Struct and Linked-List

Weng Kai

结构

声明结构类型

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    struct date {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 07;
    today.day = 31;
    today.year = 2014;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);

    return 0;
}
```

初学者最常见的
错误：漏了这个分号！

在函数内/外?

```c
#include <stdio.h>

struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today;

    today.month = 07;
    today.day = 31;
    today.year = 2014;

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);

    return 0;
}
```

- 和本地变量一样，在函数内部声明的结构类型只能在函数内部使用
- 所以通常在函数外部声明结构类型，这样就可以被多个函数所使用了

声明结构的形式

```
struct point {
    int x;
    int y;
};
```

struct point p1, p2;

p1 和 p2 都是point
里面有x和y的值

```
struct {
    int x;
    int y;
} p1, p2;
```

p1 和 p2都是一种无名结构，里面有x和y

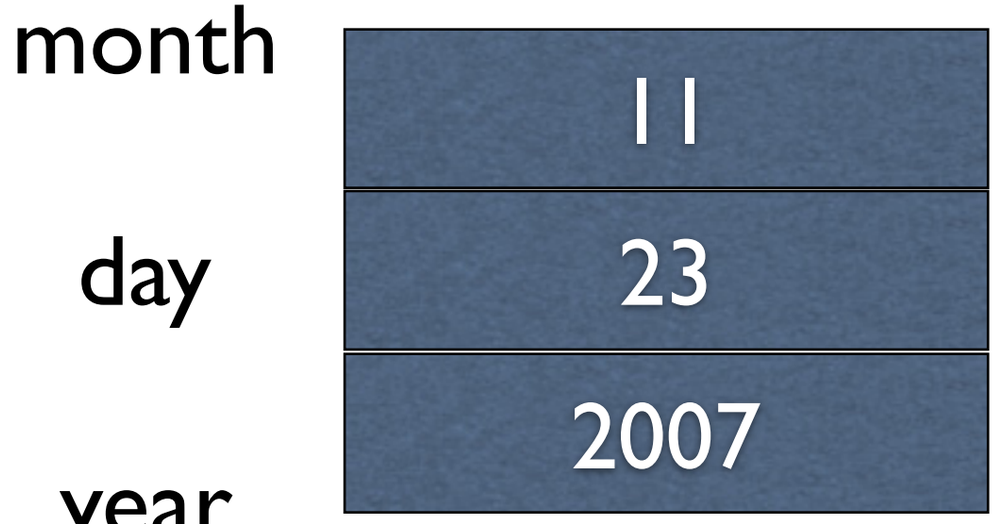
```
struct  point {
    int x;
    int y;
} p1, p2;
```

p1和p2都是point
里面有x和y的值t

对于第一和第三种形式，都声明了结构point。但是第二种形式没有声明point，只定义了两个变量

结构变量

```
struct date today;
today.month=06;
today.day=19;
today.year=2005;
```



结构的初始化

```c
#include <stdio.h>

struct date {
    int month;
    int day;
    int year;
};

int main(int argc, char const *argv[])
{
    struct date today = {07,31,2014};
    struct date thismonth = {.month=7, .year=2014};

    printf("Today's date is %i-%i-%i.\n",
        today.year,today.month,today.day);
    printf("This month is %i-%i-%i.\n",
        thismonth.year,thismonth.month,thismonth.day);

    return 0;
}
```

- struct date today = {7,2,2005};
- struct date today = { .day=12, .year=2007};

结构成员

- 结构和数组有点像
- 数组用[]运算符和下标访问其成员
  - a[0] = 10;
- 结构用.运算符和名字访问其成员
  - today.day
  - student.firstName
  - p1.x
  - p1.y

结构运算

- 要访问整个结构，直接用结构变量的名字
- 对于整个结构，可以做赋值、取地址，也可以传递给函数参数

- p1 = (struct point){5, 10}; // 相当于p1.x = 5; p1.y = 10;
- p1 = p2; // 相当于p1.x = p2.x; p1.y = p2.y;

数组无法做这两种运算！

复合字面量

- today = (struct date) {9,25,2004};

- today = (struct date) {.month=9, .day=25, .year=2004};

结构指针

- 和数组不同，结构变量的名字并不是结构变量的地址，必须使用&运算符

- struct date *pDate = &today;

结构与函数

结构作为函数参数

```
int numberOfDays(struct date d)
```

- 整个结构可以作为参数的值传入函数
- 这时候是在函数内新建一个结构变量，并复制调用者的结构的值
- 也可以返回一个结构
- 这与数组完全不同

输入结构

- 没有直接的方式可以一次scanf一个结构
- 如果我们打算写一个函数来读入结构
  - —>
- 但是读入的结构如何送回来呢?
- 记住C在函数调用时是传值的
  - 所以函数中的p与main中的y是不同的
  - 在函数读入了p的数值之后，没有任何东西回到main，所以y还是 {0, 0}

```
#include <stdio.h>

struct point  {
     int x;
     int y;  };

void getStruct(struct
point);
void output(struct point);
void main( )    {
     struct point y = {0, 0};
     getStruct(y);
     output(y);  }

void getStruct(struct point
p)   {
     scanf("%d", &p.x);
     scanf("%d", &p.y);
     printf("%d, %d", p.x, p.y);
}

void output(struct point p)
{
     printf("%d, %d", p.x, p.y);
}
```

解决的方案

- 之前的方案，把一个结构传入了函数，然后在函数中操作，但是没有返回回去
  - 问题在于传入函数的是外面那个结构的克隆体，而不是指针
    - 传入结构和传入数组是不同的

```
struct point inputPoint( )
{
        struct point temp;
        scanf("%d", &temp.x);
        scanf("%d", &temp.y);
        return temp;
}
```

```
void main( )
{
        struct point y = {0, 0};
        y = inputPoint( );
        output(y);
}
```

- 在这个输入函数中，完全可以创建一个临时的结构变量，然后把这个结构返回给调用者

也可以把y的地址传给函数，函数的参数类型是指向一个结构的指针。不过那样的话，访问结构的成员的方式需要做出调整。

结构指针作为参数

- K & R 说过 (p. 131)
  - “If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure”

指向结构的指针

```
struct date {
    int month;
    int day;
    int year;
} myday;

struct date *p = &myday;

(*p).month = 12;
p->month = 12;
```

- 用->表示指针所指的结构变量中的成员

结构指针参数

```
void main( )
{
    struct point x = {0, 0};
    inputPoint(&y);
    output(y);
}
                                    struct
point *p)
{                    • const struct point *p
    scanf(“%d”, &(p->x));
    scanf(“%d”, &(p->y));
    return p;
}
```

- 好处是传入传出只是一个指针的大小
- 如果需要保护传入的结构不被函数修改
- 返回传入的指针是一种套路

结构数组

```
struct date dates[100];

struct date dates[] = {
{4,5,2005},{2,4,2005}};
```

结构中的结构

```
struct dateAndTime {
    struct date sdate;

    struct time stime;

};
```

嵌套的结构

```
struct point {
      int x;
      int y;
 };
struct rectangle {
      struct point pt1;
      struct point pt2;
};
```

如果有变量

struct rectangle r;

就可以有：

r.pt1.x、r.pt1.y,

r.pt2.x 和 r.pt2.y

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

如果有变量定义：

```
struct rectangle r, *rp;
rp = &r;
```

那么下面的四种形式是等价的：

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

但是没有rp->pt1->x （因为pt1不是指针）

结构中的结构的数组

```
#include <stdio.h>

struct point{
      int x;
      int y;
};

struct rectangle {
      struct point p1;
      struct point p2;
};

void printRect(struct rectangle r)
{
      printf("<%d, %d> to <%d, %d>\n", r.p1.x, r.p1.y, r.p2.x, r.p2.y);
}

int main(int argc, char const *argv[])
{
     int i;
     struct rectangle rects[ ] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}; // 2 rectangles
     for(i=0;i<2;i++) printRect(rects[i]);
}
```

resizable array

Resizable Array

- Think about a set of functions that provide a mechanism of resizable array of int.
  - Growable
  - Get the current size
  - Access to the elements

the Interface

- Array array_create(int init_size);
- void array_free(Array *a);
- int array_size(const Array *a);
- int* array_at(Array *a, int index);
- void array_inflate(Array *a, int more_size);

the Array

```
typedef struct {
        int *array;

        int size;

} Array;
```

Why struct not struct *?

array_create()

```
Array array_create(int init_size) {

        Array a;

        a.array = (int*)malloc(sizeof(int)*init_size);

        a.size = init_size;

        return a;

}
```

Why Array not Array *?

array_free()

```
void array_free(Array *a) {

        free(a->array);

        a->array = NULL;

        a->size = 0;

}
```

array_size()

```
int array_size(const Array *a) {
    return a->size;
}
```

Why not take the member directly?

array_at()

```
int* array_at(Array *a, int index) {

        if ( index >= a->size ) {
            array_inflate(a, index-a->size);

        }

        return &(a->array[index]);

}
```

Why int* not int?

use array_at()

```
Array a = array_create(10);

*(array_at(&a, 5)) = 6;

*(array_at(&a, 10)) = *(array_at(&a, 5));
```

will it be better

- to have two access functions:
  - array_get(), and
  - array_set()

use get() and set()

```
Array a = array_create(10);

array_set(&a, 5, 6);

array_set(&a, 10, array_get(&a, 5));
```

memory in block

```
int* array_at(Array *a, int index) {

        if ( index >= a->size ) {
            array_inflate(a, (index/
            BLOCK_SIZE+1)*BLOCK_SIZE-a->size);

        }

        return &(a->array[index]);

    }
```

array_inflate()

```
void array_inflate(Array *a, int more_size) {

    int* p = (int*)malloc(sizeof(int)*(a->size+more_size));

    for ( int i=0; i<a->size; i++ ) p[i] = a->array[i];

    free(a->array);

    a->array = p; a->size = a->size+more_size;

}
```

array_inflate()

```
void array_inflate(Array *a, int more_size) {

    int* p = (int*)malloc(sizeof(int)*(a->size+more_size));

    memcpy((void*) p, (void*) a->array, a->size*sizeof(int));

    free(a->array);

    a->array = p; a->size = a->size+more_size;

}
```

why not take the whole array

```
int* array_get(Array* a) {
    return a->array;
}
```

lack of protection for both user and developer

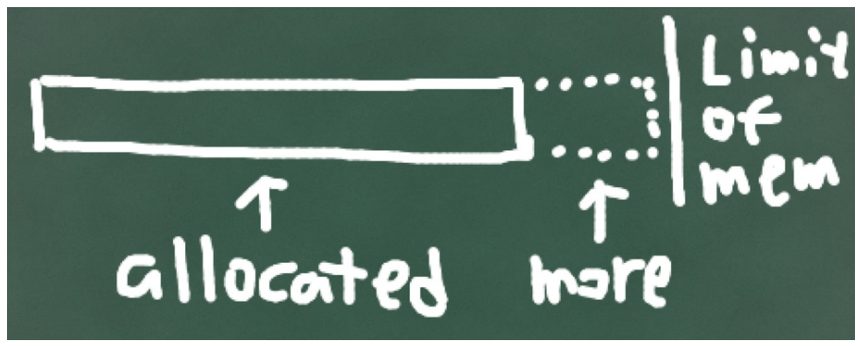
access functions

- the use of access functions seems not so elegant
  - Use operator overload in C++
  - Design specific functions for specified application
    - Do not treat it as an array

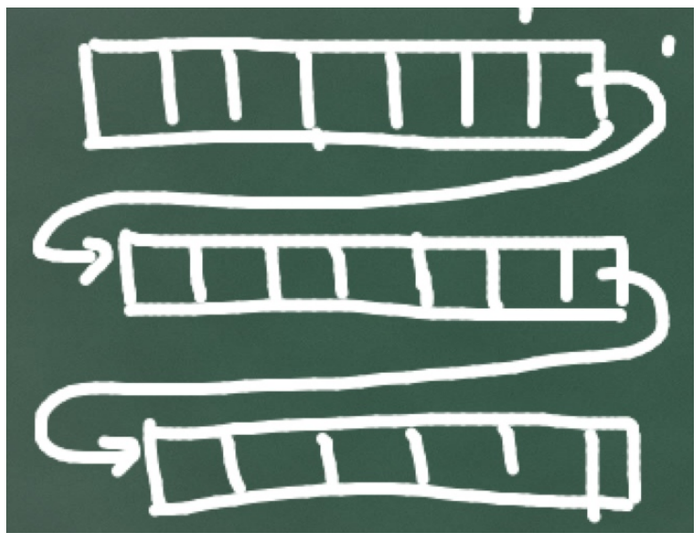
linked-array

issues

- Allocate new memory each time it inflates is an easy and clean way. But
  - It takes time to copy, and
  - may fail in memory restricted situation



linked blocks



- No copy

the Array

```
typedef struct _array{

        int *array;

        int size;

        struct _array* next;

} Array;
```

_array

use a fixed block size, but keep the variable to make it more flexible

array_create()

```
Array array_create() {

        Array a;

        a.array = (int*)malloc(sizeof(int)*BLOCK_SIZE);

        a.size = BLOCK_SIZE;

        a.next = 0;

        return a;

}
```

array_free()

```
void array_free(Array *a) {

        free(a->array);

        a->size = 0;

        if ( a->next ) {

            array_free(a->next);

            free(a->next);

        }

}
```

array_size()

```
int array_size(const Array *a) {

        if ( !a->next )

            return a->size;

        else

            return a->size+array_size(a->next);

}
```

array_at()

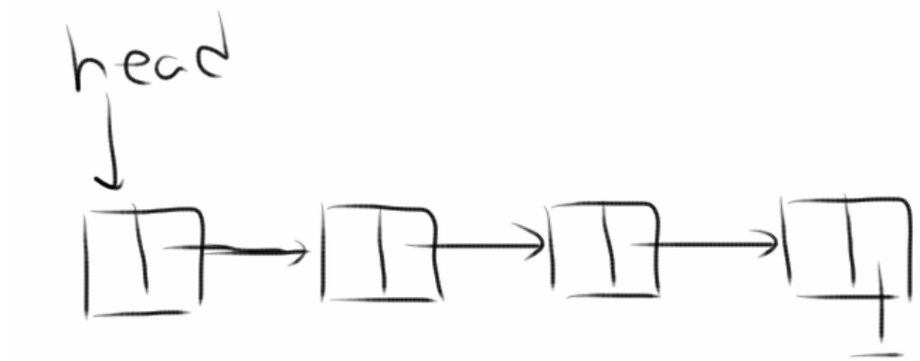
```
int* array_at(Array *a, int index) {

        if ( index < a->size ) {

            return &(a->array[index]);

        } else {

            ...

        }

}
```

array_inflate()

```
void array_inflate(Array *a) {

    // find the last block

    // allocate a new block

    // link!

}
```

Linked-List

Basic Idea



- node -- 结点

data structure

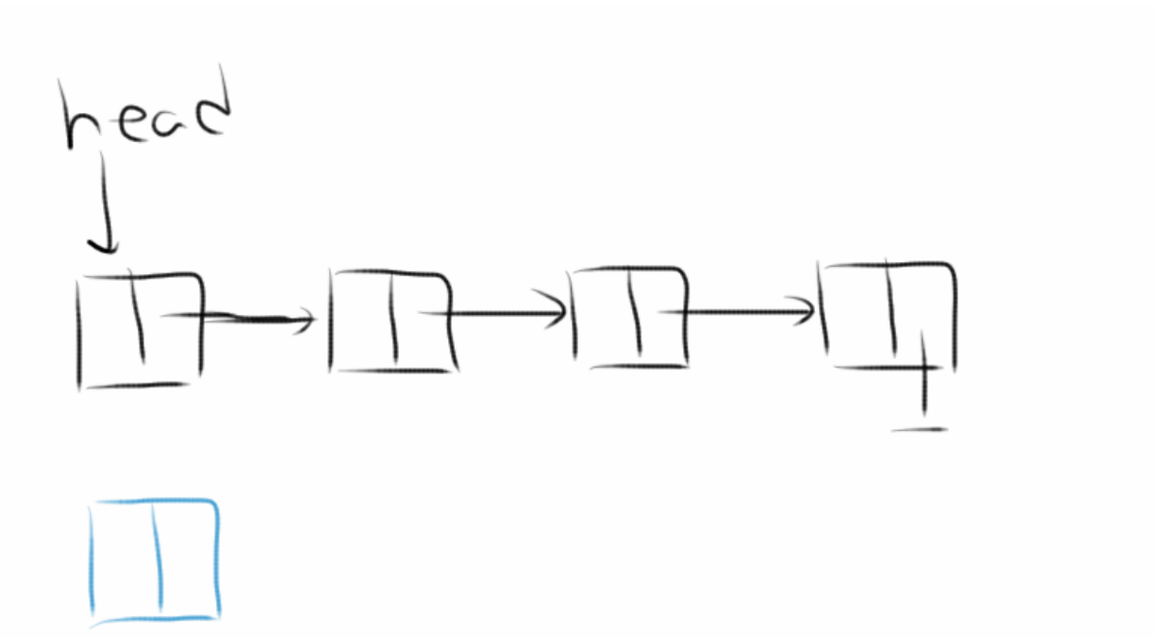
```
typedef struct _node {
    int value;

    struct _node* next;

} Node;
```

basic operation

- insert head
- traversal/search
- append tail
- remove
- clear all

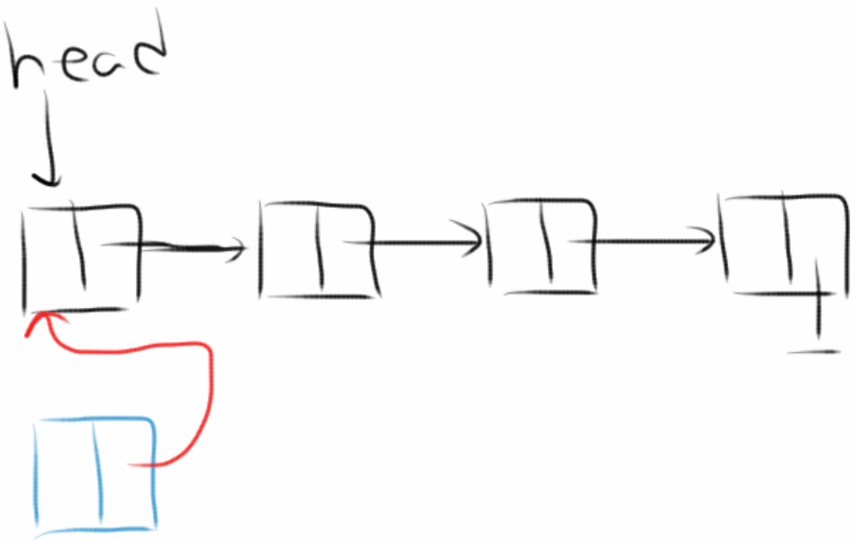
insert head 1

- Create a new node.



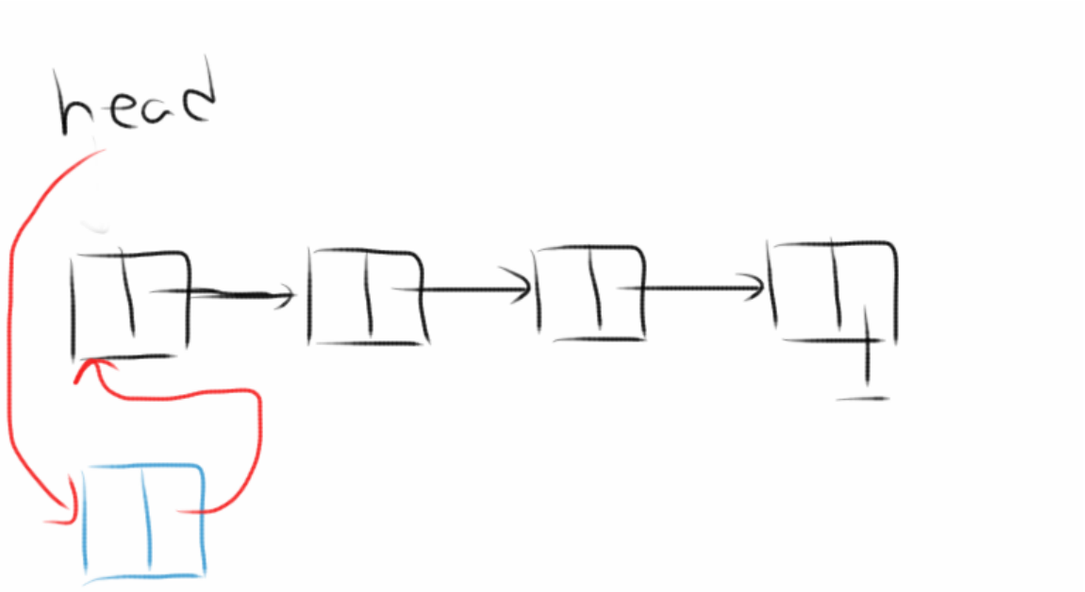
insert head 1l

- Make the new next point to the head.



insert head III

- Point the head to the new node.



insert head IV

```
Node* n =
(Node*)malloc(sizeof(Node));
n->value = i;
n->next = head;
head = n;
```

- Any boundary condition?

make it a func?

- void add_head(Node* head, int i);
- void add_head(Node** pHead, int i);
- Node* add_head(Node* head, int i);
- ??

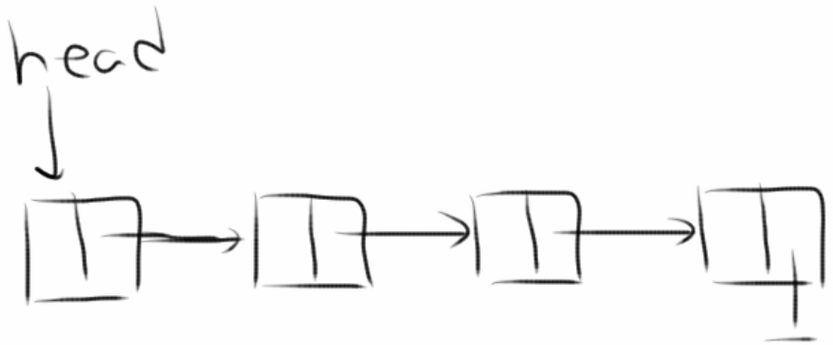
struct List

```
typedef struct {

        Node* head;

} List;


void add_head(List* list, int i);
```

traversal

```
for ( p = head; p; p=p->next ) {
}
```

- follow the next pointers



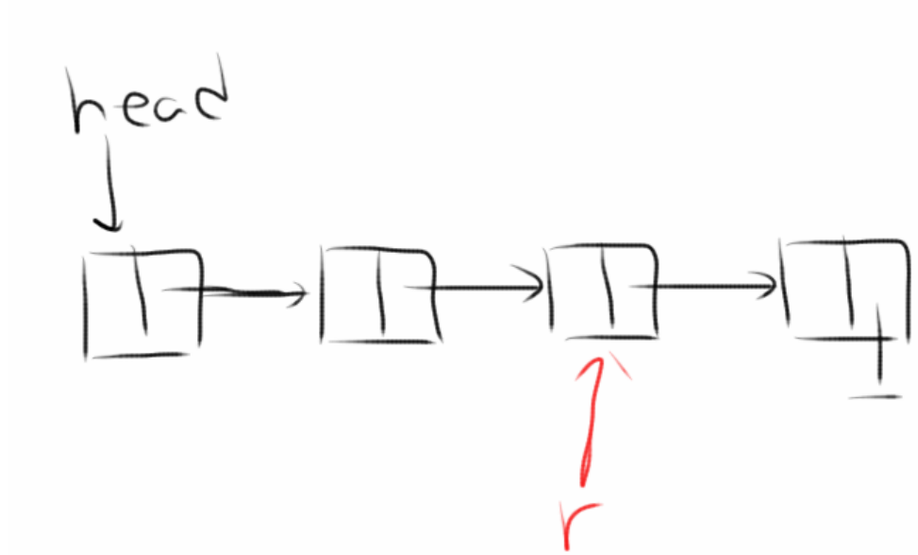
search

- find the value and return the pointer

```
ret = 0;
for ( p = head; p; p=p->next ) {
 if ( p->value == i ) {
  ret = p;break;
 }
}
```

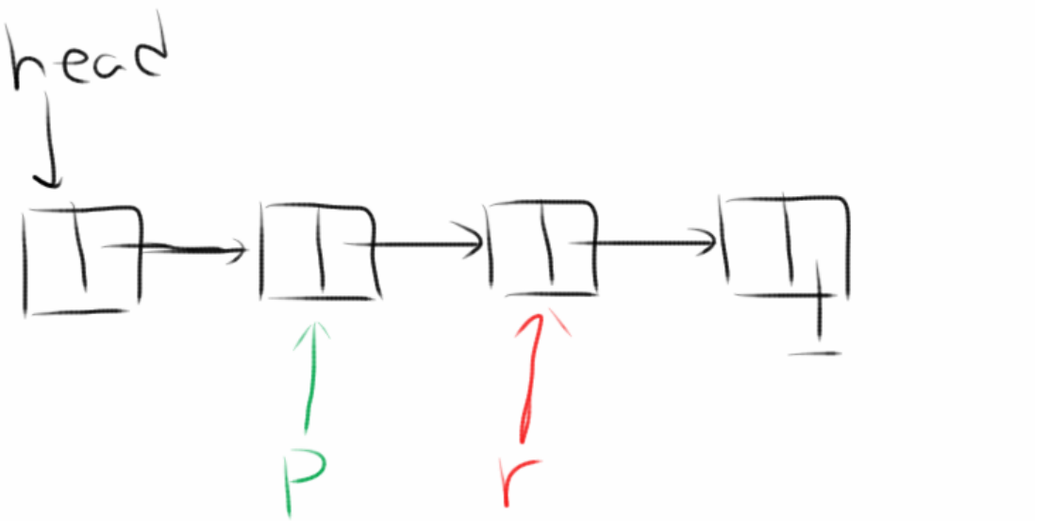
remove by a pointer I

- remove(Node* r);



remove by a pointer II

- find it’s previous node p

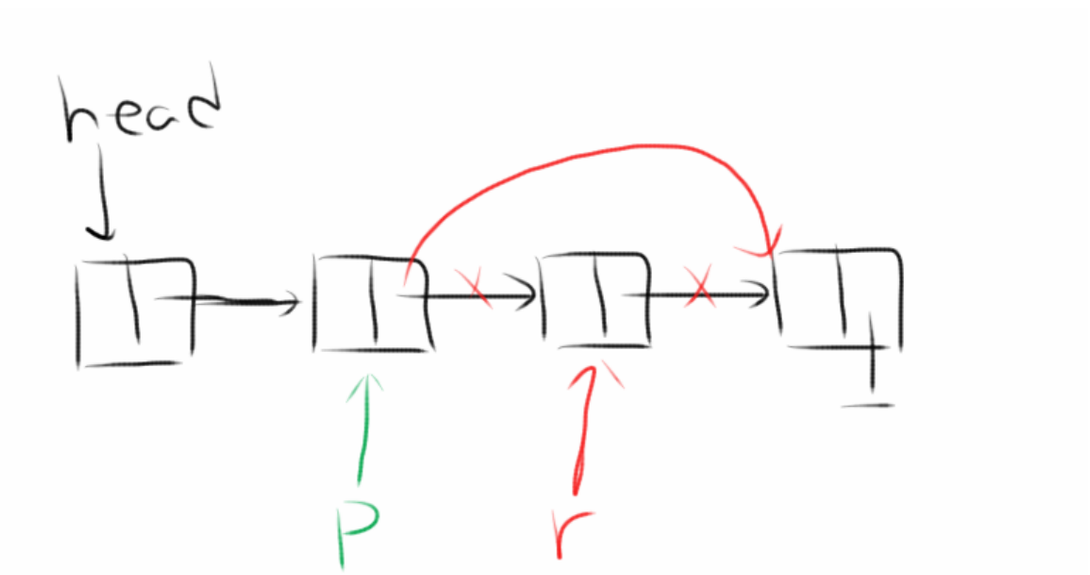


remove by a pointer III

```
for ( p = head; p; p=p-
>next ) {
  if ( p->next == r ) {
    }
}
```

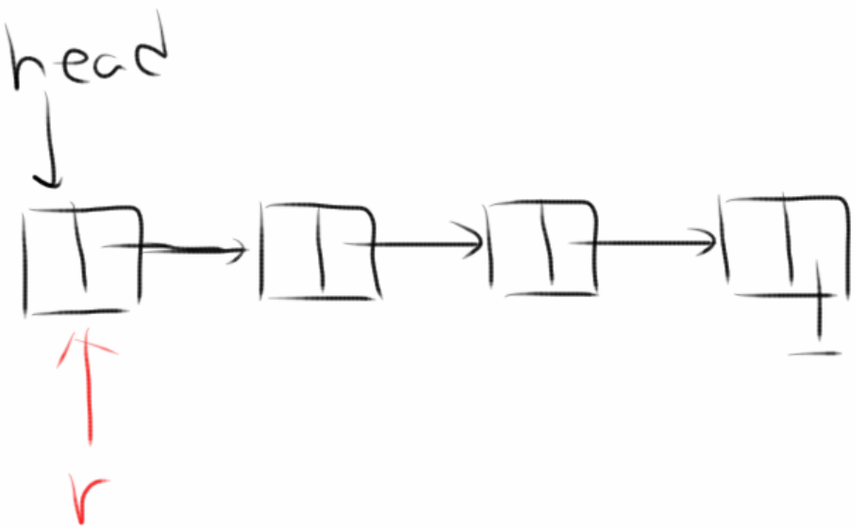
remove by a pointer IV

- p->next = r->next;



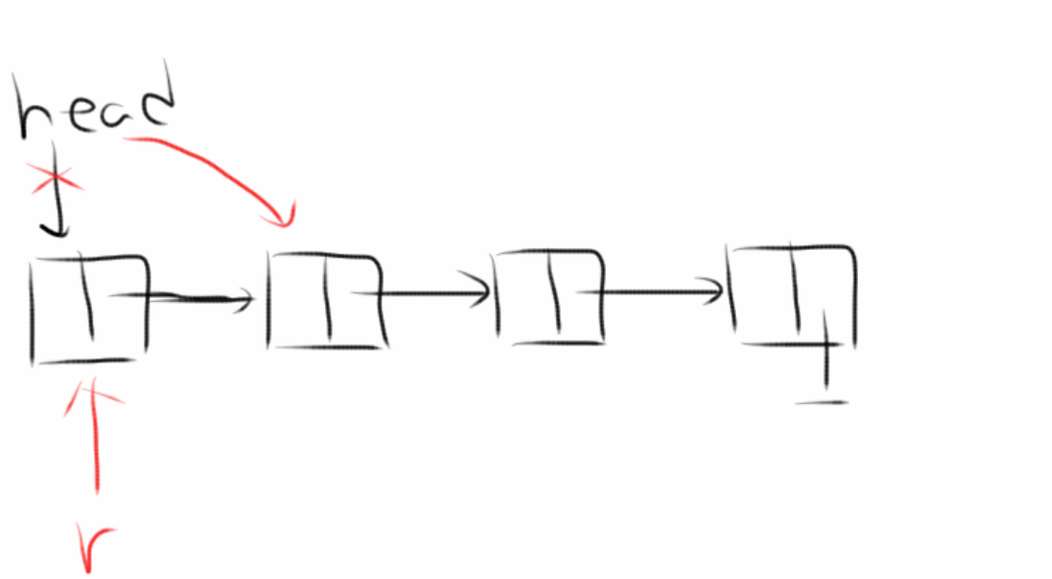
remove by a pointer V

- What if r is the head?



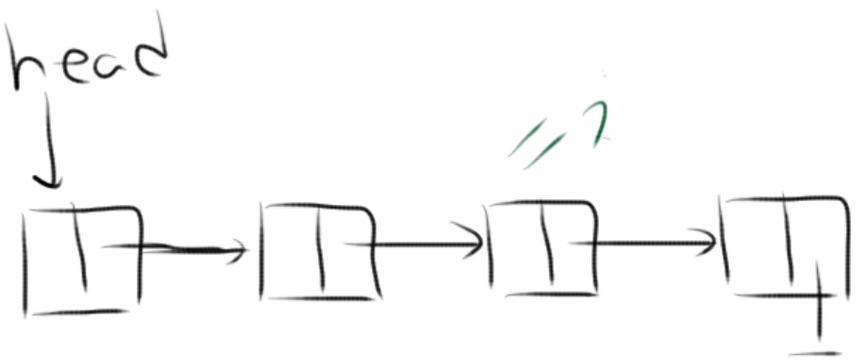
remove by a pointer VI

- new head = r->next



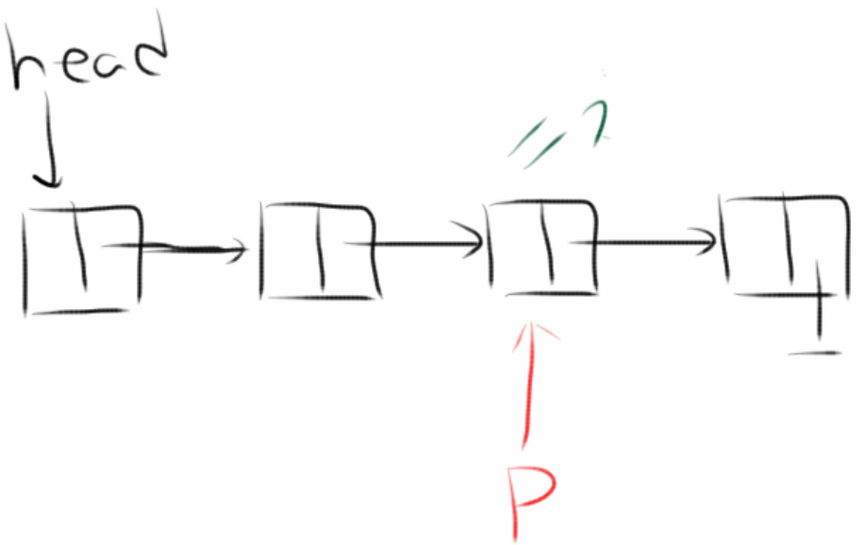
remove by a value I

- remove(int i);



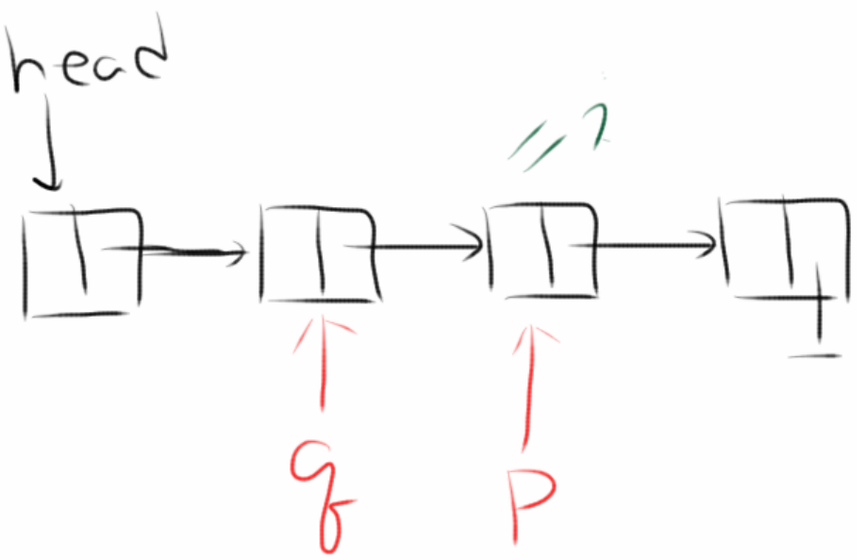
remove by a value II

- find p->value == i



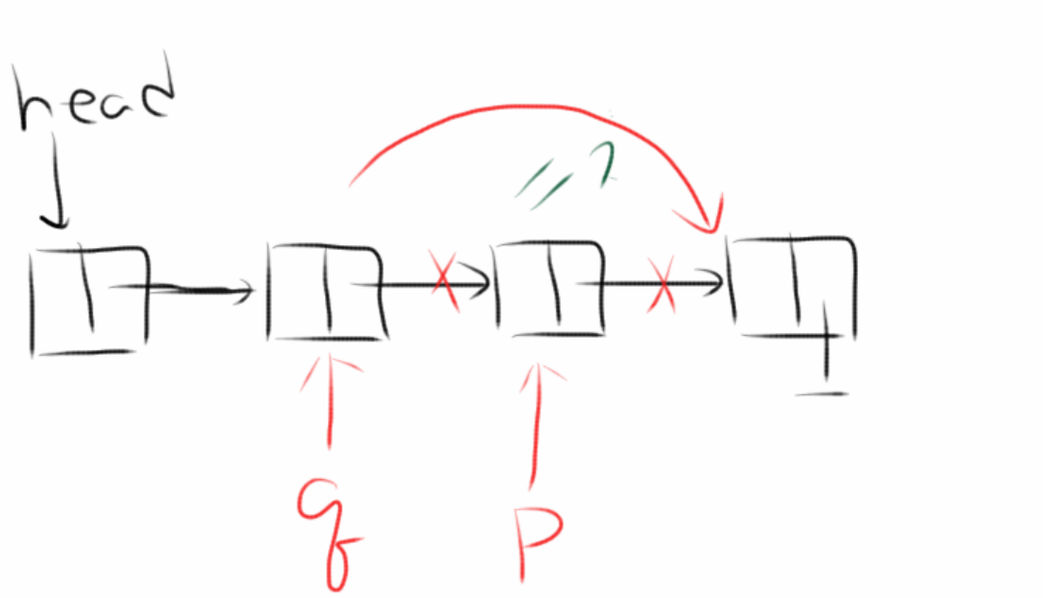
remove by a value III

- q for the previous node



remove by a value IV

- q->next = p->next;

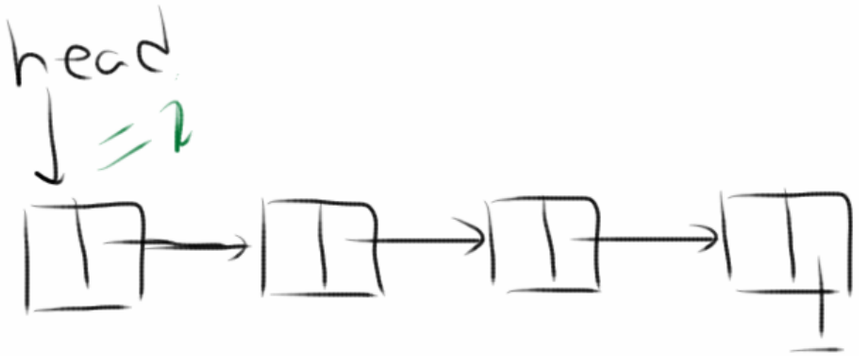


remove by a value V

```
for ( q=0,p = head; p; q=p,p=p->next ) {
  if ( p->value == i ) {
    q->next = p->next;
  }
}
```

remove by a value VI

- what if head->value == i?



How do we find the boundary?

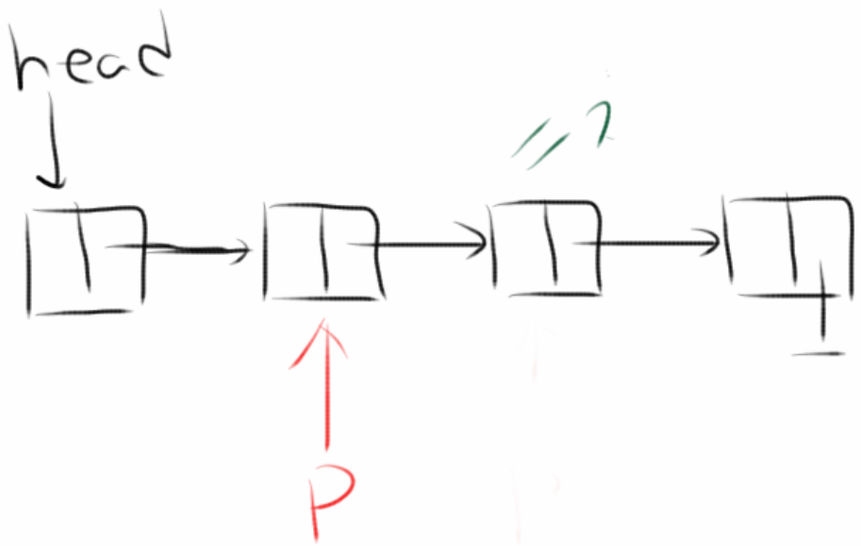
```
for ( q=0,p = head; p; q=p,p=p->next ) {
  if ( p->value == i ) {
    q->next = p->next;
  }
}
```

- Any pointer at the left of -> must be checked

remove by a value VII

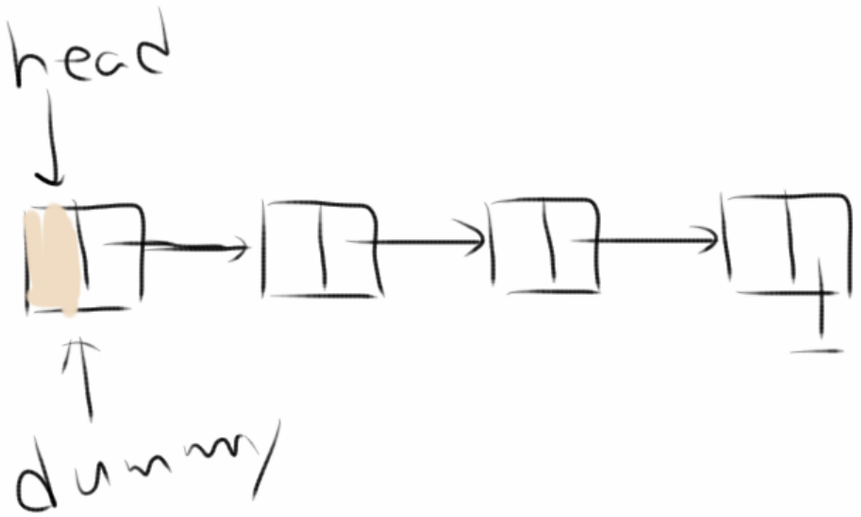
- how about test next->value?
- if (p->next && p->next->value == i)

how to do with head?



sentinel node

- a dummy head to make code smooth



all funcs with sentinel

- add_head
- append_tail
- traversal/search
- remove

any benefit?

clear the whole list I

```
void clear(Node *head)
{
  if ( head->next )
    clear(head->next);
  free(head);
}
```

clear the whole list II

```
for ( p = head; p; p=q ) {
    q = p->next;
    free(p);
}
```

append tail

- find the tail
- tail->next = n;
- n->next = 0;
- what if empty list?

struct List II

```
typedef struct {

        Node* head;

        Node* tail;

} List;



void add_head(List* list, int i);
```

all funcs with tail

- add_head
- append_tail
- traversal/search
- remove
- clear

any benefit?