COMPLEXITY & ALGORITHMS 2.1

[Replacement selection]

JANUARY 13, 2020

[MICHAEL CORNELISSE (459466) LARS WIEGERS (468477)

Table of Contents

Introduction:

This report covers assignment two of complexity and algorithms performed by Michael Cornelisse and Lars Wiegers. For the second assignment we once again had the possibility to choose between two assignments, (2.1) Replacement selection and (2.2) DEPQ we decided to implement assignments (2.1) Replacement selection. To be perfectly honest we weren't really sure what we were getting into when we started working on this assignment and we simply chose it because the pseudo code for the sorting algorithms was already provided in the slides during the lecture leading up prior to that, and the amount of research into what type of heap that was needed seemed less daunting in this assignment compared to the second one.

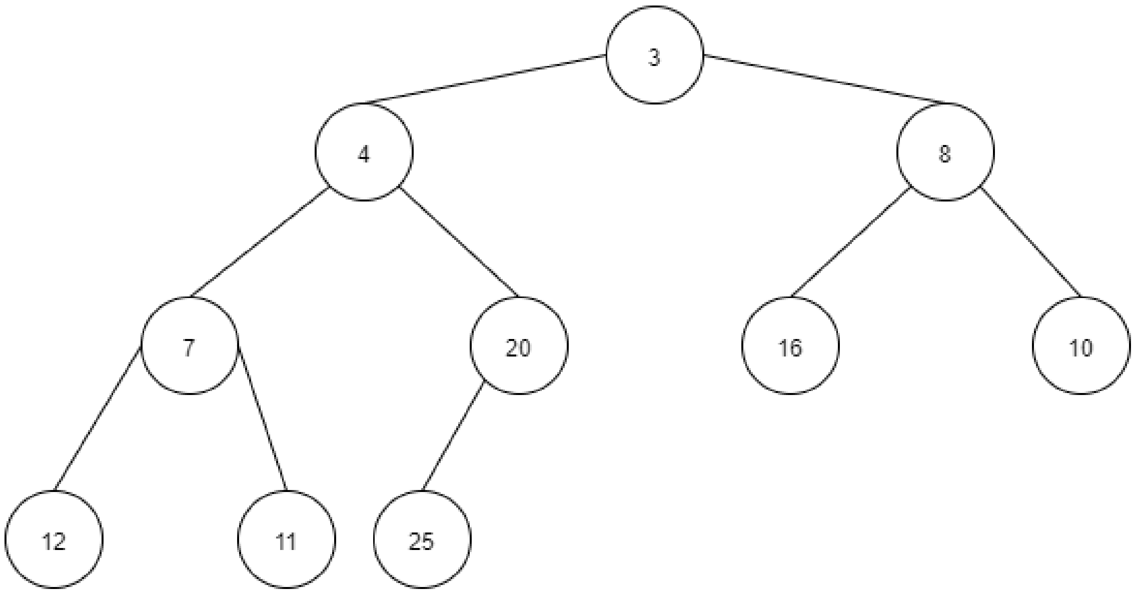
Initial analysis:

This chapter discusses the initial thoughts we had about the assignment and our initial impression of the assignment once we started working on it.

As we described above, we initial felt that this assignment was less daunting since the pseudo code for the algorithm was provided and the possible heap implementations seemed less complex compared to DEPQ. We found out quite quickly that we were a tad too optimistic in this regard initially. After reading the description again a couple times we concluded it was quite cryptic. And searching the internet for the terms specified in the description did not turn up a lot of useful results terms like DynDSHeap, dead space and terms like percolate down and percolate up (These ended up having several different definitions on the internet) these are often called bubble up and sink down or heapify up or heapify down operations. There was a lot more choices up for own personal interpretation and therefore a lot more decisions had to be based on our own judgement compared to the previous assignment.

Expected outcome of the algorithm:

Given the following input [10, 12, 8, 11, 25, 16, 3, 7, 4, 20] if we would build a min heap from this output with size ten, we would get the following result:



However, if were to take this input and run the algorithm with a heap size of three would result in the following output according to the algorithm:

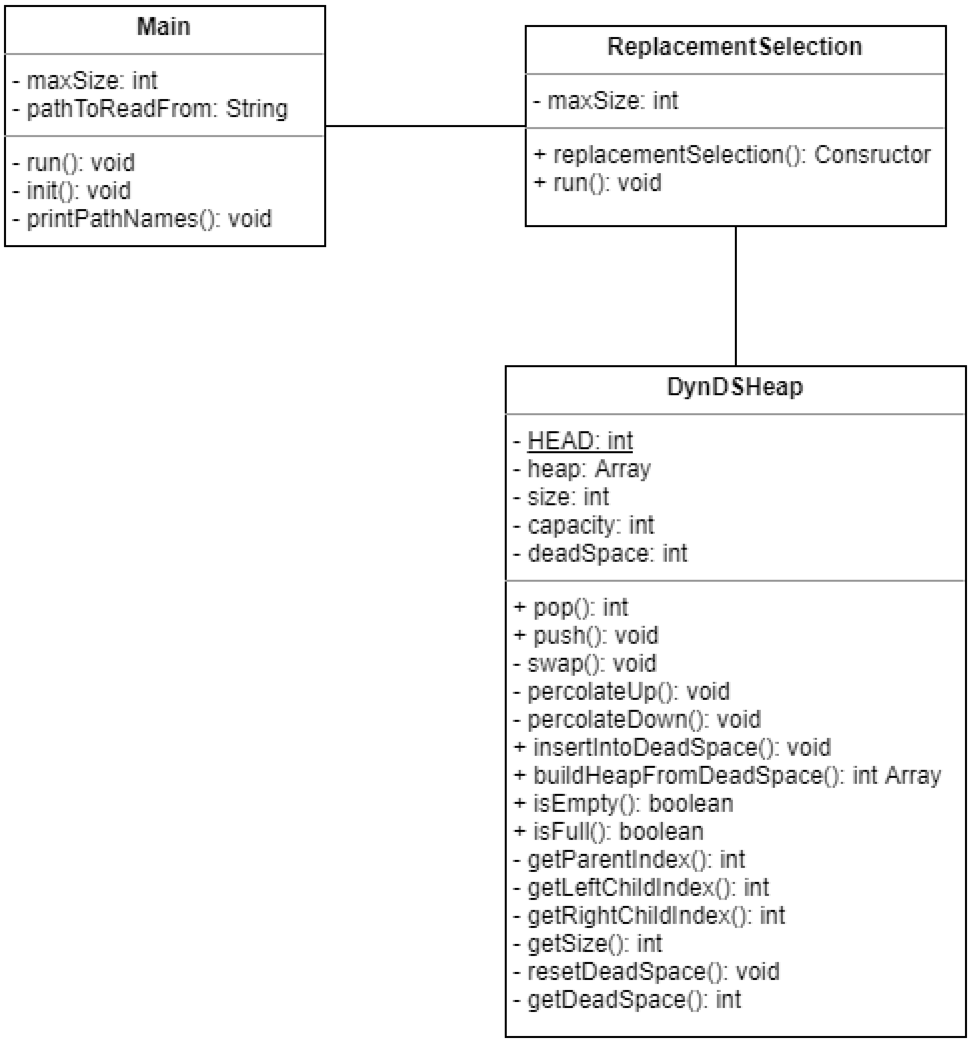
Run: 1
8
10
11
12
16
25

Run: 2
3
4
7
20

Setup:

In the following chapter we discuss our design choices for the program the following things are covered below class diagram, program overview, heap implementation, user manual and an elaboration on design choices and limitations.

Class diagram:



Program overview:

There are two executable files present within the project, class Main which serves as the driver for our assignment program that being the replacement selection algorithm with DynDsHeap. The second executable file is a utility class InputFileHandler which is a program that allows to generate input files with random integer numbers based on a range based on input provided by the user. an overview and description of all the classes are provided below.

Main program:

Main:

Main driver class for our program when run the user is asked to choose which of the files currently present in the data directory to run, and the max heap capacity.
Once these are provided the algorithm will be run.

Algorithm package:

Replacement Selection:

The algorithm of our program this class contains a constructor that takes the input file and max heap size in its arguments, and a void method that runs the algorithm.

Data structure package:

DynDsHeap:

The data structure we use within our algorithm based on a min heap implementation a more detailed description of the heap we implemented, and its methods will be provided further down below in the document in chapter Heap Implementation.

Utility package:

File Input reader and writer:

Utility class that provides a wrapper for the BufferedReader and PrintWriter and is responsible for all the reading and writing operations for our program.

Result handler:

Utility class responsible for computing the results generated by the program returning the total elements read from the file and the average of elements read.

Run Counter:

Utility class that keeps track of the number of runs being performed by the algorithm. Returns current run count and total run count.

InputFileHandler:

Utility class that allows the user to create input files with for the program with a random integer numbers with a range provided by the user.

Test package:

The test package will be discussed in the results chapter down below at page (8).

Heap implantation:

For this assignment we chose to implement a min-heap as our data-structure, to our understanding it was possible to either use a min-heap or max-heap for this algorithm. The reason we chose a min-heap is because it felt more natural to keep the min element as root and larger elements as leaves. Below is an overview of all methods used in our heap.

pop():
This method removes and returns the smallest element from the heap. After which the heap size will shrink and the head will percolate down to maintain heap balance.

push():
This method pushes a new element onto the heap, after which heap size will increase and percolate up will be performed to maintain balance

swap():
Is used to swap two node indexes, used in Percolate down to swap smallest with given index.
Used in percolate up to swap current index with that parent index.

percolateUp():
Moves current index to insert up into the heap once a push operation is performed.
Keep repeating this operation till node with the given index reaches the correct position.

percolateDown():
Moves Head of the heap down once pop operation is performed. It keeps repeating this step until the node has reached the correct position.

insertIntoDeadSpace():
Inserts an element into dead space once its established it’s no longer allowed to be added onto the heap. Heap capacity will decrease, and dead space will increase once this operation is performed.

buildHeapFromDeadSpace():
Creates a new array with the length of dead space and builds a new heap based on the elements that are stored there. After which the dead space is reset to zero and the new array is returned.

isEmpty():
Boolean method to validate if the heap is empty.

isFull():
Boolean method to validate if the heap is full.

getParentIndex():
Int method that returns the parent index.

getLeftChildIndex():
Int method that returns left child index.

getRightChildIndex():
Int method that returns right child index.

getSize():
Returns heap size.

resetDeadSpace():
Resets dead space size back to zero.

resetDeadSpace():
Resets dead space size back to zero.

getDeadSpace():
Returns current dead space size.

User manual:

Instructions for when you run the program

When you run the program, the user is presented with two choices. For the first choice the user gets presented with a list of files and can decide on which file to read. For the second choice the user can input the max heap size. After the user has made its choices the program will run, and an output file will be generated with the results.

Generating Input files

The project comes with another executable program as described earlier in the chapter above, located in the utility package called InputFileHandler. When the program is run it will generate input files based on the range provided by the user until the user enters q to quit.

Providing your own input files

It is possible to provide your own input files by placing them into the data directory in the project, just keep note of the following limitations the data inside those files and its formatting. The data can only be integer numbers and the must be present in the file in a vertical order one integer per line this is due to limitations caused by our design a further elaboration on that is provided below.

Elaboration on some flaws in the design choices:

When we started developing the program, we figured it would be the easiest to work with integers, at that moment we decided our heap to be of type int. When we started to work with the buffered reader we ran into the issue that it only would be read the numbers in a vertical order and it would crash whenever it was presented with a file containing numbers in a horizontal order. This limitation can probably be solved but due to time constraints we have chosen not too to implement this in this current version.

As specified earlier the program currently only takes integer types this puts limitations on taking in account other special cases like handling floating point numbers or doubles. This was an oversight on our part when we started developing the program and we chose not to implement this due to time constraints but are aware that this could enhance the program.

If we would have more time or write the program again in the future, we would do the following. We would make the heap a comparable type and check conditionally what type of number we would be dealing with.

Results:

This chapter discusses the reached results of the assignment. In order to verify the program was working correctly we tested multiple input files with varying input and max heap sizes and checked the output results both in the console and output file by hand. This was a very time intensive process so in addition to that we wrote several unit tests to verify the program works correctly.
Down below we provide an example run with a printout for both the console and output file as well as a breakdown of the provided tests included in the project.

The test package:

Console writer:
Wrapper class for Print stream operations.

Main Run Thread:
Used to run Main in its own thread and let the test thread sleep when no tests are being run.

Main Test:
Class containing methods for all the test cases of the program. All of which will be described bellow

whenSuccessfullyCreatesOutputFile():
Tests whether a output file gets successfully created.

whenSuccessfullyWritesToOutputFile():
Tests whether output actually gets written to an output file.

outputsRunsToTheFile():
Tests whether the line Runs gets written to an output file.

outputsExpectedRunCountToTheFile():
Tests whether run count written to output matches the expected total run count.

makeSureThatNumbersInRunAreGoingUp():
Tests whether the numbers written to output in each run are going up.

ifGivenSameInputTwiceReturnsTheSameResult():
Tests whether output is consistent between different run cycles of the program with the same input.

slowdownForWindows():
This method keeps tests predictable by taking into account that windows sometimes takes a bit longer to detect when a new file is made.

Example run of the program:

Console output:

Given the input [7, 8, 15, 14, 9, 13, 9, 5, 6, 7] with a max heap size of 5 the following printouts are produced:

```
Inserting element 7 into heap at index 0
Inserting element 8 into heap at index 1
Inserting element 15 into heap at index 2
Inserting element 14 into heap at index 3
Inserting element 9 into heap at index 4

Removing smallest element: 7 from the Heap..

Heap size after removing smallest element: 4

Inserting element 13 into heap at index 4

Removing smallest element: 8 from the Heap..

Heap size after removing smallest element: 4

Inserting element 9 into heap at index 4

Removing smallest element: 9 from the Heap..

Heap size after removing smallest element: 4

Inserting 5 into deadSpace at index 0

Removing smallest element: 9 from the Heap..

Heap size after removing smallest element: 3

Inserting 6 into deadSpace at index 1

Removing smallest element: 13 from the Heap..

Heap size after removing smallest element: 2

Inserting 7 into deadSpace at index 2

Removing smallest element: 14 from the Heap..

Heap size after removing smallest element: 1

Removing smallest element: 15 from the Heap..

Heap size after removing smallest element: 0

new heap is: [7, 6, 5]

Inserting element 7 into heap at index 0
Inserting element 6 into heap at index 1
Inserting element 5 into heap at index 2

Removing smallest element: 5 from the Heap..

Heap size after removing smallest element: 2
```

```
Removing smallest element: 6 from the Heap..

Heap size after removing smallest element: 1

Removing smallest element: 7 from the Heap..

Heap size after removing smallest element: 0
```

The following end results are writing to the output file:

```
Run: 1
7
8
9
9
13
14
15

Run: 2
5
6
7

Total number of runs: 2
Total number of elements read: 10
Average number of elements read: 5.0
```