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)
• We know essentially one way to store data at the moment, sequentially with arrays or objects.

• Today we will look at another common methodology that encompasses the second major storage pattern for data structures – linked structures.
SINGLY LINKED LISTS
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

![Diagram of a singly linked list with nodes A, B, C, D, and head, with arrows indicating the direction of the next node.]
**INSERTING AT THE HEAD**

**AddFirst**

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

1. \( n \leftarrow \text{Node}(e) \) //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
INSERTING AT THE TAIL

**AddLast**

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

1. $n \leftarrow \text{Node}(e)$ //Allocate a new node to contain element $e$
2. $n.next \leftarrow \text{null}$ //Have new node point to null
3. $l.tail.next \leftarrow n$ //Have old last node point to new node
4. $\text{tail} \leftarrow n$ //Update tail to point to new node
REMOVING AT THE HEAD

RemoveFirst

Input: List l

1. \( l.\text{head} \leftarrow l.\text{head}.\text{next} \) //Update head to point to next node in the list

2. Allow garbage collector to reclaim the former first node
EXERCISE

• Write an algorithm for finding the second-to-last node in a singly-linked list. The last node is indicated by a null next reference.
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.
DOUBLY LINKED LISTS
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**
• Insert a new node, \( q \), between \( p \) and its successor.
DELETION

- Remove a node, \( p \), from a doubly linked list.
EXERCISE

• Write an algorithm for finding the middle node of a doubly linked-list
  • With access to a method `size()`
  • Without access to a method `size()`
PROGRAMMING A GENERIC SINGLY-LINKED LIST

• Follow along as we design and implement a generic singly-linked list in Java
**CLASS DESIGN**

**DATA**

- Classes can own other classes as part of their "type." This is called **nesting**.
- It will be common in this class to have this kind of pattern

**SinglyLinkedList**

- Head node
- Tail node
- Size

**Node (Nested class)**

- Element
- Next node
CLASS DESIGN
PUBLIC FUNCTIONS

• Singly-linked List
  • SinglyLinkedList()
  • size()
  • isEmpty()
  • Element first()
  • Element last()
  • addFirst(Element e)
  • addLast(Element e)
  • Element removeFirst()

• Node
  • Node(Element e, Node next)
  • getElement()
  • getNext()
  • setNext(Node n)
FOLLOW ALONG AS WE CODE

1. First, write a test main function to exercise usage of the class
2. Second create empty public interface of the singly-linked list
3. Fill in the details and make the test pass
SinglyLinkedList\<\text{E}\> l = new SinglyLinkedList\<\text{E}\>();
l.addFirst(5);
l.addFirst(6);
System.out.println("Size after two adds: "+ l.size());
System.out.println("First: "+ l.first());
System.out.println("Last: "+ l.last());
System.out.println();
l.addLast(7);
System.out.println("Size after add last: "+ l.size());
System.out.println("Last: "+ l.last());
System.out.println();
l.removeFirst();
System.out.println("Size after removeFirst: "+ l.size());
System.out.println("First: "+ l.first());
ADD THE INTERFACE NOW

```java
/**
 * Constructor
 */
public SinglyLinkedList() {
}

/**
 * @return Size
 */
public int size() {
    return 0;
}

/**
 * @return Size == 0
 */
public boolean isEmpty() {
    return false;
}

/**
 * @return Head element
 */
public E first() {
    return (E)(new Object());
}

/**
 * @return Tail element
 */
public E last() {
    return (E)(new Object());
}

/**
 * Insert element at front of list
 * @param e Element
 */
public void addFirst(E e) {
}

/**
 * Insert element at back of list
 * @param e Element
 */
public void addLast(E e) {
}

/**
 * Remove first element of list
 * @return Removed element
 */
public E removeFirst() {
    return (E)(new Object());
}
```
ADD OUR NESTED CLASS
MAKE SURE IT IS INSIDE OF THE SINGLYLINKEDLIST CLASS

```java
/**
 * Nested node class.
 * @param <E> Element type
 */
private static class Node<E> {

    /**
     * Element
     */
    private E elem;

    /**
     * Next node
     */
    private Node<E> next;

    /**
     * Constructor.
     * @param e Element
     * @param n Next node
     */
    public Node(E e, Node<E> n) {
        elem = e;
        next = n;
    }

    /**
     * @return Element
     */
    public E getElement() {
        return elem;
    }

    /**
     * @return Next node
     */
    public Node<E> getNext() {
        return next;
    }

    /**
     * Set next node
     * @param n New next node
     */
    public void setNext(Node<E> n) {
        next = n;
    }
}
```
ADD DATA TO THE LINKED LIST

/**
 * Head
 */
private Node<E> head = null;
/**
 * Tail
 */
private Node<E> tail = null;
/**
 * Size
 */
private int size = 0;
/**
 * @return Size
 */
public int size() {
    return size;
}

/**
 * @return Size == 0
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * @return Head element
 */
public E first() {
    if(isEmpty()) return null;
    return head.getElement();
}

/**
 * @return Tail element
 */
public E last() {
    if(isEmpty()) return null;
    return tail.getElement();
}

/**
 * Insert element at front of list
 * @param e Element
 */
public void addFirst(E e) {
    head = new Node<>(e, head);
    if(size == 0)
        tail = head;
    ++size;
}

/**
 * Insert element at back of list
 * @param e Element
 */
public void addLast(E e) {
    Node<E> n = new Node<>(e, null);
    if(size == 0)
        head = n;
    else
        tail.setNext(n);
    tail = n;
    ++size;
}
/**
 * Remove first element of list
 * @return Removed element
 */

public E removeFirst() {
    if(isEmpty()) return null;
    E e = head.getElement();
    head = head.getNext();
    --size;
    if(size == 0)
        tail = null;
    return e;
}
SUMMARY

• Two major patterns of data storage
  • Sequential, localized – through arrays or objects
  • Linked, not localized – through linked lists/objects