CS50 Test Review Somewhere in Between

Annaleah Ernst, TF

Non-Exhaustive Topics

- Binary. ASCII. Algorithms. Pseudocode. Source code. Compiler. Object code. Scratch. Statements. Boolean expressions. Conditions. Loops. Variables. Functions. Arrays. Threads. Events.
- Linux. C. Compiling. Libraries. Types. Standard output.
- Casting. Imprecision. Switches. Scope. Strings. Arrays. Cryptography.
- Command-line arguments. Searching. Sorting. Bubble sort. Selection sort. Insertion sort. O. Ω.Θ. Recursion. Merge Sort.
- Stack. Debugging. File I/O. Hexadecimal. Strings. Pointers. Dynamic memory allocation.
- Heap. Buffer overflow. Linked lists.
- Hash tables. Tries. Trees. Stacks. Queues.

Non-Exhaustive Topics

- Data Types and Sizes
- Binary and Hex
- ASCII (Math)
- Floating Point Imprecision
- Pointers
- Scope
- Switches
- Pointers
- Memory
- Recursion
- Structs
- Linked Lists
- Hash Tables
- Trees and Tries
- Files I/O

Data Types and Sizes

- `char`
  - 1 byte
- `int`
  - 4 bytes
- `long long`
  - 8 bytes
- `float`
  - 4 bytes
- `double`
  - 8 bytes
- `<type> *`
  - `pointer`
  - 8 bytes (64 bit architecture)

Binary and Hex

- What is binary notation?
  - Base 2, 1 0
- What about hexadecimal?
  - Base 16 (denoted by 0x at the front)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f, where f = decimal 15
- What do we mean by different bases?
  - Ex: base 10: What is 356 equivalent to?
    - $3 * 10^2 + 5 * 10^1 + 6 * 10^0$
  - Same thing applies to binary (base 2) and hex (base sixteen)
    - <Value at place> * <base>^{<place>} + … + <Value at place> * <base>^{<1>}

Your turn! CS $32_{?}$

- What is 50_{10} in binary?
  - $1*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 0*2^0$ -> 110010
- What is 101010_2 in decimal?
  - $1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$ -> 42
- What is 111111_2 in hexadecimal?
  - Divide into 4 bit sections: 0011 1111
  - Convert each section into base 16 value: 3 f
  - 0x3f
- What is 0xA5 in binary?
  - Reverse the above process: $A_{16} = 10_{10} = 1010_2$, $5_{16} = 5_{10} = 0101_2$
  - 10100101
- If the title of this slide says CS 50_{10}, what base is it currently in?

ASCII

- At a fundamental level, chars are just numbers
- They get treated like chars depending on how we use them
- ASCII value maps them to chars

Your Turn! ACII Math

- What will the following print out?
- `int first = 65;`
- `int second = 'A' + 1;`
- `char third = 'D' - 1;`
- `char fourth = 68;`
- `Printf("%c, %c, %c, %c", first, second, third, fourth);`

- `A B C D`

Floating Point Imprecision

- Infinitely many real numbers
- Finite bits
- -> imprecision
- Just something to be aware of – use ints where possible, since they are designed in hardware and are not imprecise.

Scope

- The area in which the program has access to a variable
- “What happens in braces stays in braces”
- Global
  - Entire program has access
  - Exists for entire duration of program
- Local
  - Confined to region of code
  - eg a function, a loop, etc

Your turn! Scope.c

- What does this program output?
  - 8
  - 3

```c
#include <stdio.h>
// protoytpes
void m();
void n();

int x = 5;

int main(void)
{
    int x = 3;
    m();
    printf("%d\n", x);
}
void m()
{
    x = 8;
    n();
}
void n()
{
    printf("%d\n", x);
}
```

Switches

- Value of variable is either an int or a char in this syntax

```
switch(variable)
{
    case CONST1:
        // if the variable equals this constant,
        // execute this code
        break;
    case CONST2:
        // if the variable equals this const
        // we execute this code
        break;
    // more cases...
        .
        .
        .
    default:
        // code to be executed if none of the cases are met
}
```

Switches

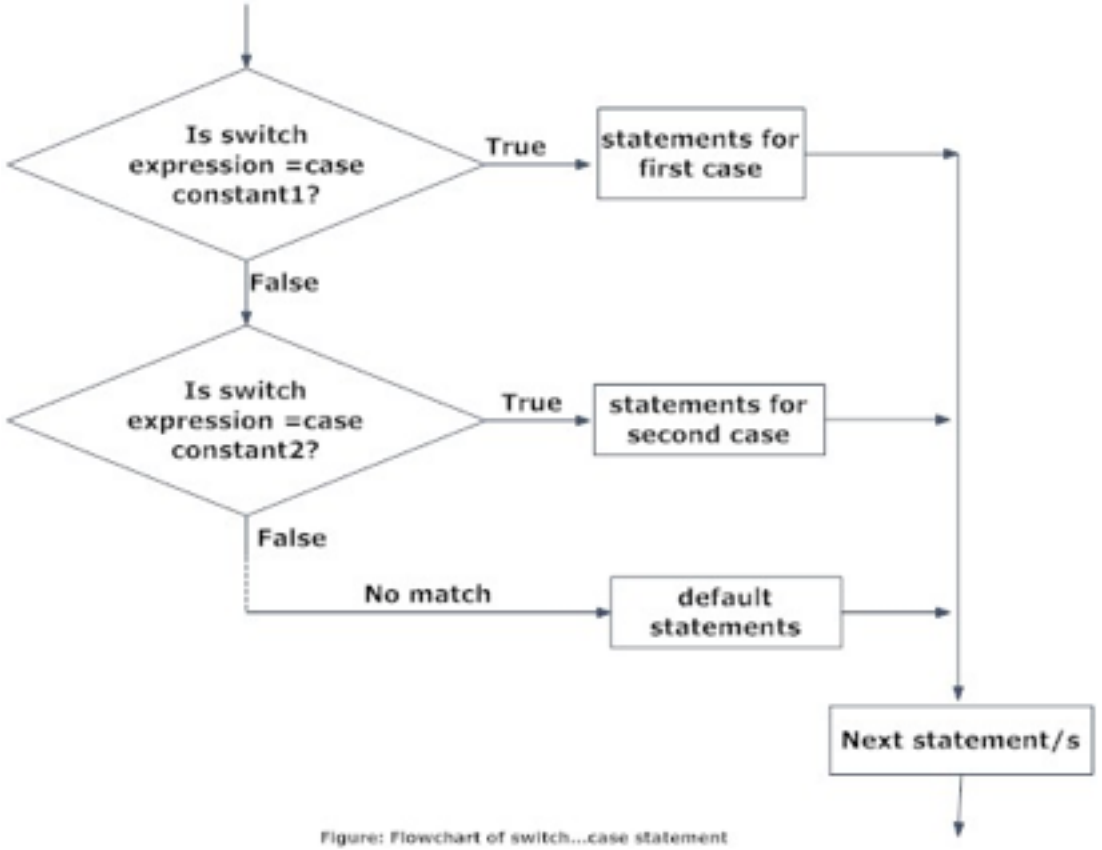


Figure: Flowchart of switch...case statement

Your Turn! Switch.c

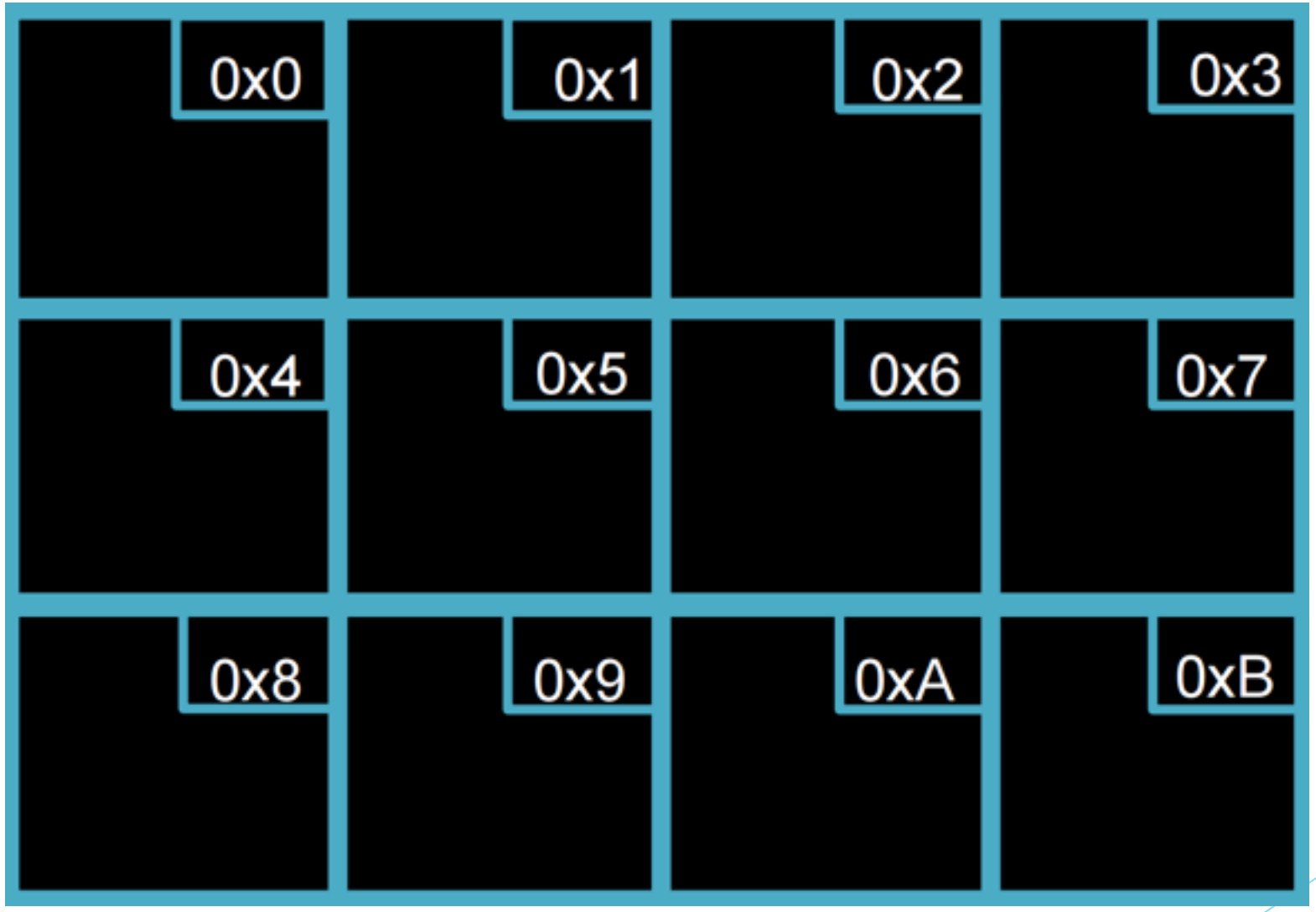
- Write a piece of code that uses a switch statement to check the first letter of the first command line argument a user inputs
- If the letter is is equal to upper or lower case A , print “It is the first letter”
- Else print “It’s not the first letter”

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc!= 2)
    {
        printf("Usage: ./switch <word>\n");
        return 1;
    }

    switch(argv[1][0])
    {
        case 'A':
            // I can leave this blank if I just want it to evaluate
            // the same code as the next case
            // in this program, both a and cause the same output
        case 'a':
            printf("It is the first letter of the alphabet.\n");
            break;
        default:
            printf("It's not the first letter of the alphabet.\n");
            break;
    }
}
```

Pointers - Memory



Creating Pointers

- <type>* <variable name>;

```
// declare some pointers
int *x;
char *y;
float *z;
```

- What are these pointing at right now?

Defining Pointers

- Either:
  - <type>* <pointer name> = &<variable of type>;
  - <type>* <pointer name> = malloc(sizeof(<type>));

```
// statically allocated (stack) pointer
int x = 9;
int *x_ptr = &x;

// dynamically allocated (heap) pointer
int *usr_ptr = malloc(sizeof(int));
if (usr_ptr == NULL)
{
    // there was a problem
    return 1;
}
```

Referencing and Dereferencing

- Referencing (ie, address of)
  - &<variable name>
- Dereferencing (ie, go to address)
  - *<point name>

Your turn! Under the hood

```
int x = 5;
int *ptr = &x;
int copy = *ptr;
```

- Fill in the table below based on this snippit of code.
- Feel free to make up addresses if you don't think they should be the same

Variable	Address	Value
x	0x04	5
ptr		
copy		

Pointer Arithmetic

- What happens when I add or subtract the value n from a pointer?
  - Adjusts pointer by n * sizeof(<pointer type>) bytes
- Recall that that's basically how an array works:
- Array[1] = *(array + 1)
  - which is equivalent to *(array + 1 * sizeof(<array type>))

Your turn! Pointer Math

- What happens when I add or subtract the value n from a pointer?
  - Adjusts pointer by n * sizeof(<pointer type>) bytes
- Based on this, fill in this chart. Assume &x = 0x04

	x	y
`int x = 5;`	5	NULL
`int *y = &x;`	5	0x04
`y += 1;`	5	0x08

Pointers – Passing by Reference

- Remember how we want to avoid global variables?
- We can do that by passing by reference
- To explore, let's look at this buggy swap function

Buggy_swap.c

- What happens when I run this program?
- How could knowing pointers help with this?

```c
#include <stdio.h>

void buggy_swap(int a, int b);

int main(void)
{
    int x = 9;
    int y = 10;

    printf("x = %d y = %d\n", x,y);

    buggy_swap(x,y);

    printf("x = %d y = %d\n", x,y);
}

/* (hypothetically) swaps a and b */
void buggy_swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}
```

Swap.c

- What if we passed our swap function the addresses of the variables we want to swap?
- Still passing a copy of information, but this time, we're passing a copy of an address – the data the address points to is still the same
- This is how we avoid using global variables (which are bad) and still get the results we want from another function
- This is called passing by reference

```c
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 9;
    int y = 10;

    printf("x = %d y = %d\n", x,y);

    swap(&x, &y);

    printf("x = %d y = %d\n", x,y);
}

/* swaps two values by using addresses */
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
    return;
}
```

Memory

- stack: block of memory set aside when a program starts running
  - each function gets its own stack frame
  - stack overflow: when the stack runs out of space, results in a program crash
- heap: region of unused memory that can be dynamically allocated using malloc (and realloc, etc.)
  - ALWAYS free ALL dynamically allocated memory to avoid leaks

Recursion

- a programming concept whereby a function calls itself
  - don’t forget to include a base case!
- pros:
  - can lead to more concise, elegant code
  - some algorithms lend themselves to recursion ▪ e.g., merge sort
- Cons
  - Can lead to buffer overflow if there’s an error with the base case

Search and Sort Run Times

	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
O	n^2	n^2	n^2	$nlogn$
Ω	n	n^2	n	$nlogn$
Θ		n^2		$nlogn$

Structs

- Allow us to create out own data type to hold data of different type
  - Recall the student struct from lecture
  - What's the difference between these two structs?

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

- This creates a new type called student
- To declare:
  - `student stu_1;`

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

- This creates a structure called student
- To declare:
  - `struct student stu_1;`

Structs: creating and accessing

- Declare using the struct name as the variable type
- Access using the . operator

```
// direct assignment
student stu_1;
stu_1.id = 88;
stu_1.id = "Anna";
```

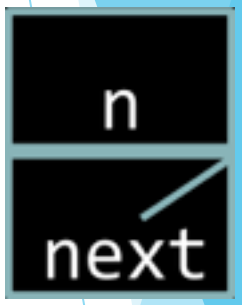
- If we have a pointer to a struct, use the arrow notation

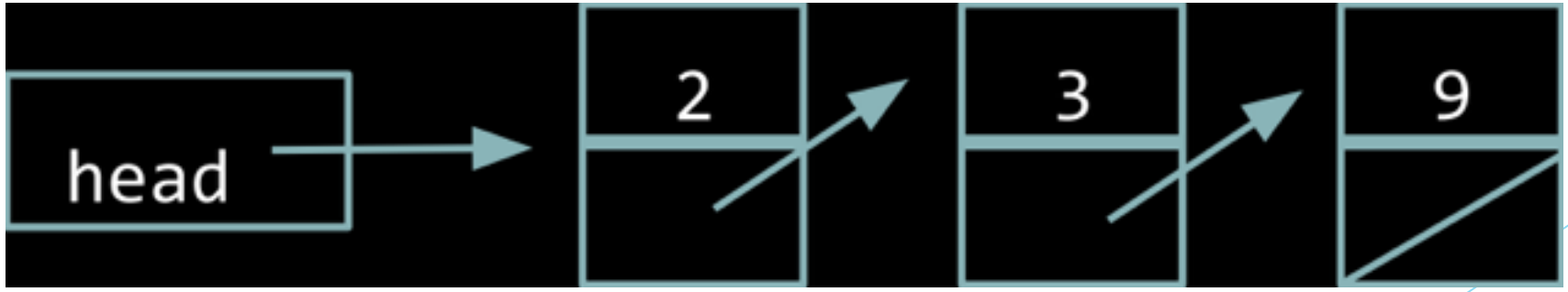
```
// and using a pointer
student *ptr = malloc(sizeof(student));
(*ptr).name = "Rob";
ptr->name = "Rob";
```

Linked List

- Uses a recursive datatype
  - A struct that points to anther version of itself
- How do we search a linked list?
- How do we insert into a linked list?
- What are advantages and disadvantages of this data structure?

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





Stacks

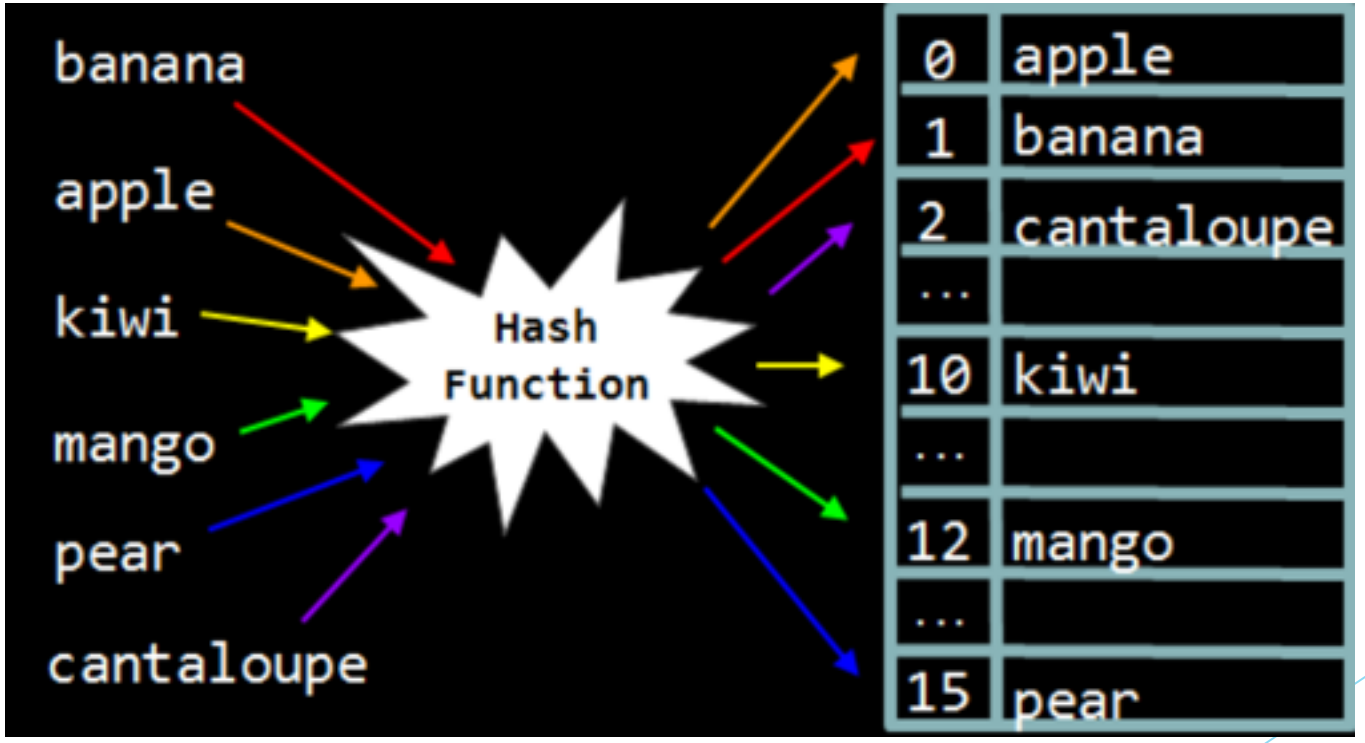
- first-in, last-out (FILO)
- elements are successively pushed down as other items are added
- elements are pushed on and popped off
- keep track of both the size and capacity
  - you need not keep track of capacity if you use a linked list rather than an array

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

Hash Table

- data structure where the position of each element is decided by a hash function
- What makes a good hash function?



Hash Table

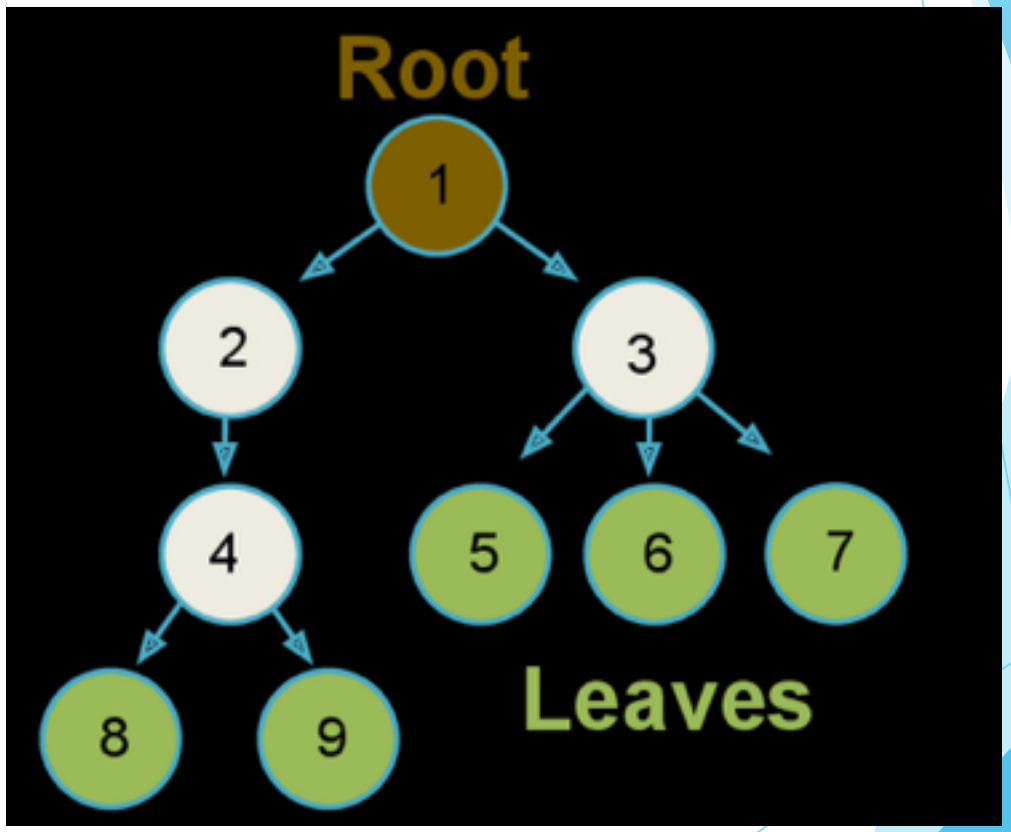
- What do we do when there are collisions?
- Linear probing – we could look for nearby empty slots
- Separate chaining – we could make each bucket a linked list and insert at the head of that

Trees and Tries

- tree: a data structure in which data is organized hierarchically
  - e.g., binary search tree
- trie: special kind of tree that behaves like a multi-level hash table

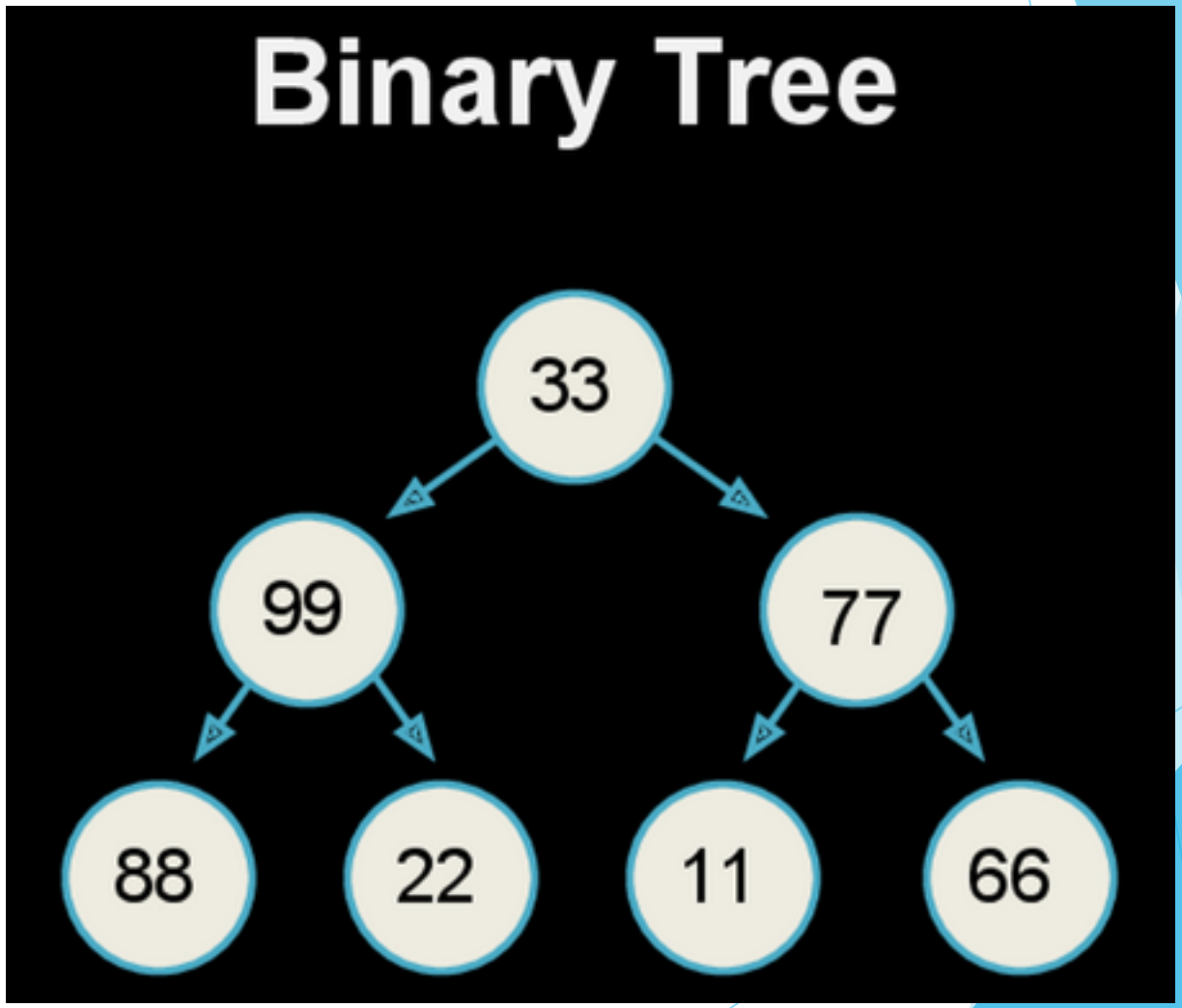
Tree

- This is the general structure for a tree
- However, randomly arranged like this, it's not very helpful
- Ideally we want balanced trees – trees that have the same number of levels to the leaves.



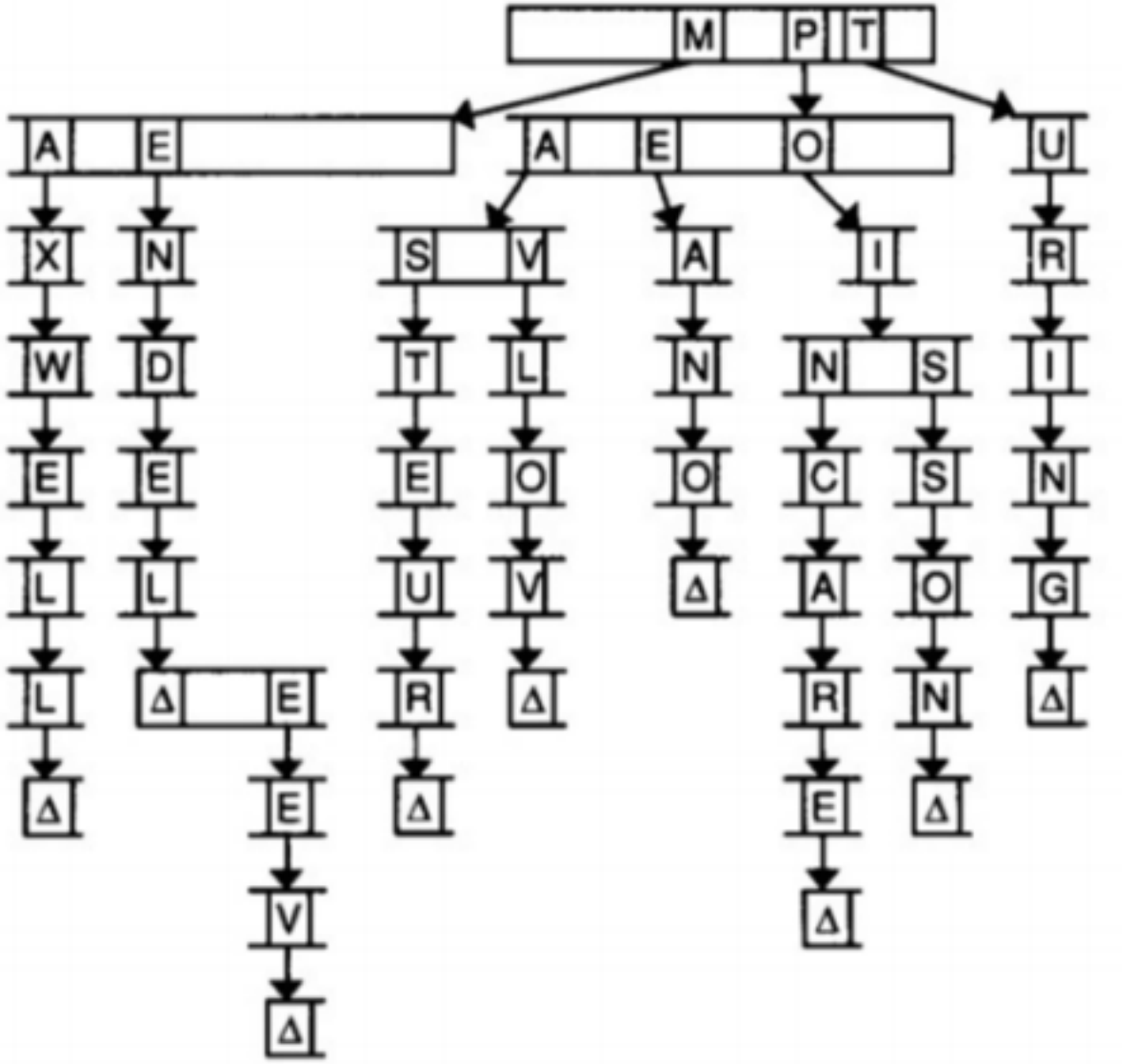
Binary Tree

- Enter the binary tree



Tries

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



File I/O

- Not something we directly addressed last section, but do take a look over the various file i/o things we learned in class
- Every time we open a file, close the file!
  - Analogous to malloc() and free.

File I/O - Commands

- FILE* fopen(<name of file>, <mode>)
- fread(<storage ptr>,<elt size>,<number of elts>,<file* stream>)
- fwrite(<cont info ptr>,<elt size>,<number of elts>,<file* stream>)
- fgets(<storage ptr>, <int size of string>, <file* stream>)
- fputs(<const char array>,<file* stream>)
- char fgetc(<file pointer>)
- fputc(<char c>, <file* stream>)
- fclose(<file pointer>)

Previous Quiz Questions to Study

- 2015: Quiz 0
  - 8, 17, 18, 23
- 2014: Quiz 0
  - 24
- 2014: Quiz 1
  - 27
- 2013: Quiz 0
  - 0, 1, 2, 12, 17, 20, 22
- 2013: Quiz 1
  - 8, 21, 22

2015: Quiz 0, Problem 8

(6 points.) Suppose that we'd like to add to the CS50 Library a function that copies a string. Complete the implementation of `CopyString`, below, in such a way that the function returns a character-for-character copy of `s`, with the copy's characters in their own block of dynamically allocated memory, terminated with `'\0'`. Return the copy (i.e., the address of the copy's first byte) unless some error occurs, in which case return `NULL`. Assume that `s` will not be `NULL`.

```
char* CopyString(char* s)
```

2015: Quiz 0, Problem 17

(4 points.) Suppose that a stack for integers is defined per the below, wherein `numbers` is an array for the stack's integers, `CAPACITY` (a constant) is the stack's capacity (i.e., maximal size), and `size` is the stack's current size.

```
typedef struct
{
    int numbers[CAPACITY];
    int size;
}
stack;
```

Complete the implementation of `push` below in such a way that it pushes `n` onto a stack, `s`, if `s` isn't already full. Assume that `s` has been declared globally. Consider `s` full only if its `size` equals `CAPACITY`. No need to return a value, even if `s` is full. Your implementation should operate in constant time.

```
void push(int n)
```

2015: Quiz 0, Problem 18

(4 points.) Suppose that a queue for integers is defined per the below, wherein `numbers` is an array for the queue's integers, `CAPACITY` (a constant) is the queue's capacity (i.e., maximal size), `size` is the queue's current size, and `front` is the index of the integer at the front of the queue.

```
typedef struct
{
    int front;
    int numbers[CAPACITY];
    int size;
}
queue;
```

Complete the implementation of `enqueue` below in such a way that it inserts `n` into a queue, `q`, if `q` isn't already full. Assume that `q` has been declared globally. Consider `q` full only if its `size` equals `CAPACITY`. No need to return a value, even if `q` is full. Your implementation should operate in constant time.

```
void enqueue(int n)
```