CSC 321: Data Structures

Fall 2018

Lists, stacks & queues

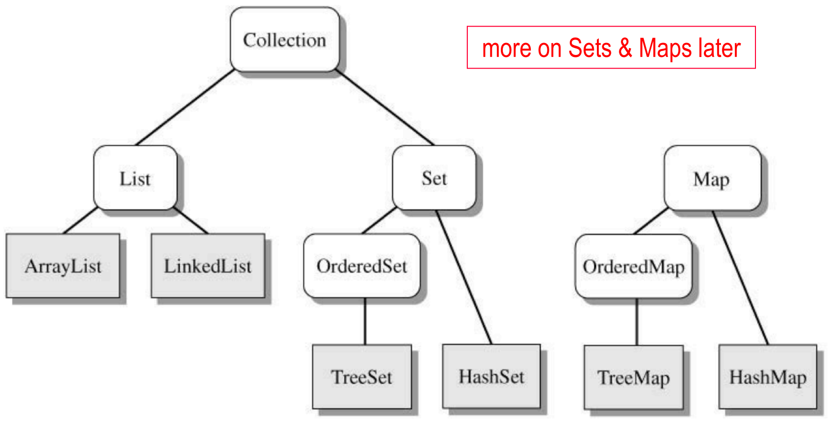
- Collection classes:
  - — List (ArrayList, LinkedList), Set (TreeSet, HashSet), Map (TreeMap, HashMap)
- ArrayList performance and implementation
- LinkedList performance
- Stacks
- Queues

Java Collection classes

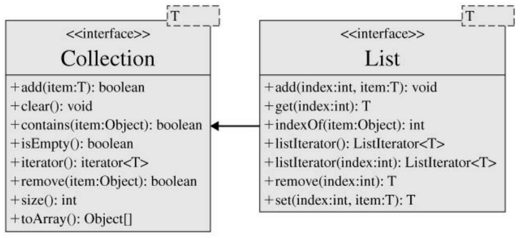
a collection is an object (i.e., data structure) that holds other objects

the Java Collection Framework is a group of generic collections
- defined using interfaces abstract classes, and inheritance

more on Sets & Maps later



ArrayList performance



T
<<interface>>
Collection
+add(item:T): boolean
+clear(): void
+contains(item:Object): boolean
+isEmpty(): boolean
+iterator(): iterator<T>
+remove(item:Object): boolean
+size(): int
+toArray(): Object[]
T
<<interface>>
List
+add(index:int, item:T): void
+get(index:int): T
+indexOf(item:Object): int
+listIterator(): ListIterator<T>
+listIterator(index:int): ListIterator<T>
+remove(index:int): T
+set(index:int, item:T): T

recall: ArrayList implements the List interface

- which is itself an extension of the Collection interface

- underlying list structure is an array

`get(index), add(item), set(index, item)` → O(1)

`add(index, item), indexOf(item), contains(item), remove(index), remove(item)` → O(N)

3

ArrayList implementation

the ArrayList class has as fields

- the underlying array
- number of items stored

the default initial capacity is defined by a constant

- capacity != size

```
public class MyArrayList<E> implements Iterable<E>{
    private static final int INIT_SIZE = 10;
    private E[] items;
    private int numStored;

    public MyArrayList() {
        this.clear();
    }

    public void clear() {
        this.numStored = 0;
        this.ensureCapacity(INIT_SIZE);
    }

    public void ensureCapacity(int newCapacity) {
        if (newCapacity > this.size()) {
            E[] old = this.items;
            this.items = (E[]) new Object[newCapacity];
            for (int i = 0; i < this.size(); i++) {
                this.items[i] = old[i];
            }
        }
    }
    .
    .
    .
```

interestingly: you can't create a generic array

```
this.items = new E[capacity];    // ILLEGAL
```

can work around this by creating an array of Objects, then casting to the generic array type

4

ArrayList: add

the add method

- throws an exception if the index is out of bounds
- calls ensureCapacity to resize the array if full
- shifts elements to the right of the desired index
- finally, inserts the new value and increments the count

the add-at-end method calls this one

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "ArrayList add()", this.size());
    if (this.items.length == this.size()) {
        this.ensureCapacity(2*this.size() + 1);
    }

    for (int i = this.size(); i > index; i--) {
        this.items[i] = this.items[i-1];
    }
    this.items[index] = newItem;
    this.numStored++;
}

private void rangeCheck(int index, String msg, int upper) {
    if (index < 0 || index > upper)
        throw new IndexOutOfBoundsException("\n" + msg +
                ": index " + index + " out of bounds. " +
                "Should be in the range 0 to " + upper);
}


public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

5

ArrayList: size, get, set, indexOf, contains

size method

- returns the item count

get method

- checks the index bounds, then simply accesses the array

set method

- checks the index bounds, then assigns the value

indexOf method

- performs a sequential search

contains method

- uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "ArrayList get()", this.size()-1);
    return items[index];
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "ArrayList set()", this.size()-1);
    E oldItem = this.items[index];
    this.items[index] = newItem;
    return oldItem;
}

public int indexOf(E oldItem) {
    for (int i = 0; i < this.size(); i++) {
        if (oldItem.equals(this.items[i])) {
            return i;
        }
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

6

ArrayList: remove

the remove method

- checks the index bounds
- then shifts items to the left and decrements the count
- note: could shrink size if becomes ½ empty

the other remove

- calls indexOf to find the item, then calls remove(index)

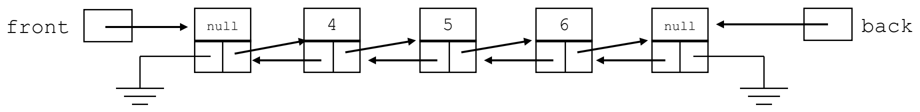
```
public void remove(int index) {
    this.rangeCheck(index, "ArrayList remove()", this.size()-1);

    for (int i = index; i < this.size()-1; i++) {
        this.items[i] = this.items[i+1];
    }
    this.numStored--;
}


public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}
```

7

ArrayLists vs. LinkedLists

LinkedList is an alternative List structure

- stores elements in a sequence but allows for more efficient interior insertion/deletion
- elements contain links that reference previous and successor elements in the list



- can access/add/remove from either end in O(1)
- if given a reference to an interior element, can reroute the links to add/remove an element in O(1) [more later when we consider iterators]

```
getFirst(), getLast(),
add(item), addFirst(), addLast()
removeFirst(), removeLast()
```
→ O(1)

```
get(index), set(index, item),
add(index, item), indexOf(item), contains(item),
remove(index), remove(item)
```
→ O(N)

8

Lists & stacks

stack

- a stack is a special kind of (simplified) list
- can only add/delete/look at one end (commonly referred to as the top)

DATA: sequence of items
OPERATIONS: push on top, peek at top, pop off top, check if empty, get size

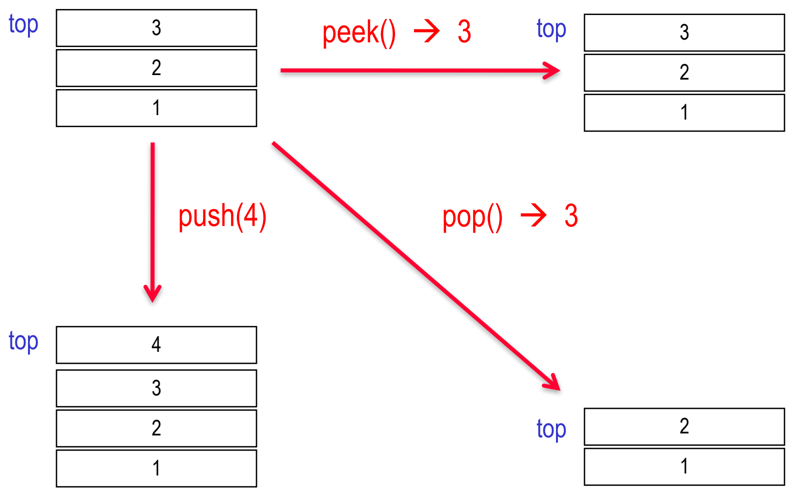
these are the ONLY operations allowed on a stack
— stacks are useful because they are simple, easy to understand
— each operation is O(1)

- PEZ dispenser
- deck of cards
- cars in a driveway
- method activation records (later)

a stack is also known as
- push-down list
- last-in-first-out (LIFO) list

9

Stack examples



10

Stack exercise

- start with empty stack
- PUSH 1
- PUSH 2
- PUSH 3
- PEEK
- PUSH 4
- POP
- POP
- PEEK
- PUSH 5

11

`Stack<T>` class

since a stack is a common data structure, a predefined Java class exists

```
import java.util.Stack;
```

- `Stack<T>` is generic to allow any type of object to be stored

```
Stack<String> wordStack = new Stack<String>();
Stack<Integer> numStack = new Stack<Integer>();
```

- standard `Stack<T>` methods

```
public T push(T item);          // adds item to top of stack
public T pop();                 // removes item at top of stack
public T peek();                // returns item at top of stack
public boolean empty();         // returns true if empty
public int size();              // returns size of stack
```

12

Stack application

consider mathematical expressions such as the following

- a compiler must verify such expressions are of the correct form

(A * (B + C))   ([A * (B + C)] + [D * E])

attempt 1: count number of left and right delimeters; if equal, then OK

what about: (A * B) +)C(

attempt 2: start a counter at 0, +1 for each left delimiter and -1 for each right

if it never becomes negative and ends at 0, then OK

what about: ([A + B) + C]

stack-based solution:

- start with an empty stack of characters
- traverse the expression from left to right
  - if next character is a left delimiter, push onto the stack
  - if next character is a right delimiter, must match the top of the stack

13

Delimiter matching

```
import java.util.Stack;

public class DelimiterChecker {
  private static final String DELIMITERS = "()[]{}<>";

  public static boolean check(String expr) {
      Stack<Character> delimStack = new Stack<Character>();

      for (int i = 0; i < expr.length(); i++) {
         char ch = expr.charAt(i);
         if (DelimiterChecker.isLeftDelimiter(ch)) {
             delimStack.push(ch);
         }
         else if (DelimiterChecker.isRightDelimiter(ch)) {
             if (!delimStack.empty() &&
                  DelimiterChecker.match(delimStack.peek(), ch)) {
                 delimStack.pop();
             }
             else {
                 return false;
             }
         }
     }

    return delimStack.empty();
  }
}
```

how would you implement the helpers?

`isLeftDelimiter`
`isRightDelimiter`
`match`

14

Run-time stack

when a method is called in Java (or any language):

- suspend the current execution sequence
- allocate space for parameters, locals, return value, …
- transfer control to the new method

when the method terminates:

- deallocate parameters, locals, …
- transfer control back to the calling point (& possibly return a value)

note: method invocations are LIFO entities

- `main` is called first, terminates last
- if main calls `foo` and `foo` calls `bar`, then
  `bar` terminates before `foo` which terminates before `main`

➔ a stack is a natural data structure for storing information about method calls and the state of the execution

15

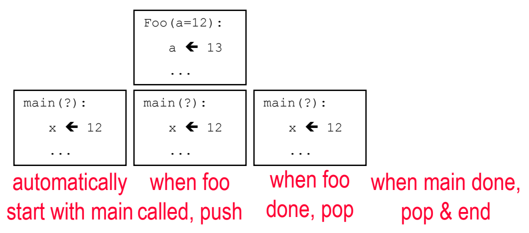
Run-time stack (cont.)

an activation record stores info (parameters, locals, …) for each invocation of a method

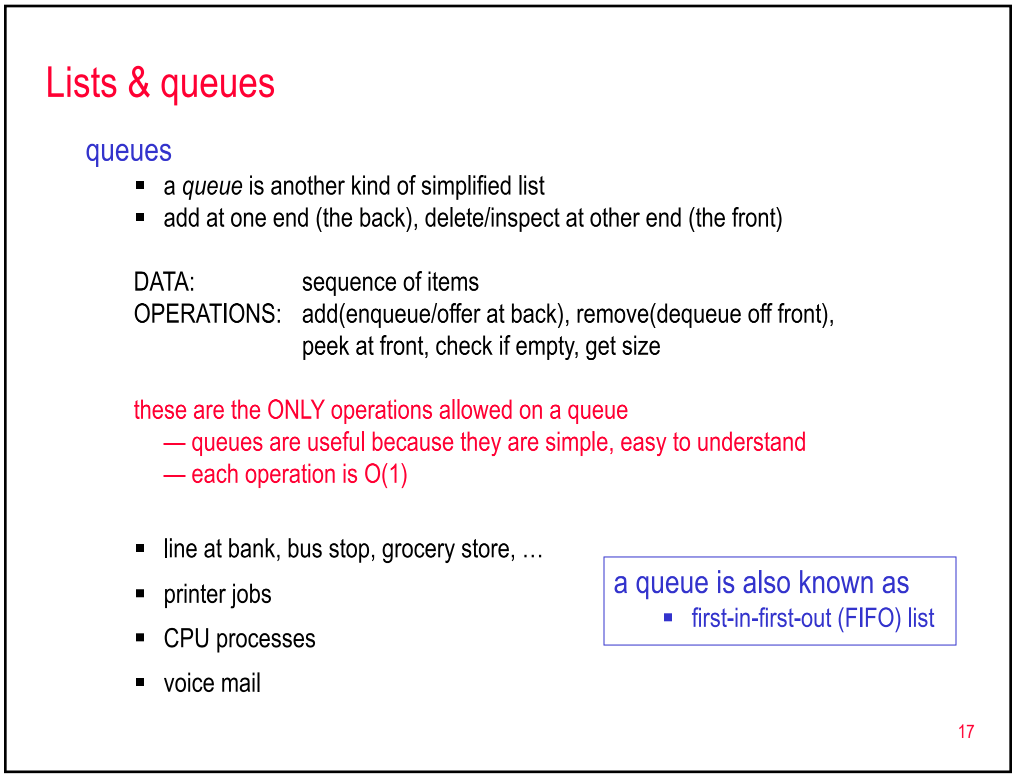
- when the method is called, an activation record is pushed onto the stack
- when the method terminates, its activation record is popped

- note that the currently executing method is always at the top of the stack

```
public class Demo {
  public static void main(String[] args) {
    int x = 12;

    Demo.foo(x);
    System.out.println("main " + x);
  }

  public static void foo(int a) {
    a++;
    System.out.println("foo " + a);
  }
}
```



16



Lists & queues

queues

- a *queue* is another kind of simplified list
- add at one end (the back), delete/inspect at other end (the front)

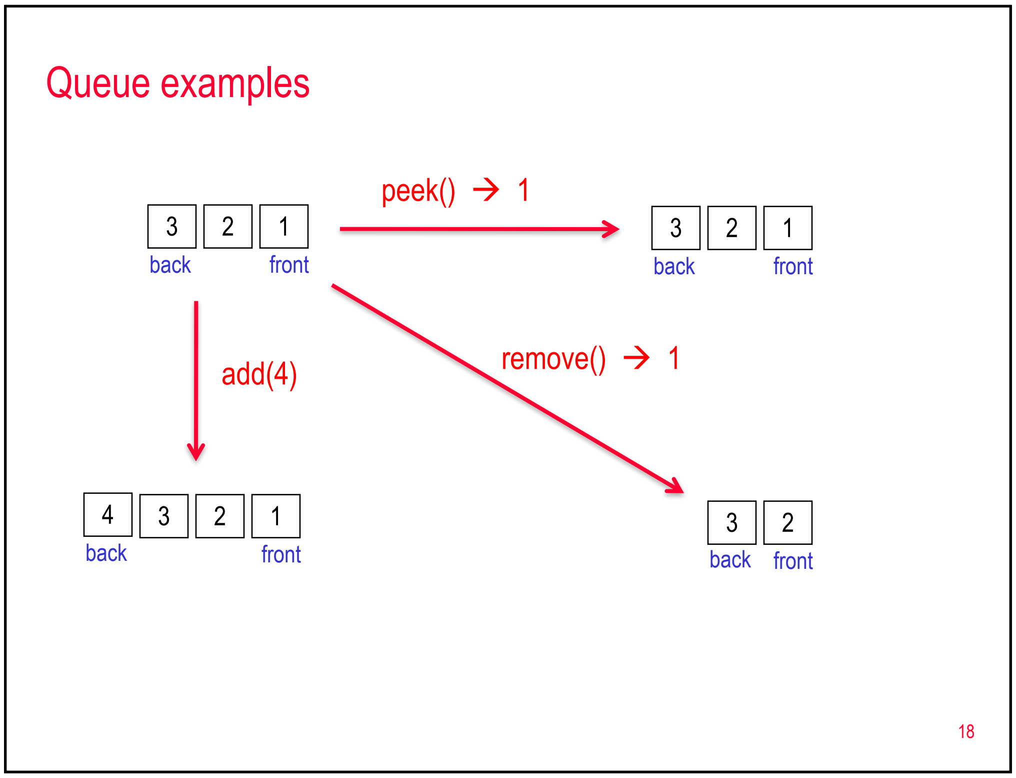
DATA:	sequence of items
OPERATIONS:	add(enqueue/offer at back), remove(dequeue off front), peek at front, check if empty, get size

these are the ONLY operations allowed on a queue
— queues are useful because they are simple, easy to understand
— each operation is O(1)

- line at bank, bus stop, grocery store, …
- printer jobs
- CPU processes
- voice mail

a queue is also known as
- first-in-first-out (FIFO) list

17



Queue examples

18

Queue exercise

- start with empty queue
- ADD 1
- ADD 2
- ADD 3
- PEEK
- ADD 4
- REMOVE
- REMOVE
- PEEK
- ADD 5

19

`Queue` interface

a queue is a common data structure, with many variations

- Java provides a `Queue` interface
- also provides several classes that implement the interface (with different underlying implementations/tradeoffs)

`java.util.Queue<T>` interface

```
public boolean add(T newItem);
public T remove();
public T peek();
public boolean empty();
public int size();
```

`java.util.LinkedList<T>` implements the `Queue` interface

```
Queue<Integer> numQ = new LinkedList<Integer>();

for (int i = 1; i <= 10; i++) {
  numQ.add(i);
}

while ( !numQ.empty() ) {
  System.out.println(numQ.peek());
  numQ.remove();
}
```

```
Queue<Integer> q1 = new LinkedList<Integer>();
Queue<Integer> q2 = new LinkedList<Integer>();

for (int i = 1; i <= 10; i++) {
  q1.add(i);
}

while ( !q1.empty() ) {
  q2.add(q1.remove());
}

while ( !q2.empty() ) {
  System.out.println(q2.remove());
}
```

20

Queues and simulation

queues are especially useful for simulating events

e.g., consider simulating a 1-teller bank

- customers enter a queue and are served FCFS (or FIFO)
- can treat the arrival of a customer and their transaction length as random events

```
What is the time duration (in minutes) to be simulated? 10
What percentage of the time (0-100) does a customer arrive? 30

2: Adding customer 1 (job length = 4)
2:   Serving customer 1 (finish at 6)
4: Adding customer 2 (job length = 3)
6:     Finished customer 1
6:   Serving customer 2 (finish at 9)
9:     Finished customer 2
```

if multiple tellers are available,

- could have a separate queue for each teller (FAIRNESS ISSUES?)
- or, could still have one queue, whenever a teller becomes free he/she serves the customer at the front