CSE 31
Computer Organization

Lecture 6 – C Memory Management (3)
Integer Representations

Announcement

- Lab #3 this week
  - Due at 11:59pm on the same day of your next lab
  - You must demo your submission to your TA within 14 days
- Reading assignment
  - Chapter 1.1 – 1.3 of zyBook
    - Do all Participation Activities in each section
    - Access through **CatCourses**
    - Due Wednesday (2/20) at 11:59pm

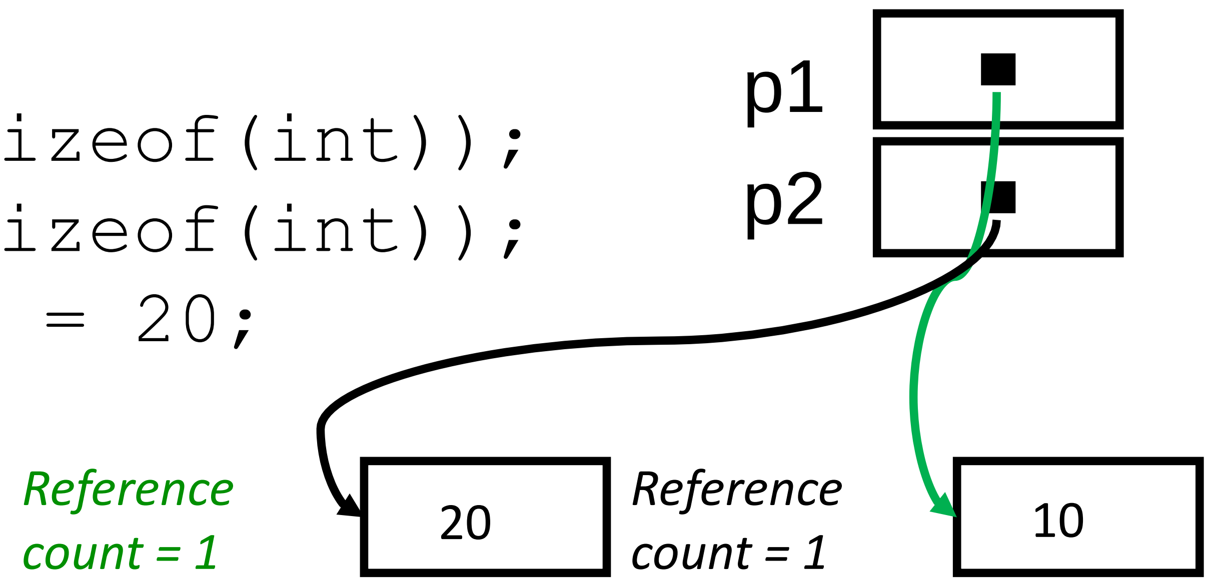
Automatic Memory Management

- Dynamically allocated memory is difficult to track
  - Why not track it automatically?
- If we can keep track of what memory is in use, we can reclaim everything else.
  - Unreachable memory is called garbage, the process of reclaiming it is called garbage collection.
- So how do we track what is in use?

Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

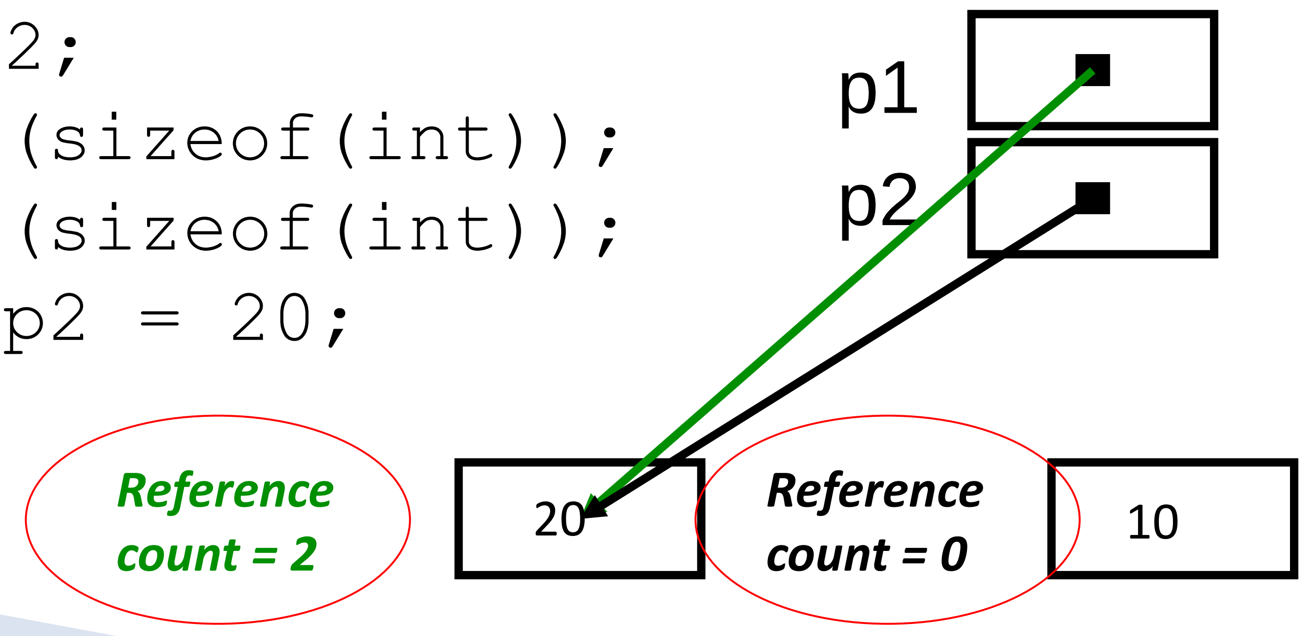
```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
p1 = p2;
```



Scheme 2: Mark and Sweep Garbage Collection

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B $\rightarrow$ A stores pointer to B
- Can start with the root set, perform a graph traversal, find all usable memory!
- 2 Phases:
  1. Mark used nodes
  2. Sweep free ones, returning list of free nodes

Mark and Sweep

- Graph traversal is relatively easy to implement recursively

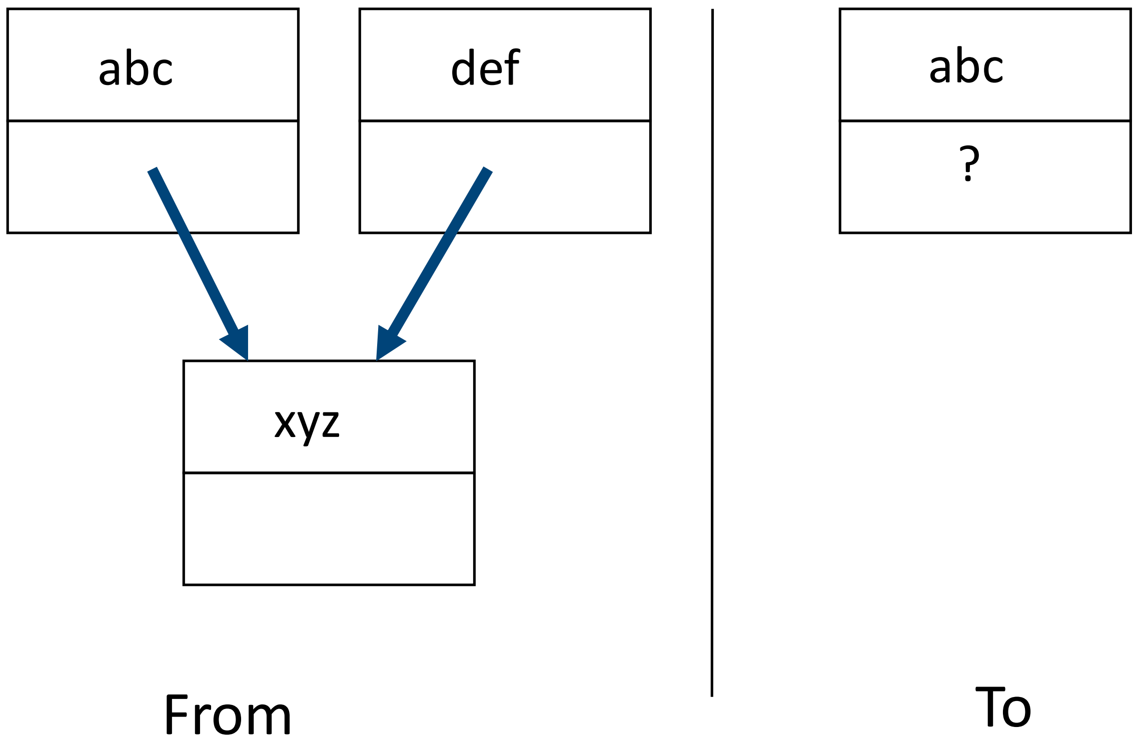
```
void traverse(struct graph_node *node) {
    /* visit this node */
    foreach child in node->children {
       traverse(child);
    }
}
```

- But with recursion, state is stored on the execution stack.
  - Garbage collection is invoked when not much memory left
- As before, we could traverse in constant space (by reversing pointers)

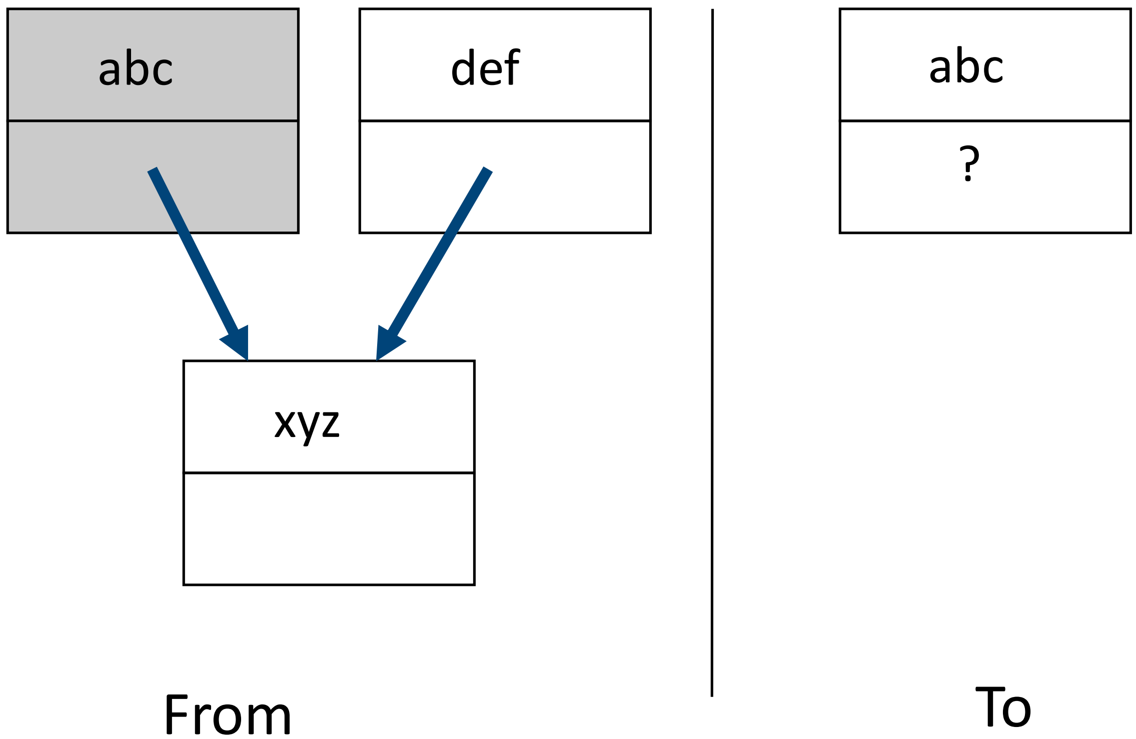
Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use “forwarding pointers” to keep consistency
  - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied

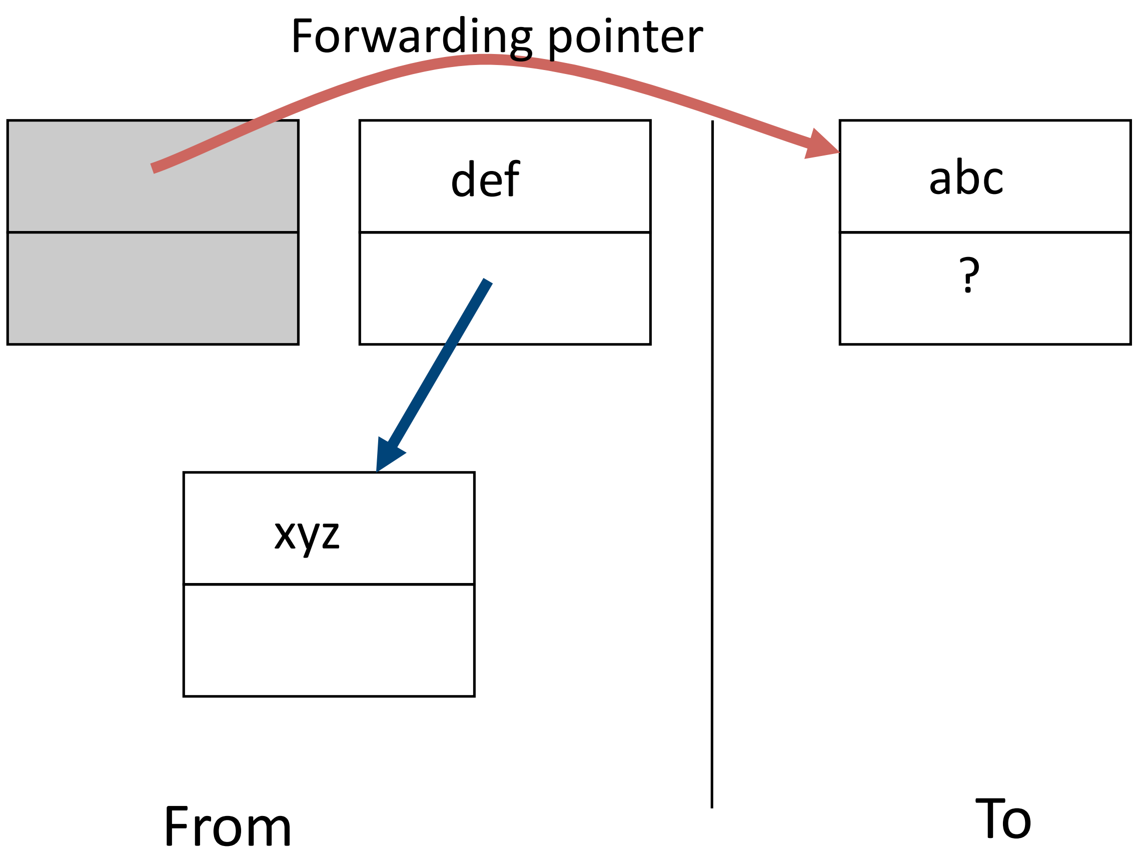
Forwarding Pointers: 1st copy "abc"



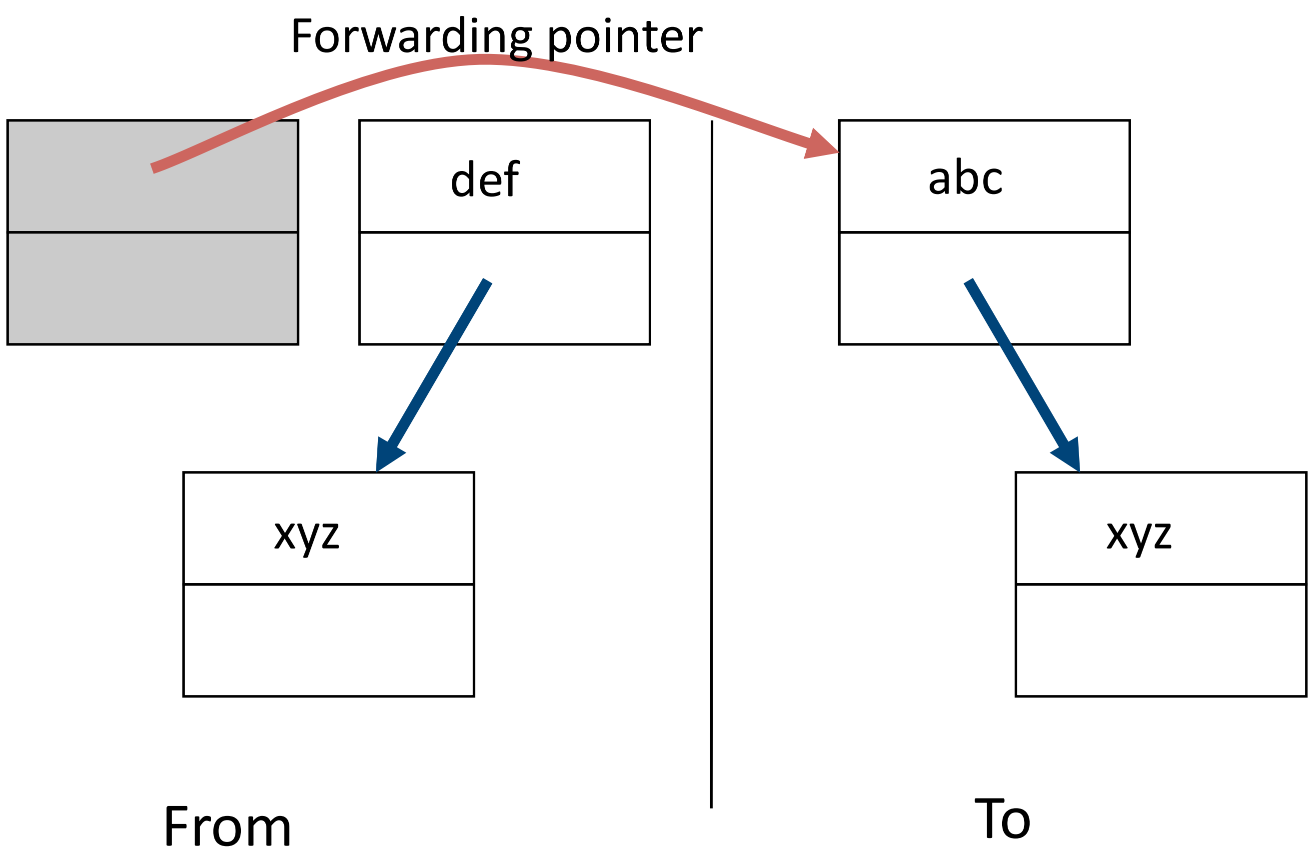
Forwarding Pointers: leave ptr to new abc



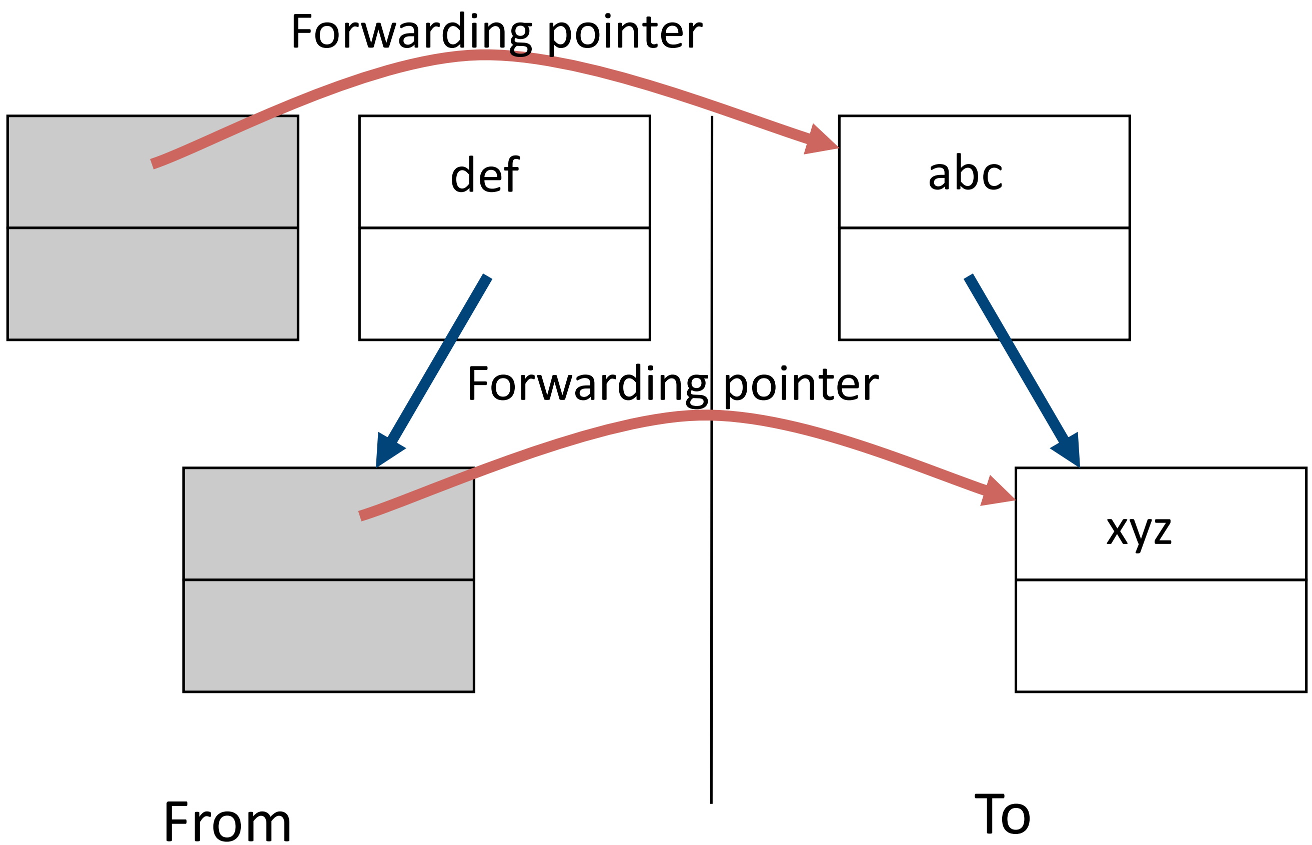
Forwarding Pointers : now copy "xyz"



Forwarding Pointers: leave ptr to new xyz

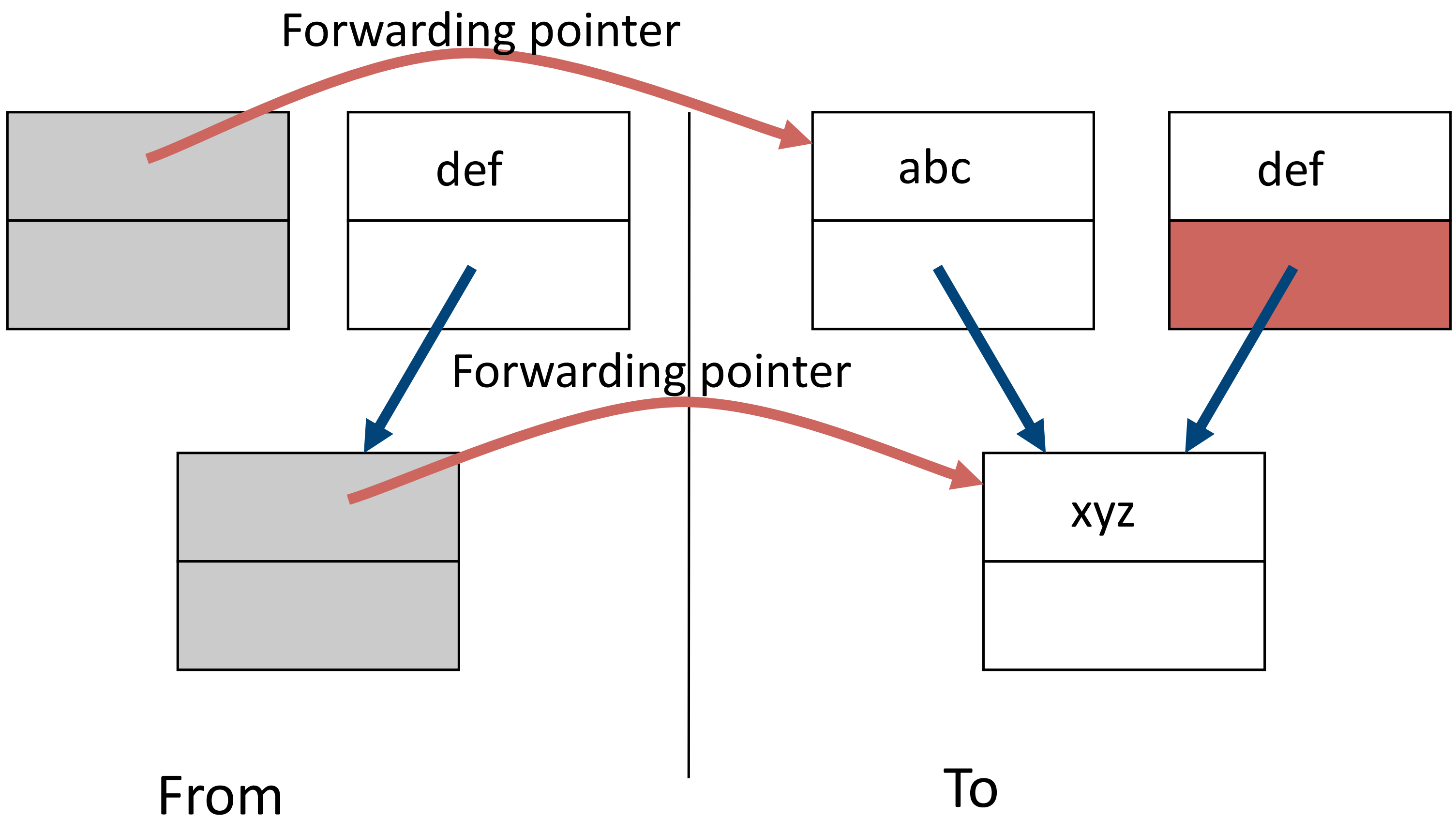


Forwarding Pointers: now copy “def”



Since xyz was already copied,
def uses xyz’s forwarding pointer
to find its new location

Forwarding Pointers



Since xyz was already copied,
def uses xyz's forwarding pointer
to find its new location

Summary

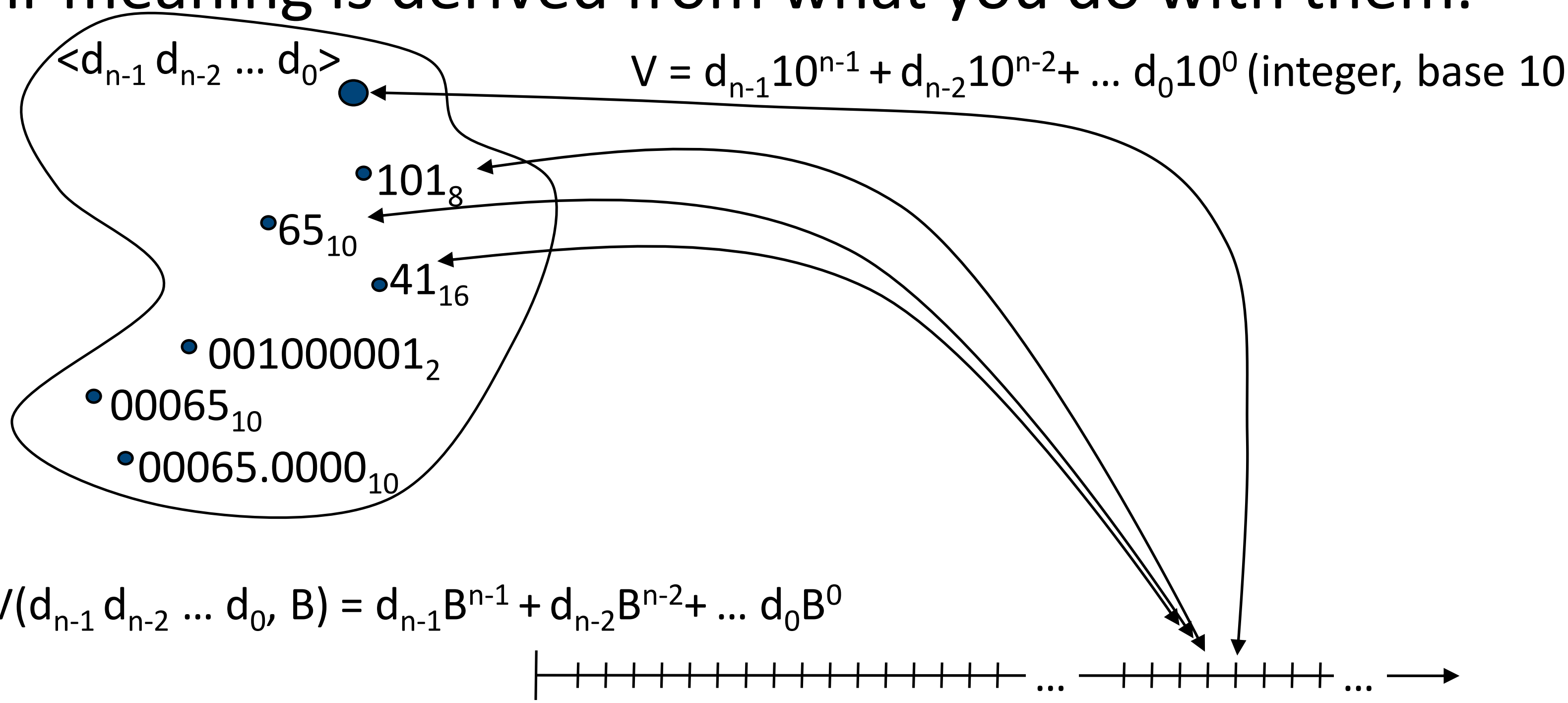
- Several techniques for managing heap via malloc and free: best-, first-, next-fit
  - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  - Each technique has strengths and weaknesses, none is definitively best
- Automatic memory management relieves programmer from managing memory.
  - All require help from language and compiler
  - Reference Count: not for circular structures
  - Mark and Sweep: complicated and slow, works
  - Copying: Divides memory to copy good stuff

Number Representations

- What do these numbers mean?
  - 101
  - 0101
- Depends on what representation!

Representation and Meaning

- Objects are represented as collections of symbols (bits, digits)
- Their meaning is derived from what you do with them.



Representation (how many bits?)

- Characters?
  - 26 letters → 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation → 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages → 8,16,32 bits ("Unicode") `www.unicode.com`
- Logical values?
  - 0 → False, 1 → True
- Colors? Ex: *Red (00)* *Green (01)* *Blue (11)*
- Remember: N bits → at most 2^N things

How many bits to represent π ?

a) 1
b) 9 (π = 3.14, so that's 011 . 001 100)
c) 64 (Since modern computers are 64-bit machines)
d) Every bit the machine has!
e) ∞

We are going to learn how to represent floating point numbers later!

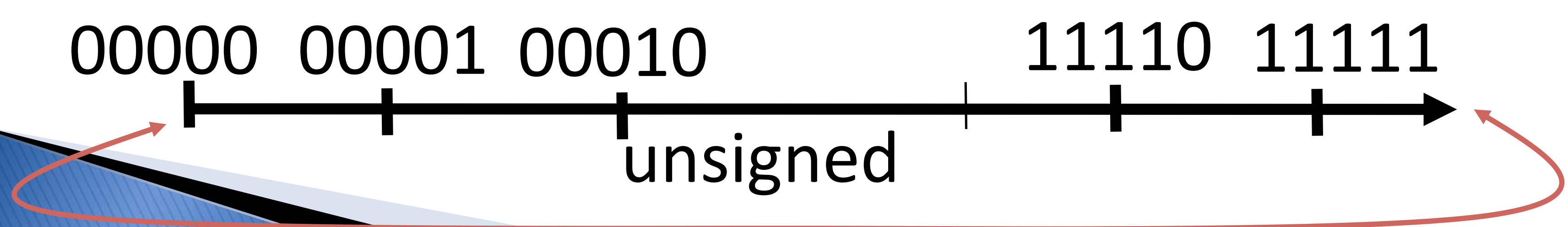
What to do with representations of numbers?

- Just what we do with numbers!
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them
- Example: 10 + 7 = 17
  - …so simple to add in binary that we can build circuits to do it!
  - subtraction just as you would in decimal
  - Comparison: How do you tell if X > Y ?

```
    1  1
    1  0  1  0
+   0  1  1  1
-------------------------
 1  0  0  0  1
```

What if too big?

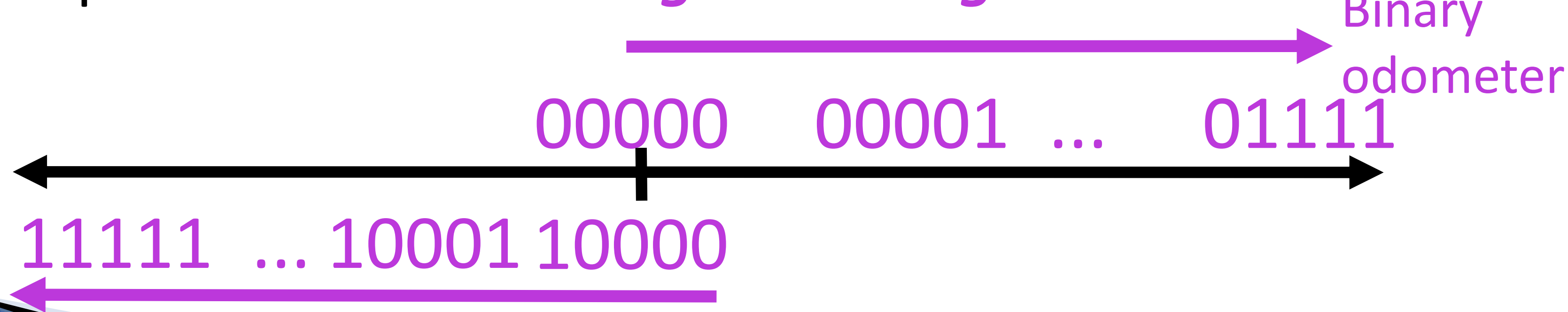
- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called “numerals”
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don’t normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, ***overflow*** is said to have occurred.



Negative Numbers

- So far, ***unsigned numbers***

- Obvious solution: define leftmost bit to be sign!
  - 0 → + , 1 → –
  - Rest of bits can be numerical value of number
- Representation called ***sign and magnitude***

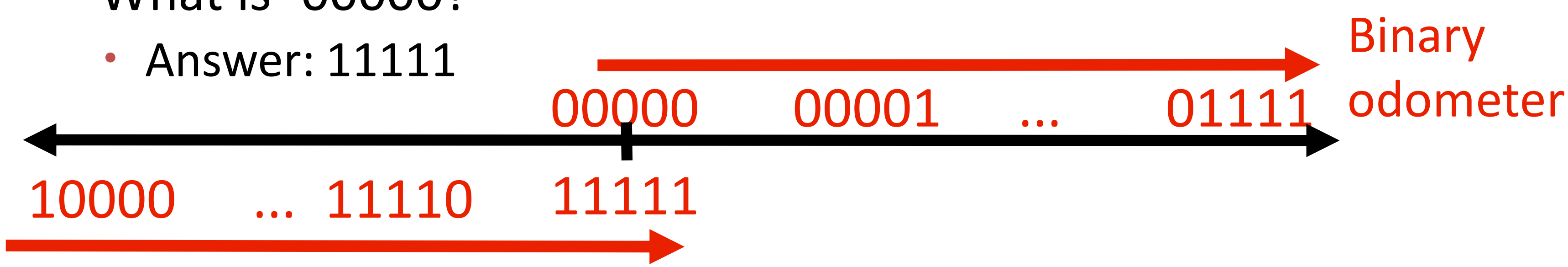


Shortcomings of Sign Magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = -0_{ten}
  - What would two 0s mean for programming?
- Also, incrementing "binary odometer", sometimes increases values, and sometimes decreases!
- Therefore sign and magnitude abandoned

Another try

- Complement the bits
  - Example: $7_{10} = 00111_2 \quad -7_{10} = 11000_2$
  - Called ***One's Complement***
  - Note: positive numbers have leading 0s, negative numbers have leadings 1s.
  - What is -00000?
    - Answer: 11111



  - How many positive numbers in N bits? 2^{N-1}
  - How many negative numbers? 2^{N-1}

Shortcomings of One's complement?

- Arithmetic is less complicate than sign & magnitude.
- Still two zeros
  - 0x00000000 = $+0_{ten}$
  - 0xFFFFFFFF = -0_{ten}
- Although used for a while on some computer products, one's complement was eventually abandoned because another solution was better.

Standard Negative # Representation

- Problem is the negative mappings "overlap" with the positive ones (the two 0s). Want to shift the negative mappings left by one.
  - Solution! For negative numbers, complement, then add 1 to the result
- As with sign and magnitude, & one's complement, leading 0s → positive, leading 1s → negative
  - 000000...xxx is ≥ 0, 111111...xxx is < 0
  - except 1...1111 is -1, not -0
- This representation is ***Two's Complement***
- This makes the hardware simple!

In C: short, int, long long, intN_t (C99) are all signed integers.

Two's Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:

  $d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$

- Example: 1101_{two}

  $= 1x\text{-}(2^3) + 1x2^2 + 0x2^1 + 1x2^0$

  $= -2^3 + 2^2 + 0 + 2^0$

  $= -8 + 4 + 0 + 1$

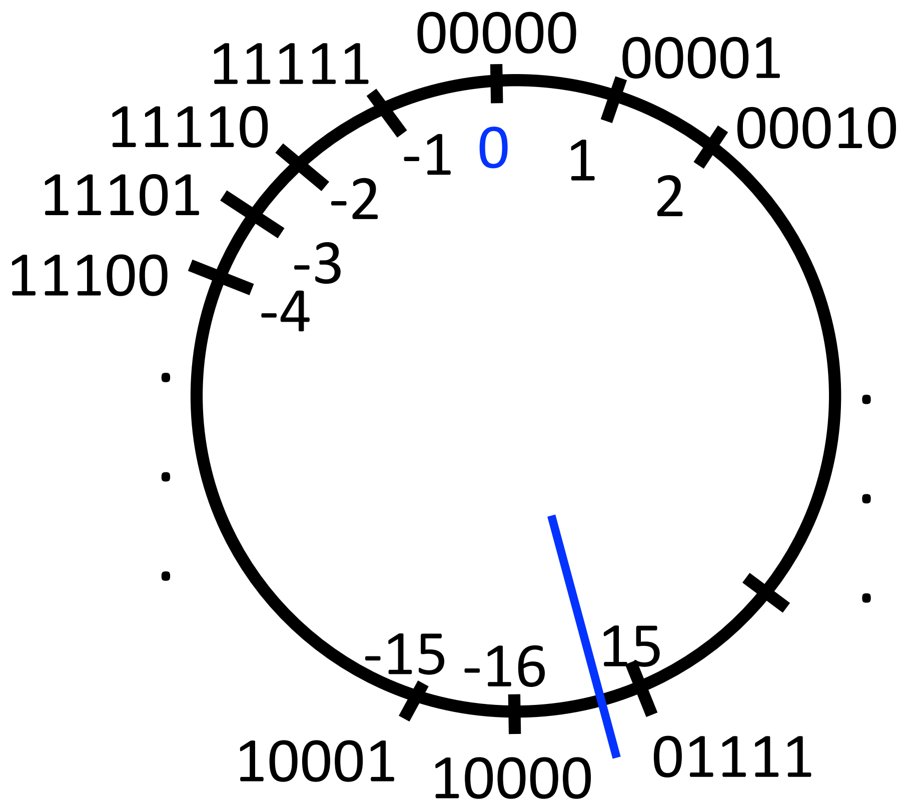
  $= -8 + 5$

  $= -3_{ten}$

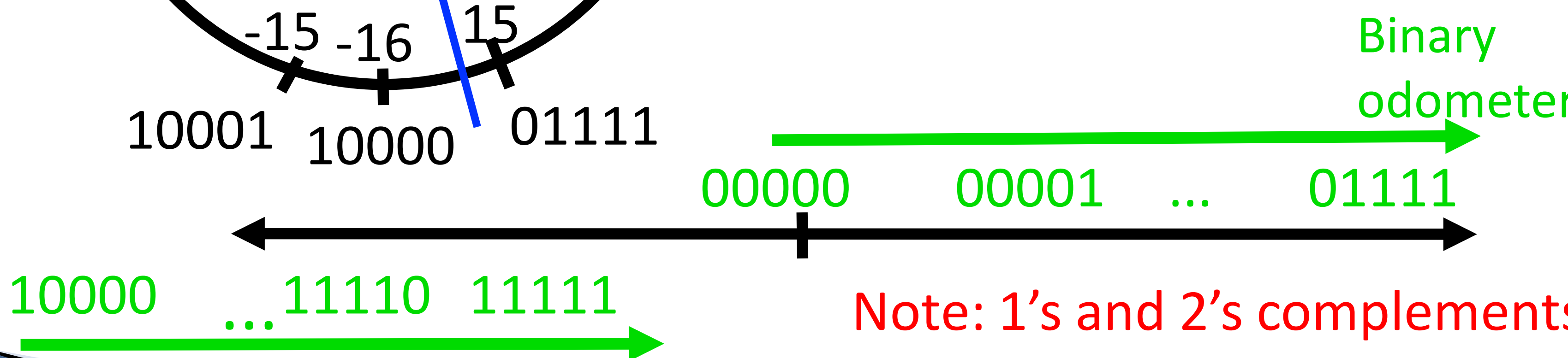
Example: -3 to +3 to -3:

x :	1101_{two}	(-3)
x':	0010_{two}	
+1:	0011_{two}	(3)
()':	1100_{two}	
+1:	1101_{two}	(-3)

2’s Complement Number “line”: N = 5

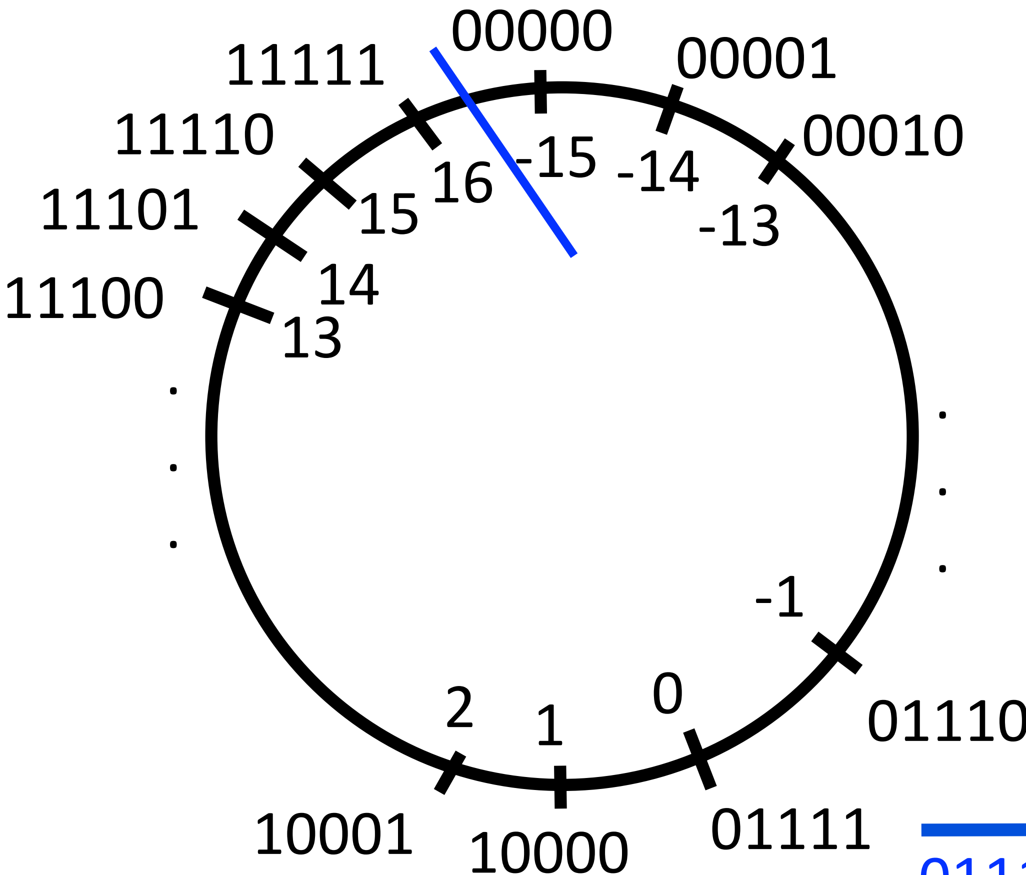


- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?
  - $2^{N-1} – 1$

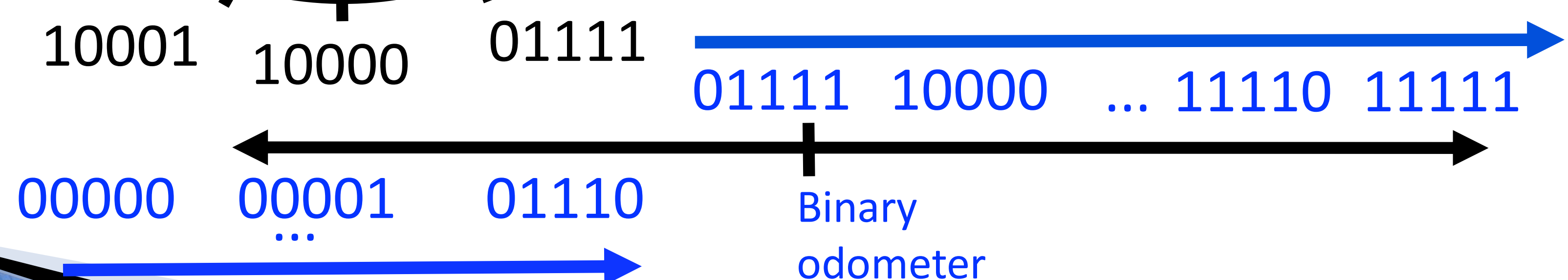


Note: 1’s and 2’s complements are used to represent negative numbers only!

Bias Encoding: N = 5 (bias = -15)



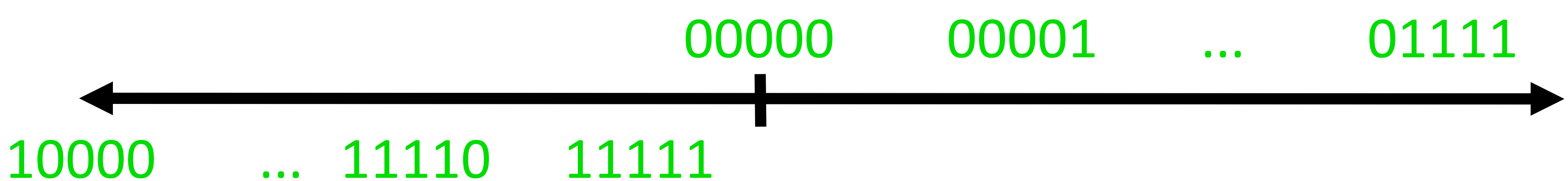
- Want 00… to represent the smallest number
- value = unsigned - bias
- Bias for N bits = $2^{N-1} – 1$
- one zero
- how many positives?
  - 2^{N-1}
  - (more than 2’s complement)



Summary

- We represent "things" in computers as particular bit patterns:
  - N bits → 2^N things
- Different integer encodings have different benefits; 1s complement and sign/mag have most problems.
- unsigned (C99's `uintN_t`) :

- 2's complement (C99's `intN_t`): universal, learn it!



- Overflow: numbers ∞ ; computers finite → errors!

Floating Point Numbers

- How best to represent: 2.75_{10}?
  - 2s Complement (but shift binary pt)
  - Bias (but shift binary pt)
  - Combination of 2 encodings
  - Combination of 3 encodings
  - We can't

Shifting ***binary point*** means "divide" number by some power of 2.

$11_{10} = 1011.0_2 \rightarrow \quad 10.110_2 = (11/4)_{10} = 2.75_{10}$

Representation of Fractions

"Binary Point" like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:

xx.yyyy

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

$10.1010_2 = 1x2^1 + 1x2^{-1} + 1x2^{-3} = 2.625_{10}$

If we assume "**fixed binary point**", range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i}	
0.	1.0	1
1.	0.5	1/2
2.	0.25	1/4
3.	0.125	1/8
4.	0.0625	1/16
5.	0.03125	1/32
6.	0.015625	
7.	0.0078125	
8.	0.00390625	
9.	0.001953125	
10.	0.0009765625	
11.	0.00048828125	
12.	0.000244140625	
13.	0.0001220703125	
14.	0.00006103515625	
15.	0.000030517578125	