Week 9 - Wednesday

COMP 2400

Last time

- What did we talk about last time?
- More on linked lists
- **`enum`**
- Bit fields

Questions?

Project 4

Saving Space

Saving space

- The next topics we'll discuss today are primarily about saving space
- They don't make code safer, easier to read, or more time efficient
- At C's inception, memory was scarce and expensive
- These days, memory is plentiful and cheap

Bit fields in a struct

- You can define a struct and define how many bits wide each element is
  - It only works for integral types, and it makes the most sense for **`unsigned int`**
  - Give the number of bits it uses after a colon
  - The bits can't be larger than the size the type would normally have
  - You can have unnamed fields for padding purposes

```
typedef struct _toppings
{
      unsigned pepperoni : 1;
      unsigned sausage   : 1;
      unsigned onions    : 1;
      unsigned peppers   : 1;
      unsigned mushrooms : 1;
      unsigned sauce     : 1;
      unsigned cheese    : 2; //goes from no cheese to triple cheese
} toppings;
```

Code example

- You could specify a pizza this way

```
toppings choices;
memset(&choices, 0, sizeof(toppings));
//sets the garbage to all zeroes
choices.pepperoni = 1;
choices.onions = 1;
choices.sauce = 1;
choices.cheese = 2; //double cheese
order(&choices);
```

Struct size and padding

- Structs are always padded out to multiples of 4 or even 8 bytes, depending on architecture
  - Unless you use compiler specific statements to change byte packing
- After the last bit field, there will be empty space up to the nearest 4 byte boundary
- You can mix bit field members and non-bit field members in a struct
  - Whenever you switch, it will pad out to 4 bytes
  - You can also have **0** bit fields which also pad out to 4 bytes

Padding example

```
struct kitchen
{
   unsigned light     : 1;
   unsigned toaster   : 1;
   int count;   // 4 bytes
   unsigned outlets   : 4;
   unsigned           : 4;
   unsigned clock     : 1;
   unsigned           : 0;
   unsigned flag      : 1;
};
```

16 bytes

Data	Bits
light	1
toaster	1
padding	30
count	32
outlets	4
unnamed	4
clock	1
unnamed	0
padding	23
flag	1
padding	31

An alternative to bitwise operations

- You can also use a pointer to a struct with bit fields to read bit values out of other types

```
typedef struct
{
      unsigned LSB      : 1;
      unsigned          : 30;
      unsigned MSB      : 1;
} bits;
```

- Which bit is which is dependent on endianness

```
bits* bitsPointer;
int number = 1;
float value = 3.7;
bitsPointer = (bits*)&number;
printf("LSB: %d\nMSB: %d\n", bitsPointer->LSB, bitsPointer->MSB);
```

Unfortunately …

- Bit fields are compiler and machine dependent
- How those bits are ordered and packed is not specified by the C standard
- In practice, they usually work
  - Most machines are little endian these days
  - You're okay if your code is always running on the same machine
- In theory, endianness and packing problems can interfere

Unions

Unions

- What if you wanted a data type that could hold any of three (or more!) different things
  - But it would only hold one at a time …
- Yeah, you probably wouldn't want that
- But, back in the day when space was important, maybe you would have
- This is exactly the problem that unions were designed to solve

Declaring unions

- Unions look like structs
  - Put the keyword **`union`** in place of **`struct`**

```
union Congressperson
{
     int district;   // Representatives
     char state[15]; // Senators
};
```

- There *isn't* a separate district and a state
  - There's only space for one at a time
  - The total space is big enough to hold the larger one
  - In this case, 15 bytes (rounded up to 16) is the larger one

Example use

- We can store into either one

```
union Congressperson representative;
union Congressperson senator;
representative.district = 1;
strcpy(senator.state, "Wisconsin");
printf("District: %d\n", senator.district);
// Whoa, what's the int value of Wisconsin?
```

- But … the other one becomes unpredictable

What's in the union?

- How can you tell what's in the union?
  - You can't!
- You need to keep separate information that says what's in the union
- Anonymous (unnamed) unions inside of structs are common

```
struct Congressperson
{
     bool senator;          // Which one?
     union
     {
           int district;    // Representatives
           char state[15]; // Senators
     };
};
```

Operands and operators

- We could use such a struct to store terms in an algebraic expression
- Terms are of the following types
  - Operands are double values
  - Operators are char values: **+**, **-**, *****, and **/**

```
typedef enum { OPERATOR, OPERAND } Type;
typedef struct
{
        Type type;
        union
        {
                double operand;
                char operator;
        };
} Term;
```

Binary Trees

Tree terminology

- A **tree** is a data structure built out of nodes with children
    - Every child has exactly one parent node
    - There are no loops in a tree
    - A tree expresses a hierarchy or a similar relationship
- The **root** is the top of the tree, the node which has no parents
- A **leaf** of a tree is a node that has no children
- An **inner node** is a node that does have children
- An **edge** or a **link** connects a node to its children
- A **subtree** is a node in a tree and all of its children

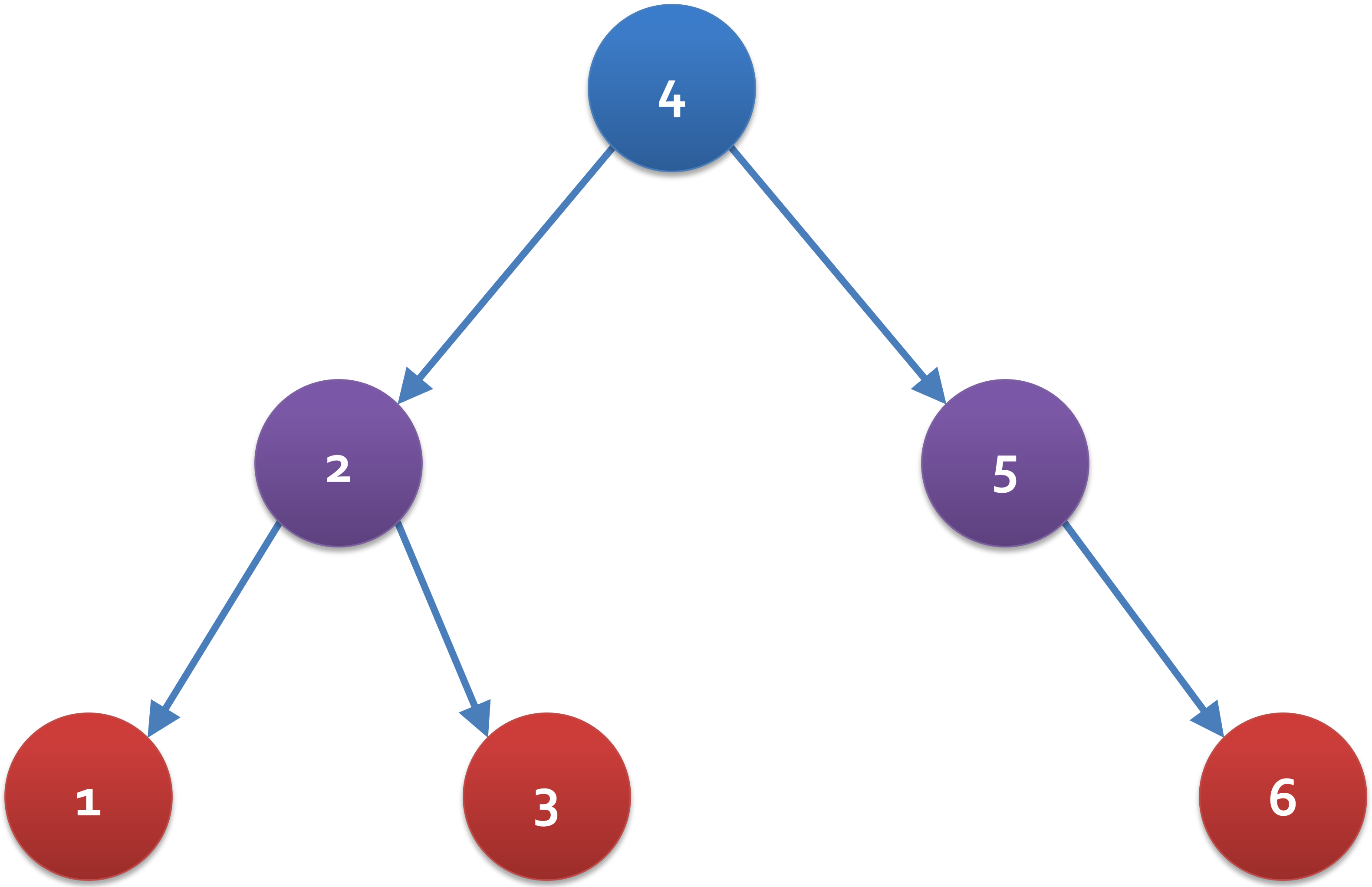
Binary tree

- A binary tree is a tree such that each node has two or fewer children
- The two children of a node are generally called the **left child** and the **right child**, respectively

Binary search tree (BST)

- A binary search tree is binary tree with three properties:
  1. The left subtree of the root only contains nodes with keys less than the root’s key
  2. The right subtree of the root only contains nodes with keys greater than the root’s key
  3. Both the left and the right subtrees are also binary search trees

Binary Search Tree



Example BST node in C

```c
typedef struct _Tree
{
     int data;
     struct _Tree* left;
     struct _Tree* right;
} Tree;
```

Finding an element in a BST

- Write a function that will find an element in a BST
- Use recursion
- **Hints:**
  - If the value is smaller than the current root, look to the left
  - If the value is larger than the current root, look to the right

```
Tree* find (Tree* root, int value);
```

Adding to a BST

- Write a function that will add an element to a BST
- Use recursion
- **Hint:** Look for the location where you would add the element, then add when you reach a **`NULL`**

```
Tree* add (Tree* root, int value);
```

Time

Time

- In the systems programming world, there are two different kinds of time that are useful
- Real time
  - This is also known as wall-clock time or calendar time
  - It's the human notion of time that we're familiar with
- Process time
  - Process time is the amount of time your process has spent on the CPU
  - There is often no obvious correlation between process time and real time (except that process time is never more than real time elapsed)

Calendar time

- For many programs it is useful to know what time it is relative to some meaningful starting point
- Internally, real world system time is stored as the number of seconds since midnight January 1, 1970
  - Also known as the Unix Epoch
  - Possible values for a 32-bit value range from December 13, 1901 to January 19, 2038
  - Systems and programs that use a 32-bit signed **`int`** to store this value may have strange behavior in 2038

`time()`

- The **`time()`** function gives back the seconds since the Unix Epoch
- Its signature is:

```
time_t time(time_t* timePointer);
```

- **`time_t`** is a signed 32-bit or 64-bit integer
- You can pass in a pointer to a **`time_t`** variable or save the return value (both have the same result)
- Typically we pass in **`NULL`** and save the return value
- Include **`time.h`** to use **`time()`**

```
time_t seconds = time(NULL);
printf("%d seconds have passed since 1970", seconds);
```

Ticket Out the Door

Upcoming

Next time…

- Finish time
- Union example
- Review for Exam 2

Reminders

- **Finish Project 4**
- Study for Exam 2
  - Next Monday in class
- Keep reading K&R chapter 7