COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 12 - Data Structures

Zheng Zheng

McMaster University

Winter 2023

Gettting Structure in your life

Up to this point, we have had no abstraction mechanisms in C besides functions. Consider the following scenario...

- Let's imagine we have a database that contains the names, email addresses, and pinball high scores of customers, stored as a CSV file:

```
"John Boring","john.boring@whatever.com",100000
"Sally Plain","sally.plain@whatever.com",200000
"Nancy Snore","nancy.snore@whatever.com",50000
"Peter Bland","peter.bland@whatever.com",150000
"Thamarias the High Elf","Elder.Cabbage@
  ancientorderofpurplewizards.com",999999999
```

We could group this data by field into arrays, or, we could use `struct` to make a record data type!

Introducing... `struct`!

- A structure groups several variables together into a single data type.
- Structures are roughly analogous to classes in Python, if you remove the methods.

```
1 struct customer {
2   char *name;
3   char *email;
4   int pinball;
5 };
```

- This creates a new data type, `customer`, with the **fields** `name`, `email` and `pinball`.

Working with Structs

The fields of a structure are accessed through **dot syntax**.

```
1  struct customer ma_bois[5];
2  ma_bois[0].name = "John Boring";
3  ma_bois[0].email = "john.boring@whatever.com";
4  ma_bois[0].pinball = 100000;
5  struct customer ma_boi = {"Sally Plain"
6                ,"sally.plain@whatever.com"
7                ,200000 };
8  printf("%s, %s, %d\n"
9         , ma_bois[0].name
10        , ma_bois[0].email
11        , ma_bois[0].pinball);
```

You can also initialize a structure in a similar manner to an array (line 5 above).

Some Quality of Life Improvements

We can avoid having to invoke the `struct` keyword when declaring a variable using our new structures. Simply include the following in the global namespace:

```
typedef struct customer Customer;
```

We can also define a default structure variable as follows:

```
struct customer {
  char *name;
  char *email;
  int pinball;
} cust_def = {"None", "None", 0};
```

This allows us to use `cust_def` whenever we want to initialize or reset a record:

```
  Customer ma_bois[5] = {cust_def, cust_def, cust_def,
    cust_def, cust_def};
```

Additional Notes on Structures

- Structure members may be variables of primitive data types, or aggregate data types, such as arrays or other structures.
- A structure may *not* contain an instance of itself.
- A structure may contain a *pointer to* an object of the same type as the structure
  - Many advanced data structures, such as trees, rely on this.
  - This is a **recursive** ir **self-referential** data structure.
- You may declare any number of variables after a structure definition, not just the default shown on the previous slide.
- You may even omit the strucutre tag (in our case `customer`), and declare all of your variables in the global namespace.

Valid Operations on Structures

Unlike classes in Python (and other object oriented languages), structures have a relatively small number of operations which may be performed on them:

- Assignment (=)
- Address of (&)
- Finding Bit Width with `sizeof()`
  - The size of a structure is (roughly) the sum of the sizes of all its fields.
  - Sometimes, blank space may be included due to the addressing of small variables.

NOTE: These are operations on variables of the structure type, not fields of the structure.

A Longer Example

```
1  // Fig. 10.2: fig10_02.c
2  // Structure member operator and
3  // structure pointer operator
4  #include <stdio.h>
5
6  // card structure definition
7  struct card {
8     char *face; // define pointer face
9     char *suit; // define pointer suit
10 };
11
12 int main(void)
13 {
14    struct card aCard; // define one struct card variable
15
16    // place strings into aCard
17    aCard.face = "Ace";
18    aCard.suit = "Spades";
19
20    struct card *cardPtr = &aCard; // assign address of aCard to cardPtr
21
```

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 1 of 2.)

A Longer Example (cont.)

```
22      printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
23         cardPtr->face, " of ", cardPtr->suit,
24         (*cardPtr).face, " of ", (*cardPtr).suit);
25   }
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 2 of 2.)

Structures + Pointers = Love

In the previous example, you may have noticed a pointer to a structure type.

- When you have a pointer to a structure, you access the fields of the structure pointed to using `->`
- This is syntactic sugar for pointer dereferencing and field access, combined into one operator:
  - `x ->y` $\equiv$ `(*x).y`
- It is almost always more efficient to pass structures *by reference* rather than *by value*
- You can allocate memory to a structure pointer the same way as everything else: using `malloc()` or `calloc()`.
  - Don't forget to `free()`!

And Here's Some Bitwise Apropos of Nothing

Bitwise operators will become very important when you start working with embedded systems, so let's review them!

- & - bitwise and
- | - bitwise or
- ^ - bitwise xor
- << - bitwise left shift
- >> - bitwise right shift
- ~ - bitwise complement

"Bitwise" means "by the bit," where individual bits are the values under consideration, as we shall see...

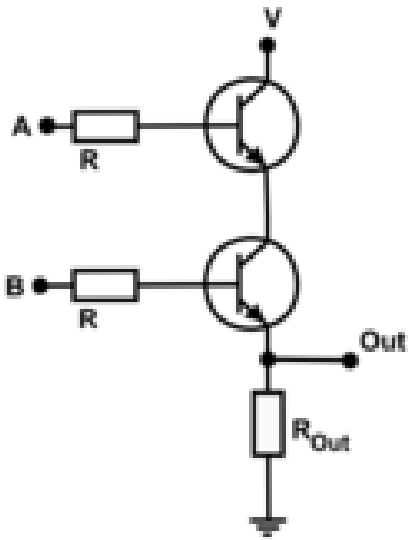
And

Bitwise AND may be thought of as the application of a logical AND gate to each of the bits in the two operands sequentially. Common C applications include bitmasking.

x	y	x & y
1	1	1
0	1	0
1	0	0
0	0	0

Variable	Dec	Binary
x	51	0b00110011
y	85	0b01010101
x & y	17	0b00010001

Transistor AND Gate

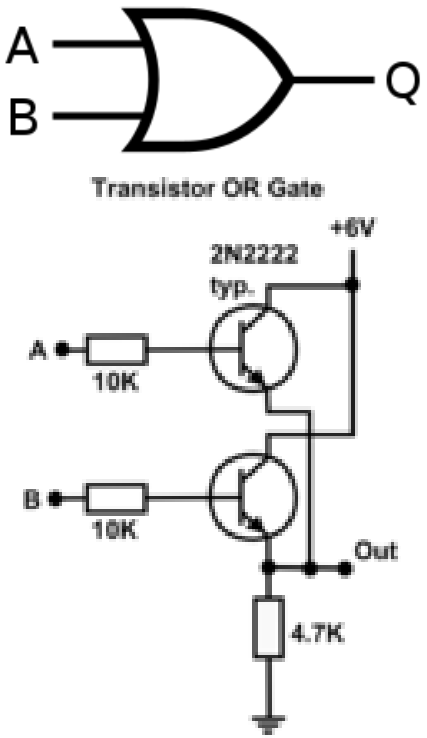


Or

Bitwise OR is applied in the same manner, but uses a logical OR operation.

x	y	x |y
1	1	1
0	1	1
1	0	1
0	0	0

Variable	Dec	Binary
x	51	0b00110011
y	85	0b01010101
x |y	119	0b01110111

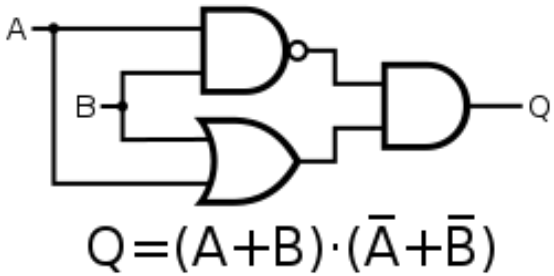


Xor

^ means "exclusive or", outputing 1 if the inputs are dissimilar. Common applications include bit toggling and arithmetic addition.

x	y	x ^y
1	1	0
0	1	1
1	0	1
0	0	0

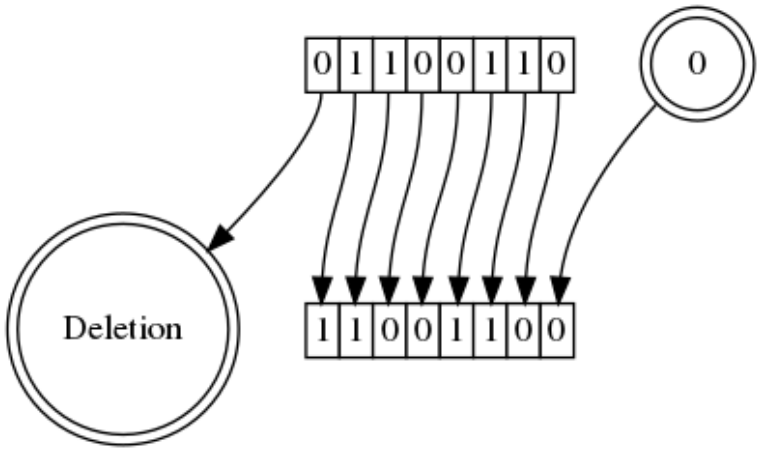
Variable	Dec	Binary
x	51	0b00110011
y	85	0b01010101
x ^y	102	0b01100110



Left Shift

The left shift operation `x <<y` moves the bit values of the number x by a number of bits y to the "left".

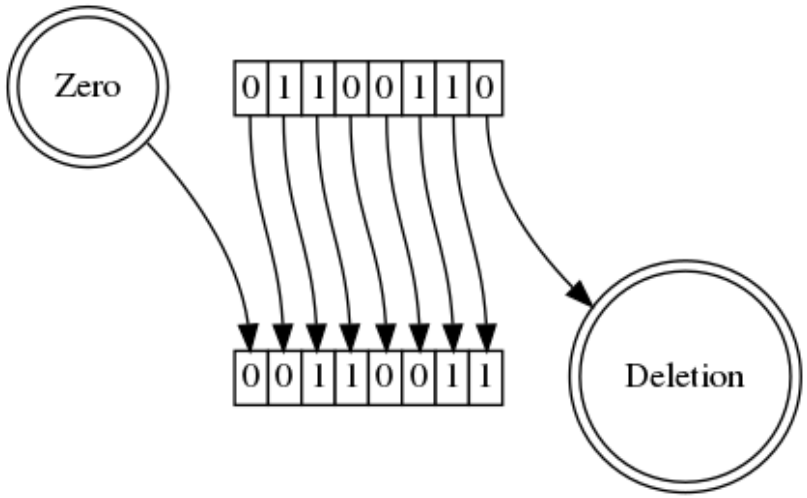
- We consider "left" to be towards larger place values, in **big endian** systems (which are most of them).
- From a pragmatic standpoint, this is precisely the same as *multiplication* by 2^y



Right Shift

The right shift operation `x >>y` moves the bit values of the number x by a number of bits y to the "right".

- Just like the last one, but in the reverse direction.
- From a pragmatic standpoint, this is precisely the same as *division* by 2^y, with truncation.



One's Complement

The complement operator ~simply takes each bit and flips it.

Variable	Dec	Binary
`x`	51	0b00110011
`~x`	204	0b11001100

If you're the type of nerd that likes to squeeze either speed or power efficiency out of your code, bitwise operations typically take far less circuitry (and in some cases time) than algebraic operations.

Some Common Operations using Bitwise

Subtraction using Bitwise

```
1  int subtract(int x, int y)
2  {
3    int borrow = 0;
4      while (y != 0)
5      {
6            borrow = (~x) & y;
7            x = x ^ y;
8            y = borrow << 1;
9      }
10     return x;
11 }
```

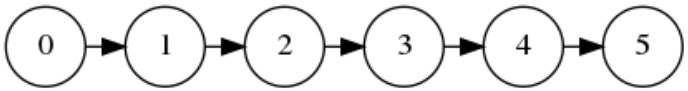
Addition using Bitwise

```
1  int Add(int x, int y)
2  {
3    int carry = 0;
4      while (y != 0)
5      {
6            carry = x & y;
7            x = x ^ y;
8            y = carry << 1;
9      }
10     return x;
11 }
```

Singly Linked Lists

A singly linked list is a graph where each node points to one other node, or no nodes, and no node is pointed to more than once.

- This effectively creates a linear data structure.



- This is similar in many respects to our old friend the array, but with some key differences.
- This is how languages such as Lisp and Haskell store lists, their primary native aggregate data type.

Linked Lists with Structures

The nodes of the graph can be expressed as objects of a structure type in C. All we need to do is take an existing structure and add the self-referential pointer `*next`...

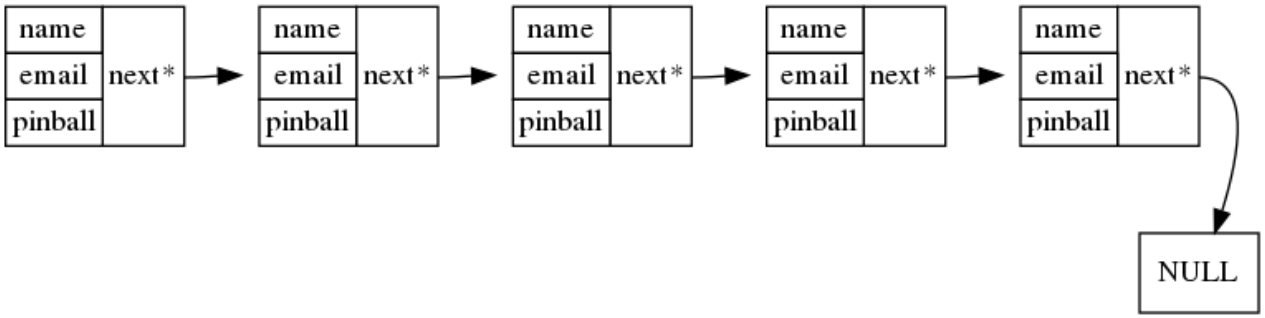
```
1 struct SLLnode {
2   char *name;
3   char *email;
4   int pinball;
5   SLLnode *next;
6 };
```

- The last element in the list points to `NULL`.
- If we wish to refer to a singly linked list in `main()` or one of our functions, we need only a single node, from which we can access the rest.

```
1 struct SLLnode linkedList;
```

Linked Lists with Structures (cont.)

If we apply our data structure to the singly linked list structure, we get the following:



The more information that is contained in the structure, the more advantageous it is to organize it in this manner.

- If you want to pass the first node by reference to a function, keep in mind that within the function you will have to dereference the argument to get to the first node's data.

Linked Lists vs Arrays

So why go to all the bother? Seems like a lot of work just to make something that functions roughly like an array.

- Because the list is not stored in contiguous memory, it can be dynamically grown or shrunk *without having to copy the whole array*.
- Because the next node is indicated with pointers, and those pointers may be modified, we can *insert an element* without having to move any of the other elements in memory.

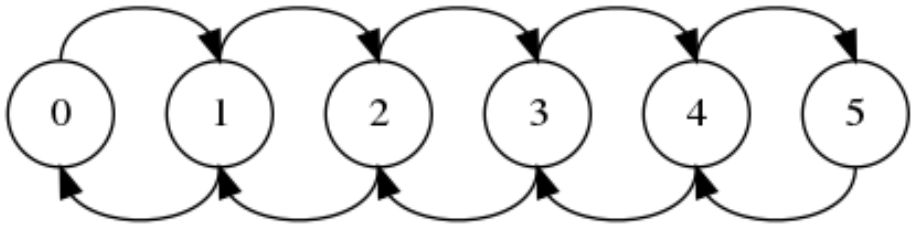
However, there are some drawbacks.

- Because the nodes are not stored in contiguous memory, we can't perform pointer arithmetic to find a particular index.
  - Instead, we have to *follow the node pointers.*
  - This means that finding the record at index `i` takes `i` operations, where an array can do it in 1.

Looking Back on Lists Past

Singly linked lists are awesome, but you can only traverse them in one direction.

- The **Doubly-Linked List** is a singly linked list, with the dded condition that each node has an edge that points to the node that points to it.
- No node may be pointed to more than twice.



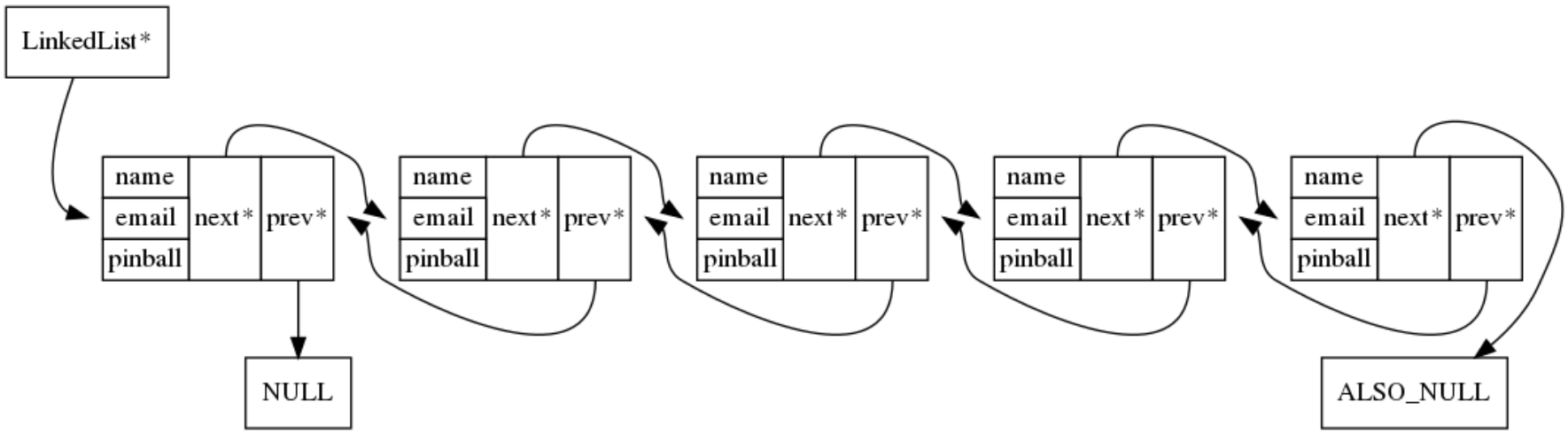
C Implementation

With the addition of one more pointer, `*prev`, we have made a singly linked list node into a doubly linked list node.

```
1 struct DLLnode {
2   char *name;
3   char *email;
4   int pinball;
5   DLLnode *next;
6   DLLnode *prev;
7 };
```

- The first element's `prev` pointer will point to `NULL`
- The last element's `next` pointer will also point to null.
- While a doubly linked list increases the amount of bookkeeping, it makes the list traversable in either direction.

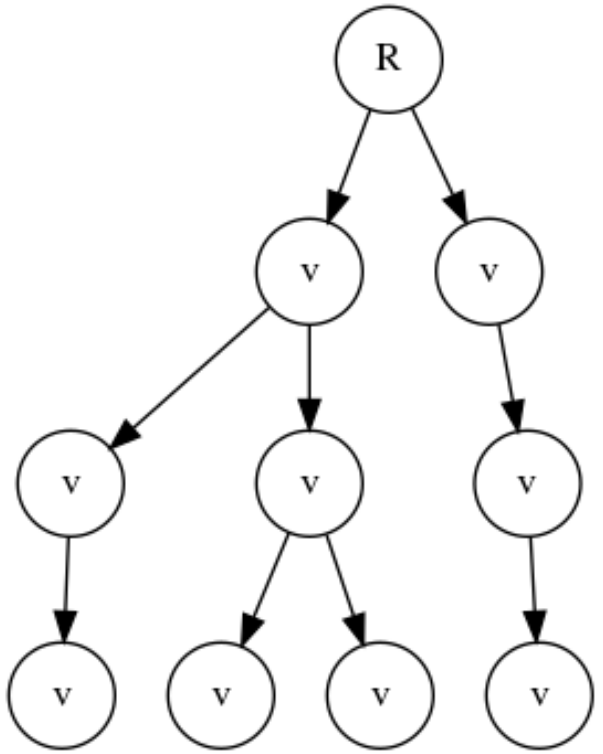
C Implementation (cont.)



Binary Trees!

A **binary tree** is an acyclic, directed graph, where each node may connect to at most two others.

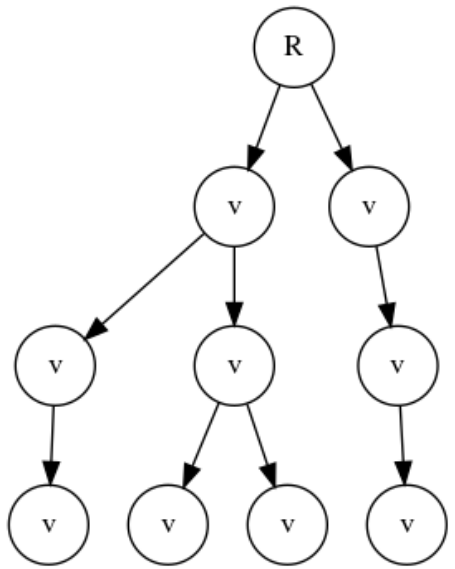
- Trees have a **root node**, here R
  - The root of the tree is the node to which no other node points.
- For every vertex v, there is exactly one path to that vertex from the root node.
- Each node has exactly one **parent** (except for the root), and up to 2 **children**.
  - *binary* tree nodes have ≤ 2 children.
  - *n-ary* tree nodes have $\leq n$ children.



Binary Trees!

- A **leaf node** is any node in the tree which does not point to any other nodes.
- A **stem node** is any node in the tree which does point to other nodes.

- The **depth** of a tree is number of nodes in the longest branch. (i.e., the distance from the root to the furthest leaf in nodes)
- The nodes in a tree are organized into **levels**
  - All the nodes of the same distance to the root node are at the same level.
- The **width** of a level is the number of nodes in that level.
- A set of $n \geq 0$ disjoint trees is called a **forest**.



Implementation Considerations

Whereas the singly linked list had only one "next" pointer, The binary tree has two.

- By convention, we call these **left** and **right**

```
1 struct  BinTree {
2   int value;
3   BinTree *left;
4   BinTree *right;
5 };
```

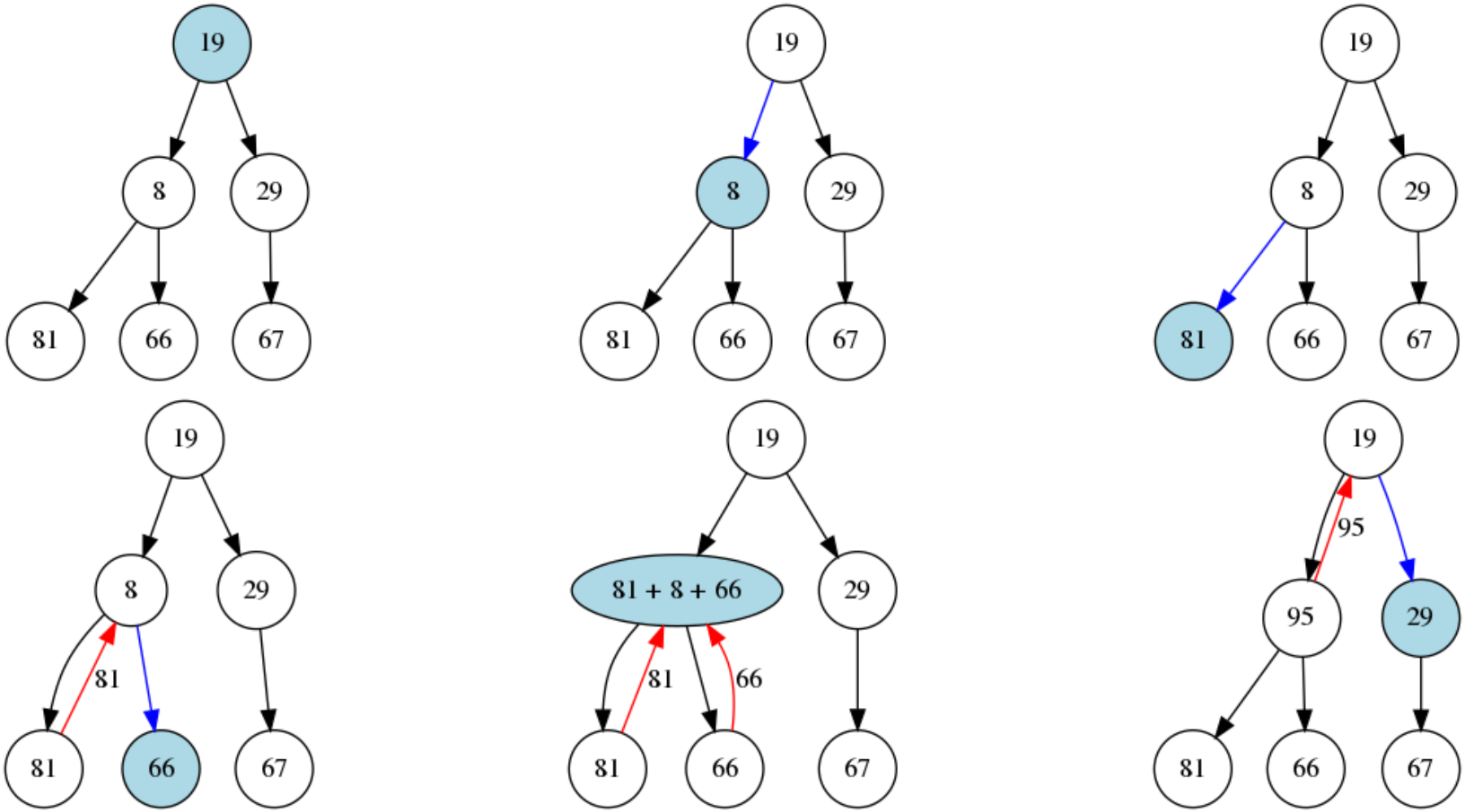
- One interesting property of trees is that they are a *naturally inductive* data structure.
  - i.e., both the nodes pointed to by `left` and `right` can be rightly considered roots of their own trees.
  - Therefore, this is a natural application for *recursive algorithms*.

Tree Summing Algorithm

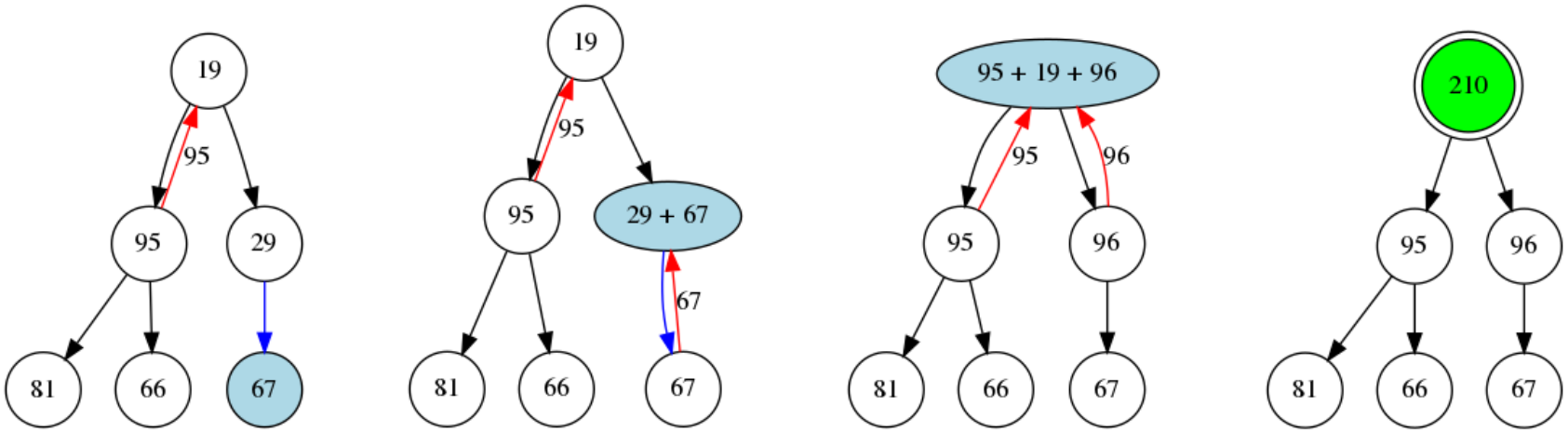
Let's design an algorithm to recursively sum all the values in a tree! The sum of a tree is the sum of the left tree, the right tree and the value of the root node.

1. If the left tree doesn't exist (i.e., is a null pointer), it's sum is zero.
2. If the left tree exists, we recursively call the summing function on the left tree, considering the node pointed to by the left pointer to be the new root node in this recursive call. We hold onto the return value for later.
3. We repeat steps 2 and 3 with the right tree.
4. We add the results of step 2 and 3 to the value of the root node, and return this value.

Example: Summing the Values in a Tree



Example: Summing the Values in a Tree (cont.)



This example overwrites node values with the results of summation. This is mostly for demonstration purposes, a real algorithm wouldn't overwrite node contents like this!

Depth vs Breadth First Search

Searching a tree can go very differently depending on how you approach it.

- **Depth-First Search** (DFS)
  - Selects a branch and follows it until a leaf node is found.
  - If a leaf is reached without the searched-for item being found, DFS backtracks to the last node it made a branch decision on (with untried branches) and tries another branch.
  - Suitable for exploring decision branches in game AI
- **Breadth-First Search** (BFS)
  - Searches all of the nodes in one level of the tree before moving on to the next.
  - Requires more traversal / bookkeeping than DFS.
  - Finds minimum branch depth much faster than DFS.

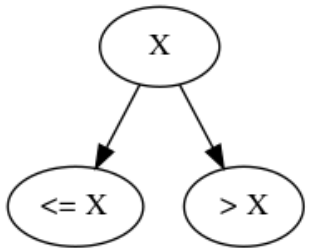
DFS vs BFS is a popular topic among people who think computer algorithms have applications to human behaviour.

Binary Search Trees!

Both DFS and BFS assume nothing about the structure of the data contained in the tree (aside from the fact it is a tree). If we give trees certain mathematical properties, we can write algorithms that vastly improve general-case runtimes!

The **Binary Search Tree Property** is as follows:

- If a node has a left child node, the value of that node is less than or equal to the value of the parent node.
- If a node has a right child node, the value of that node is greater than the parent node.



Depending on who writes the definition, duplicate elements are sometimes disallowed.

Binary Search Trees: Pros and Cons

Pros:

- Best case scenario runtime for search, insertion and deletion operations is $O(d)$, Where d is the depth of the tree.
  - In the best case scenario, the depth of the tree is $log_2(n)$, where n is the number of nodes.

Cons:

- The worst case scenario for d is n
  - This is a tree with only one branch, (basically a linked list).
- The search tree property is maintained, which requires some extra book-keeping.

To address the drawbacks of unbalanced trees, BSTs in practice often use *balancing algorithms*, such as **Red Black Trees** or **Adelson-Velsky and Landis (AVL) Trees**.

A Brief Introduction to Graph Theory

In order to solve problems, we need mathematical models in which to express those problems. A very useful mathematical model used in algorithm design is the **graph**.

- A graph is a way of modelling the relationships between objects.
- A graph is defined (mathematically) as the ordered pair:

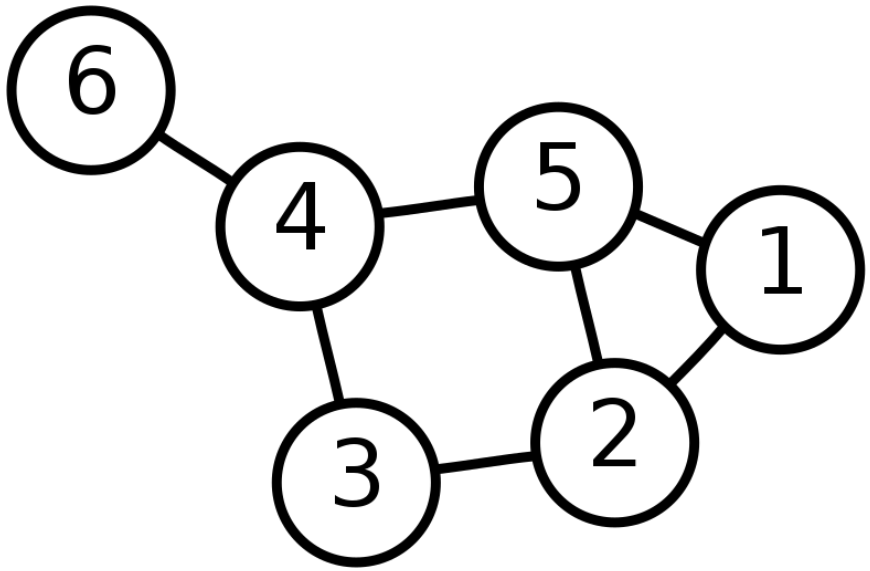
$$G = (V, E)$$

- Where V is the set of **nodes** or **vertices**, and
- Is the set of edges, conforming to :

$$E \subseteq \{\{x, y\} | x, y \in V \wedge x \neq y\}$$

- The above means that an edge is defined as a pair of vertices, where the two vertices may not be the same.

Here's an Example...



In the above...

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{\{6, 4\}, \{4, 5\}, \{4, 3\}, \{5, 1\}, \{5, 2\}, \{3, 2\}, \{1, 2\}\}$

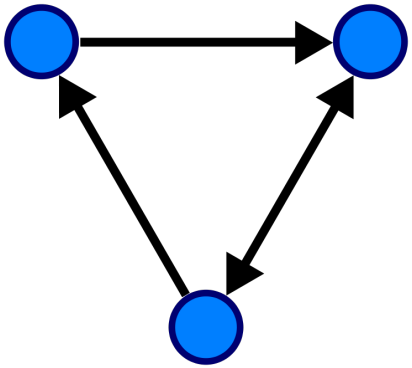
Directed Graphs

The previous two slide present graphs in their most general form. If we apply properties to graphs, we get some interesting structures.

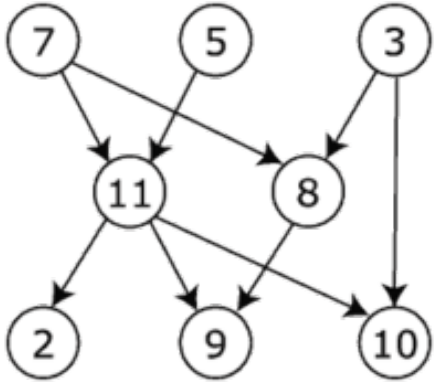
- Directed Graphs:
  - If ordering of the numbers in an edge pair matters, the graph is **directed**.
  - Simply put, the graph has *arrows* instead of *straight lines*.
- Cyclic vs Acyclic Graphs:
  - A cyclic graph contains **cycles**.
  - If it is possible to return to any node by following the edges of a directed graph, then the graph is **cyclic**.
  - Otherwise it is **acyclic** (ay-sick-lick).

Directed Graphs

Directed, Cyclic Graph



Directed, Acyclic Graph



Graphs have lots more properties than this, but this will do for now.

- In the context of C programming, we can make *nodes* with structures
- If our structure contains a pointer to another structure of the same type, this can be considered an *edge*.

Acknowledge

The contents of these slides were liberally borrowed (with permission) from slides from the Summer 2021 offering of 1XC3 (by Dr. Nicholas Moore).