CH3.
FUNDAMENTAL DATA STRUCTURES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
BASICS OF COMPUTER ORGANIZATION

COMPUTING, DATA, AND MEMORY
THE BASIC METHODS TO STORE DATA FROM 150

1. One variable per data element – does not associate data together and can be very verbose

2. Arrays – group a large amount of data all of the same type

3. Objects – group a large amount of data all of different types
Memory is storage for data and programs.

We will pretend that memory is an infinitely long piece of tape separated into different cells.

Each cell has an address, i.e., a location, and a value.

In the computer these values are represented in binary (0s and 1s) and addresses are located in hexadecimal (base 16, 0x).

```
x   y   z   ...   ...   ...
0x0  0x1  0x2  0xA  
```
We will review arrays in Java later today.

Arrays are a sequential piece of memory all of the same type.

Pointer (e.g., Java reference) – a variable that stores a memory location.
MEMORY
OBJECTS

• We will review objects in Java and learn new concepts/syntax about objects tomorrow.

• Objects are entities in your program. Another way to think about them is that they are collections of data of unassociated types.

• Objects are stored as pointers in Java, always.
BASIC COMPUTER ORGANIZATION

Central Processing Unit (CPU)
- Processes commands as 0's and 1's
- Performs arithmatic
- Requests (reads) and writes to/from memory

Input
- Files
- Keyboard
- Mouse
- Etc.

Memory
- Data encoded as 0s and 1s
- Cache
- Random Access Memory (RAM)
- Hard drive

Output
- Monitor
- Force feedback
- Files
- Etc.

Bus
TAKEAWAYS ABOUT MEMORY

- Programs can operate more efficiently when data is close together, e.g., arrays. This is called **locality** of data. The reason it works better is the cache.

- Pointers are not usually located close to the data. They hurt locality.
CH 3.1 ARRAYS
An **array** is a sequenced collection of variables all of the same type. Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, $A$, are numbered 0, 1, 2, and so on.

Each value stored in an array is often called an **element** of that array.

Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its **capacity**. We will always deal with compact arrays (all elements shoved to one end, with empty (null) elements at the other).
ADDING AN ENTRY

• To add an entry \( e \) into array \( A \) at index \( i \), we need to make room for it by shifting forward the \( n - i \) entries \( A[i], ..., A[n - 1] \)

• Time complexity for best, average, and worst cases?

```
Algorithm Add
Input: Array \( A \),
       index \( i \), element \( e \)
1. for \( k \leftarrow n \) to \( i + 1 \) do
2. \( A[k] \leftarrow A[k - 1] \)
3. \( A[i] \leftarrow e \)
4. \( n \leftarrow n + 1 \)
```
REMOVING AN ENTRY

• To remove the entry $e$ at index $i$, we need to fill the hole left by $e$ by shifting backward the $n - i - 1$ elements $A[i + 1], ..., A[n - 1]$

• Time complexity for best, average, and worst cases?

**Algorithm Remove**
**Input:** Array $A$, index $i$, element $e$

1. for $k ← i + 1$ to $n - 1$ do
3. $A[n - 1] ←$ null
4. $n ← n - 1$
EXERCISE
ARRAY EQUALITY

• With a partner, write and analyze an algorithm to compare the equality of two arrays $A$ and $B$. 
CH 3.2 SINGLY LINKED LISTS
LINKED STRUCTURES

• A **linked data structure** stores **nodes** that contain data and pointers to other nodes in the structure
  • Compare this to an array!

Example of a linked structure – graph (Ch 14)
SINGLY LINKED LIST

• A **singly linked list** is a concrete data structure consisting of a sequence of nodes, starting from a head pointer.

• Each node stores
  • element
  • link to the next node
**Algorithm** AddFirst

**Input**: List \( l \), Element \( e \)

1. Node \( n \leftarrow \text{new Node}(e) \) //Allocate new node \( n \) to contain element \( e \)
2. \( n.\text{next} \leftarrow l.\text{head} \) //Have new node point to old head
3. \( l.\text{head} \leftarrow n \) //Update head to point to new node

Note, for simplicity, this algorithm assumes the list has elements in it. A special case would need to be introduced for an empty list to set up the tail pointer.
**Algorithm** AddLast

**Input:** List $l$, Element $e$

1. Node $n$ ← new Node($e$) //Allocate a new node to contain element $e$
2. $n$.next ← null //Have new node point to null
3. $l$.tail.next ← $n$ //Have old last node point to new node
4. $l$.tail ← $n$ //Update tail to point to new node

Note, for simplicity, this algorithm assumes the list has elements in it. A special case would need to be introduced for an empty list to set up the head pointer.

What is the time complexity?
**REMOVING AT THE HEAD**

**Algorithm** RemoveFirst

**Input:** List $l$

1. $l$.head $\leftarrow l$.head.next //Update head to point to next node in the list

2. Allow garbage collector to reclaim the former first node

Note, for simplicity, this algorithm assumes the list has elements in it and does not return the removed element. Extra logic would be added in a real implementation.

**What is the time complexity?**
EXERCISE

• Write and analyze an algorithm for finding the second-to-last node in a singly-linked list. Assume the linked list does not store a tail pointer.
REMOVING AT THE TAIL

• Removing at the tail of a singly linked list is not efficient!
• There is no constant-time way to update the tail to point to the previous node.
// Nest this class inside of Linked list
private static class Node {
    // Private data
    private int elem;  // Element
    private Node next;  // Next node (link)

    // Constructor
    public Node(int e, Node n) {
        elem = e;
        next = n;
    }

    // Accessors
    public int getElement() { return elem; }
    public Node getNext() { return next; }
}

public class LinkedList {
    /* Place node class here */

    // Private data
    private Node head = null;  // List head
    private Node tail = null;  // List tail
    private int size = 0;  // List size

    // Constructor
    public LinkedList() {
    }

    // Accessors
    public int size() { return size; }

    // Modifiers
    public void addFirst(int e) {
        head = new Node(e, head);
        if (size == 0)
            tail = head;
        ++size;
    }

    /* Other algorithms */
}
CH 3.4 DOUBLY LINKED LISTS
DOUBLY LINKED LIST

- A **doubly linked list** can be traversed forward and backward.
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes that do not store data.
  - In linked structures, special nodes like this are called **sentinels**.
• Insert a new node, $q$, between $p$ and its successor.

• What is the time complexity?
DELETION

• Remove a node, $p$, from a doubly linked list.
• What is the time complexity?
EXERCISE

• Write and analyze an algorithm for finding the middle node of a doubly linked-list
  • With access to a method `size()`
  • Without access to a method `size()`
SUMMARY

• Two major patterns of data storage
  • Consecutive memory – localized, through arrays or objects
  • Linked memory – not localized, through linked objects