CSCI 102 assignment 2 – Iterators and Priority Queues

Sept 22, 2024

1 Iterators

Say we want to iterate through a linked list `list` and print every element. We consider the code

```
for (int i = 0; i < list.size(); i++){
    System.out.println(list.getAtIndex(i));
}
```

However this code has complexity $O(n^2)$. Instead, we can make `list` a `PositionList<E>` and instead use the $O(n)$ code

```
Position<E> current_pos = list.first();
for (int i = 0; i < list.size(); i++){
    System.out.println(current_pos.getElement());
    current_pos = list.after(current_pos);
}
```

`PositionList<E>` provides functionality to move forward and backward through the list and retrieve the element at each position many times, however, for iteration, we only want to go forward and retrieve each element once.

Java comes with the default interface for a list that can be iterated through only in the forward direction, `Iterable<E>`. A class that implements `Iterable` must have a method `iterator` that returns an object that can iterate through the elements of the list. This object that can iterate through the elements of the list is another default class in `java.utils`, `Iterator<E>`. Here are the definitions of the two interfaces:

```
interface Iterable<E>{
    Iterator<E> iterator();
}

interface Iterator<E>{
    boolean hasNext();
    E Next();
}
```

Conceptually, `Iterator` starts at the beginning of the list and provides the next element of the list with each call to `Next()`; `hasNext()` simply returns whether or not `Iterator` has reached the end of the list. Note: `Next()` returns elements, not positions! Take a minute to compare `Iterator` and `PositionList`.

Now if we have any list that implements `Iterable<E>` we can iterate through its elements as

```
Iterator<E> iter = list.iterator();
while (iter.hasNext()){
    E element = iter.next();
    System.out.println(element);
}
```

Java also allows one to write the above code more compactly using the syntax

```
for (E element : list){
    System.out.println(element);
}
```

which can be read as code that executes a command "for each element in the list".

- Modify the class code for `ArrayList` and `DoublyLinkedList` so that both classes implement `Iterable<E>`. You may use the methods we wrote for `DoublyLinkedList` to implement `PositionList`. Recall also that in class we outlined how to implement `PositionList` using `ArrayList`; use a similar idea.
- Include a main method in each class that adds the numbers 1 through 10 to the list and then prints the contents of the list using the code

  ```
  Iterator<E> iter = list.iterator();
  while (iter.hasNext()){
      E element = iter.next();
      System.out.println(element);
  }
  ```

  Your code should be written so that this runs in $O(n)$ time!

2 Priority Queues

In the stack ADT, the element that was added last has the highest *priority* for removal. In the queue ADT, the element with the highest priority for removal was the one which was added first. Here we will define an ADT where one specifies the priority with a `double` when inserting it to a list; this ADT will be called a `PriorityQueue`:

```
public interface PriorityQueue<E> {
    int size();
    boolean isEmpty();
    void insert(double priority, E element);
    E removeMaxPriority();
}
```

`insert` adds a `element` to the PriorityQueue with priority `priority`; `removeMaxPriority` removes and returns the element with the highest priority for removal (`null` if the object is empty). Pretend there are no ties in priority.

- Implement a `PriorityQueue<E>`. Don't worry too much about efficiency, *just give some implementation*!
- Include a main method that adds the numbers 1 through 10 to a `PriorityQueue<E>` with priorities $3, 4, 2, 7, 6, 8, 5, 9, 1, 0$ then iteratively removes each element in order of priority and prints them.
- Implement a subclass of your your implementation of `PriorityQueue`, `RandomQueue`, that implements `void insertAtRandom(E element)` by adding element into the priority queue with a random priority. Use `Math.random()` to generate random numbers.
- Bonus: Implement a `Stack<E>` and `Queue<E>` as subclasses of your implementation of `PriorityQueue<E>`. Don't worry about efficiency.
- Bonus (not for extra marks): Come up with an example of data that would naturally be stored in a `PriorityQueue<E>`.

Please submit your code and answers to the questions in a zipped folder on Brightspace by Oct 3.