Week 10 - Monday

CS222

Last time

- What did we talk about last time?
- Linked lists
- **`typedef`**
- Lab 9

Questions?

Project 5

Quotes

A good programmer is someone who looks both ways before crossing a one-way street.

Doug Linder
Systems Administrator

More on Linked Lists

An example node struct

- We'll use this definition for our node for singly linked lists

```
typedef struct _node
{
      int data;
      struct _node* next;
} node;
```

- Somewhere, we will have the following variable to hold the beginning of the list

```
node* head = NULL;
```

Remove

- Let's define a function that takes a pointer to a (possibly empty) linked list and deletes the first occurrence of a given value
  - List is unchanged if the value isn't found
- There are two possible ways to do it
  - Return the new head of the list

```
node* remove(node* head, int value);
```

  - Take a pointer to a pointer and change it directly

```
void remove(node** headPointer, int value);
```

Insert in sorted order

- Let's define a function that takes a pointer to a (possibly empty) linked list and adds a value in sorted order (assuming that the list is already sorted)
- There are two possible ways to do it
  - Return the new head of the list

```
node* insert(node* head, int value);
```

  - Take a pointer to a pointer and change it directly

```
void insert(node** headPointer, int
value);
```

Empty

- Let's write a method that will remove all the nodes from a singly linked list
  - Don't forget to free all the nodes!

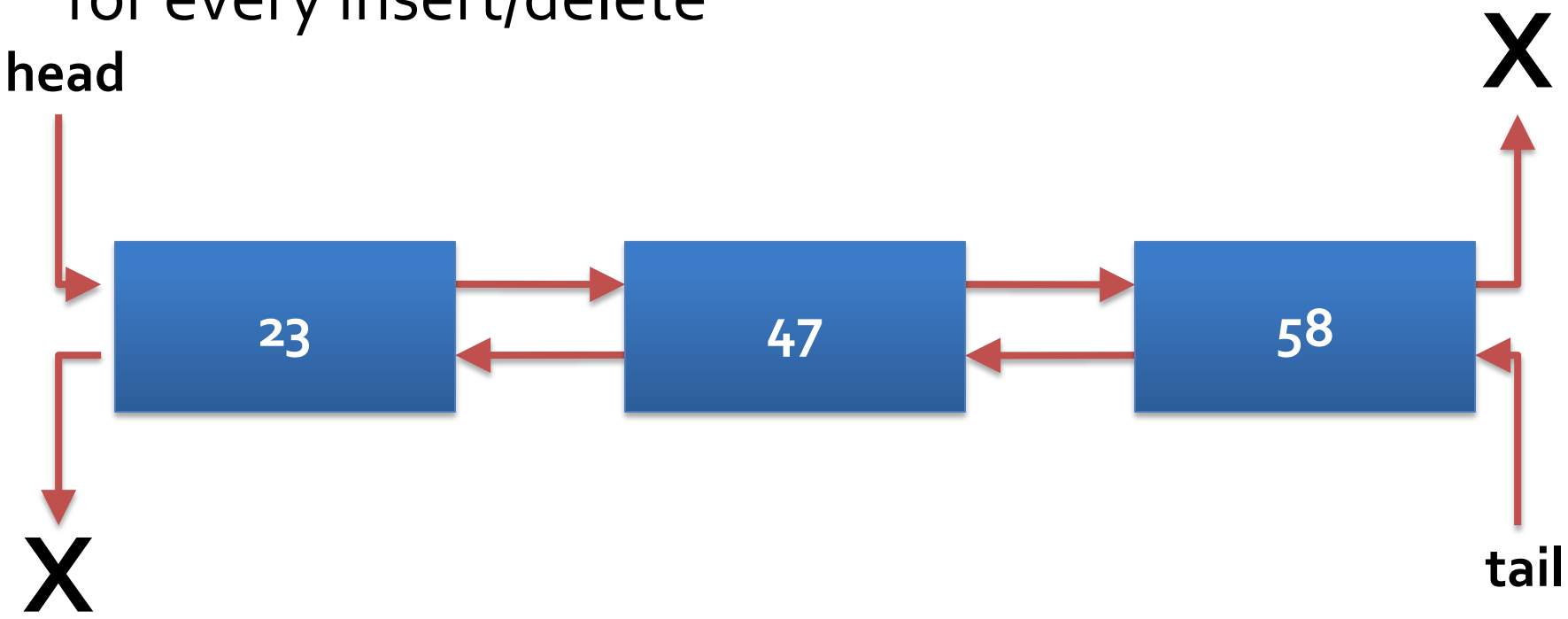
```
void empty(node* head);
```

- With this implementation, the user will have to set **`head`** to **`NULL`** manually

Doubly Linked Lists

Doubly linked list

- Node consists of data, a next pointer, and a previous pointer
- Advantages: bi-directional movement
- Disadvantages: slower, 4 pointers must change for every insert/delete



An example node struct

- We'll use this definition for our node for doubly linked lists

```
typedef struct _node
{
      int data;
      struct _node* next;
      struct _node* previous;
} node;
```

- Somewhere, we will have the following variables to hold the beginning and ending of the list

```
node* head = NULL;
node* tail = NULL;
```

Add to front

- Let's define a function that adds a value to the front of a (possibly empty) doubly linked list
- Since the head and the tail might both get updated, the only reasonable way to write the function is to take a pointer to each

```
void addToFront(node** headPointer,
node** tailPointer, int value);
```

Add to back

- Let's define a function that adds a value to the back of a (possibly empty) doubly linked list
- Since the head and the tail might both get updated, the only reasonable way to write the function is to take a pointer to each

```
void addToBack(node** headPointer, node**
tailPointer, int value);
```

Insert in sorted order

- Let's define a function that takes a pointer to a (possibly empty) doubly linked list and adds a value in sorted order (assuming that the list is already sorted)
- Since the head and the tail might both get updated, the only reasonable way to write the function is to take a pointer to each

```
void insert(node** headPointer, node**
tailPointer, int value);
```

- In some ways, this task is slightly easier with a doubly linked list

enum

- There are situations where you'd like to have a set of named constants
- In many cases, you'd like those constants to be different from each other
- What if there were a way to create such a list of constants easily?
- Enter **`enum`**!

Using enum

- To create these constants, type enum and then the names of your constants in braces

```
enum { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY };
```

- Then in your code, you can use these values (which are stored as integers)

```
int day = FRIDAY;
if( day == SUNDAY )
     printf("My 'I don't have to run' day");
```

Creating enum types

- You can also create named enum types

```
enum Color { BLACK, BLUE, GREEN, ORANGE, PURPLE,
RED, WHITE, YELLOW };
```

- Then you can declare variables of these types

```
enum Color color;
color = YELLOW;
```

- Naturally, because they are constants, it is traditional to name enum values in ALL CAPS

typedef + enum

- If you want to declare **enum** types (and there isn't much reason to, since C treats them exactly like **int** values), you can use **typedef** to avoid typing **enum** all the time

```
typedef enum { C, C_PLUS_PLUS, C_SHARP, JAVA,
JAVASCSRIPT, LISP, ML, OBJECTIVE_C, PERL, PHP,
PYTHON, RUBY, VISUAL_BASIC } Language;

Language language1 = C;
Language language2 = JAVA;
```

enum values

- **enum** values by default start at **0** and increase by one with each new constant

```
enum { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY };
```

- In this case, the constants have the following numbering
  - **SUNDAY**: 0
  - **MONDAY** : 1
  - **TUESDAY** : 2
  - **WEDNESDAY** : 3
  - **THURSDAY** : 4
  - **FRIDAY** : 5
  - **SATURDAY** : 6

Specifying values

- You can even specify the values in the **enum**

```
enum { ANIMAL = 7, MINERAL = 9, VEGETABLE = 11 };
```

- If you assign values, it is possible to make two or more of the constants have the same value (usually bad)
- A common reason that values are assigned is so that you can do bitwise combinations of values

```
enum { PEPPERONI = 1, SAUSAGE = 2, BACON = 4,
MUSHROOMS = 8, PEPPER = 16, ONIONS = 32, OLIVES
= 64, EXTRA_CHEESE = 128 };

int toppings = PEPPERONI | ONIONS | MUSHROOMS;
```

A classic enum

- One of the most common uses of **enum** is to specify a Boolean type

```
typedef enum { FALSE, TRUE } BOOLEAN;

BOOLEAN value = TRUE;
BOOLEAN flag = FALSE;
```

- It's not a perfect system, since you can assign values other than **0** and **1** to a **BOOLEAN**
- Likewise, other values are also true in C

Upcoming

Next time…

- Binary search trees
- File I/O

Reminders

- Keep working on Project 5
- Read K&R chapter 7
- **Exam 2 next Wednesday**