Linked List III

Lecture 12

Prof. Eric Polizzi
Summary previous Lecture

- **Simple Linked-list**

  ![Simple Linked-list diagram]

- **Simple Linked-list: double-ended feature**

  ![Simple Linked-list: double-ended feature diagram]

- **Methods:**
  - insertFirst, O(1)
  - deleteFirst, O(1)
  - displayList, O(N)
  - find, O(N)
  - delete, O(N)
  - insertLast, O(1)
class FirstLastpp
{
  public static void main(String[] args){
    LinkList2Ends mylist = new LinkList2Ends();
    mylist.insertFirst("Obiwan",55);
    mylist.insertFirst("Luke",20);
    mylist.insertFirst("Han",40);
    mylist.insertLast("Anakin",40);
    mylist.insertLast("Leia",20);
    mylist.insertLast("Yoda",400);
    mylist.displayList();
  }
}

Remark: repeated insertions at the front reverse the list of items, while repeated insertions at the end/rear preserve the order

Han's age is 40  Luke's age is 20  Obiwan's age is 55  Anakin's age is 40  Leia's age is 20  Yoda's age is 400
Linked-List: Implementation of Stacks

- Linked-List implementation:
  - Does not require the knowledge of the maxSize
  - Only allocate the right amount of memory for stack items
  - Push(item) → insertFirst(item)
  - Pop → deleteFirst
class LinkStack {

    private LinkList thelist;

    // constructor
    public LinkStack() { thelist = new LinkList(); }

    // methods
    public boolean isEmpty() {
        return (thelist.isEmpty);
    }

    public void push(Object item) {
        thelist.insertFirst(item);
    }

    public Object pop() {
        return thelist.deleteFirst();
    }

    public void displayStack() {
        System.out.print("Stack top===>bottom: ");
        thelist.displayList();
    }
}
Linked-List: Implementation of Stacks

```java
class StackApp1 {
    public static void main(String[] args) {
        // ArrayStack mystack = new ArrayStack(5); // array-based
        LinkStack mystack = new LinkStack(); // Linked-List-based
        for (int i = 1; i <= 10; i++) mystack.push(i);
        mystack.pop();
        mystack.pop();
        System.out.print(mystack.size());
        System.out.print(mystack.peek());
        System.out.print(mystack.pop());
        System.out.print(mystack.peek());
    }
}
```

Array-based
Result is 3 3 3 2

Linked-List-based
Result is 8 8 8 7

Remark: Linked-list provides more flexibility than arrays if one cannot predict maxSize; Speed/Complexity is similar
Linked-List: Implementation of Queues

- Linked-List implementation:
  - Does not require the knowledge of the maxSize- no need of wrapping
  - Only allocate the right amount of memory for queues items
  - Add to the rear of the queue: enqueue(item) → insertLast(item)
  - Remove from the front of queue: dequeue → deleteFirst
  - Easy to manipulate using double-ended feature
class LinkQueue {

    private LinkList2Ends thelist;

    // constructor
    public LinkQueue() { thelist = new LinkList2Ends(); }

    // methods
    public boolean isEmpty() {
        return (thelist.isEmpty);
    }

    public void enqueue(Object item) {
        thelist.insertLast(item);
    }

    public Object dequeue() {
        return thelist.deleteFirst();
    }

    public void displayQueue() {
        System.out.print("Stack front==>rear: ");
        thelist.displayList();
    }
}
class QueueApp1
{
    public static void main(String[] args)
    {
        ArrayQueue myqueue = new ArrayQueue(5); // array-based
        LinkQueue myqueue = new LinkQueue(); // Linked-List-based
        for(int i=1; i<=5; i++) myqueue.enqueue(i*10);
        System.out.println(myqueue.dequeue());
        myqueue.enqueue(60);
        System.out.println(myqueue.dequeue());
        System.out.println(myqueue.dequeue());
        System.out.println(myqueue.peekFront());
        System.out.println(myqueue.dequeue());
        System.out.println(myqueue.size());
    }
}
Abstract Data Types (ADT)

- A way of looking at a data structure, focusing on what it does and ignoring on how it does the job.
- In Java, any class represents a data structure
- ADT is a class considered without regard to its implementation
- Stacks, Queues, or Lists (sorted, unsorted) are examples of ADT: the user does not need to know how push, pop, etc. work, whether data is stored in an array, or a linked-list, etc.
- The ADT specification is what the class user sees – it is called an interface
Interfaces

- Interface contains a skeleton for public operations
  - No real code for each method
  - List of all public methods that specifies the ADT

```java
public interface StackADT{
    public boolean isEmpty();
    public void push(Object item);
    public Object pop();
    public Object peek();
    public void display();
    public int size();
}
```

```java
public interface QueueADT{
    public boolean isEmpty();
    public void enqueue(Object item);
    public Object dequeue();
    public Object peekFront();
    public void display();
    public int size();
}
```

- Different classes can implement the same interface (example: Stack)

```java
public class ArrayStack implements StackADT{
    \ to complete- Stack implemented using arrays
}
```

```java
public class LinkStack implements StackADT{
    \ to complete- Stack implemented using Linked-List
}
By decoupling the specification of the ADT from the implementation details, one can simplify the design process.

If the user relates only to the ADT interface, any changes on the implementation is backward compatible (the code will still work)

class StackApp1 {
    public static void main(String[] args) {
        //StackADT mystack = new ArrayStack(5); // array-based
        StackADT mystack = new LinkStack(); // Linked-List-based
        ...
    }
}

public void RPNcalc(LinkStack mystack) {
    // to complete
}

public void RPNcalc(StackADT mystack) {
    // to complete
}
Linked-List: Sorted List

- Insertion/removal likely to be faster than using an array (no shifts/moves), but still O(N)
- BTW: Can we use binary search on a sorted list?
  - In theory Yes, but there is no point doing it...not efficient...O(N)
- A sorted list is more difficult to implement than a sorted array
- To insert an item in a sorted list, the algorithm must first search through the list until it finds the appropriate place to put the item...There are many corner cases to consider...

Is the item to be inserted:
- At the beginning of the list?
- At the end?
- Somewhere in the middle?
- or is the list was empty to start with?
public void insert(int item) {
    Link o = new Link(item);
    Link previous = first; // first is private inside LinkedList class

    if (first == null) { // list is empty
        first = o;
        return;
    }

    if (first.data >= item) { // item needs to go at first spot of list
        o.next = first;
        first = o;
        return;
    }

    while (previous.next!=null && previous.next.data<item) {
        // traverse until end of list or spot is found
        previous = previous.next;
    }

    if (previous.next==null) { // check if at end of list
        previous.next = o;
    }

    else { // implement insertion
        o.next = previous.next;
        previous.next = o;
    }
}
Another implementation (textbook p214)

```java
public void insert(int item) {
    Link o = new Link(item);
    Link previous = null;
    Link current = first; // first is private inside LinkList class

    while (current!=null && current.data<item) {
        // traverse until end of list, if current=null or spot is found
        previous = current;
        current = current.next;
    }

    if (previous.next==null) {
        // check if at the beginning of the list
        first = o;
    }
    else {
        previous.next = o;
    }

    o.next = current;
}
```
Sorted-List Application: List Insertion Sort

- A sorted list can be used as an efficient sorting mechanism.
- Assuming we have an unsorted array, if you take the items from the array and insert them one by one to the sorted linked-list → the items will be automatically sorted. We can then put them back in the array.
- More efficient than insertion sort using array alone.
  - It is still $O(N^2)$, because of the number of times we need to make comparisons to insert in the sorted linked-list.
  - Each item is only copied twice (form array to list and list to array), so $2N$ copies vs $O(N^2)$.
- However, it requires twice as much memory (array+linked-list).