Linked Structures, LinkedStack

- References as Links
- Data Encapsulation Separate from Linking
- Linear Linked Lists and Non-linear Structures
- Managing Linked Lists
- Doubly Linked Lists
- Reading: L&C 4.1-4.3, 4.6
References as Links

• A *linked structure* is a data structure that uses object reference variables to create links between objects

• Declaring an object reference variable
  
  ```java
  Object obj = new Object();
  ```

• A diagram of an object reference variable

  ![Diagram of object reference variable]
Linear/Non-Linear Linked Structures

• Some linked structures are non-linear, e.g.

• A “linked list” is considered to be a linear structure since it can be visualized as a line
References as Links

• It is desirable to have a generic link class that can be used to link any type of objects of any class encapsulating the “real” data

• Class `LinearNode<T>` does that this way
References as Links

- A *self-referential* (or *recursive*) `LinearNode<T>` object has a pointer to another `LinearNode<T>` object

```java
public class LinearNode<T> {
    // attributes of the data in an element object
    private T element        // encapsulated element

    // link (or pointer) to another LinearNode object
    private LinearNode<T> next; // next item in list
```
References as Links

// constructor
public LinearNode(T element) // encapsulate
    {
    this.element = element // the element
        next = null; // set the link to null
    
    }

// accessor and mutator methods
public void setNext(LinearNode<T> next)
    {
    this.next = next;
    
    }
public LinearNode<T> getNext()
    {
    return next;
    
    }
public T getElement()
{
    return element;
}
public void setElement(T element)
{
    this.element = element;
}
}// end class LinearNode<T>
Linked Lists

• Unlike an array which has a fixed size, a linked list is considered to be a dynamic memory structure
• The amount of memory used grows and shrinks as objects are added to the list or deleted from the list
• The Java compiler and virtual machine allocate memory for each individual object as it is created (via the new operator) and free memory as each object is garbage collected
Managing Singly Linked Lists

• The order in which references are changed is crucial to maintaining a linked list
• Can insert a new LinearNode in two places:
  – At the front
  – In the middle or at the end
• Can delete a LinearNode in two places:
  – At the front
  – In the middle or at the end
Inserting Objects in a Linked List

- Create new LinearNode object and link at top

```java
LinearNode<T> newNode = new LinearNode<T>(element);
newNode.setNext(top);
top = newNode;
newNode = null; // may be needed for stale reference
```

// if newNode won’t go out of scope

Inserting Objects in a Linked List

- Create new LinearNode object
- Link after current LinearNode object (current could be at the end)

```java
LinearNode<T> newNode = new LinearNode<T>(element);
newNode.setNext(current.getNext());
current.setNext(newNode);
newNode = null;
```
Removing Objects from a Linked List

- Remove LinearNode object at front

```java
LinearNode<T> removed = top;
top = top.getNext();
removed.setNext(null);  // remove stale reference
```

![Diagram showing removal process of linear nodes from a linked list.](image-url)
Removing Objects from a Linked List

- Remove `LinearNode` after `current LinearNode` object (removed object could be at end)

```java
LinearNode<T> removed = current.getNext();
current.setNext(removed.getNext());
removed.setNext(null);  // remove stale reference
```
Linked Stack Implementation

• We can use the LinearNode class to implement a Stack using linking
• We use the attribute name “top” to have a meaning consistent with a stack
Linked Stack Implementation

• **push – O(1)**

```java
public void push (T element)
{
    LinearNode<T> temp = new LinearNode<T>(element);
    temp.setNext(top);
    top = temp;
    count++;
}
```

• **Note difference between the LinkedStack push method and ArrayStack push method**
Linked Stack Implementation

- pop – O(1)
  
```java
public T pop () throws EmptyStackException {
    if (isEmpty()) throw new EmptyStackException();
    T result = top.getElement();
    top = top.getNext();  // LinearNode is garbage now
    count--;
    return result;
}
```

- Note difference between the LinkedStack pop method and ArrayStack pop method
LinkedStack Implementation

• Notice that we don’t need an expandCapacity method in our LinkedStack implementation
  – The “new” operator called in the push method automatically allocates the memory for each LinearNode object when it is needed
  – When the reference to the LinearNode at top is overwritten in the pop method, the JVM garbage collector will release the memory for the now unneeded LinearNode
Doubly Linked Lists

• Each `DoubleNode` object has a reference to next `DoubleNode` and previous `DoubleNode`

```java
class DoubleNode<T> {
    private DoubleNode<T> next;
    private DoubleNode<T> prev;
    private T element;
}
```

- `null` is used to indicate the first node in the list.
- `next` and `prev` point to the next and previous nodes.
- `element` holds the object of type `T` at each node.
Doubly Linked Lists

• To add a DoubleNode object to the list, your code must set the DoubleNode `next` and `prev` variables in both the new node and its adjacent neighbors.

• To delete a DoubleNode object from the list, your code must bypass the DoubleNode `next` and `prev` variables in both neighbors adjacent to the removed node and may need to set its two stale references to `null`.
Doubly Linked List Technique

- To simplify the code for adding and removing a node to/from the list we can use a dummy node at the front and the back of the list
- Real data is in nodes linked between them
- The dummy nodes are ignored in searching the list for data

```
null  ---  T  ---  T  ---  null
   |    |    |    |
   front  Dummy  T  Dummy  back
```

count = 2 (for real data)
Traversing Linked Lists

• We can use “for” or “while” loops to traverse a linked list or a double linked list - examples:

```java
for(LinearNode<T> node = front; node != null; node = node.getNext()) { . . . }
```

```java
for(DoubleNode<T> node = back; node != null; node = node.getPrev()) { . . . }
```

```java
LinearNode<T> node = front; // or DoubleNode back
while(node != null)
{
    . . .
    node = node.getNext(); // or DoubleNode prev
}
```