

COMP90077 Assignment 2 (20% of the Subject Assessment)

An Experimental Study on Treaps

Due Date: **23:59 May 24 (Wednesday) 2023**

Prepared by **Junhao Gan** and **Tony Wirth**

Background. When writing a research paper, sometimes, in addition to theoretical analysis, it is also important to include a section to demonstrate the *performance in practice* of the proposed algorithms (and data structures) by *experiments*. A good experiment section can often maximize the chance of a paper to be accepted, as it serves as extra convincing evidence for the superiority of the proposition in the paper.

In this assignment, you are asked to conduct an experimental study on Treaps and write an experiment section to demonstrate the results as if you are writing a research paper on this data structure.

Specifications. There are **three** main tasks in this assignment: (i) implement a data generator, a randomized treap and a simple competitor; (ii) conduct the required experiments; and (iii) write a report to demonstrate and analyse the experimental results.

Task 1: Implementations.

About Programming. There is *no* specified programming language in this assignment; you can use whatever programming language you prefer and are confident with, as long as the running time of your programs can be measured properly.

The Data Generator. In the experiment, each *data element* is a pair (id, key) , where *id* is a *unique identifier* and *key* is the *search key* of the element. Both *id* and *key* are **integers**. It may be helpful to keep an integer id_{next} to record the value of next identifier. Initially, $id_{next} \leftarrow 1$. More specifically, the data generator should have an interface, called *gen_element()*, to generate a new element by the following steps.

gen_element():

- $id \leftarrow id_{next}, id_{next} \leftarrow id_{next} + 1$;
- $key \leftarrow$ an integer that is drawn uniformly at random from the range $[0, 10^7]$;
- return element (id, key) ;

Since the identifiers of the elements are unique, when two elements happen to have a same search key, we can break the tie by taking the element with smaller *id* value as the smaller between the two. Furthermore, the data generator should also have three more interfaces, *gen_insertion()*, *gen_deleteion()* and *gen_search()*, to generate an insertion, a deletion and a search operation, respectively. Specifically, these operations have the following forms:

- an insertion of an element x : $(1, x)$, where 1 indicates that this operation is an insertion, and the element x is represented by an id-key pair, i.e., $x = (id, key)$; for example, $(1, 10, 5)$ means an insertion of an element with $id = 10$ and $key = 5$;
- a deletion of a search key key_{del} : $(2, key_{del})$, where 2 indicates that this operation is a deletion, and key_{del} is the search key to be deleted; note that key_{del} is not necessarily in the current element set;

- a search of a key key_{sch} : $(3, key_{sch})$, where 3 indicates that this operation is a search, and key_{sch} is the search key to be searched.

The detailed steps of these three interfaces are as follows:

gen_insertion():

- $x \leftarrow$ a new element generated by invoking *gen_element()*;
- return $(1, x)$, an insertion for x ;

gen_deletion():

- $id_{del} \leftarrow$ an integer that is drawn uniformly at random from the range $[1, id_{next} - 1]$;
- if the element x with id_{del} has already been deleted (in this case, we don't have access to the key of x), then we do the following:
 - $key_{del} \leftarrow$ an integer that is drawn uniformly at random from the range $[0, 10^7]$;
 - return $(2, key_{del})$, a deletion for deleting the search key key_{del} ;
- otherwise, return $(2, key_{del})$, a deletion for deleting the search key $x.key$ of the element x ;

As it is possible that more than one element has the same search key, in a deletion, deleting an *arbitrary* element found with the target key is fine. As we will see shortly next, when generating a sequence of operations mixing with insertions and deletions, detecting whether an element has been deleted or not may be challenging. To achieve this, during the generation, we can maintain a *sufficiently large* array, where the cell at index i in the array is a boolean flag recording whether the element with $id = i$ has been deleted or inserted.

gen_search():

- $key_{sch} \leftarrow$ an integer that is drawn uniformly at random from the range $[0, 10^7]$;
- return $(3, key_{sch})$, a search operation with search key key_{sch} ;

The Randomized Treap. The randomized treap should be able to support three operations:

- *insert*(x): insert a new element x ;
- *delet*(key_{del}): delete an arbitrary element (if it exists) from the treap with search key key_{del} ;
- *search*(q): return an arbitrary element (if it exists) that has a search key $key = q$; otherwise, return NULL.

The Competitor. The competitor in this experiment is a *dynamic array*, which supports the following operations:

- *insert*(x): call the *push-back* operation to insert element x ; note that the length will be doubled when the current array is full as discussed in class;
- *delete*(key_{del}): scan the array from the beginning; if an element x with search key key_{del} :
 - swap x with the *last* element at the back of the array;
 - delete x from the back;

- if the current number of elements is smaller than $1/4$ length of the current array, *shrink* the array by *half* by: (i) creating a new array with *half* length, (ii) copying all elements to the new array, and (iii) deleting the old array;
- $search(key_{sch})$: scan the array from the beginning and return the first element (if it exists) with search key $key = key_{sch}$; otherwise, return NULL.

Task 2: Experiments.

Conduct the following *five* experiments.

Exp 0: Treap Height and Average Depths of Nodes. Consider a set P of 1024 integer keys $\{1, 2, 3, \dots, 1024\}$. In this experiment, we study the *average depths* of nodes in a randomized treap on P . Specifically, we repeat the following trials for 100 times:

- assign to each key a priority which is an integer chosen uniformly at random from $[1, 10^7]$; if the priorities of any two keys are the same, we consider the priority of the smaller key is smaller;
- construct a randomized treap on P with respect to their priorities;
- record the height of the treap and the depth of each node.

Exp 1: Time v.s. Number of Insertions. In this experiment, we study the *total running time* (of the randomized treap and the dynamic array, respectively) v.s. the lengths L_{ins} of *insertion-only* sequences. More specifically, we **first** generate *five* insertion-only sequences $\sigma_1, \sigma_2, \dots, \sigma_5$ respectively with $L_{ins} = 0.1M, 0.2M, 0.5M, 0.8M, 1M$, where M stands for *one million*, i.e., 10^6 . **Then** we input the **same** $\sigma_1, \dots, \sigma_5$ to each of the randomized treap and the dynamic array. To achieve this, it may be helpful to output each of the operation sequences into a file. When testing a data structure (either the randomized treap or the dynamic array), we can read the file into memory and perform all these operations accordingly.

Exp 2: Time v.s. Deletion Percentage. In this experiment, we consider a sequence of a *fixed length* $L = 1M$ of updates (including both insertions and deletions). In particular, we vary the *in-expectation percentage*, $\%_{del}$, of the deletions in five update sequences $\sigma_1, \dots, \sigma_5$, where $\%_{del} = 0.1\%, 0.5\%, 1\%, 5\%, 10\%$. Specifically, an update sequence σ with $\%_{del}$ is generated as follows:

- $\sigma \leftarrow \emptyset$;
- for $i = 1$ to $L = 1M$,
 - generate an update s_i such that: s_i is a deletion with probability $\%_{del}$, and it is an insertion with probability $1 - \%_{del}$;
 - add s_i to σ ;
- return the update sequence σ ;

Run the randomized treap and the dynamic array on these sequences $\sigma_1, \dots, \sigma_5$, and measure their corresponding total running time.

Exp 3: Time v.s. Search Percentage. Analogous to Exp 2, we consider a sequence of a *fixed length* $L = 1M$ of operations which contains insertions and searches *only*. We vary the *in-expectation percentage*, $\%_{sch}$, of the search operations in the operation sequence, where $\%_{sch} =$

0.1%, 0.5%, 1%, 5%, 10%. An operation sequence σ with $\%_{sch}$ is generated in an analogous way as the update sequence with $\%_{del}$ in Exp 2. Run both of the two algorithms on these sequences and measure their corresponding overall running time on each of the sequences.

Exp 4: Time v.s. Length of Mixed Operation Sequence. In this experiment, we consider operation sequences that mix insertions, deletions and searches with different lengths $L = 0.1M, 0.2M, 0.5M, 0.8M, 1M$. Each of these sequences is generated such that each operation is a deletion with $\%_{del} = 5\%$, a search with $\%_{sch} = 5\%$, and an insertion with the rest probability. Measure the corresponding overall running time of both of the two algorithms on each of such sequences.

Task 3: The Report.

In this Assignment, you are assessed on the basis of the report of your experimental results and the corresponding analysis (or say, explanations) on the results. Specifically, the report should contain the following components:

- Experiment Environment. Explain clearly the necessary information for others to *reproduce* your experiment, such as: the CPU frequency, memory, operating system, programming language, compiler (if applicable), and etc. **[1 mark out of 20]**
- Data Generation. Describe how you obtain or generate the data (i.e., operations in our context). Although in the above specification, certain details of the data generation are already given, you will need to describe how your data is generated with *your own words*. **[1 mark out of 20]**
- Experiment 0. **[This experiment takes 4 marks out of 20]**
 - Result Demonstration. In this experiment, you are asked to draw two diagrams:
 - * **Diagram 1:** There will be *two* lines in this diagram. The first line plots the height of the randomized treap in each of the 100 trials. The second line plots the depth of the node with $key = 512$ in each of the 100 trials. Therefore, the x -axis of the diagram shows the trial ids (i.e., 1 to 100), and the y -axis is respectively the height of the randomized treap for the first line, and the depth of the node with $key = 512$ for the second line, in the corresponding trial. **[1 mark out of 4]**
 - * **Diagram 2:** Plot the *average depth* of each of the 1024 nodes over the 100 trials with a histogram, where the x -axis shows the keys (i.e., 1 to 1024) and the y -axis show the average depth of the nodes with the corresponding key over these 100 trials (for $i = 1, 2, \dots, 1024$). **[1 mark out of 4]**
 - Analysis. In addition to showing the diagrams, you are also asked to analyse the results. For example, you will need to explain, with the theoretical results you learned, why the height of treap shows like this in your diagram, and why the average depth of each node shows like that in your diagram. **[2 marks out of 4]**
- Experiments 1 - 4. **[Each experiment takes 3 marks; in total 12 marks out of 20]**
 - Result Demonstration. Report the results of the four experiments with diagrams. Specifically, in each diagram, the x -axis represents the variable: L_{ins} in Exp 1, $\%_{del}$ in Exp 2, $\%_{sch}$ in Exp 3, and L in Exp 4. The y -axis is the corresponding total running time. You will need to *plot* the results of the corresponding algorithms. Hence, there will be two lines in each diagram. **[1.5 mark out of 3]**

- *Analysis*. In addition to showing the experimental results, you are also asked to *analyse* them. For example, you will need to explain, with certain theoretical analysis, why an algorithm performs better than the other in some cases while in other cases not. Sometimes, you will also need to analyse the results *across* different diagrams. [1.5 marks out of 3]
- *Conclusion*. Conclude your findings in these experiments. For example, in which cases, which algorithm is better. [2 marks out of 20]

Marking Scheme. The marks for each component in the report are as shown above. The grading will be based on the *clarity* and *rigor* of your description or analysis for each part. This is individual work, and you must obey **academic integrity**. However, to ensure scientific integrity, indeed **reproducibility**, your descriptions of your experiments, their design and the design of the data structures must be sufficiently clear that a hypothetical algorithm developer could reproduce the whole experimental study with your report and obtain similar results.

Submissions. You should lodge your submission for Assignment 2 via the LMS (i.e., Canvas). *You must identify yourself in **each** of your source files and the report.* Poor-quality scans of solutions written or printed on paper will *not* be accepted. There are scanning facilities on campus, not to mention scanning apps for smartphones etc. Solutions generated directly on a computer are of course acceptable. Submit *two* files:

- A *report.pdf* file, comprising your report for the experimental study.
- A *code.zip* file, containing all your sources files of the implementations for the experiments.

Do not include the testing files, as these might be large. **REPEAT: DO NOT INCLUDE TESTING FILES!** Moreover, it is very important, so that you can justify ownership of your work, that you detail your contributions in comments in your code, and in your report.

Administrative Matters

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture. As a reminder, the late penalty for non-exam assessment is *two* marks per day (or part thereof) overdue. Requests for extensions or adjustment must follow the University policy (the Melbourne School of Engineering “owns” this subject), including the requirement for appropriate evidence.

Late submissions should also be lodged via the LMS, but, as a courtesy, please also email both the two lecturers (Junhao Gan and Tony Wirth) when you submit late. If you make both on-time and late submissions, please consult the subject coordinators as soon as possible to determine which submission will be assessed.

Individual work. You are reminded that your submission for this Assignment is to be your own individual work. Students are expected to be familiar with and to observe the University’s Academic Integrity policy <http://academicintegrity.unimelb.edu.au/>. For the purpose of ensuring academic integrity, every submission attempt by a student may be inspected, regardless of the number of attempts made.

Students who allow other students access to their work run the risk of also being penalized, even if they themselves are sole authors of the submission in question. **By submitting your**

work electronically, you are declaring that this is your own work. Automated similarity checking software may be used to compare submissions.

You may re-use code provided by the teaching staff, and you may refer to resources on the Web or in published or similar sources. Indeed, you are encouraged to read beyond the standard teaching materials. However, *all* such sources *must* be cited fully and, apart from code provided by the teaching staff, you must *not* copy code.

Finally. *Despite all these stern words, we are here to help!* There is information about getting help in this subject on the LMS pages. Frequently asked questions about the Assignment will be answered in the LMS discussion group.