CS 1332 - Week 02

Monday

Topics

I. Iterator and Iterable
   A. Import statements:
      1. import java.lang.Iterable → implicit, not very involved, only requires one method to be implemented
      2. import java.util.Iterator → explicit, very involved, requires the creation of a class with several methods that must be implemented
   B. Code demo

```
import java.lang.Iterable;
import java.util.Iterator;
public class LinkedList<Type> implements Iterable<Type> {
      private Node<Type> head;
      /** constructor omitted **/

      public Iterator<Type> iterator() { // for Iterable
            return new LLit(this);
      }

      private class LLit implements Iterator<Type> {
            private Node<Type> current;
            private LLit(LinkedList list) {
                  current = head; // gives us starting point
            }
            public boolean hasNext() {
                  return current != null;
            }
            public Type next() {
                  Node<Type> tmp = null;
                  if(hasNext()) {
                        tmp = current;
                        current = current.next;
                        return tmp.data;
                  } else {
                        return null;
                  }
            }
      }
      /** private inner Node class omitted **/
}
```

C. Iterable - requires the iterator() method to be overridden which returns the iterator

D. Iterator - an object which iterates through the structure
   1. Our iterator will be similar to the loops we've used to manually traverse through the list
   2. Requires two methods to be overridden: next() and hasNext()
      a) hasNext() → returns if current is NOT null (if the iterator *has* a *next* data element to return)
      b) next() → returns the *next* data the iterator has yet to give you (the data from the current node) and moves current to the next node

E. Code demo: using the iterator

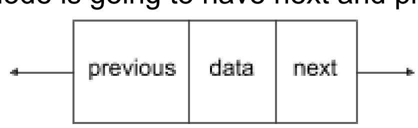
```
/** in some main method **/
LinkedList<String> courses = new LinkedList();
courses.add("1332"); // add some more courses...

// implicitly use iterator
for (String course: courses) {
      System.out.println(course);
}
// explicitly use iterator
Iterator<String> courseit = courses.iterator();
while(courseit.hasNext(){
      System.out.println(courseit.next());
}
```

   1. Iterators are implicitly called when using for-each loops - there's no stopping it and you can't really use it to alter list values
   2. You can explicitly use the iterator by instantiating an Iterator to step through the linked list and have more control over each step

II. Doubly Linked Lists

A. Generally always have both a head and tail pointer

B. A doubly linked node is going to have next and previous pointers and data.



C. For a DLL of size 0, both the head and tail point to null

D. For a DLL of size 1, both the head and tail point to the single node

E. For a DLL of size > 1:
   1. Adding to the front:
      a) Create the new node
      b) Connect the node to the LL → set the new node's next to the head
      c) Connect the LL to the node → set the head's prev to the new node
      d) Point the head to the new node
   2. Adding to the back:
      a) Create the new node

            b) Connect the node to the LL → set the new node's prev to the tail
            c) Connect the LL to the node → set the tail's next to the new node
            d) Point the tail to the new node
        3. Make sure you always change the head/tail after everything else has been connected, we do not want to lose these references!
        4. Removing from the back:
            a) Set the tail to tail's previous
            b) Set the tail's next to null
        5. Removing from the front:
            a) Set the head to head's next
            b) Set the head's previous to null
        6. Edge case: removing from a list of length 1 → set head and tail to null

III. Circular Singly Linked Lists
    A. The last node in the list points back to the head.
    B. We can no longer use current == null to check if we've reached the end of our list
        1. We must use current == head (reference equality) to terminate our loop
    C. How to NOT add to the front:
        1. Create a new node, point it to the head, and move the head
        2. Resetting the last node to point to the new head would be O(n) because we would have to iterate through the list to find it
    D. Adding to the front in O(1) (the correct way):
        1. Create a new, empty node
        2. Connect the node to the CLL → set the new node's next to head's next
        3. Connect the CLL to the node → set head's next to the new node
        4. Put the data from the head into the new node
        5. Put the data we want to add into the head node
    E. Adding to the back in O(1):
        1. Perform the steps to add to the front
        2. Now, just move the head to head's next and the data you just added is now at the back of the list
    F. Removing from the front in O(1) (when size > 1):
        1. Save the data from the head somewhere to return later
        2. Copy the data from head's next into the head
        3. Set head's next pointer to head's next's next (essentially cutting the node at index 1 out of the list)
    G. Unfortunately, removing from the back cannot be optimized with a data manipulation trick → it will be O(n) to iterate to the node before the last one

Wednesday

<u>Topics</u>

I. Recursion Basics
    A. Definition: a method repeatedly calls itself

B. Must haves:
    1. Base case/terminating condition (can have multiple)
    2. Recursive call to function (can have multiple)
    3. A parameter that advances toward termination (can have several)
C. Termination is very important to prevent infinite recursion
D. Basic structure:

```
rFunction (parameter)
     if (parameter meets terminating condition)
          return value
     else
          return rFunction(changed parameter)
          // self-call can be before return but not after
```

II. Math-based recursion - classic examples: factorial, fibonacci
A. Example: compound interest = $A = P*(1 + (r/n))^{nt}$
B. Function: recursive IRA

```
rIRA(p, r, t) // principle, rate, time
     if (t <= 1)
          return p;
     else
          return Math.pow((1+r/4),4) * rIRA(p,r,t-1) + p;
```

C. Investing $2000 a year with an 8% growth rate...
    1. From 40 to 70: ~$240,000
    2. From 30 to 70: ~$550,000
    3. From 20 to 70: $1,250,000 ($100,000 of your own money was invested)

III. LinkedList recursion
A. Example: a LL contains all data in sorted order so all duplicates are contiguous, now remove all the duplicates
B. Function: remove all duplicates

```
rRemove(c)// current node
     if (c == null)
          return null;
     else
          c.next = rRemove(c.next);
          if (c.next != null && c.data == c.next.data)
               return c.next; // cut out a duplicate
          else
               return c; // makes no changes to the list
```

C. How it works: starting from the end of the list, this function “collapses” duplicate chains; the last duplicate in the chain is technically the one that stays in the list
D. Trace through code with the following example:
    1. head → [2] → [3] → [3] → null

Friday

Topics

I. Stacks Intro: Array, SLL, and DLL operation review

Time Complexity	Access	Search	Add (to front)	Remove (from front)
Array	O(1)	O(n)	O(n)	O(n)
SLL	O(n)	O(n)	O(1)	O(1)
DLL	O(n)	O(n)	O(1)	O(1)

   A. Access - accessing an element at a given index
      1. Arrays - we can access data at a given index in constant time
      2. LL - even if we have index information (eg. storing index as an attribute of the node), we cannot access that index without iterating to it
   B. Search - accessing an element at an unknown index
      1. Always O(n) for unsorted structures
   C. Add (to front)
      1. Arrays - we must shift everything to create an empty spot
      2. LL - easily add to front with head pointer
   D. Remove (from front)
      1. Arrays - we must shift everything to fill empty spot
      2. LL - easily move the head pointer over one
   E. Remove (from back, with tail)
      1. SLL - O(n); DLL - O(1)

II. Stacks
   A. Abstract Data Type (ADT) - a conceptual outline for how a data structure should be implemented (eg. expected behaviors)
      1. Data structures are the concrete implementations of ADTs
      2. Multiple implementations exist with different backing structures
   B. A stack can be backed by an Array, a SLL, or a DLL.
   C. What does a stack do?
      1. Examples: a Pringles can, the recursive stack, a laundry pile
      2. To add, we'll "push" items onto the stack
      3. To remove, we'll "pop" from the stack, but we can only access what was pushed last: "Last In, First Out," a.k.a. LIFO
      4. We cannot access anything other than what is at the top of the stack
      5. Stacks are very linear and are implemented with linear structures: Arrays and LinkedLists (Singly or Doubly)
   D. Stack operations - implementation depends on backing structure
      1. void push(x) - add
      2. x pop() - remove
      3. x top()/peek() - returns the next item to pop without actually removing it
      4. bool isEmpty()

5. void clear()

III. SLL-backed stack
   A. When the stack is empty, the SLL's head is null
   B. When you push, you add to the front → push 1,3,3,2
      1. head→ [2] → [3] → [3] → [1] → null
   C. When you pop, you remove from the front → pop 2,3,3,1
   D. All stack operations deal with the head (to clear the stack → set head to null)

IV. Array-backed stack
   A. When the stack is empty, the array size is 0
   B. When you push, add first element to index 0, where do we put the next element?
      1. We could add to the front, but this is O(n)
      2. Instead, we will add and remove from the back (at index = size)
         a) It is important to keep track of size and capacity (resize)
   C. The "top" of the stack is at index size - 1
   D. When you pop, you'll remove from index size - 1
      1. Option 1: decrement size and then remove from index size.
      2. Option 2: remove from index size - 1 and then decrement size
   E. Clearing the stack
      1. Option 1: reset size to 0 and just overwrite old data (O(1))
      2. Option 2: reset size to 0 and delete everything in the array (O(n))
      3. Option 3: reset size to 0 and reassign backing array to new array (O(1))

Stacks	SLL	Array
push	O(1) - head	O(1) (amortized) - size
pop	O(1) - head	O(1) - size - 1
top	O(1) - head	O(1) - size - 1
isEmpty	O(1) - head	O(1) - size
size	O(1)	O(1)
clear	O(1) - head	Depends on implementation
resize	O(1) - just add a new node	O(n)

V. DLL- backed
   A. Just like SLL-backed but you can add/remove from the head or tail (make sure whichever end you add to you also remove from)