链表

链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而顺序表相应的时间复杂度分别是O(logn)和O(1)。
使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。
在计算机科学中，链表作为一种基础的数据结构可以用来生成其它类型的数据结构。链表通常由一连串节点组成，每个节点包含任意的实例数据（data fields）和一或两个用来指向上一个/或下一个节点的位置的链接（"links"）。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的访问往往要在不同的排列顺序中转换。而链表是一种自我指示数据类型，因为它包含指向另一个相同类型的数据的指针（链接）。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。
链表可以在多种编程语言中实现。像Lisp和Scheme这样的语言的内建数据类型中就包含了链表的访问和操作。程序语言或面向对象语言，如C/C++和Java依靠易变工具来生成链表。

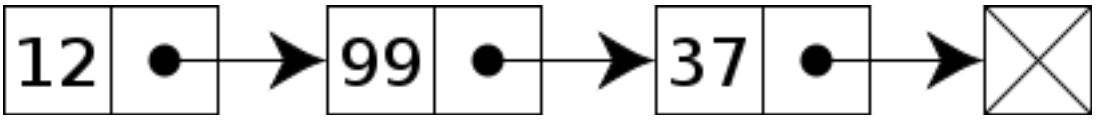
历史

链表开发于1955–56，由当时所属于兰德公司（英语：RAND Corporation）的艾伦纽维尔（Allen Newell），克里夫肖（Cliff Shaw）和赫伯特西蒙（Herbert Simon）在他们编写的信息处理语言（IPL）中做为原始数据类型所编写。IPL被作者们用来开发几种早期的人工智能程序，包括逻辑推理机，通用问题解算器和一个计算机象棋程序。

结构

单向链表

链表中最简单的一种是单向链表，它包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个节点，而最后一个节点则指向一个空值。

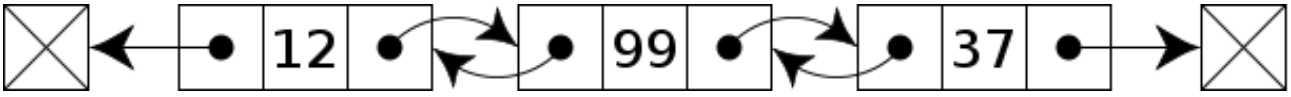


一个单向链表包含两个值: 当前节点的值和一个指向下一个节点的链接

一个单向链表的节点被分成两个部分。第一个部分保存或者显示关于节点的信息，第二个部分存储下一个节点的地址。单向链表只可向一个方向遍历。
链表最基本的结构是在每个节点保存数据和到下一个节点的地址，在最后一个节点保存一个特殊的结束标记，另外在一个固定的位置保存指向第一个节点的指针，有的时候也会同时储存指向最后一个节点的指针。一般查找一个节点的时候需要从第一个节点开始每次访问下一个节点，一直访问到需要的位置。但是也可以提前把一个节点的位置另外保存起来，然后直接访问。当然如果只是访问数据就没必要了，不如在链表上储存指向实际数据的指针。这样一般是为了访问链表中的下一个或者前一个（需要储存反向的指针，见下面的双向链表）节点。
相对于下面的双向链表，这种普通的，每个节点只有一个指针的链表也叫**单向链表**，或者**单链表**，通常用在每次都只会按顺序遍历这个链表的时候（例如图的邻接表，通常都是按固定顺序访问的）。
链表也有很多种不同的变化：

双向链表

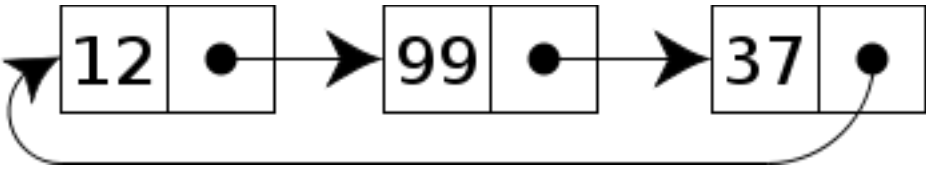
一种更复杂的链表是“双向链表”或“双面链表”。每个节点有两个连接：一个指向前一个节点，（当此“连接”为第一个“连接”时，指向空值或者空列表）；而另一个指向下一个节点，（当此“连接”为最后一个“连接”时，指向空值或者空列表）



一个双向链表有三个整数值: 数值, 向后的节点链接, 向前的节点链接
在一些低级语言中, XOR-linking 提供一种在双向链表中通过用一个词来表示两个链接（前后），我们通常不提倡这种做法。
双向链表也叫**双链表**。**双向链表**中不仅有指向后一个节点的指针，还有指向前一个节点的指针。这样可以从任何一个节点访问前一个节点，当然也可以访问后一个节点，以至整个链表。一般是在需要大批量的另外储存数据在链表中的位置的时候用。双向链表也可以配合下面的其他链表的扩展使用。
由于另外储存了指向链表内容的指针，并且可能会修改相邻的节点，有的时候第一个节点可能会被删除或者在之前添加一个新的节点。这时候就要修改指向首个节点的指针。有一种方便的可以消除这种特殊情况的方法是在最后一个节点之后、第一个节点之前储存一个永远不会被删除或者移动的虚拟节点，形成一个下面说的循环链表。这个虚拟节点之后的节点就是真正的第一个节点。这种情况通常可以用这个虚拟节点直接表示这个链表，对于把链表单独的存在数组里的情况，也可以直接用这个数组表示链表并用第0个或者第-1个（如果编译器支持）节点固定的表示这个虚拟节点。

循环链表

在一个 **循环链表**中, 首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。要转换一个循环链表，你开始于任意一个节点然后沿着列表的任一方向直到返回开始的节点。再来看另一种方法，循环链表可以被视为“无头无尾”。这种列表很利于节约数据存储缓存， 假定你在一个列表中有一个对象并且希望所有其他对象迭代在一个非特殊的排列下。
指向整个列表的指针可以被称作访问指针。



用单向链表构建的循环链表

循环链表中第一个节点之前就是最后一个节点，反之亦然。循环链表的**无边界**使得在这样的链表上设计算法会比普通链表更加容易。对于新加入的节点应该是在第一个节点之前还是最后一个节点之后可以根据实际要求灵活处理，区别不大(详见下面实例代码)。当然，如果只会在最后插入数据（或者只会在之前），处理也是很容易的。
另外有一种模拟的循环链表，就是在访问到最后一个节点之后的时候，手工的跳转到第一个节点。访问到第一个节点之前的时候也一样。这样也可以实现循环链表的功能，在直接用循环链表比较麻烦或者可能会出现问题的时候可以用。

块状链表

块状链表本身是一个链表，但是链表储存的并不是一般的数据，而是由这些数据组成的顺序表。每一个块状链表的节点，也就是顺序表，可以被叫做一个**块**。
块状链表通过使用可变的顺序表的长度和特殊的插入、删除方式，可以在达到$O(\sqrt{n})$的复杂度。块状链表另一个特点是相对于普通链表来说节省内存，因为不用保存指向每一个数据节点的指针。

其它扩展

根据情况，也可以自己设计链表的其它扩展。但是一般不会在边上附加数据，因为链表的点和边基本上是一一对应的（除了第一个或者最后一个节点，但是也不会产生特殊情况）。不过有一个特例是如果链表支持在链表的一段中把前和后指针反向，反向标记加在边上可能会更方便。
对于非线性的链表，可以参见相关的其他数据结构，例如树、图。另外有一种基于多个线性链表的数据结构：跳表，插入、删除和查找等基本操作的速度可以达到O(nlogn)，和平衡树一样。

存储结构

链表中的节点不需要以特定的方式存储，但是集中存储也是可以的，主要分下面这几种具体的存储方法：

共用存储空间

链表的节点和其它的数据共用存储空间，优点是可以存储无限多的内容（不过要处理器支持这个大小，并且存储空间足够的情况下），不需要提前分配内存；缺点是由于内容分散，有时候可能不方便调试。

独立存储空间

一个链表或者多个链表使用独立的存储空间，一般用数组或者类似结构实现，优点是可以自动获得一个附加数据：唯一的编号，并且方便调试；缺点是不能动态的分配内存。当然，另外的在上面加一层块状链表用来分配内存也是可以的，这样就解决了这个问题。这种方法有时候被叫做**数组模拟链表**，但是事实上只是用表示在数组中的位置的下标索引代替了指向内存地址的指针，这种下标索引其实也是逻辑上的指针，整个结构还是链表，并不算是被模拟的（但是可以说成是**用数组实现的链表**）。

链表的应用

链表用来构建许多其它数据结构，如堆栈，队列和他们的派生。

节点的数据域也可以成为另一个链表。通过这种手段，我们可以用列表来构建许多链性数据结构；这个实例产生于Lisp编程语言，在Lisp中链表是初级数据结构，并且现在成为了常见的基础编程模式。 有时候，链表用来生成联合数组，在这种情况下我们称之为联合数列。这种情况下用链表会优于其它数据结构，如自平对分查找树（self-balancing binary search trees）甚至是一些小的数据集合。不管怎样，一些时候一个链表在这样一个树中创建一个节点子集，并且以此来更有效率地转换这个集合。

C代码实例

范例代码是一个ADT（抽象数据类型）双向环形链表的基本操作部分的实例（未包含线程安全机制），全部遵从ANSI C标准，代码遵从GPL版权许可。

接口声明

```
#ifndef LLIST_H
#define LLIST_H

typedef void node_proc_fun_t(void*);
typedef int node_comp_fun_t(const void*, const void*);
```

```
typedef void LLIST_T;

LLIST_T *llist_new(int elmsize);
int llist_delete(LLIST_T *ptr);

int llist_node_append(LLIST_T *ptr, const void *datap);
int llist_node_prepend(LLIST_T *ptr, const void *datap);

int llist_travel(LLIST_T *ptr, node_proc_fun_t *proc);

void llist_node_delete(LLIST_T *ptr, node_comp_fun_t *comp, const void *key);
void *llist_node_find(LLIST_T *ptr, node_comp_fun_t *comp, const void *key);

#endif
```

接口实现

类型确定

```
struct node_st {
        void *datap;
        struct node_st *next, *prev;
};

struct llist_st {
        struct node_st head;
        int elmsize;
        int elmnr;
};
```

初始化和销毁

```
LLIST_T*
llist_new(int elmsize)
{
        struct llist_st *newlist;
        newlist = malloc(sizeof(struct llist_st));
        if (newlist == NULL) {
                return NULL;
        }
```

```
        newlist->head.datap = NULL;
        newlist->head.next = &newlist->head;
        newlist->head.prev = &newlist->head;

        newlist->elmsize = elmsize;

        return (void*)newlist;
}

int
llist_delete(LLIST_T *ptr)
{
        struct llist_st *me=ptr;
        struct node_st *curr, *save;

        for (curr=me->head.next ; curr!=&me->head ; curr=save) {
                save=curr->next;
                free(curr->datap);
                free(curr);
        }

        free(me);

        return 0;
}
```

节点插入

```
int
llist_node_append(LLIST_T *ptr, const void *datap)
{
        struct llist_st *me=ptr;
        struct node_st *newnodep;

        newnodep = malloc(sizeof(struct node_st));
        if (newnodep == NULL) {
                return -1;
        }
        newnodep->datap = malloc(me->elmsize);
```

```
        if (newnodep->datap == NULL) {
                free(newnodep);
                return -1;
        }

        memcpy(newnodep->datap, datap, me->elmsize);

        me->head.prev->next = newnodep;
        newnodep->prev = me->head.prev;
        me->head.prev = newnodep;
        newnodep->next = &me->head;

        return 0;
}

int
llist_node_prepend(LLIST_T *ptr, const void *datap){
    struct llist_st *me=ptr;
    struct node_st *newnodep;

    newnodep = malloc(sizeof(struct node_st));
    if (newnodep == NULL) {
        return -1;
    }
    newnodep->datap = malloc(me->elmsize);
    if (newnodep->datap == NULL) {
        free(newnodep);
        return -1;
    }

    memcpy(newnodep->datap, datap, me->elmsize);

    me->head.next->prev = newnodep;
    newnodep->next = me->head.next;
    me->head.next = newnodep;
    newnodep->prev = &me->head;

    return 0;
}
```

遍历

```
int
llist_travel(LLIST_T *ptr, node_proc_fun_t *proc){
    struct llist_st *me=ptr;
    struct node_st *curr;

    for (curr=me->head.next ; curr!=&me->head ; curr=curr->next)
        proc(curr->datap);/* proc(something you like)*/

    return 0;
}
```

删除和查找

```
void
llist_node_delete(LLIST_T *ptr, node_comp_fun_t *comp, const void *key){
    struct llist_st *me=ptr;
    struct node_st *curr;

    for (curr=me->head.next;curr!=&me->head;curr=curr->next) {
        if ( (*comp)(curr->datap, key) == 0 ) {
            struct node_st *_next,*_prev;
            _prev = curr->prev,_next = curr->next;
            _prev->next = _next,_next->prev = _prev;

            free(curr->datap);
            free(curr);
            break;
        }
    }
    return;
}

void*
llist_node_find(LLIST_T *ptr, node_comp_fun_t *comp, const void *key)
{
        struct llist_st *me=ptr;
        struct node_st *curr;
```

```
        for (curr=me->head.next;curr!=&me->head;curr=curr->next) {
                if ( (*comp)(curr->datap, key) == 0 ) {
                        return curr->datap;
                }
        }
        return NULL;
}
```

C宏实例

以下代码摘自Linux内核2.6.21.5源码(部分)，展示了链表的另一种实现思路，未采用ANSI C标准，采用GNU C标准，遵从GPL版权许可。

```
struct list_head {
        struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;
}

static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

static inline void list_add(struct list_head *new, struct list_head *head)
{
```

```
        __list_add(new, head, head->next);
}

static inline void __list_del(struct list_head * prev, struct list_head * next)
{
        next->prev = prev;
        prev->next = next;
}


static inline void list_del(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = NULL;
        entry->prev = NULL;
}

#define __list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_for_each_entry(pos, head, member)                          \
        for (pos = list_entry((head)->next, typeof(*pos), member);      \
             prefetch(pos->member.next), &pos->member != (head);        \
             pos = list_entry(pos->member.next, typeof(*pos), member))
```