Master's thesis

Efficiency of Self-Adjusting Heap Variants

LM Hartmann

Freie Universität Berlin

October 2020

E-Mail address:	maria.hartmann@fu-berlin.de
Student Number:	5039594
Target Degree:	Master of Science
Subject:	Computer Science
Advisor:	Prof. Dr. László Kozma
2nd Advisor:	Prof. Dr. Wolfgang Mulzer

This copy is an edited version of the thesis that was submitted originally. The following changes were made: Minor spelling corrections in the "Acknowledgements" section; reformatting from US letter format to A4 format.

Statement of authorship

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Maria Hartmann

Date

Acknowledgements

This has been a challenging time to write my thesis, and I would like to thank all those who have supported me along the way. First and foremost, I wish to thank Prof. Dr. László Kozma, who has been a wonderfully encouraging and supportive thesis supervisor. He has also suggested that I flaunt my Latin skills more, so

> Gratias tibi propter opportunitatem auxiliumque volo.

Furthermore, I would like to thank the friends and family members who have listened to me during moments of frustration, and whose steadfast encouragement has been immensely helpful. I am also grateful to everyone who has read previous drafts of my thesis and offered suggestions.

Contents

Abstract vi

1 Introduction and Outline 1

2 Description of Heaps 3
2.1 Pairing Heap Variants 4
2.1.1 Standard Pairing Heap 4
2.1.2 Stable Standard Pairing Heap 4
2.1.3 Front-To-Back Pairing Heap 5
2.1.4 Stable Front-To-Back Pairing Heap 5
2.1.5 Back-To-Front Pairing Heap 5
2.1.6 Stable Back-To-Front Pairing Heap 5
2.1.7 Multipass Pairing Heap 5
2.1.8 Stable Multipass Pairing Heap 5
2.1.9 Lazy Pairing Heap 5
2.1.10 Stable Lazy Pairing Heap 6
2.2 Smooth Heap Variants 6
2.2.1 Smooth Heap 6
2.2.2 Non-Stable Smooth Heap 7
2.3 Known Running Time Bounds 7

3 General Considerations for Experiments 8
3.1 Implementation 8
3.2 Experimental Questions 8

4 Experiments in Sorting Mode 9
4.1 Uniformly Random Permutations 10
4.2 Classes of Semi-Random Data 10
4.2.1 Continuous Sorted Subsequences 10
4.2.2 Non-continuous Sorted Subsequences 11
4.2.3 Localised Permutation 13
4.2.4 Near-Neighbour Permutation 14
4.3 Jordan Permutation 14
4.4 Experiments and Results 17
4.4.1 Uniformly Random Permutations 18
4.4.2 Continuous Sorted Subsequences 19
4.4.3 Non-Continuous Ascending Sorted Subsequences 22
4.4.4 Localised Permutation 23
4.4.5 Near-Neighbour Permutation 24
4.5 Jordan Permutation 25
4.6 General Observations and Hypotheses 25

5 Priority Queue in Dijkstra's Algorithm 26
5.1 Implementation 27
5.2 Experiment: Graphs of Variable Size 27
5.3 Experiment: Graphs with Variable Connectivity 28
5.4 Experiment: Smooth Heap Variants 28
5.5 General Results 31

5.6 Conclusion and Outlook . . . 31

6 Decrease-Key in Sort Heap Does Not Take $O(\log\log n)$ Time **31**
6.1 Statement of Original Result . . . 32
6.2 Implementation . . . 32
6.3 Potential Analysis . . . 32
6.4 Flaw in the Proof . . . 33
6.5 Counterexample . . . 33
6.6 Corrected Analysis . . . 35

7 Stable Linking Can Always Be Done Sequentially **35**
7.1 Problem . . . 35
7.2 Algorithm and Proof . . . 36

8 Appendix **40**

List of Figures

1 Stable and non-stable linking of three heaps . . . 4
2 Pseudocode of smooth heap consolidation given by Kozma et al. [7]. The operation link(x) here denotes the stable-linking of x to its right neighbour. . . . 7
3 Sample outputs of continuous sorted subsequence generator. Note the shorter subsequence of length 12 in both cases. . . . 11
4 Sample outputs of non-continuous sorted subsequence generator. . . . 12
5 Sample outputs of localised-permutation generator for fixed list length and varying values of standard deviation. . . . 13
6 Sample outputs of near-neighbour-permutation generator for fixed list length 512 *and* fixed standard deviation. . . . 14
7 Top left: the Jordan curve that generates J_3. Bottom: The top left curve folded on itself generates J_4. Top right: folding the x-axis back over the top-left curve is equivalent to the bottom curve; it generates the same Jordan permutation. . . . 15
8 Sorting a uniformly random permutation with different heap variants. Size of input list varies from $n = 100$ to $n = 10000$ in increments of 100. Sorting on each size was performed five times on different permutations and results were averaged. 18
9 Overall number of linking operations and comparisons needed to sort permutations of the class of continuous sorted subsequences. . . . 20
10 Overall number of linking operations and comparisons needed to sort permutations of the class of non-continuous sorted subsequences. . . . 22
11 Overall number of linking operations and comparisons needed to sort permutations of the class of localised permutations. . . . 23
12 Overall number of linking operations and comparisons needed to sort permutations of the class of near-neighbour permutations. . . . 24
13 Number of comparisons required by different heap variants to sort Jordan permutations of length up to 2^{17}. . . . 25
14 Number of link operations and comparisons needed to find shortest paths in random graphs of variable size. . . . 27
15 Number of link operations and comparisons, respectively, needed to find shortest paths in random graphs of fixed size and variable connectivity. . . . 28

16 Number of link operations and comparisons, respectively, performed by different smooth heap variants to find shortest paths in random graphs of variable connectivity. . . . 30
17 Counterexample for the proof. . . . 34

Abstract

The *pairing heap* is an efficient, easy-to-implement self-adjusting heap which is in widespread practical use. Multiple variants of this heap have been proposed in the original paper by Fredman, Sedgewick, Sleator and Tarjan, but these have proven difficult to analyse. Furthermore, a related heap – the *smooth heap* – has recently been proposed and partial theoretical analysis of this heap appears promising. *Stable linking*, which is a twist on the *linking* operation used by pairing heaps, is a characteristic operation of this heap. This operation can be retrofitted into the pairing heap, giving rise to further heap variants. We implement these variants and measure efficiency by counting the number of link operations and the number of comparisons performed overall in various experimental settings, viz. in *sorting mode* and when used as priority queues in Dijkstra's algorithm for shortest paths.

The results suggest that the original variant of the pairing heap is generally superior to its other variants, and that the smooth heap may outperform the pairing heap and its variants in certain cases. The smooth heap performs particularly well in sorting mode – when sorting inputs which contain many sorted subsequences – and perhaps also in use cases where the number of `decrease-key` operations dominates the others. In these cases, smooth heaps perform fewer comparisons and fewer link operations than the pairing heap and its variants. Furthermore, in more general settings the smooth heap appears to perform fewer link operations than all pairing heap variants, at the cost of requiring slightly more comparisons. Depending on the relative cost of linking operations and comparisons this might mean that the smooth heap outperforms the pairing heap in practice even in general cases.

It is also shown that none of the heaps considered here are capable of sorting *Jordan permutations*, a special permutation class, in linear time.

Furthermore, two theoretical results for related heaps are shown: First, that the *sort heap*, proposed by Iacono and Özkan [6] as an instance of their *pure heap model* which has amortised cost $O(\log \log n)$ `insert` and `decrease-key` operations and $O(\log n \log \log n)$ amortised `delete-min`, does not in fact conform to those worst-case bounds, because the proof of this bound for `decrease-key` is flawed.

Finally, we show a property of instances of the *stable heap model*, to which the smooth heap conforms. We prove that the order in which `stable-links` are performed during consolidation of the forest into a single heap is not relevant for the outcome, because any resulting tree can also be constructed using a sequence of stable-links which requires only $O(n)$ pointer moves.

1 Introduction and Outline

In 1986, Fredman, Sedgewick, Sleator and Tarjan proposed the pairing heap as a more practical – i.e. easier to implement – alternative to previous efficient self-adjusting heaps such as the Fibonacci heap. They showed in the same paper that the amortised running time bounds nearly match those of the Fibonacci heap, making the pairing heap that desirable combination of efficient theoretical performance and practical usability. Consequently, implementations of the pairing heap have been in widespread use for some time.

In addition to the main pairing heap variant that is in use today, Fredman et al. also suggested several ideas for modifications to the pairing heap in the original paper. While fairly competitive running time bounds of $O(1)$ amortised running time for `make-heap` and `find-min` and $O(\log n)$ for all other heap operations have been shown for the main variant[2],[1] (which we shall henceforth refer to as the *standard pairing heap* to differentiate it from the other variants), less is known about the efficiency of the four proposed variants. In light of the success of the standard pairing heap, it would certainly be interesting to bound the efficiency of its variants; however this task has proven difficult. There have been partial results on generalised pairing heaps[1], but it is clear that the analysis of this general model is not straightforward either. Therefore, since it appears unlikely that a comprehensive theoretical analysis of pairing heap variants will be forthcoming soon, we attempt to gain insight by performing an experimental study. Given the practical use of the standard pairing heap, an experimental comparison of the standard version with its variants might provide actionable results if a non-standard variant is shown to possess an advantage over the standard variant in certain cases. Furthermore, results might give insight into heap behaviours to fuel renewed attempts at theoretical analysis, or at least serve to redirect efforts towards the variants that appear most promising.

Beyond the pairing heap, Kozma and Saranurak have recently proposed the *smooth heap* [7], which shares similarities with the pairing heap and its variants. A partial running time analysis of this heap appears promising enough that its performance might rival that of the standard pairing heap, but to our knowledge this has never been tested in practice. In fact, Kozma et al. suggest that the performance of smooth heap in sorting mode may be *instance-optimal*, i.e. optimal within a constant factor, among stable heaps, a fairly broad range of heaps that share some of its characteristics.

The pairing heap and its variants are fundamentally based on an atomic operation called *linking*, which connects two neighbouring nodes in a list. Neighbours are repeatedly *linked* together to create and maintain a tree or forest structure. The smooth heap works similarly, but features a slightly modified version of this linking operation called *stable linking*. The two operations can be interchanged without any further modification of data structures, which leads to the question how stable-linking and non-stable linking heaps compare.
Therefore we compare not only the performance of the pairing heap, its variants and and the smooth heap, but also introduce additional variants by exchanging stable and non-stable linking in all possible manners.

Furthermore, it has been shown [6] that there is a lower bound of $\Omega(\frac{\log\log n}{\log\log\log n})$ on the amortised cost of the `decrease-key` operations for all heaps that conform to the so-called *pure heap model*, whose instances are related to the pairing heap. This lower bound was later improved to $\Omega(\log\log n)$ [5]. The standard pairing heap and some of its variants conform to this model, but do not or are not known to match the lower bound. In fact, the only known heap that claims to match this lower bound so far is the *sort heap*, which is proposed as an example in the same paper. However, we will demonstrate that the analysis of this heap contains a flaw and that therefore its claim about the amortised cost of `decrease-key` is not true. This re-opens the

optimality question for this heap model, because there no longer exists a proof that the lower bound on decrease-key is at most $\Omega(\log\log n)$.

2 Description of Heaps

All types of heaps that we will discuss here conform to the following heap interface:

> `insert(x)` inserts element x into the heap
>
> `make-heap()` creates a new empty heap
>
> `delete-min()` removes the item with the smallest key from the heap, restructures the heap if necessary, and returns the removed minimum
>
> `merge(h)` merges the heap with heap h
>
> `decrease-key(`$x; \Delta$`)` decreases the key of node x, assumed to be in the heap, by Δ.

All heaps considered here are implemented as trees of nodes or as forests of such trees. Each tree observes the minimum heap property, i.e. any node in a tree has a key greater than or equal to the key of its parent. We will use a node and its key interchangeably in cases where a differentiation between the two is not necessary. Furthermore, all heap variants in these experiments employ a sub-operation that we call *linking*. Linking is the basic operation that connects two separate heap items with one another according to certain rules. In the "normal" (i.e. *non-stable*) variant, the keys of two root items x, y are compared and the item whose key is greater becomes the leftmost child of the other.

In the *stable* variant of the linking operation (we will call this *stable linking*), the order of the nodes prior to linking determines where the new child is added relative to any pre-existing siblings:
Let $x; y$, in that order, be the nodes to be linked. If $x:key \quad y:key$, then y becomes the rightmost child of x. Otherwise $x:key > y:key$ and x becomes the leftmost child of y. See Figure 1 for an illustration of both linking variants. Note that we use *linking* and *non-stable linking* to refer to the same operation, depending on emphasis.

The *pairing heap* and its variants were proposed[2] by Fredman et al. This original paper focuses on the description and analysis of one data structure which we will call the *standard pairing heap*, but in addition to this they describe four more heap variants based on the same underlying idea of pairwise linking; these are called the *front-to-back pairing heap*, the *back-to-front pairing heap*, the *multipass pairing heap* and the *lazy pairing heap*. We will discuss the heap and its variants in more detail below.

The *smooth heap*, introduced by Kozma et al. [7], also uses a pairwise linking strategy, but performs *stable linking*, where the order of the two elements prior to linking dictates the outcome of the link. Note that it is possible to interchange stable and non-stable linking in all previously mentioned data structures without the need for any additional modifications. In doing so, we obtain five additional stable variants of the pairing heap corresponding to the non-stable ones described above; we shall refer to these by their previously established name with *stable* prefixed, e.g. *stable front-to-back pairing heap*. Inversely, we can also replace the stable linking in the smooth heap with non-stable linking to obtain a *non-stable smooth heap* variant. If a heap variant is referenced without *stable* or *non-stable* prefixed, this refers to the established variant, i.e. the non-stable variant for pairing heaps and the stable variant fro the smooth heap.

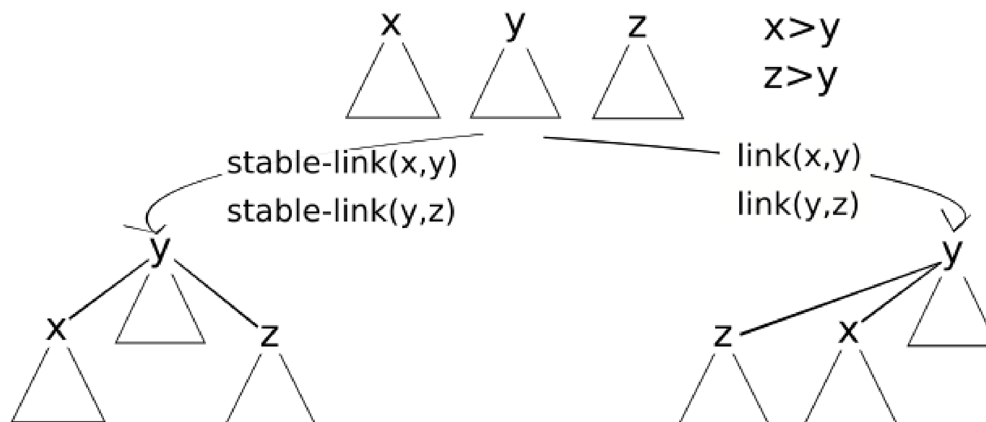


Figure 1: Stable and non-stable linking of three heaps

2.1 Pairing Heap Variants

2.1.1 Standard Pairing Heap

In the *standard pairing heap*, a single tree of items with the heap property is maintained at all times. There is no limit on the number of children of each node. Heap operations are defined as follows:

insert(x) A new node x is inserted by connecting it to the root of the tree with a non-stable linking operation. This means that x either becomes the new root and parent of the previous root (if it is smaller than the previous minimum) or otherwise it becomes the new leftmost child of the root. Previous children of the root remain unchanged.

delete-min() This operation consists of three phases. First, the current minimum (the root of the tree) is removed, leaving its children as k disjoint subtrees. Then, in the *pairing phase*, a single left-to-right pass is performed over these orphaned subtrees, linking separate pairs of neighbouring siblings together into $k/2$ new disjoint subtrees. Finally, the *combining pass* consolidates these subtrees by repeatedly linking the current rightmost tree to its neighbour until only one tree remains. Because this combining pass begins with the rightmost tree and then repeatedly adds the tree to the left of the combined tree, this is called left-to-right combining.

merge(h) The two heaps are merged by linking their respective roots together.

decrease-key(x, Δ) The first parameter points to the node x in the heap. This node is cut out of the tree from its current position, along with the subtree rooted at x. Then the key of x is decreased by Δ and the subtree is merged with the remaining tree (that is, x is linked to the root of the tree).

2.1.2 Stable Standard Pairing Heap

All operations are equivalent to those described for the *standard pairing heap* with the exception that all linking operations are replaced by stable-linking operations.

2.1.3 Front-To-Back Pairing Heap

This variant also maintains a single heap-ordered tree. `Insert` and `merge` are defined exactly as for the standard pairing heap. `Delete-min` again consists of the extraction of the root, a single pairing pass on the orphaned children of the removed node and consolidation by combining the remaining subtrees, but in contrast to the standard variant the pairing and combining pass both move left-to-right on the list of subtrees. Both consolidation steps may be combined in a single pass. The `merge` and `decrease-key` operations also remain unchanged compared to the standard pairing heap.

2.1.4 Stable Front-To-Back Pairing Heap

This is defined as the non-stable front-to-back pairing heap, with all linking operations replaced by stable linking operations.

2.1.5 Back-To-Front Pairing Heap

This heap may also be defined relative to the standard pairing heap. Again, a single tree is maintained and the `insert` and `merge` operations are the same as in the standard pairing heap. The only difference in the `delete-min` operation is that the pairing and combining pass are both performed right-to-left. `Merge` and `decrease-key` remain unchanged.

2.1.6 Stable Back-To-Front Pairing Heap

Equivalent to the back-to-front pairing heap with all linking operations replaced by stable linking operations.

2.1.7 Multipass Pairing Heap

This variant also shares the definitions of `insert, merge, and decrease-key` with the standard pairing heap. In the `delete-min` operation, the combining pass is omitted and replaced by repeated left-to-right pairing passes until the orphaned subtrees have been consolidated into a single tree.

2.1.8 Stable Multipass Pairing Heap

This variant is equivalent to the multipass pairing heap with all linking operations replaced by stable linking.

2.1.9 Lazy Pairing Heap

In this version of the pairing heap, a forest of trees (all satisfying the heap property) in the form of a root-node top-list is maintained instead of just the one heap-ordered tree of previous variants. Insertion is performed by concatenating a new one-node tree containing the new key x to the current list. Two heaps are merged by simply concatenating their respective top-lists. Every execution of `delete-min` performs a single left-to-right pairing pass over the list of trees, linking distinct pairs of neighbours together and keeping track of the smallest root element encountered during the pass. Due to the heap property of all trees in the forest, this must yield the absolute minimum after completing the pairing pass. Finally, this minimum is deleted and the resulting orphaned subtrees are concatenated to the top-list. `Decrease-key` is performed by cutting the relevant node and its subtree from its current position in a tree, decreasing the key of the node and adding the new subtree to the forest.

2.1.10 Stable Lazy Pairing Heap

All operations are defined identically to those in the non-stable lazy pairing heap with the exception that `delete-min` performs stable linking instead of non-stable linking.

2.2 Smooth Heap Variants

2.2.1 Smooth Heap

The smooth heap is somewhat different from the pairing heaps defined above. Like the pairing heap, it maintains all elements in a single tree that satisfies the heap property, but – unlike most of the pairing heap variants – its consolidation strategy does not include a pairing or a combining phase, but instead seeks out local maxima in the list of keys and begins linking around such maxima. Furthermore, the smooth heap uses stable linking instead of non-stable linking by default.

The `insert` operation is defined similar to that of the stable standard heap: the element to be inserted is stable-linked with the current root of the tree. Two heaps are `merged` by stable-linking their respective trees.

`Delete-min` deletes the current root, leaving an ordered top-list of orphaned children. These are consolidated as follows: The top-list is traversed from left to right until a local maximum is encountered, i.e. a node which has no immediate neighbour in the top-list whose key is greater. If this node has only one neighbour, it is stable-linked with that neighbour. Otherwise it is stable-linked with the greater of its two neighbours. Then the search for local maxima continues, starting from the root of the newly linked tree until only one node remains in the top-list or the end of the list is reached. If only one node remains, the consolidation is complete; otherwise the rightmost node in the top-list is a local maximum and the remaining top-list is sorted by keys in ascending order. In this case the consolidation can be completed by one final right-to-left combining pass using stable linking. Consider also the pseudocode of this consolidation in Figure 2.

In one of the experiments we will later consider different implementations of `decrease-key`, but the original definition is the same as for the stable standard pairing heap: the affected node and its subtree are detached from its current position in the tree, the key is decreased and the resulting subtree is linked to the root of the remaining tree.

```
        Input: A list of nodes, with x initially pointing to the leftmost node
(LTR)   while x.next ≠ null do
            if (x.key < x.next.key) then x ← x.next
            else                                              ▷ Local maximum found
                while (x.prev ≠ null) do
                    if (x.prev.key > x.next.key) then
                        x ← x.prev
                        link(x)
                    else
                        x ← x.next
                        link(x.prev)
                        continue (LTR)
                link(x)
(RTL)   while x.prev ≠ null do
            x ← x.prev
            link(x)
```

Figure 2: Pseudocode of smooth heap consolidation given by Kozma et al. [7]. The operation `link(x)` here denotes the stable-linking of x to its right neighbour.

2.2.2 Non-Stable Smooth Heap

This is a variant of the smooth heap described in the previous section, where the stable linking operation is replaced by non-stable linking.

2.3 Known Running Time Bounds

In the original paper on pairing heaps it is shown that the standard pairing heap performs `make-heap` and `find-min` in $O(1)$ amortised time and all other heap operations in $O(\log n)$ amortised [2]. It has been conjectured that `insert`, `decrease-key` and `meld` can also be performed in $O(1)$ time amortised, but this is now known to be false, as Fredman has shown that `decrease-key` can take $\Omega(\log\log n)$ time in generalised pairing heaps [1]. Furthermore, it has been shown (also [2]) that all heap operations of the multipass pairing heap as well as the lazy pairing heap take $O(\log n \log\log n = \log\log\log n)$ amortised time and that `delete-min` has an amortised bound of $O(\sqrt{n})$ in all (non-stable) pairing heap variants. Additional improvements on the upper bounds of the back-to-front pairing heap and the multipass pairing heap exist, but neither of these are known to be tight.

The original paper on smooth heap [7] contains an analysis of its performance in *sorting mode*, i.e. when using a heap to sort an input list in ascending order by inserting all elements and removing the minimum one-by-one until the heap is empty. It is conjectured that the smooth heap is instance-optimal among *stable heaps* in sorting mode.

3 General Considerations for Experiments

3.1 Implementation

All heap variants were implemented in an object-oriented fashion in Python 3 as implementations of the same generic heap interface. This heap interface defines the functions that each heap must implement, that is the operations `make-heap`, `insert`, `delete-min` and `merge`. Because some heap variants also support `delete` or `find-min` in their definition, these are included in the heap interface as well. Note, however, that these two operations are not implemented consistently across all heaps, and that they are not used in the experiments.

Each implementation of a heap operation maintains two counters: one of the total number of links performed while executing the operation and one of the number of comparisons. For some variants of the pairing heap these numbers are always the same, i.e. the only comparisons performed by the heap are the single ones that decide on the outcome of a single linking operation. In this case the respective heap operations return only this one number (in addition to any return value this operation might have in the regular heap interface); otherwise they return a two-tuple containing in first place the number of comparisons performed during this operation and the number of link operations in second.

Pairing heaps are implemented as heaps of instances of a `Node` class. This class defines a `key` field and pointers `leftChild, rightChild, parent, prevSibling` and `nextSibling`. The `key` field must be filled in all instances of `Node`, but of the pointers only those should be used that are necessary for the respective implementation. The pairing heap variants described in [2] are constructed there to require as few pointers as possible, which leads to different pointer structures in different heaps. The heaps were implemented to adhere to the theoretical description as closely as possible, so owing to the different structures there are different pointers in use for different variants. Defining all possible pointers in the `Node` class enables us to use the same class across all heap implementations.

A single general implementation of the pairing heap interface bundles all pairing heap variants by assigning a numerical identifier to each. When this pairing heap is instantiated with a heap type identifier given as a parameter, it instantiates a pairing heap of this type, defers all calls to its heap operations to this instance internally and passes on its return values. In other words, this heap can effectively act like any one of the implemented variants. A second parameter may be passed to this general heap during instantiation to specify whether heap operations return the link count or the comparison count. If this parameter is not defined, the heap always returns the comparison count.
All implementations are accessible in a dedicated git repository [3].

3.2 Experimental Questions

The experiments were designed to investigate multiple questions:

To compare performances of the smooth heap, the standard pairing heap and the four variants of the pairing heap on general input lists. Since the smooth heap has – to our knowledge – never been implemented before, but the theoretical analysis appears promising, it is interesting to see how well it performs against the pairing heap in practice. Furthermore, the pairing heap variants appear difficult to analyse theoretically (though progress has been made), so an experimental comparison of these variants is warranted. There have been experimental studies involving pairing heap before (e.g. [8]), but these have not included all pairing heap variants. Since the smooth heap has been introduced recently, it has also not been involved in experimental comparisons to date, nor has the concept of stable versus unstable linking.

To explore the effects of stable versus unstable linking in these data structures. The original linking operation used in the pairing heap links two nodes together so that one node always becomes the leftmost child of another. The outcome of the linking operation is only dependent on the keys of both nodes. As mentioned before, the smooth heap utilises a modified version of this operation, where the order of the two nodes involved in the linking operation influences how the nodes will be linked together. The stable and non-stable linking operations can be exchanged for each other in all heap variants without requiring any additional modification of the definition of each data structure. This allows us to study whether stable linking or non-stable linking have a significant impact on heap performance; particularly whether a potential comparable or even superior performance of the smooth heap may be explained thusly.

To compare the performance of heap variants on input lists that are ordered to some degree. It is conceivable that the design of different heap variants is particularly well suited to input lists with a certain structure, for example partially ordered lists. Since this is not always obvious from theory, we generate different classes of structured inputs experimentally and compare the performance of heaps on these inputs. This might lead to practical rules of thumb on which heaps to utilise for certain inputs; perhaps even provide a new theoretical perspective.

The main intention of these experiments is to compare the performance of different heap variants relative to one another, not to show absolute bounds. Many experiments which compare data structures in practice measure efficiency in terms of the real elapsed time during an operation. We, however, choose to count instead the exact number of comparisons and link operations that are performed. The reason for this is that we are not only attempting to compare the use-case performance of these heaps, but also to gain further theoretical insight into data structures which are not fully understood. The choice to count the number of comparisons then is a natural one, as bounding the number of comparisons is a common focus of running time analysis. The number of linking operations is also counted, because (1) we consider variants of the smooth heap in our experiment which involve sorting, and there the number of comparisons and the number of links are not necessarily proportional, and (2) while the number of linking operations may be proportional to the number of comparisons for all heaps in sorting mode and for most in general, these proportions may differ between heap variants, and these differences in itself are interesting. We do not include pointer changes because these are proportional to the number of comparisons in all heaps when implemented carefully.

4 Experiments in Sorting Mode

The first set of experiments considers the performance of heaps in a particular scenario – the *sorting mode*. In this special case, heaps are used to sort an input list of length n: the list is traversed left-to-right and elements are inserted into a heap in the order in which they are encountered. Once n `insert` operations have been performed (i.e. all elements have been inserted), a sequence of n `delete-min` operations is performed on the resulting heap. Maintaining a list of keys in the order in which they are removed finally results in the list sorted in ascending order. Note that this special case never makes use of the `decrease-key` operation, meaning we are able to evaluate the performance of just the `insert` and `delete-min` operations separately in this first experiment.

In the original paper on the smooth heap, Kozma and Saranurak conjecture that the smooth heap is an instance-optimal stable heap in sorting mode. Furthermore, it is shown that smooth heaps perform well on certain specific types of input, and it is hypothesised that this may extend to other structured inputs as well. Our experiments may show how well the smooth heap performs on such inputs in practice. Moreover, nothing is known about the constant factors involved in

the running time of smooth heap on those inputs for which theoretical bounds have been proven. We will see in our experiments whether the performance of smooth heap holds up in practice compared to pairing heaps. To investigate these questions we perform experiments in sorting mode not only on uniformly random inputs, but also on *semi-random* input lists.

For our purposes, we call any permutation of a sorted list *semi-random* that was generated to have a structure beyond uniform randomness. We define and explore several different classes of such permutations. For each class of semi-random permutations we define a parameterised generator function, each dependent on some "randomness parameter" whose value influences the "degree of randomness" of the resulting permutation. The type of this parameter is different for each generator function. Classes of semi-randomness and the associated parameters are discussed in detail in the next section.

Each sorting mode experiment generates a random or semi-random permutation of an initially sorted list containing unique integer elements $1, \ldots, n$, with n variable or fixed. For each value of the respective experiment parameter, one run of the experiment performs input and extraction of the same permutation on each heap type in turn. For each variable parameter in each input class, the result (i.e. the overall number of linking operations or comparisons required for sorting the input) is the average of five such runs. The full experiment then consists of repeated test runs determining an average number of comparisons and linking operations for one variable parameter.

Once the experiment has concluded, the counts for each heap are plotted over the respective parameter values.

4.1 Uniformly Random Permutations

The input is a uniformly random permutation of a list containing unique integer elements 1 to n. One test run performs input and extraction of the same permutation on each heap type in turn. length n of the input list.

4.2 Classes of Semi-Random Data

We define several different classes of semi-random data, each representing a type of permutation that is generated using randomness to some degree, but about whose structure more information remains known than about a uniformly random permutation. In contrast to the experiment on uniformly random data, we choose a fixed list size for these experiments and run the sorting test on variable randomness parameter values instead. For our purposes, all instances of any class are assumed to be lists of pairwise distinct elements $1, \ldots, n$.

4.2.1 Continuous Sorted Subsequences

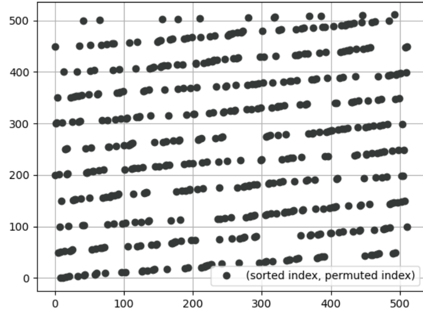
An instance of length n of this class is a permutation that, for a given subsequence length k, consists entirely of a sequence of continuous sorted subsequences of length k. All subsequences are sorted in the same order, i.e. either all subsequences are in ascending order or all subsequences are in descending order. On occasion this class may be partitioned into two subclasses of sequences sorted in ascending and descending order, respectively, to be considered separately.

In our experiments we implement a generator for permutations of this type by first shuffling the input data uniformly at random, then partitioning this random permutation into subsequences of the given length and sorting them. If the overall length of the list does not divide evenly by the subsequence length, the last sorted subsequence of the permutation will be shorter than the

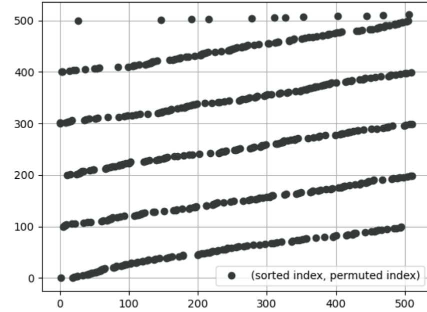
rest.

Algorithm 1: generateContinuousSubsequencePermutation((A, k))

Data: sorted list A, subsequence length k
Result: random permutation of class of continuous subsequence permutations
i 0
while $i < A.length$ **do**
    sort elements at indices $[i, min(i + k \quad 1, n)]$ in A in-place
    i=i+k
end
return A



Continuous sorted subsequence permutation of length 512 and subsequence length 50.



Continuous sorted subsequence permutation of length 512 and subsequence length 100.

Figure 3: Sample outputs of continuous sorted subsequence generator. Note the shorter subsequence of length 12 in both cases.

Figure 3 shows two different outputs (both in ascending order) of the permutation generator for continuous sorted subsequences. The x-axis represents the index of each element in the sorted list and the y-axis is the index of the same element in the permutation. The individual subsequences can be clearly seen.

4.2.2 Non-continuous Sorted Subsequences

An instance of length n of this class is a permutation that, for a given subsequence length k, consists of an intermix of sorted subsequences of length at least k. All subsequences are sorted in the same order, either ascending or descending.

In our experiments a generator for this type of permutation was implemented to work in two stages: First a permutation of the class of continuous sorted subsequences is generated, then in the second step these subsequences are intermixed while preserving the internal order of each subsequence.

Let $x_{i,j}$ be elements of the list, where $0 \quad i \quad \lceil n/k \rceil$ denotes the number of the continuous sorted subsequence that contains the element in this permutation and $0 \quad j \quad k$ the index of

the element in the i-th subsequence. We define a randomised key generator for elements $x_{i,j}$:

$$\kappa(x_{i,j}) = \begin{cases} p_{i,j} & \text{if } j = 0 \\ \kappa(x_{i,j-1}) + p_{i,j} & \text{if } n \bmod k > 0 \wedge i = \lceil n/k \rceil \\ \kappa(x_{i,j-1}) + \alpha p_{i,j} & \text{otherwise} \end{cases}$$

where $p_{i,j} \in [0, 1)$ is drawn uniformly at random for each element and $\alpha = n/(n \bmod k)$. Note that this factor is used only if $n \bmod k > 0$, so it is well-defined for these purposes. A permutation is now generated as follows:

Algorithm 2: generateNonContinuousSubsequencePermutation(A, k)

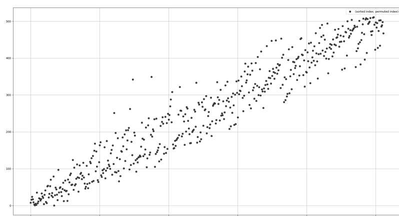
Data: sorted list A, subsequence length k
Result: random permutation of A s.t. the result consists of sorted non-continuous subsequences of length k

```
A' ← generateContinuousSubsequencePermutation(A, k)
foreach x in A' do
    x.key ← κ(x.value)
end
A'' ← A' sorted by keys
return A''
```

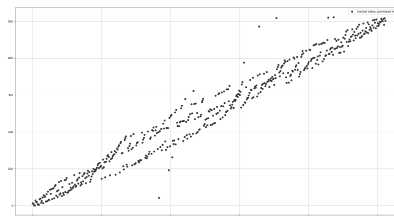
In other words, we assign a random key to each item in the list while maintaining the invariant that the key of an element must be larger than that of its predecessor in the same continuous sorted subsequence. We generate keys additively: The key for an item in a subsequence is generated by drawing a uniformly random number from the interval $[0, 1)$ and adding this number to the key of the item's immediate predecessor in the same subsequence. If the item is the first element of a subsequence, then the key is simply the result of the random draw. The list of elements is then re-sorted by these keys to obtain the final permutation.

We treat the case separately where there is one subsequence that is shorter than the rest, i.e. where the length of subsequences does not evenly divide the length of the entire permutation. If we were to use the same key generator scheme in this case, larger values from the shorter subsequence would be more likely to receive smaller keys and would so appear closer to the beginning of the list than comparable values in other subsequences. We counteract this by scaling the random value that is added when generating a new key by a factor of $\alpha = n/(n \bmod k)$.

The variable parameter on which the experiment is run is the subsequence length k.



Non-continuous sorted subsequence permutation with list length 512, $k = 50$.



Non-continuous sorted subsequence permutation with list length 512, $k = 100$.

Figure 4: Sample outputs of non-continuous sorted subsequence generator.

Figure 4 shows two permutations of fixed size and different standard deviation generated by this generator. It is possible to make out different ascending subsequences and where they intermix.

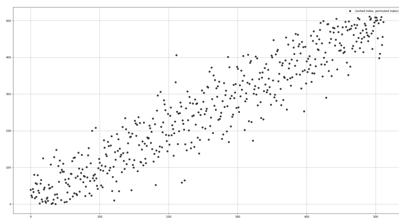
4.2.3 Localised Permutation

This type of permutation is characterised by the property that the indices of elements in the permutation are scattered around their original indices in the sorted list by a Gaussian distribution. A generator for this type of permutation was implemented as follows:

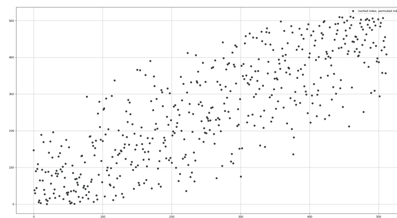
A key for each element of the list is drawn from a Gaussian normal distribution centred at the index of that element in the sorted list. The list is re-sorted by these keys to obtain the permutation.
Let $x_1, \ldots, x_n$ be the sorted input list. We define a random key generator function $(x_i) = p_i$ such that p_i is a random value drawn from the Gaussian normal distribution with mean $_i = i/n$ and variance given as an experimental parameter. The standard deviation of the Gaussian distribution is variable and is used as an experimental parameter to adjust the "randomness" of the generator output.

Algorithm 3: generateLocalisedPermutation($A,$)

Data: sorted list A, subsequence length k
Result: random permutation of class of localised permutations
foreach x *in* A **do**
  $x.key \leftarrow (x.value)$
end
$A' \leftarrow A$ sorted by keys
return A'



Localised permutation of length 512, = 0.1.



Localised permutation of length 512, = 0.2.

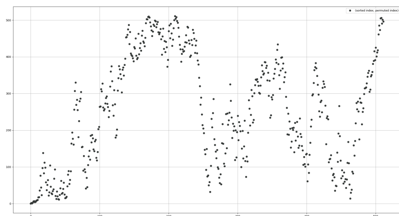
Figure 5: Sample outputs of localised-permutation generator for fixed list length and varying values of standard deviation.

Figure 5 shows two sample outputs of this generator. Both are lists of the same size, but the permutation on the left was generated with a smaller standard deviation than the one on the right. As the images demonstrate, this results in a greater spread of elements in the permutation.

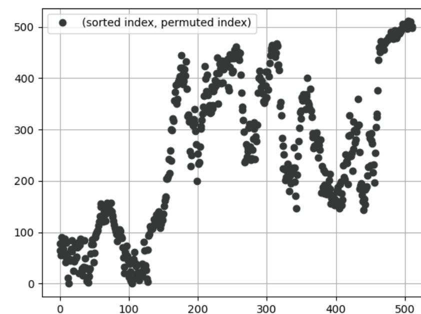
4.2.4 Near-Neighbour Permutation

Permutations of this type are characterised by the property that elements that are very close together in the sorted list are also likely to be close together in the permutation (but not necessarily close to their original indices).

In our implementation such a permutation is again generated by using a randomised key generator to reorder elements. Let $x_1, \ldots, x_n$ be the input list of sorted elements. The key generator is defined as $(x_i) = p_i$, where p_i is drawn from a Gaussian normal distribution with mean $p_{i\ 1}$ and standard deviation is passed as a parameter to the permutation generator.



Near-neighbour permutation of size 512, generated with $= 0.1$



near-neighbour permutation of size 512, generated with $= 0.1$

Figure 6: Sample outputs of near-neighbour-permutation generator for fixed list length 512 *and* fixed standard deviation.

Figure 6 shows two outputs of the generator for fixed list sizes *and* a fixed standard deviation. Permutation shapes are very different, but elements that are close in the sorted list remain close in the permutation. The x-axis represents the index of each element in the sorted list and the y-axis is the index of the element in the permutation.

In further experiments with the permutation generator (the results of which are not plotted here) it appears that a change in standard deviation has little effect on the overall distribution of the permutation.

4.3 Jordan Permutation

Let C be a Jordan curve, i.e. a closed curve in the plane which does not intersect itself. A *Jordan permutation* is obtained by numbering the intersections of C with the x-axis in the order in which they are encountered when following the curve trajectory. Hoffman, Mehlhorn, Rosenstiehl and Tarjan have shown that Jordan permutations can be sorted with a linear number of comparisons and a linear overhead [4]. In the same paper it is speculated that the same can be performed in a simpler way by using a splay tree in sorting mode. In 1995, this topic was taken up again by Callahan and discussed in a mailing list chain with Wilber [9], where they appear to work out a proof that sorting a Jordan permutation using a splay tree – or, in fact, any binary tree which uses rotations – requires more than $O(n)$ comparisons in the worst case. The proof is by counterexample; Callahan presents an algorithm to construct a series of Jordan sequences for which sorting is "hard" in the sense that it requires a superlinear amount of comparisons.

In the initial discussion of the problem, it is noted that the question could also be approached experimentally. This is what we shall do here to check whether the pairing heap or the smooth heap can sort a Jordan permutation in $O(n)$ comparisons in the worst case. We use the algorithm from Callahan's counterexample to generate valid Jordan permutations. As we will see, these sequences are sufficient to disprove linearity for sorting by pairing or smooth heap. Note that this input class generates a specific series of input lists deterministically; no randomness is involved.

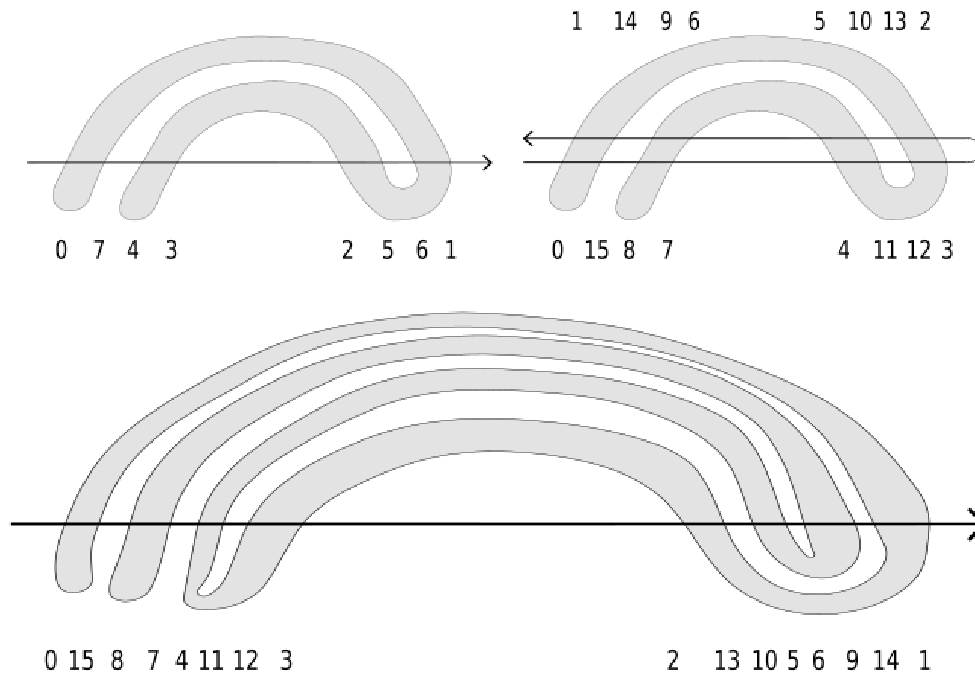


Figure 7: Top left: the Jordan curve that generates J_3. Bottom: The top left curve folded on itself generates J_4. Top right: folding the x-axis back over the top-left curve is equivalent to the bottom curve; it generates the same Jordan permutation.

From a geometric point of view, these sequences are generated by a curve which is repeatedly folded over on itself (see Figure 7). This is also the concept of the permutation generator: Starting from a base sequence $(0, 1)$, which is the shortest of this class of Jordan permutations, a longer sequence of length 2^n is generated as follows:

Algorithm 4: generateJordanPermutation(n)

Data: permutation length power n
Result: Jordan permutation of length 2^n from Callahan's class of counterexamples
if $n = 1$ **then**
| **return** $(0; 1)$
end
A generateJordanPermutation(n 1)
foreach *x in A* **do**
| x x 2
end
A A reverse(A)
foreach *x at odd index in A* **do**
| x $x + 1$
end
return A

Lemma 4.1. *Algorithm 4 generates a valid Jordan permutation.*

Proof. Let J_n be the Jordan permutation of length 2^n which is generated by generateJordanPermutation(n). We prove the claim by induction over n, by showing a Jordan curve that generates the permutation. **Base Case:** $n = 1$ $J_1 = (0; 1)$.

Clearly there exists a simple closed curve in the plane which generates this permutation. An example is a circle which is properly intersected by the x-axis.

Induction Step: n ! $n + 1$

Assume that J_n can be generated by a Jordan curve such as the one in Figure 7 generating the Jordan permutation. The claim is that J_{n+1} can be generated from the curve that is created by folding the Jordan curve that generates J_n over on itself. This is topologically equivalent to folding the x-axis upwards and back on itself so that it intersects the curve that generates J_n twice (we can assume that the upper part of the folded x-axis runs parallel to the lower part at an infinitely small distance, i.e. it intersects all the same folds of the curve that the lower part intersects). See Figure 7 for an illustration. This view makes it fairly easy to count the new intersections generated by the upper part of the x-axis. Now each intersection of the Jordan curve and the lower part of the x-axis has a corresponding intersection on the upper part of the x-axis. We consider two separate cases:

Case 1: The $(k+1)$-th intersection of the curve with the lower part of the x-axis is generated by the curve passing the axis in a downwards direction. Then, since the first intersection is an upwards pass of the curve through the lower axis and the axes are defined so that the curve must intersect both before turning back, the upper axis must have been intersected exactly $k + 1$ times before this intersection occurs. Because we start numbering the intersections from zero, the number of this intersection point in the Jordan curve J_{n+1} is $2(k + 1) \quad 1 = 2k + 1$.

Case 2: The $(k+1)$-th intersection of the curve with the lower part of the x-axis is generated by the curve passing it upwards. This means that the curve must have intersected the lower axis k times before, and the axes are defined so that the curve must have intersected the upper axis the same number of times to pass under the lower axis. Then the total number of intersections between the curve and the x-axis prior to this intersection is $2k$, so the number label of this intersection in J_{n+1} is $2k + 1 \quad 1 = 2k$.

We call a pair of intersection points, one with the lower axis and one with the upper axis, *associated* if one immediately follows the other along the trajectory of the curve. Note that each intersection has exactly one other associated. The number labels of the intersections with the upper axis can be defined with respect to the associated intersection with the lower axis: if the

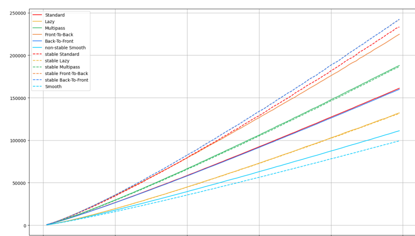
curve is passing upwards through the upper axis, the number of the intersection is the successor of the last (also upwards) intersection of the curve with the lower axis, i.e. $2k+1$ by case 2 assuming the last lower intersection was the $k+1$-th with the lower axis. Similarly, if the curve is passing downwards through the upper axis, the next intersection must be with the lower axis. If that next intersection is the $k+1$-th of the curve and the lower axis, the number label of the upper intersection is $(2k+1) - 1$ by case 1.

From the way the axis is defined, we know that intersections occur in the order *upwards through lower axis, upwards through upper axis, downwards through upper axis, downwards through lower axis*, repeated as many times as necessary. If we read off these labels in the order in which they appear on the axis, the i-th intersection along the lower part of the axis is associated with the intersection that appears in $(2^n - i)$-th place along the upper part of the axis. Now note that we defined the labels of the intersections with this folded axis in terms of the Jordan permutation J_n, which is generated by the intersections with only the lower axis. Let $J_n = x_0, \dots, x_{m-1}$ be the Jordan permutation of length $m = 2^n$. Putting the proof together, we have the first half of J_{n+1} as $2x_0, 2x_1 + 1, 2x_2, \dots, 2x_{m-1}, 2x_{m-1} + 1$ and the second half (reading along the upper part of the axis right-to-left) $2x_{m-1}, 2x_{m-1} + 1, \dots, 2x_1, 2x_0 + 1$. The final result is exactly the one produced from J_n by the algorithm: The first half is every element of J_n doubled, with elements at odd indices additionally incremented by one, and the second half of the result is the list created by doubling every element of J_n, reversing the resulting list, and again incrementing every element at an odd index in this list. Therefore we have shown how to find a Jordan curve that generates the same permutation as the algorithm, and so the output of the algorithm is a valid Jordan permutation. □

4.4 Experiments and Results

All experiments on semi-random data were performed on input lists of fixed size, typically $n = 10000$, with variable input to a randomness parameter that is specific to each class of permutations. Each parameter configuration was run five times and the overall number of links and comparisons required to perform a sorting of the input list were averaged. In each plot, heap types are marked by colour, with variants that use stable linking denoted by dashed lines and variants that use non-stable linking by continuous lines. Note that larger variants of all result plots are included in the appendix.

4.4.1 Uniformly Random Permutations



Average number of link operations.

Average number of comparisons.

Figure 8: Sorting a uniformly random permutation with different heap variants. Size of input list varies from $n = 100$ to $n = 10000$ in increments of 100. Sorting on each size was performed five times on different permutations and results were averaged.

In the number of comparisons, the standard variant and the back-to-front variant are effectively tied for the lowest number. This is followed by the non-stable smooth heap and the stable smooth heap variant, with the non-stable variant needing slightly fewer comparisons. The number of comparisons by the multipass and the stable multipass variants are again effectively the same.

The remaining heap variants, appearing in the order front-to-back, stable standard, stable back-to-front and stable front-to-back, lazy and stable lazy, perform significantly worse. Like their respective non-stable variants, the number of comparisons by the stable standard and the stable back-to-front heaps are quite similar. The lazy heap and its stable variant perform effectively the same.

We make two main observations – apart from the relative performance of heap variants – for this first experiment:

It appears that the direction in which a pairing pass is performed (front-to-back or back-to-front) has little effect on the eventual total number of comparisons in this case, but the direction of a combining pass does. The standard heap variant and the back-to-front variant differ only in the direction of the pairing pass and perform very similarly, as do the stable standard and stable back-to-front variants.

Contrastingly, for those heaps that use a combining pass, the direction of that pass seems to matter. The standard heap and the front-to-back variants are different only in the direction of their combining pass, but front-to-back needs significantly more comparisons than the standard variant.

This observation seems reasonable, because the combining pass has a much greater effect on the overall structure of the heap that is formed from the list of roots. After a single pairing pass, the overall structure is nearly the same regardless of the pass' direction: Half of all roots previously in the top list have been linked to another, and the other half have each gained exactly one new child. In fact, if the length of the top list was even, the same neighbours are linked with each other in either direction. If the length of the top list was odd, a single root will not have been linked during the pairing pass.

A single combining pass, however, determines the entire structure of the tree resulting from the top list, and here the direction of the pass can be crucial. At the first glance, it seems

that a front-to-back pass on a certain input list (of even length) should be equivalent to a back-to-front pass on the same list reversed, and that both variants should therefore perform the same on average on uniformly random input.

However, a closer examination shows that this is not so in the case of non-stable linking. The main observation to make is that in both cases non-stable linking links one child as the leftmost child of the other. Consider the individual linking operations during a front-to-back combining pass. From left to right, we compare the current root node with the leftmost node of that part of the top-list that has not been consolidated yet. Two nodes x, y in the top-list which become children of the same root r during front-to-back combining reverse their relative order: if x is left of y before the combining pass, it is encountered first during the pass and at that point becomes the leftmost child of r. When y is encountered, it is inserted to the left of all other current children of r, including x.

In a back-to-front combining pass the order of roots in the top-list is preserved if they become siblings. It is not obvious why this would influence the performance of the respective heaps as it does, but there is no other difference between the two heap variants which is not dependent on input. On the other hand, it *is* true that the stable front-to-back heap and the stable back-to-front heap are equivalent to each other if one is constructed on the reversed input of the other. Consequently, the experiment shows that both variants perform nearly the same number of comparisons and the same number of links on average over multiple uniformly random input lists. In fact both variants performed so similar in the experimental run whose results are shown here that their curves overlap almost completely in Figure 8.

Heap variants that perform a combining pass seem to be more affected by a change between stable and non-stable linking. The multipass variants and the lazy variants, which perform only pairing passes, show almost no difference in the number of comparisons between their stable and non-stable variants.

For the back-to-front and standard variants, which do perform a combining pass, the total number of comparisons is significantly higher for the respective stable variants. However, for the front-to-back variant the use of stable linking seems to make less of a difference

Furthermore, we observe that both smooth heap variants perform significantly fewer link operations than all pairing heap variants. The lazy pairing heap variants perform next-best in the number of links, beating all other pairing heap variants.

4.4.2 Continuous Sorted Subsequences

Recall that the variable parameter that was used for experiments on this class was the length of sorted subsequences.

Average number of link operations. Average number of comparisons.

Figure 9: Overall number of linking operations and comparisons needed to sort permutations of the class of continuous sorted subsequences.

Figure 9 shows the results of the sorting experiment on instances of the class of continuous sorted subsequences.

The first observation is that – for all heap variants except the two smooth heap variants and the stable standard pairing heap – the number of linking operations and comparisons decreases as the length of sorted subsequences increases. For the smooth heap variants and the stable standard heap, the number of operations does not decrease monotonously with increasing subsequence length, but the general trend is nonetheless towards fewer operations as there are sharp decreases at certain critical parameter values. This decreasing trend is to be expected, because longer subsequences mean that more of the input is already sorted and so less work should remain to be done by the data structure.

Neither the comparison curve nor the link curve are smooth, but instead decrease more strongly at certain parameter values. For some heap variants, this leads to "step-like" curves with a sharp decrease between two parameter values, while the change in others is much more gradual. It appears that the steps occur at points where the subsequence length divides the overall input length evenly, i.e. where all subsequences truly have the same length.

> In the comparison chart, the curves of the smooth heap, the non-stable smooth heap, the front-to-back heap and the stable back-to-front heap are markedly more "step-like" than the others. All others except the stable standard heap have flatter smooth arcs between these points. The stable standard heap is a special case in that its curve is much more erratic than the others, but its trend seems to be similar to that of the stable back-to-front heap. Multiple runs show that the curve zig-zags differently each time, so the exact trajectory of the curve is most likely not a pattern.
>
> For these very structured inputs, it is likely that the heap variants tend to form substructures which roughly correspond to the sorted subsequences. Then the more pronounced "step-like" behaviour could be interpreted to mean that in these heap variants the cost of interaction *between* these substructures dominates the cost of internal substructure changes. Then the number of substructures – that is, the number of sorted subsequences – would be more important than their respective sizes, and this explains the sudden drop in the number of operations when the number of sorted subsequences decreases.
>
> For all values of the subsequence length parameter, the smooth heap and the non-stable smooth heap outperform the others in the number of comparisons. The number of comparisons required by the smooth heap and the non-stable smooth heap are apparently the same or almost exactly the same at the optimal points, but if the subsequence length does

not exactly divide the overall list length, the non-stable smooth heap does slightly better than the stable variant the longer the shortest subsequence becomes.

In the number of comparisons the multipass heap performs next best for almost all values of subsequence length, followed by its stable variant. Exceptions are very short subsequence lengths (k 200 for the non-stable variant, k 300 for the stable variant, with $n = 10000$), where the non-stable back-to-front variant and the non-stable standard variant perform better, and subsequence lengths greater than half the list length, but roughly shorter than $(3=4)n$. In that range, the stable back-to-front variant and the non-stable front-to-back variant both do slightly better than the stable multipass variant, but worse than the non-stable version. For some subsequence lengths in this interval the stable standard variant also outperforms stable multipass, but this is not consistent.

The non-stable variants of multipass, front-to back and lazy consistently outperform their stable counterpart. For the standard and back-to-front heap versions, the picture is less clear: for smaller subsequence lengths (up to about $k = 2500 = n=4$, the observation is true for these variants as well, but from then on the stable versions begin outperforming their non-stable counterparts at critical points, and from about $k = 3400 > n=3$, the stable back-to-front heap uses consistently fewer comparisons than its non-stable version. The stable standard heap performs consistently better than the non-stable standard heap between $k = 5000 = n=2$ and $k = 8000$.

For the smooth heap variants, the multipass heap variants and the lazy variants, the respective stable and non-stable versions have relatively similar values and similar shapes for the number of comparisons, while for the other variants one of each stable/non-stable pair has the smoother curve version and the other has not.

Note that the number of linking operations performed during sorting is different from the number of comparisons only for the smooth heap and lazy heap variants; all other variants perform comparisons exactly when linking. Therefore the following discussion of the linking behaviour of heap variants will focus primarily on those four variants.

The curve of the number of linking operations performed by either smooth heap variant is noticeably smoother than the curve showing the number of comparisons of the same. This indicates that a substantial part of the links performed by the smooth heap variants are those which require more than one comparison, i.e. those around a local maximum which is not located at an endpoint of the current top-list.

Whereas the stable smooth heap needs slightly fewer comparisons than its non-stable variant (for long shortest subsequences), it uses slightly more links.

The smooth heap variants are overall best in both the number of link operations and the number of comparisons, but the lazy heap variants do not use much more linking operations than the smooth variants. (They do, however, need more comparisons than all other heap variants.)

4.4.3 Non-Continuous Ascending Sorted Subsequences

Average number of link operations. Average number of comparisons.

Figure 10: Overall number of linking operations and comparisons needed to sort permutations of the class of non-continuous sorted subsequences.

For this class of permutations the variable parameter is again the length of sorted subsequences. We consider subsequence lengths 100 k 10000, with step size 100.

We can observe the same trend in the number of comparisons as before: longer subsequences lead to fewer comparisons required for sorting. It seems that the "step-like" behaviour that was observed for the input class of continuous sorted subsequences appears here as well, as a sharper decline is visible at $k = 5000 = n$=2 for the stable standard heap, possibly the front-to-back heap and the stable back-to-front heap. However, this is somewhat difficult to tell because of the general erratic nature of these results.

All curves are much less smooth than for continuous sorted subsequences. This is to be expected, because it is likely that heaps perform better when sorting continuous subsequences, and the performance on instances of this permutation class depends on how sorted subsequences appear by chance.

The non-stable standard heap and the non-stable back-to-front heap behave very similarly in the number of comparisons, as they do on instances of the class of continuous sorted subsequences. They start off outperforming or at least matching all other heap variants for short subsequence lengths ($k = 100$ for $n = 10000$) – this matches the performances for uniformly random input permutations – and then do progressively worse relative to the other variants as subsequence length increases, until they are among the worst-performing for $k = n$. The relative performance of heaps on very long subsequence length is mostly similar to that on continuous sorted subsequences. This is reasonable, because for long subsequence lengths longer segments of the list become continuously sorted.

Unlike with continuous sorted subsequences, stable back-to-front and non-stable front-to-back do better than either multipass variant for longer subsequences. It is not clear whether this is an artefact of this particular run of the experiment or if this is due to structural causes.

4.4.4 Localised Permutation

Average number of link operations. Average number of comparisons.

Figure 11: Overall number of linking operations and comparisons needed to sort permutations of the class of localised permutations.

Recall that the position of each element in a permutation of this class is drawn from a Gaussian distribution centred around its index in the sorted list, and that the randomness parameter for this class is the standard deviation of that Gaussian distribution.

The number of comparisons increases for all heap variants as the value of the standard deviation increases. All curves in the plot are very smooth, including that of the stable standard heap variant, which

For this class of permutations, the relative performance of heaps seems quite settled already for small values of the randomness parameter. Beyond very small values of the standard deviation, the only change in the relative order of heaps from least to most comparisons for a given parameter values is at $0.075; n = 10000$, where the front-to-back variant and the stable front-to-back variant exchange places, while the stable standard variant remains between them in the order. The performance of all other heap variants remains in the same relative order throughout.

For this class of inputs, the standard pairing heap and the back-to-front pairing heap outperform both smooth heap variants and both multipass variants. This stands in contrast to the continuous and non-continuous subsequence classes, but resembles the relative heap performances on uniformly random permutations. At the maximum parameter value $= 0.2$, only the fact that the stable front-to-back heap performs slightly fewer comparisons than the stable back-to-front heap differs from the relative performance of heaps on uniformly random permutations. The performance of both would likely converge for a larger standard deviation.

This observation leads us to hypothesise that the advantage of smooth heap and multipass heap depends on the existence of sorted (not necessarily continuous) subsequences in the input, an idea which is further supported by the following experiment on the class of near-neighbour permutations.

The relative performance of heap variants for number of links also matches that of the experiment on uniformly random permutations: the non-stable smooth heap performs the lowest number of comparisons, followed by the stable smooth heap and the two lazy heap variants; the other heaps perform the same number of links as they perform comparisons.

4.4.5 Near-Neighbour Permutation

Average number of linking operations. Average number of comparisons.

Figure 12: Overall number of linking operations and comparisons needed to sort permutations of the class of near-neighbour permutations.

Permutations of this class are also generated by drawing keys from a Gaussian distribution, but for this class the Gaussian distribution is centred around the last previously drawn key. The randomness parameter is again the standard deviation of the Gaussian distribution.

We observe that for increasing values of the standard deviation there is little change in the number of comparisons required to sort the input. This is probably because, as we have noted before when describing this class of permutations, the value of this "randomness parameter" in fact seems to have little discernible effect on the generated instances. On further consideration we can see that this is due to the definition of the generator function: if the Gaussian distribution from which each key is drawn is centred at the value of the previous key, the standard deviation has no real effect on the relative spread of keys, only on their absolute difference.

On this class the stable smooth heap performs fewer overall comparisons than all other heap variants, followed fairly closely by non-stable smooth heap and stable multipass. This has similarities to the relative performances on the class of continuous sorted subsequences, but the non-stable multipass variant performs very differently here; it requires many more comparisons than its stable variant to sort the input.

The standard and front-to-back pairing heaps again perform very similarly and are next-best after the smooth heap variants and stable multipass. The stable back-to-front variant does nearly as well, but the stable standard heap performs much worse, which again contrasts with the performance on instances of the class of continuous sorted subsequence permutations. The non-stable multipass and stable front-to-back variants perform similarly to each other. The non-stable front-to-back heap does worse than its stable variant, but better than the stable standard variant. The two lazy variants perform the highest number of comparisons, but the non-stable variant does slightly better than the stable one – unlike for the multipass variants, even though both use a similar consolidation approach.

The stable back-to-front heap performs much better relatively to the other heaps than e.g. on the class of localised permutations, but the stable standard heap, which on that class behaves similarly to stable back-to-front, does not.

4.5 Jordan Permutation

The Jordan permutation generator was used to generate input lists of lengths 2^i, for $0 < i \leq 17$. These lists were sorted once by each heap variant. It is sufficient to perform only one sorting run for each permutation length because the generator algorithm is deterministic, as are the heap variants.

Figure 13: Number of comparisons required by different heap variants to sort Jordan permutations of length up to 2^{17}.

The results show by visual inspection that none of the heap variants is capable of sorting this type of input using only a linear number of comparisons. The superlinearity of results becomes more clear by overlaying a linear plot, but this is omitted here due to the limited figure size. Since we have seen before that the inputs we generate are valid Jordan permutations, we conclude that Jordan permutations cannot generally be sorted in $O(n)$ comparisons using pairing heap or smooth heap.

4.6 General Observations and Hypotheses

After discussing the results of the *sorting mode* experiments in detail, we can draw general conclusions with respect to our experimental questions.

Conclusion 4.1. *The choice of stable versus non-stable linking can have significant effects on heap performance in sorting mode, but we are unable make general recommendations on the use of stable linking versus non-stable linking.*

It appears that neither stable linking nor non-stable linking can be said to have an advantage over the other in the general case. However, both variants do not always perform equally well. Depending on the type of the heap and the structure of the input, one linking variant may perform significantly better than another. For example, on uniformly random input lists, the back-to-front heap and the standard heap perform much better using non-stable linking operations than stable ones. On inputs that are instances of our class of non-continuous sorted ascending subsequences, the same heaps perform better using stable linking for long subsequences.

Conclusion 4.2. *There is an indication for choosing smooth heap over all pairing heap variants on certain inputs in sorting mode, namely those that contain sorted subsequences. Furthermore, if the cost of linking dominates that of comparisons, then there is a much more general indication for choosing smooth heap.*

Both smooth heap variants perform very well when sorting any of our input classes that contains sorted subsequences, even if there are many sequences that are short relative to the overall length of the input. For the class of continuous sorted subsequences, both variants perform fewer comparisons than all others even for the smallest subsequence length ($k = 100$ for $n = 10000$, or 1% of the overall input length), and for the class of non-continuous sorted subsequences they do the same starting at subsequence lengths of about $k = 200$ for the stable-linking variant and $k = 300$ for the non-stable variant (2% and 3%, respectively, of the overall input length).Plotting instances of the near-neighbour permutation class suggests that these, too, contain many sorted subsequences, both in ascending and in descending order. Our experiments show that the smooth heap variants outperform all other heaps on this permutation type as well, requiring the lowest number of comparisons and link operations for sorting.

As we have observed before, both smooth heap variants consistently perform significantly fewer linking operations than all other heap variants. It may be expected that linking operations in practice carry a higher cost than comparisons, as multiple pointers must be reassigned during a single linking. Thus the smaller number of linking operations may be more significant for practical performance than the number of comparisons, especially as the difference between the number of comparisons for smooth heap and pairing heap variants is usually less than the difference in the number of linking operations.

Conclusion 4.3. *On uniformly random input and semi-random input that does not in general include many sorted subsequences, the non-stable standard pairing heap and the non-stable back-to-front pairing heap are most efficient in the number of comparisons in sorting mode.*

The other two permutation classes which we have considered (localised permutation and uniformly random permutation) do not explicitly contain many sorted subsequences. On instances of these classes both smooth heap variants are consistently outperformed by the non-stable standard pairing heap and the non-stable back-to-front heap.

This conclusion does not necessarily hold if we consider link operations as well: then the smooth heap variants may have an advantage in practice, for the reasons discussed in Conclusion 4.2

Conclusion 4.4. *There is no single most efficient pairing heap variant in sorting mode. The relative performance of different variants is dependent on the structure of the input.*

We have seen in the experiments that pairing heap variants perform differently well on different input classes. The non-stable standard and the non-stable back-to-front heap have a clear advantage over the other pairing heap variants for uniformly random input as well as for instances of the localised permutation class. On instances of the near-neighbour permutation class, however, the stable multipass pairing heap performs fewer comparisons than all other pairing heap variants and matches the performance of the non-stable smooth heap.

5 Priority Queue in Dijkstra's Algorithm

This experiment simulates usage of heaps in a more practical scenario, i.e. as priority queue when running Dijkstra's algorithm. Dijkstra's algorithm is a well-known algorithm to find the shortest path from a source node in a weighted graph $G = (V; E)$ to all other nodes in the graph. In a

single run of the algorithm, there will be $|V|$ `insert` operations, $|V|$ `delete-min` operations and $O(|E|)$ `decrease-key` operations being performed in total.

We perform three experiments with Dijkstra's algorithm: One with a fixed edge probability and variable number of vertices and two with a fixed number of vertices and variable edge probability. Inputs are generated from a random graph model which is described in the following section. The first two experiments compares the performance of the same heap variants as in previous experiments, while the third focuses on exploring the effect of minor implementation changes to the smooth heap.

5.1 Implementation

Dijkstra's algorithm was implemented in python using the same pairing heap interface from previous experiments as a priority queue. Graphs were generated using the python library package `networkX` [10], which contains various functions to generate graphs of arbitrary size and shape. Experiments were performed on random graphs generated according to the Erdős-Rényi model: Given model $G_{n,p}$, a graph with n vertices is constructed by adding an edge between each pair of vertices with probability p. Weights were then assigned to edges by drawing integer values uniformly at random from the interval $[1, 10n]$.

The experimental setup is similar to that of previous experiments. For each experiment iteration, a random graph is generated. Then Dijkstra's algorithm is run once for each heap type on the same graph, counting the number of linking operations or comparisons. Results are averaged over multiple such iterations.

5.2 Experiment: Graphs of Variable Size

This experiment was performed on input graphs of variable size, generated with a fixed edge probability of 0.01. This small edge probability was chosen to allow computation of shortest paths even on large graphs. The number n of vertices varied from 100 to 10000. For each value of n Dijkstra's algorithm was run five times on five independently generated random graphs and the results were averaged.

Average number of linking operations.

Average number of comparisons.

Figure 14: Number of link operations and comparisons needed to find shortest paths in random graphs of variable size.

The results show that the smooth heap performs as well on this problem as in the previous sorting experiments. The standard pairing heap and the multipass pairing heap variants perform the lowest number of comparisons of the pairing heap variants, but the stable smooth heap

performs the lowest number of comparisons overall. The non-stable smooth heap performs a similar number of comparisons as the multipass variants.

5.3 Experiment: Graphs with Variable Connectivity

For this second experiment involving Dijkstra's algorithm, each random input graph $G_{n;p}$ is generated with a fixed size of $n = 500$ vertices and a variable edge probability p, with $0.01 \leq p \leq 1$ and step size 0.01. As before, five separate graph inputs are generated for each distinct value of the variable parameter and the result over these inputs is averaged.

Average number of linking operations. Average number of comparisons.

Figure 15: Number of link operations and comparisons, respectively, needed to find shortest paths in random graphs of fixed size and variable connectivity.

In this experiment we can observe the direct and indirect effect of a large number of (fairly random) `decrease-key` operations on efficiency. As the edge probability rises while the number of nodes in the graph remains fixed, the expected number of neighbours of each node increases. This in turn means that the expected number of calls to the `decrease-key` function increases as well, while the number of `insert` and `delete-min` operations remains fixed at $n = 500$ each. We can assume that the cost of inserting a graph vertex is unaffected by the structure of the graph. However, the same cannot be said about `delete-min`: Each execution of `decrease-key` changes the structure of the heap, and these changes may also show up in the number and order of roots in the top-list when it is consolidated during `delete-min`. This is meant by the indirect effect of decrease-key. Finally, the direct effect is the number of comparisons performed by the `decrease-key` calls themselves.

We observe some noticeable changes in the relative performance of heap variants: It appears that a large number of `decrease-key` operations is favourable to the lazy pairing heap variants, as these begin to outperform e.g. the standard pairing heap once a certain ratio of `decrease-key` operations to `delete-min` is reached. This is most likely explained by the fact that, because of its lazy linking, subtrees affected by `decrease-key` will tend to be smaller than in other variants. Both smooth heap variants again perform very well compared to the other variants, as do both multipass variants. This indicates that the `decrease-key` implementation of smooth heap is also competitive, perhaps for similar reasons as that of the lazy heap. The stable-linking smooth heap performs notably fewer comparisons than the non-stable variant in this case.

5.4 Experiment: Smooth Heap Variants

This experiment is focused on exploring different smooth heap variations in an effort to isolate the reasons for their different performance.

In the original paper on smooth heap [7], Kozma and Saranurak propose to implement `decrease-key` by removing the node whose key is being decreased along with its subtree and linking this tree to the root after its key has been decreased. In addition to the original version of the smooth heap, two further variants have been implemented for this experiment which differ mainly in the `decrease-key` operation. The first variant is due to as yet unpublished work by Corey Sinnamon. When decreasing the key of a node, that node is replaced by its leftmost child in its original position in the heap. The node whose key is decreased and its remaining subtree are removed and placed in a buffer forest of heaps that is maintained separately. A `clean-buffer` operation that links all roots in the buffer into a single tree and links this to the root of the separate top-list is called either when the buffer exceeds length $\log n$ or during an `delete-min` call before the minimum is removed. This consolidation is performed by sorting all roots in the buffer by keys and subsequently linking them into a path. An explicit implementation of mergesort is used to enable an exact count of the number of comparisons that are performed.

The second variant is due to László Kozma and also replaces the subtree rooted at the node whose key is decreased with the leftmost subtree of that node. A buffer forest is employed similarly to the previous variant, but the `clean-buffer` operation links all roots in the buffer into a treap in the same way as in the main top-list during an `delete-min` operation.

These smooth heap variants and the original one differ in three ways: Firstly in the use of a buffer, and secondly in their strategy for dealing with the removal of a node from the heap. Recall that the original smooth heap cuts the node from its current position in the heap together with its entire subtree, while the other variants leave the subtree rooted at the leftmost child of the node in its previous place. The final difference is that the original smooth heap performs a consolidation of the forest at the beginning of the `delete-min` operation, while the other two variants clean only their buffers before removing the minimum and consolidate the forest afterwards.

The question is in what way each of these three differences affects the performance of the heap. This question was approached experimentally by implementing further minor variants to the three smooth heap versions. For each of the three smooth heaps, a variant was implemented that differs from that heap only in the choice of leaving the left child of a node when it is cut out of the heap during decrease-key, i.e. the variant of the original smooth heap leaves the left child in place and the variants of the other two heaps do not. A final variant possesses a buffer which is consolidated with the `treapify` operation, removes the entire subtree of a node during `decrease-key` and performs consolidation of the forest before the minimum is deleted during `delete-min`. Since we consider these four new variants only to settle this one question, we refrain from naming them and refer to them simply in relation to the smooth heap they are a variant of.

Average number of linking operations. Average number of comparisons.

Figure 16: Number of link operations and comparisons, respectively, performed by different smooth heap variants to find shortest paths in random graphs of variable connectivity.

The results of this experiment show no significant difference between the number of comparisons performed by each of the main three smooth heap variants and their respective minor variant. From this follows our next conclusion:

Conclusion 5.1. *In the implementation of the decrease-key operation for the smooth heap there is no signi cant di erence in performance between leaving a child of the a ected node in its place in the heap and removing its entire subtree.*

The more `decrease-key` operations are performed relative to the number of `insert` and `delete-min` operations, the more comparisons the smooth heap variant with a sorting buffer performs compared to other variants. This relatively high number of comparisons performed by the `decrease-key` function of this variant is likely due to the sorting that must be carried out during each consolidation of the buffer. Apparently the benefits of consolidating the roots in the buffer into a path do not outweigh the higher cost of sorting. However, the other smooth heap variant which uses a buffer with `treapify` shows that the concept of the buffer itself is not necessarily a disadvantage compared to the original variant of smooth heap which uses none. These variants perform nearly equally well, with the variants with a buffer needing only slightly more comparisons.

The standout performance is the hybrid variant which combines a buffer with `treapify` consolidation with early consolidation of the main forest list. Based on our earlier reasoning, the combination of these two characteristics must be the cause of the improved performance of this smooth heap variant. Note that the buffer consolidation and the standard consolidation of the main top list of roots after a `delete-min` are performed by the same function. The improvement in performance over the original smooth heap is likely due to the fact that (1) the buffer is consolidated not only if `delete-min` is called, but more importantly also when a threshold length of log *heap:size* is reached, leading to potentially smaller substructures than in heaps which do not consolidate based on length thresholds. However, as we can see, this is only an advantage when a consolidation of both buffer and forest is performed *before* the minimum is deleted. The consolidation of the forest before the removal of the minimum has the advantage that the heap need not maintain an explicit pointer to the minimum at all times, because the minimum is identified implicitly at the time of removal as the root of the consolidated heap. We assume that these two advantages accumulate when combined, but that there is not necessarily any additional dynamic between them which leads to further improvement.

From the results, it appears that the difference in the number of comparisons between the original smooth heap and the variants containing a buffer with `treapify` is likely of a constant nature.

Conclusion 5.2. *The use of a bu er for the decrease-key operation can reduce the overall number of comparisons if both bu er and forest are consolidated before the removal of the minimum.*

Note that the use of a buffer introduces some additional overhead unrelated to the number of comparisons, so this observation does not necessarily mean that the buffer variant would actually perform faster than the original variant in practice. However, given that the gains of the modified variant in the number of comparisons are not insignificant, it is also possible that this translates to an improved practical performance.

5.5 General Results

We observe that the relative performance of heap variants changes when a large number of `decrease-key` operations is carried out. Unlike is sorting mode, in this case the standard pairing heap is outperformed by several pairing heap variants, namely both lazy heap variants, the stable back-to-front heap, the non-stable front-to-back heap and in particular both variants of multipass. Therefore we see a strong indication for preferring the smooth heap over the pairing heap in practice in practical applications where the number of `decrease-key` operations is expected to dominate the number of `delete-min` operations. If the use case is not well defined, a good compromise would seem to be the multipass variant of the pairing heap, whose performance often lies between that of the standard pairing heap and the smooth heap.

As with the experiments in sorting mode, we cannot make any general statement about the effects of stable linking versus non-stable linking, as this appears to depend on the structure of each specific heap. We can, however, conclude that the smooth heap consistently outperforms all pairing heap variants on this task, and that there are possibilities to reduce the number of comparisons performed by the smooth heap further, though probably only by a constant factor.

5.6 Conclusion and Outlook

We have seen that the smooth heap delivers very promising results in practice and performs better - both in the number of comparisons and in the number of links - than all pairing heap variants in many cases, particularly when sorting lists which contain sorted subsequences and in use cases where the number of `decrease-key` operations dominates other heap operations. These results indicate that the smooth heap might replace the standard pairing heap in certain practical applications for which the use case fits its strengths. We have also shown first results for variants of the original smooth heap which suggest that improvement (most likely by a constant factor) is possible. These improvement strategies have so far only been tested with stable linking and in a single limited scenario, but it might be worthwhile to pursue these ideas further.

The last experiment also gives rise to the question whether the introduction of a `decrease-key` buffer with `treapify` consolidation would also lead to improvement in pairing heap variants. This would be interesting even if this improvement is only by a constant factor, because the standard pairing heap is in widespread practical use, where such improvements matter.

6 Decrease-Key in Sort Heap Does Not Take $O(\log \log n)$ Time

Iacono and Özkan [6] proposed the *sort heap* as an example of a heap that has an amortized `decrease-key` bound of $O(\log \log n)$, requires no augmented data and conforms to Iacono and

Özkans's *pure heap* model. However, the proof of this running time contains a flaw, which we will discuss first and then demonstrate by giving a counterexample.

6.1 Statement of Original Result

A heap conforms to the *pure heap model* if it satisfies the following constraints: The heap consists of nodes which have a `key` attribute. After each execution of a `decrease-key`, `delete-min` or `insert` operation, the heap is a forest of subheaps. The `insert(x)` operation is implemented so that the new element x is added as the new leftmost heap to the forest of heaps. `Decrease-key`(x, Δ) is implemented by detaching x and its subtree from its current parent, decreasing $x.key$ by Δ and adding x with its subtree to the forest as the new leftmost heap. `Delete-min` performs a series of pointer-based operations to link the heaps of the forest into a single tree whose root contains the minimum key. The root is then removed, the minimum key returned, and the children of this root become the roots of heaps in the new forest. At a glance, permitted *pointer-based operations* are those that move a pointer between linked nodes or query the state of a node, e.g. whether it has siblings or children. The original definition of the model in [6] contains an exhaustive list of permitted operations.

The main result shown in the paper is that any heap which conforms to the pure model and has `delete-min` and `insert` operations with amortised cost $O(\log n)$ performs `decrease-key` in $\Omega(\frac{\log \log n}{\log \log \log n})$ amortised time, later improved to $\Omega(\log \log n)$ [5]. However, we are most interested in the example data structure they present: The *sort heap* conforms to the pure heap model and is claimed to have amortised cost $O(\log \log n)$ `insert` and `decrease-key` operations and $O(\log n \log \log n)$ amortised `delete-min`.

6.2 Implementation

The sort heap implements the three heap operations `insert`, `decrease-key` and `delete-min` as follows:

> `insert(x)` inserts a new heap consisting of a single node with key x in the leftmost position of the forest.
>
> `decrease-key`$(v, \Delta x)$ cuts node v from its parent, decreases its key by Δx and inserts the resulting subtree rooted at v as a new leftmost heap into the forest.
>
> `delete-min()` splits the roots of the forest left-to-right into groups of size $4 \log n$. The roots in each group are sorted in descending order and then paired in one left-to-right pass, resulting in one new heap per group. These heaps are paired in arbitrary order until only one heap remains. The root of this heap is the minimum. The root is removed and its resulting orphaned children form the roots of the new forest.

6.3 Potential Analysis

For any node x in the heap, define $L(x)$ as size of the induced subtree of x and $R(x)$ as the sum of the sizes of the induced subtrees of all right siblings of x. Furthermore, let $S(x) := L(x) + R(x)$. The node x is called *left-heavy* if $L(x) \geq \frac{2}{3}S(x)$, *right-heavy* if $R(x) \geq \frac{2}{3}S(x)$ and *transitional* otherwise. Let the *raw potential* of a node be defined as follows:

$$(x) := \begin{cases} 0 & \text{if } x \text{ is left-heavy} \\ \frac{R(x) - 1/3 S(x)}{1/3 S(x)} & \text{if } x \text{ is transitional} \\ 1 & \text{if } x \text{ is right-heavy} \end{cases}$$

The final potential of a node is defined as

$$(x) := \quad (x) c \log\log n;$$

where c is a constant whose value is chosen later in the proof. The following two lemmas and one observation are taken from [6] without change and hold true as stated:

Observation 15. *A single pairing only changes the raw potential of the two nodes directly involved in the pairing; their raw potentials can only change by one.*

Lemma 16. *In a heap of size n, the sum of raw potentials of k nodes which are mutual siblings is at least $k \quad \log_{\frac{3}{2}} n$.*

Lemma 17. *In a heap of size n, the sum of the raw potentials of any set S of nodes which are mutual ancestors/descendants (i.e. a subset of a root-to-leaf path) is at most $\log_{\frac{3}{2}} n$.*

6.4 Flaw in the Proof

Lemma 18 of the original paper is where the problem appears:

Lemma 18. *Removing a node and its subtree from a heap increases the sum of the raw potential of the remaining heap by at most 13.*

We shall shortly show with a counterexample that this is not generally true, but first we will discuss the flaw in the argumentation of the proof.

The proof begins by arguing that direct (former) ancestors of the node that is removed are the only nodes whose potential may decrease. These nodes are called $x_1; x_2; \ldots; x_\ell$. The proof then invokes an argument from Lemma 17 to state that $S(x_i) \quad \frac{3}{2} S(x_{i+1})$ and uses this to derive a lower bound for the size of $S(x_i)$ before x is removed. The oversight is this: the argument in Lemma 17 was made with regard to right-heavy or transitional nodes, but we cannot make such an assumption about the x_i that are considered here. We only know by definition that all x_i will be right-heavy or transitional *after x is removed*, but not before.

We will next present a counterexample to show that the lemma itself is in fact not true.

6.5 Counterexample

Lemma 6.1. *Removing a node and its subtree from a heap may increase the sum of the raw potentials of the remaining nodes in the order of $\log n$.*

Proof. Let x be the root of the subtree T_x that is removed. The only nodes whose potential may change by removing T_x are the ancestors x_i of x, for whom the size $L(x_i)$ of the left subtree will decrease, and the left siblings y_{ij} of the ancestors, for whom the size $R(y_{ij})$ of the sum of the size of the subtrees of all right siblings will decrease. Because only the right weight changes for any left sibling, and because it can only decrease, left siblings of the ancestors of x will only become left-heavier, so by definition their potential may only decrease.

Therefore we need only consider the direct ancestors of x when bounding the increase in potential. We construct a tree that maximises the number of ancestors of x whose potential increases when x and its subtree are removed. We will now describe such a tree and show that the number of nodes who flip from left-heavy to right-heavy may be in the order of $\log n$.

Let T be a tree with left spine $x_1, \dots, x_\ell, x$, where x is the root of the subtree that will be removed and x_ℓ is its parent. Then we have by definition

$$L(x_\ell) = M + 1 = L'(x_\ell) + M,$$

where M is the size of the subtree of x. We choose subtrees of the ancestors x_i $(1 \leq i \leq \ell)$ such that

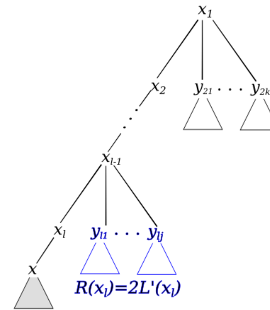


Figure 17: Counterexample for the proof.

$$R(x_i) = 2L'(x_i) = R'(x_i).$$

This means that all x_i will be *just* right-heavy after the subtree has been removed. To show a counterexample for Lemma 18, it is in fact enough to bound the number of x_i whose potential increases by 1, i.e. those that flip from left-heavy to right-heavy.
Using $S(x_i) = L(x_i) + R(x_i) = 3L'(x_i) + M$, we get

$$\begin{aligned}
&x_i \text{ left-heavy before removal of } x \\
\iff\quad & L(x_i) \geq \frac{2}{3}S(x_i) \\
\iff\quad & L(x_i) \geq \frac{2}{3}(3L'(x_i) + M) \\
\iff\quad & L'(x_i) + M \geq 2L'(x_i) + \frac{2}{3}M \\
\iff\quad & \frac{M}{3} \geq L'(x_i).
\end{aligned}$$

Since

$$\begin{aligned}
L'(x_i) &= S'(x_{i+1}) + 1 \\
&= 3L'(x_{i+1}) + 1 \\
&= 3^{l-1}L'(x_\ell) + \sum_{j=0}^{l-1} 3^j \\
&= \frac{3^{l-i}-1}{2},
\end{aligned}$$

we get

$$\begin{aligned}
& M \geq 3\,\frac{3^{l-i}-1}{2} \\
\iff\quad & l - i \leq \log_3\left(\frac{2M}{3} + 1\right).
\end{aligned}$$

In other words, there are $k = \left\lfloor \log_3 \frac{2M+1}{3} \right\rfloor$ nodes in the tree whose potential increases by 1 when x is removed. These are the lowest k nodes. The potential of all other $x_i, i < \ell - k,$

increases by less than 1 or remains at 0, but by design none decreases. The only nodes whose potential might decrease would be left siblings of ancestors of the disconnected node (the x_i), but our example tree is designed so that no such siblings exist. Therefore, by choosing e.g. $M = \frac{n}{2}$, we can show that the increase in the sum of the raw potential of remaining nodes is at least in the order of $\log n$ if k is. To show this last condition, we simply observe

$$\begin{aligned} S'(x_0) &= 3L'(x_0) \\ &= 3 \quad \frac{3^l \quad 1}{2} \end{aligned}$$

and use $S'(x_\cdot) \quad n \quad M$ to obtain

$$l \quad \log_3 \quad \frac{2}{3}(n \quad M) + 1 \quad :$$

□

This immediately implies that the amortised cost of `decrease-key` is not in $O(\log \log n)$. However, we can show that the amortised cost does not get worse than $O(\log n \log \log n)$.

6.6 Corrected Analysis

Lemma 6.2. *Removing a node and its subtree from a heap increases the sum of the raw potential of the remaining nodes by at most* $\log_{\frac{3}{2}} n$.

Proof. Recall that the only nodes whose potential may increase after a node is detached are the former direct ancestors of that node. Then the lemma follows directly from Lemma 17. □

Lemma 6.3. *The amortised cost of* `decrease-key` *in a sort heap is* $O(\log n \log \log n)$.

Proof. The actual cost of detaching a subtree and adding it to the forest of heaps is 1. In the detached subtree only the potential of the new root may change if it loses right siblings. However, this can only lead to a decrease in the node's potential. By Lemma 6.3, the potential of the tree that is left behind after the node is detached increases by no more than $\log_{\frac{3}{2}} n$. Combining these observations gives an amortised cost of at most $1 + \log_{\frac{3}{2}} n \; 2 \; O(\log n \log \log n)$. □

7 Stable Linking Can Always Be Done Sequentially

7.1 Problem

In the original paper on the smooth heap [7], the authors also define the more general *stable heap model*, of which their smooth heap is an instance. A *stable heap* instance of this model consists of a forest of heaps which are consolidated into a single tree during the *delete-min* operation. The model requires that this consolidation consists of a sequence of `stable-link` operations on pairs of neighbours which result in a single tree. The model does not consider pointer moves, but allows stable-linking between neighbours in the forest list in any order. We consider the question whether the order of link operations in the consolidation sequence matters, i.e. whether any tree that can be constructed using stable-linking operations in arbitrary order can also be constructed using only a linear number of pointer moves on the forest list. If this is true, we can encode any stable heap tree as a string of $O(n)$ symbols (using only symbols for *move pointer to the left*, *move pointer to the right* and *perform stable linking*), giving us an upper bound on the number of valid stable heap trees.

7.2 Algorithm and Proof

Lemma 7.1. *For any heap created from the list of roots $x_1; : : : ; x_n$ by stable-linking neighbours in arbitrary order until only one root remains in the top-list, we can construct the same tree using only a linear number of pointer moves.*

We prove this by giving an algorithm that takes the target tree as input and produces a sequence of linking operations that construct the same tree. We claim that this sequence always requires no more than $O(n)$ pointer moves. Note that the children of a node can be conceptually split into left and right children, i.e. children that were linked to the root from the left versus children that were linked from the right. For simplicity we assume that the input tree contains information about left and right children; otherwise we could infer this information from the original list of roots.

Algorithm 5: ConstructSequence($T; x$)

Data: template tree T, current node x
Result: ordered sequence S of linking pairs to construct T
if *x is a leaf* **then**
  return empty sequence
end
S ()
foreach *left child y of x from outermost to innermost* **do**
  S' ConstructSequence($T; y$)
  S S S'
end
foreach *left child y of x from innermost to outermost* **do**
  S S $((y; x))$
end
foreach *right child y of x from outermost to innermost* **do**
  S' ConstructSequence($T; y$)
  S S S' $((x; y))$
end
return S

We will first prove that this algorithm is correct, i.e. that the sequence of pairing operations it produces is feasible and, when executed, constructs the input tree. Then we will show that the returned sequence of pairing operations can always be executed with a linear number of pointer moves.

Lemma 7.2. *Let $l = v_1; : : : v_n$ be an ordered list of nodes. A linking operation $(u; v)$ is feasible i u and v are neighbours in the top-list at the time of the operation. Let T be a tree that can be produced from l by performing a sequence of n 1 feasible stable-linking operations. Then* `construct-sequence(T, T.root)` *returns a sequence of feasible stable-linking operations which, when executed, reconstruct T from l.*

Proof. We prove the claim by structural induction on the height h of T.

Base Case 1: $h = 0$

The tree consists of a single root node u. The tree is equivalent to the input list; no linking is required in this case. The algorithm correctly returns an empty sequence.

Base Case 2: $h = 1$

Let u be the root of T, $x_1, \ldots, x_l; 0 \le l \le n$ the left children of u from leftmost to right, if they exist, and $x_{l+1}, \ldots, x_r$ the right children of u from the innermost (i.e. leftmost) right child to the right. Since $h = 1$, all existing children are leaves and there exists at least one child. Then the first loop at the highest recursion level of the algorithm, attempting to construct the subtrees rooted at children of the root, returns an empty sequence. The rest of the algorithm constructs the ordered output sequence

$$S = ((x_i, u) | l \ge i) \circ ((u, x_j) | l < j \le r),$$

where the respective subsequences are ordered by descending i and ascending j, respectively. By the definition of stable-linking the original list must be structured such that $l = l_L \circ u \circ l_R$, where l_L is the left-to-right ordered list of left children and l_R the left-to-right ordered list of right children.

If u has left children, the first linking operation in the sequence links the left child with the largest index to u. As we have just seen, this must be a direct neighbour of u in the original root list. During linking the left child is pushed down out of the root list, and so its left neighbour becomes the new neighbour of u. This new neighbour is exactly the node which will linked to u next during the sequence. This continues until there are no roots to the left of u left in the top-list. So we can conclude that the sequence linking all left children to the root is feasible, since roots are always neighbours at the time of linking. The right children, too, are linked to the root from innermost to outermost, and the same argument may be used to show feasibility of the resulting sequence. In this base case, we can infer the equivalence of the input tree and the tree produced by the output sequence directly from our reconstruction of the original top-list and the definition of stable linking, since all nodes are linked only with the root, which we know to contain the smallest key, and stable linking dictates that siblings have the same relative order as they had in the original top-list.

Induction Step: $h \rightarrow h + 1$

The induction hypothesis is that the claim is true for all trees of height less or equal h.

Again, let u be the root of the tree, $x_1, \ldots, x_l; 0 \le l \le n$ the left children of u from leftmost to right, if they exist, and $x_{l+1}, \ldots, x_r$ the right children of u from left to right. By definition of stable linking, the original top-list l must be structured such that $x_1, \ldots, x_l$ appear in the same order fully to the left of u, and $x_{l+1}, \ldots, x_r$ in this order to the right of u. Let S_i be the sequence returned by the recursive call $\texttt{construct_sequence}(T, x_i)$. Since x_i is a child of the root, the subtree rooted at x_i has height $\le h$. Then by the induction hypothesis S_i is feasible and correctly constructs the subtree.

The algorithm returns the sequence $S = S_1, S_2, S_l, (x_l, u), \ldots, (x_1, u), S_{l+1}, (u, x_{l+1})$. The first half of the sequence, up to and including (x_1, u), first constructs the subtrees of all left children of the root and then links these to the root. We can see that the result is correct by a combination of the induction hypothesis and the argument from the base case. Now we consider the right children and their subtrees: They, too, are constructed left-to-right, but a child is linked to the root as soon as the subtree has been constructed completely. Constructing the subtree of a right child removes all nodes of this subtree from the top-list, with the exception of the right child itself. Because each subtrees are constructed before it is linked to the root, and because this is done left-to-right, each linking operation must be feasible. Correctness of the result again follows from the induction hypothesis for the subtrees and from the argument about direct children of the root from the proof of the base case. □

What remains to show is that the sequence that is returned by the algorithm can always be executed using only a linear number of pointer moves.

Lemma 7.3. *For any heap created from the list of roots $x_1; : : : ; x_n$ by stable-linking neighbours in arbitrary order until only one root remains in the top-list, the linking sequence constructed by the algorithm can be executed with $O(n)$ pointer moves on the initial list.*

Proof. Let `stable-link(x)` be the operation that stable-links x with its right neighbour. If x becomes the child of this neighbour during linking, this operation includes a single pointer move to the new root. Let S be the sequence of linking operations returned by the algorithm. At each recursion level, the sequence to build the current subtree is constructed by passing left-to-right over the left children of the current root, constructing their subtrees, and then linking the results to the current root in a subsequent right-to-left pass. For the right children of the current root subtrees are constructed and linked in a single left-to-right pass. Observe that each subtree in the completed tree corresponds to an interval in the original top-list. When executing the linking sequence, each single pointer move is performed for one of three reasons:

(i) to pass over left-to-right from the root of a newly constructed left subtree to the leftmost element (i.e. leftmost child or root) of a new left subtree,

(ii) to reach the left participant of a `stable-link` operation or

(iii) to move the pointer during a `stable-link`(x) operation in which x (the left participant) becomes the left child of the other.

Let k be the overall number of left children anywhere in the completed tree. Then a maximum of $k \quad 1$ pointer moves in total are performed due to (i), because the subtree of each left child is constructed completely before moving on to the next. We include pointer moves over left children that are leaves in this count.
Pointer moves due to (ii) are performed up to $n \quad 1$ times, as each of these is directly followed by a linking operation. The number of moves due to (iii) is trivially bounded by the number of linking operations $n \quad 1$.
Therefore, the overall number of pointer moves can be bounded by $k \quad 1+n \quad 1+n \quad 1 \; 2 \; O(n)$. □

References

[1] Michael L. Fredman. “On the Efficiency of Pairing Heaps and Related Data Structures”. In: 46.4 (1999). ISSN: 0004-5411. DOI: 10.1145/320211.320214. URL: https://doi.org/10.1145/320211.320214.

[2] Michael Fredman, Robert Sedgewick, Daniel Sleator, and Robert Tarjan. “The Pairing Heap: A New Form of Self-Adjusting Heap”. In: *Algorithmica* 1 (Nov. 1986), pp. 111–129. DOI: 10.1007/BF01840439.

[3] *Git Repository containing experiment implementations (see branch \thesis")*. URL: https://git.imp.fu-berlin.de/hlm/pairingheap/-/tree/thesis (visited on 10/02/2020).

[4] Kurt Hoffman, Kurt Mehlhorn, Pierre Rosenstiehl, and Robert E. Tarjan. “Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees”. In: *Inf. Control.* 68 (1986), pp. 170–184.

[5] John Iacono and Özgür Özkan. “A Tight Lower Bound for Decrease-Key in the Pure Heap Model”. In: *CoRR* abs/1407.6665 (2014). arXiv: 1407.6665. URL: http://arxiv.org/abs/1407.6665.

[6] John Iacono and Özgür Özkan. “Why Some Heaps Support Constant-Amortized-Time Decrease-Key Operations, and Others Do Not”. In: *ICALP*. 2014.

[7] László Kozma and Thatchaphol Saranurak. “Smooth Heaps and a Dual View of Self-Adjusting Data Structures”. In: *SIAM Journal on Computing* 0.0 (0), STOC18-45-STOC18–93. DOI: 10.1137/18M1195188. eprint: https://doi.org/10.1137/18M1195188. URL: https://doi.org/10.1137/18M1195188.

[8] Daniel H. Larkin, Siddhartha Sen, and Robert Endre Tarjan. “A Back-to-Basics Empirical Study of Priority Queues”. In: *CoRR* abs/1403.0252 (2014). arXiv: 1403.0252. URL: http://arxiv.org/abs/1403.0252.

[9] *Mailing Chain on Jordan Permutation*. URL: https://www.ics.uci.edu/~eppstein/junkyard/jordan-splay.html (visited on 09/29/2020).

[10] *NetworkX*. URL: https://networkx.github.io/ (visited on 07/23/2020).

8 Appendix

In the following the figures of sections 4 and 5 (displaying the results of experiments) are repeated in a larger format.

Figure 13: Number of comparisons required by different heap variants to sort Jordan permutations of length up to 2^{17}.

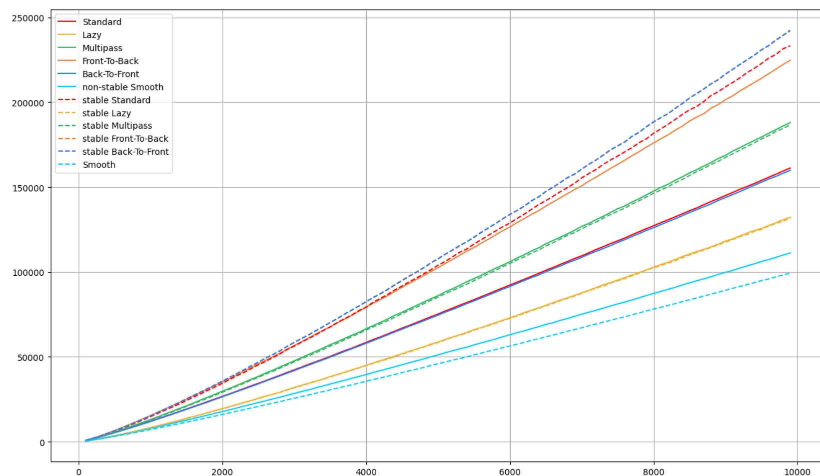


Figure 8a: Average number of link operations. Sorting a uniformly random permutation with different heap variants. Size of input list varies from $n = 100$ to $n = 10000$ in increments of 100.

Figure 8b: Average number of comparisons. Sorting a uniformly random permutation with different heap variants. Size of input list varies from $n = 100$ to $n = 10000$ in increments of 100.

Figure 9a: Average number of link operations needed to sort permutations of the class of continuous sorted subsequences.

Figure 9b: Average number of comparisons needed to sort permutations of the class of continuous sorted subsequences.

Figure 10a: Average number of link operations needed to sort permutations of the class of non-continuous sorted subsequences.

Figure 10b: Average number of comparisons needed to sort permutations of the class of non-continuous sorted subsequences.

Figure 11a: Average number of link operations needed to sort permutations of the class of localised permutations.

Figure 11b: Average number of comparisons needed to sort permutations of the class of localised permutations.

Figure 12a: Average number of linking operations needed to sort permutations of the class of near-neighbour permutations.

Figure 12b: Average number of comparisons needed to sort permutations of the class of near-neighbour permutations.

Figure 14a: Average number of link operations needed to find shortest paths on random Erdős-Rényi graphs of variable size and fixed edge probability $p = 0.1$.

Figure 14b: Average number of comparisons needed to find shortest paths on random Erdős-Rényi graphs of variable size and fixed edge probability $p = 0{:}1$.

Figure 15a: Average number of link operations needed to find shortest paths on random Erdős-Rényi graphs with variable connectivity. $n = 500$, $0 < p \quad 1$, step size $0{:}01$.

Figure 15b: Average number of comparisons needed to find shortest paths on random Erdős-Rényi graphs with variable connectivity. $n = 500; 0 < p$ 1, step size 0.01.

Figure 16a: Average number of link operations needed by different smooth heap variants to find shortest paths in random Erdős-Rényi graphs.

Figure 16b: Average number of comparisons needed by different smooth heap variants to find shortest paths in random Erdős-Rényi graphs.