DATA MANAGEMENT

Engineering in Computer Science

ACADEMIC YEAR

2023/2024

External Multi-Pass Sorting Visualizer

GitHub

JACOPO FABI

1809860

ANNA CARINI

1771784

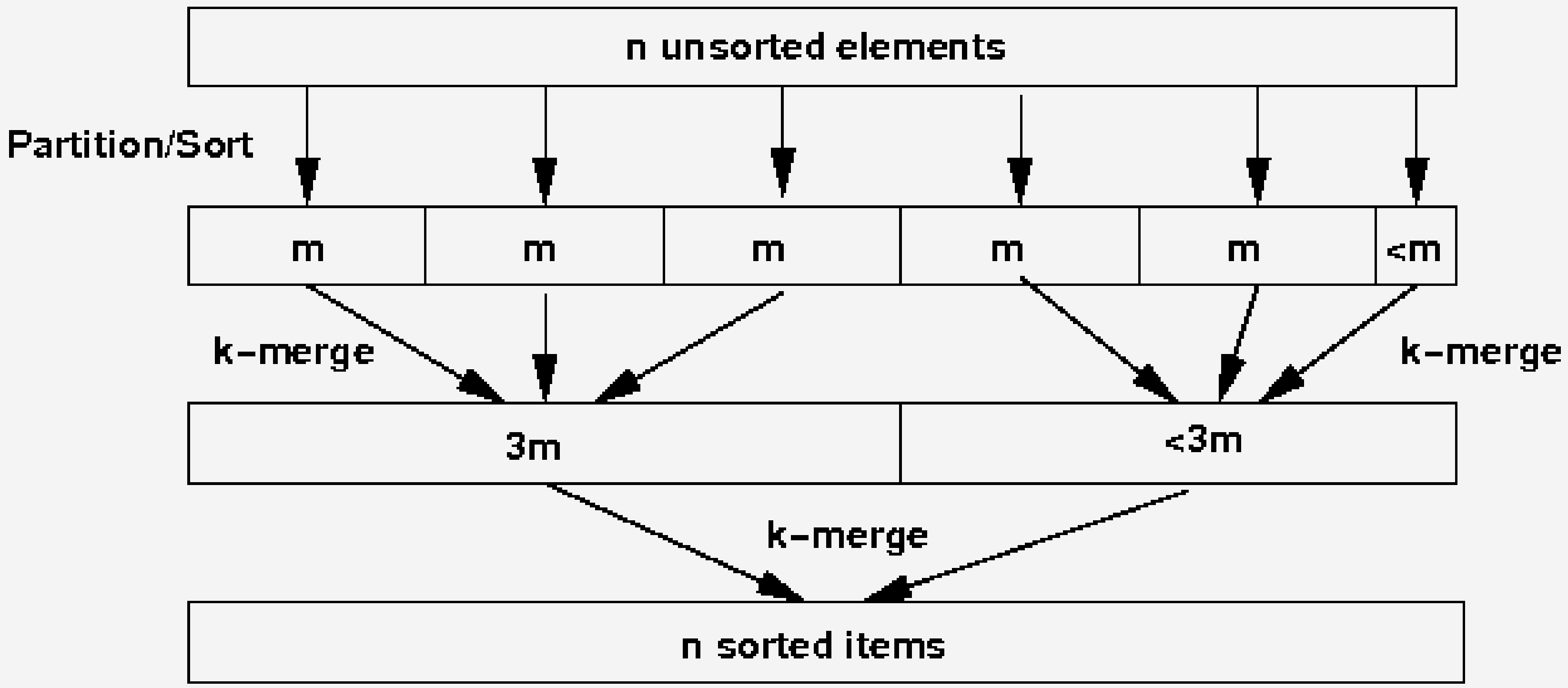
The algorithm

Reason

External sorting is a class of algorithms used when the data to be sorted does not fit into main memory.

Parameters

- M = number of frames in the buffer
- B(R) = number of pages in secondary memory that need to be sorted



Algorithm's phases:

1. Sorting
The pages are divided into chunks, each of at most M pages. Each chunk is loaded in the buffer, sorted, and written back to secondary memory.

2. Merging
The sorted chunks are sort-merged in a series of passes. At each pass, M-1 chunks are merged by loading them one page at a time in the buffer, and using the remaining frame to write the output.

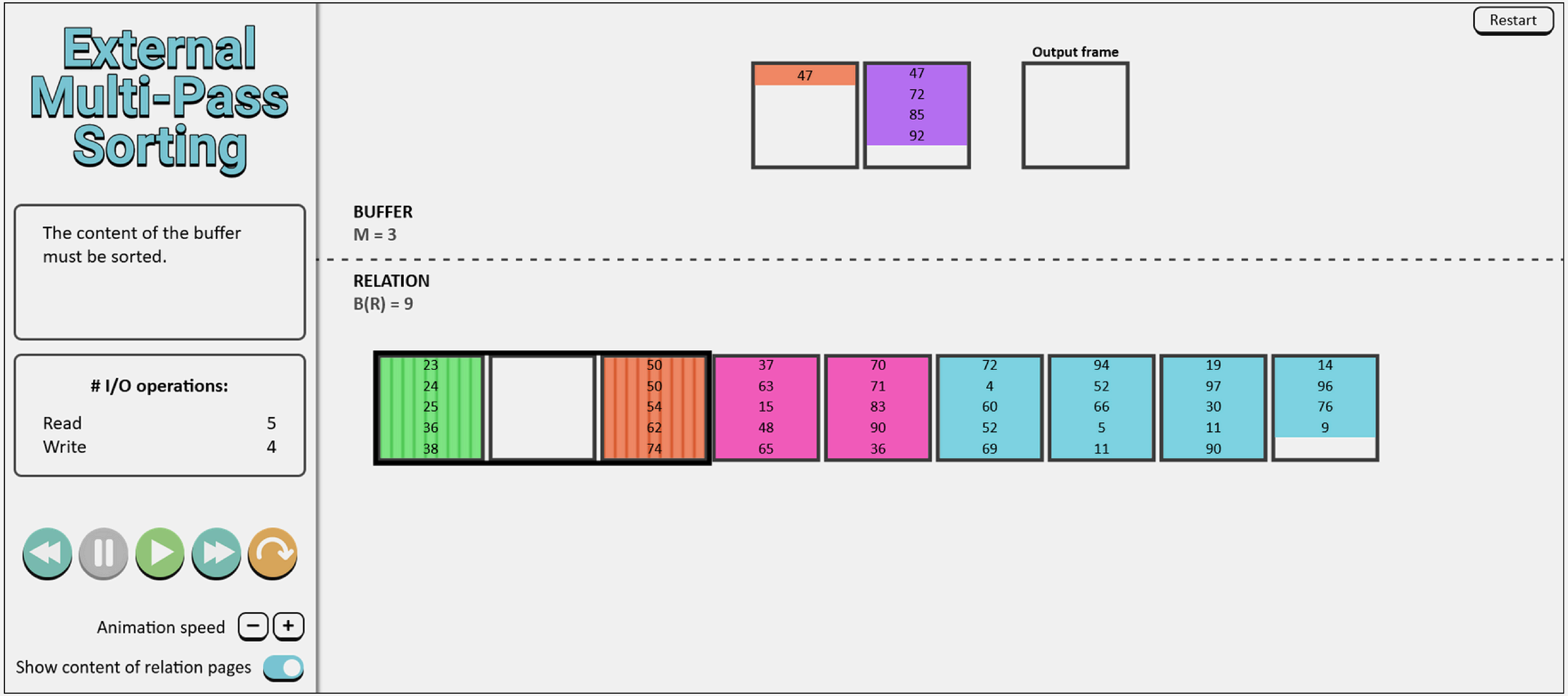
Tools

Two.js | Javascript library that provides a two-dimensional drawing API, specialized for vector graphics. Provides an easy interface to draw basic shapes, and functions to translate and scale the elements.

Tween.js | Javascript animation engine that provides a simple tween function, to smoothly interpolate values, and generate an animation.

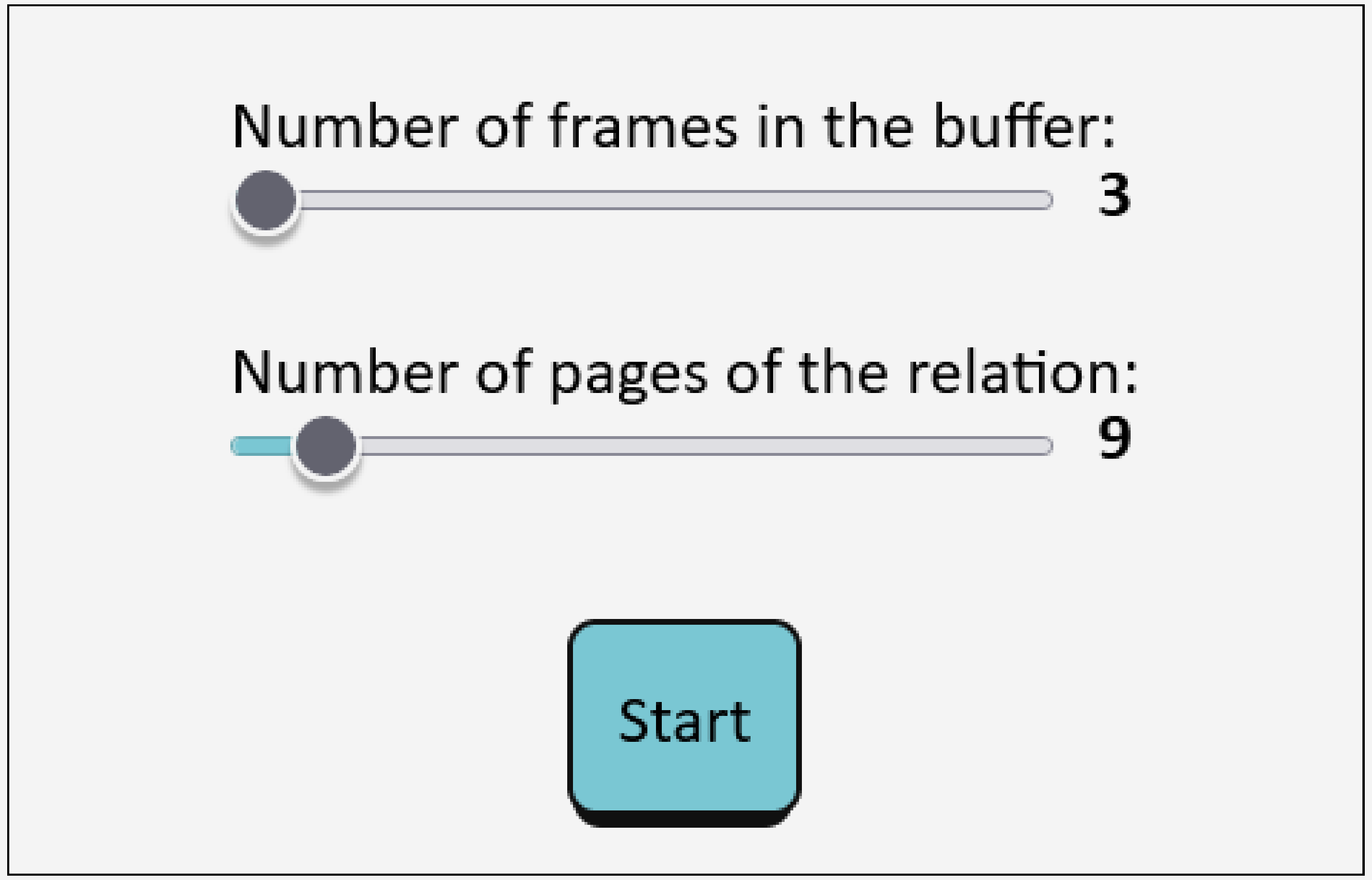
Github Pages | Free web hosting service provided by GitHub. It allows users to publish static web applications directly from a GitHub repository.

The application



Functionalities

1 MENU



Through the menu the user can decide the **size of the buffer** (how many frames, including the output frame) and the **size of the relation** (how many pages).

Functionalities

2 CONTROLS

Play	Spacebar
Pause	Spacebar
Next	Enter
Jump	Right arrow key
Undo	Left arrow key

Next
Executes a single step of the algorithm, showing the animation.

Jump
Executes a single step skipping the animation.

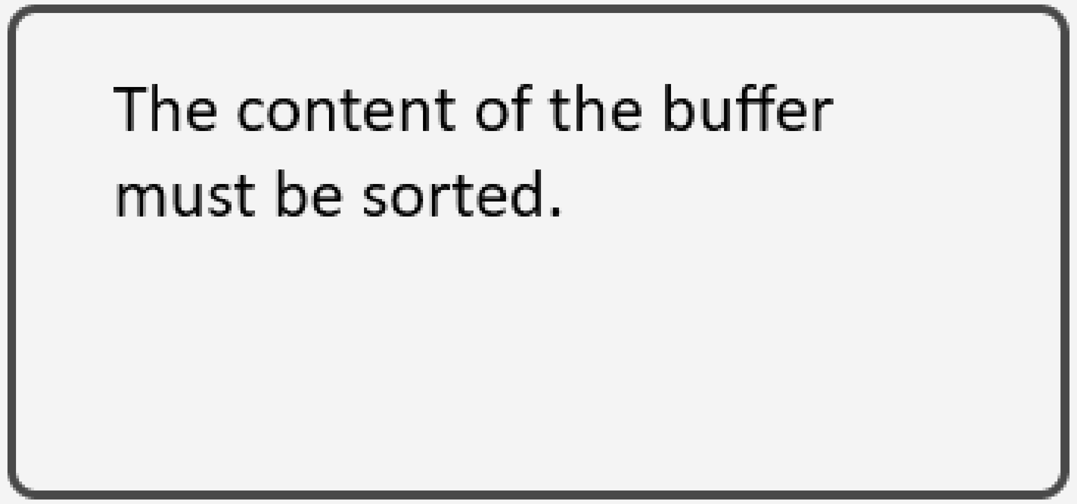
Undo
Undoes the last step that has been executed, restoring the previous states of the relation and buffer.

Play
Starts playing the algorithm automatically, step by step.

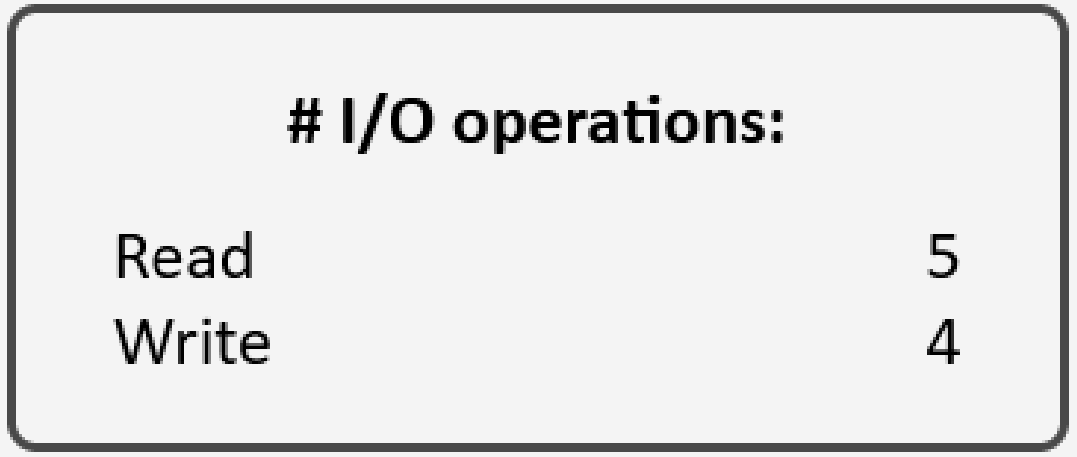
Pause
If the algorithm is being played automatically, it pauses it.

Functionalities

3 INFORMATION BOXES



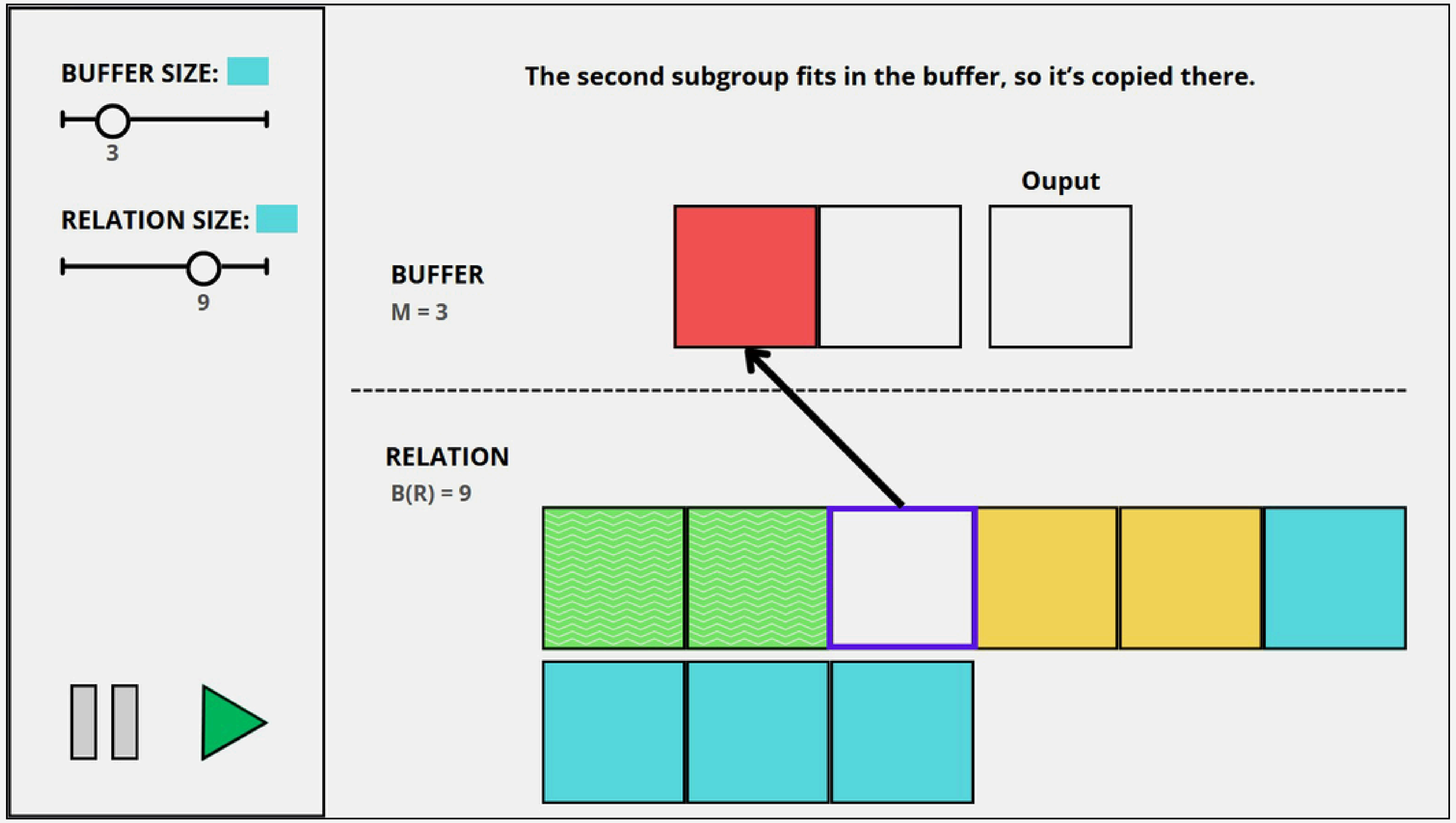
This box gives information about the **current state** of the application. It is updated every time a step is executed, to explain to the user what is happening. It is hidden during the automatic play.



This box shows the **amount of read and write** operations that have been done up to the current step of the execution. It's useful to understand the cost of the algorithm.

Graphical choices

- When the relation is divided into sub-groups, each sub-group has a **different color** to differentiate them.
- The group that is currently taken into consideration is highlighted by **framing it with a black box**.
- To show which sub-groups have been sorted we applied a **texture** on their pages.



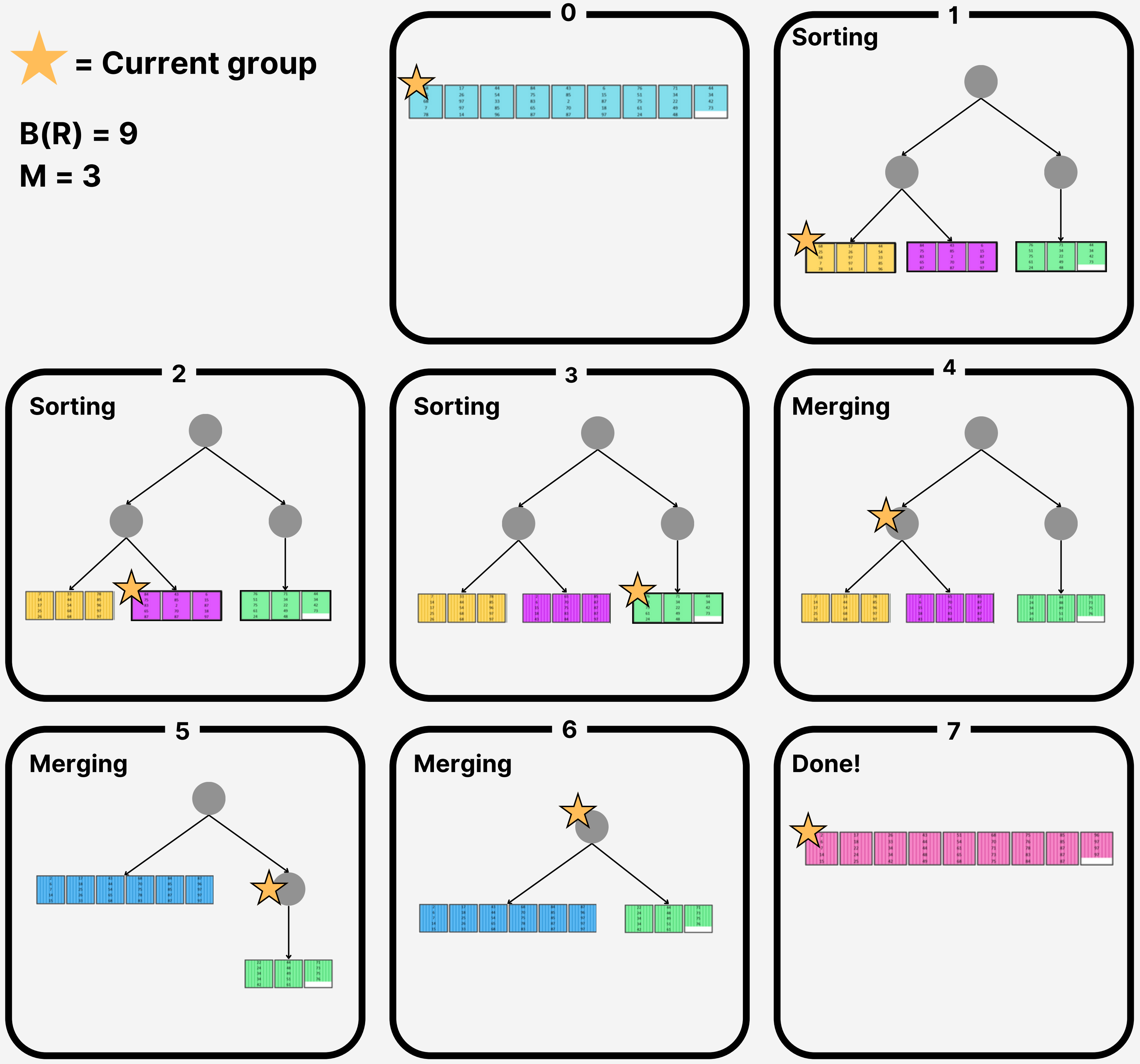
Mock Up

Implementation

1 TREE STRUCTURE

The **relation** is a tree

State 0: The current group is the root, the whole relation.
State 1: The tree structure is created.
States 2 - 3 - 4: The sub-runs are sorted.
States 5 - 6: The sorted sub-runs are merged, the tree's depth decreases.
State 7: All the runs have been merged together, the whole relation is sorted.



Implementation

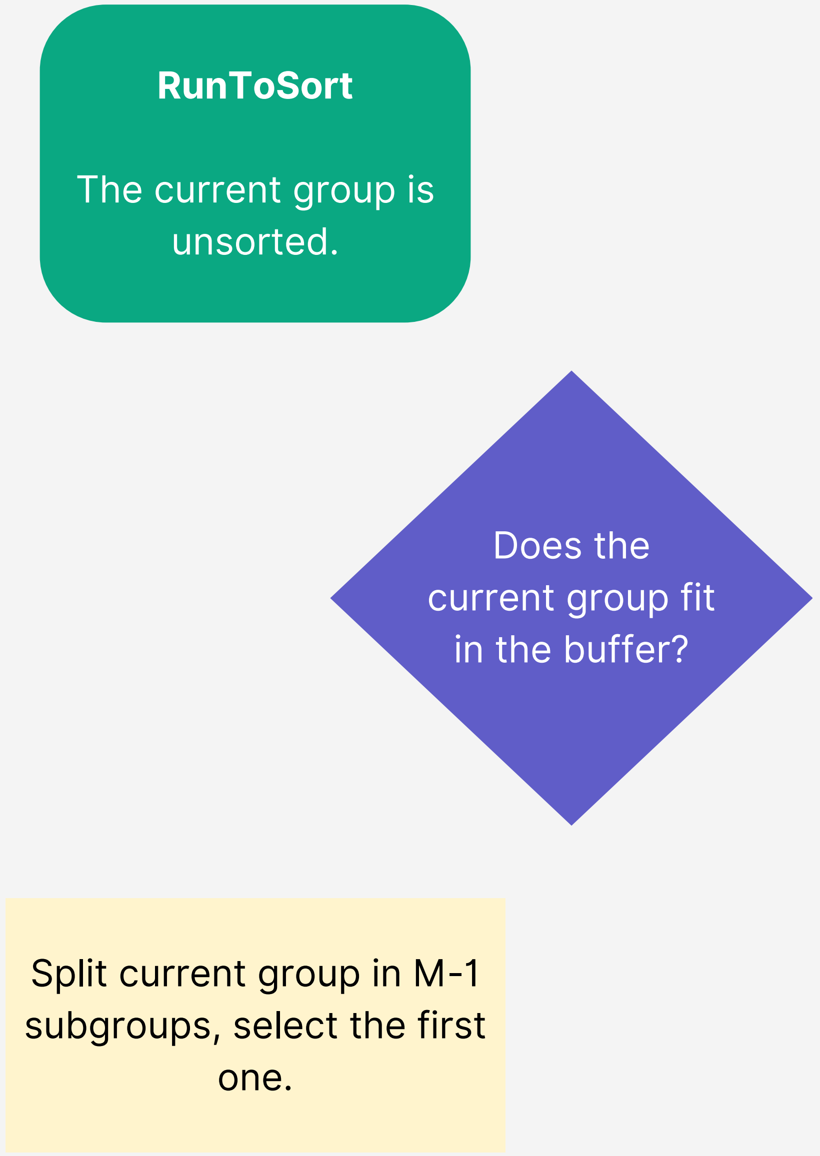
2 APPLICATION STATES

When designing the application, we defined some **states**: points in which the execution of the algorithm can be interrupted.

We define how and under which conditions the application should pass from one state to another through the "Flow Chart".

- The **green squares** with rounded corners represent the states of the application.
- The **purple rhombuses** represent checks that the application does to know which state it should transition to.
- The **yellow squares** explain the side-effects associated to the transitions.

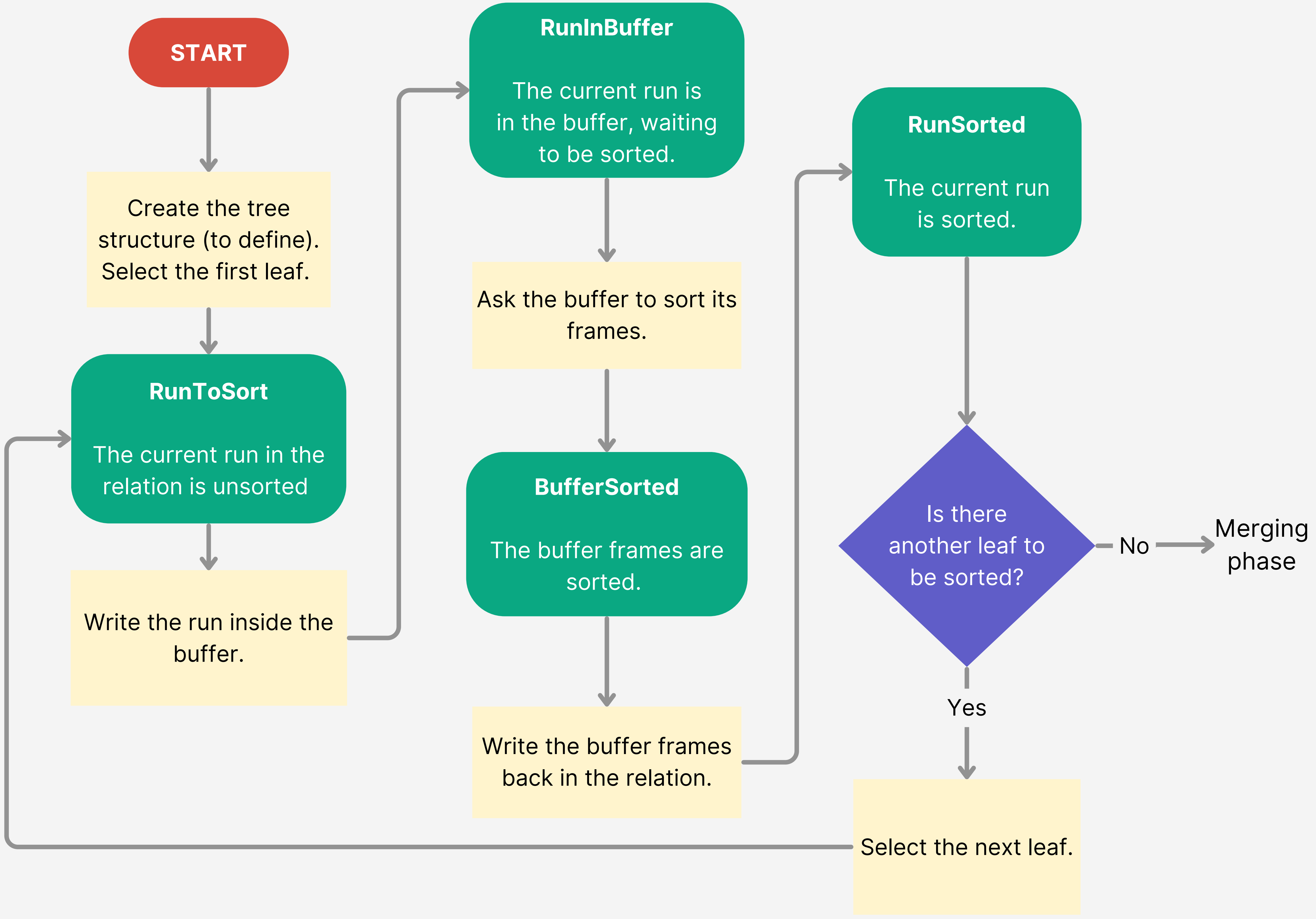
Whenever a user presses the "Next" button, the application does the necessary checks and actions specified by the flow chart and then moves to a new state, showing an animation.



Implementation

3 FLOW CHART

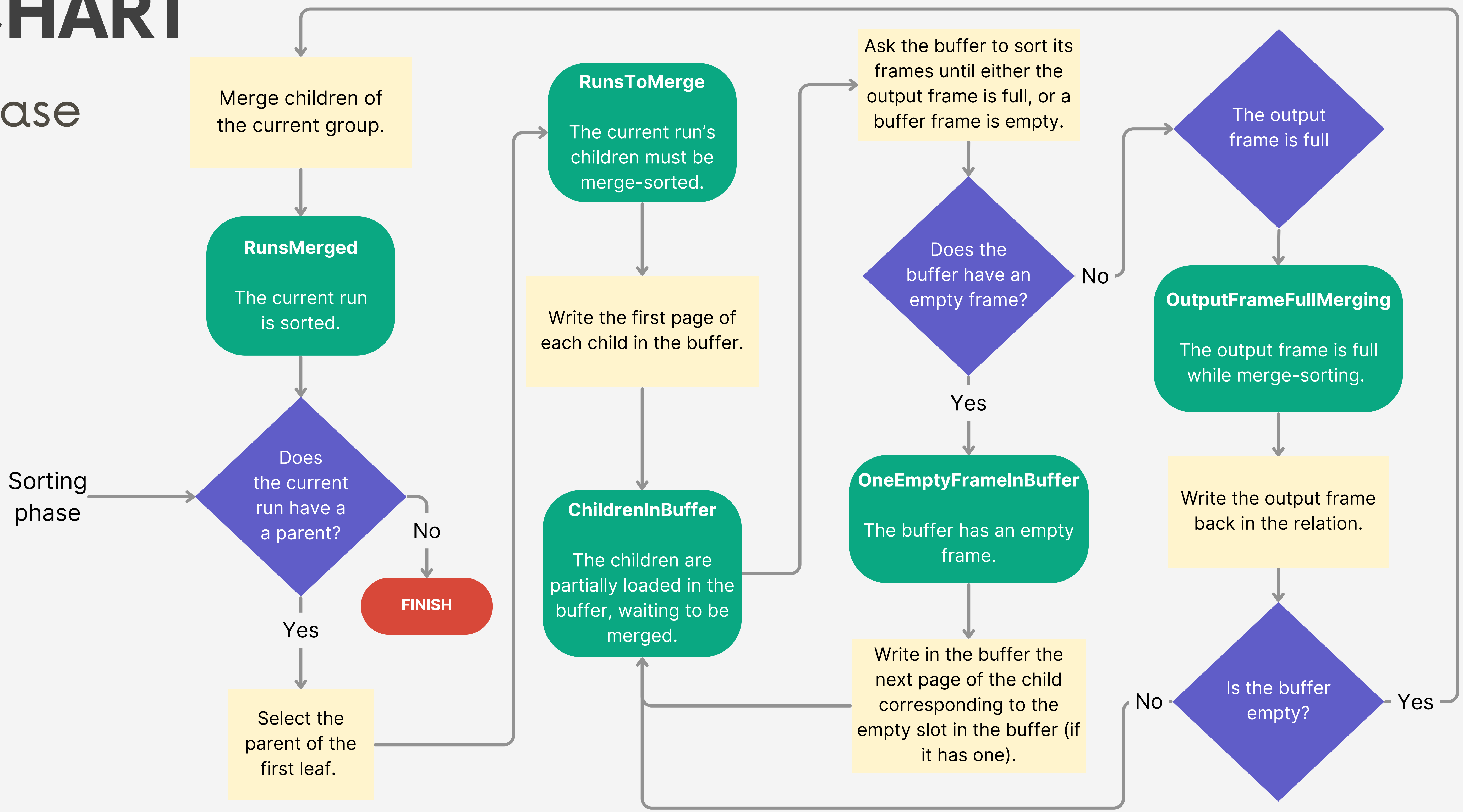
Sorting phase



Implementation

3 FLOW CHART

Merging phase



Implementation

4 UNDO OPERATION

Undo is managed using an **array**. Before executing each state change, a function is pushed onto this array. When an undo is performed, the last function is obtained from the array using pop, and executing this function allows the system to return to the previous state.

This function is different for each state change: in fact, it is not necessary to save the entire state to restore it. It is sufficient to define a function that does the exact **opposite** of what was done to change the state, which most of the time requires only a few data as input.

THANKS!