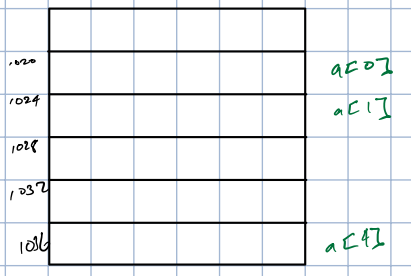
Part 1

Lists can grow & shrink over time
to find n-th item, time $\propto n$

Linked lists construct w/ structs & malloc
grow/shrink over time
accessing nth term is proportional to n

Array
accessing n th term is proportional to constant
fixed length



can't change size of stack allocated arrays ↝ same w/ shrinking
heap allocated might grow w/ realloc ↝ can shrink but space might not be useful

* arrays shouldn't change

change granularity
allocate one list elem. @ a time
give up on requirement that adjacent list elements will be stored in adjacent memory locations

benefits
easy to grow

drawbacks
have to track locations

typedef struct intlist intlist_t

```
struct intlist
    int val
    intlist *next
```

often put this in file, not header

build back to front point to next element

```
intlist_t *cons(int val, intlist_t *rest)
    intlist_t *rv = (intlist_t*) malloc(sizeof(intlist_t))
    if (rv == NULL)
        exit(1)
    rv -> val = val
```

free list

```
void list_free(intlist_t *lst)
    if(lst ...
```

```
    rv -> rest = res
    return rv
```

```
bool    is_empty(intlist_t *lst)
```

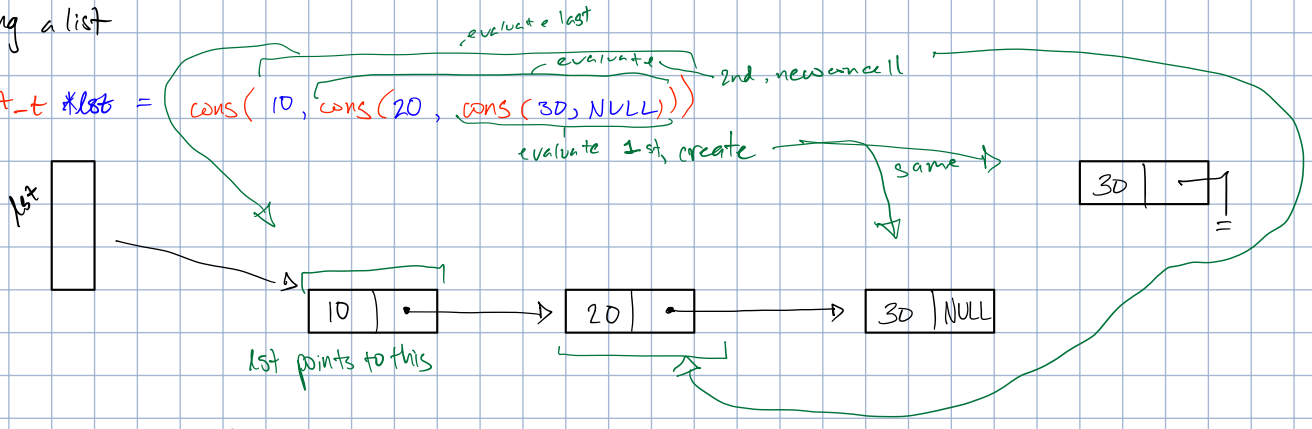
```
int     first(intlist_t *lst)
    return  lst -> val
```

```
intlist_t  *next(intlist_t *lst)
    assert (lst != NULL)
    return  lst->next
```

```
    return
list_free(rest(lst))
free(lst)
```

```
void  list_free(intlist_t *lst)
int list_t  *next
while(lst != NULL)
    next = lst->next
    free(lst)
    lst = next
```

making a list



lst points to beginning of list
—▷ struct contains 10 & pointer to second elem.

lst -> next second
—▷ struct contains 20 & pointer to third elem

lst -> next -> next
—▷ struct contains 30 & NULL pointer

lst -> val —▷ 10

lst -> next -> val —▷ 20

lst -> next -> next -> val —▷ 30

lst -> next -> next -> next —▷ NULL

```
int   sumlist(intlist_t * lst)
    if (is_empty(lst))
        return 0

    return   first(lst) + sumlist(next(lst))
```

```
int   sumlist(intlist_t * lst)
    int  rv=0
    while(lst != NULL)
        rv += lst->val
        lst = lst->next
    return rv
```

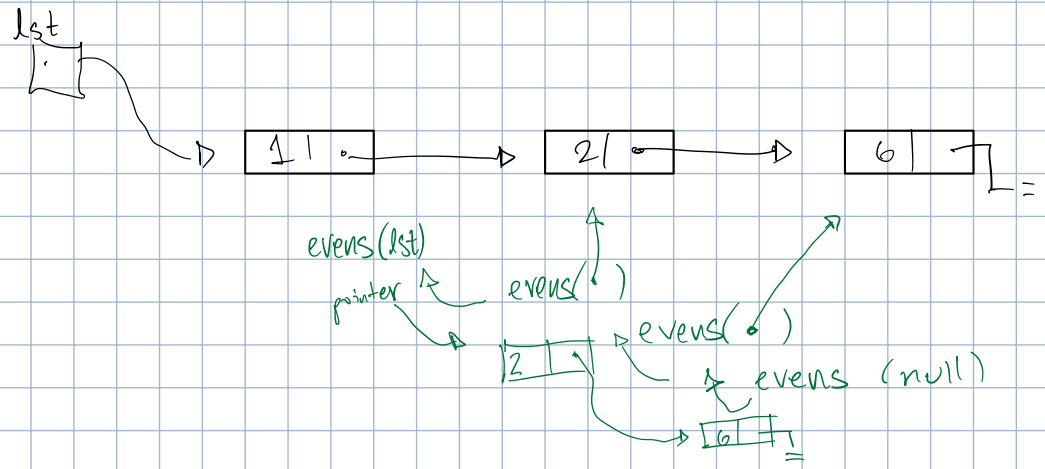
f^n can't look @ previous lst elem.

nondestructive: new list!

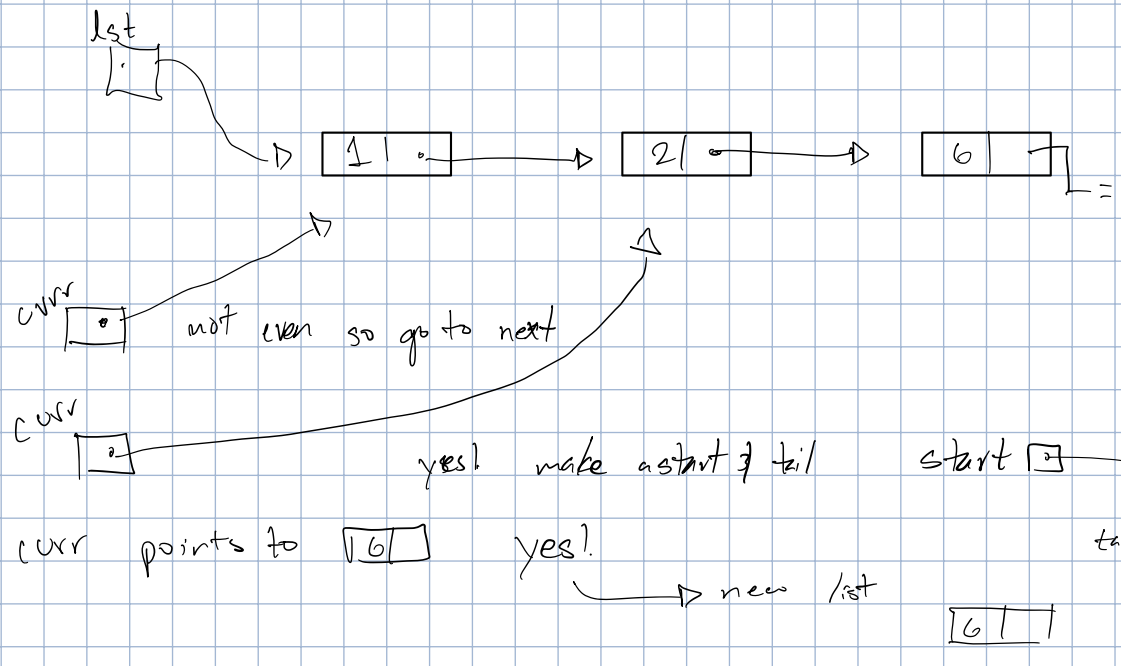
```
int_list  *evens(intlist_t * lst)
    if  (is_empty(lst))
        return  null
```

recursively in a squiggles copy file

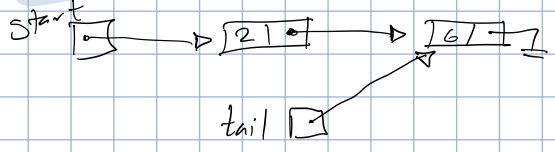
```
if (first(lst) % 2 == 0)
    return cons(first(lst) * first(lst), squares_copy(rest(lst)))
return evens(rest(lst))
```



iterative



now re define tail



Zero, One, Many

```
int_list *evens2(intlist_t *in)
    intlist_t *out = null
    intlist_t *tail                    *tail of out
    while (in != null)                 3 cases: empty, one, many
        if ((in->val) %2==0)           empty case
                                       is it even?
            intlist_t *item = cons(in->val, null)
            if (out==null)             this is first even
                out = item
            else
                tail->next = item      redefine tail
```

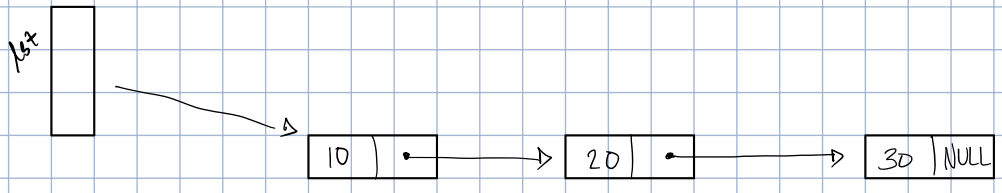
```
        tail = item

        in = in -> next

    return out
```

```
int_list *evens3( intlist_t * in )

    intlist_t *dummystart= cons( 0, null )
    intlist_t *tail = dummy start

    intlist_t *curr = in

    while ( curr != null )
        if ((curr->val)%2 == 0 )
            tail ->next = cons(curr-> val, null )

            tail = tail -> next

        curr = curr ->next

    intlist_t *rv = dummy-start ->nxt
    free ( dummy-start )

    return rv
```

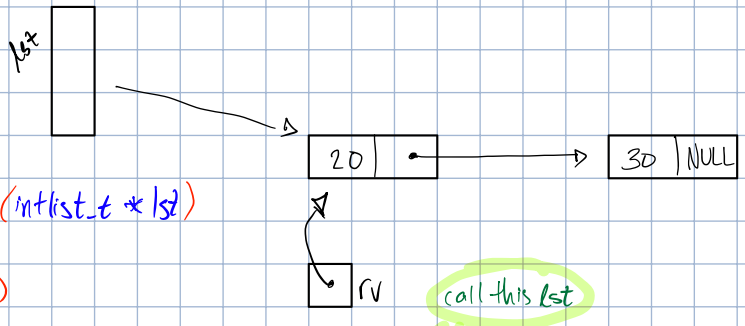
Part 3

remove first element

version 1



lst = remove_front(lst)

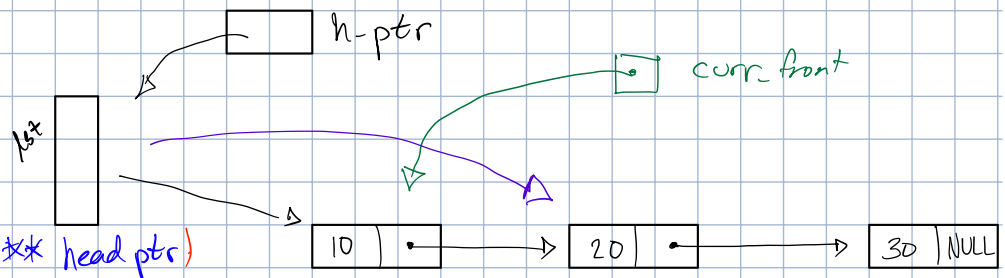



```
intlist_t *remove_front(intlist_t * lst)

    assert (lst != NULL)

    intlist_t *rv = lst -> next   keep track of next of lst
    free(lst )                    free space for first element in list

    return rv
```

call this lst

version 2

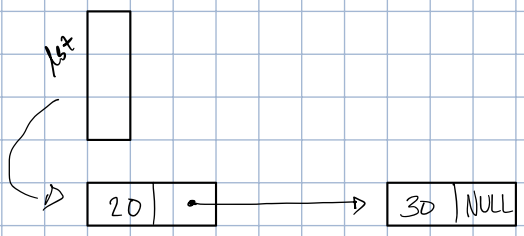
remove_front(& lst)



void remove_front(intlist_t ** head_ptr)

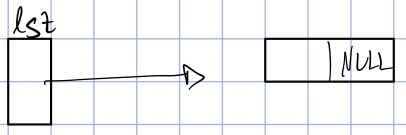
assert (head_ptr != NULL) input isn't null

assert (*head_ptr != NULL) lst isn't empty

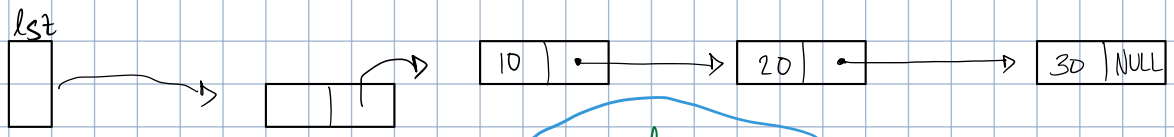


intlist_t *curr_front = *head_ptr
*head_ptr = curr_front -> next ★
free(curr_front)

dummy head

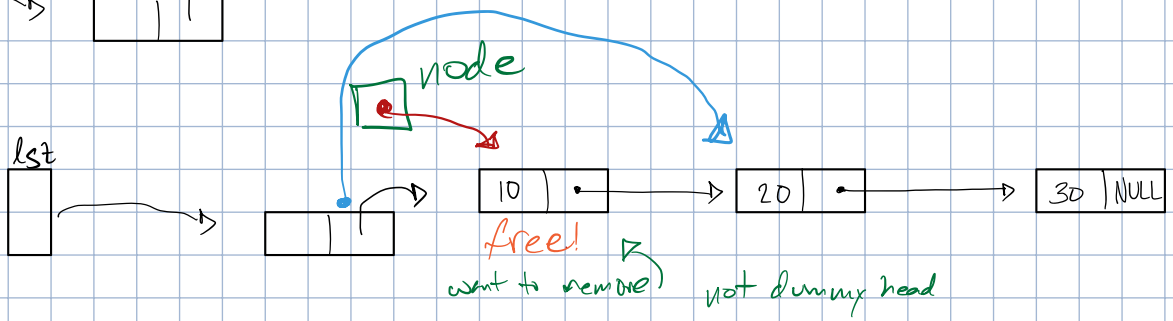


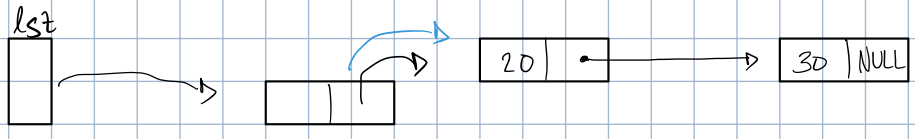
length 1 empty list



version 3

lst isn't null
lst isn't empty





void remove_front(intlist_t * lst)

assert (lst -> NULL)
assert (lst -> next != NULL) lst must have at least 1 element

intlist_t *node = lst -> next ★ grab node to remove

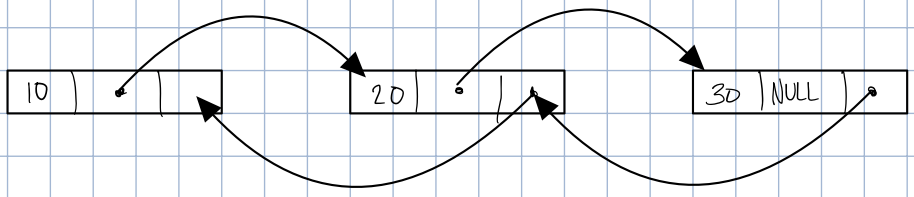
lst -> next = node -> next ★ link around the node

free(node) ★

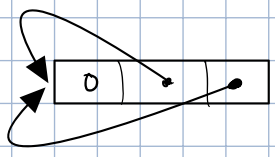
don't change lst

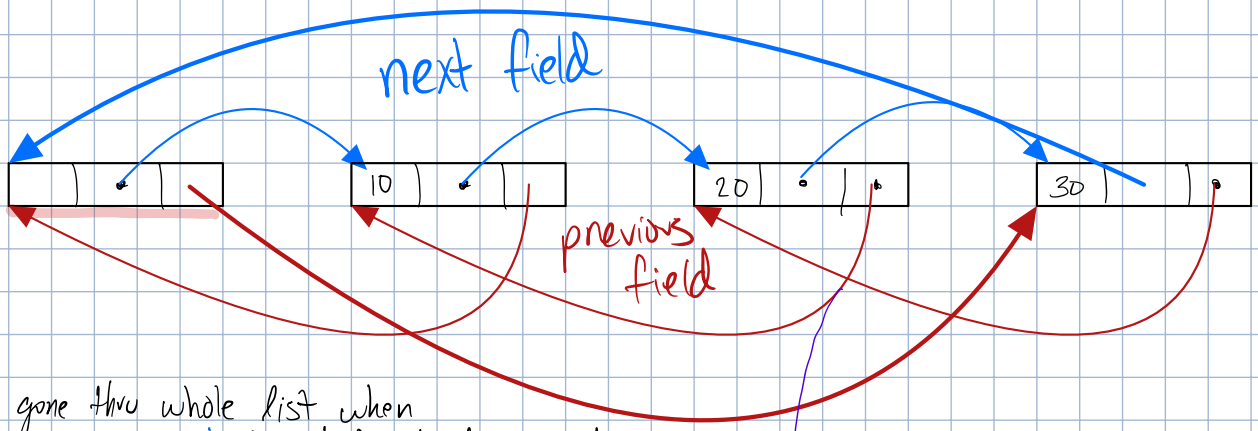
version 1: client update ptr
version 2:
version 3: dummy node

doubly linked lists



dummy node





we've gone thru whole list when we increment & get back dummy node

```
void dll_remove_front(dll_t *lst)

    assert(lst!=NULL)
    assert(lst->next!=NULL)

    dll_t *node = lst->next

    lst->next = node->next
    node->next->prev = lst

    free(node)
```

redefine next's previous this points to lst now