Lecture 15. Dynamic Data Structures

- Pointers and structures can be used to build data structures that ***expand*** and ***shrink*** during execution, e.g., lists, stacks, queues, trees, …
- Dynamic data structures are constructed using ***self-referential*** structure types

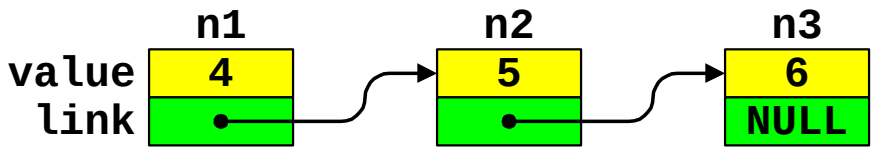
```
struct node {
    int value;
    struct node *link;
};
```

Declares a structure type with two fields

`value`	holds a integer
`link`	holds a pointer to a `struct` node

The type `struct` node is defined in terms of ***itself*** — self reference

```
struct node n1, n2, n3;

n1.value = 4;
n1.link = &n2;
n2.value = 5;
n2.link = &n3;
n3.value = 6;
n3.link = NULL;
```



Builds a ***singly linked list*** with 3 nodes holding 4, 5, and 6

Lists

- Use a pointer to ***traverse*** a list — follow the `link` fields until you reach `NULL`

```
struct node *p;

for (p = &n1; p != NULL; p = p->link)
    printf("%d\n", p->value);
```

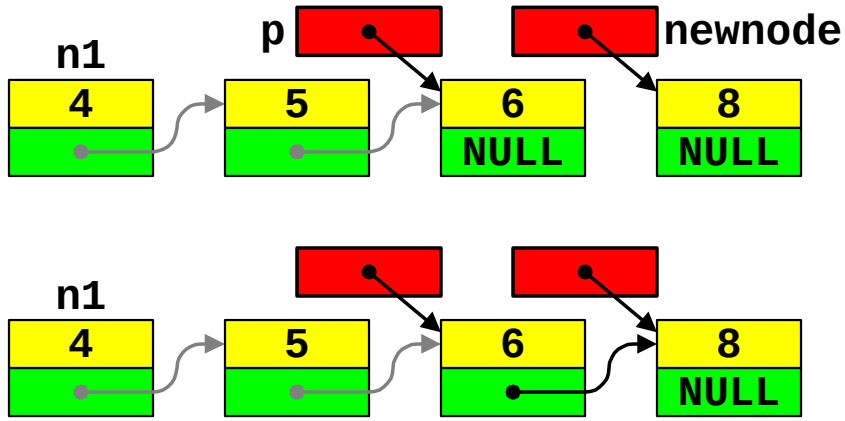
```
4
5
6
```

- Use `emalloc`/`malloc` to allocate as many `struct` nodes as needed

```
struct node *newnode = emalloc(sizeof (struct node));
newnode->value = 8;
newnode->link = NULL;
```

- To add a new node at the end of the list, walk a pointer down to the ***last node***

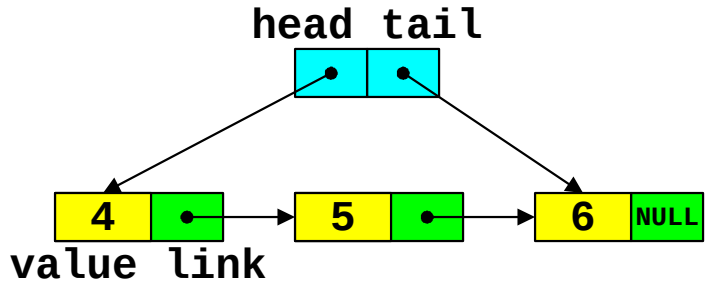
```
for (p = &n1; p->link != NULL; p = p->link)
    ;
p->link = newnode;
```



List Headers

- Using a header node often simplifies list manipulations

```
struct intlist {
    struct node *head;
    struct node *tail;
};
```



- Important boundary conditions

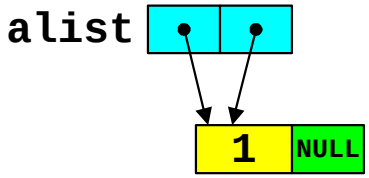
```
struct intlist alist;
alist.head = alist.tail = NULL;
```

creates an ***empty list***

so does `struct intlist alist = { NULL, NULL };`

```
struct node *p = emalloc(sizeof (struct node));

p->value = 1;
p->link = NULL;
alist.head = alist.tail = p;
```

creates a ***one-node list***



- List headers can be allocated, too, if you need an arbitrary number of lists (as opposed to a list of arbitrary length)

```
struct intlist *mylist = emalloc(sizeof (struct intlist));
```

A Simple List Module

- **The *interface* defines the list types and list-manipulation functions**

```
/* Lists of ints */

struct intnode {
    int value;
    struct intnode *link;
};

struct intlist {
    struct intnode *head;
    struct intnode *tail;
};

extern void intlist_addhead(struct intlist *list, int value);
/* adds a new node holding value at the beginning of list */

extern void intlist_addtail(struct intlist *list, int value);
/* Adds a new node holding value at the end of list */

extern int  intlist_remhead(struct intlist *list);
/* Removes the node at the beginning of a non-empty list
    and returns the value from that node */
```

  This interface appears in `intlist.h`

- **This kind of interface is an *abstract data type* because it defines a type and the operations on values of that type**

Implementing the List Module

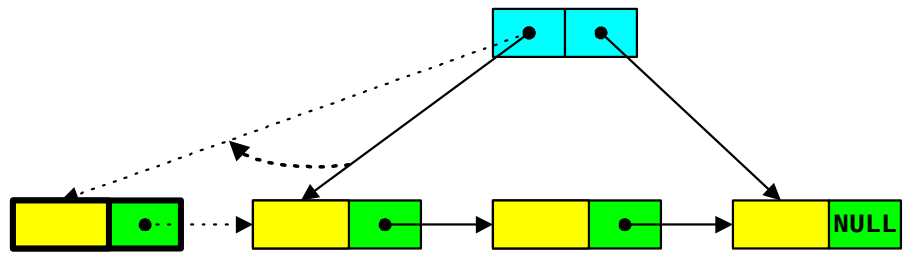
- **The *implementation* defines the functions specified in the interface**

```
/* Implementation of lists of ints */

#include <stdlib.h>
#include "intlist.h"
#include "misc.h"

void intlist_addhead(struct intlist *list, int value) { ... }
void intlist_addtail(struct intlist *list, int value) { ... }
extern int intlist_remhead(struct intlist *list) { ... }
```

This implementation appears in `intlist.c`

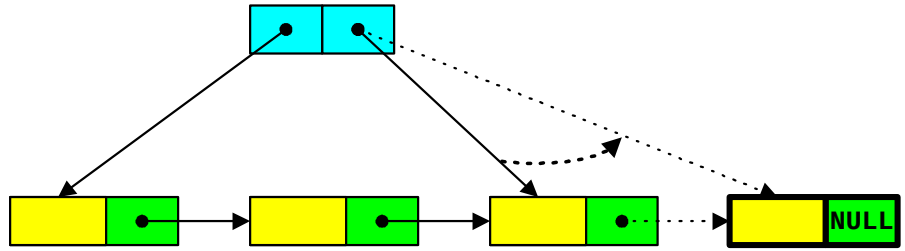
- **Adding a new node at the *head* of an `intlist` — beware *boundary conditions***

```
void intlist_addhead(struct intlist *list, int value) {
    struct intnode *p = emalloc(sizeof (struct intnode));

    p->value = value;
    if (list->head == NULL) {
        p->link = NULL;
        list->head = list->tail = p;
    } else {
        p->link = list->head;
        list->head = p;
    }
}
```



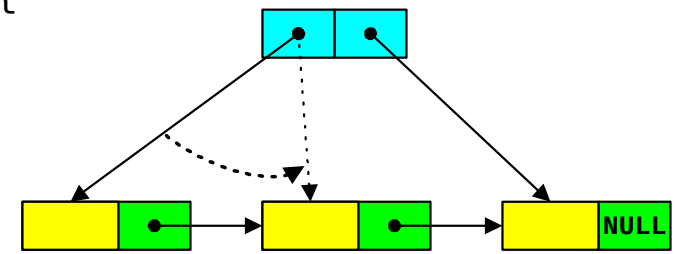
Implementing the List Module, cont'd

```
void intlist_addtail(struct intlist *list, int value) {
    struct intnode *p = emalloc(sizeof (struct intnode));

    p->value = value;
    p->link = NULL;
    if (list->tail == NULL)
        list->head = list->tail = p;
    else {
        list->tail->link = p;
        list->tail = p;
    }
}
```



- **When a node is deleted, it is also *deallocated***

```
int intlist_remhead(struct intlist *list) {
    int value;
    struct intnode *p = list->head;

    if (list->head == list->tail)
        list->head = list->tail = NULL;
    else
        list->head = p->link;
    value = p->value;
    free(p);
    return value;
}
```




```
free(p);
return p->value;
```

Wrong! **Why?**

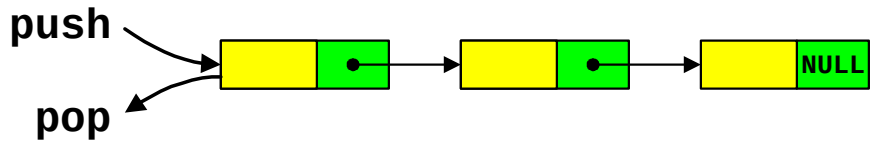
Sorting Revisited

- **Another way to sort an arbitrary number of integers**
  1. **Read them into an `intlist`, thus determining the number of integers**
  2. **Allocate an array**
  3. **Pour the integers in the list into the array**
  4. **Sort it and print it**

```
#include <stdio.h>
#include "quicksort.h"
#include "intlist.h"
#include "misc.h"

int main(void) {
    int i, n, *ptr, x;
    struct intlist input = { NULL, NULL };

    for (n = 0; scanf("%d", &x) == 1; n++)
        intlist_addtail(&input, x);
    ptr = emalloc(n*sizeof (int));
    for (i = 0; i < n; i++)
        ptr[i] = intlist_remhead(&input);
    quicksort(ptr, 0, n - 1);
    for (i = 0; i < n; i++)
        printf("%d\n", ptr[i]);
    return 0;
}
```

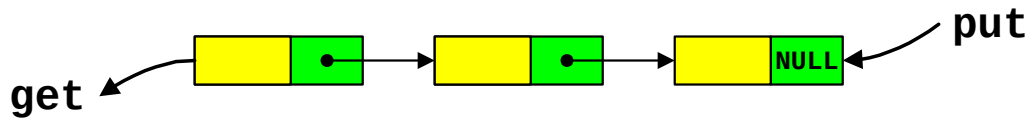
Other Kinds of Lists

- **Stacks: Add/remove nodes at only one end**



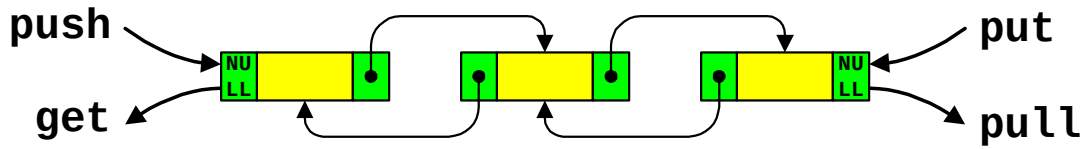

```
push    intlist_addhead
pop     intlist_remhead
```

- **Queues: Add nodes at the tail, remove nodes from the head**




```
put     intlist_addtail
get     intlist_remhead
```

- **What about `intlist_remtail`? Need a *doubly* linked list for efficient removal**
- **Deques: Add/remove nodes at either end**




```
push    intlist_addhead          put     intlist_addtail
get     intlist_remhead          pull    intlist_remtail
```