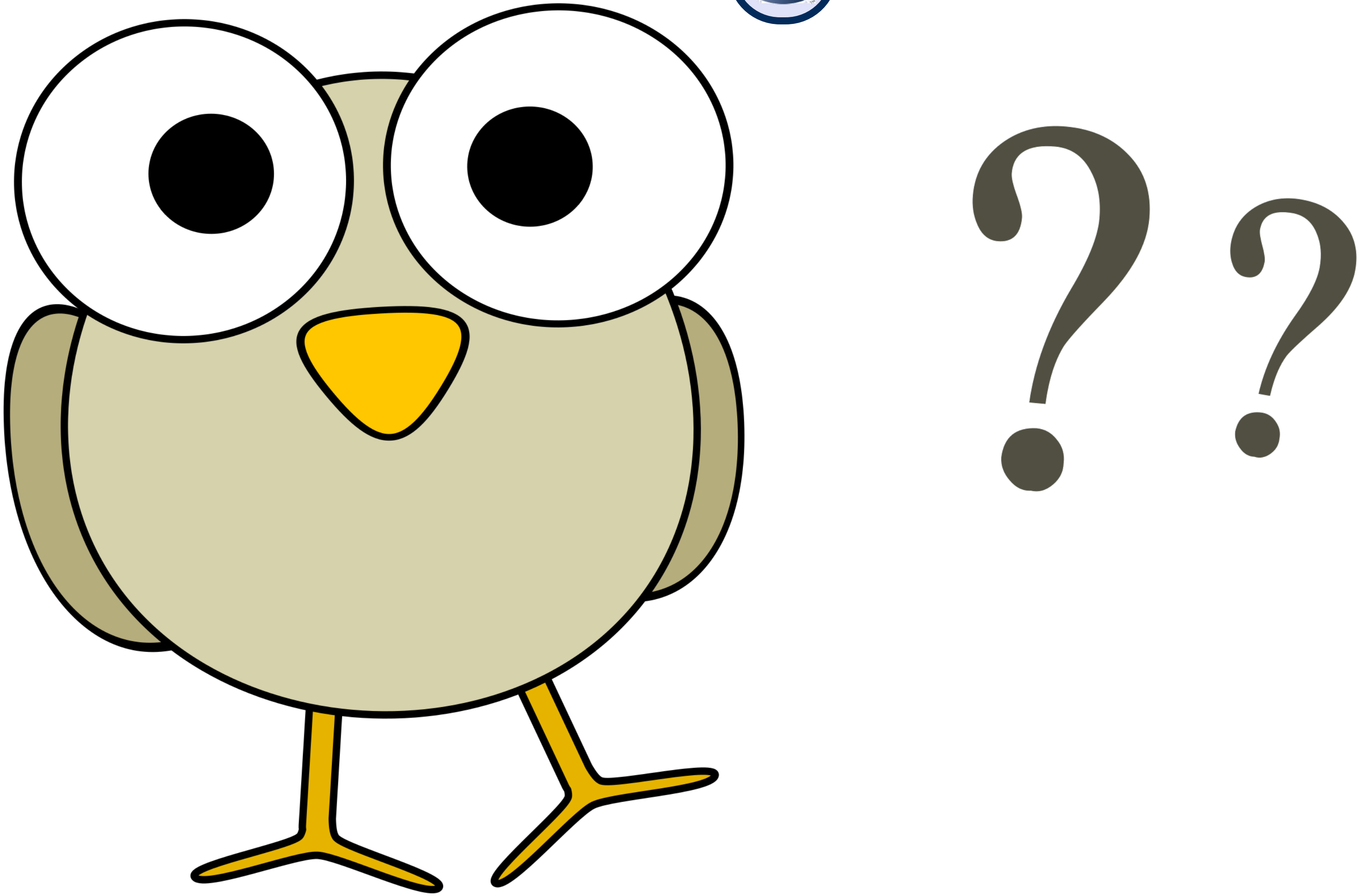
CSCI 200: Foundational Programming Concepts & Design Lecture 34

Arrays vs. Linked Lists

Previously in CSCI 200

- Array identifier points to base address of array
- Array stored in one contiguous block of memory
- Offset used to determine memory location of specific element
- Array operations & Big O complexity
- Pointer Math & Arrays
  - All pointers are arrays
    - Pointing to a single entity is just an array of size 1
  - All arrays are pointers

Questions?



Learning Outcomes For Today

- Discuss the pros/cons of using an array.
- Discuss the pros/cons of using a linked list.
- Compare and contrast the benefits of using an array or a linked list.
- Analyze the run-time cost of each operation and explain how to perform the following operations on an array and a linked list: addition, removal, traversal, search.
- Group data using a `struct`.

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

Data Structure Operations

Operation	Array
Element Access	$O(1)$
Traversal	$O(n)$
Add	$O(n)$
Delete	$O(n)$
Search	$O(n)$
Min / Max	$O(n)$

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

Data Structure Operations

Operation	Array
Element Access	O(1)
Traversal	O(n)
Add	O(n)
Delete	O(n)
Search	O(n)
Min / Max	O(n)

Arrays & Functions

- Pass Array By Pointer

```
void print_array(const int* const P_ARRAY, const int SIZE) {
  for(int i = 0; i < SIZE; i++) {
    cout << P_ARRAY[i] << " ";
  }
}
```

Vector v. Dynamic Array

- Vector wraps a Dynamic Array
- And...
- But...

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

struct

- Acts as a container to *structure* our data
- Creates our own custom data type – that we can use to make variables!

```
struct StructureName {
    dataType variableName;
    dataType variableName;
    // more data members
};
```

Ends in semi-colon!

```
int main() {
    StructureName myStructVar;
}
```

A Person struct

```
struct Person {
    double height;
    double weight;
    short age;
    char gender;
    char hairColor;
    char eyeColor;
    bool rightHandDominant;
    bool rightEyeDominant;
};
```

```
int main() {
    Person person1, person2;
    return 0;
}
```

Can chain together

```
struct ImperialHeight {
    int feet;
    int inches;
};
struct Person {
    ImperialHeight height;
    double weight;
};
int main() {
    Person person1;
    person1.height.feet = 5;
    person1.height.inches = 7;
}
```

Difference between `struct` and `class`

- `class`
  - By default, all members are private
- `struct`
  - By default, all members are public
- Other than that?
  - Identical

class v struct

```
class PointClass {
  int x, y;
};

struct PointStruct {
  int x, y;
};

int main() {
  PointClass classObject;
  classObject.x = 1;               // ERROR! x is private

  PointStruct structObject;
  structObject.x = 1;              // OK! x is public
}
```

Which to use?

- Depends
  - Simply data storage with no validation or manipulation logic?
    - `struct` (everything is public by default, can be accessed by anyone, can be set to anything, needs to be validated external to class)
  - Need data validated and have controlled methods to manipulate the data?
    - `class` (everything is private by default, need to explicitly mark what should be accessible outside the class, validated inside of class)

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

Linked List Concept

- Instead of 1 n-element array
- “Chain” together n 1-element arrays

Linked List Node

- A linked list node contains
  - The value for that element
  - A pointer to the next element

- Create the Node as a struct!

Node Struct

- A "recursive" data structure

```
struct Node {
  int value;
  Node *pNext;
};
```

- Recursive Data Structure:
  - Defined in terms of itself, contains reference to itself
  - composed of instances of the same data structure

Linked List Operations

1. Make a Node
2. Add a Node to the front
3. Get node `i`
4. Print/Traverse/Find/Min/Max/Size the List
5. Print backwards

Data Structure Operations

Operation	Array
Element Access	
Traversal	
Add	
Delete	
Search	
Min / Max	

Data Structure Operations

Operation	Array
Element Access	O(1)
Traversal	O(n)
Add	O(n)
Delete	O(n)
Search	O(n)
Min / Max	O(n)

Data Structure Operations

Operation	Array	Linked List
Element Access	O(1)	
Traversal	O(n)	
Add	O(n)	
Delete	O(n)	
Search	O(n)	
Min / Max	O(n)	

Data Structure Operations

Operation		Array	Linked List
Element Access		O(1)	
Traversal	Forwards		
	Backwards		
Add	Front		
	Middle		
	Back		
Delete	Front		
	Middle		
	Back		
Search		O(n)	
Min / Max		O(n)	

Data Structure Operations

Operation		Array	Linked List
Element Access		O(1)	
Traversal	Forwards	O(n)	
	Backwards		
Add	Front	O(n)	
	Middle		
	Back		
Delete	Front	O(n)	
	Middle		
	Back		
Search		O(n)	
Min / Max		O(n)	

Data Structure Operations

Operation		Array	Linked List
Element Access		O(1)	O(n)
Traversal	Forwards	O(n)	O(n)
	Backwards		O(n^2)
Add	Front	O(n)	O(1)
	Middle		
	Back		
Delete	Front	O(n)	
	Middle		
	Back		
Search		O(n)	O(n)
Min / Max		O(n)	O(n)

Singly-Linked List

- What we've been doing
- Each node has one link direction

Data Structure Operations

Operation		Array	Singly-Linked List
Element Access		O(1)	O(n)
Traversal	Forwards	O(n)	O(n)
	Backwards		O(n^2)
Add	Front	O(n)	O(1)
	Middle		
	Back		
Delete	Front	O(n)	
	Middle		
	Back		
Search		O(n)	O(n)
Min / Max		O(n)	O(n)

Doubly-Linked List

- Each node has two link directions

```
struct Node {
  int value;
  Node *pNext;
  Node *pPrev;
};
```

Linked List Operations

1. Make a Node
2. Add/Remove a Node to the front
3. Add/Remove a Node to the back
4. Get Node `i`

Linked List Operations

1. Make a Node
2. Add/Remove a Node to the front
3. Add/Remove a Node to the back
4. Get Node i
5. Add/Remove a Node to the middle
6. Traverse the list forwards/backwards

Data Structure Operations

Operation		Array	Singly-Linked List	Doubly-Linked List
Element Access		O(1)	O(n)	O(n)
Traversal	Forwards	O(n)	O(n)	O(n)
	Backwards		O(n^2)	O(n)
Add	Front	O(n)	O(1)	O(1)
	Middle		O(n)	O(n)
	Back		O(1)	O(1)
Delete	Front	O(n)	O(1)	O(1)
	Middle		O(n)	O(n)
	Back		O(n)	O(1)
Search		O(n)	O(n)	O(n)
Min / Max		O(n)	O(n)	O(n)
Memory		n*sizeof(T) contiguous	n*(sizeof(T)+8) fragmented	n*(sizeof(T)+16) fragmented

Circularly-Linked List

- Can be singly- or doubly- linked
- Singly-
  - Tail next points to Head
- Doubly-
  - Head prev points to Tail
  - Tail next points to Head
- List operation concerns?
- Uses?

On Tap For Today

- Array Operations
- Array Concerns
- Grouping Data of Different Types
- Linked List
- Practice

To Do For Next Time

- Set5 due tomorrow
- Wednesday Quiz 5 – inheritance
- Can continue L6A for LinkedList tests

Add a Node to the Front

- Make newNode
- Set newNode value
- Set newNode next to head
- Set newNode prev to null
- Set head prev to newNode
- Set head to newNode

Traverse a List Forwards

- Create currentNode pointer
- Set to head
- while currentNode is not null
  - Access value
  - Set currentNode to next node

Traverse a List Backwards

- Create currentNode pointer
- Set to tail
- while currentNode is not null
  - Access value
  - Set currentNode to prev node

Remove a Node from the front

- Create nodeToDelete pointer
- Set to head
- Set head to head's next
- Set head prev to null
- Delete nodeToDelete

Get Node *i*

- Init counter = 0
- Create currentNode pointer set to head
- while counter < i && currentNode is not null
  - Increment counter
  - Set currentNode to next node
- If currentNode exists, return value
- Else, throw exception

Add a Node to the back

- Make newNode
- Set newNode value
- Set newNode next to null
- Set newNode prev to tail
- Set tail next to newNode
- Set tail to newNode

Remove a Node from the back

- Create nodeToDelete pointer
- Set to tail
- Set tail to tail's prev
- Set tail next to null
- Delete nodeToDelete

Add a Node to the middle

- Init counter = 0
- Create currentNode pointer set to head
- while counter < i-1 && currentNode is not null
  - Increment counter
  - Set currentNode to next node
- If currentNode exists
  - Make newNode and set value
  - Set newNode next to currentNode next
  - Set newNode prev to currentNode
  - Set currentNode next prev to newNode
  - Set currentNode next to newNode

Remove a Node from the middle

- Init counter = 0
- Create currentNode pointer set to head
- while counter < i-1 && currentNode is not null
  - Increment counter
  - Set currentNode to next node
- If currentNode exists
  - Create nodeToDelete and set to currentNode next
  - Set currentNode next to currentNode next next
  - Set currentNode next prev to currentNode
  - Delete nodeToDelete