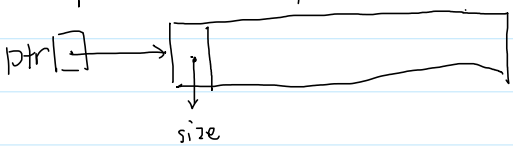
动态内存分配

2022年3月22日 9:50

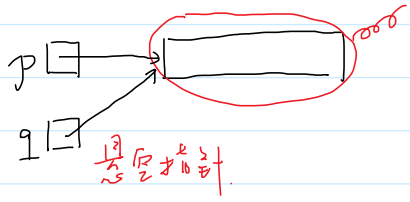
void free (void *ptr);

Q. ptr只是指向申请内存块的首地址，那free函数如何知道该释放多大的内存空间？



(2) 使用free函数虽然可以避免内存泄漏，但是也会引入一个新问题：悬空指针.

```
p = malloc( -- );
q = p;
free(p);
strcpy (q, "abc");
```



悬空指针是非常容易出现的，释放指针p，会导致所有指向相同内存块的指针都"悬空".

链表

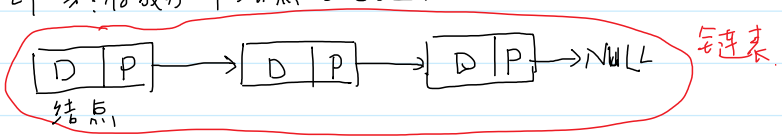
2022年3月22日 10:01

链表：用一条“链”将所有结点串联起来

结点：

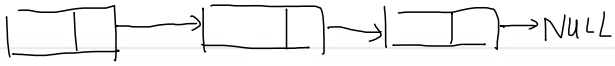
数据域

指针域：存放另一个结点的地址

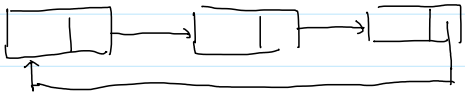


1. 链表分类

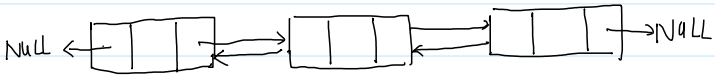
单向链表：



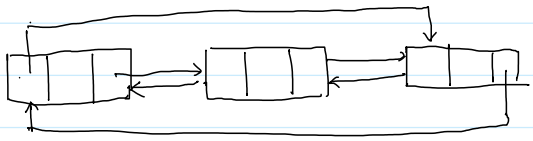
单向循环链表：



双向链表：



双向循环链表：



循环链表在实际生产中用得比较少，循环链表在处理环状数据结构时特别有用（约瑟夫环）

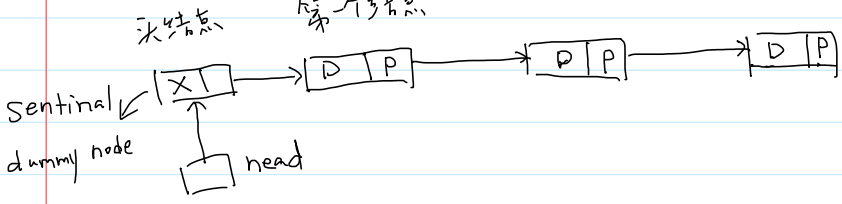
```c
typedef struct node_s {
    int val;
    struct node_s* next;
} Node;

Node* add_to_list(Node* list, int val);

int main(void) {
    Node* list = NULL;
    list = add_to_list(list, 1);
    list = add_to_list(list, 2);
    list = add_to_list(list, 3);
    list = add_to_list(list, 4);

    return 0;
}

Node* add_to_list(Node* list, int val) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Error: malloc failed in add_to_list.\n");
        exit(1);
    }
    // 头插法
    newNode->val = val;
    newNode->next = list;
    return newNode;
}
```



二级指针

2022年3月22日 10:54

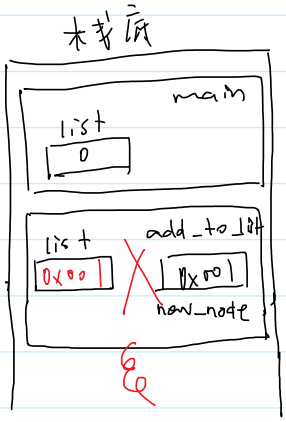
```c
typedef struct node_s {
    int val;
    struct node_s* next;
} Node;

void add_to_list(Node* list, int val);

int main(void) {
    Node* list = NULL;
    add_to_list(list, 1);
    add_to_list(list, 2);
    add_to_list(list, 3);
    add_to_list(list, 4);

    return 0;
}

void add_to_list(Node* list, int val) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Error: malloc failed in add_to_list.\n");
        exit(1);
    }
    // 头插法
    newNode->val = val;
    newNode->next = list;
    list = newNode;
}
```



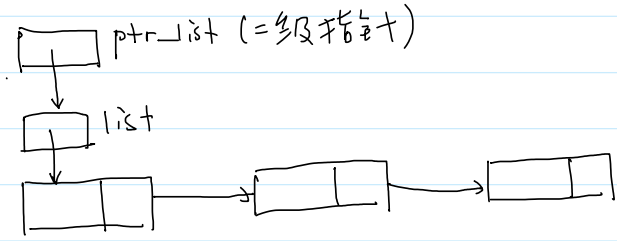

```c
typedef struct node_s {
    int val;
    struct node_s* next;
} Node;

void add_to_list(Node** ptr_list, int val);

int main(void) {
    Node* list = NULL;
    add_to_list(&list, 1);
    add_to_list(&list, 2);
    add_to_list(&list, 3);
    add_to_list(&list, 4);

    return 0;
}

void add_to_list(Node** ptr_list, int val) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Error: malloc failed in add_to_list.\n");
        exit(1);
    }
    // 头插法
    newNode->val = val;
    newNode->next = *ptr_list;
    *ptr_list = newNode;
}
```

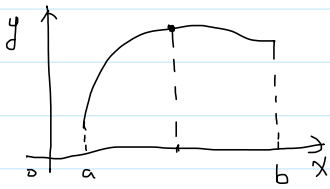


函数指针

2022年3月22日 11:07

指向函数的指针. 函数也会有地址.

练习, 对任意一个函数返回值为double, 参数类型double, 给定两点a,b, 求 $f(\frac{a+b}{2})$.




```c
#define PI 3.1415926

double average(double (*f) (double), double a, double b);

int main(void) {
    double avg = average(sin, 0, PI);
    printf("%lf", avg);

    return 0;
}

//double average(double (*f) (double), double a, double b) {
//  return (*f)((a + b) / 2);
//}
                    callback 回调函数
double average(double f (double), double a, double b) {
    return f((a + b) / 2);
}
```

(2) qsort

↳ quick 快速排

```
Defined in header <stdlib.h>
void qsort( void *ptr, size_t count, size_t size,
            int (*comp)(const void *, const void *) );          (1)
```

可以对任意类型的数组进行排列, 不管元素类型是什么。

Q 排序的前提? 排序的目的?

比较 查找

ptr → 指向要排序的数组

count → 数组中元素的个数

size → 元素的大小

comp → 比较函数 如果第一个参数大于第二个参数返回正值, 如果第一个参数等于第二个参数返回零, 如果第一个参数小于第二个参数返回负值.

```c
typedef struct student_s {
    int number;
    char name[25];
    int chinese;
    int math;
    int english;
} Student;

int compare(const void* p1, const void* p2);

int main(void) {
    Student students[5] = { {1, "liuyifei", 100, 100, 100}, {2, "wangyuyan", 99, 100, 100},
        {3, "zhaolinger", 100, 99, 100}, {4, "xiaolongnv", 100, 100, 99}, {5, "baixiuzhu", 100, 100, 99}};
    qsort(students, SIZE(students), sizeof(Student), compare);

    return 0;
}
```

```c
// 比较规则：总分从高到底，语文成绩(高-->低)，数学成绩(高-->低)，英语成绩(高-->低)，姓名(字典顺序从小到大进行比较)
int compare(const void* p1, const void* p2) {
    Student* s1 = (Student*)p1;
    Student* s2 = (Student*)p2;
    int total1 = s1->chinese + s1->english + s1->math;
    int total2 = s2->chinese + s2->english + s2->math;
    if (total1 != total2) {
        return total2 - total1;
    }
    if (s1->chinese != s2->chinese) {
        return s2->chinese - s1->chinese;
    }
    if (s1->math != s2->math) {
        return s2->math - s1->math;
    }
    if (s1->english != s2->english) {
        return s2->english - s1->english;
    }
    return strcmp(s1->name, s2->name);
}
```

数据结构

2022年3月22日 14:40

数组和链表是构建其它更复杂数据结构的基础.

#1. 链表

#2. 栈、队列

#3. 哈希表

#4. 红黑树(BST, BinarySearchTree)

算法:

排序算法

二分查找

分治(divide & conquer)

链表

2022年3月22日 14:52

1. 链表的基本操作

单链表：

增（在某个结点后面添加） O(1)

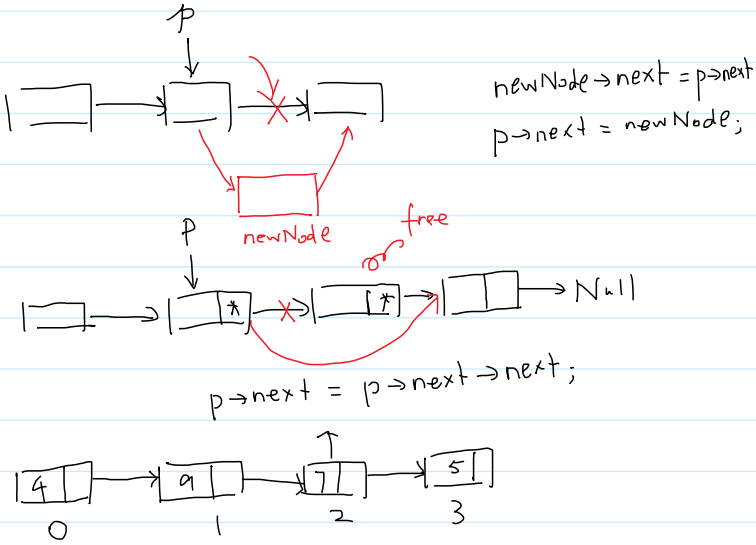
删（删除某个结点后面的结点） O(1)

查：① 根据索引查找值 O(n)

② 查找与特定值相等的结点

a. 大小有序 O(n)

b. 大小无序 O(n)



双向链表，除了单链表的基础操作之外，还有一些额外的基本操作

增（在某个结点前面添加） （O(1)，单链表：O(n)）

删（删除某个结点） （O(1)，单链表：O(n)）

查：① 查找前驱结点 （O(1)，单链表：O(n)）

② 根据索引查找值 （O(n)，平均遍历 $\frac{n}{4}$ 个元素，单链表平均遍历 $\frac{n}{2}$）

③ 查找与特定值相等的结点

a. 大小有序 （O(n)，记录上一次查找结点，平均遍历 $\frac{n}{4}$，单链表平均遍历 $\frac{n}{2}$）

b. 大小无序 （O(n)）

总结：虽然双向链表占用更多的内存空间，但是在很多操作上面优于单链表

所以在实际生产中更倾向使用双向链表

思想：用空间换取时间

缓存其实就是用空间换取时间

缓存淘汰策略：

① FIFO （First In First Out）

② LFU （Least Frequently Used）

③ LRU （Least Recently Used）

2. 如何基于LRU，用链表实现LRU算法

添加（尾结点是最近一直没有访问的数据） 时间复杂度：O(n)

a. 元素存在

删除元素所在结点

将一个结点在前面添加

删除元素所在结点，
在第一个结点前面添加

b. 元素不存在

① 缓存满了

删除尾结点，

在第一个结点前面添加

② 缓存未满

在第一个结点前面添加

改进方法：哈希表+链表（拓展知识）