CSE 1320

Week of 04/22/2019

Instructor : Donna French

Stack

```
typedef struct node
{
    int node_number;
    struct node *next_ptr;
} node;

node *StackTop = NULL;
```

Stack Push

```
void push(node **StackTop, int NodeNumber)
{
   node *NewNode = malloc(sizeof(node));
   NewNode->node_number = NodeNumber;
   NewNode->next_ptr = NULL;

   if (*StackTop == NULL)
   {
      *StackTop = NewNode;
   }
   else
   {
      NewNode->next_ptr = *StackTop;
      *StackTop = NewNode;
   }
}
```

Stack Pop

```
void pop(node **StackTop)
{
   node *TempPtr = *StackTop;

   if (*StackTop == NULL)
   {
      printf("Pop not executed - stack is empty\n\n");
   }
   else
   {
      free(*StackTop);
      *StackTop = TempPtr->next_ptr;
   }
}
```

Queue

```
typedef struct node
{
    int node_number;
    struct node *next_ptr;
} node;

node *QueueHead = NULL, *QueueTail =  NULL;
```

Queue Enqueue

```
void enQueue(int NewNodeNumber, node **QueueHead, node **QueueTail)
{
   node *NewNode = malloc(sizeof(node));
   NewNode->node_number = NewNodeNumber;
   NewNode->next_ptr = NULL;

   /* Queue is empty */
   if (*QueueHead == NULL)
   {
      *QueueHead = *QueueTail = NewNode;
   }
   else
   {
      (*QueueTail)->next_ptr = NewNode;
      *QueueTail = NewNode;
   }
}
```

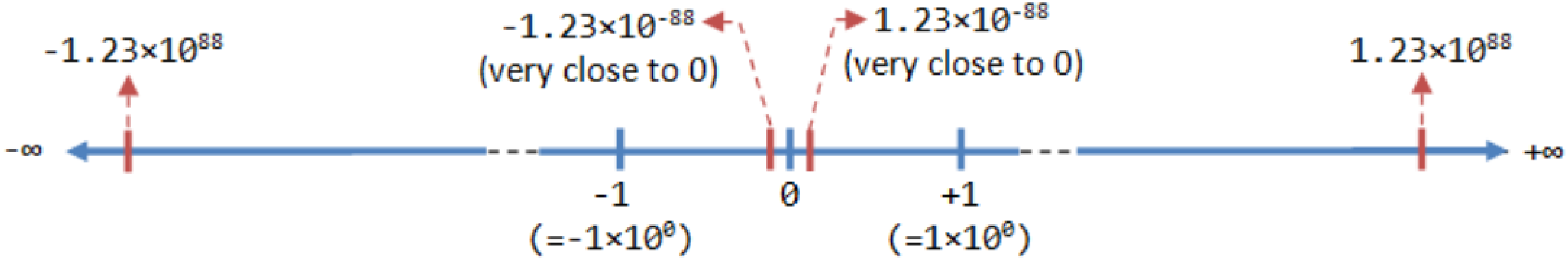
```c
void deQueue(node **QueueHead)
{
    node *TempPtr = (*QueueHead)->next_ptr;

    if (*QueueHead == NULL)
    {
        printf("Queue is emtpy\n\n");
    }
    else
    {
        free(*QueueHead);
        *QueueHead = TempPtr;
    }
}
```

Queue Dequeue

Floating Point

A floating-point number (or real number) can represent a very large (1.23×10^88) or a very small (1.23×10^-88) value.

It could also represent very large negative number (-1.23×10^88) and very small negative number (-1.23×10^88), as well as zero:



Floating-point Numbers (Decimal)

Floating Point

A floating-point number is typically expressed in scientific notation, with a fraction (F), and an exponent (E) of a certain radix (r), in the form of $F \times r^E$.

fraction is also called the mantissa.

Decimal numbers use radix of 10 ($F \times 10^E$)

Binary numbers use radix of 2 ($F \times 2^E$).

ra·dix

/ˈrādiks,ˈradiks/

noun

1. MATHEMATICS
   the base of a system of numeration.

Floating Point

Representation of floating point numbers is not unique.

```
55.66
    5.566×10^1
    0.5566×10^2
    0.05566×10^3
```

The fractional part can be ***normalized***.

In the normalized form, there is only a single non-zero digit before the radix point.

Floating Point

In the normalized form, there is only a single non-zero digit before the radix point.

For example, decimal number `123.4567` can be normalized as

    1.234567×10^2

binary number `1010.1011B` can be normalized as

    1.0101011B×2^3

Floating Point

It is important to note that floating-point numbers suffer from *loss of precision* when represented with a fixed number of bits (e.g., 32-bit or 64-bit).

This is because there are *infinite* number of real numbers (even within a small range of 0.0 to 0.1).

On the other hand, a *n*-bit binary pattern can represent a *finite* 2^*n* distinct numbers.

Hence, not all the real numbers can be represented. The nearest approximation will be used instead which results in loss of accuracy.

Floating Point

An *n*-bit binary pattern can represent a *finite* 2^*n* distinct numbers.

Computer uses *a fixed number of bits* to represent a piece of data, which could be a number, a character, or others.

A *n*-bit storage location can represent up to 2^*n* distinct entities.

For example, a 3-bit memory location can hold one of these eight binary patterns:

`000`, `001`, `010`, `011`, `100`, `101`, `110`, or `111`.

Hence, it can represent at most 8 distinct entities.

Floating Point

It is also important to note that floating number arithmetic is very much less efficient than integer arithmetic.

Floating point arithmetic can be speed up with a dedicated floating-point co-processor but this extra step still adds extra time.

Use integers if your application does not require floating-point numbers.

Floating Point

In computers, floating-point numbers are represented in scientific notation of fraction (F) and exponent (E) with a radix of 2, in the form of

$$F \times 2^E$$

Both E and F can be positive as well as negative.

Modern computers adopt **IEEE 754 Standard** for representing floating-point numbers.

There are two representation schemes: 32-bit single-precision and 64-bit double-precision.

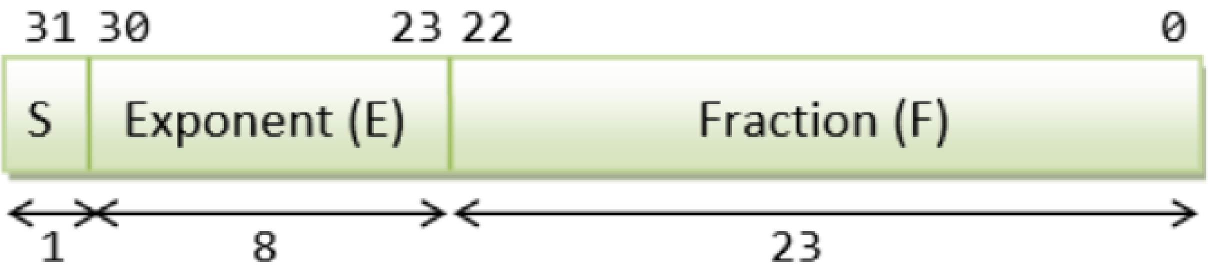
Floating Point

IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.

The following 8 bits represent exponent (E). The remaining 23 bits represents fraction (F).



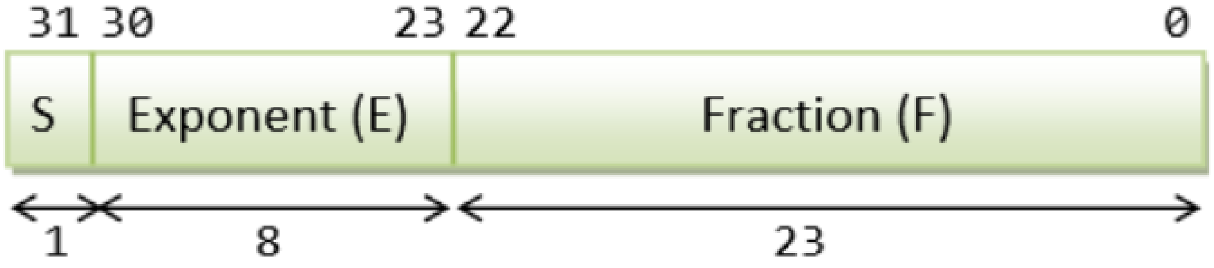
32-bit Single-Precision Floating-point Number

Floating Point

IEEE-754 32-bit Single-Precision Floating-Point Numbers

Floating-point numbers are represented in scientific notation of fraction (F) and exponent (E) with a radix of 2 F×2^E.

$+3.5_{10}$



32-bit Single-Precision Floating-point Number

+1.75x2^(128-127)

$+1.75_{10} = 1.11_2$
$128_{10} = 10000000_2$

$\underline{0}$ $\underline{10000000}$ $\underline{110\ 0000\ 0000\ 0000\ 0000\ 0000}$

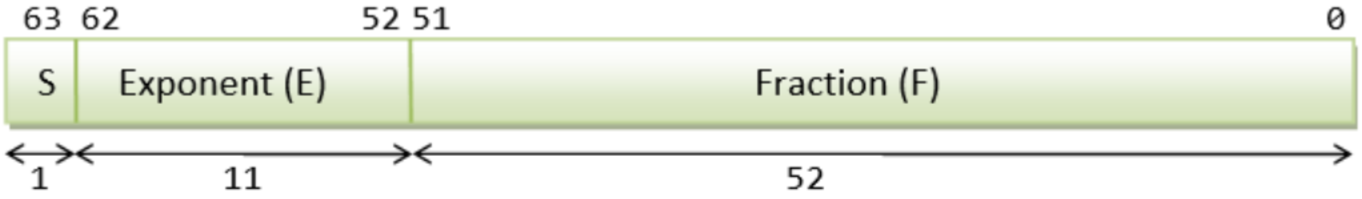
Floating Point

IEEE-754 64-bit Double-Precision Floating-Point Numbers

In 64-bit single-precision floating-point representation:

The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.

The following 11 bits represent exponent (E). The remaining 52 bits represents fraction (F).



64-bit Double-Precision Floating-point Number

```c
float f;
double d;

f = 1.0/3;
d = 1.0/3;

if (f == d)
{
    printf("f equals d!");
}
else
{
    printf("of course they don't equal!");
}
```

Why use 1.0 instead of 1?

```
Breakpoint 1, main () at floatingDemo.c:10
10              f = 1.0/3;
(gdb) step
11              d = 1.0/3;
(gdb)
13              if (f == d)
(gdb) p f
$1 = 0.333333343
(gdb) p d
$2 = 0.33333333333333331
(gdb) step
19                      printf("of course they don't equal!");
```

NaN

IEEE 754 floating point numbers can represent *NaN* (not a number).

NaN, not a number, is a numeric data type value representing an undefined or unrepresentable value, especially in floating-point arithmetic.

Systematic use of NaNs was introduced by the IEEE 754 floating-point standard in 1985.

For example, 0/0 is undefined as a real number, and is therefore represented by NaN.

The square root of a negative number is an imaginary number and cannot be represented as a real number, so is represented by NaN.

NaNs may also be used to represent missing values in computations.

```
float f1;
float f2;

f1 = sqrt(-1);
f2 = sqrt(-1);

if (f1 == f2)
{
    printf("f1 equals f2!");
}
else
{
    printf("of course they don't equal!");
}
```

```
Breakpoint 1, main () at floatingDemo.c:11
11              f1 = sqrt(-1);
(gdb) step
12              f2 = sqrt(-1);
(gdb)
14              if (f1 == f2)
(gdb) p f1
$3 = -nan(0x400000)
(gdb) p f2
$4 = -nan(0x400000)
(gdb) step
21              printf("of course they don't equal!");
(gdb) c
Continuing.
of course they don't equal!
Program exited normally.
```

```c
float f1;
float f2;

f1 = sqrt(-1);
f2 = sqrt(1);

if (isnan(f1))
{
    printf("f1 isnan() is true");
}
if (isnan(f2))
{
    printf("f2 isnan() is true");
}
```

```
Breakpoint 1, main () at floatingDemo.c:11
11              f1 = sqrt(-1);
(gdb) step
12              f2 = sqrt(1);
(gdb)
15              if (isnan(f1))
(gdb) p f1
$5 = -nan(0x400000)
(gdb) p f2
$6 = 1
(gdb) p isnan(f1)
$7 = 1
(gdb) p isnan(f2)
$8 = 0
(gdb) c
Continuing.
f1 isnan() is true
```

Given the declaration "`float f`", when is the expression "`if (f==f)`" false?

When `f` is a NaN.

Given

```
float f1 = M_PI;
double d1 = M_PI;
```

Is

```
if (f1 == d1)
```

always true?

No

```c
float f = sqrt(-1);
float f1 = M_PI;
double d2 = M_PI;

if (f == f)
{
      printf("f == f");
}
else
{
      printf("f != f");
}

if (f1 == d2)
{
      printf("f1 == d2");
}
else
{
      printf("f1 != d2");
}
```

```
Breakpoint 1, main () at floatingDemo.c:8
8                float f = sqrt(-1);
(gdb) step
9                float f1 = M_PI;
(gdb)
10               double d2 = M_PI;
(gdb)
16               if (f == f)
(gdb) p f
$1 = -nan(0x400000)
(gdb) p f1
$2 = 3.14159274
(gdb) p d2
$3 = 3.1415926535897931
(gdb) step
23                       printf("f != f");
(gdb)
26               if (f1 == d2)
(gdb)
32                       printf("f1 != d2");
```

Converting Decimal Fractions to Binary

9.25

Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

0.25 * 2 = **0**.5 We keep the **0**

Next we disregard the whole number part of the previous result (the 0 in this case) and multiply by 2 once again. The whole number part of this new result is the *second* binary digit to the right of the point.

0.5 * 2 = **1**.0 We keep the **1**

9_{10} in binary is 1001_2 so 9.25 is

1001.01_2

Converting Decimal Fractions to Binary

9.625

Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

0.625 * 2 = **1**.250 We keep the **1**

Next we disregard the whole number part of the previous result (the **1** in this case) and multiply by 2 once again.

0.250 * 2 = **0**.5 We keep the **0**

The whole number part of this new result is the *second* binary digit to the right of the point. We will continue this process until we get a zero as our decimal part or until we recognize an infinite repeating pattern.

0.50 * 2 = **1**.00 We keep the **1**

$9.625_{10} = 1001.\mathbf{101}_2$

Converting Decimal Fractions to Binary

109.5

Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

0.50 * 2 = **1**.00 We keep the **1**

We got a 0 for the decimal part so we are done…

$109.5_{10} = 1101101.\mathbf{1}_2$

Converting Decimal Fractions to Binary

$0.1 = .0001100110011..._2$

0.1 * 2 = 0.20	We keep the 0
0.2 * 2 = 0.40	We keep the 0
0.4 * 2 = 0.80	We keep the 0
0.8 * 2 = 1.60	We keep the 1
0.6 * 2 = 1.20	We keep the 1
0.2 * 2 = 0.40	We keep the 0
0.4 * 2 = 0.80	We keep the 0
0.8 * 2 = 1.60	We keep the 1
0.6 * 2 = 1.20	We keep the 1
0.2 * 2 = 0.40	We keep the 0
0.4 * 2 = 0.80	We keep the 0
0.8 * 2 = 1.60	We keep the 1
0.6 * 2 = 1.20	We keep the 1

…or until we recognize an infinite repeating pattern

Two's Complement

How to calculate the two's complement of a number

0000	= 0	1111	= -1
0001	= 1	1110	= -2
0010	= 2	1101	= -3
0011	= 3	1100	= -4
0100	= 4	1011	= -5
0101	= 5	1010	= -6
0110	= 6	1001	= -7
0111	= 7	1000	= -8

Take

5_{10} which is 0101_2

invert it 1010_2

and add 1

$$\begin{array}{r} 1010_2 \\ +\quad 1 \\ \hline 1011_2 \end{array}$$

-5_{10}

Take

3_{10} which is 0011_2

invert it 1100_2

and add 1

$$\begin{array}{r} 1100_2 \\ +\quad 1 \\ \hline 1101_2 \end{array}$$

-3_{10}

Two's Complement

$-12_{10} = 0100_2$

$-16_{10} = 11110000_2$

$-25_{10} = 11100111_2$

$-100_{10} = 10011100_2$

$-55_{10} = 11001001_2$

$-150_{10} = 01101010_2$

Floating Point

$-315{,}400_{10}$

0.3154×10^6	
sign	negative (1)
mantissa	3154
exponent	6

1101.101_2

0.1101101×2^4	
sign	positive (0)
mantissa	1101101
exponent	100_2

-0.00101_2

-0.101×2^{-2}	
sign	negative (1)
mantissa	101
exponent	10_2 (-2_{10})

1.001_2

0.1001×2^1	
sign	positive
mantissa	1001
exponent	1_2

Converting Decimal Fractions to Binary

$12.5_{10} = 1100.1_2$

$170.75_{10} = 10101010.11_2$

$100.2_{10} = 1100100.0011001100110011\ldots_2$

$9.25_{10} = 1001.01_2$

$4.625_{10} = 100.101_2$

$7.875_{10} = 111.111_2$

$4.375_{10} = 100.011_2$

$300.875_{10} = 100101100.111_2$