SCC.111: SLIDING TILE PUZZLE

AIMS

The aim of this task is to get more practice in the use of classes, the Swing package and your first encounter with using inheritance.

Your task this week is to implement a simple, but fully functional swing application - a sliding tile puzzle game, as illustrated below. I'm sure most of you recognise the puzzle.

expand for details

The aim of the game is to allow the player to move tiles left, right, up or down within the puzzle. The challenge being that you can only move tiles into the empty space… Ultimately, a player tries to reform the original image by moving the tiles back, one by one until every tile is in the correct location and puzzle is complete. If you're not sure how the puzzle should work, ask your tutor.

PREPARATION

To create this game, you will need a class to represent the tiles " `TilePiece` ", a class to represent the puzzle itself " `TilePuzzle` ", and a driver class to run the game.

We are generous this week, and have provided you with the TilePiece class and the Driver class. You can get that from the following repo:

```
git clone https://scc-source.lancs.ac.uk/scc.Y1/scc.111/Week18
cd Week18
```

The `TilePiece` class is a simple class that inherites from `JButton` . It has a constructor that takes the image filename as a parameter and the coordinates of the button (x,y). Feel free to check you the code and comments.

Make sure that all the resources and code you create is in this directory.

Now you can use the provided python script to split the infolab image into 12 tiles by running the following:

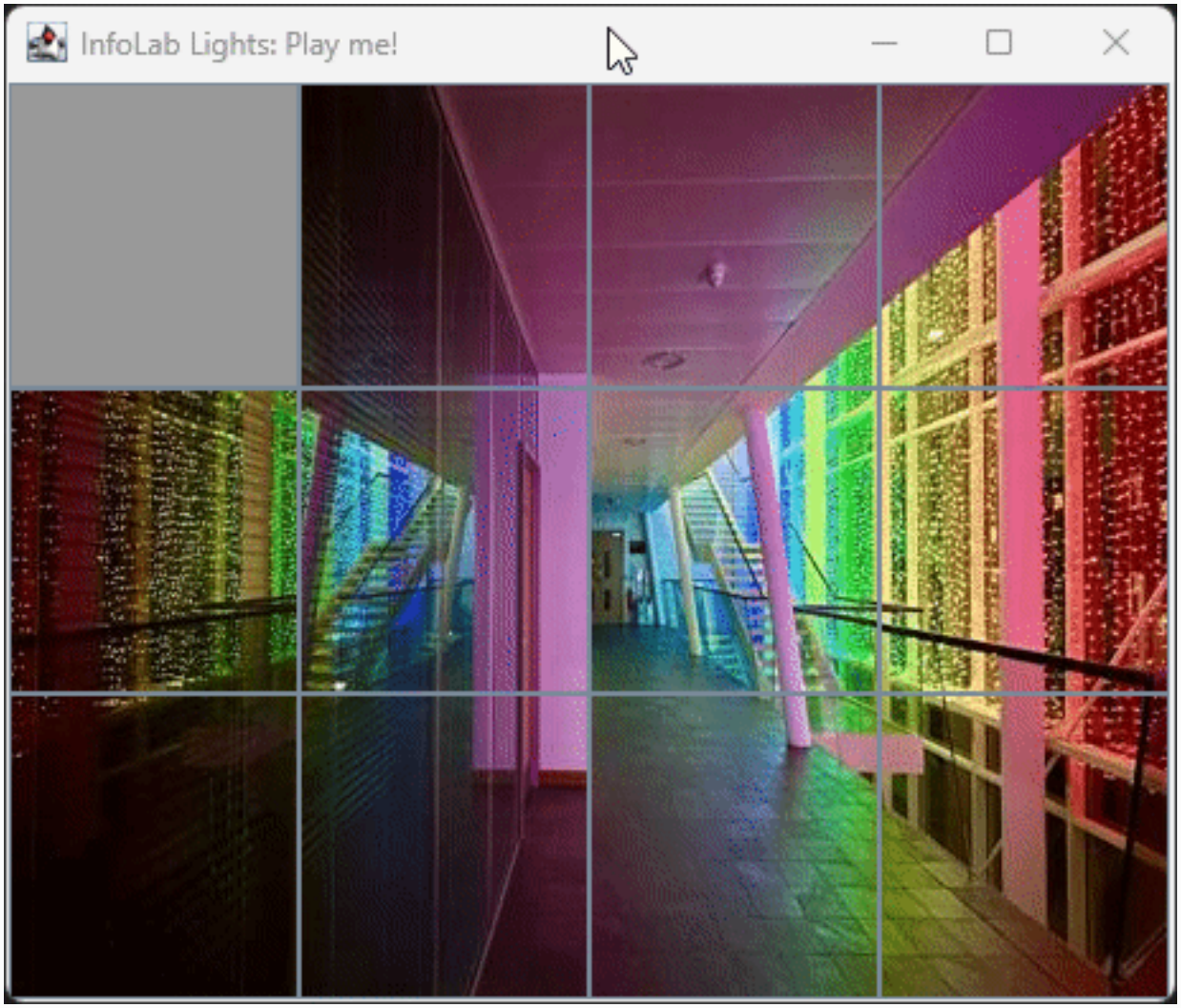


Figure 1: InfoLab Lights

```
python3 img2tiles.py infolab.jpeg .
```

You can use different images if you like. Make sure your respect the copyright of the image. You might need to run `pip3 install Pillow` if you want to run the python script on your machine, or if you get `No module named 'PIL'` error.

TASK 1: CREATE A TILEPUZZLE CLASS

To get started need a GUI to represent the puzzle and hold the `TilePiece` s. First:

- ☐ Create a java class to represent your GUI (let's call it TilePuzzle). DO NOT put a main method in your `TilePuzzle` class.
- ☐ Create an attribute called tiles, it should represent an array of 12 TilePieces.
- ☐ Create a blankTile attribute, which should represent an `TilePiece` that uses the blank image provided in the GitLab repo.
- ☐ Create a constructor for your `TilePuzzle` class, and in it write code to create a GUI like the one shown above. HINT: This is much easier than it sounds. This can be achieved in about 8 lines of code in your constructor. Think about what class the tiles could be represented by and which `LayoutManager` you want to use.

Checkout lecture 31 slides for a refresher.

- In the constructor, initialise each `TilePiece` object with the appropriate image filename and coordinates. Add each `TilePiece` to the panel and link it to the `ActionListener` using `tile.addActionListener(this)` . Don't forget to assign the blank image to the blankTile. *TIPS: you can use two nested for loops to create the tiles.*
- Set the `JFrame` size to `frame.setSize(475,400);`

TASK 2: IMPLEMENT THE ACTIONPERFORMED FUNCTION

Now we need to add some actual functionality behind the GUI. Let's develop your code so that when we click on a tile, it will appear to swap places with the blank square. Swapping a tile with a blank tile is nowhere near as difficult as it sounds. Remember that we can't move buttons around the GUI, as Java

controls the location of the buttons via Layout Managers. However, we can make it LOOK LIKE the tiles have changed place by swapping the images displayed by the buttons. This way the buttons themselves never need to change place…

- The `TilePiece` class has a method `public void exchangeImageWith(TilePiece p)`. This method takes a `TilePiece` as a parameter, and swap the image shown on the `TilePiece` the method is called on with the one provided.
- Write an actionListener in your `TilePuzzle` class that can detect when any of your `TilePiece` s are clicked on with the mouse. Write code so that when a tile is clicked, your swap method is used to make that tile swap with the `blankTile`. TIP: you start by the following:

```
public void actionPerformed(ActionEvent e)
    {
        TilePiece t = (TilePiece) e.getSource();    // Get an object reference to the
↪   tile which has been pressed

        // your gameplay code goes here

    }
```

TIP: refer to slides 16-19 for a refresher on how to use *action listeners*.

VALIDATION

Notice you can swap any tile with the blank tile, regardless of where that tile is on the board. This is not the proper behaviour for a sliding tile puzzle!

- Update your code so that your swap method only swaps tiles that are adjacent to one another (i.e. directly above/below/left or right). You should not permit diagonal movement.
- The `TilePiece` class has a method `public boolean isAdjacentTo(TilePiece p)` that check if the `TilePiece` is adjacent to another `TilePiece`. You can use it for this part.

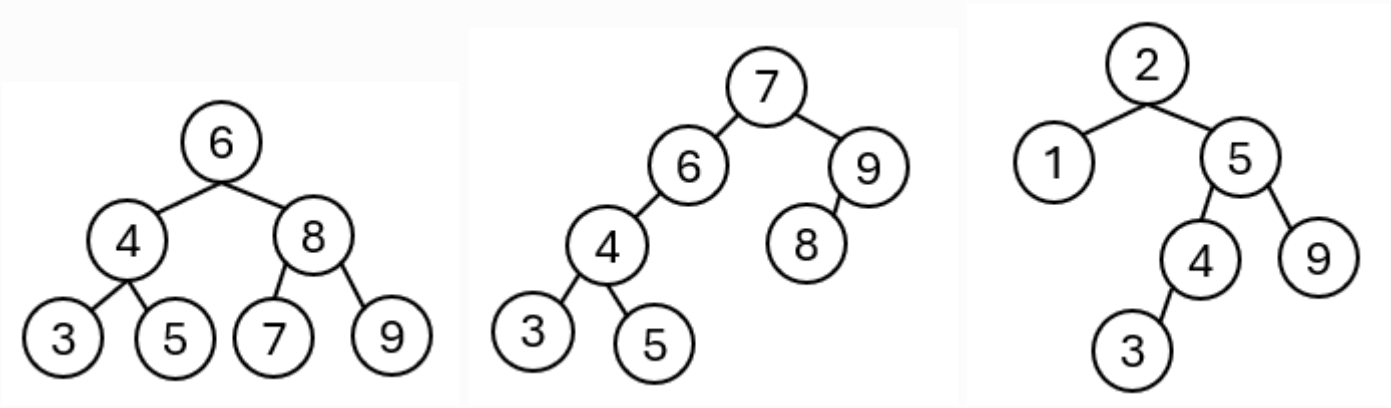
TEST YOUR GAME

Use the Driver class to test your game.

```
//Compile your code
javac *.java
//Run your code
java Driver
```

SCC.121: UNDERSTANDING BINARY SEARCH TREES

In the lectures, we gave a basic implementation of BSTs. Below, you will find some questions that will help you better understand BSTs, and also how to apply your BST data structure for other problems.



1) For the 3 trees (BSTs in fact) above, what are their preorder, inorder, postorder and level order traversals?
2) Implement the binary search tree (with no self-balancing) in Java.
3) Treesort:
   - Implement treesort, that is, use the binary search tree class to sort a series of integer (or string) entries. (Hint: an integer entry is the key for a key-value pair inserted in the BST.)
   - Is the treesort algorithm different from quicksort, and if yes, how?
4) Use the binary search tree to implement a priority queue data structure. More concretely, your data structure must implement the following operations:
   - **boolean isEmpty()**: whether queue has no elements,
   - **void insert_with_priority(Key):** insert an element in the queue with a given priority,
   - **Value pull_highest_priority_element()**: remove the element with the highest priority in a queue and removes it from the queue,
   - **Value peek()**: returns the value with highest priority in queue.

5) By adding a variable to count the number of nodes in every node's subtree, implement a range(Key low, Key high) to return the values whose associated keys are in that low to high range. You can also implement a getMiddle() operation that returns the value whose key is the middle key in the BST.

SCC.131: MULTIPLICATION GAME WITH ASSEMBLY

GOALS

This week, we will carry on our task of implementing a multiplication game with the micro:bit device. The main goal of this exercise is to implement a complex algorithm in ARM assembly. We will also practice how to call assembly functions from C++. If you have completed the task, please have a chat with us to provide feedback on the exercise. You can also use this session to catchup on your previous with ARM assembly.

The plan

For this exercise, you will implement a simple multiplication game with your micro:bit device. The main goal of this exercise is to implement a complex algorithm in ARM assembly. We will also practice how to call assembly functions from C++.

You will initially need to implement a multiplication algorithm in Assembly. For the first step, you will use the CPUlator platform to validate code correctness. After that, you will need to code in C++ a minimal micro:bit interface that will allow users to generate random numbers and then multiply them. The multiplication logic will be realized in assembly, using an algorithm called **Ethiopian Multiplication** which uses shift operations and additions to multiply any two positive numbers.

TASK 1: ETHIOPIAN MULTIPLICATION IN ASSEMBLY

The Ethiopian multiplication algorithm is an ancient method of multiplying two numbers using a process that breaks down multiplication into a series of additions and bit shift operations. It's based on the principle that multiplying by 2 (doubling) and adding numbers can achieve the same result as traditional multiplication. This algorithm is particularly interesting for assembly because it leverages basic operations such as addition, subtraction, bit shifting, and conditional branching, which are well-suited to low-level hardware manipulation.

The multiplication algorithm involves the following steps:

1. **Write down the two numbers to be multiplied** at the top of two columns.
2. **Halve the number in the first column and double the number in the second column**, writing the results below the original numbers.
3. **Continue doubling and halving** each subsequent number, writing the results below the previous ones, until the number in the first column is reduced to 1.
4. **Strike out** every row where the number in the first column is even.
5. **Sum up** all the numbers that are left in the second column. This sum is the result of the multiplication.

For example, let's multiply the numbers 17 and 34:

A	B
17	34
8	~~68~~
4	~~136~~
2	~~272~~
1	544

$$17x34 = 34 + 544 = 578$$

You can also view a nice video from the BBC explaining the algorithm.

In order to help you better understand the algorithmic part of this exercise, we provide below a C implementation of the algorithm.

```c
#include <stdio.h>

// Function to perform Ethiopian Multiplication
int ethiopian_mult(int a, int b) {
    int result = 0; // Initialize result to 0

    // The while loop stops when a == 0.
    while (a >= 1) {
```

```
        if (a % 2 != 0) { // If 'a' is odd, add 'b' to the result
            result += b;
        }
        a = a >> 1; // Halve 'a'
        b = b << 1; // Double 'b'
    }

    return result; // Return the multiplication result
}
```

Your task at this stage is to implement the Ethiopian multiplication algorithm in ARM assembly. We strongly recommend you implement your code in the form of an ARM function that accepts a single input argument, which is the address of an integer array. You can hardcode an integer array for this step in the .data section of your program and use some hard-coded values. **Please revise the conventions for Assembly function implementation and make sure your function follows these conventions.** Below you can find a list of key Assembly Instructions and Concepts:

- `mov` : To load values into registers.
- `lsl` and `lsr` : Logical shift left and right instructions, useful for doubling and halving numbers.
- `add` : To sum up values.
- `cmp` and `beq` : Compare instructions and branch if equal, useful for checking if a number is even (test if the least significant bit is 0) and skipping the addition step.
- `b` : Unconditional branch instruction for loops.

A good template to start your code is the following:

```
@ source/eth_mult.s
    .syntax unified
    .data
    val: .word 14, 16

    .text
```

```
_start:
    @ TODO: Load the address of the array into register r0

    @ Call the function eth_mult
    bl eth_mult
    @ Make the function loop indefinitely
    b _start

    .global eth_mult
eth_mult:
    @ TODO: Implement your Ethiopian multiplication algorithm here
    bx lr
```

TASK 2: CODAL SETUP

The CoDAL build environment and the GCC compiler provide out-of-the-box support to build and link ARM assembly with C/C++ code. To start the integration job, you must start a new CoDAL project by re-downloading the git repo. Below you can find some command line commands to clone the CoDAL example repo to the folder `~/h-drive/microbit-asm/` (You can safely store the project in any location on your computer).

```
cd ~/h-drive/
git clone http://scc-source.lancs.ac.uk/scc.Y1/scc.131/microbit-v2-samples.git
↪  microbit-asm
cd microbit-asm/
```

Start in the background a build process by running the command `python3 build.py` on your command line terminal while you are in the micro:bit folder.

> **Note:** If you want to add support for CoDAL on your personal devices, you can find an install guide on the lab moodle page. The guide will help you set up the CoDAL build environment on your personal computer (details for Windows, OSX, and Linux). *We cannot guarantee to support personal computer setups, so please be aware that the lab computers are the only platforms that we will guarantee that things will work.*

Let's now create the template code for our project. We firstly need to modify the `source/main.cpp` file.

```
// source/main.cpp
#include "MicroBit.h"

MicroBit uBit; // The MicroBit object

// --- DECLARE GLOBAL VARIABLES, STRUCTURE(S) AND ADDITIONAL FUNCTIONS (if needed)
↪  HERE ---
int val[2] = {1, 1};

// This links your Assembly function with the CoDAL runtime.
extern "C"
{
    void eth_mult(int val[]);
}

// Event handler for buttons A and B pressed together
void onButtonAB(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

// Event handler for button A
void onButtonA(MicroBitEvent e)
```

```
{
    // DEVELOP CODE HERE
}

// Event handler for button B
void onButtonB(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

int main()
{
    // Initialise the micro:bit
    uBit.init();
    // Initialize the random number generator
    uBit.seedRandom();

    // Ensure that different levels of brightness can be displayed
    uBit.display.setDisplayMode(DISPLAY_MODE_GREYSCALE);

    // Set up listeners for button A, B and the combination A and B.
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_A, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonA);
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_B, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonB);
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_AB, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonAB);

    // You probably do not need to do anything after this point in the main method.

    // Enter the scheduler indefinitely and, if there are no other processes running,
```

```
    // let the processor go to a power-efficient sleep WITHOUT ceasing execution.
    release_fiber();
}
```

The code above provides the scaffold to manage three events: pressing button A (`onButtonA`), pressing button B (`onButtonB`), and pressing buttons A and B simultaneously (`onButtonAB`). Additionally, it registers your eth_mult function implemented in Assembly, and will execute it every time you press buttons A or A and B simultaneously. You should implement code to add functionality to the three event handlers.

You should also save the assembly code you implemented in step 1 in a new file called `source/eth_mult.s` . The build system will automatically detect and build the new file in the source folder and link the code with the C++ code.

TASK 3: INTERFACE

You should hopefully have now a working implementation of the Ethiopian Multiplication code in Assembly. In this step, you need to implement a basic interface with the micro:bit to generate random numbers and feed them to your Assembly function. Here is a breakdown of the interface that you need to implement:

- Press button A to generate a random number between 1 and 99 for the variable “a” and to show it on the LED display.
- Press button B to generate another random number between 1 and 99 for “b” and to show it on the LED display.
- Press buttons A and B simultaneously on the micro:bit to find out what the product is - that’s what the answer would be if the numbers “a” and “b” were multiplied together. This part of the program should use an Ethiopian multiplication algorithm implemented in assembly.
- You can use this project in a competitive two-player game, where the two random numbers are read out, and each player must shout out the correct answer first to win a point.

You already have a set of 3 event handlers to manage different button press events and a global array variable `val` which can hold up to two elements and which you should use to store your random numbers. Your micro:bit implementation should generate a random number when buttons A or B are pressed, store the result in the first or second array element in variable `val` respectively, and print the

number values using the LED screen.

Random number generation in CoDAL is a **bit** different from what you are used to in C. The runtime provides the following function:

```
int microbit_random(int max);
```

The function will return a random integer number between the values [0, max]. You can use this function to generate a and b values, but you probably want to skip 0 values, as the result will be zero.

For your number displaying functionality, you can use the LED screen and print values using the method:

```
uBit.display.print(int a);
```

TASK 4: CALLING ASSEMBLY FROM C++

In this final step, we will apply your knowledge of memory instructions and functions in assembly, in order to implement the functionality that will pass the variable val into your assembly Ethiopian multiplication function and return the multiplication result to your C++ code. It is worth focusing a bit on the C++ code below:

```
// This links your Assembly function with the CoDAL runtime.
extern "C"
{
    void eth_mult(int val[]);
}
```

This line informs the compiler that a C function is available in another file that implements a function called eth_mult. The function accepts a single array pointer variable as input and will be returning an int variable. We also included the following line in our .S file:

```
.global eth_mult
```

This line tells the assembler that the eth_mult function is a global symbol and can be accessed from other files. The assembler will create a record of this function in the `.o` file and the linker will be able to link

the function with your C++ code.

Another thing to look for in your implementation is how the `val` variable is passed to your Assembly function. The parameter will effectively be a pointer to the address of the first element of the array. The pointer will point to an address in the data section. You can use that value as the second operand in the memory instructions **str** and **ldr**, to store and load data from main memory. The C++ assembly code generated by the compiler will be responsible for loading the address of the array into register `r0` before calling the function. If you implemented your code to read the value from an array in Stage 2, then you shouldn't need to change anything in your Assembly code. In your C++ code, you should be able to call the function by using the following line:

```
eth_mult(val);
```

At this stage, it is good to revise the material about functions in Lecture 6 of the assembly part of the course and figure out how parameters are passed into a variable and how results can be returned from an assembly function.

HACKER EDITION

SCC.111: SLIDINGPUZZLE++

Still looking for more?

Add a scramble button to the game. When pressed, the scramble button should randomly distribute the tiles across the game board, ready for the player to put back together! Take care not to place the tiles in a configuration that is unsolvable though!

Detect when the game is won. If the player manages to put all the tiles in their correct place, display a victory message. This message could simply be a System.out.println message, changing the title of your window, some text displayed in a JLabel, or even in a pop up window!

SCC.121: SPLAY TREES

In this part, your goal is to code the splay tree data structure in Java. Splay trees are BSTs that ensure recently accessed key-value pairs are quick to access again. In other words, they are self-balancing BSTs. However, unlike the 2-3 tree presented in the lecture, splay trees are not height-balanced and thus do not guarantee $O(\log n)$ worst-case time complexity for insertion, deletion and look-up (called find() in the lectures).

Yet, splay trees are popular because they have the following advantages.

- If key-value pairs are accessed randomly, but from a non-uniform random distribution (say, random prime numbers), then splay trees are extremely efficient. (The prime numbers will be contained by the higher nodes, and thus faster to access.) For those who are interested in the specifics, the look-up operation is proportional to the **entropy** of the access pattern.
- Even when key-value pairs are not accessed randomly, it still holds that recently accessed key-value pairs are quick to access again.

In summary, splay trees are quite useful for practical applications (e.g., for implementing caches or garbage collection).

Insert, Delete and Look-up Operations

Splay trees are relatively simple modifications of the BST data structure. More precisely, the insertion, deletion and look-up operations are first executed just as in the BST, and after that a **splaying operation** on some node x takes place. The splaying operation ensures that the key-value pair corresponding to x moves up the tree and is placed at the root.

More precisely:

- *Insertion*: A node x with the input key-value pair is inserted as in the naive BST. Then, do a splay operation on x.
- *Deletion*: The node x corresponding to the input key is deleted as in the naive BST. After which, do a splay operation on the parent node of x.
- *Look-up*: Find the node x associated with the key input as in the naive BST. After which, do a splay operation on x.

Splaying Operation

The splaying operation is done through **tree rotations**. For two nodes x and p, the figure below shows a tree rotation on x to the right: x is moved towards the root, and to the root if p was initially the root. Inverting the rotation (or going opposite of the arrow's direction) gives a tree rotation on p to the left.

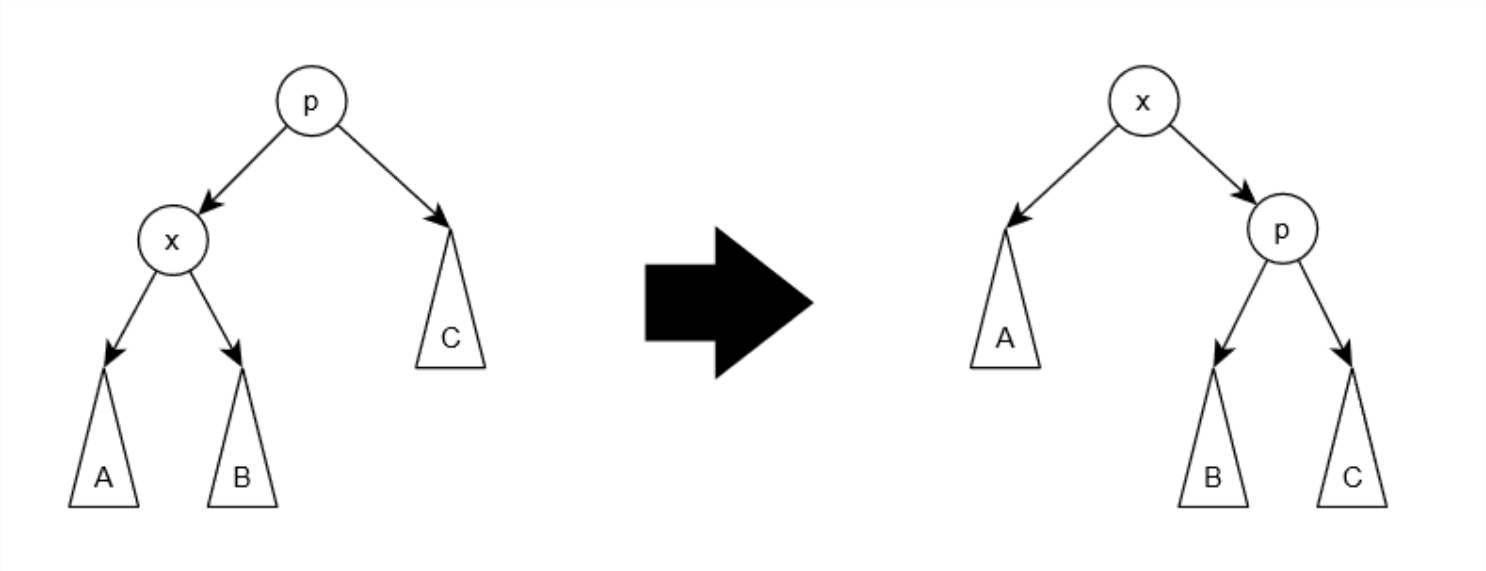
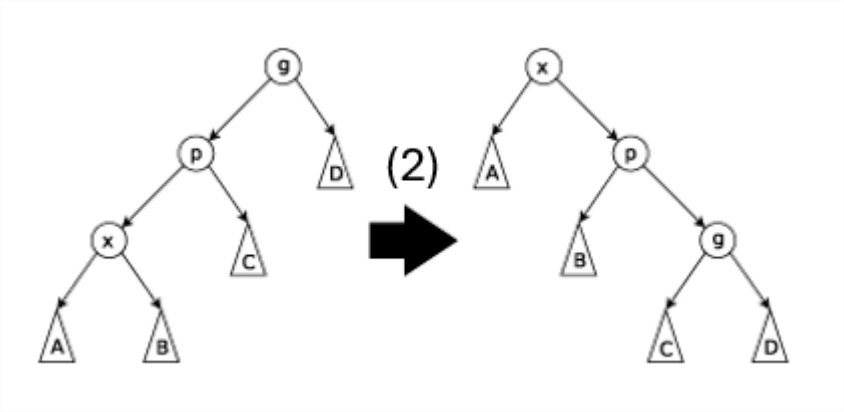
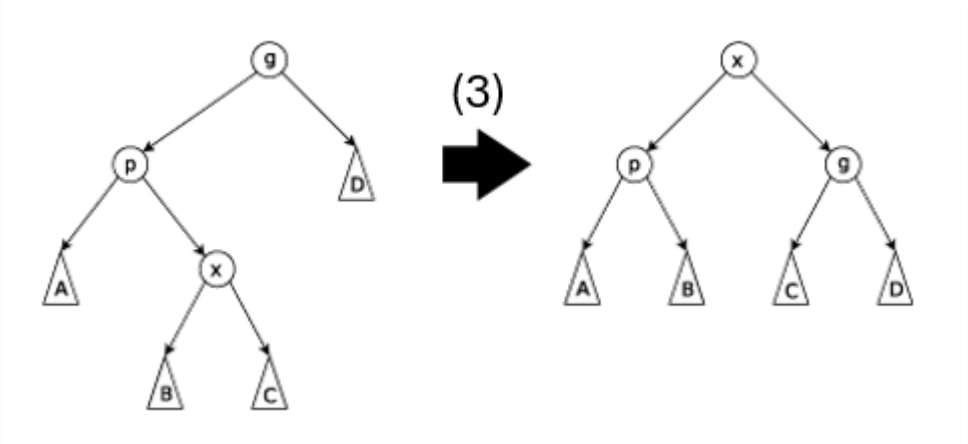


Figure 2: Tree rotation to the right

Three cases can come up in the splaying operation (on some node x):

1. *x is the right (resp., left) child of p, and p is the root*: Then do a tree rotation on x to the right (resp., left).
2. *x is the right (resp., left) child of p and its parent p also the right (resp., left) child of the grandparent g of x*: Do two left (resp., right) rotations, the first on p and the second on x. The result (for two left rotations) is illustrated in the first figure below.
3. *x is the right (resp., left) child of p and its parent p the left (resp., right) child of the grandparent g of x*: Do one left rotation on x, followed by one right rotation on x (resp., right rotation on x followed by left rotation on x). The result (for left then right rotation) is illustrated in the second figure below.


g
p
D
x
C
A
B
(2)
x
A
p
B
g
C
D


g
p
D
A
x
B
C
(3)
x
p
g
A
B
C
D

SCC.131: MULTIPLICATION GAME WITH ASSEMBLY

Task 5: Negative number support

The algorithm you have implemented for your Ethiopian multiplication can perform operations with positive numbers. Can you extend your code to support negative numbers?

> **Hint**: A good way to approach this task is to convert negative numbers to positive using the two's complement conversion algorithm and perform the multiplication. Don't forget though to convert your result to a negative value, if exactly one of the input numbers was negative.

Task 6: Inline Assembly

Your project implements your Assembly as a function, which is invoked by the C++ runtime. In lecture 9 we discussed the mechanism of inline assembly. Can you convert your function invocation into handwritten inline Assembly?