Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 4: Stacks and Queues

Reading materials

- Goodrich, Tamassia, Goldwasser (6th), chapter 6
- OpenDSA (https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/): chapter 9.8-13

Contents

Joanna Klukowska
joannakl@cs.nyu.edu

1 Stacks ADT

Stacks are structures in which elements are always added and removed from the same end (depending on how you visualize the stack, you may wish to think of that end as the top of the stack). Stacks are **last in first out** (or **LIFO**) structures.

1.1 Example: CharStack ADT - Informal Specification

The CharStack contains a (possibly empty) collection of objects of type `Character`. The stack supports the following operations:

insert / push This operation adds a Character object, given as a parameter, to the top of the stack of characters.

remove / pop This operation removes and returns a Character object from the top of the stack of characters.

peek This operation returns a Character object from the top of the stack of characters.

toString This operation produces a meaningful String representation of the stack.

1.2 Example: CharStack ADT - Formal Specification

```
1
2
3 public interface CharStack {
4
5   /**
6     * Add a Character object to the top of the stack
7     * @param item
8     *   character to be added to the stack
9     */
10  public void push ( Character item ) ;
11
12  /**
13    * Remove and return a Character object from the top of the stack
14    * @return
15    *   Character from the top of the stack is returned and removed
16    *   from the stack. If stack is empty, null is returned.
17    */
18  public Character pop () ;
19
20  /**
21    * Return a Character object from the top of the stack.
22    * @return
23    *   Character from the top of the stack is returned.
24    *   If stack is empty, null is returned.
25    */
26  public Character peek () ;
27
28
29  /**
30    * Produces string representation of the stack.
31    * @return
32    *   Returns a String object that contains all characters
33    *   stored on the stack. Character objects are separated
34    *   by spaces. The top of the stack is the rightmost character
35    *   in the returned string.
36    */
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
37
38   public String toString () ;
39 }
40
41
42
```

See http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html for Java's generic `Stack<E>` class.

2 Queue ADT

Queues are structures in which elements are added to one end (rear/back of a queue) and removed from the other end (front of a queue). Queues are **first in first out** structures (**FIFO**).

2.1 Example: ProcessQueue ADT - Informal Specification

The **`ProcessQueue`** contains a (possibly empty) collection of objects of type **`Process`**. **`Process`** is a user defined class representing a process waiting to be scheduled to run on the core of a CPU of a computer. The queue supports the following operations:

insert / enqueue This operation adds a **`Process`** object, given as a parameter, to the end of the queue of processes.

remove / dequeue This operation removes and returns a **`Process`** object from the front of the queue of processes.

toString This operation produces a meaningful **`String`** representation of the stack.

2.2 Example: PrintJobQueueADT - Formal Specification

```
1 public interface PrintJobQueue {
2   /**
3     * Add a PrintJob object to the queue.
4     * @param item
5     *   PrintJob object to be added to the queue
6     */
7   public void enqueue ( PrintJob item ) ;
8
9   /**
10    * Remove and return the PrintJob object
11    * from the front of the queue
12    * @return
13    *   The PrintJob object from the front of the queue.
14    */
15  public PrintJob dequeue () ;
16
17  /**
18    * Return the PrintJob object
19    * from the front of the queue
20    * (nothing is removed from the queue)
21    * @return
22    *   The PrintJob object from the front of the queue.
23    */
24  public PrintJob peek () ;
25
26  /**
27    * Compute a string representation of the queue.
28    * @return
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
29      *     String object representing the queue. The string
30      *     should contain the current list of PrintJobs on the
31      *     queue. Each job should be on a separate line
32      *     listing the username of the user who submitted the job,
33      *     the number of pages remaining and total
34      *     and time that the job was submitted.
35      */
36    public String toString () ;
37  }
```

See http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html for Java's generic Queue¡E¿ interface. Notice how different methods are handled there when the queue is empty or full.

3 Array Based Implementation Stack and Queue

We first look at the implementation of the three ADTs using storage based on arrays. Many of the implemetnation considerations are the same as we discussed for an array-based implementation of the list: how big should the initial array be, what should be the rules for growing the array, where exactly (which intexes) should the newly added elements be placed, etc.

3.1 Implementing a Stack ADT Using an Array

When implementing a stack using list, we have an additional decision to make: where should the **top** element be.

Where should the top be? The **top** is the end at which we need to perform all push (addition), pop (removal) and peek operations. We have two options:

- index zero (or front) of the array - this is a very bad choice that leads to slow operations (do not ever use this!), why?
- index of the last element of the array - this leads to efficient implementation of the operations even though the top of the stack ends up moving as the stack grows (kind of like in a stack of real things)

3.2 Example: Character Stack

The following class provides the implementation for the **CharStack** interface from the previous section. For the documentation of the methods see the interface.

```
public class CharStackArray implements CharStack {

  private Character [] list;
  private int size;
  private int capacity;
  private static int DEFAULT_CAPACITY = 16;

  /**
   * Constructs an empty stack of characters with a specified capacity
   * (if provided capacity is less than or equal to zero, the capacity
   * is set to default 16).
   * @param capacity current capacity of an array for storing characters
   */
  public CharStackArray( int capacity )
  {
    //if capacity is negative or zero, reset it to
    //default value
    if (capacity <= 0 ) capacity = DEFAULT_CAPACITY;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    //allocate the array for storing characters
    list = new Character [capacity];

    //set the initial values
    size = 0;
    this.capacity = capacity;
  }
  /**
   * Constructs an empty stack of characters with a default capacity of 16.
   */
  public CharStackArray( )
  {
    this(DEFAULT_CAPACITY);
  }
  @Override
  public void push(Character item) {

    if (item == null ) return; //ignore null elements

    //if the stack is full, allocate a larger array
    if (  size == capacity )
      makeLarger();
    //add the new value to the top of the stack
    list[size] = item;
    size++;
  }
  @Override
  public Character pop() {
    //if stack is empty return null reference
    if ( size == 0 ) return null;
    //otherwise remove and return character from the top of the stack
    else {
      size--;
      return list[size];
    }
  }
  @Override
  public Character peek() {

    //if stack is empty return null reference
    if ( size == 0 ) return null;
    //otherwise remove and return character from the top of the stack
    else {
      return list[size-1];
    }
  }
  /*
   * Allocates an array twice the size of the current array used for
   * storing the stack and copies all the data to the new array.
   */
  private void makeLarger ()
  {
    //allocate larger array
    Character [] newList = new Character [capacity * 2 ];
    //copy the data over to the new array
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    for (int i = 0; i < capacity; i++)
    {
      newList[i] = list[i];
    }
    //reset list reference to the new array
    list = newList;
    //reset the capacity to the new value
    capacity = 2*capacity;

  }
  /* (non-Javadoc)
   * @see java.lang.Object#toString()
   */
  public String toString() {
    //prints the stack as a string
    //"stack(size): space separated elements"
    StringBuffer stack = new StringBuffer();
    stack.append( "stack(" +size + "): " );
    for (int i = 0; i < size; i++ ) {
      stack.append( list[i] + " " );
    }
    return stack.toString();
  }

}
```

3.3 Implementing a Queue ADT Using an Array

This one turns out to be a bit more complicated. We need to add to one end of the array and remove from the other. To optimize the performance of all the operations we want to make sure that we do not shift elements in the array during these operations. This leads to the implementation in which both ends move and we need to keep track of indexes that are the front and back of the queue.

3.4 Example: Print Job Queue implementation

The following class provides the implementation for the **`PrintJobQueue`** interface from the previous section. For the documentation of the methods see the interface.

```


public class PrintJobQueueArray implements PrintJobQueue {

  protected final static int DEFAULT_CAPACITY = 16;
  protected PrintJob [] queue;
  protected int capacity;
  protected int size;
  protected int front;
  protected int back;

  /**
   * Construct a PrintJobQueueArray object with a specified capacity
   * (capacity allows to predetermine the initial size
   * of array used for storage of the queue).
   * @param capacity
   *   initial capacity of an array to be used to store
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
   *   the queue
   */
  public PrintJobQueueArray(int capacity) {
    this.capacity = capacity;
    queue = new PrintJob [capacity];
    front = 0;
    back = capacity -1; //this way the next item entered will
            //wrap to the zero location
  }
  /**
   * Construct a PrintJobQueueArray object.
   */
  public PrintJobQueueArray() {
    this(DEFAULT_CAPACITY);
  }
  public void enqueue(PrintJob item) {
    //make sure we are not adding a null job
    if (item == null )
      return;
    //check if the queue is big enough to add another element
    if ( size == capacity )
      makeLarger();
    //advance back to point to the next availabe spot
    back = (back+1)%capacity;
    queue[back] = item;
    size++;
  }
  @Override
  public PrintJob dequeue() {
    if (size == 0)      return null;
    PrintJob nextJob = queue[front];
    //optionally we could set the removed location to null:
    queue[front] = null;
    front = (front+1)%capacity;
    size--;
    return nextJob;
  }
  @Override
  public PrintJob peek() {
    if (size == 0)
      return null;
    PrintJob nextJob = queue[front];
    return nextJob;
  }
  /*
   * Allocates an array twice the size of the current array used for
   * storing the queue and copies all the data to the new array.
   */
  private void makeLarger ()
  {
    //allocate larger array
    PrintJob [] newQueue = new PrintJob [capacity * 2];
    //copy the data over to the new array
    int current = front;
    for (int i = 0; i < capacity; i++)
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
    {
      newQueue[i] = queue[current];
      current = (current+1) % capacity;
    }
    //reset list reference to the new array
    queue = newQueue;
    //reset the capacity to the new value
    capacity = 2*capacity;
    //reset front and back
    front = 0;
    back = size-1;

  }
  /* (non-Javadoc)
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    StringBuffer queueString = new StringBuffer();
    int current;
    for (int i = 0; i < size; i++ ) {
      current = (front+i)%capacity;
      queueString.append( queue[current].toString() );
    }

    return queueString.toString();
  }
}
```

4 Reference-Based Implementation of Stack and Queue

4.1 Stacks

In implementing linked structure based stacks we are going to reuse the code developed in the previous section. The stack class implementation needs to implement the interface that we have been using previously (but this time it is written for String objects):

```
1 public interface StringStack {
2   public void push ( String item ) ;
3   public String pop () ;
4   public String peek () ;
5   public String toString () ;
6 }
```

4.1.1 Wrapper Implementation

The quick and dirty solution is to define a stack class that has our LinkedList¡String¿ as a data field and calls its respective methods. This approach reuses methods of the LinkedList¡String¿ class and **wraps** them in its own methods appropriate for a stack.

The problem with this implementation is that it doubles the number of function calls for each operation.

Joanna Klukowska
joannakl@cs.nyu.edu

4.1.2 Another Inefficient Implementation

We can also implement the stack class with its own `head` reference, `numOfElements` variable and the methods

- `push()` that is identical to `addBack()`,
- `pop()` that is identical to `removeBack()`,
- `peek()` that uses `get()` method to obtain the last element,
- `toString()`.

This approach avoids making duplicate method calls, but is still very inefficient: we need to traverse the entire list for every insertion, deletion and get operation.

4.1.3 An Efficient Stack Implementation

Since a stack data structure only requires accesses (push, pop, peek) on one end of the linked list, the top of a stack should be at the very beginning of the list. This way every access takes only one step and there is never a reason to traverse the entire list.

An efficient linked list based stack implementation should have its own head reference, (optional) numOfElements variable and the methods

- `push()` that is identical to `addFront()`,
- `pop()` that is identical to `removeFront()`,
- `peek()` that is identical to `get()` called with last index,
- `toString()`.

4.2 Queues

As with the stack, in implementing linked structure based queues we are going to reuse the code developed for the linked list. The queue class implementation needs to implement the interface that we have been using previously.

A queue requires accesses to both ends of the list, so if we only keep the reference to the **`head`** of the list, on one of its operations the entire list has to be traversed. For an efficient implementation of a reference-based queue, we will need to keep a **`tail`**-like reference in the implementation.

There are two options to consider:

- add new elements at the **`head`** and remove from the **`tail`**
- add new elements at the **`tail`** and remove from the **`head`**

The first of those approaches results in $O(N)$ performance for removals (since the **`tail`** reference cannot easily be updated to the node before the one that needs to be removed. The second one gives $O(1)$ performance on adding and removing from the queue.

To summarize we need a reference based implementation that keeps a **`front`** reference, pointing to the first node (or **`head`**), and a **`back`** reference, pointing to the last node (or **`tail`**).