# 第5章 链表

链表可以说是一种最基础的数据结构了。链表由一组元素以一种特定的顺序组合或链接在一起，在维护数据的集合时很有用。这一点同我们常用到的数组很相似。然而，链表在很多情况下比数组更有优势。特别是在执行插入和删除操作时链表拥有更高的效率。链表需要动态的开辟存储空间，也就是存储空间是在程序运行时分配的。由于在很多应用中数据的大小在编译时并不能确定，因此这种动态分配空间的特性也是链表的一个优点。

本章将涵盖以下主题：

单链表

最简单的链表形式。元素之间由一个单独的指针所链接。这种结构的链表允许我们从第一个元素开始遍历到最后一个元素。

双向链表

这种形式的链表元素之间通过两个指针链接。双向链表可以正向，也可以反向遍历。

循环链表

这种形式的链表最后一个元素的指针域指向链表的首元素，而不是设为NULL。这种结构的链表允许我们进行循环遍历。

一些链表的应用包括：

邮件列表

这种应用常在电子邮件程序中看到。因为无法预测一个邮件列表到底有多长，发件人需要在发送消息之前先建立一个地址链表。

滚动列表

图形用户界面中常能看到这种组件。通常在滚动列表中与每个条目相关的数据并不显示出来。一种管理这种“隐藏”数据的方式是维护一个链表，链表中每个元素保存一个在滚动列表中出现的条目数据。

多项式计算

数学中的一个重要组成部分。但对于大多数编程语言来说并没有将其作为原生的数据类型加以支持。如果我们让链表中的每个元素存储多项式中的其中一项，那么链表对于表示出一个多项式就是非常有用的，比如3x 2 + 2x + 1。

内存管理（本章中将会讲到）

内存管理是操作系统中的一项重要任务。操作系统必须决定如何为系统中运行的进程分配和回收存储空间。链表能够用来跟踪可供分配的内存片段信息。

LISP

一种在人工智能领域中非常重要的编程语言。LISP，List Processor的简写，在进行符号处理时大量使用了链表。

文件的链式分配

为了消除磁盘上的外部碎片而采用的一种文件分配方式，但只适合于顺序访问。文件按照块的方式进行组织，每一块都包含一个指向文件下一块数据的指针。

其他的数据结构

有一些数据结构的实现需要依赖于链表，它们是栈、队列、集合、哈希表和图。这些数据结构都将在本书中出现。

## 5.1 单链表介绍

单链表，通常简称为链表，它是由单个元素之间通过一个指针彼此链接起来而组成。每个元素包含两个部分：数据成员和一个称为next的指针。通过采用这种结构，将每个元素的next指针设为指向其后面的元素（见图5.1）。最后一个元素的next指针被设为NULL，简单的表示链表的尾端。链表开始处的元素是“头”，链表末尾的元素称为“尾”。

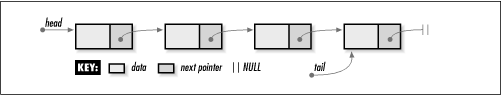


图5.1 元素链接在一起组成一个链表

要访问链表中的某个元素，我们从链表头开始，通过next指针从一个元素到另一个元素连续的遍历直到找到我们所需要的那个元素为止。以单链表来说，只能以一个方向进行遍历——从头到尾——因为每个元素中并没有维护其前一个元素的链接。因此，如果我们从链表头开始移动到了某个元素，然后我们希望访问这个位置之前的某个元素，则我们必须又从头开始（尽管某些时候我们事先知道需要访问这个元素，并保存了指向该元素的指针）。通常，这个缺点不是什么大问题。当有必要的时候，我们可以采用双向链表或循环链表。

从概念上说，可以把链表想象成一系列连续的元素。然而，由于这些元素是动态分配的（在C语言中通过调用malloc），因此很重要的一点是，切记这些元素通常实际上都是分散在内存空间中的（见图5.2）。元素与元素之间的链接关系只是为了确保所有的元素都可以访问到。带着这种思考，我们等会会看到当维护元素之间的链接信息时需要特别的小心。如果我们错误的丢失了一个链接，则从这个位置开始往后的所有元素我们都无法访问到了。因此，俗话说“你的弱点有多弱，你的强度就有多强”非常适用于描述链表的特点。

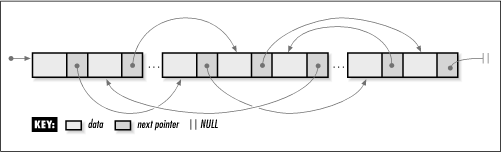


图5.2 链表元素链接在了一起，但在内存空间中它们是分散开的

## 5.2 单链表接口的定义

***list\_init***

void list\_init(List \**list*, void (\**destroy*)(void \**data*));

**返回值**

无

**描述**

初始化由参数list指定的链表。必须在链表做其他操作之前调用。当list\_destroy调用时，destroy参数提供了一种释放动态分配数据空间的方法。例如，如果链表中包含采用malloc动态分配的数据，当链表被销毁时destroy应该被设为free用来释放数据空间。对于包含了好几个动态分配成员的结构化数据，destroy应该被设为一个用户自定义的函数，通过对每一个动态分配的成员以及对结构体自身调用free来释放空间。对于链表中包含的不需要释放的数据，destroy应该设为NULL值。

**复杂度**



***list\_destroy***

void list\_destroy(List \**list*);

**返回值**

无

**描述**

销毁由参数list指定的链表。调用list\_destroy后任何其他的操作都不允许执行，除非再次调用list\_init。list\_destroy将链表中所有的元素都移除，如果传给list\_init的参数destroy不为NULL的话，则调用destroy所指定的函数，对链表中每个移除的元素数据施行资源回收操作。

**复杂度**

， 代表链表中的元素个数。

***list\_ins\_next***

int list\_ins\_next(List \**list*, ListElmt \**element*, const void \**data*);

**返回值**

如果插入元素成功返回0， 失败返回-1。

**描述**

在element所指定的元素后插入一个新元素。如果element指定为NULL，则新元素插入到链表头部。新元素包含一个指向data的指针，因此只要该元素还在链表中，data所引用的内存空间就应该保持合法。管理data所引用的存储空间是调用者的责任。

**复杂度**



***list\_rem\_next***

int list\_rem\_next(List \**list*, ListElmt \**element*, void \*\**data*);

**返回值**

如果移除元素成功返回0， 否则返回-1。

**描述**

移除由element所指定的元素后的那个元素。如果element设定为NULL，则移除链表头元素。调用返回后，data指向被移 除元素的数据部分。由调用者负责管理data所引用的存储空间。

**复杂度**



***list\_size***

int list\_size(const List \**list*);

**返回值**

链表中元素的个数

**描述**

这是一个宏，用来计算由参数list指定的链表中元素个数

**复杂度**

****

***list\_head***

ListElmt \*list\_head(const List \**list*);

**返回值**

指向链表中头元素的指针

**描述**

这是一个宏，返回由参数list指定的链表中的头元素的指针

**复杂度**

****

***list\_tail***

ListElmt \*list\_tail(const List \**list*);

**返回值**

指向链表末尾元素的指针

**描述**

这是一个宏，返回由参数list指定的链表中的尾部元素的指针

**复杂度**

****

***list\_is\_head***

int list\_is\_head(const ListElmt \**element*);

**返回值**

如果element所指定的元素是头节点，返回1，否则返回-1。

**描述**

这是一个宏，用来判断由element所指定的元素是否是链表头节点

**复杂度**

****

***list\_is\_tail***

int list\_is\_tail(const ListElmt \**element*);

**返回值**

如果element所指定的元素是链表末尾节点，返回1，否则返回-1.

**描述**

这是一个宏，用来判断element所指定的链表元素是否为链表的末尾节点。

**复杂度**

****

***list\_data***

void \*list\_data(const ListElmt \**element*);

**返回值**

节点中保存的数据。

**描述**

这是一个宏，返回由element所指定的节点元素中保存的数据。

**复杂度**

****

***list\_next***

ListElmt \*list\_next(const ListElmt \**element*);

**返回值**

返回由element所指定的节点的下一个节点。

**描述**

这是一个宏，返回链表中由element所指定的节点的下一个节点。

**复杂度**

****

## 5.3 单链表的实现与分析

我们回顾一下链表元素的组成：一个数据成员和一个指向链表中下一个元素的指针。结构体ListElmt表示链表中的单个元素（见例子5.1）。如同你所预料到的，这个结构体拥有两个成员，就是我们上面所说的数据成员和指针成员。结构体List则表示链表这种数据结构（见例子5.1）。这个结构体由5个成员组成：size表示链表中的元素个数；match并不由链表本身使用，而是由从链表数据结构继承而来的新类型所使用；destroy是封装之后传递给list\_init的析构函数；head是指向链表中头节点元素的指针；tail则是指向链表中末尾节点元素的指针。

**代码示例5.1 链表抽象数据类型的头文件**

/\* list.h \*/

#ifndef LIST\_H

#define LIST\_H

#include <stdlib.h>

/\* Define a structure for linked list elements. \*/

typedef struct ListElmt\_

{

void \*data;

struct ListElmt\_ \*next;

} ListElmt;

/\* Define a structure for linked lists. \*/

typedef struct List\_

{

int size;

int (\*match)(const void \*key1, const void \*key2);

void (\*destroy)(void \*data);

ListElmt \*head;

ListElmt \*tail;

} List;

/\* Public Interface \*/

void list\_init(List \*list, void (\*destroy)(void \*data));

void list\_destroy(List \*list);

int list\_ins\_next(List \*list, ListElmt \*element, const void \*data);

int list\_rem\_next(List \*list, ListElmt \*element, void \*\*data);

#define list\_size(list) ((list)->size)

#define list\_head(list) ((list)->head)

#define list\_tail(list) ((list)->tail)

#define list\_is\_head(list, element) ((element) == (list)->head ? 1 : 0)

#define list\_is\_tail(element) ((element)->next == NULL ? 1 : 0)

#define list\_data(element) ((element)->data)

#define list\_next(element) ((element)->next)

#endif

### 5.3.1 list\_init

list\_init用来初始化一个链表以便于其他的操作能够执行（见例子5.2）。初始化链表是一种简单的操作，我们只要把链表的size成员设为0，把函数指针成员destroy设为我们定义的析构函数，head和tail指针全设为NULL即可。

list\_init的计算复杂度为，因为初始化过程中的所有步骤都能在一个恒定的时间内完成。

### 5.3.2 list\_destroy

　　　list\_destroy用来销毁链表（见例子5.2），其意义就是将链表中的所有元素都移除。如果调用list\_init时destroy参数不为NULL的话，则当每个元素都被移除时将调用destroy一次。list\_destroy总的计算复杂度为，代表链表中的元素个数，这是因为list\_rem\_next的计算复杂度为，而移除每个元素时都将调用list\_rem\_next一次。

### 5.3.3 list\_ins\_next

list\_ins\_next将一个元素插入到由element参数所指定的元素之后（见例子5.2）。该调用将新元素的数据域指向由用户传递进来的数据上。插入新元素的处理步骤很简单，但需要特别小心。有两种情况需要考虑：插入到链表头部和插入到其它位置。

总的来说，要插入一个元素到链表中，我们将新元素的next指针指向它之后的那个元素，然后我们将新元素位置之前的节点next指针指向我们新插入的元素（见图5.3）。但是，当从链表头部插入时，新元素之前没有别的节点了。因此，在这种情况下，我们将新元素的next指针指向当前链表的头部，然后重置头节点指针，使其指向我们的新元素。回顾一下前一节中的接口设计，当传入的element为NULL时代表新的元素将插入到链表头部。另外需要注意的是，无论何时当我们插入的元素位于链表末尾时，我们都必须更新链表的tail成员使其指向新的节点。最后，我们更新统计链表中节点个数的size成员。

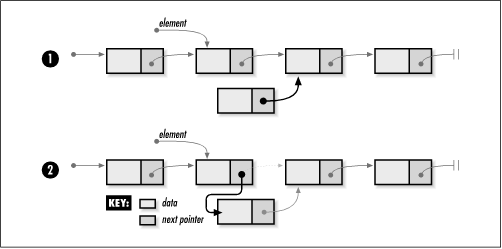


图5.3 将节点插入到链表中

### 5.3.4 list\_rem\_next

list\_rem\_next移除由element所指定的元素之后的那个节点（见例子5.2）。移除element之后的那个节点而不是移除element本身，关于为何要这样设计的原因我们将在本章的问与答中进行讨论。同插入节点类似，这个调用也需要考虑两个因素：移除的是头节点以及移除其余位置上的节点。

移除操作是很简单的，但同样需要注意一些细节问题（见图5.4）。总的来说，为了从链表中移除一个元素，我们将要移除的目标节点前一个元素的next指针指向目标节点的下一个元素。但是，当移除的目标节点是头结点时，目标节点之前并没有其它元素了，因此在这种情况下，我们将链表的head成员指向目标节点的下一个元素。同插入操作一样，当传入的element为NULL时就代表链表的头结点需要被移除。另外，无论何时我们移除的目标节点是链表的尾部节点时，我们都必须更新tail指针，使其指向新的尾节点，或者当移除操作使得整个链表成为空链表时需要把tail设置为NULL。最后，我们更新链表的size成员，使其减1。当这个调用返回时，入参data将指向移除节点的数据域。

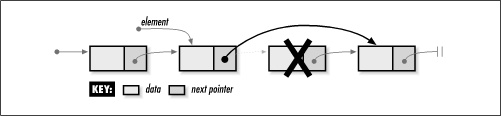


图5.4 从链表中移除节点

list\_rem\_next的计算复杂度为，因为所有的移除步骤都在恒定的时间内完成。

### 5.3.5 list\_size, list\_head, list\_tail, list\_is\_tail, list\_data以及list\_next

这些宏实现了链表中的一些简单操作（见例子5.1）。总的来说，它们提供了快速访问和检测List和ListElmt结构体中的成员的能力。这些操作的计算复杂度都是的，因为访问和检测结构体的成员都可以在恒定的时间内完成。

**代码示例 5.2 链表抽象数据类型的实现**

/\* list.c \*/

#include <stdlib.h>

#include <string.h>

#include "list.h"

/\* list\_init \*/

void list\_init(List \*list, void (\*destroy)(void \*data))

{

/\* Initialize the list. \*/

list->size = 0;

list->destroy = destroy;

list->head = NULL;

list->tail = NULL;

return;

}

/\* list\_destroy \*/

void list\_destroy(List \*list)

{

void \*data;

/\* Remove each element. \*/

while (list\_size(list) > 0)

{

/\* Call a user-defined function to free dynamically allocated data. \*/

if (list\_rem\_next(list, NULL, (void \*\*)&data) == 0 && list->destroy

!= NULL)

{

list->destroy(data);

}

}

/\* No operations are allowed now, but clear the structure as a precaution. \*/

memset(list, 0, sizeof(List));

return ;

}

/\* list\_ins\_next \*/

int list\_ins\_next(List \*list, ListElmt \*element, const void \*data)

{

ListElmt \*new\_element;

/\* Allocate storage for the element. \*/

if ((new\_element = (ListElmt \*)malloc(sizeof(ListElmt))) == NULL)

return -1;

/\* Insert the element into the list. \*/

new\_element->data = (void \*)data;

if (element == NULL)

{

/\* Handle insertion at the head of the list. \*/

if (list\_size(list) == 0)

list->tail = new\_element;

new\_element->next = list->head;

list->head = new\_element;

}

else

{

/\* Handle insertion somewhere other than at the head. \*/

if (element->next == NULL)

list->tail = new\_element;

new\_element->next = element->next;

element->next = new\_element;

}

/\* Adjust the size of the list to account for the inserted element. \*/

list->size++;

return 0;

}

/\* list\_rem\_next \*/

int list\_rem\_next(List \*list, ListElmt \*element, void \*\*data)

{

ListElmt \*old\_element;

/\* Do not allow removal from an empty list. \*/

if (list\_size(list) == 0)

return -1;

/\* Remove the element from the list. \*/

if (element == NULL)

{

/\* Handle removal from the head of the list. \*/

\*data = list->head->data;

old\_element = list->head;

list->head = list->head->next;

if (list\_size(list) == 1)

list->tail = NULL;

}

else

{

/\* Handle removal from somewhere other than the head. \*/

if (element->next == NULL)

return -1;

\*data = element->next->data;

old\_element = element->next;

element->next = element->next->next;

if (element->next == NULL)

list->tail = element;

}

/\* Free the storage allocated by the abstract datatype. \*/

free(old\_element);

/\* Adjust the size of the list to account for the removed element. \*/

list->size--;

return 0;

}

## 5.4 使用链表的例子：页帧管理

在一些能够支持虚拟内存的系统中我们发现了关于链表的一种应用。虚拟内存是一种地址空间的映射机制，它允许进程（运行的程序）不必完全加载到物理内存中也可以得到运行。这种方式的一个优点是：进程可以使用到比系统实际所允许的物理内存大得多的地址空间。另一个优点是多个进程能够共享系统的内存以并发的方式执行。

运行在虚拟内存机制下的进程需要处理虚拟地址。这些地址对于进程来说就像是物理地址一样，但使用前必须由操作系统做转换。我们采用由专门的硬件所支持的页表来快速执行地址转换的工作。每一个进程都有它自己的页表，将它的虚拟地址空间映射到物理内存中的页帧上。当某个进程引用了一个虚拟地址，页表中的某项需要检查并决定该页关联到哪个物理页帧上（见图5.5）。当进程引用了一个不在物理页帧上的虚拟地址时，会导致系统产生一个页错误并为之在物理内存中分配一个页帧。为什么进程的页面会从物理内存中移除是另一个问题。一种导致进程页被移除的场景是：当某个页面被访问到的频率同其它页面相比很低时，而且该页帧在别处被用到的情况下。

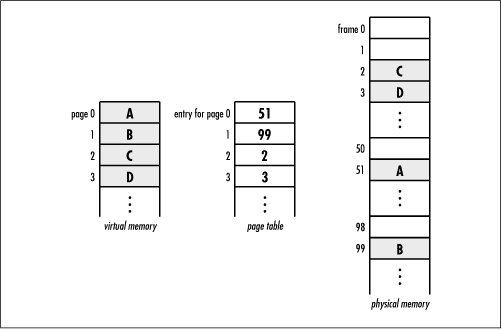


图5.5 虚拟内存映射到物理内存

下面这个例子就是针对我们刚刚描述过的页帧管理所设计的。为此，将介绍两个函数，

alloc\_frame和free\_frame（见例子5.3）。alloc\_frame和free\_frame采用链表来维护可供分配的页帧。函数alloc\_frame从空闲页帧链表中获取到空闲页帧号。给定这个特定的页，将页帧号放到页表中来检查该页面应该对应哪个物理页帧。一旦某个页面从物理内存中移除后，函数free\_frame接受一个页帧号并将其放回到空闲页帧链表中去。这两个函数假定在执行之前操作系统已经将所有的空闲页帧都插入到空闲页帧链表中了。稍后在本章关于循环链表的例子中将看到，当空闲页帧链表为空时去调用alloc\_frame会发生什么。

用链表来管理页帧是一种非常好的方法，因为页帧的分配将涉及到频繁的插入和删除操作，而且这些操作都发生在链表头。alloc\_frame和free\_frame的计算复杂度都是的，因为这两个函数都只是简单的分别调用list\_rem\_next以及list\_ins\_next，而这两个函数在前面已经分析过了，它们都是的操作。

**代码示例5.3 页帧管理的函数实现**

/\* frames.c \*/

#include <stdlib.h>

#include "frames.h"

#include "list.h"

/\* alloc\_frame \*/

int alloc\_frame(List \*frames)

{

int frame\_number, \*data;

/\* Return that there are no frames available. \*/

if (list\_size(frames) == 0)

return -1;

else

{

/\* Return that a frame could not be retrieved. \*/

if (list\_rem\_next(frames, NULL, (void \*\*)&data) != 0)

return -1;

else

{

/\* Store the number of the available frame. \*/

frame\_number = \*data;

free(data);

}

}

return frame\_number;

}

/\* free\_frame \*/

int free\_frame(List \*frames, int frame\_number)

{

int \*data;

/\* Allocate storage for the frame number. \*/

if ((data = (int \*)malloc(sizeof(int))) == NULL)

return -1;

/\* Put the frame back in the list of available frames. \*/

\*data = frame\_number;

if (list\_ins\_next(frames, NULL, data) != 0)

return -1;

return 0;

}

## 5.5 双向链表介绍

双向链表，如同其名字所暗示的那样，链表元素之间由两个指针链接。双向链表中的每一个元素都由3个部分组成：除了数据成员和next指针外，每个元素还包含一个指向其前驱元素的指针，称为prev指针。双向链表的组成是这样的：将一些元素链接在一起使得每个元素的next指针都指向其后继的元素，而每个元素的prev指针都指向其前驱元素。为了标识出链表的头和尾，我们将第一个元素的prev指针和最后一个元素的next指针设为NULL。

要反向遍历整个双向链表，我们使用prev指针以从尾到头的顺序连续访问各个元素。因此为每个元素增加了一个指针的代价，换来的是双向链表比单链表提供了更为灵活的访问方式。当我们知道某个元素存储在链表中的某处时，我们可以明智的选择按照何种方式去访问到它，这会非常有帮助。例如，双向链表的一种灵活性在于它提供了一种比单链表更直观的方式去移除一个元素。

## 5.6 双向链表接口的定义

***dlist\_init***

void dlist\_init(DList \**list*, void (\**destroy*)(void \**data*));

**返回值**

无

**描述**

初始化由参数list所指定的双向链表。该函数必须在双向链表做其它任何操作之前调用。当dlist\_destroy被调用时，这里传入的destroy参数提供了一种释放动态分配数据的方法。它的工作方式同前文叙述的list\_destroy很相似。对于双向链表中包含不需要手动释放空间的数据，destroy参数应该设为NULL。

**复杂度**

****

***dlist\_destroy***

void dlist\_destroy(DList \**list*);

**返回值**

无

**描述**

销毁由参数list所指定的双向链表。该函数调用后其他操作都不允许再执行，除非用户再次调用dlist\_init。dlist\_destroy函数将双向链表中的所有元素都移除，如果传给dlist\_init的参数destroy不为NULL的话，则调用destroy所指定的函数，对链表中每个移除的元素数据施行资源回收操作。

**复杂度**

**，**这里n代表双向链表中的元素个数。

***dlist\_ins\_next***

int dlist\_ins\_next(DList \**list*, DListElmt \**element*, const void \**data*);

**返回值**

如果插入操作成功，返回1，否则返回-1。

**描述**

将元素插入到由element所指定的元素之后。当插入到空链表中时，element可能指向任何位置，为了避免混淆，element此时应该设为NULL。新的元素包含一个指向data的指针，因此只要该元素仍在链表中时，data所引用的内存空间就应该保持合法。由调用者负责管理data所引用的存储空间。

**复杂度**

****

***dlist\_ins\_prev***

int dlist\_ins\_prev(DList \**list*, DListElmt \**element*, const void \**data*);

**返回值**

插入操作成功则返回0，否则返回-1。

**描述**

将元素插入到由element所指定的元素之前。当插入到空链表中时，element可能指向任何位置，为了避免混淆，element此时应该设为NULL。新的元素包含一个指向data的指针，因此只要该元素仍在链表中时，data所引用的内存空间就应该保持合法。由调用者负责管理data所引用的存储空间。

**复杂度**



***dlist\_remove***

int dlist\_remove(DList \**list*, DListElmt \**element*, void \*\**data*);

**返回值**

移除操作成功则返回0，否则返回-1。

**描述**

从链表中移除由element所指定的元素。函数返回后，参数data将指向存储在元素中的数据域。由调用者负责管理data所引用的存储空间。

**复杂度**

****

***dlist\_size***

int dlist\_size(const DList \**list*);

**返回值**

链表中的元素个数

**描述**

这是一个宏，用来计算由参数list所指定的双向链表中的元素个数。

**复杂度**

****

***dlist\_head***

DListElmt \*dlist\_head(const DList \**list*);

**返回值**

返回链表的头元素。

**描述**

这是一个宏，用来返回由参数list所指定的双向链表中的头元素。

**复杂度**

****

***dlist\_tail***

DListElmt \*dlist\_tail(const DList \**list*);

**返回值**

返回链表的尾元素。

**描述**

这是一个宏，用来返回由参数list所指定的双向链表的尾部元素。

**复杂度**

****

***dlist\_is\_head***

int dlist\_is\_head(const DListElmt \**element*);

**返回值**

如果由element所指定的元素是链表头节点则返回1，否则返回0。

**描述**

这是一个宏，用来判断由参数element所指定的元素是否为链表头元素。

**复杂度**

****

***dlist\_is\_tail***

int dlist\_is\_tail(const DListElmt \**element*);

**返回值**

如果由参数element所指定的元素是链表尾元素则返回0，否则返回-1。

**描述**

这是一个宏，用来判断由参数element所指定的元素是否为链表的尾部元素。

**复杂度**



***dlist\_data***

void \*dlist\_data(const DListElmt \**element*);

**返回值**

返回由element所指定的链表元素的数据域。

**描述**

这是一个宏，用来返回由参数element所指定的链表元素的数据域。

**复杂度**

****

***dlist\_next***

DListElmt \*dlist\_next(const DListElmt \**element*);

**返回值**

返回由element所指定的元素的下一个元素。

**描述**

这是一个宏，用来返回由参数element所指定的链表元素的后继元素。

**复杂度**



***dlist\_prev***

DListElmt \*dlist\_prev(const DListElmt \**element*);

**返回值**

返回由element所指定的元素的前驱元素。

**描述**

这是一个宏，用来返回双向链表中由element所指定的元素的前驱元素。

**复杂度**

****

## 5.7 双向链表的实现与分析

回顾一下双向链表的组成：一个数据成员，一个指向下一个元素的next指针以及一个指向前一个元素的prev指针。数据结构DListElmt代表双向链表中的元素（见例子5.4）。如同你所预料的，这个结构体拥有3个我们前文描述过的成员。数据结构DList代表双向链表（见例子5.4），该结构的成员同我们前面介绍的单链表相似。

**代码示例5.4 双向链表抽象数据类型的头文件**

/\* dlist.h \*/

#ifndef DLIST\_H

#define DLIST\_H

#include <stdlib.h>

/\* Define a structure for doubly-linked list elements. \*/

typedef struct DListElmt\_

{

void \*data;

struct DListElmt\_ \*prev;

struct DListElmt\_ \*next;

}DListElmt;

/\* Define a structure for doubly-linked lists. \*/

typedef struct DList\_

{

int size;

int (\*match)(const void \*key1, const void \*key2);

void (\*destroy)(void \*data);

DListElmt \*head;

DListElmt \*tail;

}DList;

/\* Public Interface \*/

void dlist\_init(DList \*list, void (\*destroy)(void \*data));

void dlist\_destroy(DList \*list);

int dlist\_ins\_next(DList \*list, DListElmt \*element, const void \*data);

int dlist\_ins\_prev(DList \*list, DListElmt \*element, const void \*data);

int dlist\_remove(DList \*list, DListElmt \*element, void \*\*data);

#define dlist\_size(list) ((list)->size)

#define dlist\_head(list) ((list)->head)

#define dlist\_tail(list) ((list)->tail)

#define dlist\_is\_head(element) ((element)->prev == NULL ? 1 : 0)

#define dlist\_is\_tail(element) ((element)->next == NULL ? 1 : 0)

#define dlist\_data(element) ((element)->data)

#define dlist\_next(element) ((element)->next)

#define dlist\_prev(element) ((element)->prev)

#endif

### 5.7.1 dlist\_init

dlist\_init用来初始化一个双向链表以便稍后能够执行其它操作（见例子5.5）。初始化过程同单链表一样。dlist\_init的计算复杂度为，因为初始化过程中的所有步骤都能在恒定的时间内完成。

### 5.7.1 dlist\_destroy

dlist\_destroy用来销毁一个双向链表（见例子5.5）。总的来说，该操作意味着将链表中的所有元素都移除。当该函数不为NULL时将以函数指针的方式作为参数传给dlist\_init，每当一个元素被移除时都将调用它。dlist\_destroy的计算复杂度为，这里n代表链表中的元素个数。这是因为对于每个元素的移除操作都需要调用一次复杂度为的dlist\_remove函数。