# CS1083 Week 10: Linked Lists

David Bremner

2018-03-12

# Outline

Linked Data Structures

Phone List example

List Classes

Generic Lists

Sortable lists

# "Linked" Examples from real life

- The snow phone list
- The chain of command
- Treasure Hunt

## Main Ideas

- Can only move locally
- Can add or delete in the middle relatively easily.

# Recursive Data Structures

```
class Link implements Comparable {
  private Comparable data;
  private Link next;
     ⋮
}
```

- A *recursive data structure* (*recursive class*) is one that contains instance variables of the same class.

```
this.next=new Link();
```

# Comparison with Arrays

## Linked Lists

- Insert at ends easy
- Insert in middle easy, if you are already there

## Arrays

- Support random access
- Fixed size
- inserting in the beginning / middle is expensive

# Comparison with Arrays

## Linked Lists

- Insert at ends easy
- Insert in middle easy, if you are already there

## Arrays

- Support random access
- Fixed size
- inserting in the beginning / middle is expensive

# Comparison with ArrayLists

## Linked Lists

- Insert at ends easy
- Insert in middle easy, if you are already there

## ArrayList

- Provide List interface for underlying array
- Adding at end is inexpensive
- Adding anywhere is easy for programmer, but potentially expensive
- grows by reallocation and copy

# Comparison with ArrayLists

## Linked Lists

- Insert at ends easy
- Insert in middle easy, if you are already there

## ArrayList

- Provide List interface for underlying array
- Adding at end is inexpensive
- Adding anywhere is easy for programmer, but potentially expensive
- grows by reallocation and copy

Linked Data Structures

Phone List example

List Classes

Generic Lists

Sortable lists

# Phone list example

```
public static void main(String[] args){
    PhoneNode link1=
        new PhoneNode("David Bremner",
                      "555-1212");
    PhoneNode link2=
        new PhoneNode("Bela Lugosi",
                      "666-1522");
    PhoneNode link3=
        new PhoneNode("Jesse James",
                      "IAM-DEAD");
```

# Phone list example

```
public static void main(String[] args){
    PhoneNode link1=
        new PhoneNode("David Bremner",
                      "555-1212");
    PhoneNode link2=
        new PhoneNode("Bela Lugosi",
                      "666-1522");
    PhoneNode link3=
        new PhoneNode("Jesse James",
                      "IAM-DEAD");
    link1.setNext(link2);
    link2.setNext(link3);
```

# Phone list example

```
public static void main(String[] args){
    PhoneNode link1=
        new PhoneNode("David Bremner",
                      "555-1212");
    PhoneNode link2=
        new PhoneNode("Bela Lugosi",
                      "666-1522");
    PhoneNode link3=
        new PhoneNode("Jesse James",
                      "IAM-DEAD");
    link1.setNext(link2);
    link2.setNext(link3);
    System.out.println(
```

# Phone list implementation I

```
public class PhoneNode{
String name;
String number;
PhoneNode next=null;
```

# Phone list implementation I

```
public class PhoneNode{
String name;
String number;
PhoneNode next=null;
public PhoneNode(String theName,
                 String theNumber){
    name=theName;
    number=theNumber;
    next=null;
}
```

# Phone list mutators

```
public void setName(String argName) {
    this.name = argName;
}
```

# Phone list mutators

```
public void setName(String argName) {
    this.name = argName;
}
public void setNumber(String argNumber) {
    this.number = argNumber;
}
```

# Phone list mutators

```
public void setName(String argName) {
    this.name = argName;
}
public void setNumber(String argNumber) {
    this.number = argNumber;
}
public void setNext(PhoneNode argNext) {
    this.next = argNext;
}
```

# Phone list accessors

```
public String getName()  {
    return this.name;
}
```

# Phone list accessors

```
public String getName()  {
    return this.name;
}
public String getNumber()  {
    return this.number;
}
```

# Phone list accessors

```
public String getName()  {
    return this.name;
}
public String getNumber()  {
    return this.number;
}
public PhoneNode getNext()  {
    return this.next;
}
```

Linked Data Structures

Phone List example

List Classes

Generic Lists

Sortable lists

# List Classes

- Usually, but not always, one wants to wrap a linked list data structure (made of nodes) in a List class
- a more natural home for list manipulation operations.
- `head` needs to be updatable;

# PhoneList class

```
public class PhoneList{
    private PhoneNode head;

    public PhoneList(){
        head=null;
    }
    public void insert(String name, String number){
        PhoneNode newNode=new PhoneNode(name,number);

        newNode.setNext(head);

        head=newNode;
    }
                              ⋮
```

# List Traversals

```
public void print(){
    for (PhoneNode current=head; current!=null;
         current=current.getNext()){
        current.print();
    }
}
```

# List Traversals

```
public void print(){
    for (PhoneNode current=head; current!=null;
         current=current.getNext()){
        current.print();
    }
}

public static void main(String[] args){
  PhoneList friends=new PhoneList();
  friends.insert("me", "555-1212");
  friends.insert("myself", "555-1213");
  friends.insert("myself", "555-1214");
  friends.print();
}
```

# Generic Lists

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

# Generic Lists

```
public class Node<C> {
private C data;
private Node<C> next;
public Node(C the_data){
    data=the_data;
    next=null;
}

public C getData(){
    return data;
}

public void setNext(Node<C> theNext){
```

# Generic Lists

```
public static void main(String args[]){
    Node<String> list=null;









}
```

# Generic Lists

```
public static void main(String args[]){
    Node<String> list=null;
    for (int i=0; i<args.length; i++){




    }



}
```

# Generic Lists

```
public static void main(String args[]){
    Node<String> list=null;
    for (int i=0; i<args.length; i++){
        Node<String> node=
            new Node<String>(args[i]);



    }


}
```

# Generic Lists

```
public static void main(String args[]){
    Node<String> list=null;
    for (int i=0; i<args.length; i++){
        Node<String> node=
            new Node<String>(args[i]);
        node.next=list;
        list=node;
    }



}
```

# Generic Lists

```
public static void main(String args[]){
    Node<String> list=null;
    for (int i=0; i<args.length; i++){
        Node<String> node=
            new Node<String>(args[i]);
        node.next=list;
        list=node;
    }
    for (Node<String> n=list ; n!=null ;
         n=n.next){
        System.out.println(n.data);
    }
}
```

# Inserting Nodes at the head

```
public void insertFirst(Object o){
    ListNode node=new ListNode(o);



}
```

# Inserting Nodes at the head

```
public void insertFirst(Object o){
    ListNode node=new ListNode(o);
    node.setNext(head);

}
```

# Inserting Nodes at the head

```
public void insertFirst(Object o){
    ListNode node=new ListNode(o);
    node.setNext(head);
    head=node;
}
```

# Inserting Nodes at end

```
public void insertLast(Object o){
    ListNode node=new ListNode(o);
    if (head==null){
        head=node;
    } else {
        ListNode current=head;




        current.setNext(node);
    }
}
```

# Inserting Nodes at end

```
public void insertLast(Object o){
    ListNode node=new ListNode(o);
    if (head==null){
        head=node;
    } else {
        ListNode current=head;
        while(current.getNext()!=null){
            current=current.getNext();
        }
        current.setNext(node);
    }
}
```

# Insertion Example

```
public static void main(String[] args){
  List test=new List();
  test.insertLast("hello");
  test.insertFirst("goodbye");
  test.insertLast("are you still here");

  test.print();
}
```

# Removing the First Node

```
public Object removeFirst(){
    if (head==null)
        throw new
            IllegalArgumentException(
                "No first element");



}
```

# Removing the First Node

```
public Object removeFirst(){
    if (head==null)
        throw new
            IllegalArgumentException(
                "No first element");
    Object retVal=head.getData();


}
```

# Removing the First Node

```
public Object removeFirst(){
    if (head==null)
        throw new
            IllegalArgumentException(
                "No first element");
    Object retVal=head.getData();
    head=head.getNext();

}
```

# Removing the First Node

```
public Object removeFirst(){
    if (head==null)
        throw new
            IllegalArgumentException(
                "No first element");
    Object retVal=head.getData();
    head=head.getNext();
    return retVal;
}
```

# removeLast

```
public void removeLast(){
    if (head==null)
        throw new
          IllegalArgumentException(
                     "Empty List");
```

# removeLast

```
public void removeLast(){
    if (head==null)
        throw new
          IllegalArgumentException(
                      "Empty List");
    ListNode current=head;
```

# removeLast

```
public void removeLast(){
    if (head==null)
        throw new
          IllegalArgumentException(
                          "Empty List");
    ListNode current=head;
    ListNode previous=null;
    while(current.getNext()!=null){


    }
```

# removeLast

```
public void removeLast(){
    if (head==null)
        throw new
          IllegalArgumentException(
                          "Empty List");
    ListNode current=head;
    ListNode previous=null;
    while(current.getNext()!=null){
        previous=current;
        current=current.getNext();
    }
```

# removeLast

```
public void removeLast(){
    if (head==null)
        throw new
          IllegalArgumentException(
                   "Empty List");
    ListNode current=head;
    ListNode previous=null;
    while(current.getNext()!=null){
        previous=current;
        current=current.getNext();
    }
    if (previous != null)
        previous.setNext(null);
```

# remove example

```
public static void main(String[] args){
    List test=new List();
    test.insertLast("hello");
    test.insertFirst("goodbye");
    test.insertLast("are you still here");

    test.print();
    System.out.println("");
    test.removeFirst();
    test.print();


    System.out.println("");
    test.removeLast();
    test.print();
```

# Sortable lists: ComparableNode

```
class ComparableNode implements

Comparable<ComparableNode> {
    private Comparable data;
    private ComparableNode next;




    ⋮
```

# Sortable lists: ComparableNode

```
class ComparableNode implements

Comparable<ComparableNode> {
    private Comparable data;
    private ComparableNode next;
    public ComparableNode(Comparable comp){
        data=comp;
        next=null;
    }
    ⋮
```

# Sortable lists: ComparableNode

```
class ComparableNode implements

Comparable<ComparableNode> {
    private Comparable data;
    private ComparableNode next;
    public ComparableNode(Comparable comp){
        data=comp;
        next=null;
    }
    ⋮
    public int compareTo(ComparableNode other){
        return data.compareTo(other.data);
    }
```

# Keeping track of more: length, tail pointer

```
public class MergeList{
private ComparableNode first,last;
private int length;
```

# Keeping track of more: length, tail pointer

```
public class MergeList{
private ComparableNode first,last;
private int length;
public MergeList(){
    this(null,null,0);
}
```

# Keeping track of more: length, tail pointer

```
public class MergeList{
private ComparableNode first,last;
private int length;
public MergeList(){
    this(null,null,0);
}
public MergeList(ComparableNode theFirst,
                 ComparableNode theLast,
                 int theLength){



}
```

# Keeping track of more: length, tail pointer

```
public class MergeList{
private ComparableNode first,last;
private int length;
public MergeList(){
    this(null,null,0);
}
public MergeList(ComparableNode theFirst,
                 ComparableNode theLast,
                 int theLength){
    first=theFirst;
    last=theLast;
    length=theLength;
}
```

# Inserting at front is still easy

```
public void insertFirst(Comparable key) {
    ComparableNode newNode=
        new ComparableNode(key);






}
```

# Inserting at front is still easy

```
public void insertFirst(Comparable key) {
    ComparableNode newNode=
        new ComparableNode(key);
    newNode.setNext(first);




}
```

# Inserting at front is still easy

```
public void insertFirst(Comparable key) {
    ComparableNode newNode=
        new ComparableNode(key);
    newNode.setNext(first);
    first=newNode;



}
```

# Inserting at front is still easy

```java
public void insertFirst(Comparable key) {
    ComparableNode newNode=
        new ComparableNode(key);
    newNode.setNext(first);
    first=newNode;
    if(last == null) {
        last=newNode;
    }
}
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){



}

public void insertLast(ComparableNode newNode){
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);

}

public void insertLast(ComparableNode newNode){
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);
    insertLast(newNode);
}

public void insertLast(ComparableNode newNode){
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);
    insertLast(newNode);
}

public void insertLast(ComparableNode newNode){
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);
    insertLast(newNode);
}

public void insertLast(ComparableNode newNode){
    if (last==null){


    } else {
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);
    insertLast(newNode);
}

public void insertLast(ComparableNode newNode){
    if (last==null){
        first=newNode;
        last=newNode;
    }  else {
```

# Inserting at the end is now easier

```
public void insertLast(Comparable key){
    ComparableNode newNode=
        new ComparableNode(key);
    insertLast(newNode);
}

public void insertLast(ComparableNode newNode){
    if (last==null){
        first=newNode;
        last=newNode;
    }  else {
        last.setNext(newNode);
        last=newNode;
```

# Operations on entire lists

## join

```
public void join(MergeList other){
  length=length+other.length;
  this.last.setNext(other.first);
  this.last=other.last;
}
```

# splitting a list in two

```
public MergeList split(int where){
    ComparableNode cursor=first;
```

# splitting a list in two

```
public MergeList split(int where){
    ComparableNode cursor=first;
    for (int i=1; i< where; i++){
        cursor=cursor.getNext();
    }
```

# splitting a list in two

```
public MergeList split(int where){
    ComparableNode cursor=first;
    for (int i=1; i< where; i++){
        cursor=cursor.getNext();
    }
    MergeList secondHalf=
        new MergeList(cursor.getNext(),
                      last,
                      length-where);
```

# splitting a list in two

```
public MergeList split(int where){
    ComparableNode cursor=first;
    for (int i=1; i< where; i++){
        cursor=cursor.getNext();
    }
    MergeList secondHalf=
        new MergeList(cursor.getNext(),
                      last,
                      length-where);
    last=cursor;
    length=where;
    cursor.setNext(null);
```

# splitting a list in two

```
public MergeList split(int where){
    ComparableNode cursor=first;
    for (int i=1; i< where; i++){
        cursor=cursor.getNext();
    }
    MergeList secondHalf=
        new MergeList(cursor.getNext(),
                      last,
                      length-where);
    last=cursor;
    length=where;
    cursor.setNext(null);
    return secondHalf;
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){




    }
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){
        ComparableNode transfer;




        result.insertLast(transfer);
    }
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){
        ComparableNode transfer;
        if (this.getFirst().
            compareTo(other.getFirst()) <= 0)
            transfer=this.removeFirst();
        else
            transfer=other.removeFirst();
        result.insertLast(transfer);
    }
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){
        ComparableNode transfer;
        if (this.getFirst().
            compareTo(other.getFirst()) <= 0)
            transfer=this.removeFirst();
        else
            transfer=other.removeFirst();
        result.insertLast(transfer);
    }
    if (this.getLength()>0) result.join(this);
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){
        ComparableNode transfer;
        if (this.getFirst().
            compareTo(other.getFirst()) <= 0)
            transfer=this.removeFirst();
        else
            transfer=other.removeFirst();
        result.insertLast(transfer);
    }
    if (this.getLength()>0) result.join(this);
    if (other.getLength()>0)
```

# Merging Sorted Linked Lists

```
void merge(MergeList other){
    MergeList result=new MergeList();
    while(!(this.isEmpty()||other.isEmpty())){
        ComparableNode transfer;
        if (this.getFirst().
            compareTo(other.getFirst()) <= 0)
            transfer=this.removeFirst();
        else
            transfer=other.removeFirst();
        result.insertLast(transfer);
    }
    if (this.getLength()>0) result.join(this);
    if (other.getLength()>0)
```

# Merge sort for linked lists

```
public void sort(){
```

# Merge sort for linked lists

```
public void sort(){

    if (getLength()<=1)
        return;
```

# Merge sort for linked lists

```
public void sort(){

    if (getLength()<=1)
        return;

    MergeList secondHalf;
```

# Merge sort for linked lists

```
public void sort(){

    if (getLength()<=1)
        return;

    MergeList secondHalf;

    secondHalf=this.split(getLength()/2);
```

# Merge sort for linked lists

```
public void sort(){

    if (getLength()<=1)
        return;

    MergeList secondHalf;

    secondHalf=this.split(getLength()/2);

    this.sort();
    secondHalf.sort();

```

# Merge sort for linked lists

```
public void sort(){

    if (getLength()<=1)
        return;

    MergeList secondHalf;

    secondHalf=this.split(getLength()/2);

    this.sort();
    secondHalf.sort();

    this.merge(secondHalf);
```