Overview

- Review: Arrays and Binary Search
- Review: Linked Lists
- Stacks via arrays & linked-lists
- Queues via arrays/linked-lists
- Doubly-linked lists

For further reading, refer to Open Data Structures Book (Chapters 2 and 3)
Data Structures vs Abstract Data Types

- A **data structure** is a model for the organization of data and their storage allocation on a computer.

- An **abstract data type** consists of:
  1. a collection of data items, and
  2. a set of allowable operations on the data items.

- In this module, we define three abstract data types: stack, queue, deque.

- For each ADT we examine different possible implementations using different data structures: array, linked list, or doubly-linked list.
**Insertion Sort**

```
int [] A = new int [8];
A[0] = 16;
...
for (int i = 0 ; i < 8 ; i++)
```

- When the control gets to `int [] A = new int [8];`, eight consecutive memory location, each enough for storing an integer (e.g., each 4 bytes), are dedicated on the main memory to store `A`.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>42</td>
<td>0</td>
<td>-3</td>
<td>1</td>
<td>50</td>
<td>63</td>
<td>-9</td>
</tr>
</tbody>
</table>
Arrays review

- Arrays tend to be easy to code.
- Given an index \( i \), accessing the corresponding array value \( A[i] \) takes \( O(1) \) time
  - This is done through random access.
  - e.g., \( \text{staff}[85].\text{getLastName}() \);
- What is the time complexity of adding an element to an array that is not currently full when order is unimportant? \( \rightarrow O(1) \)
- What is the time complexity of deleting an element if we know the element’s index in the array? \( \rightarrow O(1) \)
Static vs. Dynamic Arrays

- What is the difference between `int a[10];` and `int* a = new int a[10];`?

- For `int a[10];` an array is created at the **compile time**
  - The memory for it will be a part of the memory assigned for the code.
  - Can you have the following?
    - `Scanner scan = new Scanner(System.in);`
    - `int num = scan.nextInt();`
    - `int a[num];`
    - No because the size of the array should be known at the compile time!

- For `int a[] = new int[num];` an array is created at the **run time**; the value of `num` is not needed at the compile time.
an important property of arrays is the ability to search a sorted array in \(O(\log n)\) time

```java
// Returns the location of x in given array arr[l..r]
public static int binSearch(int[] A, int x) {
    return binSearchRec(A, 0, A.length - 1, x);
}

private int binSearchRec(int[] arr[], int lo, int hi, int x) {
    int result = -1;
    int mid = (lo + hi) / 2;
    if (lo <= hi && key == A[mid])
        result = mid;
    else if (lo <= hi && key < A[mid])
        result = binSearchHelper(A, key, lo, mid - 1);
    else if (lo <= hi)
        result = binSearchHelper(A, key, mid + 1, hi);
    return result;
}
```
Binary Search

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
2 & 5 & 6 & 8 & 9 & 9 & 12 & 15 & 17 & 20 & 21 & 22 & 29 & 31 & 32 & 45 \\
\end{array}
\]

\[
\text{binarySearch}(A, 0, 16, 22); \rightarrow \text{mid} = 8, A[8] < 22 \rightarrow \\
\text{binarySearch}(A, 9, 16, 22); \rightarrow \text{mid} = 12, A[12] > 22 \rightarrow \\
\text{binarySearch}(A, 9, 11, 22); \rightarrow \text{mid} = 10, A[10] < 22 \rightarrow \\
\text{binarySearch}(A, 11, 11, 22); \rightarrow \text{mid} = 11, A[11] = 22 \\
\text{return } 11
\]
Time complexity of binary search

- On every step, the number of possible array cells to search decreases by half.
  - Initially, \( n \) cells must be searched.
  - After the first step, at most \( n/2 \) cells remain to be searched.
  - After the second step, at most \( n/4 \) cells remain to be searched, etc.
  - After \( i \) steps, \( n/2^i \) cells remain to be searched.
  - In the worst case, the search terminates when the number of cells to be searched is one, i.e., \( n/2^i = 1 \) which gives \( n = 2^i \), that is, \( i = \log n \).

- So, in the worst case, \( O(\log n) \) steps are required, each taking constant time \( \rightarrow \) binary search takes \( O(\log n) \) in the worst-case.
Linked List

- In addition to the array data structure, you have seen the linked list
Each item in the list will be an instance of class Node:

Here is version 1 for class Node. What’s wrong with it?

```java
public class Node {
    public int value;
    public Node next; // a pointer to the next node

    // constructor which gets both value and the pointer
    public Node(int newValue, Node newNext) {
        value = newValue;
        next = newNext;
    }

    // constructor which gets only the value; next is set to null
    public Node(int newValue) {
        this(newValue, null);
    }
}
```
Linked List with Information Hiding

- Each item in the list will be an instance of class `Node`:
  - A better version with information hiding

```java
public class Node {
    protected int value;
    protected Node next;

    public Node(int newValue, Node newNext) {
        value = newValue;
        next = newNext;
    }

    public Node(int newValue) { this(newValue, null); }

    public int getValue() { return value; }
    public void setValue(int newValue) { value = newValue; }
    public Node getNext() { return next; }
    public void setNext(Node newNext) { next = newNext; }
}
```
Linked List with Generic Types

- Each item in the list will be an instance of class Node:
  - An even better version which supports **generic types**:
  - Here E is a parameter that can be an int, double, etc.

```java
public class Node<E> {
    protected E value;
    protected Node<E> next;
    public Node(E newValue, Node<E> newNext) {
        value = newValue;
        next = newNext;
    }
    public Node(E newValue) { this(newValue, null); }
    public E getValue() { return value; }
    public void setValue(E newValue) { value = newValue; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> newNext) { next = newNext; }
}
```
Linked List

- A linked list is basically a pointer (named head) to a node
- Plus a set of operations like insert and deleted.

```java
public class LinkedList<E> {
    protected Node<E> head;
    public LinkedList() { head = null; }
    public void insert(E newValue) {
        head = new Node<E>(newValue, head);
    }
    public void delete() {
        if (head != null) head = head.getNext();
    }
}
```
Linked List example

```
LinkedList<Integer> myList = new LinkedList<Integer>();
myList.insert(42);
myList.insert(-30);
myList.insert(16);
myList.delete();
```
## Dictionary Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unordered Array</th>
<th>Ordered Array</th>
<th>Unordered Linked List</th>
<th>Ordered Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert/delete at start</td>
<td>O(n)</td>
<td>N/A</td>
<td>0(1)</td>
<td>N/A</td>
</tr>
<tr>
<td>insert/delete at end</td>
<td>O(1)</td>
<td>N/A</td>
<td>0(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>search</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>insert/delete at current position</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>N/A</td>
</tr>
<tr>
<td>search and insert/delete</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Stack ADT

- Stack is an abstract data type
  - **Data:** a collection of elements
  - **Operations:** push(item), pop(), isEmpty(), top()
- Stack works based on the LIFO (Last In First Out) principle
- Stack Examples:
  - keeping track of function calls (e.g., recursion)
  - parsing brackets in a mathematical expression
  - a web browser’s “back” button
  - passengers boarding/exiting subway car
  - “undo” button on word processor
Stack Example

isEmpty(); → true  push(20); push(25); push(12); isEmpty();
→ false  pop(); → 12  push(60); pop(); → 60  pop(); → 25
isEmpty(); → false  pop(); → 20  isEmpty(); → true
Like any other abstract data type, a stack needs to be implemented via a data structure.

We often use an array or a linked list to implement stacks.
Implementing a Stack Using an Array

- index points to the next empty location in the array.

The stack contains four items:

\[
\begin{array}{cccc}
12 & 99 & 30 & 5 \\
\end{array}
\]

We pop the top item:

\[
\begin{array}{cccc}
12 & 99 & 30 & \\
\end{array}
\]
public class Stack implements StackADT<Integer> {
    protected Integer[] data;
    protected int index;
    protected int size;

    public Stack(int newSize) {
        size = newSize;
        data = new Integer[size];
        index = 0;
        // initially empty
    }

    public void push(Integer item) {
        if (index < size)
            data[index++] = item;
    }

    public boolean isEmpty() {
        return (index == 0);
    }

    public Integer top() {
        Integer topItem = null;
        if (!isEmpty())
            topItem = data[index -1];
        return topItem;
    }

    public Integer pop() {
        Integer topItem = top();
        if (!isEmpty())
            index--;
        return topItem;
    }
}
Stack Code Example

```java
Stack st = new Stack(10);

st.push(25);

st.push(5);

Integer x = st.pop();  // 5 is returned

st.push(8);

boolean test = st.isEmpty();  // false

x = st.pop();

x = st.pop();

test = st.isEmpty();  // true
```
Implementing Stacks with Linked Lists

Initially, the stack is empty
Push(21)
Push(42)
Push(30)
Push(16)
pop() → 16 is returned
public class Stack<E> implements StackADT<E> {
    protected Node<E> head;
    public Stack() { head = null; }

    public boolean isEmpty() { return (head == null); }
    public void push(E item) {
        head = new Node<E>(item, head);
    }
    public E top() {
        E topItem = null;
        if (head != null)
            topItem = head.getValue();
        return topItem;
    }
    public E pop() {
        E topItem = top();
        if (head != null)
            head = head.getNext();
        return topItem;
    }
}
Implementing a Stack Using Linked List

```java
Stack<Integer> st = new Stack<Integer>();

st.push(25);

st.push(5);

Integer x = st.pop();

st.push(8);

boolean test = st.isEmpty(); // (false is returned)

x = st.pop();

x = st.pop();

test = st.isEmpty(); // (true is returned)
```
Arrays vs. Linked List for Stacks

What are the time complexities of push(), pop(), and isEmpty?

- In an array, you just need to increment/decrement the “index” for push/pop; for isEmpty, just look at it → all operations in constant time if the array is not full
- In a linked list, you just need to update the head pointer for push/pop (and create a new node for push); is Empty is just checking the head pointer → all constant time

- The array size is fixed. If the number of pushed items exceed the size of the array, it becomes **full**. In case of pushing an item to a full array, you need to copy all elements to a larger array which takes \( O(n) \) time → linked lists have a slight advantage over arrays!
Arrays & Linked for Stacks Summary

- Both arrays and linked lists can be used to implement stacks
- In most cases, all operations (push, pop, top, isEmpty) take constant time
  - In case of arrays, once the array become full, a push requires creating a larger array and copying all items from the old to the new ones
  - This means that in the worst case, push takes $O(n)$ time.
Queue ADT

- Queue is an abstract data type
  - **Data**: a collection of elements
  - **Operations**: enqueue(item), dequeue(), isEmpty(), first()

- Queue works based on the FIFO (First In First Out) principle

- Queue Examples:
  - telephone operator
  - serving customers at the cashier
  - Tim Hortons line
  - cars driving on a single-lane road

- Like Stacks, we can use either arrays or linked lists to implement queues
Implementing a Queue Using an Array

- As always, array has a fixed size
- There are two pointers, head and tail
  - Head points to the first empty location before the head of the queue
  - Tail points to the first empty location after the tail of the queue.
Implementing a Queue Using an Array

The queue is initially empty:

- head = 0
- tail = 1

The queue contains three items:

- 34
- 7
- 55

- tail = 4

We dequeue the item at the head:

- head = 1
- 34

We enqueue 10:

- 7
- 55
- 10

- tail = 5

After three more calls to dequeue the queue is again empty:

- 7
- 55
- 10

- tail = 5
Implementing a Queue Using an Array

- ‘head’ and ‘tail’ are *cyclic*
  - The location before the first index is \( m - 1 \) (array has size \( m \))
  - The location after the last index is 0

![Cyclic Queue Diagram]

Works like a circular array
Implementing a Queue Using an Array

```java
public class Queue implements QueueADT<Integer> {
    protected Integer[] data;
    protected int head;
    protected int tail;
    protected int size;
    // size of the array
    protected int current;
    // number of items in the queue

    public Queue(int newSize) {
        size = newSize;
        data = new Integer[size];
        head = 0;
        tail = 1;
        current = 0;
    }

    public boolean isEmpty() {
        return (current == 0); }

    public void enqueue(Integer item) {
        if (current < size) {
            data[tail] = item;
            current++;
            tail = (tail + 1) % size
        }
    }

    public Integer dequeue() {
        Integer firstItem = null;
        if (!isEmpty()) {
            head = (head + 1) % size;
            current--;
            firstItem = data[head];
        }
        return firstItem;
    }
}
```
Implementing a Queue Using an Array

Queue q = new Queue(3);
q.enqueue(19);
q.enqueue(21);
q.enqueue(35);
q.enqueue(43); (full; the queue does not change)
Integer x = q.dequeue();
q.enqueue(58);
x = q.dequeue();
x = q.dequeue();
Implementing a Queue Using a Linked List

- We require two pointers, head and tail that point to the first and last nodes in the list.

The queue is initially **empty**:

We **enqueue** 16:

After three more calls to **enqueue**:

We call **dequeue**:
public class Queue<E> implements QueueADT<E> {
    protected Node<E> head, tail;

    public Queue() {
        head = null;
        tail = null;
    }

    public void enqueue(E item) {
        Node<E> temp = new Node<E>(item);
        if (tail != null)
            tail.setNext(temp);
        tail = temp;
        if (head == null)
            head = tail;
    }

    public boolean isEmpty() {
        return (head == null);
    }

    public E dequeue() {
        E firstItem = null;
        if (!isEmpty()) {
            firstItem = head.getValue();
            head = head.getNext();
            if (head == null)
                tail = null;
        }
        return firstItem;
    }
}
Implementing a Queue Using a Linked List

Queue<Integer> q = new Queue<Integer>();
q.enqueue(16);
q.enqueue(21);
q.enqueue(35);
q.enqueue(43);
Integer x = q.dequeue();
q.enqueue(58);
x = q.dequeue();
Arrays vs. Linked List for Queues

- What are the time complexities of enqueue, dequeue, and isEmpty?
  - In an array, you just need to increment/decrement two “indices” for head/tail for enqueue/dequeue and increment/decrement current; for isEmpty, check the value of current → all operations in constant time if the array is not full.
  - In a linked list, you just need to update the head/tail pointers (and create a new node for enqueue); is Empty is just checking the head pointer → all constant time

- The array size is fixed. If the number of items exceed the size of the array, it becomes full. In case of enqueuing an item to a full array, you need to copy all elements to a larger array which takes $O(n)$ time → linked lists have a slight advantage over arrays!
Arrays & Linked for Queues Summary

- Both arrays and linked lists can be used to implement queues.
- In most cases, all operations take constant time.
  - In case of arrays, once the array become full, an enqueue requires creating a larger array and copying all items from the old to the new ones.
  - This means that in the worst case, enqueue takes $O(n)$ time.
Doubly-linked Lists

- A data structure similar to linked list
  - Each node contains an extra pointer called previous pointer, along with next pointer and data which are there in singly linked list.
Implementing a Stack Using an Array

public class DoubleNode<E> {
    protected E value;
    protected DoubleNode<E> next, previous;
    public DoubleNode(E newValue, DoubleNode<E> newNext, DoubleNode<E> newPrevious) {
        ... }
    public E getValue() { return value; }
    ...
}

public class DoublyLinkedList<E> {
    protected DoubleNode<E> head, tail;
    ...
}

an alternative implementation with inheritance

public class DoubleNode<E> extends Node<E> {
    protected DoubleNode<E> previous;
    public DoubleNode(E newValue, DoubleNode<E> newNext, DoubleNode<E> newPrevious) {
        super(E, newNext);
        previous = newPrevious;
    }
    public DoubleNode(E newValue) {
        this(newValue, null, null);
    }
}
Doubly-linked Lists

- Inserting an item with value $x$ to the front of the list:
  - step 0: Create a new DoubleNode $N$ with the $x$ as its value
  - step 1: Let the current head’s previous point to $N$
  - step 2: Let $N$’s head point to current head.
  - step 3: Let head point to $N$.

- What is the time complexity of insert?
  - Like linked list, inserting at front takes constant time (just update a few pointers)
  - The time is independent of the length of the list $\rightarrow O(1)$
Doubly-linked Lists

- Deleting an item from the front of the list (similar to linked lists):
  - Let head to point to whatever its next is.
  - Let head’s previous to be Null.
- Time complexity is independent of the length of the list $\rightarrow O(1)$. 
Doubly-linked Lists Summary

- Maintaining an additional pointer takes a little extra space compared to linked lists.
- The time complexity of insert/deleted at the beginning of the list remain unchanged.
- If you need to delete a node $X$, and you only have a pointer to $X$ in the doubly linked list, you can delete $X$ in constant time.
  - Because you can access its previous node $W$ and let it point to its next node $Y$.
  - In the regular linked lists you have to traverse the whole list!
Deque (Double Ended Queue)

- A generalization of Stack and Queues!
  - `addFront(val)` → add a node with value val to the front of the queue (similar to push)
  - `removeFront` → remove the node at the front of the queue (similar to pop and dequeue)
  - `addBack` → add a node with value val to the back of the queue (similar to dequeue)
  - `removeBack` → remove the node at the front of the queue
Deque (Double Ended Queue)

- We can insert an item at either end and remove an element from either end.
- Like a queue, a deque can be implemented using either an array or a doubly-linked list.
  - You will do it in Assignment 3 :)
- Stack is a deque with only addFront (push) and removeFront (pop) operations.
- Queue is a deque with only addBack (enqueue) and removeFront (dequeue) operations.