ITI 1121. Introduction to Computing II
Winter 2018

Assignment 3
(Last modified on March 15, 2018)

Deadline: March 23, 2018, 11:30 pm

[PDF]

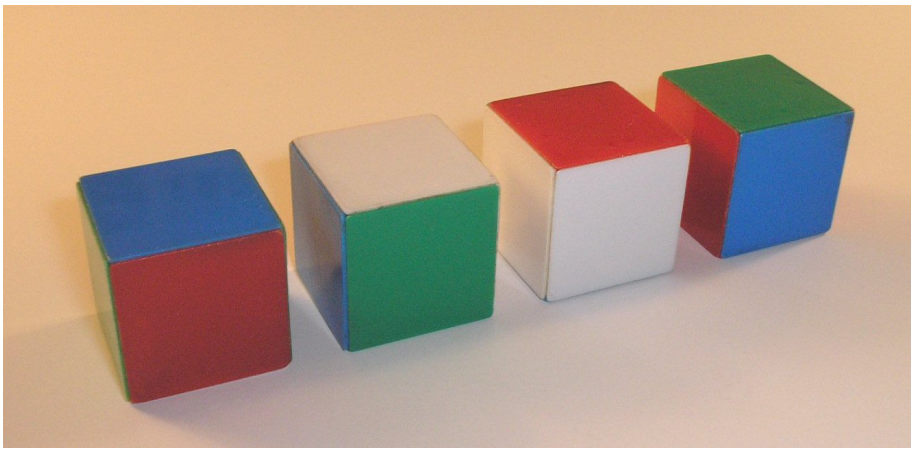
Learning objectives

- Implementing a queue-based algorithm
- Explain the term deep copy in your own words
- Throwing exceptions when necessary
- Explaining state space search algorithms in your own words

Background

State space search is a paradigm from the field of **artificial intelligence** where solutions to a problem can be found by searching through a space of states. One of the simplest strategies is called **breadth-first search**. It explores the space as follows: starting from an initial state, all the states that can be reached in one step are visited. Next, the algorithm visits all the states that can be reached in two steps, then three steps, etc. The partial solutions that are generated are kept in a queue. Each time a partial solution is generated, the algorithm checks that the solution is valid, if not, the solution is discarded. If the solution is valid, then the algorithm checks if it reaches the goal. If so, the solution is saved. The solutions that are valid but that are not reaching the goal yet are kept in the queue.

Breadth-first search has many applications. Herein, we use it to efficiently find a solution for the puzzle known as **instant insanity**. The puzzle comprises four cubes with faces painted with four colours. Each cube is unique. The goal for solving the puzzle is stacking the four cubes one on the top of another to create a tower where each face of the tower has no duplicated colour, equivalently, each face of the tower has all the four colours. Each cube has 24 unique orientations, therefore, the brute-force approach, also known as generate-and-test, considers all 24^4 solutions!



Source: https://commons.wikimedia.org/wiki/File:InstantInsanity.jpg

Implementation

This assignment comprises three classes, **Cube**, **Solution**, and **Solve**, as well as the enum type **Color**.

- Objects of the class **Cube** are used to represent the cubes in the puzzle **instant insanity**.
- Objects of the class **Solution** represent partial solutions to the problem. A solution stores 1, 2, 3 or 4 cubes.
- Finally, the class **Solve** provides two methods for solving the **instant insanity** problem: **breadthFirstSearch** and **generateAndTest**.

Follow the instructions below. Make sure to through exceptions when appropriate.

1 Color

Create an **enum type**, called **Color**, to represent the four colours for the faces of the cubes: BLUE, GREEN, RED, and WHITE. See Appendix B for further information about **enum types**.

2 Cube

Implement the class **Cube**.

- Each cube memorizes the colours of its six faces.
- The constructor **Cube(Color[] faces)** is used to create a new cube with specific colours for its faces. The array specifies the colours in the following order: up, front, right, back, left, and down.
- A **Cube** has (six) getters returning the **Color** of each face: **getUp**, **getFront**, **getRight**, **getBack**, **getLeft**, and **getDown**.
- A **Cube** has a **toString** method that returns a **String** representation of the **Cube**. The following example shows the expected output:

```
1 Cube c;
2 c = new Cube(new Color[]{Color.BLUE, Color.GREEN, Color.WHITE, Color.GREEN, Color.BLUE, Color.RED});
3 System.out.println(c);
```

```
[BLUE, GREEN, WHITE, GREEN, BLUE, RED]
```

2.1 Changing the orientation of a cube

Each cube has 24 possible orientations: 6 ways to select the side facing up and 4 rotations. We propose a convenient mechanism to iterate through all 24 possible orientations.

- **hasNext()** return **true** if and only if a call to the method **next** would succeed, and **false** otherwise.
- Each call to the method **next** changes the orientation of the cube. It throws an exception **IllegalStateException** if the call to **next** sets the orientation of the **Cube** to one that has been seen since the last call to the method **reset**.
- The method **reset** puts the cube in its original orientation (the orientation that the cube had when it was first created).

The following Java program illustrates the intended use for **hasNext**, **next**, and **reset**.

```
1  public class Test {
2
3      public static void main(String[] args) {
4
5          Cube c;
6          c = new Cube(new Color[]{Color.BLUE, Color.GREEN, Color.WHITE, Color.GREEN, Color.BLUE, Color.RED});
7
8          c.reset();
9          while (c.hasNext()) {
10             c.next();
11             System.out.println(c);
```

```
12          }
13
14          System.out.println("reset:");
15          c.reset();
16          while (c.hasNext()) {
17              c.next();
18              System.out.println(c);
19          }
20
21      }
22  }
```

The execution produces the following output [1]:

```
> java Test
[BLUE, GREEN, WHITE, GREEN, BLUE, RED]
[BLUE, BLUE, GREEN, WHITE, GREEN, RED]
[BLUE, GREEN, BLUE, GREEN, WHITE, RED]
[BLUE, WHITE, GREEN, BLUE, GREEN, RED]
[GREEN, WHITE, BLUE, BLUE, RED, GREEN]
...
[BLUE, GREEN, RED, GREEN, BLUE, WHITE]
reset:
[BLUE, GREEN, WHITE, GREEN, BLUE, RED]
[BLUE, BLUE, GREEN, WHITE, GREEN, RED]
[BLUE, GREEN, BLUE, GREEN, WHITE, RED]
[BLUE, WHITE, GREEN, BLUE, GREEN, RED]
[GREEN, WHITE, BLUE, BLUE, RED, GREEN]
...
[BLUE, GREEN, RED, GREEN, BLUE, WHITE]
```

Let's define four operations to change the orientation of the cube: **Rotate**, **RightRoll**, **LeftRoll**, and **Identity**.

- **Rotate**: rotates the cube to the right around the top-bottom axis so that the left side is now facing front.
- **RightRoll**: rolls the cube to the right around the back-front axis so that the left side is now up.
- **LeftRoll**: rolls the cube to the left around the back-front axis so that the right side is now up.
- **Identity**: returns all the faces to their original state (colours).

Following a call to the method **reset**, each call to the method **next** changes the orientation of the cube according to the following list of operations:

- Identity, Rotate, Rotate, Rotate, RightRoll, Rotate, Rotate, Rotate, RightRoll, Rotate, Rotate, Rotate, LeftRoll, Rotate, Rotate, Rotate, LeftRoll, Rotate, Rotate, Rotate, RightRoll, Rotate, Rotate, Rotate.

You must add all the necessary instance variables to implement the methods: **hasNext**, **next** and **reset**.

2.2 Copy

Implement the following two methods:

- **Cube(Cube other)**: the constructor initializes **this** cube to be an identical but independent copy of **other**. In other words, **this Cube** is a **deep copy** of the cube designated by **other**. See Appendix A for information about **deep copy**.
- **Cube copy()**: returns a **deep copy** of this **Cube**.

[1] The complete listing is shown in Appendix: C.

3 Solution

A **Solution** is a data structure to store cubes. It represents a partial solution to the **instant insanity problem**. It portrays a pile of n cubes, where $n \in [1, 2, 3, 4]$. Implement the following two constructors:

- **Solution(Cube[] cubes)**: initializes this solution using the cubes provided in the array **cubes**. Because the cubes are mutable, the solution must also copy the cubes.
- **Solution(Solution other, Cube c)**: initializes this solution using the specified information. It receives a partial solution and a cube. The new solution has the same elements, in the same order, as the solution designated by **other**. The value **null** is a valid value for **other**, but not for **cube**. Make sure that **this** solution and **other** do not share cubes. See Appendix A.

Each **Solution** has the following instance methods:

- **int size()**: returns the logical size of the data structure. In other words, it returns the number of cubes that are stored in this solution.
- **Cube getCube(int pos)**: returns the reference of the **Cube** at the specified position.
- **boolean isValid()**: returns **true** if each side of the pile of cubes has no duplicated colour, and **false** otherwise.
- **boolean isValid(Cube next)**: returns **true** the solution would remain valid when adding the cube designated by **next** to the solution, and **false** otherwise.
- **String toString()**: returns a **String** representation of the solution.

3.1 getNumberOfCalls and resetNumberOfCalls

The class **Solve** implements two algorithms for solving the **instant insanity problem**. It compares the two methods in terms of the number of calls to the method **isValid** of solutions. Add the necessary variables to implement the following two methods:

- **getNumberOfCalls** returns the total number of calls to the method **isValid** of any object of the class **Solution** since the last call to the method **resetNumberOfCalls**.
- **resetNumberOfCalls** is used to initialize the statistic.

4 Solve

The class **Solve** provides three class methods: **generateAndTest**, **breadthFirstSearch**, and **main**. Use the following four cubes:

- BLUE,GREEN,WHITE,GREEN,BLUE,RED
- WHITE,GREEN,BLUE,WHITE,RED,RED
- GREEN,WHITE,RED,BLUE,RED,RED
- BLUE,RED,GREEN,GREEN,WHITE,WHITE

4.1 generateAndTest

- Queue<Solution> generateAndTest(): The method **generateAndTest** finds all the solutions to the **instant insanity problem** by exhaustively generating all the possible solutions. It returns a **Queue** that contains all the valid solutions to the problem.
- At the start of its execution, the method resets the statistic for counting the number of calls to **isValid**. After having found all the solutions, the method prints the total number of calls to **isValid**.

4.2 breadthFirstSearch

- Queue<Solution> breadthFirstSearch (): The method **breadthFirstSearch** finds all the solutions to the **instant insanity problem** using the "breadth-first-search" algorithm presented in class. It returns a **Queue** that contains all the valid solutions to the problem.
- At the start of its execution, the method resets the statistic for counting the number of calls to **isValid**. After having found all the solutions, the method prints the total number of calls to **isValid**.

Here is a brief summary of the algorithm seen in class:

Result: The list of all valid solutions.
Create two queues: **open** and **solutions**;
Initialize **open** to contain all the valid initial solutions;
while ***open*** *is not empty* **do**
    Take the front element out of the **open** queue, call this **current**;
    foreach *possible extension of the current solution* **do**
        if *the extension is valid* **then**
            if *the extension reaches goal* **then**
                add to **solutions**;
            else
                add to **open**;
            end
        end
    end
end

4.3 main

Create a **main** method with the following content:

```
1  long start, stop;
2
3  System.out.println("generateAndTest:");
4  start = System.currentTimeMillis(); // could also use nanoTime
5
6  generateAndTest();
7
8  stop = System.currentTimeMillis();
9  System.out.println("Elapsed time: " + (stop-start) + " milliseconds");
10
11 System.out.println("breadthFirstSearch:");
12 start = System.currentTimeMillis();
13
14 breadthFirstSearch();
15
16 stop = System.currentTimeMillis();
17 System.out.println("Elapsed time: " + (stop-start) + " milliseconds");
```

5 Bonus question (10 marks)

Alas, the proposed algorithms are generating redondant (symmetric) solutions! In a directory called bonus, for the class **Solve**, implement a method called **bonus()** that returns non-redondant solutions. Add a file called **README.txt** where you explain your strategy.

On Internet, you will will find receipes for solving the **instant insanity problem**. Hard coding these solutions is not enough to earn bonus marks, you must provide an algorithm that efficiently removes redundancy from the state search space.

Files

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **a3_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment

(simply repeat the same number if your team has one member). The folder must contain the following files.

- A text file **README.txt** which contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- The source code of **all** your files (all the necessary files to compile and execute your program): **Color.java**, **Cube.java**, **Solution.java**, **Solve.java**, **Queue.java** and **LinkedQueue.java** or **CircularQueue.java**.
- The corresponding JavaDoc doc directory.
- StudentInfo.java, properly completed and properly called from your main method.

WARNINGS

- Failing to strictly follow the submission instructions will cause automated test tools to fail on your submission. Consequently, your submission will **not** get marked.
- A tool will be used to detect similarities between submissions. We will run that tool on all submissions, across all the sections of the course (including French sections). Submissions that are flagged by the tool will receive a mark of 0.
- It is your responsibility to ensure that your submission is indeed received by the back-end software, Brightspace. If your submission is not there by the deadline, it will obviously **not** get marked.
- Late submissions will not be accepted.

Resources

- `https://www.youtube.com/watch?v=t89nWBE3y84`
- The original idea for the assignment comes from: *Introduction to functional programming* by Richard Bird and Philip Wadler, Prentice Hall, 1988. Page 165.
- `https://en.wikipedia.org/wiki/Instant_Insanity`
- Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2010. Chapter 3, Solving problems by searching.

A Shallow copy versus Deep copy

As you know, objects have variables which are either a primitive type, or a reference type. Primitive variables hold a value from one of the language primitive type, while reference variables hold a reference (the address) of another object (including arrays, which are objects in Java).

If you are copying the current state of an object, in order to obtain a duplicate object, you will create a copy of each of the variables. By doing so, the value of each instance primitive variable will be duplicated (thus, modifying one of these values in one of the copy will not modify the value on the other copy). However, with reference variables, what will be copied is the actual reference, the address of the object that this variable is pointing at. Consequently, the reference variables in both the original object and the duplicated object will point at the same address, and the reference variables will refer to the same objects. This is known as a **shallow** copy: you indeed have two objects, but they share all the objects pointed at by their instance reference variables. The Figure 1 provides an example: the object referenced by variable **b** is a shallow copy of the object referenced by variable **a**: it has its own copies of the instances variables, but the references variables **title** and **time** are referencing the same objects.

Often, a shallow copy is not adequate: what is required is a so-called **deep** copy. A deep copy differs from a shallow copy in that objects referenced by reference variable must also be recursively duplicated, in such a way that when the initial object is (deep) copied, the copy does not share any reference with the initial object. The Figure 2 provides an example: this time, the object referenced by variable **b** is a deep copy of the object referenced by variable **a**: now, the references variables **title** and **time** are referencing different objects. Note that, in turn, the objects referenced by the variable **time** have also been deep-copied. The entire set of objects reachable from **a** have been duplicated.

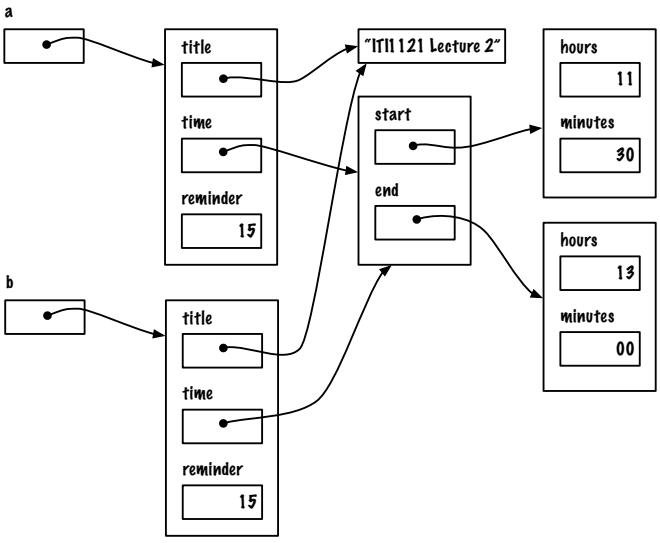


Figure 1: A example of a shallow copy of objects.

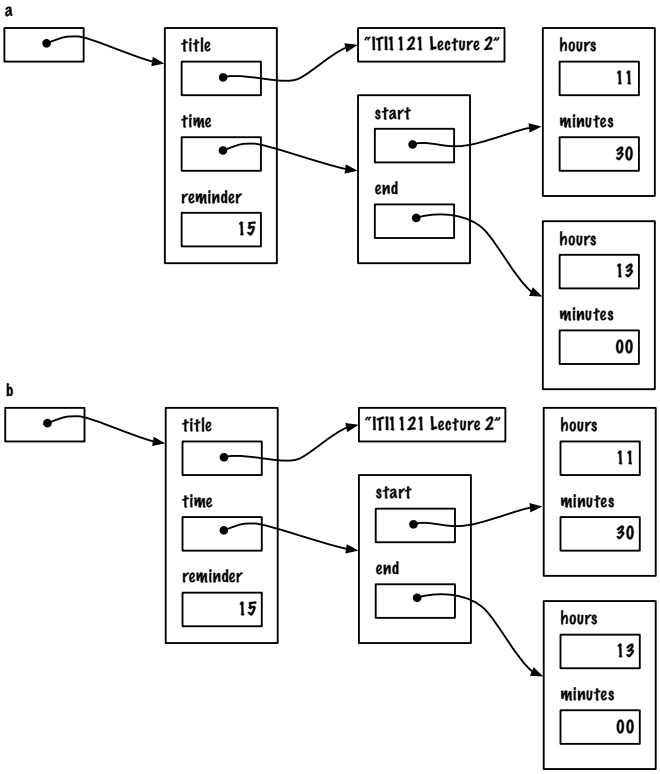


Figure 2: A example of a deep copy of objects.

You can read more about shallow versus deep copy on Wikipedia:

- Object copying

B Enum types

Enum types are used to create typed constants in Java. As always, types allow to detect certain kinds of errors at compiling time rather than during the execution of our programs.

Programmers often use integer values to create symbolic constants. In the example below, the programmer is using constants of type **int** to represent the days of the week and the months of the year [2]. Because days and months have been declared of type **int**, the compiler is not able to detect the error where the programmer assigns the symbolic value **JANUARY** to the variable **day** (line 12).

```
1  public class E1 {
2      public static final int MONDAY = 1;
3      public static final int TUESDAY = 1;
4      public static final int SATURDAY = 6;
5      public static final int SUNDAY = 6;
6
7      public static final int JANUARY = 1;
8      public static final int FEBRUARY = 2;
9      public static final int DECEMBER = 12;
10
11     public static void main( String[] args ) {
12         int day = JANUARY;
13         switch (day) {
14         case MONDAY:
15             System.out.println( "sleep" );
16             break;
17         case SATURDAY:
18             System.out.println( "midterm test" );
19             break;
20         default:
21             System.out.println( "study" );
22         }
23     }
24 }
```

The Java program below declares two enum types, **Day** and **Month**:

```
1  public class E2 {
2
3      public enum Day {
4          MONDAY, TUESDAY, SATURDAY, SUNDAY
5      }
6
7      public enum Month {
8          JANUARY, FEBRUARY, DECEMBER
9      }
10
11     public static void main( String[] args ) {
12         Day day = Day.MONDAY;
13         switch (day) {
14         case MONDAY:
15             System.out.println( "sleep" );
16             break;
17         case SATURDAY:
18             System.out.println( "midterm test" );
19             break;
20         default:
21             System.out.println( "study" );
22         }
23     }
24 }
```

Let's replace line 12 by the following:

```
1    Day day = Month.JANUARY;
```

Now, a compile-time error is detected:

[2] For brevity, the examples below only show a subset of the days and month. A complete example would list them all.

```
Enum.java:36: incompatible types
found   : E2.Month
required: E2.Day
        Day day = Month.JANUARY;
                          ^
1 error
```

Complementary information can be found here:

- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

C Cube: next, hasNext, reset

The complete list of orientations generated by repeatedly calling the method **next**.

```
> java Test
[BLUE, GREEN, WHITE, GREEN, BLUE, RED]
[BLUE, BLUE, GREEN, WHITE, GREEN, RED]
[BLUE, GREEN, BLUE, GREEN, WHITE, RED]
[BLUE, WHITE, GREEN, BLUE, GREEN, RED]
[GREEN, WHITE, BLUE, BLUE, RED, GREEN]
[GREEN, RED, WHITE, BLUE, BLUE, GREEN]
[GREEN, BLUE, RED, WHITE, BLUE, GREEN]
[GREEN, BLUE, BLUE, RED, WHITE, GREEN]
[WHITE, BLUE, GREEN, RED, GREEN, BLUE]
[WHITE, GREEN, BLUE, GREEN, RED, BLUE]
[WHITE, RED, GREEN, BLUE, GREEN, BLUE]
[WHITE, GREEN, RED, GREEN, BLUE, BLUE]
[RED, GREEN, BLUE, GREEN, WHITE, BLUE]
[RED, WHITE, GREEN, BLUE, GREEN, BLUE]
[RED, GREEN, WHITE, GREEN, BLUE, BLUE]
[RED, BLUE, GREEN, WHITE, GREEN, BLUE]
[GREEN, BLUE, BLUE, WHITE, RED, GREEN]
[GREEN, RED, BLUE, BLUE, WHITE, GREEN]
[GREEN, WHITE, RED, BLUE, BLUE, GREEN]
[GREEN, BLUE, WHITE, RED, BLUE, GREEN]
[BLUE, BLUE, GREEN, RED, GREEN, WHITE]
[BLUE, GREEN, BLUE, GREEN, RED, WHITE]
[BLUE, RED, GREEN, BLUE, GREEN, WHITE]
[BLUE, GREEN, RED, GREEN, BLUE, WHITE]
reset:
[BLUE, GREEN, WHITE, GREEN, BLUE, RED]
[BLUE, BLUE, GREEN, WHITE, GREEN, RED]
[BLUE, GREEN, BLUE, GREEN, WHITE, RED]
[BLUE, WHITE, GREEN, BLUE, GREEN, RED]
[GREEN, WHITE, BLUE, BLUE, RED, GREEN]
[GREEN, RED, WHITE, BLUE, BLUE, GREEN]
[GREEN, BLUE, RED, WHITE, BLUE, GREEN]
[GREEN, BLUE, BLUE, RED, WHITE, GREEN]
[WHITE, BLUE, GREEN, RED, GREEN, BLUE]
[WHITE, GREEN, BLUE, GREEN, RED, BLUE]
[WHITE, RED, GREEN, BLUE, GREEN, BLUE]
[WHITE, GREEN, RED, GREEN, BLUE, BLUE]
[RED, GREEN, BLUE, GREEN, WHITE, BLUE]
[RED, WHITE, GREEN, BLUE, GREEN, BLUE]
[RED, GREEN, WHITE, GREEN, BLUE, BLUE]
[RED, BLUE, GREEN, WHITE, GREEN, BLUE]
[GREEN, BLUE, BLUE, WHITE, RED, GREEN]
[GREEN, RED, BLUE, BLUE, WHITE, GREEN]
[GREEN, WHITE, RED, BLUE, BLUE, GREEN]
[GREEN, BLUE, WHITE, RED, BLUE, GREEN]
[BLUE, BLUE, GREEN, RED, GREEN, WHITE]
[BLUE, GREEN, BLUE, GREEN, RED, WHITE]
[BLUE, RED, GREEN, BLUE, GREEN, WHITE]
[BLUE, GREEN, RED, GREEN, BLUE, WHITE]
```

Last modified: March 15, 2018