CSC 321: Data Structures

Fall 2018

Linked structures

- nodes & recursive fields
- singly-linked list
- doubly-linked list
- LinkedList implementation
- iterators

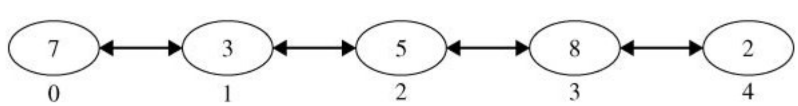
1

ArrayLists vs. LinkedLists

to insert or remove an element at an interior location in an ArrayList requires shifting data → O(N)

LinkedList is an alternative structure

- stores elements in a sequence but allows for more efficient interior insertion/deletion
- elements contain links that reference previous and successor elements in the list



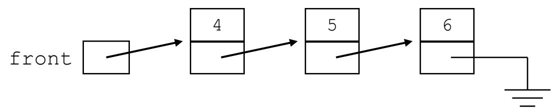
- can add/remove from either end in O(1)
- if given reference to an interior element, can reroute the links to add/remove in O(1)

2

Baby step: singly-linked list

let us start with a simpler linked model:

- must maintain a reference to the front of the list
- each node in the list contains a reference to the next node



analogy: human linked list

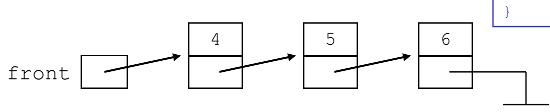
- I point to the front of the list
- each of you stores a number in your left hand, point to the next person with right

Recursive structures

recall: all objects in Java are references

- we think of the box as the Node, but really the Node is a reference to the box
- each Node object stores data and (a reference to) another Node
- can provide a constructor and methods for accessing and setting these two fields

```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```



Exercises

to create an empty linked list:

```
front = null;
```

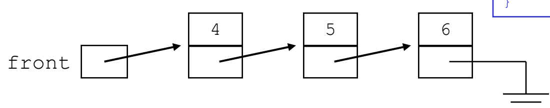
to add to the front:

```
front = new Node<Integer>(3, front);
```

remove from the front:

```
front = front.getNext();
```

```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```



5

Exercises

get value stored in first node:

get value in kth node:

indexOf:

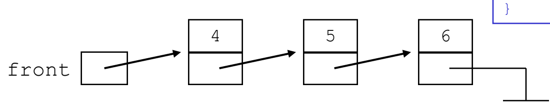
add at end:

add at index:

remove:

remove at index:

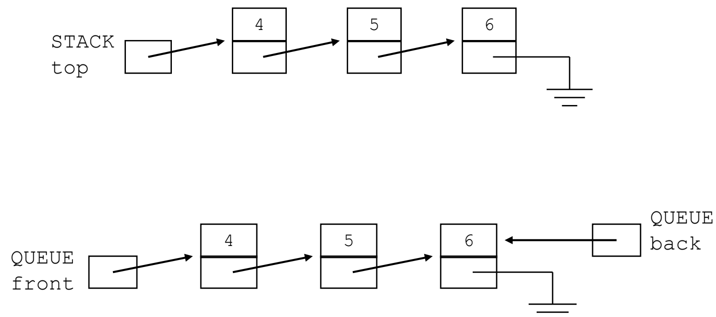
```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```



6

Linked stacks & queues

singly-linked lists are sufficient for implementing stacks & queues



7

Linked stack implementation

```
public class LinkedStack<E> {
    private Node<E> top;
    private int numNodes;

    public LinkedStack() {
        this.top = null;
        this.numNodes = 0;
    }

    public boolean empty() {
        return (this.size() == 0);
    }

    public int size() {
        return this.numNodes;
    }

    public E peek() throws java.util.NoSuchElementException {
        if (this.empty()) {
            throw(new java.util.NoSuchElementException());
        }
        else {
            return this.top.getData();
        }
    }

    . . .
```

efficient to keep track of current size in a field – must update on each push/pop

a method that attempts to access an empty stack should throw a NoSuchElementException

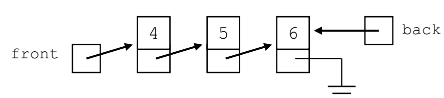
8

Linked stack implementation

```
  . . .

   public void push(E value) {
       this.top = new Node<E>(value, this.top);
       this.numNodes++;
   }

   public E pop() throws java.util.NoSuchElementException {
       if (this.empty()) {
           throw(new java.util.NoSuchElementException());
       }
       else {
           E topData = this.top.getData();
           this.top = this.top.getNext();
           this.numNodes--;
           return topData;
       }
   }

}
```

9

Linked queue implementation




```
public class LinkedQueue<E> {
    private Node<E> front;
    private Node<E> back;
    private int numNodes;

    public LinkedQueue() {
        this.front = null;
        this.back = null;
        this.numNodes = 0;
    }

    public boolean empty() {
        return (this.size() == 0);
    }

    public int size() {
        return this.numNodes;
    }

    public E peek() throws java.util.NoSuchElementException {
        if (this.empty()) {
            throw(new java.util.NoSuchElementException());
        }
        else {
            return this.front.getData();
        }
    }
    . . .
```

efficient to keep track of current size in a field – must update on each add/remove

a method that attempts to access an empty queue should throw a NoSuchElementException

10

Linked queue implementation

```
    . . .

    public void add(E value) {
        Node<E> toBeAdded = new Node<E>(value, null);
        if (this.back == null) {
            this.back = toBeAdded;
            this.front = this.back;
        }
        else {
            this.back.setNext(toBeAdded);
            this.back = toBeAdded;
        }
        this.numNodes++;
    }

    public E remove() throws java.util.NoSuchElementException {
        if (this.empty()) {
            throw(new java.util.NoSuchElementException());
        }
        else {
            E frontData = this.front.getData();
            this.front = this.front.getNext();
            if (this.front == null) {
                this.back = null;
            }
            this.numNodes--;
            return frontData;
        }
    }
}
```

normally, adding only affects the back (unless empty)

normally, removing only affects the front (unless remove last item)

11

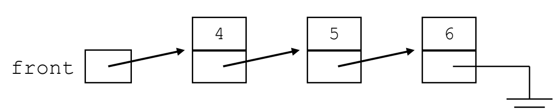
LinkedList implementation

we could implement the LinkedList class using a singly-linked list

- however, the one-way links are limiting

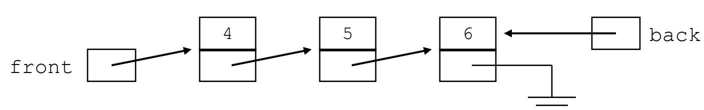
- to insert/delete from an interior location, really need a reference to the previous location

  i.e., remove(item) must traverse and keep reference to previous node, so that when the correct node is found, can route links from previous node



- also, accessing the end requires traversing the entire list

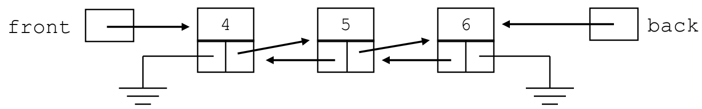
  can handle this one special case by keeping a reference to the end as well



12

Doubly-linked lists

a better, although more complicated solution, is to have bidirectional links



- to move forward or backward in a doubly-linked list, use previous & next links
- can start at either end when searching or accessing
- insert and delete operations need to have only the reference to the node in question
- big-Oh?

`add(item)`	`add(index, item)`
`get(index)`	`set(index, item)`
`indexOf(item)`	`contains(item)`
`remove(index)`	`remove(item)`

13

Exercises

to create an empty list:

```
front = null;
back = null;
```

to add to the front:

```
front = new Dnode<Integer>(3, null, front);
if (front.getNext() == null) {
    back = front;
}
else {
   front.getNext.setPrevious(front);
}
```

remove from the front:

```
front = front.getNext();
if (front == null) {
    back = null;
}
else {
    front.setPrevious(null);
}
```

```
public class DNode<E> {
    private E data;
    private DNode<E> previous;
    private DNode<E> next;

    public DNode(E d, DNode<E> p, DNode<E> n) {
        this.data = d;
        this.previous = p;
        this.next = n;
    }

    public E getData() {
        return this.data;
    }

    public DNode<E> getPrevious() {
        return this.previous;
    }

    public DNode<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setPrevious(DNode<E> newPrevious) {
        this.previous = newPrevious;
    }

    public void setNext(DNode<E> newNext) {
        this.next = newNext;
    }
}
```

14

Exercises

get value stored in first node:

get value in kth node:

indexOf:

add at end:

add at index:

remove:

remove at index:

```
public class DNode<E> {
    private E data;
    private DNode<E> previous;
    private DNode<E> next;

    public DNode(E d, DNode<E> p, DNode<E> n) {
        this.data = d;
        this.previous = p;
        this.next = n;
    }

    public E getData() {
        return this.data;
    }

    public DNode<E> getPrevious() {
        return this.previous;
    }

    public DNode<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setPrevious(DNode<E> newPrevious) {
        this.previous = newPrevious;
    }

    public void setNext(DNode<E> newNext) {
        this.next = newNext;
    }
}
```


Dummy nodes

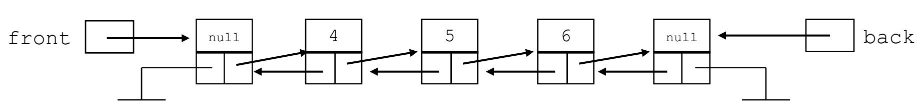
every time you add/remove, you need to worry about updating front & back

- add only affects the back, unless the list is empty (then, `front = back;`)
- remove only affects the front, unless the list becomes empty (then, `back = null;`)

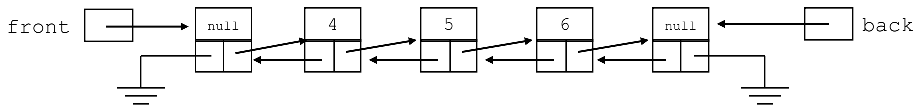
the ends lead to many special cases in the code

SOLUTION: add dummy nodes to both ends of the list

- the dummy nodes store no actual values
- instead, they hold the places so that the front & back never change
- removes special case handling



Exercises



to create an empty list (with dummy nodes):

```
front = new DNode<Integer>(null, null, null);
back = new DNode<Integer>(null, front, null);
front.setNext(back);
```

remove from the front:

```
front.setNext(front.getNext().getNext());
front.getNext().setPrevious(front);
```

add at the front:

```
front.setNext(new DNode<Integer>(3, front, front.getNext());
front.getNext().getNext().setPrevious(front.getNext());
```

get value stored in first node:

get value in kth node:

indexOf:

add at end:

add at index:

remove:

remove at index:

17

LinkedList class structure

the LinkedList class has an inner class

- defines the DNode class

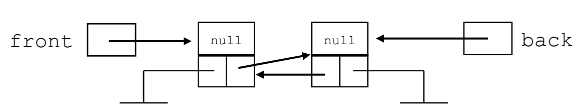
fields store

- reference to front and back dummy nodes
- node count

the constructor

- creates the front & back dummy nodes
- links them together
- initializes the count

```
public class MyLinkedList<E> implements Iterable<E>{
    private class DNode<E> {
        . . .
    }

    private DNode<E> front;
    private DNode<E> back;
    private int numStored;

    public MyLinkedList() {
        this.clear();
    }

    public void clear() {
        this.front = new Dnode<E>(null, null, null);
        this.back = new Dnode<E>(null, front, null);
        this.front.setNext(this.back);
        this.numStored = 0;
    }
```



18

LinkedList: add

the add method

- similarly, throws an exception if the index is out of bounds
- calls the helper method getNode to find the insertion spot
- note: getNode traverses from the closer end
- finally, inserts a node with the new value and increments the count

add-at-end similar

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "LinkedList add()", this.size());

    DNode<E> beforeNode = this.getNode(index-1);
    DNode<E> afterNode = beforeNode.getNext();

    DNode<E> newNode = new DNode<E>(newItem,beforeNode,afterNode);
    beforeNode.setNext(newNode);
    afterNode.setPrevious(newNode);

    this.numStored++;
}

private DNode<E> getNode(int index) {
    if (index < this.numStored/2) {
        DNode<E> stepper = this.front;
        for (int i = 0; i <= index; i++) {
            stepper = stepper.getNext();
        }
        return stepper;
    }
    else {
        DNode<E> stepper = this.back;
        for (int i = this.numStored-1; i >= index; i--) {
            stepper = stepper.getPrevious();
        }
        return stepper;
    }
}


public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

19

LinkedList: size, get, set, indexOf, contains

size method

- returns the item count

get method

- checks the index bounds, then calls getNode

set method

- checks the index bounds, then assigns

indexOf method

- performs a sequential search

contains method

- uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "LinkedList get()", this.size()-1);
    return this.getNode(index).getData();
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "LinkedList set()", this.size()-1);
    DNode<E> oldNode = this.getNode(index);
    E oldItem = oldNode.getData();
    oldNode.setData(newItem);
    return oldItem;
}

public int indexOf(E oldItem) {
    DNode<E> stepper = this.front.getNext();
    for (int i = 0; i < this.numStored; i++) {
        if (oldItem.equals(stepper.getData())) {
            return i;
        }
        stepper = stepper.getNext();
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

20

LinkedList: remove

the remove method

- checks the index bounds
- calls getNode to get the node
- then calls private helper method to remove the node

the other remove

- calls indexOf to find the item, then calls remove(index)

```
public void remove(int index) {
    this.rangeCheck(index, "LinkedList remove()", this.size()-1);
    this.remove(this.geNode(index));
}

public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}

private void remove(DNode<E> remNode) {
    remNode.getPrevious().setNext(remNode.getNext());
    remNode.getNext().setPrevious(remNode.getPrevious());
    this.numStored--;
}
```

could we do this more efficiently?

do we care?

Collections & iterators

many algorithms are designed around the sequential traversal of a list

- ArrayList and LinkedList implement the List interface, and so have get() and set()
- ArrayList impementations of get() and set() are O(1)
- however, LinkedList implementations are O(N)

```
for (int i = 0; i < words.size(); i++) {        // O(N) if ArrayList
    System.out.println(words.get(i));            // O(N²) if LinkedList
}
```

philosophy behind Java collections

1. a collection must define an <u>efficient</u>, general-purpose traversal mechanism
2. a collection should provide an *iterator*, that has methods for traversal
3. each collection class is responsible for implementing iterator methods

Iterator

the `java.util.Iterator` interface defines the methods for an iterator

```
interface Iterator<E> {
    boolean hasNext();     // returns true if items remaining
    E next();              // returns next item in collection
    void remove();         // removes last item accessed
}
```

any class that implements the Collection interface (e.g., List, Set, …) is required to provide an `iterator()` method that returns an iterator to that collection

```
List<String> words;
. . .
Iterator<String> iter = words.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

both ArrayList and LinkedList implement their iterators efficiently, so O(N) for both

ArrayList iterator

an ArrayList does not really need an iterator

- get() and set() are already O(1) operations, so typical indexing loop suffices
- provided for uniformity (`java.util.Collections` methods require *iterable* classes)
- also required for enhanced for loop to work

to implement an iterator, need to define a new class that can

- access the underlying array (→ must be inner class to have access to private fields)
- keep track of which location in the array is "next"

"foo"	"bar"	"biz"	"baz"	"boo"	"zoo"
0	1	2	3	4	5

nextIndex: 0

MyArrayList iterator

`java.lang.Iterable` interface declares that the class has an iterator

inner class defines an Iterator class for this particular collection (accessing the appropriate fields & methods)

the iterator() method creates and returns an object of that class

```
public class MyArrayList<E> implements Iterable<E> {

    . . .

    public Iterator<E> iterator() {
        return new ArrayListIterator();
    }

    private class ArrayListIterator implements Iterator<E> {
      private int nextIndex;
      public ArrayListIterator() {
          this.nextIndex = 0;
      }

      public boolean hasNext() {
          return this.nextIndex < MyArrayList.this.size();
      }

      public E next() {
          if (!this.hasNext()) {
              throw new java.util.NoSuchElementException();
          }
          this.nextIndex++;
          return MyArrayList.this.get(nextIndex-1);
      }

      public void remove() {
         if (this.nextIndex <= 0) {
             throw new RuntimeException("Iterator call to " +
                      "next() required before calling remove()");
         }
         MyArrayList.this.remove(this.nextIndex-1);
         this.nextIndex--;
      }
   }
}
```


Iterators & the enhanced for loop

given an iterator, collection traversal is easy and uniform

```
MyArrayList<String> words;
. . .
Iterator<String> iter = words.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

as long as the class implements Iterable<E> and provides an iterator() method, the enhanced for loop can also be applied

```
MyArrayList<String> words;
. . .
for (String str : words) {
    System.out.println(str);
}
```

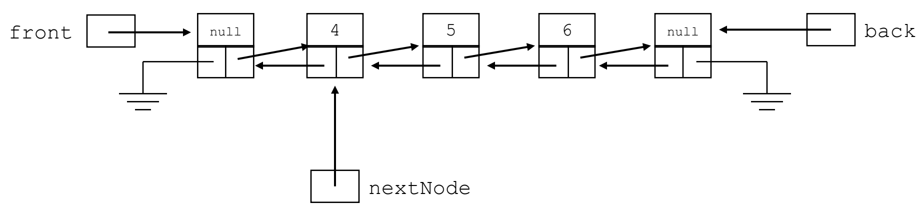
LinkedList iterator

a LinkedList does need an iterator to allow for efficient traversals & list processing

- get() and set() are already O(N) operations, so a typical indexing loop is $O(N^2)$

again, to implement an iterator, need to define a new class that can

- access the underlying doubly-linked list
- keep track of which node in the list is "next"



27

MyLinkedList iterator

again, the class implements the Iterable<E> interface

inner class defines an Iterator class for this particular collection

iterator() method creates and returns an object of that type

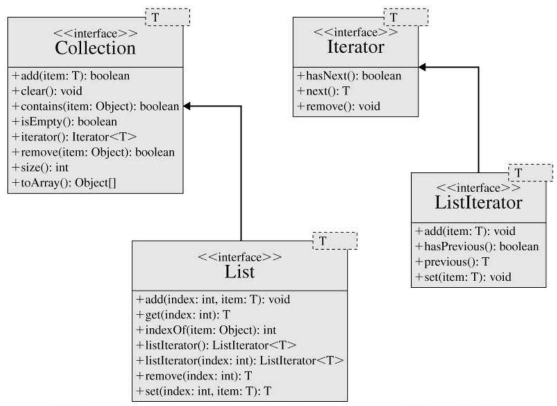
```
public class MyLinkedList<E> implement Iterable<E> {

    . . .

    public Iterator<E> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<E> {
      private DNode<E> nextNode;
      public LinkedListIterator() {
          this.nextNode = MyLinkedList.this.front.getNext();
      }

      public boolean hasNext() {
          return this.nextNode != SimpleLinkedList.this.back;
      }

      public E next() {
          if (!this.hasNext()) {
              throw new java.util.NoSuchElementException();
          }
          this.nextNode = this.nextNode.getNext();
          return this.nextNode.getPrevious().getData();
      }

      public void remove() {
         if (this.nextNode == front.getNext()) {
             throw new RuntimeException("Iterator call to " +
                        "next() required before calling remove()");
         }
         MyLinkedList.this.remove(this.nextNode.getPrevious());
      }
  }
}
```

28

Iterator vs. ListIterator

`java.util.Iterator` defines methods for traversing a collection

an extension, `java.util.ListIterator`, defines additional methods for traversing lists



29