# CS50 Section 5
# Somewhere in Between

Annaleah Ernst, TF

# Agenda

- Quick recap
  - Resources
  - Bitwise Operators
  - Structs
- Linked Lists
- Hash Tables
- Trees
- Tries
- Time allowing:
  - Stacks
  - Queues
  - Data Compression
    - Huffman Coding

# Resources

- Static resources
  - CS50 Study - study.cs50.net
  - CS50 Manual - manual.cs50.net
  - Reference50 - reference.cs50.net
  - Style Guide - manual.cs50.net/style/
    - style50
  - Walkthroughs && Shorts
  - man
  - debug50
  - valgrind –leak-check=full
- Dynamic Resources
  - CS50 Discuss - cs50.harvard.edu/discuss
  - Harvard Slack - harvard.slack.com/signup
  - Office hours
  - Classmates
  - Me!

# Bitwise operators

- Allow us to manipulate individual bits
- & AND
  - gives 1 if both arguments are 1
- | OR
  - gives 1 if at least one argument is 1
- ~ NOT
  - flips the given bit
- ^ XOR
  - gives 1 if exactly 1 argument is 1
- << left shift
- >> right shift
  - shifts a bit the given number of places in the given direction

# You turn! Bitwise

- 0 & 1
  - 0 & 1 = 0
- 1 & 1
  - 1 & 1 = 1
- 0 | 1
  - 0 |1 = 1
- 1 | 1
  - 1 | 1 = 1
- 1010 & 0101
  - 0000

- 0 ^ 1
  - 0 ^ 1 = 1
- 1 ^ 1
  - 1 ^ 1 = 0
- ~0
  - ~0 = 1
- ~1
  - ~1 = 0
- 1010 | 0101
  - 1111

# Pointers

- Just variables containing addresses!
- They point to other values
- To go to what they point to, use the * operator
  - dereferencing
- To get the address of a variable, use the & operator
  - referencing

# Your turn! – buggy_swap.c

- I’ve implemented a swap function…but it doesn’t seem to be working!
- When I call this function, my variable up in main remain unswapped!
- How can I change this function so that it actually swaps my variables?

```c
void buggy_swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}
```

# Structs

- Create our own special data type
- Declare using the struct name as the variable type
- Access using the . operator if we have it directly, or the arrow operator if we have a pointer to a struct

```
typedef struct
{
    int id;
    string name;
} student;
```

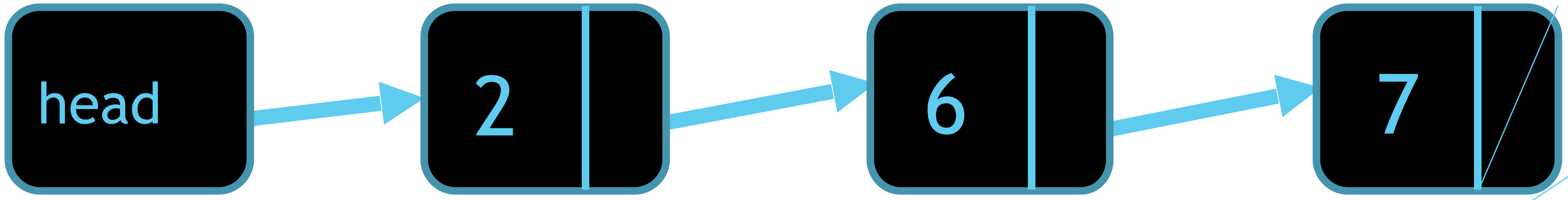
```
// direct assignment
student my_student;
my_student.name = "Anna";
my_student.id = 88;
```

```
// and using a pointer
student *stu_ptr = malloc(sizeof(student));
stu_ptr->name = "Rob";
stu_ptr->id = 50;

// this will do the same thing
(*stu_ptr).name = "Rob";
(*stu_ptr).id = 50;
```

# Linked Lists

- Uses a recursive datatype
  - A struct that points to another version of itself
- Characteristics:
  - The head – a pointer to the first element in the list
  - Each element contains a value and a pointer to next element
  - The last element points to NULL




- *boxes not drawn to scale!

# Linked List - Nodes

- Made of special structs that contain pointers to the next element
- Traditionally, we call these structs “nodes”
- Here’s an example of an int linked list, but note that linked lists can be of any data type (how?)

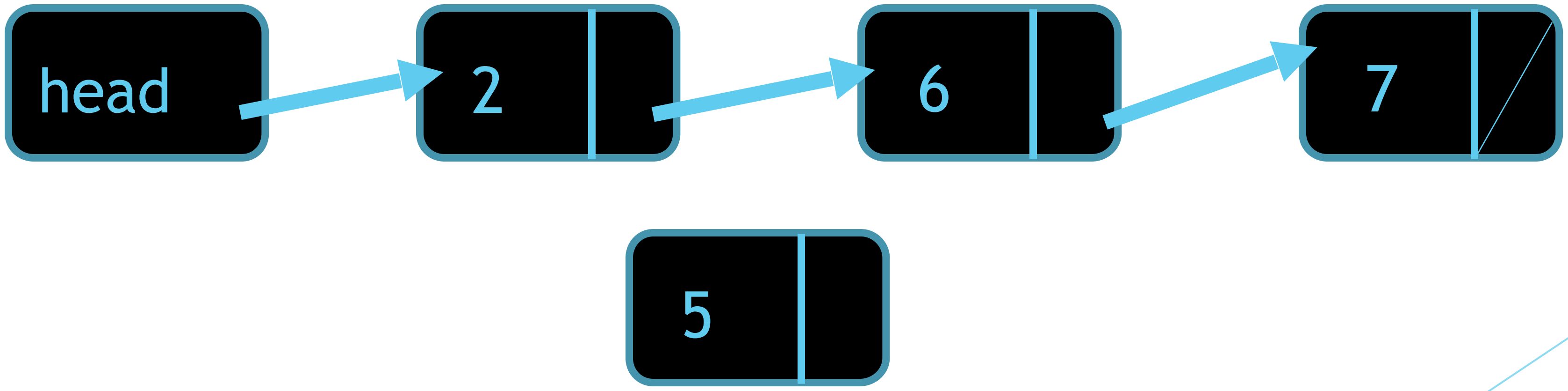
```c
typedef struct node
{
    int n;
    struct node *next;
} node;
```

```c
// what will these lines print?
node new_node;
new_node.n = 1;
printf("%i\n", new_node.n);

node* ptr_node = &new_node;
printf("%i\n", (*ptr_node).n);
printf("%i\n", ptr->n);
```

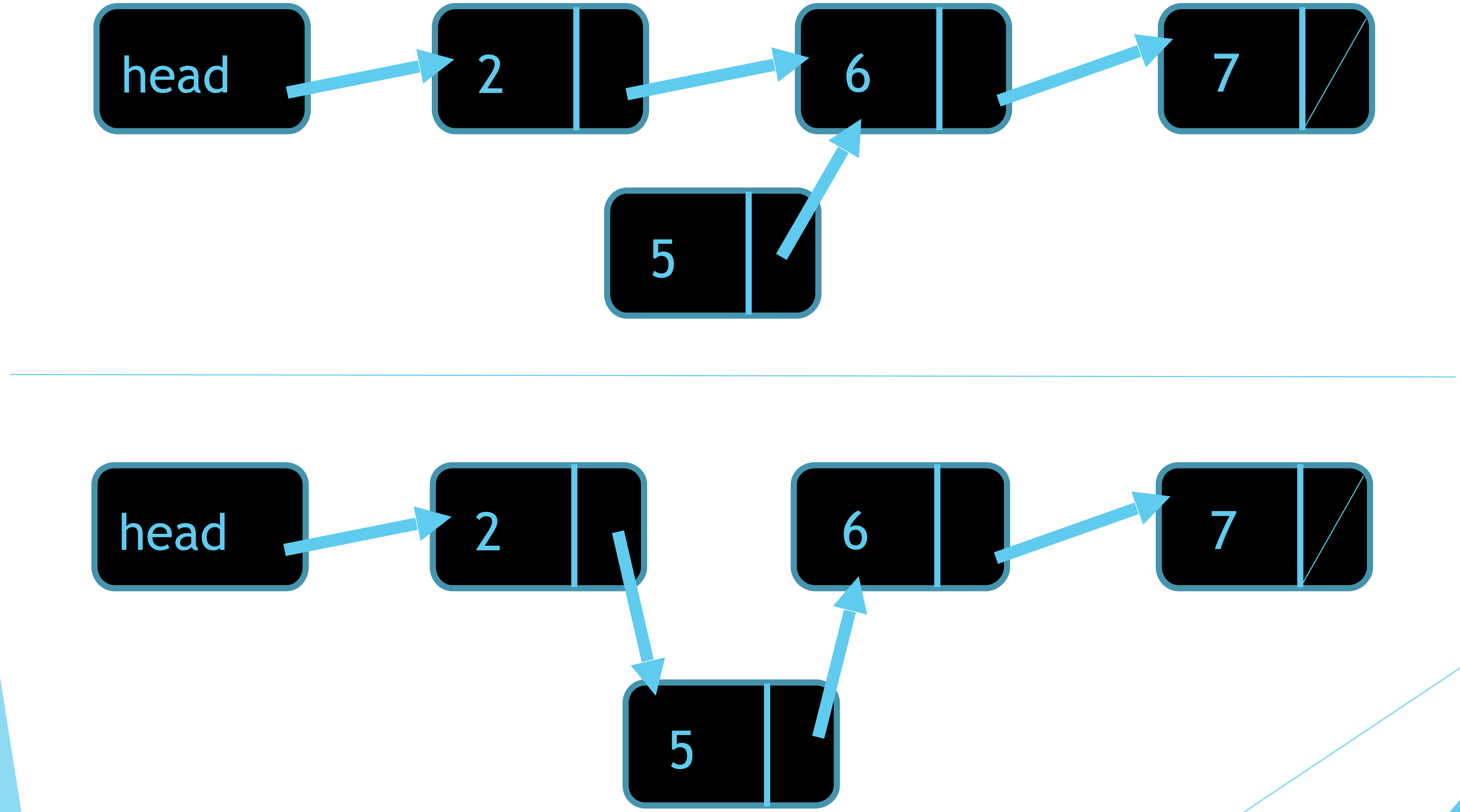
# Linked List - Insert

- When inserting, order matters!
  - Eg, the order in which we change what pointers are pointing to
  - Very easy to accidentally orphan the list
- How do we insert?



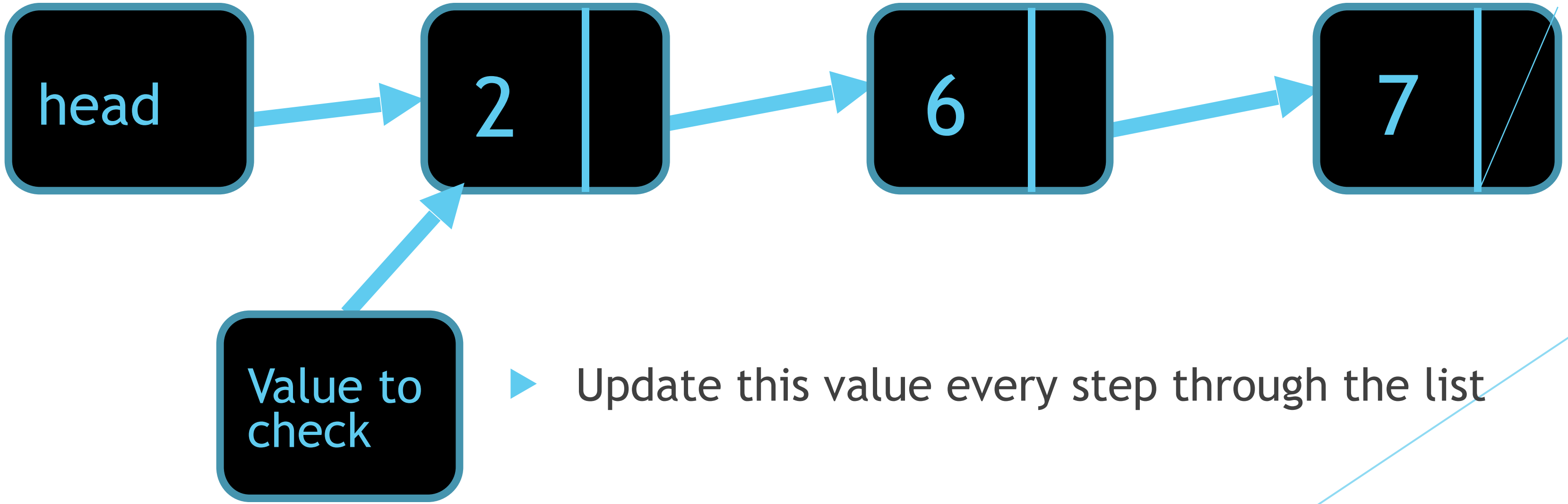

- *boxes not drawn to scale!

# Linked List – Insert (sorted)



# Linked List - Search

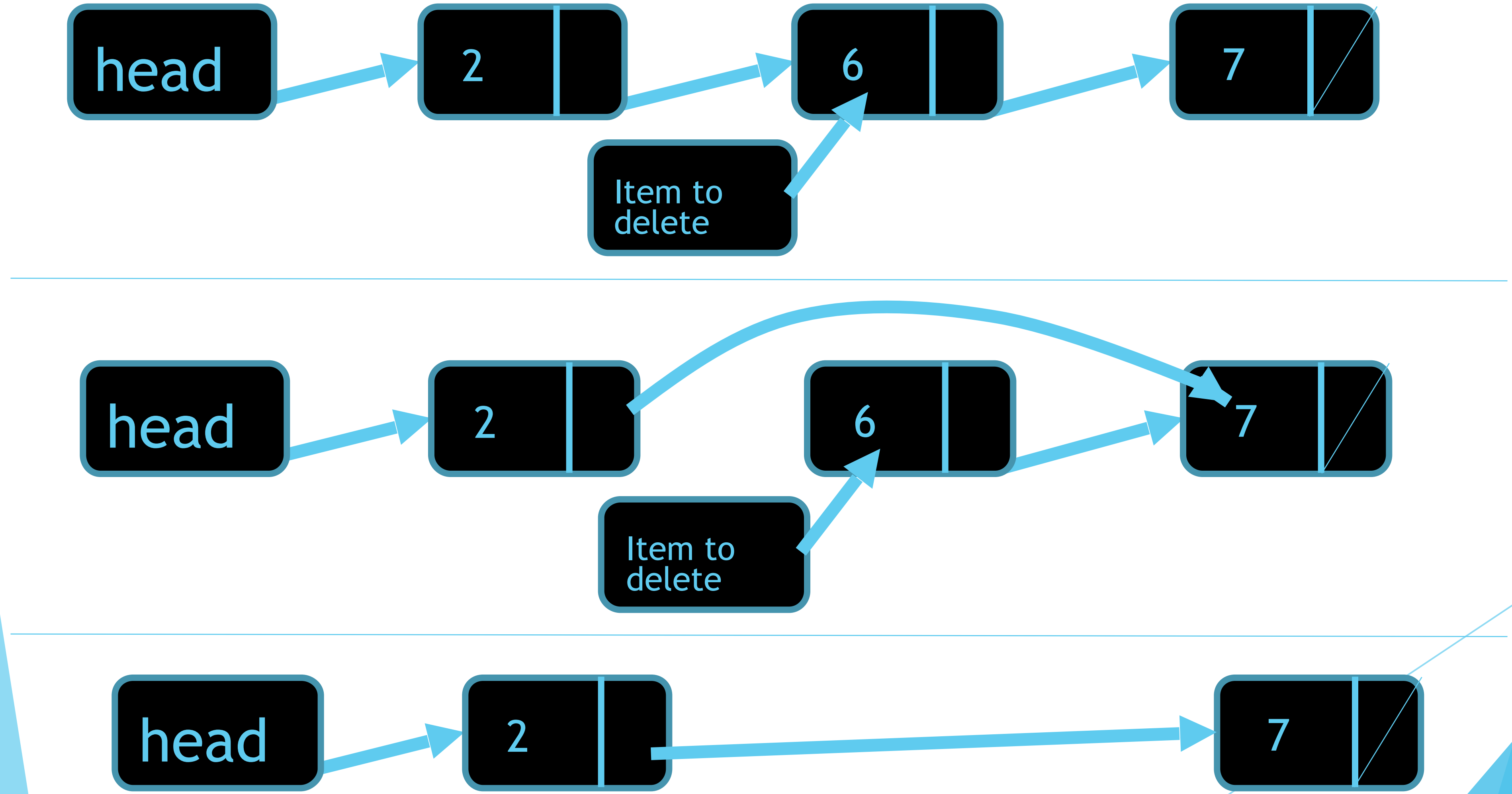
- Go through the list node by node and compare to value you're looking for
  - What would the difference between sorted and unsorted be?
  - How can you tell you've gotten to the end?
    - Next points to NULL




- Update this value every step through the list

# Linked List - delete

- Let’s say we want to delete 6…



# You turn! linked.c

- Write a function that…
  - Prints out the contents of an integer linked list, from head to end

# You turn! linked.c

- Write a function that inserts a value into a sorted linked list
  - ```
    bool insert_sorted(int value, node *list);
    ```
  - Keep it sorted from largest to smallest
  - Don’t insert duplicates
  - Let the user know if the insert was successful
  - Assume a global variable called node* head to keep track of the head of the list
    ```

# linked.c - psuedocode

- Create a new node (malloc space, put number in node)
- Create prev node pointer and current node pointer
- Go through list
  - If value < this node value
    - Insert before and return true
  - If value > this node value
    - Update pointers
    - Go to the next node
  - If value == node
    - Free and then return false

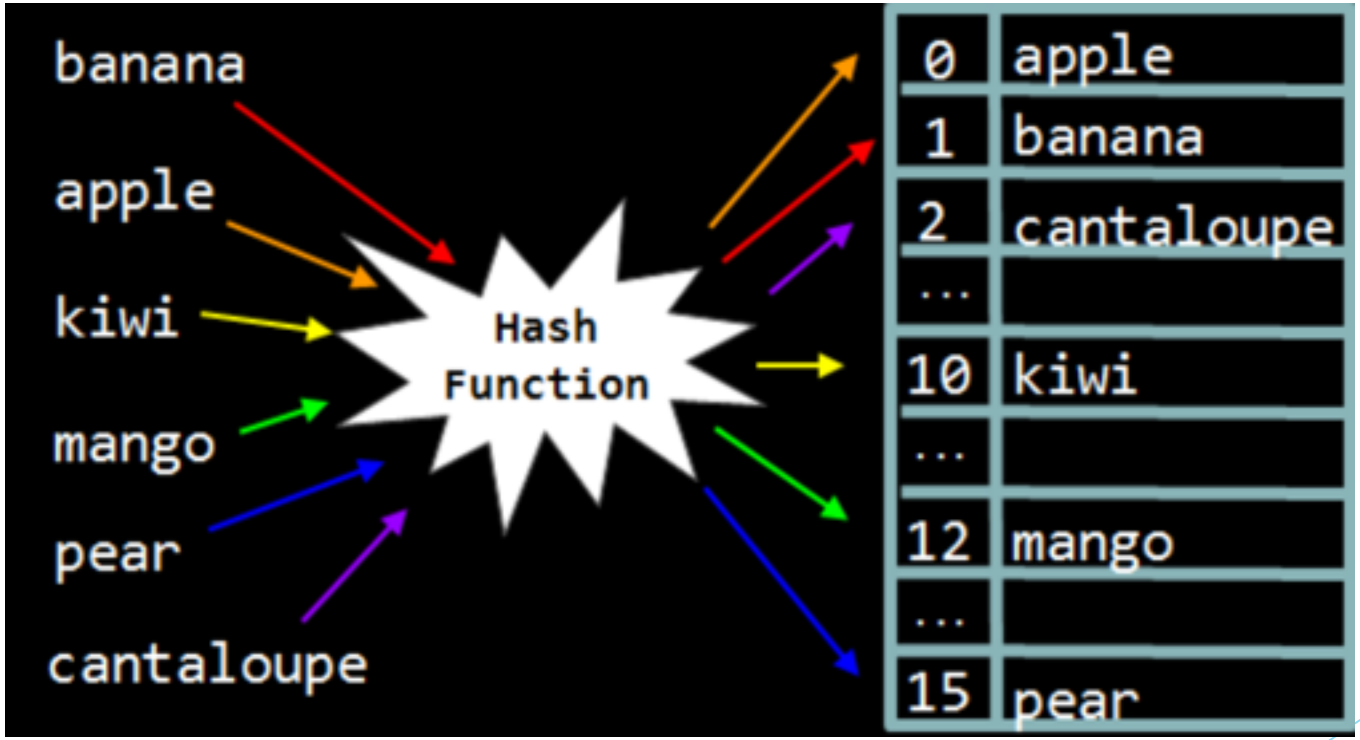
# Doubly linked lists

- So far, we’ve only been talking about singly linked lists
  - Next only points in one direction
- We can also make doubly linked lists
  - Keep track of forward and previous
- Operations on doubly linked list similar to on singly linked lists, but now there are two pointers to update!

```
// doubly linked list node
typedef struct node
{
    int n;
    struct node *next;
    struct node *prev;
} node;
```

# Hash Table

- data structure where the position of each element is decided by a hash function
  - A function that converts the input data into an integer
- What makes a good hash function?



# Hash Tables

- At it’s core, just an array and a function
- Function hashes input and assigns them an index
- Placed into table based on input
- YOU DO NOT HAVE TO WRITE YOUR OWN HASH FUNCTION
- It’s easy to write a hash function…very very difficult to write a good one
  - Low collisions, unique keys
- Find one online and treat it as a black box – just cite where it’s coming from
  - I’ll send out some options
- Ideally, a hash table will have very low collisions

# Hash Tables – the Hash function

- I repeat, YOU DO NOT HAVE TO WRITE YOUR OWN, just cite where you got it
- Any function that accepts a `char*` and returns an `int` can be used as a hash function
  - Ex, hashing on the first letter of a word

```c
// a really bad hash function (high collision)
int hash_function(char *word)
{
    // hash based on first letter of the string
    int hash = toupper(word[0] - 'A');
    return hash % MAX_HASH_LEN;
}
```

# Hash Tables – the Hash function

- In this case, you don’t have to understand it to use it
- Here’s an example of a professional hash function with high efficiency:

```c
/*
 * djb2 hash function for hashing the values in dictionary
 * http://www.cse.yorku.ca/~oz/hash.html
 */
unsigned long hash_function(const char *str)
{
   unsigned long hash = 5381;
    int c;

    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash % MAX_HASH_LEN;
}
```

- Feel free to use any hash functions at [http://www.cse.yorku.ca/~oz/hash.html](http://www.cse.yorku.ca/~oz/hash.html) as long as you cite your sources!

# Hash Tables – Resolving Collisions

- In an ideal world, there would be no collisions
  - In this case, we'd get constant lookup time
- A couple options:
  - Chain items
  - Probe for open index
- For chaining:
  - each bucket of the array is actually another data structure
    - Eg, a linked list
  - Ideally, we still want a function with low collisions so that we can take advantage of the speed of a hash table

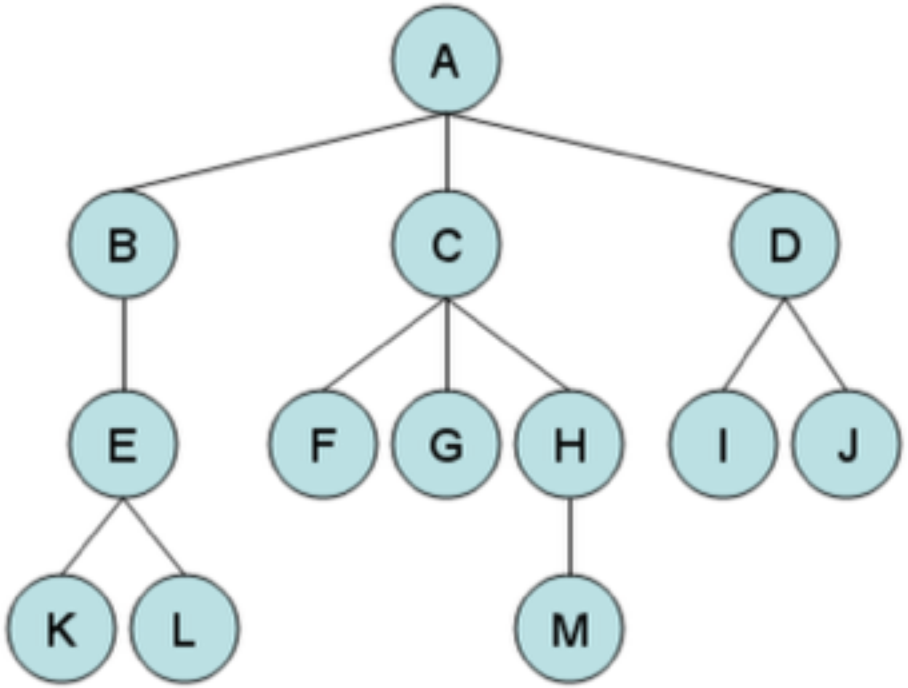
# Hash Tables – lists as Buckets

- To resolve collisions, we can just make an array of linked lists
- ```
  node *hash_table[MAX_HASH_LEN];
  ```
- Now we have an array of node pointers
- When we have a new key to insert
  - Create a node* and store the key within
  - We hash it (call the hash function on it)
  - Go to that index of the array
  - If it’s empty, just put in the node
  - If it’s full insert it into the linked list
    - Where should we insert it into the list?
  ```

# Big O – Hash Tables and Linked lists

- What's the big O run time of insertion/deletion in a hash table?
  - O(n) (Note: the runtime is more like n/k, where k is the number of buckets in the hash table. While asymptotically this is the same, in the real world, it runs better)
  - In a perfect world, both these operations would be O(1)!
- What's the big O of inserting into an unsorted linked list?
  - O(1)
- What's the big O of finding a value in a linked list?
  - O(n)

# Tree

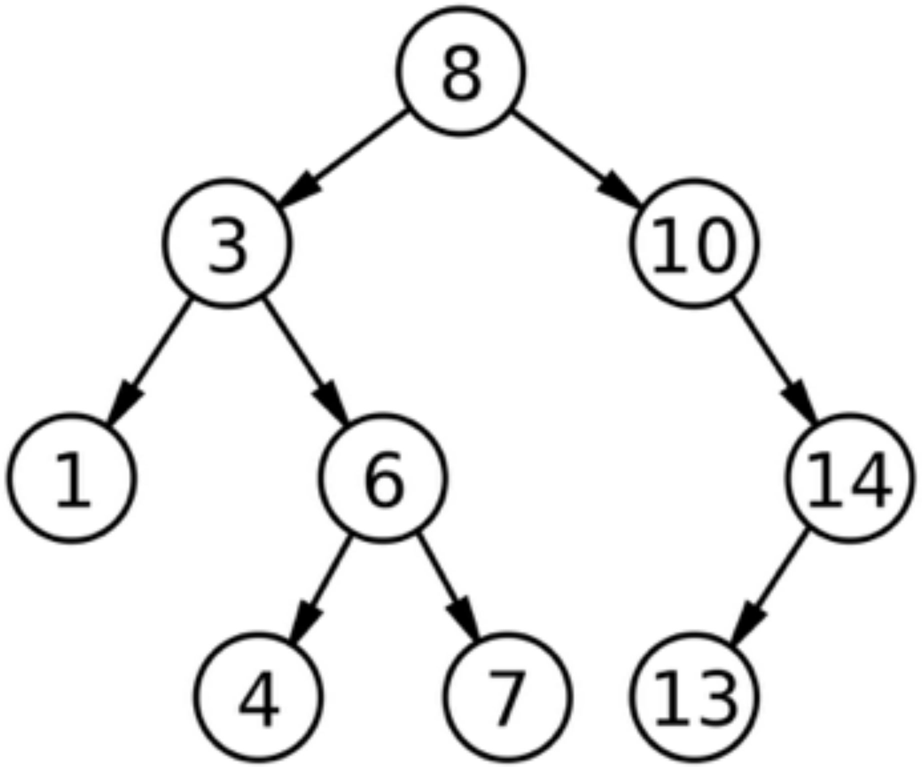



https://www.cpp.edu/~ftang/courses/CS241/notes/images/trees/tree1.bmp

- Trees are hierarchically arranged data structures
  - Nodes have parents and children
    - Nodes in trees can have any number of children
  - Top of the tree is called the root
  - Bottom of the tree (pointing to nothing) are leaves

# Binary Tree

- Ideally we want balanced trees – trees that have the same number of levels to the leaves.
- Nodes in binary trees have at most two children
- Here we have a binary search tree. How long would it take to find the value 22?




https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Binary_search_tree.svg/2000px-Binary_search_tree.svg.png

# Tries

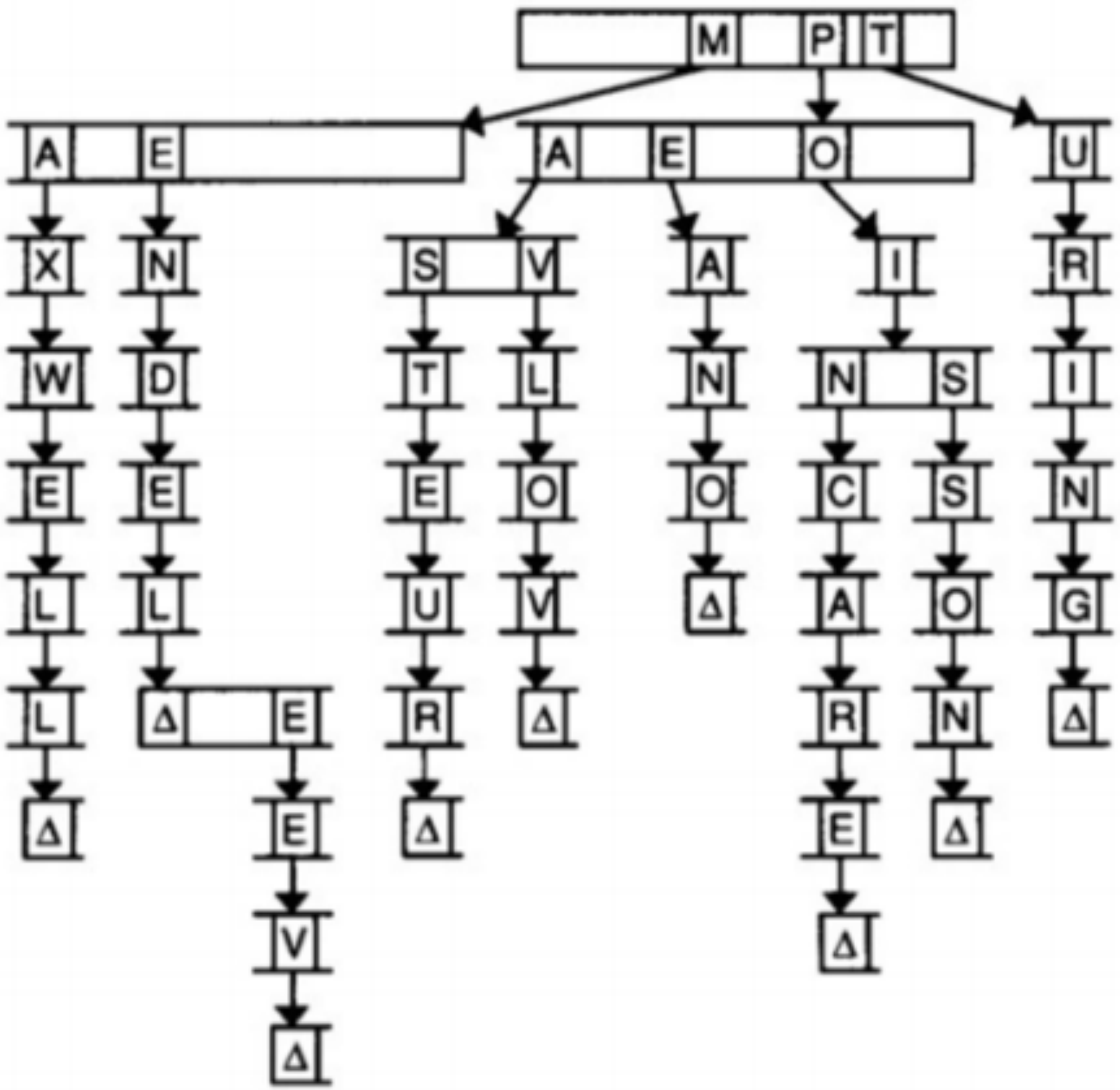
- All tries are trees, but not all trees are tries!
- Capitalizes on near constant look up time of arrays – very fast
- Trade-off –HUGE amount of space needed
- Instead of simply creating a pointer to a new node, it creates and array of pointers:

```
typedef struct node
{
    // marker for end of word
    bool is_word;

    // array of node pointers to our children
    struct node *children[LEN_ALPHA];
} node;
```

# Tries

- pro: provide constant time lookup (in theory)
- con: use large amounts of memory!



# Stacks

- first-in, last-out (FILO)
- elements are successively pushed down as other items are added
- elements are pushed on and popped off
- keep track of both the size and capacity
  - you need not keep track of capacity if you use a linked list rather than an array

# Queues

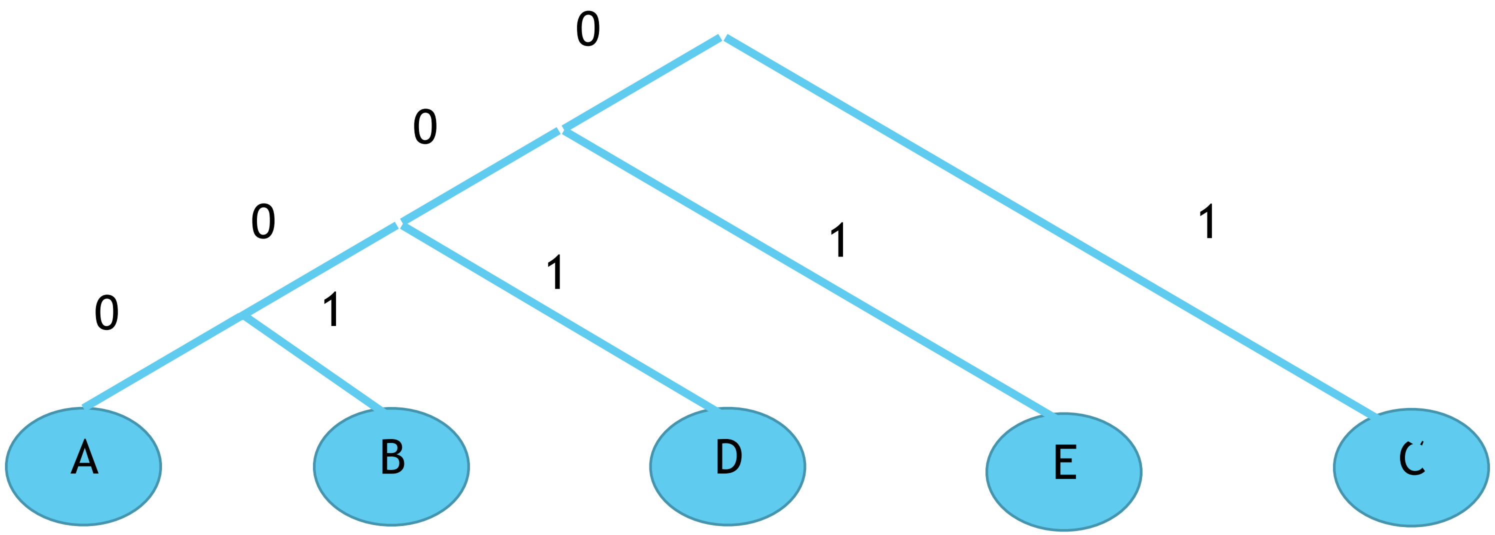
- first-in, first-out (FIFO)
- picture a line!
- elements are enqueued and dequeued
- keep track of the size, capacity, and head
  - you need not keep track of capacity if you use a linked list rather than an array

# Huffman Coding

- Data compression
- Typically used for text files
- Chars take 8 bits
  - but if we know the frequency with which letters appear, perhaps we can do better
- We can represent the most frequently used chars as less bits!
- Let’s do an example…

# Huffman Coding

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| ***Frequency*** | ***0.1*** | ***0.1*** | ***0.35*** | ***0.15*** | ***0.3*** |




| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| ***Encoding*** | ***0000*** | ***0001*** | ***1*** | ***001*** | ***01*** |

# Huffman Coding

- All Huffman coded files must adhere to the *prefix property*
  - No Huffman code for any character may be the prefix of another character's code
  - eg, we can't have both A = 1 and C = 10 since decoding will be ambiguous!
- In order to decompress, we need a *frequency table* as part of the file
  - This lets us figure out which codes go to which letters
- Cons of encoding
  - If our text file contains lot's of diverse characters, Huffman coding might not actually shrink the file size
  - We have to decode every time we want to use the file