Last week – linked lists

- a list of data objects where each element points to the next element in the list
- A convenient method for handling large dynamic databases
- Linked lists consist of “nodes” which:
  - carry a data object (“item”)
  - point to the next node (“next pointer”)
- Linked list needs to keep track of the “head” of the list, i.e., the first element.

Two additional kinds of data structure that are related to linked lists

(given a linked list, you can generate these data structures)

Stacks and Queues

Stack: last element in is the first element out
- washing dishes
- interrupted projects with something of higher priority

Queue: first element in is the first element out
- like a lineup at the bank – in some sense this is “fair”

These data structures are fundamental in computer science
Mostly useful for “working” data

Stacks

- As data arrives, data objects are placed at the “top” of the stack
- New arrivals are “pushed” on the stack
- As data is used/serviced, data objects are also removed from the top of the stack

- Departing objects are “popped” from the stack
- Push to enter, pop to leave

A stack can be straightforwardly implemented using a linked list

- “top” of the stack -- “head” pointer
- Push – add a new element to the head

head
Node 1
data
next->

Push:

head	(old head)
New node	Node 1
data	data
next ->	next ->

- Pop – Delete an element from the head

head	
Dep node	Node 1
data	data
next->	next ->

- just assign head pointer to Dep node's next pointer.
- A stack implementation is very straightforward to accomplish using a linked list

Implementation strategies:

Strategy 1: use inheritance to extend a linked list into a stack

```
public class Stack extends LinkedList
{
        // don't need any new instance fields
        // new methods:
        public MyObject pop()
                // remove an object from the list and return the
                // data item, of type MyObject
        public void push(MyObject newObject)
                // create a new LinkedListNode containing newObject
                // and push it onto the stac
}
```

- this implementation has the advantage that the linked list functionality is still available
- e.g., you can traverse the stack and inspect all the data elements if you wish
- Having access to the entire linked list might be a disadvantage in some circumstances (e.g., it gives you less control).

Strategy 2: linked list is an instance field of the stack

```
public class Stack
{
        private LinkedList l;
        public MyObject pop() // pops data from stack
        public void push(MyObject newObject) // pushes data on stack
}
```

- This implementation behaves like a “pure”stack – you are only allowed to push() and pop(), all the linked list functionality is hidden (because the LinkedList object is private within Stack).

Queues

A queue is similar to a stack except:

- The first element to arrive is the first element to leave
- Elements leave the queue in the same order that they arrived

Implementing a queue as a linked list:

- adding an element: identical to the stack case; element is added to the head.
- Deleting an element: remove the last element in the linked list and return its data.
- Elements "enter" and "leave" the queue

Node n	Node n+1
data	data
next ->	next

seek out node n, use its next pointer to retrieve the data from node n+1, and then set node n's next pointer to null (i.e., it is the last element in the linked list)

Same two strategies can be used to implement a queue from a linked list:

1. use inheritance: queue extends the linked list

```
public class Queue extends LinkedList
{
    public MyObject leave() // causes an element to leave the queue
    public void enter(MyObject newObject)
        // causes an element to enter the queue
}
```

2. Queue contains LinkedList as a private instance field

```
public class Queue
{
    LinkedList l;
    public MyObject leave()
    public void enter(MyObject newObject)
}
```

- with the queue, when objects exit the queue, they exit from the end of the linked list
- This is inconvenient, since we don't keep track of the location of the end of the list; and even if we did, it would not be convenient to delete an element from the end (because the tail of the list doesn't keep track of the element immediately before it).

Doubly Linked List

- unlike the normal linked list, here a node is connected both to the node after (through “next”) but also the node before (through “previous”).
- It is efficient to perform operations (e.g., adding and deleting nodes) on both the head of the list and the “tail” (i.e., the end) of the list

- each node would contain “next” and “previous” pointers

```
public class DoublyLinkedListNode
{
    DoublyLinkedListNode next, previous;
    MyObject data;
    // other methods, etc.
}
```

- the “next” pointer points “down” the list, as in a normal linked list

- the “previous” pointer points “up” the list.
- The doubly linked list class will keep track of the “head” pointer (like in a linked list) but also the “tail” pointer (i.e., the end of the list)

e.g.

Head			Tail
Node 1	Node 2	Node 3	Node 4
data	data	data	data
prev=null	<- prev	<- prev	<- prev
next ->	next ->	next ->	next = null

“doubly linked” because each node contains two pointers

Why do this?
- e.g., implement a queue much more efficiently (can jump directly to the end of the list and add/delete nodes)
- good for any application where both the top and the end of the list are important

Adding a node

Adding to the middle of a doubly linked list:

Node a	Node b	Add Node c between a & b
data	data	
<- prev	<- prev	
next ->	next ->	

- Node a's Next points to node c
- Node c's Next is node a's old next (node b)
- Node b's Prev points to node c

- Node c's Prev points to node b's old previous (node a)

Node a	Node c	Node b
data	data	data
<- prev	<- prev	<- prev
next ->	next ->	next ->

Exception: head

Head Add node c before node b

Node b

data

prev = null

next ->

- Follow the procedure above without node a
- Adjust the head pointer in the end to the new head, node c

	Head	
	Node c	Node b
	data	data
null	<- prev	<- prev
	next ->	next ->

Adding to the tail – like adding to the head, except without the “after” element instead of the “before” element. Finally, adjust the tail pointer.

Deleting a node from a doubly linked list

Start with

Node a	Node b	Node c	Delete node b
data	data	data	
<- prev	<- prev	<- prev	
next ->	next ->	next ->	

- assign node a's next pointer to node b's next pointer
- assign node c's previous pointer to node b's previous pointer

Node a		Node c
data		data
<- prev		<- prev
next ->		next ->
	Node b	-- no longer in the list
	data	
	<- prev	
	next ->	

Deleting from the head: Do as above, but without node a; adjust the head pointer to point to node c.

Deleting from the tail: Do as above, but without node c; adjust the tail pointer to point to node a.

Linked Lists and Inheritance

- Using inheritance, it is possible to implement a linked list where each node in the linked list contains different data types.

2 ways to do this:

1. Linked list where every node contains a fundamental data type, but some nodes contain extra data

```
public class ListNode
{
	private MyObject item;
	private ListNode next;
	// other methods ...
}

public class ExtendedListNode extends ListNode
{
	// add another data object ...
	private OtherObject otherItem;
}
```

- in this example, every list node contains a MyObject data item, and a “next” pointer to the next list node
- Some list nodes contain extra data in the form of a MyOtherObject data item.

- Extended list nodes still contain “next” pointers and the public methods of normal ListNodes by virtue of inheritance.

- say you don't want a common data type
- but, you want to ensure that every list node contains some kind of data
- make ListNode an abstract class

```
public abstract class ListNode
{
	private ListNode next; // the only thing list nodes need in
	                                    //common
	// want to enforce the idea that list nodes contain data
	// enforce that through the abstract methods
	// use the data type “Object” which applies to all objects
	public abstract Object getItem();
	public abstract void setItem(Object newObject);
}
```

- when you implement getItem() and setItem(), you must properly cast the Object type to whatever type you are using
- Generally, when you do so, the compiler will give a warning
- This is generally done when the linked list node is not an inner class

Remarks about test 2:

- 2 hours, in class, starts at 2:30 and ends at 4:30
- Similar in length and structure to Test 1
- Material will focus on material covered after test 1
- Test will be open any printed resource.

GUI:
- components, buttons, labels, text fields, etc.
- menus
- panels and layouts
- events, listeners, handle events and how they influence window behaviour

Searching and sorting:
- Searching a list (sorted/unsorted)
- Sorting a list (bubble/merge/quick)
- Sorting and interfaces (Comparable interface)

Linked Lists
- Singly linked lists
- Templates and inner classes
- Stacks and queues
- Doubly linked lists

Office hours: Tuesday and Thursday from 11:30-12:30, other times by appointment

CSEB 2022