Artifact for Concurrent Size

This project contains the artifact for the paper "Concurrent Size" by Gal Sela and Erez Petrank, OOPSLA 2022 (https://doi.org/10.1145/3563300).

The artifact supports all the claims that appear in the evaluation section in the paper: it produces the data for all the graphs in the paper as well as the graphs themselves.

Artifact directory structure

```
 README.{md,pdf}      - this documentation file
run_measurements_and_produce_graphs.sh - the main measurement execution file
pc_graphs.tex        - a latex file to present the graphs in a pdf file after producing all
                       graphs by running run_measurements_and_produce_graphs.sh with
                       multicoreServer=F on a PC
server_graphs.tex    - similar to pc_graphs.tex, but for a run with multicoreServer=T
                       on a mulicore server
run_tests.sh         - a test execution file
compile.sh           - compiles the Java code
LICENSE              - the license for this project: GNU General Public License v3.0
algorithms/          - the concurrent data structures
└──baseline/         - the baseline skip list, hash table and BST data structures,
|                      which do not support a size method.
|                      (We also include BSTTransformedToRemoveLpAtMark for reference
|                      though it does not appear in the measurements. This is a variant
|                      of the BST on which we build to apply our size methodology, in
|                      which the linearization point of a delete is at the marking step.)
└──size/             - data structures transformed according to our methodology
|  └──core/          - the implementation of our methodology, where the main file is
|                      SizeCalculator.java
└──vcas/             - a BST with a snapshot mechanism, based on the paper "Constant-Time
|                      Snapshots with Applications to Concurrent Data Structures"
└──iterator/         - a skip list with a snapshot mechanism, based on the paper
|                      "Lock-Free Data-Structure Iterators"
measurements/        - code for running measurements and producing graphs
└──Main.java         - the main file for running both measurements and tests
└──support/          - auxiliary java code
└──adapters/         - classes that adapt the data structures' methods to a unified
|                      interface used in measurements
└──python_scripts/   - auxiliary python code
```

Getting started

0. Prerequisite installations

Java

OpenJDK may be installed as follows -

- On Linux:

  ```
  sudo apt-get install openjdk-17-jdk openjdk-17-jre -y
  ```

- On Mac:

  ```
  curl -s "https://get.sdkman.io" | bash
  # then, in a new shell:
  sdk install java
  ```

Python3

Python3 may be installed by downloading the latest python version from https://www.python.org/downloads/ and installing it.

In addition, the numpy and matplotlib libraries are required. They may be installed as follows -

- On Linux:

  ```
  sudo apt-get install python3-matplotlib
  ```

- On Mac:

  ```
  pip3 install matplotlib
  ```

1. Set parameter values

The parameters in `run_measurements_and_produce_graphs.sh` control the measurement execution.

1.1. For running on a personal computer

To adjust the parameters to the number of cores in the machine running the measurements and create threads up to its number of virtual threads, the thread count parameters should be set according to the core count. This may be done as follows:

```
multicoreServer=F
```

This way, the number of threads in the measurements is made suitable for up to 8 concurrent threads, which may be appropriate for instance for a PC with 4 cores that enables hyperthreading.

1.2. For basic quick testing

For running less repetitions of each measurement and running each repetition for less time, the following parameters of `run_measurements_and_produce_graphs.sh` may be reduced:

```
warmupRepeats=1
repeats=2
runtime=1
```

To run only part of the measurements, simply comment out the other python commands in the `measurements` array.

2. Run measurements

To run all measurements and produce data and graph files, erase the `results` and `graphs` folders if they exist (after backing them up if required) so that the new outputs will not be mixed with old ones, then run:

```
./run_measurements_and_produce_graphs.sh
```

Details about the execution products appear below, in the *step-by-step guide*.

3. Gather graphs

To nicely present all produced graphs in a pdf file (similar to the figures in the paper in Section 9 - Evaluation), compile the `server_graphs.tex` or `pc_graphs.tex` file (depending on whether `run_measurements_and_produce_graphs.sh` was run with `multicoreServer=T` or `F`) in a latex compiler, e.g., online in Overleaf or locally by running the following command:

```
# command taken from https://tex.stackexchange.com/a/459470/246141
pdflatex -halt-on-error pc_graphs.tex | grep '^!.*' -A1 --color=always
```

Note that to produce such a pdf file, no python command in the `measurements` array in `run_measurements_and_produce_graphs.sh` may be commented out (or else, the produced graphs will appear in the `graphs` folder, but a pdf file will not be producible as described here).

Step-by-step guide to run full evaluation

Run tests

To run unit tests for all data structures before running the evaluation, simply run:

```
./compile.sh
./run_tests.sh
```

Run measurements and produce all outputs

Follow steps 1-3 (excluding step 1.2) as they appear above under *Getting started*.

Note that the full measurements run for several hours.

- To run only part of the measurements, comment out the other python commands in the `measurements` array in `run_measurements_and_produce_graphs.sh`. There, a description of each command, including the figures it relates to, appears in a comment above the command.
- To run a certain measurement on part of the data structures, erase the other data structures from the `dataStructures` list in the beginning of the corresponding python script.

For instance, to measure the overhead for a BST only, set the `dataStructures` list in `measurements/python_scripts/run_java_experiments_overhead.py` as follows:

```
dataStructures = ["BST", "SizeBST"]
```

Products

The standard output

The standard output describes the course of the execution, where the first part is the output of the Java code compilation, and the second part is the output of running the measurements.
For each measurement, the python measurement command that runs it is echoed to standard output.
After this line, a line of the form `Running <Java_run_title>` appears for each Java run executed for this measurement.
Then, a dot appears for each repetition of that run (including warmup repetitions).
When the measurement is over, the current date and time are printed before proceeding to the next measurement.

Raw results

The numeric results are written to csv files in the folder `results`. These files contain elaborate raw data.

Graphs

The graphs are written to png files in the folder `graphs`. The graphs visualize the key takeaways of the results.

Pdf with figures

The pdf produced in step 3 above gathers all graphs and presents them in a manner similar to the paper.

Verify results

Use the pdf produced in step 3 to verify the measurement results and compare them with the claims and figures that appear in the paper under Section 9 - Evaluation.

Different machines may yield different throughputs, but general trends should be the same:

1. Figures 7-9: *Overhead*
   The overhead of our size-supporting data structures (SizeHashTable, SizeBST and SizeSkipList) in comparison to the baseline data structures (HashTable, BST and SkipList) may vary; sometimes our data structures may also perform better (which is demonstrated in TP-loss bars going upwards instead of downwards, expressing a TP gain). The argument we make here is that the overhead incurred by our methodology is relatively not high (namely, the bars are not expected to go deeply downwards).
2. Figure 10-11: *Varying data-structure size*
   The size thread throughput of our size-supporting data structures (SizeHashTable, SizeBST and SizeSkipList), presented in Figure 10, is not expected to significantly decrease with increased data-structure size.
   On the other hand, the size thread throughput of the snapshot-based competitors (VcasBST-64 and SnapshotSkipList), presented in Figure 11, *is* expected to decrease.
3. Figure 12: *Scalability*
   The size thread throughput is expected to improve with increased number of threads (as long as there are enough cores to run them concurrently).
4. Figure 10-12: *Comparison to snapshot-based size*
   Our size-supporting data structures (SizeHashTable, SizeBST and SizeSkipList) are expected to perform significantly better than the snapshot-based competitors (VcasBST-64 and SnapshotSkipList).