Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Languages

Spring 2021-2022

Programming Exam 7

author: Merve Asiler

Due date: 3 June 2022, Friday, 23:59

1 Problem Definition

In this exam, you are going to construct a puzzle game. In this game, there will be 3 main classes:

- The `class Puzzle` representing the puzzle board,
- The `class Piece` representing the puzzle pieces, and
- The `class Edge` representing the edges of the puzzle pieces.

2 Classes

2.1 `class Edge`

The problem starts with the implementation of the `Edge` class. `Edge` is an abstract base class which has 4 derivations:

1. `class StraightEdge`
   This derivation is used to represent the edge of a puzzle piece drawn by a straight line, i.e. an edge not having any inward or outward bend. (Fig 1a)
2. `class InwardEdge`
   This derivation is used to represent the edge of a puzzle piece having an inward bend. (Fig 1b)
3. `class OutwardEdge`
   This derivation is used to represent the edge of a puzzle piece having an outward bend. (Fig 1c)
4. `class CompositeEdge` [CANCELLED]
   This derivation is used to represent a composite edge constructed by splicing edges of any type from end to end. See Figure 2. Objects of this type are constructed in the same way of the other derivations of `Edge` classes. On the other hand, its constituent edges are added via `CompositeEdge* addEdge(Edge* edge)` method of this class one by one. According to their order of being added, each constituent edge is stored in a `vector<Edge*>` variable.

(a) A puzzle piece having all `StraightEdge`s.

(b) A puzzle piece having an `InwardEdge` on its right side.

(c) A puzzle piece having an `OutwardEdge` on its right side.

Figure 1: Illustration of different edge types on puzzle pieces.

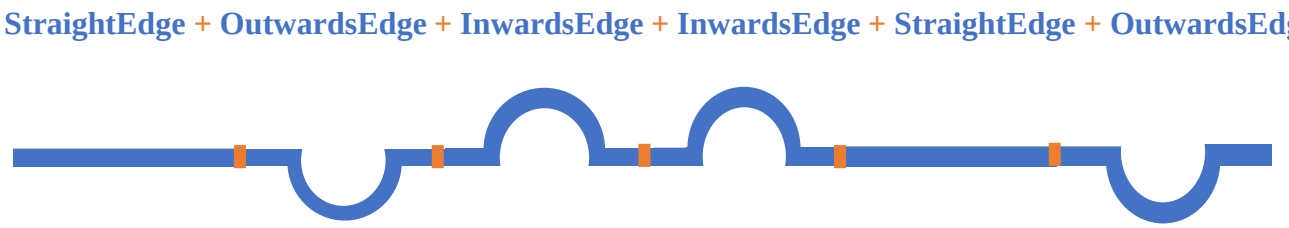

Figure 2: A sample `CompositeEdge`.

Each class object will be constructed with an `int id`. The class owns more than one method, yet from those only the `virtual bool matchWith(Edge& edge)` method will be graded. Still, you have to implement all methods since they will be needed as a helper method for `virtual bool matchWith(Edge& edge)` or for other class' methods. In the pdf, explanation of `virtual bool matchWith(Edge& edge)` method is given only. The other methods of this class are trivial and explained in the template files.

- `virtual bool matchWith(Edge&):`
  This method takes reference to an `Edge` object and tries to match the current edge object with the one given in the argument. It returns true or false according to the following rule:

  1. An edge of the type `StraightEdge` can be matched with only the edges of its own type. In other words, if its `matchWith` method is called with an edge of the type `StraightEdge`, it returns true, otherwise it returns false.
  2. An edge of the type `InwardsEdge` can be matched with only an edge of the type `OutwardsEdge`. In other words, if its `matchWith` method is called with an edge of the type `OutwardsEdge`, it returns true, otherwise it returns false.
  3. An edge of the type `OutwardsEdge` can be matched with only an edge of the type `InwardsEdge`. In other words, if its `matchWith` method is called with an edge of the type `InwardsEdge`, it returns true, otherwise it returns false.
  4. An edge of the type `CompositeEdge` can be matched with only an edge of the type `CompositeEdge`. **Moreover, numbers of the constituent edges inside the caller and callee `CompositeEdge` object are required to be the same and the ones in each corresponding rank have to be matchable edges.** [`CompositeEdge` **part is CANCELLED**]

2.2 `class Piece`

`Piece` class represents the puzzle pieces with 4 edges. Each piece has almost the same size and equals to a square when the bumps on its edges are ignored. It includes an array with 4 elements storing its edges in the type of `Edge*`, yet each edge may be an object of a different `Edge` derivation. Some of the sample pieces are illustrated in Figure 3.

This class does not have any special method other than it constructor, destructor, copy constructor and `operator<< ()` methods. The last one is already implemented for you. It is constructed by specifying its edges as an argument to its constructor and always in the order of left, bottom, right and top sides.

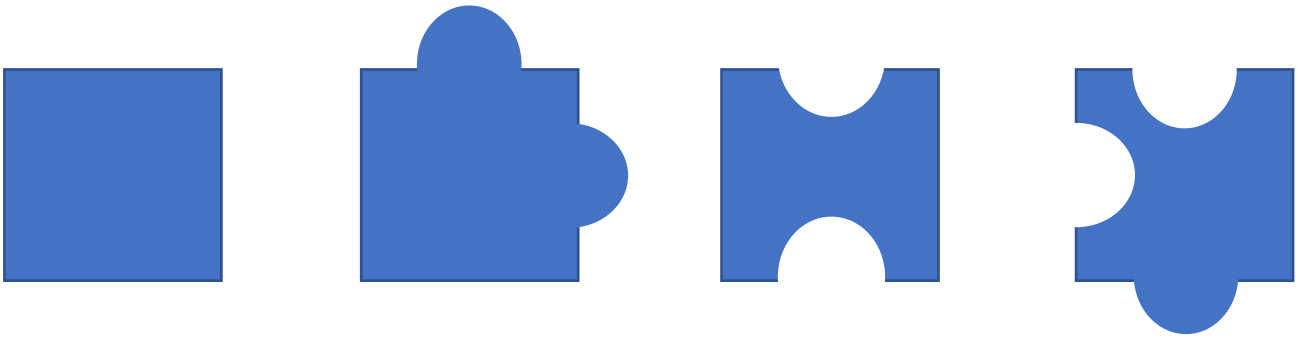

Figure 3: Some of the sample `Piece` objects.

2.3 `class Puzzle`

This class represents a 2D puzzle consisting of 2^N by 2^N pieces. It is asked to implement in the form of a recursive data structure as follows:

```
class Puzzle {
    Puzzle* top_left;
    Puzzle* top_right;
    Puzzle* bottom_left;
    Puzzle* bottom_right;
    int size;
    int first_row, first_column;
    const Piece* piece;

    public:
        ...
}
```

Above, each of the `top_left`, `top_right`, `bottom_left` and `bottom_right` variables represents the one fourth of the puzzle locating on its top left, top right, bottom left and bottom right part, respectively. Note that each of those quarter parts is itself a `Puzzle` consisting of 2^{N-1} by 2^{N-1} pieces.

You can also consider a `Puzzle` object as a 2D array of 2^N x 2^N. In order to identify which sub-puzzle a `Puzzle` object represents, it carries `first_row` and `first_column` values showing the initial array index and `size` value indicating the number of row/column elements, which is 2^N, of the corresponding sub-puzzle array of the size 2^N x 2^N. In the inputs, it is guaranteed that the outermost `Puzzle` object will be initiated with a size of 2^N x 2^N where $N >= 0$. See the Figure 4.

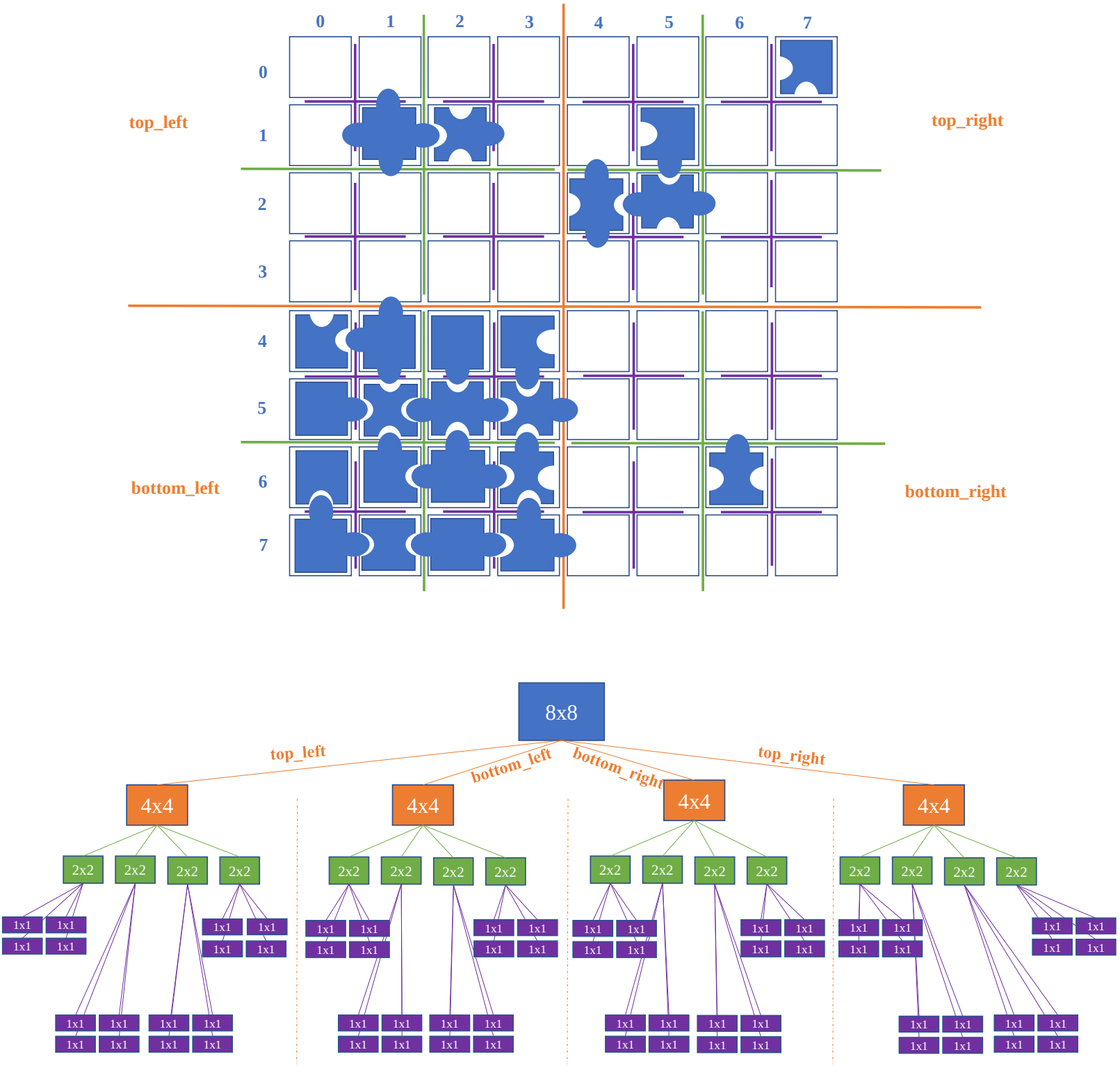

Figure 4: Illustration of a partially filled puzzle visual (above) and the recursive tree structure (below) corresponding to a 8x8 puzzle.

Additionally, all `Puzzle` objects include a `piece` value of type `Piece` pointer, yet only the innermost Puzzle objects have a non-NULL pointer whereas the others have a NULL pointer. Notice that an innermost Puzzle object is the size of the 1 x 1 array, semantically corresponding to a `Piece` object in fact. We call such 1 x 1 puzzles as **cell**.

ATTENTION: For the sub-puzzles of the outermost `Puzzle` object, they should point to `NULL` unless they do not contain a `Piece` object in them or their subparts. In other words, do NOT construct sub-puzzles unless they are required. Otherwise, it may result wrong outputs.

- `void placePiece(const Piece&, int[2]):`
  This method takes a `Piece` object in its first argument and places it into the cell (1 x 1 sub-puzzle) whose index is given as a 2D array in the second argument, representing its $< row\_id, column\_id >$. The details are given in the .cpp file.
- `const Piece& getPiece(int[2]):`
  This method returns the piece in the cell whose indices are given in the argument. If there is no piece in the given location, then the method throws an `EmptyCellRequest` exception. The details are given in the .cpp file.
- `Puzzle crop(int[2], int[2]) const:`
  This method takes the part of the puzzle whose initial and final $< row, column >$ indices are given in the initial and final arguments, respectively. The requested part will be specified as an exact sub-puzzle. The details are given in the .cpp file.
- `void patch(const Puzzle&):` **[CANCELLED]**
  This method pastes (patches) the puzzle given in the argument onto the current puzzle. The patching part corresponds to some sub-space of the current puzzle, yet it may not correspond to an exact sub-puzzle. In other words, it may partially or totally overlap with one or more sub-puzzles. See the Figure 5. The details are given in the .cpp file.

3 Grading

Although only the methods given below will be graded, all the other methods should also be implemented for proper system execution.

- `virtual bool matchWith(Edge&);` ..... 36 points
- `virtual void breakMatch();` ..... 13 points
- `void placePiece(const Piece&, int[2]);` ..... 16 points
- `const Piece& getPiece(int[2]) const;` ..... 17 points
- `Puzzle crop(int[2], int[2]) const;` ..... 18 points
- Memory Leak ..... 50% penalty over the grade of the corresponding question

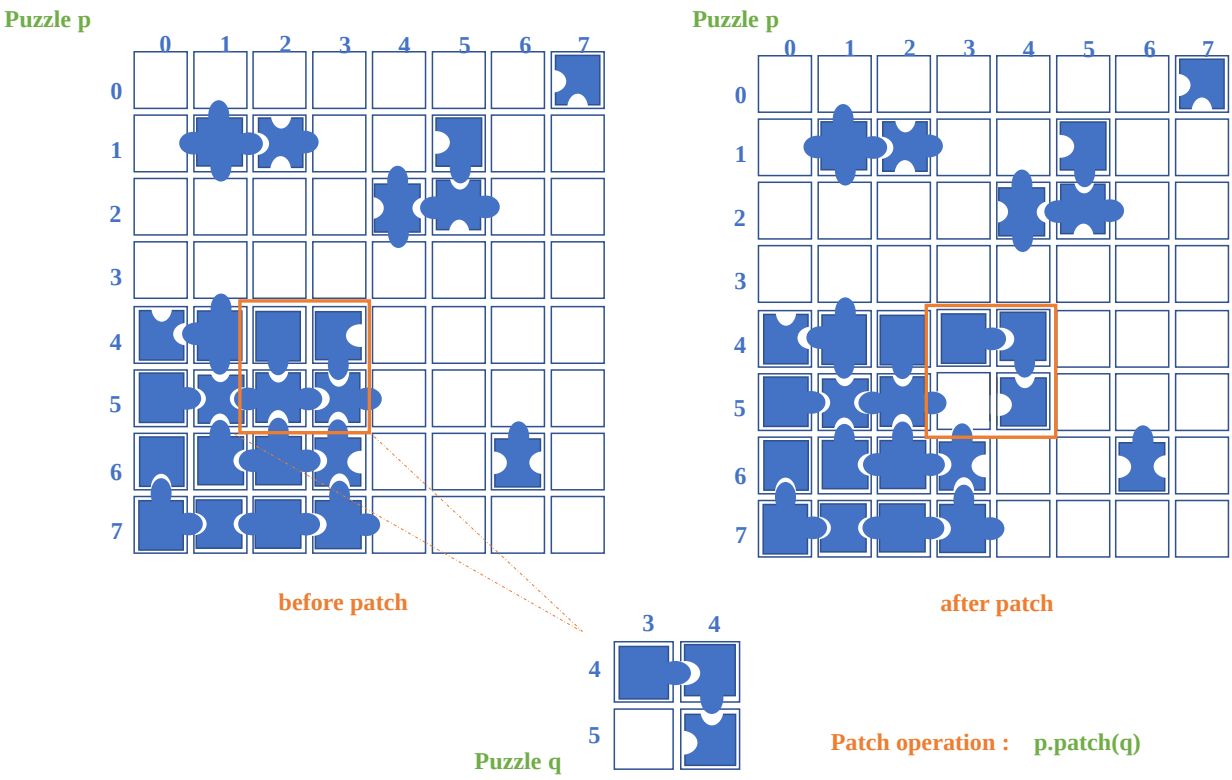

Figure 5: A sample patch operation.

4 Regulations

1. **Implementation and Submission:** The template files "*.h" and "*.cpp" are available in the Virtual Programming Lab (VPL) activity called "PE7" on OdtuClass. At this point, you have two options:

   - You can download the template files, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it. "*.h" and "*.cpp" files are given to you so that you can work on your local machines. If you choose first option, you have to submit these files as well but they will not be included into evaluation process. There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Library:** Using any C++ library other than the already givens is strictly FORBIDDEN.

3. **Friend classes & methods:** Using `friend` keyword is strictly FORBIDDEN.

4. **Programming Language:** You must code your program in C++. Your submission will be compiled with g++ on VPL. You are expected to make sure your code compiles successfully with g++ using the flags -ansi -pedantic.

5. **Cheating:** We have zero tolerance policy for cheating. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

6. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official test cases to determine your actual grade after the deadline.