论是建立双向循环链表还是在双向循环链表中进行插入、删除和查找操作，其操作方法和步骤都跟单链表类似。只不过要注意两点：

（1）凡是在操作中遇到有修改链的地方，都要进行前驱和后继两个指针的修改。

（2）单链表操作算法中的判断条件：p= =NULL 或p!=NULL ,在循环链表的操作算法中则需改为：p!= L,其中L为链表的头指针。

代码展示

（1）输入链表的长度和各元素的值，用尾插法建立双向循环链表，并输出链表中各元素值，观察输入的内容与输出的内容是否一致。

#include<iostream>

using namespace std;

struct DULNODE

{

int data; /\*数据域\*/

struct DULNODE\* prior; /\*指向前驱结点的指针域\*/

struct DULNODE\* next;/\*指向后继结点的指针域\*/

};

int main()

{

DULNODE\* Head = new DULNODE;

DULNODE\* r;

r = Head;

int data;

cin >> data;

while (data)

{

DULNODE\* L = new DULNODE;

L->data = data;

r->next = L;

L->next = Head;

L->prior = r;

r = L;

cin >> data;

}

DULNODE\* s;

s = Head->next;

while (s != Head)

{

cout << s->data << endl;;

s = s->next;

}

运行示例

2

3

6

5

8

4

0//输入结束

2

3

6

5

8

4

请按任意键继续. . .

system("pause");

return 0;

}

分析：

双向循环链表的区别在于双向和循环，双向意味着一个节点既存储前指针又存储后指针，循环意味着最后一个指针的后指针指向头结点，实现了链表的循环。因此，在读取链表时，要注意判断结束的标准。

（2）在双向循环链表的第i个元素之前/后插入一个值为x的元素，并输出插入后的链表中各元素值。

代码展示

#include<iostream>

using namespace std;

struct DULNODE

{

int data; /\*数据域\*/

struct DULNODE\* prior; /\*指向前驱结点的指针域\*/

struct DULNODE\* next;/\*指向后继结点的指针域\*/

};

void insertfront(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* p = head;

while (i<=position)

{

p= p->next;

i++;

if (p == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* s = new DULNODE;

s->data = data;

(p->prior)->next=s;

s->prior = p->prior;

s->next = p;

p->prior = s;

}

void insertbehind(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* s = head;

while (i <= position)

{

s = s->next;

i++;

if (s == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* k = new DULNODE;

k->data = data;

(s->next)->prior = k;

k->prior = s;

k->next = s->next;

s->next = k;

}

int main()

{

DULNODE\* Head = new DULNODE;

DULNODE\* r;

r = Head;

int data;

cin >> data;

while (data)

{

DULNODE\* L = new DULNODE;

L->data = data;

r->next = L;

L->next = Head;

L->prior = r;

r = L;

cin >> data;

}

int position;

cin >> position;

cin >> data;

insertbehind(Head, position, data);

DULNODE\* s;

s = Head->next;

while (s != Head)

{

cout << s->data << endl;;

s = s->next;

}

system("pause");

return 0;

}

代码运行示例：

1. 前插入

2

3

7

5

4

8

9

0//输入链表完成

3//输入前插入的结点位置

2//输入插入的元素

2

3

2

7

5

4

8

9//输出插入后的链表

请按任意键继续. . .

1. 后插入

2

3

7

5

4

8

9

0//输入链表完成

3//输入后插入的结点位置

2//输入插入的元素

2

3

7

2

5

4

8

9//输出插入后的链表

请按任意键继续. . .

分析

使用双向循环链表进行插入的优势在于可以在前面或者后面插入，而不像单向链表只能向后插入，不过，需要注意的是，双向链表在每次插入时必须修改前后结点的前/后指针，还要同时为插入的结点进行前后指针的赋值。其他操作就与单链表大同小异了。

//（3）删除双向循环链表中第i个元素，并输出删除后的链表中各元素值。

代码展示：

#include<iostream>

using namespace std;

struct DULNODE

{

int data; /\*数据域\*/

struct DULNODE\* prior; /\*指向前驱结点的指针域\*/

struct DULNODE\* next;/\*指向后继结点的指针域\*/

};

void insertfront(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* p = head;

while (i<=position)

{

p= p->next;

i++;

if (p == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* s = new DULNODE;

s->data = data;

(p->prior)->next=s;

s->prior = p->prior;

s->next = p;

p->prior = s;

}

void insertbehind(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* s = head;

while (i <= position)

{

s = s->next;

i++;

if (s == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* k = new DULNODE;

k->data = data;

(s->next)->prior = k;

k->prior = s;

k->next = s->next;

s->next = k;

}

void deletenode(DULNODE\* head, int position)

{

int i = 1;

DULNODE\* s = head;

while (i <= position)

{

s = s->next;

i++;

if (s == head)

{

cout << "ERROR" << endl;

return;

}

}

(s->prior)->next = s->next;

(s->next)->prior = s->prior;

delete s;

}

int main()

{

DULNODE\* Head = new DULNODE;

DULNODE\* r;

r = Head;

int data;

cin >> data;

while (data)

{

DULNODE\* L = new DULNODE;

L->data = data;

r->next = L;

L->next = Head;

L->prior = r;

r = L;

cin >> data;

}

int position;

cin >> position;

deletenode(Head, position);

DULNODE\* s;

s = Head->next;

while (s != Head)

{

cout << s->data << endl;;

s = s->next;

}

system("pause");

return 0;}

代码运行示例

2

3

7

5

4

8

9

0//输入链表完成

3/输入想要删除的结点位置

2

3

5

4

8

9/输出删除后的链表

请按任意键继续. . .

分析：

删除结点的操作比较简单，就是在找到对应位置的结点后将前后结点进行链接，再删除该结点即可。

//（4）在双向循环链表中查找值为x元素，如果查找成功，则显示该元素在链表中的位置，否则显示该元素不存在。

代码展示：

#include<iostream>

using namespace std;

struct DULNODE

{

int data; /\*数据域\*/

struct DULNODE\* prior; /\*指向前驱结点的指针域\*/

struct DULNODE\* next;/\*指向后继结点的指针域\*/

};

void insertfront(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* p = head;

while (i<=position)

{

p= p->next;

i++;

if (p == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* s = new DULNODE;

s->data = data;

(p->prior)->next=s;

s->prior = p->prior;

s->next = p;

p->prior = s;

}

void insertbehind(DULNODE\* head, int position,int data)

{

int i = 1;

DULNODE\* s = head;

while (i <= position)

{

s = s->next;

i++;

if (s == head)

{

cout << "ERROR" << endl;

return;

}

}

DULNODE\* k = new DULNODE;

k->data = data;

(s->next)->prior = k;

k->prior = s;

k->next = s->next;

s->next = k;

}

void deletenode(DULNODE\* head, int position)

{

int i = 1;

DULNODE\* s = head;

while (i <= position)

{

s = s->next;

i++;

if (s == head)

{

cout << "ERROR" << endl;

return;

}

}

(s->prior)->next = s->next;

(s->next)->prior = s->prior;

delete s;

}

void search(DULNODE\* head, int num)

{

DULNODE\* s = head->next;

int i = 1;

while (s != head)

{

if (s->data == num)

{

break;

}

s = s->next;

i++;

}

if (s->data != num)

{

cout << "404 NOT FOUND" << endl;

return;

}

else

{

cout << "是第" << i << "个元素" << endl;

}

}

int main()

{

DULNODE\* Head = new DULNODE;

DULNODE\* r;

r = Head;

int data;

cin >> data;

while (data)

{

DULNODE\* L = new DULNODE;

L->data = data;

r->next = L;

L->next = Head;

L->prior = r;

r = L;

cin >> data;

}

int num;

cin >> num;

search(Head, num);

system("pause");

return 0;

}

代码运行示例

1. 成功查找到元素

2

3

7

5

4

8

9

0//链表输入完成

3//输入想要查找的元素

是第2个元素//输出元素所在位置

请按任意键继续. . .

1. 未成功查找到元素

2

3

7

5

4

8

9

0//链表输入完成

1//输入想要查找的元素

404 NOT FOUND//未找到该元素，程序报错并结束运行

请按任意键继续. . .

分析：

利用双向循环链表进行查找的方式很简单，只要从头节点往后依次查询即可。

**实验总结：**

这次实验使我系统性地接触链表这一结构概念，而且不是以封装好后的形式，而是以最基本、也是最底层的逻辑出发，探究链表这一结构的优缺点、与数组的异同，以及对于插入排序的独特优势。虽然学习的过程不容易，但是在完成所有任务后仍然感到十分有成就感，也为本学期的实验作业画上了一个圆满的句号。先人有云：“实践是检验真理的唯一标准”，唯有通过不断的练习与实验，在机器面前一行一行地敲出代码，才能真正深入而贴切地感受到一门语言及其背后的逻辑思维的魅力，期待下个学期，加油！