Brian Loughran

Assignment 6 – More on Lists

Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail.

The questions in this assignment give you the opportunity to explore a new data structure and to experiment with the hybrid implementation in Q3.

1. A deque (pronounced deck) is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends left and right. This is an access-restricted structure since no insertions or deletions can happen other than at the ends. Implement the deque as a doubly-linked list (not circular, no header). Write InsertLeft and DeleteRight.

```
Class Node {
   Node next = new Node;
   Node prev = new Node;
   Item item = new Item;

   Node(Item item) {
      this.item = item;
   }
}

Class Deque {
   Item head = new Item;
   Item tail = new Item;

   Deque() {
      this.head = null;
      this.tail = null;
   }

   public insertLeft(Item item) {
      Node node = new Node(item)
      this.head.prev = node;
      node.next = this.head;
      node.prev = null;
      this.head = node;
   }

   public deleteRight() {
      try {
         this.tail.prev.next = null;

      } catch(NullPointerException e) {
         throw new NullPointerException("Cannot Delete deque is empty");
```

```
        }
    }
}
```

2. Implement a deque from problem 1 as a doubly-linked circular list with a header. Write InsertRight and DeleteLeft.

```
Class Node {
    Node next = new Node;
    Node prev = new Node;
    Item item = new Item;

    Node(Item item) {
        this.item = item;
    }
}

Class Deque {
    Item header = new Node

    Deque() {
        this.header.item = null;
        this.header.next = this.head;
        this.header.tail = this.tail;
    }

    public void InsertRight(Item item) {
        Node node = new Node(item);
        this.header.next = node;
        node.next = header;
    }

    public void DeleteLeft() {
        try {
            header.prev.prev.next = header // set second to last node to point to header
        } catch (NullPointerException e) {
            throw new NullPointerException("Cannot delete from empty deque");
        }
    }
}
```

3. Write a set of routines for implementing several stacks and queues within a single array. Hint: Look at the lecture material on the hybrid implementation.

Brian Loughran

```
Class Hybrid {
   Item[] array = new Item[];

   Hybrid(int size) {
      this.arrray = new Item[size];
   }

   public int getNextOpenIndex() {
      int size = this.array.getLength();
      for(int j = 0; j < size; j++) {
         if(this.array[j] == null) {
            return j;
         }
      }
      return -1; // error code if array is full
   }
}

Class HybridElement {
   Item item = new Item();
   int next = 0;
   int index = 0;

   HybridElement (Item item, int next, in index) {
      this.item = item;
      this.next = next;
      this.index = index;
   }
}

Class HybridStack {
   HybridElement head = null;

   public void push(Item item) {
      openIndex = Hybrid.getNextOpenIndex();
      if(openIndex == -1) {
         throw new RuntimeException("Array full");
      }
      HybridElement element = new HybridElement (item, this.head.index, openIndex);
      this.head = element;
   }

   public Item pop() {
      HybridElement oldHead = this.head
      this.head = this.head.next;
      return oldHead;
```

```
        }
    }


    Class HybridQueue {
        HybridElement head = null;
        HybridElement tail = null;

        public void push(Item item) {
            openIndex = Hybrid.getNextOpenIndex();
            if(openIndex == -1) {
                throw new RuntimeException("Array full");
            }
            HybridElement element = new HybridElement (item, this.head.index, openIndex);
            this.head = element;
        }

        public Item pop() {
            HybridElement element = this.head;
            HybridElement predecessor = null;
            while(element.next != null) { // search for new tail
                predecessor = element;
                element = element.next;
            }
            this.tail = predecessor; // set new tail
            return element; // pop the old tail
        }
    }
}
```