TIE-20106 Data structures and algorithms, spring 2020

Programming assignment 1: Misse Meni? (Where’d it go?)

Last modified on 03/25/2020

Changelog

Below is a list of substantial changes to this document after its initial publication:

- 19.2. Made `stop_regions()` a compulsory operation and `region_bounding_box()` a non-compulsory operation.
- 5.3. Added a note emphasizing that implementation of often called operations should be faster than linear.
- 9.3. Added clarification that you cannot add region 1 as a subregion to region 2, if region 2 is already a direct or indirect subregion of region 1 (i.e. there cannot be cycles).
- 18.3. Updated the example output at the end, which was not up to date with the one in course-upstream repository.
- 24.3. Made it clearer that clear_all() also clears out regions in addition to stops.
- 25.3. Fixed the example run. It mistakenly included region_bounding_box (which is not compulsory) and didn’t have stop_regions (which is compulsory).

Topic of the assignment

The topic of the assignment this year is public transportation (to honor the near completion of the Tampere tram project). The first phase is to create a program which can tell information about bus stops and regions related to them (suburbs, municipalities, price zones, etc.). In the second phase the program will be extended to also include bus routes and allow bus journey searches. Some operations in the assignment are compulsory, others are not (compulsory = required to pass the assignment, not compulsory = can pass without it, but it is still part of grading).

In practice the assignment consists of implementing a given class, which stores the required information in its data structures, and whose methods implement the required functionality. The main program and the Qt-based graphical user interface are provided by the course (running the program in text-only mode is also possible).

Terminology

Below is explanation for the most important terms in the assignment:

- **Stop.** Every bus stop has a unique *integer ID, name* (which consists of characters A-Z, a-z, 0-9, space, and dash -) and *location (x,y)*, where x and y are integers (the scale is approximately in metres, the origin (0,0) of the coordinate system is arbitrary).
- **Region.** Every bus stop can belong to at most one region. Additionally each region can contain an arbitrary number of (sub)subregions, so that every region can belong to at most one “upper” region. An example of this could be a suburb (Hervanta) that belongs to a municipality (Tampere). Each region has a unique *ID* (which consists of characters A-Z, a-z, 0-9) and a *name* (which consists of characters A-Z, a-z, 0-9, space, and dash -). *The region relationships cannot for cycles, i.e. region 1 cannot be a subregion of region 2, if region 2 is already a direct or indirect subregion of region 1. The assignment does not have to prepare for attempts to add cyclic regions in any way.*

In the assignment one goal is to practice how to efficiently use ready-made data structures and algorithms (STL), but it also involves writing one’s own efficient algorithms and estimating their performance (of course it’s a good idea to favour STL’s ready-made algorithms/data structures when their can be justified by performance). In other words, in grading the assignment, asymptotic performance is taken into account, and also the real-life performance (= sensible and efficient implementation aspects). “Micro optimizations” (like do I write ”a = a+b;” or ”a += b;”, or how do I tweak compiler’s optimization flags) do not give extra points.

The goal is to code an as efficient as possible implementation, under the assumption that all operations are executed equally often (unless specified otherwise on the command table). In many cases you’ll probably have to make compromises about the performance. In those cases it helps grading when you document your choices and their justification in the **document file** that is submitted as part of the assignment. (Remember to write the document in addition to the actual code!)

Especially note the following (some of these are new, some are repeated because of their importance):

- The main program given by the course can be run either with a graphical user interface (when compiled with QtCreator/qmake), or as a textual command line program (when compiled just with g++ or some other C++ compiler). In both cases the basic functionality and students' implementation is exactly the same.
- **Hint** about (not) suitable performance: If the performance of any of your operations is worse than $\Theta(n \log n)$ on average, the performance is definitely *not* ok. Most operations can be implemented much faster. *Addition: This doesn’t mean that n log n would be a **good** performance for many operations. Especially for often called operations even linear performance is quite bad.*
- **As part of the assignment, file datastructures.hh contains a comment next to each operation. Into that comment you should write your estimate of the asymptotic performance of the operation, which a short rationale for you estimate.**

- **As part of the assignment submission, a document should be added to git (in the same directory/folder as the code). This document should contain reasons for choosing the data structures used in the assignment (especially performance reasons). Acceptable formats are plain test (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**
- Implementing operations `stops_closest_to()`, `remove_stop()`, `region_bounding_box()`, and `stops_common_region()` are not compulsory to pass the assignment. **If you only implement the compulsory parts, the maximum grade for the assignment is 3.**
- If the implementation is bad enough, the assignment can be rejected.
- In performance the essential thing is how the execution time changes with more data, not just the measured seconds. More points are given if operations are implemented with better performance.
- Likewise more points will be given for better real performance in seconds (if the required minimum asymptotic performance is met). **But** points are only given for performance that comes from algorithmic choices and design of the program (for example setting compiler optimization flags, using concurrency or hacker optimizing individual C++ lines doesn't give extra points).
- The performance of an operation includes all work done for the operation, also work possibly done during addition of elements.

On sorting

Sorting names should by done using the default string class "<" comparison, which is ok because names only allow characters a-z, A-Z, 0-9, space, and a dash -. Stops with equal names can be in any order with respect to each other.

Operations `min_coord`(), `max_coord`(), and `stops_coord_order`() require comparison of coordinates. The comparison is based on the "normal" euclidean distance from the origin $\sqrt{x^2+y^2}$ (the coordinate closer to origin comes first). If the distance to origin is the same, the coordinate with the smaller y-coordinate comes first. Coordinates with with equal distances and y-coordinates can be in any order with respect to each other.

In the non-compulsory operation `stops_closest_to()` stops are ordered based on their distance. In that case distance is the "normal" euclidean distance $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$. Stops with the same distance can be in any order with respect to each other.

About implementing the program and using C++

The programming language in this assignment is C++17. The purpose of this programming assignment is to learn how to use ready-made data structures and algorithms, so using C++ STL is very recommended and part of the grading. There are no restrictions in using C++ standard library. Naturally using libraries not part of the language is not allowed (for example libraries provided by Windows, etc.). *Please note however, that it's very likely you'll have to implement some algorithms completely by yourself.*

Structure and functionality of the program.

Part of the program code is provided by the course, part has to be implemented by students.

Parts provided by the course

Files *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui* (you are **NOT ALLOWED TO MAKE ANY CHANGES TO THESE FILES)**

- Main routine, which takes care of reading input, interpreting commands and printing out results. The main routine contains also commands for testing.
- If you compile the program with QtCreator or qmake, you'll get a graphical user interface, with an embedded command interpreter and buttons for pasting commands, file names, etc in addition to keyboard input. The graphical user interface also shows a visual representation of stops, their regions, results of operations, etc.

File *datastructures.hh*

- `class Datastructures`: The implementation of this class is where students write their code. The public interface of the class is provided by the course. You are **NOT ALLOWED TO CHANGE THE PUBLIC INTERFACE** (change names, return type or parameters of the given public member functions, etc., of course you are allowed to add new methods and data members to the private side).
- Type definition `StopID`, which used as a unique identifier for each stop (and which is used as a return type for many operations). There can be several stops with the same name (and even same coordinates), but every stop has a different id.
- Type definition `Coord`, which is used in the public interface to represent (x,y) coordinates. As an example, some comparison operations (==, !=, <) have been implemented for this type.
- Type definition `RegionID`, which used as a unique identifier for each region. There can be several regions with the same name, but every region has a different id.
- Type definition `NAME`, which is used for names of stops and regions, for example. (The type is a string, and the main routine allows names to contain characters a-z, A-Z, 0-9, space, and dash -).
- Constants `NO_STOP`, `NO_REGION`, `NO_NAME`, and `NO_COORD`, which are used as return values, if information is requested for a stop or region that doesn't exist.

File *datastructures.cc*

- Here you write the code for the your operations.
- Function `random_in_range`: Like in the first assignment, returns a random value in given range (start and end of the range are included in the range). You can use this function if your implementation requires random numbers.

On using the graphical user interface

When compiled with QtCreator, a graphical user interface is provided. It allows running operations and test scripts, and visualizing the data. The UI has some phase 2 controls, which are currently disabled (greyed out).

The UI has a command line interface, which accepts commands described later in this document. In addition to this, the UI shows graphically created stops and regions (*if you have implemented necessary operations, see below*). The graphical view can be scrolled and zoomed. Clicking on a stop or region name prints out its information, and also inserts the ID on the command line (a handy way to give commands ID parameters). The user interface has selections for what to show graphically.

Note! The graphical representation gets all its information from student code! **It's not a depiction of what the "right" result is, but what information students' code gives out.** The UI uses operation `all_stops`() to get a list of stops, and asks their information with `get_...`() operations. If drawing regions is on, they are obtained with operation `all_regions`(), and the name of each region with `get_region_name`(). The location and size of the region is obtained by calling `region_bounding_box`(). **NOTE! Operation `region_bounding_box()` is a non-compulsory operation. If it has not been implemented, regions are not drawn graphically.**

Parts of the program to be implemented as the assignment

Files *datastructure.hpp* and *datastructure.cpp*

- `class Datastructures`: The given public member functions of the class have to be implemented. You can add your own stuff into the class (new data members, new member functions, etc.)
- **In file *datastructures.hh*, for each member function you implement, write an estimation of the asymptotic performance of the operation (with a short rationale for your estimate) as a comment above the member function declaration.**

Additionally the readme.pdf mentioned before is written as a part of the assignment.

Note! The code implemented by students does not print out any output related to the expected functionality, the main program does that. If you want to do debug-output while testing, use the `cerr` stream instead of `cout` (or `qDebug`, if you you use Qt), so that debug output does not interfere with the tests.

Commands recognized by the program and the public interface of the Datastructures class

When the program is run, it waits for commands explained below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class (implemented by students). Some commands are completely implemented by the code provided by the course.

If the program is given a file as a command line parameter, the program executes commands from that file and then quits.

The operations below are listed in the order in which we recommend them to be implemented (of course you should first design the class taking into account all operations).

Command **Public member function**	**Explanation**
`stop_count` `int stop_count()`	Returns the number of stops currently in the data structure.
`clear_all` `void clear_all()`	Clears out the data structures (after this `all_stops()` and `all_regions()` return empty vectors). *This operation is not included in the default performance tests.*
`all_stops` `std::vector<StopID> all_stops()`	Returns a list (vector) of the stops in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*
`add_stop ID Name (x,y)` `bool add_stop(StopID id, Name const& name, Coord xy)`	Adds a stop to the data structure with given unique id, name, and coordinates. Initially a stop is not part of any region. If there already is a stop with the given id, nothing is done and `false` is returned, otherwise `true` is returned.
`stop_name ID` `Name get_stop_name(StopID id)`	Returns the name of the stop with given ID, or NO_NAME if such stop doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*
`stop_coord ID` `Coord get_stop_coord(StopID id)`	Returns the name of the stop with given ID, or value NO_COORD, if such stop doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*
(The operations below should probably be implemented only after the ones above have been implemented.)	
`stops_alphabetically` `std::vector<StopID> stops_alphabetically()`	Returns stop IDs sorted according to alphabetical order of stop names. Stops with the same name can be in any order with respect to each other.
`stops_coord_order` `std::vector<StopID> stops_coord_order()`	Returns stop IDs sorted according to their coordinates (defined earlier in this document). Stops with the same location can be in any order with respect to each other.
`min_coord` `StopID min_coord()`	Returns the stop with the smallest coordinate (based on the ordering of coordinates described earlier). If there are several such stops, returns one of them. If no stops exist, NO_STOP is returned.
`max_coord` `StopID max_coord()`	Returns the stop with the largest coordinate (based on the ordering of coordinates described earlier). If there are several such stops, returns one of them. If no stops exist, NO_STOP is returned.

Command **Public member function**	**Explanation**

`find_stops name`
`std::vector<StopID>`
`find_stops(Name const& name)`
	Returns all stops with the given name, or an empty vector, if no such stops exist. The order of the returned stops can be arbitrary (the main routine sorts them based on their ID). *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests.*

`change_stop_name ID newname` `bool change_stop_name(StopID id, Name const& newname)`	Changes the name of the stop with given ID. If such stop doesn't exist, returns `false`, otherwise `true`.
`change_stop_coord ID (x,y)` `bool change_stop_coord(StopID id, Coord newcoord)`	Changes the location of the stop with given ID. If such stop doesn't exist, returns `false`, otherwise `true`.
(The operations below should probably be implemented only after the ones above have been implemented.)	
`add_region ID Name` `bool add_region(RegionID id, Name const& name)`	Adds a region to the data structure with given unique id and name. Initially the added region is not a subregion of any region, and it doesn't contain any stops or regions. If there already is a region with the given id, nothing is done and `false` is returned, otherwise `true` is returned.
`region_name ID` `Name get_region_name(RegionID id)`	Returns the name of the region with given ID, or NO_NAME if such region doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*

`all_regions`
`std::vector<RegionID>`
`all_regions()`
	Returns a list (vector) of the regions in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*

`add_stop_to_region StopID RegionID` `bool add_stop_to_region(StopID id, RegionID parentid)`	Adds a given stop to a given region. If no stop or region exists with the given IDs, or if the stop already belongs to a region, nothing is done and `false` is returned, otherwise `true` is returned.
`add_subregion_to_region RegionID RegionID` `bool add_subregion_to_region(RegionID id, RegionID parentid)`	Adds a given region as a subregion to another region. If no regions exist with the given IDs, or if the given region is already a subregion of some region, nothing is done and `false` is returned, otherwise `true` is returned.

`stop_regions ID`
`std::vector<RegionID>`
`stop_regions(StopID id)`
	Returns a list of regions to which the given stop belongs either directly or indirectly. The returned vector first contains the region to which the stop belongs, then the region that region belongs to, etc. If no stop with given ID exists, a vector with a single element NO_REGION is returned.

Command **Public member function**	**Explanation**
(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	
`creation_finished` `void creation_finished()`	Performance tests call this operation after most of the stops and regions have been created (after calling this new stops and regions are added seldom). This operation doesn't have to do anything, but you can use it for algorithmic optimizations, if you want.

`stops_closest_to ID`
`std::vector<StopID>`
`stops_closest_to(StopID id)`
	Returns five stops closest to the given stop in order of increasing distance (based on the ordering of coordinates described earlier). If there are less than six stops, of course returns less stops. If the ID does not correspond to any stop, a vector with a single element NO_STOP is returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`remove_stop ID` `bool remove_stop(StopID id)`	Removes a stop with the given id. If a stop with given id does not exist, does nothing and returns `false`, otherwise returns `true`. If the stop to be removed is part of a region, it's of course removed from the region. *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests. Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`region_bounding_box ID`
`std::pair<Coord, Coord>`
`region_bounding_box(RegionID id)`
	Returns the region's *bounding box*, i.e. the smallest rectangle inside which all region's stops fit (including stops of subregions, their subregions, etc.). In other words, calculates the minimum and maximum x and y values of the coordinates of region's stops, the first coordinate is the lower left corner, the second the upper right corner. If the region and its subregions (and their subregions, etc.) do not contain any stops, both coordinates are returned as NO_COORD. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`stops_common_region ID ID`
`RegionID`
`stops_common_region(StopID id1,`
`StopID id2)`
	Returns the "lowest" common region in the subregion hierarchy for the stops. That is, a region to which both stops belong either directly or indirectly, but both stops do not belong to any of the region's subregions.. If either stop does not correspond to any stop, or if no common region exists, returns NO_REGION.

(The following operations are already implemented by the main program.)	

Command **Public member function**	**Explanation**
random_add n (implemented by main program)	Add *n* new stops with random id, name, and coordinates (for testing). With a 80 % probability the stop is also added to a random region. Note! The values really are random, so they can be different for each run.
random_seed n (implemented by main program)	Sets a new seed to the main program's random number generator. By default the generator is initialized to a different value each time the program is run, i.e. random data is different from one run to another. By setting the seed you can get the random data to stay same between runs (can be useful in debugging).
read ”filename” (implemented by main program)	Reads more commands from the given file (This can be used to read a list of stops from a file, run tests, etc.)
stopwatch on / off / next (implemented by main program)	Switch time measurement on or off. When program starts, measurement is "off". When it is turned "on", the time it takes to execute each command is printed after the command. Option "next" switches the measurement on only for the next command (handy with command "read" to measure the total time of a command file).
perftest all/compulsory/cmd1;cmd2... **timeout n n1;n2;n3...** (implemented by main program)	Run performance tests. Clears out the data structure and add *n1* random stops and some regions (see random_add). Then a random command is performed *n* times. The time for adding elements and running commands is measured and printed out. Then the same is repeated for *n2* elements, etc. If any test round takes more than *timeout* seconds, the test is interrupted (this is not necessarily a failure, just arbitrary time limit). If the first parameter of the command is *all*, commands are selected from all commands. If it is *compulsory*, random commands are selected only from operations that have to be implemented. If the parameter is a list of commands, commands are selected from that list (in this case it's a good idea to include also random_add so that elements are also added during the test loop). If the program is run with a graphical user interface, "stop test" button can be used to interrupt the performance test (it may take a while for the program to react to the button).
testread "in-filename" "out-filename" (implemented by main program)	Runs a correctness test and compares results. Reads command from file in-filename and shows the output of the commands next to the expected output in file out-filename. Each line with differences is marked with a question mark. Finally the last line tells whether there are any differences.
help (implemented by main program)	Prints out a list of known commands.

`Command` **`Public member function`**	**Explanation**
quit (implemented by main program)	Quit the program. (If this is read from a file, stops processing that file.)

"Data files"

The easiest way to test the program is to create "data files", which can add a bunch of stops and regions. Those files can then be read in using the "read" command, after which other commands can be tested without having to enter those stops every time by hand.

Below are examples of a data files, one of which adds stops, the other regions:

- *example-stops.txt*

```
# Add stops
add_stop 1 One (1,1)
add_stop 2 Two (6,2)
add_stop 3 Three (0,6)
add_stop 4 Four (7,7)
add_stop 5 Five (4,4)
add_stop 6 Six (2,9)
```

- *example-regions.txt*

```
# Add regions and stops to regions
add_region S Small
add_stop_to_region 1 S
add_stop_to_region 2 S
add_stop_to_region 3 S
add_region L Large
add_subregion_to_region S L
add_stop_to_region 4 L
add_stop_to_region 5 L
add_stop_to_region 6 L
```

Example run

Below are example outputs from the program. The example's commands can be found in files *example-compulsory-in.txt* and *example-all-in.txt*, and the outputs in files *example-compulsory-out.txt* and *example-all-out.txt*. I.e., you can use the example as a small test of compulsory behaviour by running command
`testread "example-compulsory-in.txt" "example-compulsory-out.txt"`

```
> clear_all
Cleared everything.
> stop_count
Number of stops: 0
> read "example-stops.txt"
** Commands from 'example-stops.txt'
> # Add stops
> add_stop 1 One (1,1)
One: pos=(1,1), id=1
> add_stop 2 Two (6,2)
Two: pos=(6,2), id=2
> add_stop 3 Three (0,6)
Three: pos=(0,6), id=3
> add_stop 4 Four (7,7)
Four: pos=(7,7), id=4
> add_stop 5 Five (4,4)
Five: pos=(4,4), id=5
> add_stop 6 Six (2,9)
Six: pos=(2,9), id=6
>
** End of commands from 'example-stops.txt'
> stop_count
Number of stops: 6
> stop_name 1
Stop ID 1 has name 'One'
One: pos=(1,1), id=1
> stop_coord 5
Stop ID 5 is in position (4,4)
Five: pos=(4,4), id=5
> stops_alphabetically
1. Five: pos=(4,4), id=5
2. Four: pos=(7,7), id=4
3. One: pos=(1,1), id=1
4. Six: pos=(2,9), id=6
5. Three: pos=(0,6), id=3
```

```
6. Two: pos=(6,2), id=2
> min_coord
One: pos=(1,1), id=1
> max_coord
Four: pos=(7,7), id=4
> stops_coord_order
1. One: pos=(1,1), id=1
2. Five: pos=(4,4), id=5
3. Three: pos=(0,6), id=3
4. Two: pos=(6,2), id=2
5. Six: pos=(2,9), id=6
6. Four: pos=(7,7), id=4
> change_stop_name 5 Two
Two: pos=(4,4), id=5
> find_stops Two
1. Two: pos=(6,2), id=2
2. Two: pos=(4,4), id=5
> read "example-regions.txt"
** Commands from 'example-regions.txt'
> # Add regions and stops to regions
> add_region S Small
Region: Small: id=S
> add_stop_to_region 1 S
Added stop One to region Small
Region: Small: id=S
One: pos=(1,1), id=1
> add_stop_to_region 2 S
Added stop Two to region Small
Region: Small: id=S
Two: pos=(6,2), id=2
> add_stop_to_region 3 S
Added stop Three to region Small
Region: Small: id=S
Three: pos=(0,6), id=3
> add_region L Large
Region: Large: id=L
> add_subregion_to_region S L
Added subregion Small to region Large
> add_stop_to_region 4 L
Added stop Four to region Large
Region: Large: id=L
Four: pos=(7,7), id=4
> add_stop_to_region 5 L
Added stop Two to region Large
Region: Large: id=L
Two: pos=(4,4), id=5
> add_stop_to_region 6 L
Added stop Six to region Large
Region: Large: id=L
Six: pos=(2,9), id=6
>
** End of commands from 'example-regions.txt'
> all_regions
1. Large: id=L
2. Small: id=S
> region_name S
Region ID S has name 'Small'
Small: id=S
```

```
> stop_regions 1
Regions for stop One: pos=(1,1), id=1
1. Small: id=S
2. Large: id=L
> quit
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-stops.txt* and *example-regions.txt* have been read in.