CS 310: Doubly Linked Lists and Iterators

Chris Kauffman

Week 5-1

Career Fair

Logistics

- October Wed 7 (Sci/Eng) and Thu 8 (Humanities)
- 11 a.m. to 4 p.m. Dewberry Hall, JC
- Info for students here

Workshops

- Interview skills, resume prep, etc
- Info and list is here

Logistics

HW 2: Upcoming

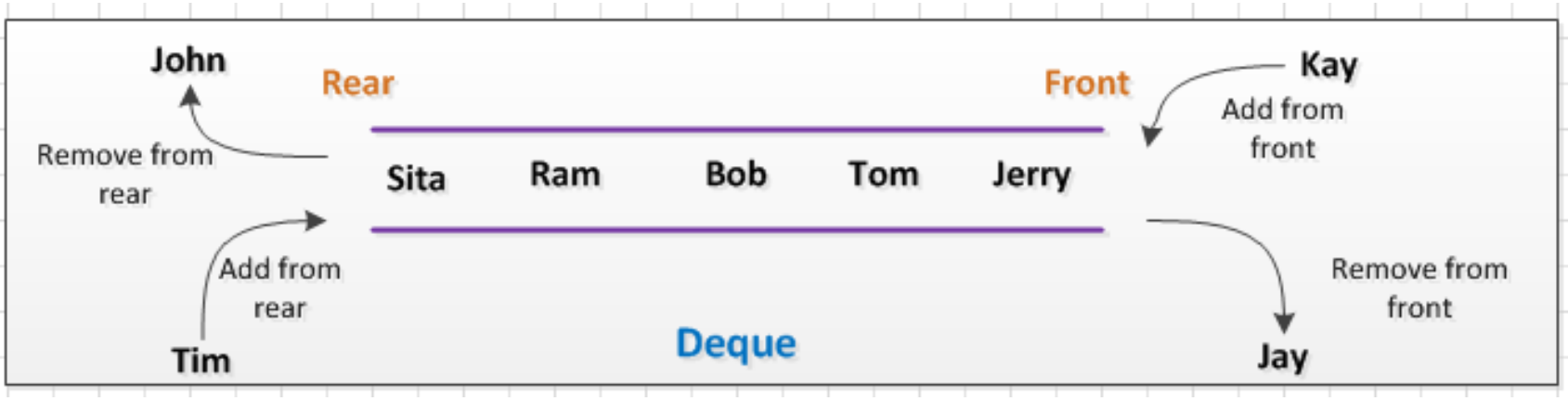
- Interesting design issues, want to spare you too much trouble
- Posted Mon/Tue, 2 week turn around

Reading

- Weiss Chapter 17: Linked Lists
- Weiss Chapter 20: Hash Tables

The Deque: Double Ended Queue

- Add and remove at both ends
- `interface Deque` in `java.util.Deque`
- Several implementations in Java like `ArrayDequeue` and `LinkedList`



Source

Work It

Print elements front to back

```
class ArrayList/LinkedList{
  public void printAll(){...}
}
```

- `ArrayList` implementation
- `SinglyLinkedList` implementation
- Make both $O(N)$

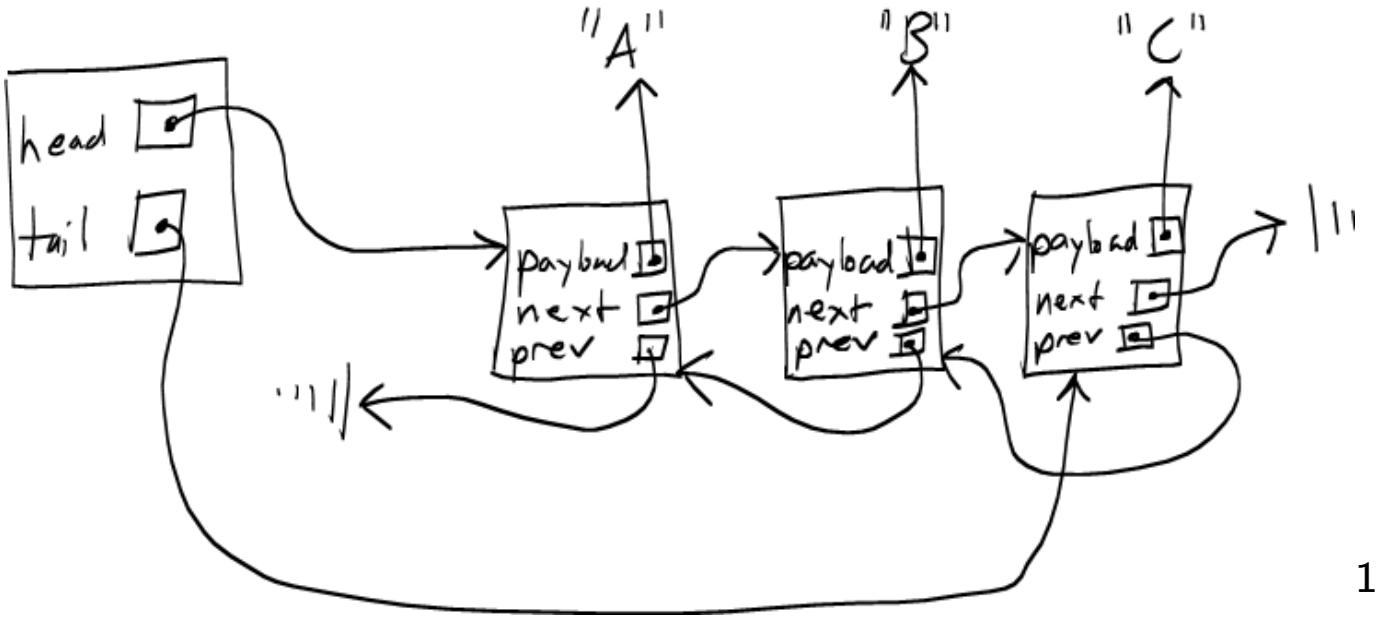
Print elements back to front

```
class ArrayList/LinkedList{
  public void printAllReverse(){...}
}
```

- `ArrayList` implementation
- `SinglyLinkedList` implementation (!)
- Can both be $O(N)$?

Double Your Fun

- Singly linked nodes: only `next`
  - `Node n = new Node(data,next);`
- Doubly linked also has `previous`
  - `Node n = new Node(data, previous, next);`



1

How about `printAllReverse()` now

[1] Source: David H. Hovemeyer's notes

To Header or Not to Header

- May be able to simplify using extra space
- Auxiliary 'header' and 'tailer' nodes
- Draw pictures to understand these
- Weiss uses header/tailer nodes
- Consider code below for `add(x)` to the back of a linked list

No Header

```
public void add(T x){
  if(empty()){
    head =
      new Node(x,null,null);
    tail = head;
  }
  else{
    tail.next =
      new Node(x,tail,null);
    tail = tail.next;
  }
}
```

With Header/Tailer

```
public void add(T x){
  Node n =
    new Node(x,tail.prev,
                tail);
  tail.prev.next = n;
  tail.prev = n;
}
```

- Always have head/tail nodes
- No special cases for empty
- Requires changes to `get(i)`

Relevance Note

Part of HW 2 will involve implementing a basic doubly linked list

- Constraint: no use of `java.util.LinkedList`
- Some functionality will require more control than standard class
- Will provide version of Weiss's doubly-linked list as a starting point; you must modify and complete it
- His implementation is two-headed: special node for front and rear, always there

A Problem

Recall

- `ArrayList.get(i)` : $O(1)$
- `LinkedList.get(i)` : $O(n)$

Trouble

```
List<Integers> l = ...;
int sum = 0;
for(int i=0; i<l.size(); i++){
  sum +=  l.get(i);
}
```

What is the complexity of the loop?

Peeking Inside with Iterators

Arrays are simple

- `get/set` anything
- `add/remove` is obvious
- Very clear how data is laid out

Just about every other data structure is less so

- Getting/setting nontrivial
- Must preserve some internal structure - control access
- Element-by-element needs to be done carefully

These qualities give rise to iterators

- A view of a data structure
- Allows sequential access and modification

Iterators

Give access to a position in a list (or other data structure)

```
public interface ListIterator<T>{
  // Can the iterator be moved?
  public boolean hasNext( );
  public boolean hasPrevious( );

  // Move the iterator
  public T next( );
  public T previous( );

  // Modify the container
  public void add(x);
  public void remove( );
}
```

Warning: In Between

List Iterators have slightly complex semantics: *between* list elements

Removing

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
                [ A B C D]
                 ^
itr.next()      [ A B C D ]
A                  ^
itr.remove()    [ B C D ]
                 ^
itr.next()      [ B C D ]
B                  ^
itr.next()      [ B C D ]
C                     ^
itr.remove()    [ B D ]
                   ^
itr.remove()    [ B D ] //Error
                   ^
```

Next/Previous

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
                [ A B C D ]
                 ^
itr.next()      [ A B C D ]
A                  ^
itr.next()      [ A B C D ]
B                     ^
itr.previous()  [ A B C D ]
B                  ^
itr.previous()  [ A B C D ]
A                 ^
itr.next()      [ A B C D ]
A                  ^
itr.remove()    [ B C D ]
                 ^
```

Iterator Semantics

- Use `next()/previous()` to move
- `next()/previous()` returns element "moved over"
- `remove()` removes element that was returned from last `next()/previous()`
- Illegal to w/o first calling `next/previous`
- `add(x)` puts `x` before whatever `next()` would return

Once you wrap your head around it, not too bad

- Weiss's implementation in `LinkedList` is slightly complex
- JGrasp has a tough time drawing iterators

Exercise: Draw the Final List

```
LL l = new LL([A, B, C, D])
iter = l.iterator()
iter.next()
iter.next()
iter.add("X")
iter.previous()
iter.add("Y")
iter.next()
iter.next()
iter.remove()
iter.next()
iter.add("W")
iter.previous()
iter.remove()
```

What would you do?

```
// l = [A, B, C, D];
it1 = l.iterator().next().next();
it2 = l.iterator().next();
// l = [ A B C D ]
//           1
//         2
it1.remove();
it2.next(); // ??
```

Where should `it2` be now?