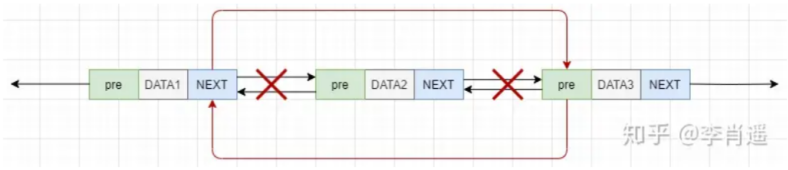
Final

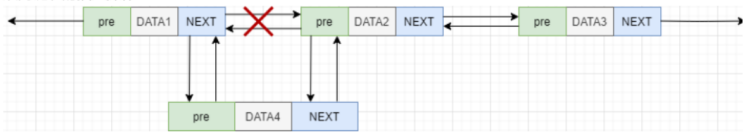
双向链表的删除



选择需要删除的结点->选中这个结点的前一个结点->将前一个结点的next指针指向自己的下一个结点->选中该节点的下一个结点->将下一个结点的pre指针修改指向为自己的上一个结点。在进行遍历的时候直接将这一个结点给跳过了，之后，我们释放删除结点，归还空间给内存

```c
//删除元素
line * deleteLine(line * head,int data){
    //输入的参数分别为进行此操作的双链表，需要删除的数据
    line * list=head;
    //遍历链表
    while (list) {//直到null都没找到，停止操作
        //判断是否与此元素相等
        //删除该点方法为将该结点前一结点的next指向该节点后一结点
        //同时将该结点的后一结点的pre指向该节点的前一结点
        if (list->data==data) {
            list->pre->next=list->next;
            list->next->pre=list->pre;
            free(list);
            printf("--删除成功--\n");
            return head;
        }
        list=list->next;
    }
    printf("Error:没有找到该元素，没有产生删除\n");
    return head;
}
```

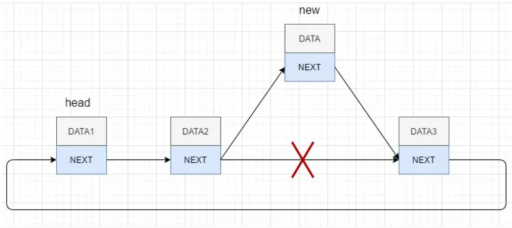
双向链表的插入操作



对于每一次的双向链表的插入操作，首先需要创建一个独立的结点，并通过malloc操作开辟相应的空间；
实现代码：

```c
//插入数据
line * insertLine(line * head,int data,int add){
    //三个参数分别为：进行此操作的双链表，插入的数据，插入的位置
    //新建数据域为data的结点
    line * temp=(line*)malloc(sizeof(line));
    temp->data=data;
    temp->pre=NULL;
    temp->next=NULL;
    //插入到链表头，要特殊考虑
    if (add==1) {
        temp->next=head;
        head->pre=temp;
        head=temp;
    }else{
        line * body=head;
        //找到要插入位置的前一个结点
        for (int i=1; i<add-1; i++) {
            body=body->next;
        }
        //判断条件为真，说明插入位置为链表
        if (body->next==NULL) {
            body->next=temp;
            temp->pre=body;
        }else{
            body->next->pre=temp;
            temp->next=body->next;
            body->next=temp;
            temp->pre=body;
        }
    }
    return head;
}
```

循环单链表的插入
如图，对于插入数据的操作，可以创建一个独立的结点，通过将需要插入的结点的上一个结点的next指针指向该节点，再由需要插入的结点的next指针指向下一个结点的方式完成插入操作。




```c
//插入元素
list *insert_list(list *head,int pos,int data){
    //三个参数分别是链表，位置，参数
    list *node=initlist();  //新建结点
    list *p=head;       //p表示新的链表，
    list *t;            //和temp类似
    t=p;
    node->data=data;
    if(head!=NULL){
        for(int i=1;i<pos;i++){
            t=t->next;  //走到需要插入的位置处
        }
        node->next=t->next;
        t->next=node;
        return p;
    }
    return p;
}
```

遍历双链表

```c
//遍历双链表,同时打印元素数据
void printLine(line *head){
    line *list = head;
    int pos=1;
    while(list){
        printf("第%d个数据是:%d\n",pos++,list->data);
        list=list->next;
    }
}
```

双链表的基本操作

优势：从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。
设计代码：

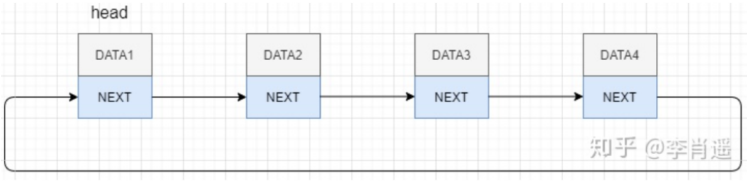
```c
typedef struct line{
    int data;           //data
    struct line *pre;   //pre node
    struct line *next;  //next node
}line,*a;
```

双链表的创建：
先创建头结点，然后逐步的进行添加双向链表的头结点是有数据元素的，也就是头结点的data域中是存有数据的，这与一般的单链表是不同的。
对于逐步添加数据，先开辟一段新的内存空间作为新的结点，为这个结点进行的data进行赋值，然后将已成链表的上一个结点的next指针指向自身，自身的pre指针指向上一个结点。

```c
//创建双链表
line* initLine(line * head){
    int number,pos=1,input_data;
    //三个变量分别代表结点数量，当前位置，输入的数据
    printf("请输入创建结点的大小\n");
    scanf("%d",&number);
    if(number<1){return NULL;} //输入非法直接结束
    //////头结点创建//////
    head=(line*)malloc(sizeof(line));//重要的部分
    head->pre=NULL;
    head->next=NULL;
    printf("输入第%d个数据\n",pos++);
    scanf("%d",&input_data);
    head->data=input_data;

    line * list=head;
    while (pos<=number) {
        //创建新节点
        line * body=(line*)malloc(sizeof(line));
        body->pre=NULL;
        body->next=NULL;
        printf("输入第%d个数据\n",pos++);
        scanf("%d",&input_data);
        body->data=input_data;
        //头结点和新节点相互连接
        list->next=body;
        body->pre=list;
        list=list->next;
    }
    return head;
}
```

循环链表



定义代码：

```c
typedef struct list{
    int data;
    struct list *next;
}list;
//data为存储的数据，next指针为指向下一个结点
```

循环单链表初始化

先创建一个头结点并且给其开辟内存空间，在开辟内存空间成功之后，将头结点的next指向head自身，创建一个init函数来完成；

```c
//初始结点
list *initlist(){
    list *head=(list*)malloc(sizeof(list));
    if(head==NULL){
        printf("创建失败，退出程序");
        exit(0);
    }else{
        head->next=NULL;
        return head;
    }
}
    //////////初始化头结点//////////////
    list *head=initlist();
    head->next=head;
```

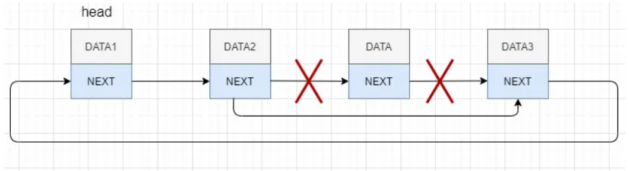
为了重复创建和插入，我们可以在init函数重新创建的结点next指向空，而在主函数调用创建之后，将head头结点的next指针指向自身。

循环单链表的遍历

先找到尾节点的位置，由于尾节点的next指针是指向头结点的，所以不能使用 链表本身是否为空（NULL） 的方法进行简单的循环判断，我们需要通过 判断结点的next指针是否等于头结点的方式 进行是否完成循环的判断。

```c
//遍历元素
int display(list *head) {
    if(head != NULL) {
        list *p  = head;
        //遍历头节点到，最后一个数据
        while(p->next != head ) {
            printf("%d   ",p->next->data);
            p = p->next;
        }
        printf("\n");   //换行
        //把最后一个节点赋新的节点过去
        return 1;
    } else {
        printf("头结点为空!\n");
        return 0;
    }
}
```

循环单链表的删除




```c
//删除元素
int delete_list(list *head) {
    if(head == NULL) {
        printf("链表为空！\n");
        return 0;
    }
    //建立临时结点存储头结点信息（目的为了找到退出点）
    //如果不这么建立的化需要使用一个数据进行计数标记，计数达到链表长度时自动退出
    //循环链表当找到最后一个元素的时候会自动指向头元素，这是我们不想让他发生的
    list *temp = head;
    list *ptr = head->next;

    int del;
    printf("请输入你要删除的元素：");
    scanf("%d",&del);

    while(ptr != head) {
        if(ptr->data == del) {
            if(ptr->next == head) {
                temp->next = head;
                free(ptr);
                return 1;
            }
            temp->next = ptr->next;    //核心删除操作代码
            free(ptr);
            //printf("元素删除成功！\n");
            return 1;
        }
        temp = temp->next;
        ptr = ptr->next;
    }
    printf("没有找到要删除的元素\n");
    return 0;
}
```

程序如何打开文件

```c
FILE *fopen (const char* path,  const char* mode)
//path is the file name(从文件目前的工作目录开始)
//mode 决定文件是reading, writing or both
//return a new stream for success or NULL for failure
```

"r" or "rb" read only
"w" or "wb" write only (after deleting, 如果文件不存在，会创建一个新文件，程序从文件开头写入内容。如果文件存在，则该会被截断为零长度，重新写入)
"a" or "ab" append (write only, * at end，您的程序会在已有的文件内容中追加内容)
"r+" / "r+b" / "rb+" open r / w (read/write)
"w+"/"w+b"/"wb+" truncate文件会被截断为零长度, then r/w
"a+"/"a+b"/"ab+" append r/w，读取会从文件的开头开始，写入则只能是追加模式。

如何关闭这个文件

```c
int fclose (FILE* stream);
//return 0 for success. and EOF(-1) for failure
```

从文件中读取字符

```c
int fgetc (FILE* stream);//从stream指向的输入文件中读取一个字符，返回值是读取的字符，如果发生错误则返回 EOF(the int -1),返回0xFF才是byte
int getc(FILE* stream);
```

写入字符

```c
int fputc (int c, FILE* stream); //函数 fputc()把参数c的字符值写入到stream所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 EOF。
int putc (int c, FILE* stream);
```

从文件中读取字符串

```c
char *fgets(char *s, int size, FILE *stream);//s is an array of characters
                                             //size is the size of the array
                                           //stream is the stream from which to read
                                            //returns s or NULL on failure
//注意：从stream所指向的输入流中读取n-1个字符。它会把读取的字符串复制到缓冲区s，最后追加一个null字符来终止字符串。
//如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。end of a line (ASCII 0x0A or 0x0D),end of the input (such as a file),end of array s (leaving room for a NUL)
```

向文件中写入一个string

```c
int fputs (const char* s, FILE *stream);
//把字符串s写入到stream所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回EOF。
writing a string to stdout
int puts (const char* s);
//puts adds an end of line sequence  (linefeed, ASCII 0x0A, on Unix) to the end of the string (fputs does not).
```

stdin标准输入（键盘），stdout标准输出（屏幕）
用scanf/printf 用于标准I/O with stdin/stdout

```c
int scanf (const char* format, ...);
//reads formatted input from stdin.从键盘读取并格式化。
int printf (const char* format, ...);
//writes formatted output to stdout.发送格式化输出到屏幕。
```

用fscanf从文件流stream上读取标准化输入

```c
int fscanf (FILE* stream, const char* format, [argument...]]);
//根据数据格式 const char * format, 从文件FILE* stream中 , 读取数据存储到 [argument...]参数中.
//returns number of conversions返回转换次数 or -1 on failure
```

用fprintf向文件流stream上写标准化输出

```c
int fprintf (FILE* stream, const char* format, ...);
//returns number of characters printed打印的字符数 or negative number on failure
例如：fprintf (out, "%d\n", p->n_nodes);
```

二进制I/O函数的输入输出
用fread从文件流stream上读取binary输入

```c
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements,
FILE *stream);
//ptr指向data存储的地址
//一个element的size
//element的数量
//stream从那个文件中读
//returns number of element read or 0 on failure
```

用fwrite写入binary输出到文件流stream

```c
size_t fwrite(const void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);
//向stream从那个文件中写
//returns number of element written or 0 on failure
```

2. **(7 points)** Write the code to prepare streams for I/O files based on the command-line arguments.

```
in_file = fopen (argv[2], "r");

if (NULL == in_file) {

    return -1;

}

out_file = fopen (argv[3], "w");

if (NULL == out_file) {

    fclose (in_file);

    return -1;

}
```

---

3. **(10 points)** Write the code to apply the chosen operation to every line of the input file and write the result to the output file.

```
while (2 == fscanf (in_file, "%d%d", &a, &b)) {

    if (0 > fprintf (out_file, "%d\n", (*(func_arr[func_index])) (a, b))) {

        fclose (in_file);

        fclose (out_file);

        return -1;

    }

}
```

---

4. **(5 points)** Write the code to release resources and return success.

```
fclose (in_file);

fclose (out_file); // can check return value here instead of fprintf above

return 0;
```

```
int32_t file_reduce (const char* fname)
{
    FILE* in;  // input stream
    FILE* out; // output stream
    // First, write code to prepare the streams for use.

    if (NULL == (in = fopen (fname, "r")) ||
        NULL == (out = fopen ("out.txt", "w"))) {
        if (NULL != in) {
            fclose (in);
        }
        return 0;
    }
```

```
    // Read the input file and produce the output.
    int last = EOF, char;

    while (EOF != (char = fgetc (in))) {
        if (last != char) {
            fputc (char, out);
            last = char;
        }
    }
```

```
    // Clean up and return.

    fclose (in);
    return (0 == fclose (out) ? 1 : 0);
}
```