SOFT HEAPS VISUALIZED

A Thesis
by
SHANE MCCANN

Submitted to the School of Graduate Studies
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

December 2023
Department of Computer Science

SOFT HEAPS VISUALIZED

A Thesis
by
SHANE MCCANN
December 2023

APPROVED BY:

Raghuveer Mohan, Ph.D.
Chairperson, Thesis Committee

Tinghao Feng, Ph.D.
Member, Thesis Committee

Andrew Polonsky, Ph.D.
Member, Thesis Committee

James Fenwick, Ph.D.
Chairperson, Department of Computer Science

Ashley Colquitt, Ph.D.
Associate Vice Provost and Dean, Cratis D. Williams School of Graduate Studies

Abstract

SOFT HEAPS VISUALIZED

Shane McCann
B.S., Appalachian State University
M.S., Appalachian State University

Chairperson: Raghuveer Mohan, Ph.D.

The *soft heap* is an approximate priority queue data structure originally proposed by Bernard Chazelle [3]. Chazelle used it to develop the fastest deterministic algorithm to compute a minimum spanning tree of a connected graph [2]. The soft heap allows some items to become "corrupted", in which their keys are artificially increased. This allows us to overcome the sorting lower bound in the comparison-based model of computation. However, some of the corrupted items exit the heap out of order. The number of corrupted items in the heap is upper-bounded by εm, where ε is an error rate determined by the user, and m is the number of insertions into the heap. This corruption allows soft heap operations to run in constant amortized time for a suitable $\varepsilon = O(1)$, making it ideal for applications where speed is prioritized over precision. Such applications include finding an approximate median of a set of items, and in dynamic maintenance of percentiles [3].

Chazelle's initial implementation of the soft heap uses a collection of binomial heaps. This was simplified in both implementation and analysis, by Kaplan and Zwick [10], and later by Kaplan, Zwick, and Tarjan [9], both of which use binary heap-ordered trees instead. In this thesis, we develop a visualization tool to visualize the soft heap implementation of Kaplan, Zwick, and Tarjan. We also visualize three applications in which soft heaps are used. Our web-based tool can be easily extended to make visualizations of other complex data structures and algorithms.

For completeness, we provide a new presentation of the implementation and the analysis of Kaplan, Zwick, and Tarjan's soft heap data structure. We hope that our presentation, along with the visualization tool, makes complex data structures and algorithms more accessible to early (and intermediate) computer science students. Whereas existing implementations may

corrupt items during both insertions and deletions, we provide a slight modification of the `delete-min` operation, to include only single *fillings*, and thereby not corrupt any more items during deletions in the soft heap. Thus, our modification allows the soft heap data structure to be used as a black box, where corruption only comes from the `insert` operation, and ensures that only at most εm total items (either in the heap or removed) are corrupted.

Acknowledgements

First, I would like to thank my advisor, Dr. Raghuveer Mohan. After taking his graduate algorithms course, I told him that I was interested in learning more in data structures and algorithms. That lead to me undertaking this thesis. Without his guidance, this thesis would likely still be incomplete. I especially thank him for his patience while working on this thesis. In times of stress, he was encouraging, and understanding in my lack of progress. I also got to know Dr. Mohan outside an academic context. While we may often disagree, I do appreciate his perspective on many things. It has been a pleasure to get to know him.

I am thankful to both Dr. Tinghao Feng and Dr. Andrew Polonsky for serving on my committee. I was a teaching assistant for Dr. Feng in his first semester at Appalachian State, and I enjoyed working with him. His experience with data visualization made him an excellent choice for my committee. I was both a student of Dr. Polonsky's and a teaching assistant of his. Learning from him and working with him piqued my interest in programming language theory, and I hope to continue to study it along with him in the future. His knowledge in theoretical computer science made him an excellent committee member as well.

Dr. Patricia Johann is one of the main reasons that I ended up joining the master's program at Appalachian State. In my final semester in undergrad, I was considering joining the master's program, though I had a lot of self-doubt. I was in one of her courses, and one day after class, she wanted to meet to encourage me to join this program. Without that, it is likely that I would not have joined.

While in this program, I have met several wonderful people. Each of them has played some part in the completion of this thesis, and in the maintenance of my sanity and happiness.

My dear friends Phillip Davis and Maddie Wyatt have become two of my most favorite people. Whenever I think of either of them, I instantly feel a smile come to my face. Through

Phillip and Maddie, I have had the opportunity to experience many new things. I cannot adequately describe my love for both of them. (And an extra-special thank you to my god-kitty, Petal.)

I began to know Val Lapensée-Rankine as she finished her master's thesis. Our discussions about her thesis influenced me to take on a thesis of my own. I have only briefly known her sister, Danielle, but in that short time while finishing this thesis, she has been so encouraging. Danielle has been such a wonderful office-mate and friend. I greatly enjoy the time that I spend with both Val and Danielle.

When I first met Marty McClearen, he was a student in a lab I was assisting. Over time, we became good friends as graduate students, and have shared many a drink discussing various topics; from computer science, to relationships, and to many other facets of life.

Courtney Dixon has been such a joy to work with. Our shared interest in improving computer science education is just one of the ways I relate with her. Our upbringings were similar in some ways, and I've sought her advice multiple times. Plus, she laughs at all of my jokes – even the unfunny ones (of which there are none).

I have known my old friend Chris Shambach since middle school. Even though our paths have split and merged again multiple times, we still manage to maintain our friendship. For that, and for his unending love and support, I will always be grateful.

Of course, I am thankful for my family as well. My parents, Adrianne and Dave, have been supportive of my education, every step of the way. Without their support, and the invaluable assistance that I've received from them, I would not have made it this far. I am especially thankful for their patience while I have continued my education, and made increasingly infrequent visits. My brother, Matthew, has been supportive as well. Every time I visit, he checks in about my work and how it's progressing.

Finally, I would also like to thank Emma Robinson. Emma has been my counselor for much of the last year and a half. With her counsel, I have grown so much that I barely recognize myself from before. From overcoming anxiety in relationships, to feeling more comfortable expressing myself, and to better understanding myself and how I think. I fondly remember every laugh and every tear shed with her. I always look forward to meeting her, and I am so privileged to have gotten to know her.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graphs are well-studied combinatorial objects that are comprised of nodes and connections between pairs of nodes called edges. A spanning tree of a connected graph contains all nodes of the graph, and a subset of the edges that allows one to travel from one node to another. Some graphs are weighted, where each edge has some value associated with it that represents the cost of traversing that edge. A minimum spanning tree (MST) is a spanning tree that minimizes the total cost of all edges in the tree. MSTs are used in the design of networks and circuits to find the least cost way of soldering digital components. Their application also extends to areas like clustering, speech recognition, and image processing [8].

Two of the most popular algorithms for computing MSTs are Prim's algorithm and Kruskal's algorithm. (See [8] for the history of the MST problem.) These algorithms require a sorting step to sort the edges of the graph in $O(m \log n)$ time, where n is the number of nodes, and m is the number of edges in the graph. The problem of sorting has a lower bound of $\Omega(n \log n)$ in the comparison-based model of computation. In an effort to overcome this sorting bottleneck, Bernard Chazelle designed and used an approximate priority queue called the *soft heap* [3]. This allowed Chazelle to compute an MST in $O(m\alpha(m, n))$ time, where α is the inverse Ackermann function, which grows very slowly and is nearly constant for all practical purposes [2]. This is currently the fastest deterministic algorithm for computing an MST of a connected graph in the comparison-based model of computation. The soft heap is designed to sacrifice accuracy for the sake of efficiency by "corrupting" some items in the heap. Given some error rate ε, the soft heap corrupts no more than εm items, where m is the total number

of elements inserted into the heap [3].

Although soft heaps were designed with Chazelle's MST algorithm in mind, there are other useful applications. We can perform approximate sorting by inserting a sequence of items into a soft heap. These items exit the soft heap nearly sorted. Soft heaps can also be used for selection, where the goal is to return the k^{th} smallest item from a list of items. Selecting the median is often used as a black box in complex divide and conquer algorithms. The algorithm to perform selection using a soft heap is friendlier than the classic deterministic algorithm of Floyd and Rivest [6], and runs in linear time.

Soft heaps use elegant algorithm design techniques, which may be of interest to the data structures enthusiast. In this thesis, we build a web application that visualizes the different soft heap operations, as well as its applications in near-sorting, median-finding, and dynamic percentile maintenance. Our web application is written to be easily extended to visualize other advanced data structures like skew heaps [18], splay trees [17], Zip trees [19], B-trees [1], and more. These visualizations help in forming an intuitive understanding of these data structures and their applications, and may offer insights into some non-trivial academic results in the field.

1.1 Preliminaries

1.1.1 Priority Queues

A *priority queue* is a data structure that maintains a set of elements with associated priority, and elements are removed in order of priority. We assume a min-priority queue, that is, the lower the value of the key, the higher an element's priority. Priority queues are often used to schedule tasks (such as jobs in computing) and sort items, as well as a black box in more complex algorithms like Dijkstra's shortest path algorithm, or common algorithms for computing maximum flows in graphs [8].

All priority queues support the following operations:

- `insert(`e`, `k`)`: insert a new element e with key or priority k.
- `delete-min`: remove and return the element with minimum key.

Dijkstra's shortest path algorithm also requires the `decrease-key` operation:

- `decrease-key(`e`, `Δk`)`: given a pointer e to an element in the priority queue, decrease its key by Δk. Note that this *increases* e's priority.

Some priority queues may also support:

- `increase-key(`e`, `Δk`)`: given a pointer e to an element in the priority queue, increase its key by Δk, thereby *decreasing* its priority.
- `delete(`e`)`: Given a pointer to an element e in the queue, remove e.
- `find-min`: return an element with the minimum key.

Priority queues are commonly implemented as heap-ordered trees, which are recursively defined as follows:

i. An empty tree is heap-ordered.

ii. A node x is heap-ordered if

   a) the left and right subtrees of x are heap-ordered, and

   b) x's key is less than or equal to the keys of its children.

We refer to this as the *heap-property*. In the next sections, we discuss two common ways to implement a priority queue using heap-ordered binary trees.

1.1.2 Binary Heaps

Binary heaps are nearly-complete binary trees, where each level of the tree is filled except for the last level. They satisfy the heap property, which causes the minimum element to always be at the root and gives us direct access to this element. A common way to implement a binary heap is to encode it in an array. Elements of an array can be accessed in a tree-like fashion. Given an element i in the array, you can access its parent by $\lfloor \frac{i-1}{2} \rfloor$, its left child by $2i+1$, and its right child by $2i+2$. Note that the array uses 0-based indexing, so the first element starts at index 0 [1].

In addition to priority queue operations, binary heaps use the following:

[1] There are alternate implementations that use 1-based indexing, in which the first element of the heap starts at index 1. This allows us to store the number of elements in a valid heap at index 0.

- `sift-up(`c`)`: repeatedly swap element c with its parent until the heap property is satisfied between element c and its parent
- `sift-down(`p`)`: repeatedly swap element p with the child with the smallest key until the heap property is satisfied between element p and its children

Both of these operations run in $O(\log n)$ time since they only span the height of the binary heap. We can implement priority queue operations using both `sift-up` and `sift-down` in a binary heap.

- `insert(`e`)`: place a new element e at the end of the heap, then `sift-up(`e`)`.
- `delete-min`: swap the root element and the last element e, then `sift-down(`e`)`. The number of elements in the heap decreases by one, so the last element in the array is no longer part of the heap.
- `decrease-key(`e`,` Δk`)`: decrease element e's key by Δk, then `sift-up(`e`)`.
- `increase-key(`e`,` Δk`)`: increase element e's key by Δk, then `sift-down(`e`)`.
- `delete(`e`)`: swap element e and the last element l, then call both `sift-up(`l`)`, and `sift-down(`l`)`.

Since `sift-up` and `sift-down` both run in $O(\log n)$ time, all the priority queue operations also run in only $O(\log n)$ time. For more details on binary heaps, please see popular textbooks on algorithms, like [16] and [5].

1.1.3 Mergeable Heaps

Mergeable (or meldable) heaps are a class of heaps that support the `merge` operation. Given two heaps H_1 and H_2, `merge(`H_1`,`H_2`)` combines both heaps into a single heap and destroys H_1 and H_2 in the process. If heaps are implemented as binary heap-ordered trees, then we can perform all priority queue operations in terms of `merge`.

- `insert(`e`,` H`)`: create a single-element heap with e, then merge with heap H.
- `delete-min`: remove the root node, then merge its left and right subtrees.

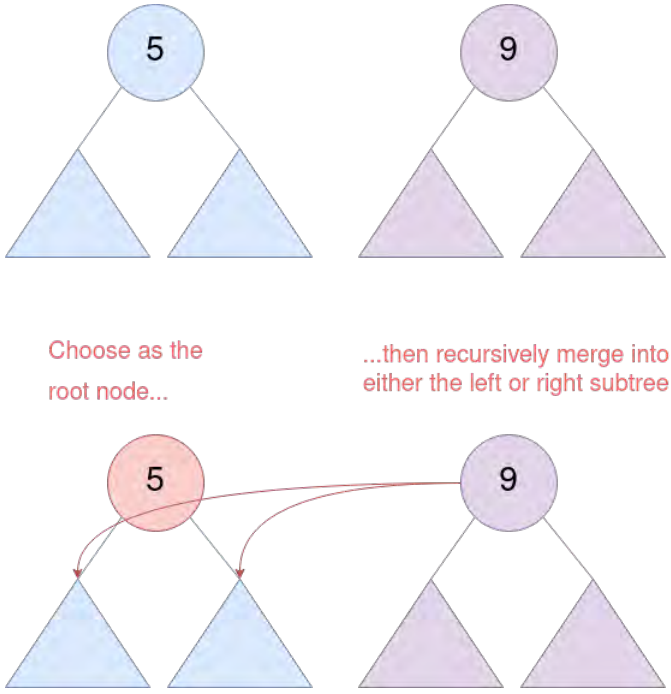
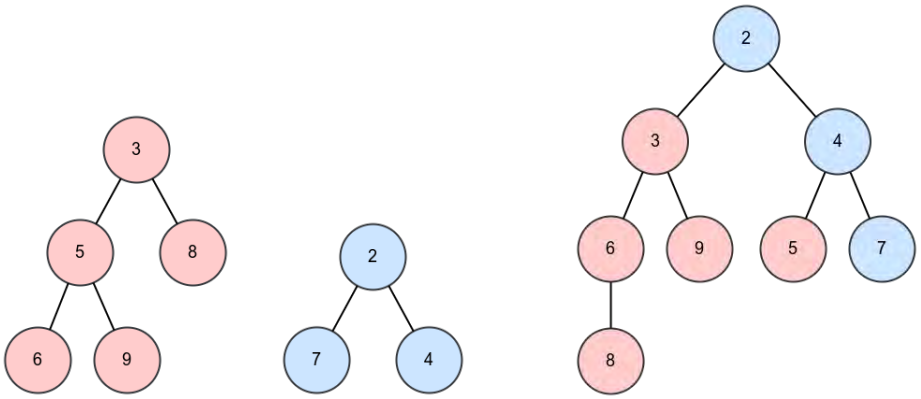


Figure 1.1: Recursive View of Merging Two Heaps

- `find-min`: return the root node.
- `delete(`e`)`: decrease the key of e to $-\infty$ so that it becomes the root, then call `delete-min`.
- `decrease-key(`e`,` Δk`)`: detach e and its subtree from its parent, decrease its key by Δk, then merge the tree back into the heap.
- `increase-key(`e`,` Δk`)`: delete e then re-insert it with its key increased by Δk.

To merge two heaps, take the smaller root between the two and make it the root of the new heap. Then recursively merge the other heap into either the left subtree or the right subtree of the root. Figure 1.1 shows this top-down recursive process, and Figure **??** shows an example. There are several ways to choose which subtree to merge with, depending on specific heap implementations. Augmenting each node of the tree with the heights or sizes of their subtrees guarantees $O(\log n)$ time for merge by always recursing on subtrees with smaller heights or sizes respectively.

The *leftist heap* is an example of a mergeable heap. It is implemented as a binary tree, and its name is derived from the fact that each node is left-heavy – that is, the size of the



(a) Two Heap-ordered Binary Trees (b) Both Trees Merged into One

Figure 1.2: An Example of Merging Two Heaps

left subtree of a node is at least as large as the size of its right subtree. A node in a leftist heap is augmented with the size of its subtree [4]. Other implementations augment nodes with the height of their subtree or with the shortest root-to-null path in their subtree [18] [11]. Should a node not be left-heavy, then we swap its children to maintain this property. When merging leftist heaps, we can always merge into the right subtree of a node, since it is always the smallest subtree. After merging, we fix any violations in the heap property by swapping left and right subtrees of non-left heavy nodes along the insertion path of the newly inserted element. Merging leftist heaps takes $O(\log n)$ time [4].

The *skew heap* [18] is another example of a mergeable heap, which guarantees $O(\log n)$ amortized times for every priority queue operation without having the need to maintain augmented information solely for the purposes of balancing. An operation is considered to run in $O(f(n))$ amortized time if any sequence of k operations (starting from an empty heap) takes $O(kf(n))$ time in the worst case. This is an upper bound over a sequence of operations, which is no worse than the total time it takes to perform the same sequence of operations on a data structure with worst-case guarantees. One can also think about amortized running times as taking an average over a sequence of operations.

1.2 The Soft Heap

The soft heap is a class of mergeable heaps. It supports `delete-min` and `delete` in $O(\log \frac{1}{\varepsilon})$ amortized time, where ε is an error rate. All other operations run in only $O(1)$ amortized time. This does not violate the sorting lower bound. Given an error rate $0 \leq \varepsilon \leq \frac{O(1)}{n}$, all soft heap operations run in $O(\log n)$ amortized time as well, giving a runtime of $O(n \log n)$ to sort n comparable elements. If $\varepsilon = \frac{1}{O(1)}$, then soft heap operations run in $O(1)$ amortized time, though items come out only nearly-sorted.

There are a few different implementations of the soft heap: Bernard Chazelle's original design [3], Haim Kaplan and Uri Zwick's simplification [10], and an updated version of their simplified soft heap by Kaplan, Zwick, and Robert Tarjan [9]. This thesis discusses the implementation of the latter, as we find it more intuitive to implement and analyze, though we will discuss key differences between the different versions in Chapter 2.

As mentioned before, the main feature that differentiates the soft heap from other heaps is "corruption." This corruption does not change the actual data inside the heap, but it does change the order in which items come out of the heap. Corruption occurs by increasing the key of some items through what Chazelle calls "carpooling." Nodes in a soft heap are allowed to contain multiple items. Nodes begin with a single item, but can acquire more items at certain points during heap updates. This is determined by the error rate ε. Given an appropriate error rate (say $\varepsilon = \frac{1}{O(1)}$), the amortized time complexity of all operations is $O(1)$, making them more efficient than a standard priority queue. We will look at soft heap operations and corruption in more detail in Chapter 2.

Our primary contribution in this thesis is the creation of a web-based visual tool that can be used to visualize data structures and algorithms. Our web application is built using the Vue.js front-end framework and Cytoscape.js for graph visualizations. We provide abstractions of useful Cytoscape functions and common procedures involving Cytoscape in an effort to make the development of new visualizations more friendly.

To demonstrate the visualization and how it can be used, we have implemented an animated version of the soft heap. Users can perform the top-level soft heap operations `insert`, `find-min`, and `delete-min`. The visualization animates the addition, movement, and se-

lection of nodes and edges to give the user a dynamic representation of how the soft heap operates. In addition to a general soft heap visualization, we have also created visualizations of some applications discussed in Chapter 3.

We have implemented the soft heap in Java, C++, Python, and JavaScript. Users of these implementations may use them as black boxes, such that only εm items become corrupted in total. Our extensive testing suite consists of testing scripts for each language, as well as input files that can be used to test any additional implementations. These are publicly available via GitHub [12]. Most importantly, we have implemented the soft heap in TypeScript, the language we use to power our visualization. The visualization is also publicly available in a separate GitHub repository [13].

In addition to these contributions, we have also slightly modified existing implementations of the soft heap. This modification prevents additional corruptions from occurring during deletions, which allows us to use the soft heap as a black box in larger algorithms.

The rest of the thesis is organized as follows. In Chapter 2, we will discuss the implementations of all the soft heap operations, as well as provide an analysis of their amortized time complexities. For this, we present the analysis of Kaplan et al.'s soft heap [9], but hope to make it more accessible. We also discuss our modification to prevent additional corruption during `delete`. In Chapter 3, we discuss some applications of the soft heap data structure to problems like near-sorting and selection. In Chapter 4, we discuss our visualization tool, and how it can be extended to visualize other data structures and algorithms. We conclude in Chapter 5 with a summary of our contributions and comment on future work.

Chapter 2

The Soft Heap

The soft heap data structure was first proposed by Chazelle [3]. This implementation uses a collection of binomial trees [21]. Later, Kaplan and Zwick [10] simplified the soft heap data structure by using only a collection of heap-ordered binary trees, which are easier to implement. Slight modifications to this were made by Kaplan, Zwick and Tarjan [9], who also simplified much of the analysis. In this chapter, we discuss this implementation, as well as some differences between the three different versions. For completeness, we present the analysis of Kaplan, Zwick, and Tarjan. To make a better distinction between the Kaplan and Zwick implementation and the Kaplan, Zwick, and Tarjan implementation, we will refer to the former as the KZ implementation, and the latter as the KZT implementation.

2.1 Structure

The soft heap consists of heap-ordered binary trees. A node x may have a left child, $x.left$, and a right child, $x.right$, though one or both may be null. Each node has a rank, $x.rank$. The rank of a node does not dynamically change as the soft heap changes. This has a notable effect on deletions that we will discuss later. The root of each tree is connected in a singly-linked list. We call this the *root list*. If a node x is a root node, $x.next$ points to the next root in the root list. Note that each tree in the root list has a unique rank, so there is only one tree of a specific rank in the root list. Any operation that puts multiple trees on the root list with equal ranks, then these trees are linked together to form a tree of higher rank. We describe

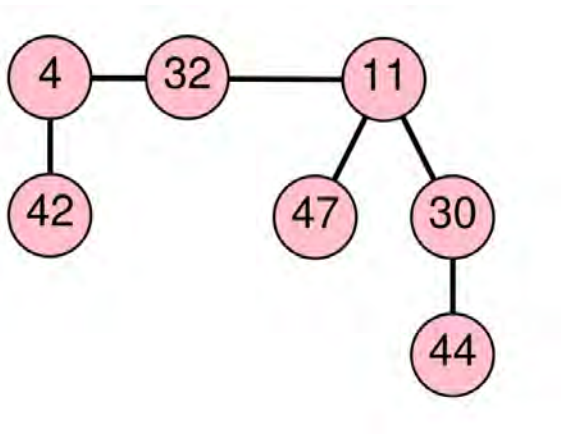


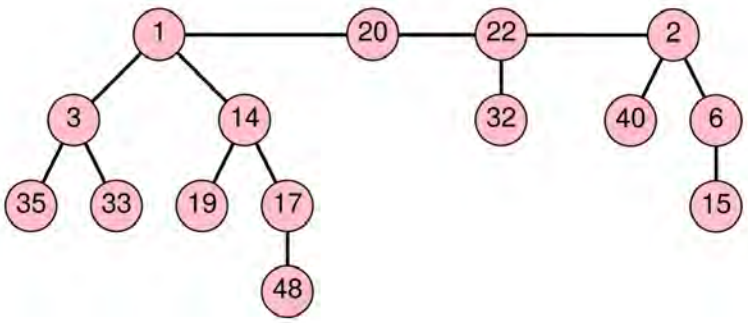
Figure 2.1: A Soft Heap with Three Heap-ordered Binary Trees

this process in more detail later in the chapter. We sometimes use the words "root node" and "tree" interchangeably to refer to either the root node, or the entire tree on the root list. A simple example of a soft heap can be seen in Figure 2.1.

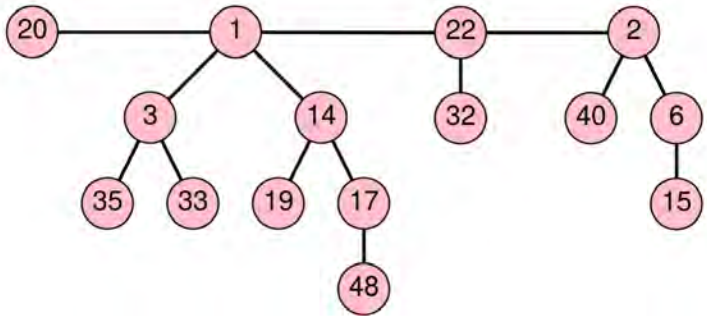
The root list is ordered depending on the operation being performed on the soft heap. There are two possible orderings for the root list: *findable order* and *meldable order*. When the root list is in findable order, it is ordered as follows: a tree x with rank r that has the minimum key as its root, followed by all trees with ranks less than r in increasing order, followed by all trees with ranks greater than r in findable order. This ordering is used when performing `find-min` or `delete-min`. Since the first tree in the root list has the minimum key in findable order, we can quickly find the node that contains the minimum key.

To perform `insert` and `meld`, we convert the root list into meldable order. Meldable order contains a tree with the minimum rank, followed by the remaining trees in findable order. By converting the root list to meldable order before melding or inserting, we can quickly resolve any rank conflicts by linking two trees, x and y, of equal rank together: by making a new tree z with $z.rank = x.rank + 1$.

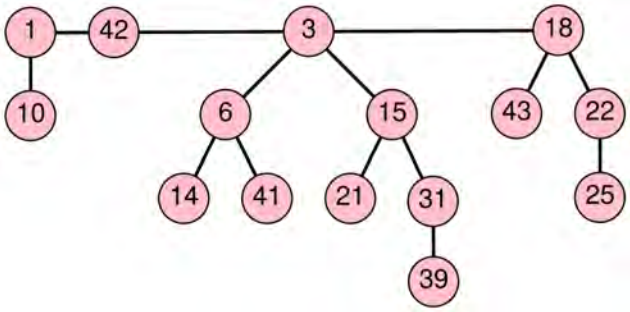
The soft heap can quickly convert from meldable order to findable order by simply swapping the first two trees if necessary. Let x be the tree with the smallest rank, and y be the tree with minimum key. In findable order, y is the first tree on the root list, followed by x. In meldable order, x is the first tree on the root list, followed by y. So swapping these trees allows



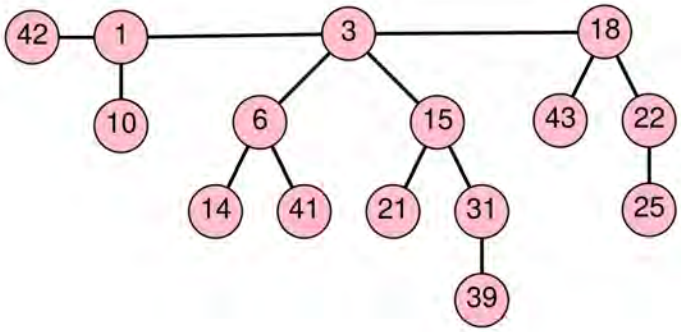
(a) Heap 1 in Findable Order



(b) Heap 1 in Meldable Order



(c) Heap 2 in Findable Order



(d) Heap 2 in Meldable Order

Figure 2.2: Two Soft Heaps Shown in Findable and Meldable Orders

us to change between meldable and findable orders. Note that no swap will be necessary if x is the tree with the smallest rank as well as the minimum key. Examples of soft heaps in findable and meldable orders can be seen in Figure 2.2.

In typical heaps and binary trees, a node holds just a single item. However, soft heap nodes may contain multiple items. These items belong to a node x's item list, $x.set$. Item lists are implemented as circular singly-linked lists. The item list is represented by a canonical item, which is the last item in the list, and therefore $x.set$ could interchangeably refer to the item list, or the last item on the list based on the context. If $x.set$ points to the last item e, then $e.next$ points to the first item in the list. Implementing item lists in this way enables constant time item movement by simply linking the end of one item list to the beginning of another. Considering that nodes in a soft heap may contain multiple items, we need a way to maintain heap-order. Each node x contains its own key, $x.key$, separate from the keys of the items in the node, which assumes the value of the highest key in its item list. The structure of a single node with multiple keys is shown in the figure below.

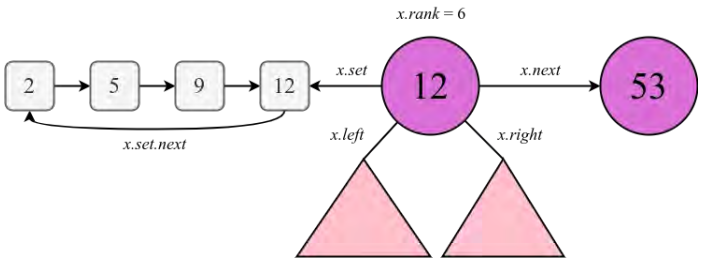


Figure 2.3: A Node x and its Data

When a node contains multiple items, we consider those items to be *corrupted*. As a result of corruption, items exit the soft heap out of order when we perform `delete-min`. Nodes accumulate multiple items during a *filling* operation, which moves items from a child node to its parent node. Calling a `fill` operation on a node x moves the item list of the child of x with the smallest key, say y, to $x.set$. This is done by concatenating $y.set$ to $x.set$, and x assumes the key of y. Then we recursively call `fill` on y. A second filling on x is done if $x.rank$ is greater than threshold t and $x.rank$ is even. This is called *double filling*. The threshold t is determined by the error rate ε, such that $0 \leq \varepsilon \leq 1$, and $t = \lceil \log \frac{3}{\varepsilon} \rceil$. An example

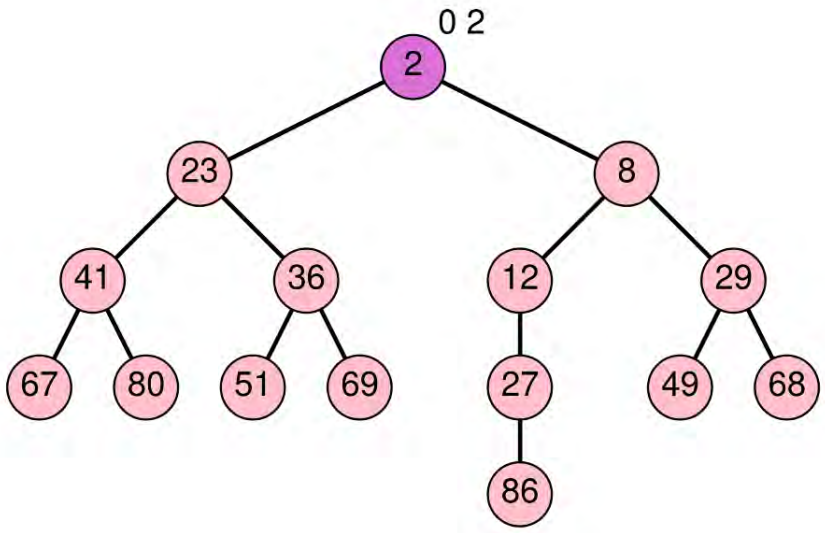


Figure 2.4: A Soft Heap with a Corrupted Item with Key 0.

of a soft heap with corrupted items can be seen in Figure 2.4. The soft heap allows the number of corrupted items to be at most εm, where m is the total number of insertions performed on the heap.

Note that deletions may cause more items to become corrupted. Even for heaps with as few as one hundred items, we found that over 90% of the items were corrupted with $\varepsilon = \frac{1}{2}$. These implementations are not technically incorrect, as each one asserts that at most εm corrupted items exist *inside* the heap. This *does not* include corrupted items that have already exited the heap through `delete-min`.

We propose an implementation of the soft heap that performs extra fillings only during insertions. Our implementation simply checks if the top-level operation being performed is an insertion, and only performs an extra filling if that and the aforementioned criteria are true. We find that the total number of corrupted items at any time (both in the heap and out of the heap) is at most εm, allowing us to use the soft heap as a black box. Note that this does not change the asymptotic complexities of all the operations, but increases the practical runtime overhead of all operations during fillings by testing this extra check.

2.2 Operations

The operation `make-heap` creates a single *Null node* of the soft heap. The *Null* node is a node with the same attributes as a normal node in the soft heap, but we set both its rank and its key to infinity, and all other attributes to the true null. Doing this simplifies condition checking throughout other operations. For example, given nodes x and y, if we wanted to check **if** $x.rank > y.rank$ **then** {...}, that will be the single condition rather than **if** $x \neq null$ **and** $y \neq null$ **and** $x.rank > y.rank$ **then** {...}.

Algorithm 1: `make-heap()`

```
1 return Null
```

Once we call `make-heap`, we have a pointer to a soft heap. As mentioned earlier, the soft heap is always kept in findable order. We can convert between findable order and meldable order by swapping the first two trees in the root list. Specifically, we call `rank-swap` to swap trees based on their ranks, to convert a soft heap in findable to meldable order, and call `key-swap` to swap trees based on their keys, to convert a soft heap in meldable order to findable order. Note that no trees are swapped if the tree containing the smallest key also contains the smallest rank.

Algorithm 2: `rank-swap(h)`

```
1 x ← h.next;
2 if h.rank ≤ x.rank then
3 |   return h
4 else
5 |   h.next ← x.next;
6 |   x.next ← h;
7 |   return x
8 end
```

We can think of `insert` as a wrapper function. Here, we toggle the *inserting* flag to allow double-fillings to occur. Given an item e, make e into a single element soft heap, then meld the new heap into the heap h. Before melding, heap h is converted into meldable order, and after the meld, the result is converted back into findable order.

Algorithm 3: `key-swap(h)`

```
1 x ← h.next;
2 if h.key ≤ x.key then
3 |   return h
4 else
5 |   h.next ← x.next;
6 |   x.next ← h;
7 |   return x
8 end
```

Algorithm 4: `insert(e, h)`

```
1 inserting = True;
2 r ← make-root(e);
3 h ← key-swap(meldable-insert(r, rank-swap(h)));
4 inserting = False;
5 return h
```

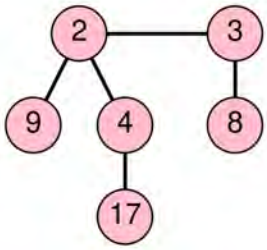
We use the `make-root` function to create a new root node. The new root holds just the item passed to the function, and the node assumes the item's key. Trees begin with a rank of zero, and we point any node pointers to the $Null$ node.

Algorithm 5: `make-root(e)`

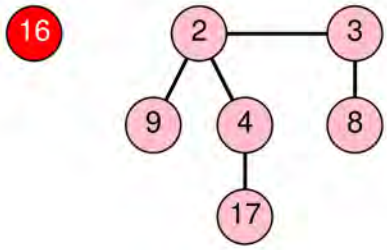
```
1 x ← new-node();
2 x.set ← e;
3 e.next ← e              ▷ link e to itself to form a circularly linked list;
4 x.key ← e.key;
5 x.rank ← 0;
6 x.left ← Null;
7 x.right ← Null;
8 x.next ← Null;
9 return x
```

Given tree x and tree h in meldable order, `meldable-insert` first checks if the rank of x is less than the rank of h. If so, we simply point $x.next$ to h, and return the resultant heap in findable order. (See Figure 2.5.) If $x.rank$ is not less than $h.rank$, we have a rank conflict. Since tree x was just created with a single key, we know $x.rank = 0$. This must mean that $h.rank = 0$ also, since we cannot have a node with negative rank. Since trees x and h have equal ranks, we must link them to create a new tree. (See Figure 2.6.)

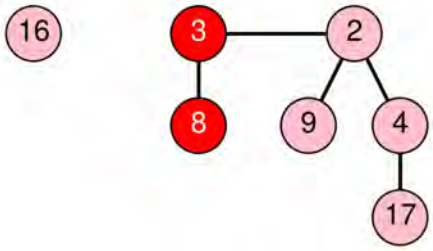
When linking two trees, we create a new root node, making both trees its children. The



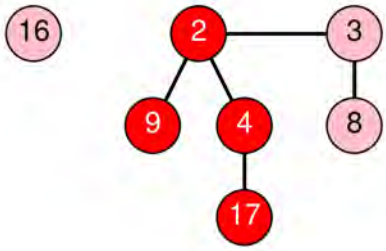
(a) The Heap Before Inserting



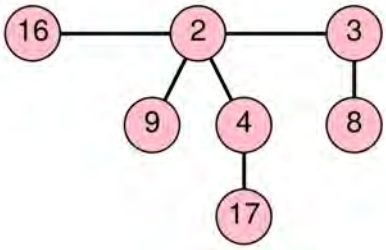
(b) Inserting Item with Key 16



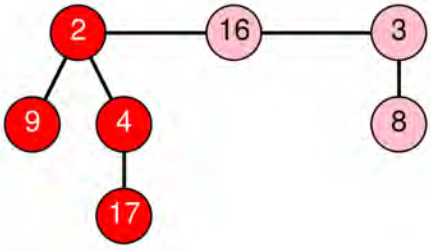
(c) Converting Heap to Meldable Order



(d) No Rank Conflict, Perform `key-swap`



(e) The Heap with 16 Now Inserted



(f) Findable Order is Restored

Figure 2.5: Inserting 16 into a Soft Heap

Algorithm 6: `meldable-insert(`x`, `h`)`

```
1 if x.rank < h.rank then
2 |   x.next ← key-swap(h);
3 |   return x
4 else
5 |   return meldable-insert(link(x, h), rank-swap(h.next))
6 end
```

rank of this node is one greater than the two children. The new node has no items yet, so we fill it using `defill`. When inserting an item with the key of 5 into the final heap shown in Figure 2.5, we can see a series of `link` operations being performed in the heap.

Algorithm 7: `link(`x`, `y`)`

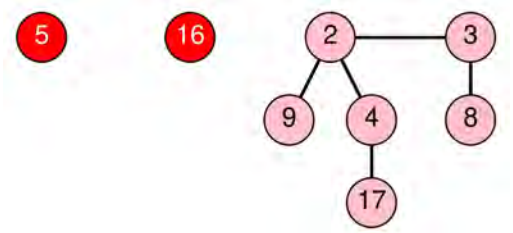
```
1 z ← new-node();
2 z.set ← null;
3 z.rank ← x.rank + 1;
4 z.left ← x;
5 z.right ← y;
6 defill(z);
7 return z
```

The `fill` and `defill` operations are most relevant to item corruption, and therefore the key operations of the soft heap. They are mutually recursive functions, where `defill` performs a single `fill` operation, then checks if it can do an extra filling. Given node x, call `fill` on x. If we are inserting, and $x.rank$ is even, and $x.rank > t$, perform an additional `fill` operation on x again.

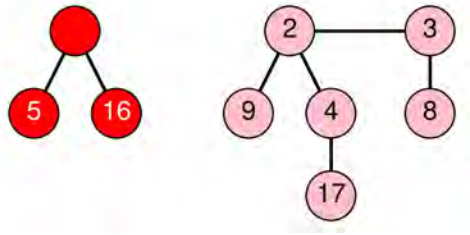
Algorithm 8: `defill(`x`)`

```
1 fill(x);
2 if inserting and x.rank > t and x.rank is even then
3 |   fill(x)
4 end
```

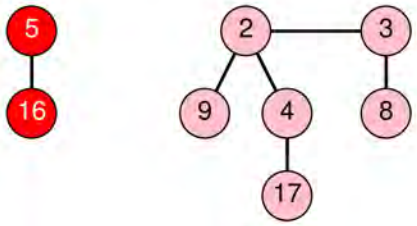
The `fill` operation performs the actual filling of a node. We choose to always fill a node x with items from $x.left$. To ensure that $x.left$ is the child with the minimum key, we swap the left and right children if necessary. Node x then assumes the key of $x.left$, increasing the key of x. Node x's item list may be empty if its items may have filled its parent node. If so, we simply set $x.set$ to $x.left.set$. Otherwise, we need to concatenate $x.left.set$ onto $x.set$.



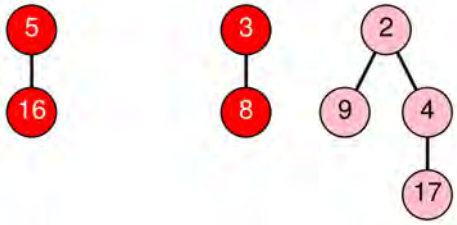
(a) Two Rank Zero Trees, x (Key 5) and y (Key 16), need to be Linked



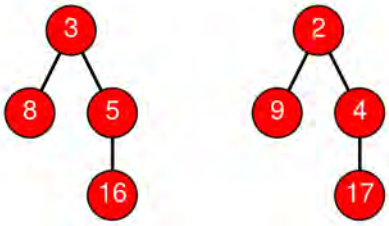
(b) A New Root is Created, with x as its Left Child and y as its Right Child



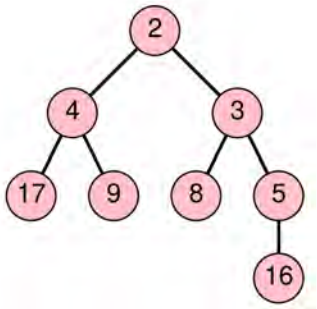
(c) The Root is Filled by the Left Child



(d) The Previous Linking Causes Another Rank Conflict



(e) Linking to Resolve the New Conflict Causes Another Conflict



(f) A Final `link` is Performed Leaving a Single-tree Heap

Figure 2.6: Inserting 5 Causes a Series of `link` Operations

We can do this by swapping pointers so that $x.set.next$ now points to $x.left$'s first item, and $x.left.set.next$ now points to x's first item. Node x then assumes the key of $x.left$, thereby corrupting the original items in its set that were present before the filling. Finally, we check if $x.left$ has a left child. If it does not, $x.right$ becomes $x.left$. Otherwise, we recursively perform `defill` on $x.left$. Figure 2.7 shows the steps of the `fill` operation, and Figure 2.8 shows the result of a double-even filling.

Algorithm 9: `fill(`x`)`

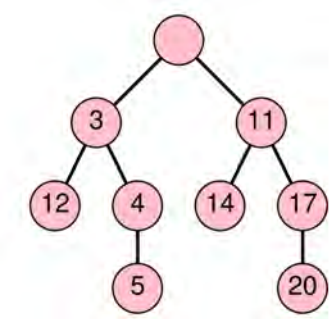
```
1  if x.left.key > x.right.key then
2  |  x.left ↔ x.right                          ▷ swap the left and right nodes
3  end
4  x.key ← x.left.key;
5  if x.set is empty then
6  |  x.set ← x.left.set
7  else
8  |  x.set.next ↔ x.left.set.next              ▷ concatenate x and x.left items
9  end
10 x.left.set ← Null;
11 if x.left.left = Null then
12 |  x.left ← x.right;
13 |  x.right ← Null
14 else
15 |  defill(x.left)
16 end
```

We can call `find-min` to get an item with minimum key. The soft heap is already in findable order, so the root with minimum key is at the start of the root list. `find-min` can return just the item, but we can also return the key of the root node. By returning both, we can determine if the item has been corrupted by comparing the item's key with the node's key. $h.set.next$ is simply the first item in h's item list, but any arbitrary item in the item list may be returned.

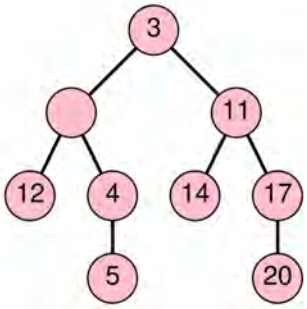
Algorithm 10: `find-min(`h`)`

```
1  return (h.set.next, h.key)
```

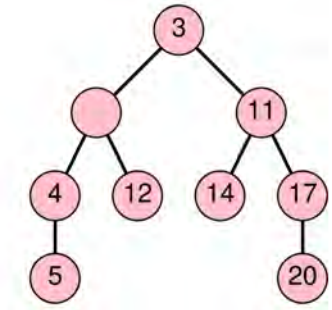
To remove an item with minimum key, we use `delete-min`. Again, the soft heap is already in findable order, so `delete-min` assumes the first tree has the minimum key. If there are multiple items in the minimum root h, then we simply return the first item, removing it



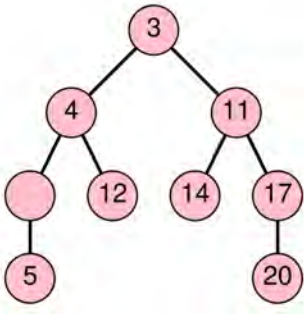
(a) Linking Two Rank-two Trees



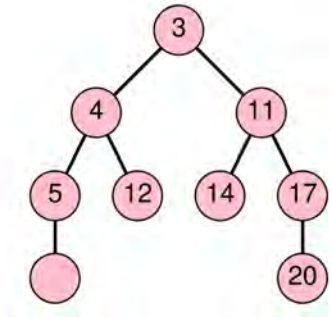
(b) The Root x is Filled by $x.left$



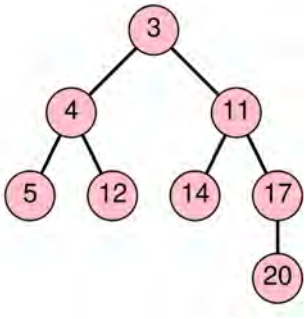
(c) Before Filling $x.left$, its Children Swap to Maintain Heap Order



(d) Node $x.left$ is Filled

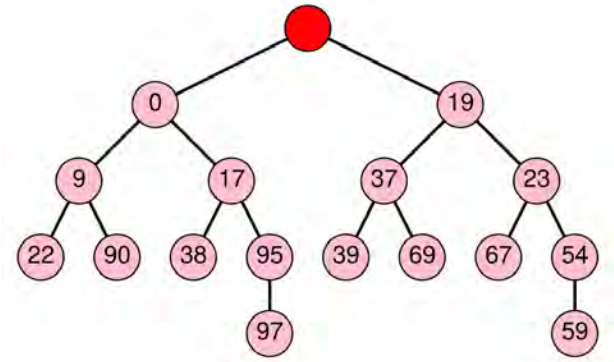


(e) Then $x.left.left$ is Filled

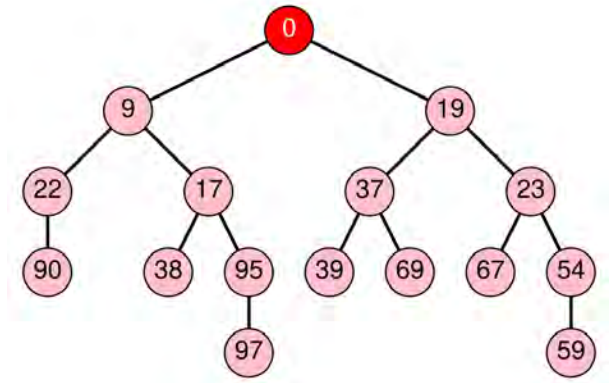


(f) The Leaf Node Cannot be Filled, so it is Deleted

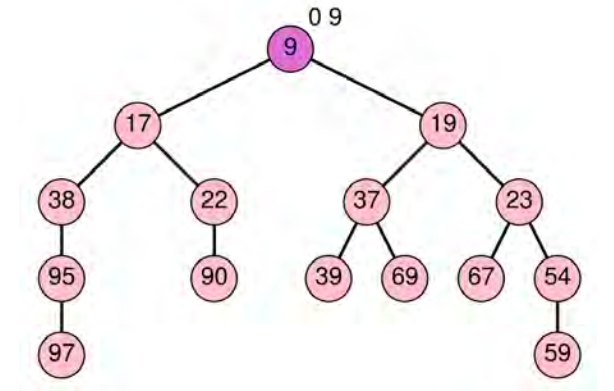
Figure 2.7: A Step-by-step Example of a `fill` Operation



(a) Linking Two Rank-three Trees Creates a Single Rank-four Tree



(b) A Single Filling is Performed, and Another One Begins on the Root Node



(c) The Second Filling is Completed

Figure 2.8: An Example of a Double-even Filling

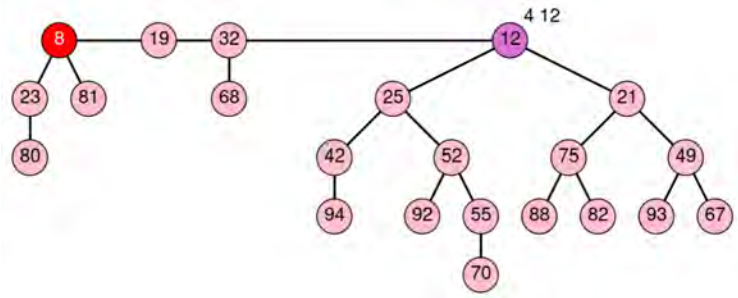
by updating pointers in the item list. Otherwise, we have a singleton node. We empty $h.set$, and store $h.rank$ in k. If h has no children, the tree is now empty, and we set $h = h.next$. If h does have children, we call `defill` on h. At this point, h now contains a different key, and the soft heap may no longer be in findable order. To restore findable order, we need to reorder the trees up to rank k, using the `reorder` operation.

Algorithm 11: `delete-min(`h`)`

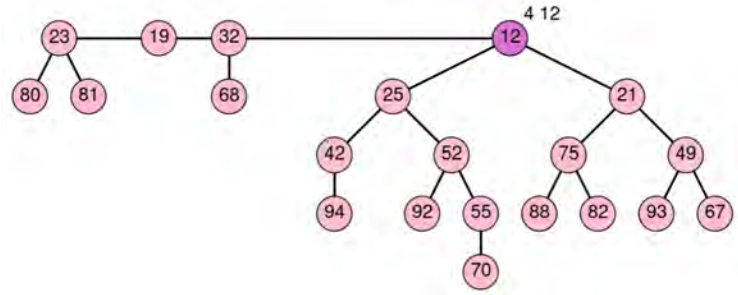
```
1  e ← h.set.next;
2  if e.next ≠ e then
3  |  h.set.next ← e.next
4  else
5  |  h.set ← null;
6  |  k ← h.rank;
7  |  if h.left = Null then
8  |  |  h = h.next
9  |  else
10 |  |  defill(h)
11 |  end
12 |  reorder(h, k)
13 end
14 return (e, h.key)
```

To `reorder` the trees and restore findable order, notice that all trees up to k are in increasing order of ranks, and all trees after rank k are in findable order. Therefore, we only need to traverse at most $k + 1$ trees to find the one with the smallest key and bring it back to the front of the root list. One way to accomplish this is in a recursive procedure as follows. We swap (using `rank-swap`) the first tree h with the next tree t in the root list if h's rank is smaller than t's. Then we recursively restore the rest of the trees in findable order. Once out of recursion, we have a pointer t to the first tree returned in findable order among the remaining trees. We can then swap h with t if t's key is smaller than h's key (using `key-swap`). This restores the findable order of all the trees on the root list. Unrolling the recursive process, we are swapping h successively with all trees with ranks less than k. Then the minimum key is either in h or $h.next$, and the tree with the minimum key is brought to the front. Then we swap from back to front the tree with the smallest key, ensuring that the tree with the smallest key is at the front of the root list. This ensures the findable order of trees. (See Figure 2.9.)

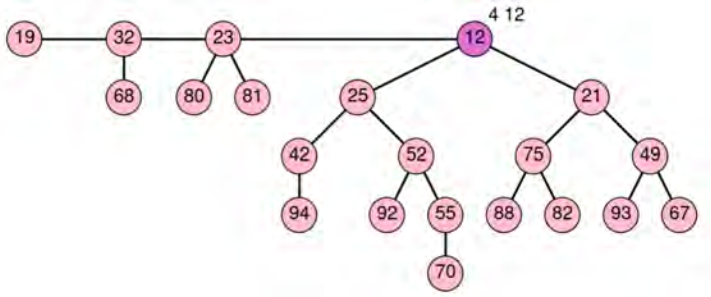
Finally, we have `meld`. Like `insert`, it can also be thought of as a wrapper function.



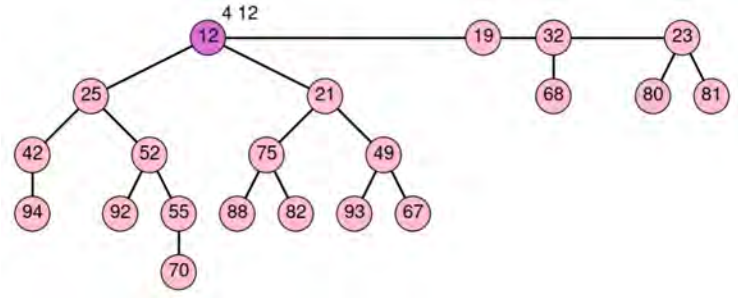
(a) A Soft Heap Before a Call to `delete-min`. Item with Key 8 Will be Deleted.



(b) The Item is Removed, Causing the Root of the First Tree to be Filled. The First Tree Remains a Rank-two Tree.



(c) The Root List is Reordered by Swapping the First Tree to its Correct Position by Rank



(d) Findable Order is Restored by Swapping the Tree with Minimum Key to the Front

Figure 2.9: An Example of `delete-min`.

Algorithm 12: `reorder(`h`, `k`)`

```
1 if h.next.rank < k then
2 |   h ← rank-swap(h);
3 |   h.next ← reorder(h.next, k)
4 end
5 return key-swap(h)
```

The `meld` operation converts both heaps to meldable order before melding, then converts the newly melded heap to findable order. It uses the helper function `meldable-meld` to do the actual melding before returning the root list back into findable order.

Algorithm 13: `meld(`h_1`, `h_2`)`

```
1 h1 ← rank-swap(h1);
2 h2 ← rank-swap(h2);
3 return key-swap(meldable-meld(h1, h2))
```

The `meldable-meld` function is given two nodes h_1 and h_2 as input, where h_1 points to the first tree on the first heap, and h_2 points to the first tree on the second heap. Without loss of generality, let h_1 be the root with the smaller rank. If h_2 is $Null$, then the result of the merged heap is just h_1. Otherwise, we recursively meld $h_1.next$ with h_2, and get a pointer to the first node h on the root list. We finally perform a single `meldable-insert` to insert a single tree h_1 into h.

Algorithm 14: `meldable-meld(`h_1`, `h_2`)`

```
1 if h1.rank > h2.rank then
2 |   h1 ↔ h2
3 end
4 if h2 = Null then
5 |   return h1
6 else
7 |   h ← meldable-meld(rank-swap(h1.next), h2);
8 |   return meldable-insert(h1, h)
9 end
```

2.3 Analysis

Let m be the total number of insertions made in the soft heap, and d be the total number of deletions. (Specifically, the total number of calls to `insert` and `delete-min`, respectively.) We let the error rate ε be $0 \leq \varepsilon \leq 1$, and the threshold $t = \lceil \log \frac{3}{\varepsilon} \rceil$. The analysis of the soft heap data structure consists of two parts: (i) show that there are only εm corrupted or bad items at any point in the heap, and (ii) show that `delete-min` takes $O(\log \frac{1}{\varepsilon})$ amortized time, while all other soft heap operations run in only $O(1)$ amortized time. Both of these depend on finding bounds on the number and size of a tree of rank k.

2.3.1 Bounds on the Number and Size of a Tree of Rank k

Lemma 1. *The number of trees of rank k is at most $\frac{m}{2^k}$.*

Proof. We can prove this by induction on k. As a base case, we have $k = 0$, and the number of rank 0 trees is at most $\frac{m}{2^0} = m$. This is trivially true, since there cannot be more than m trees of rank 0. Now assume that the statement is true for any tree of rank less than k. Then the number of trees with ranks $k-1$ is at most $\frac{m}{2^{k-1}}$. Since we combine two trees of ranks $k-1$ together to form a tree of rank k, we have only at most $\frac{\frac{m}{2^{k-1}}}{2} = \frac{m}{2^k}$ trees of rank k. □

Lemma 2. *Let $s(k)$ denote the size (or the number of items) in a node of rank k. If $k \leq t$, then $s(k) = 1$. Otherwise, $s(k) \leq 2^{\lceil (k-t)/2 \rceil}$.*

Proof. Consider the case when $k \leq t$. When a new node is formed with a single item, we have a rank 0 node with only one item. A node of rank k is only filled when it is emptied, therefore we only move items from a node of rank $k-1$. By induction, since $s(k-1) = 1$, we have $s(k) = 1$.

Now, consider the case when $k > t$. If k is odd, we only perform a single filling, and therefore $s(k) \leq s(k-1)$. When k is even, we perform a double filling, so $s(k) \leq 2s(k-1)$. So every two levels above the threshold t, the number of items in a node exponentially increases. In general, we can express this as $s(k) \leq 2^{\lceil (k-t)/2 \rceil}$, where $k-t$ is the number of levels above the threshold. Table 2.1 shows this exponential growth. More formally, one can prove the above quantity by induction on $k-t$, which is the number of levels above t. We leave this formal proof as an exercise for the reader.

Rank k	Upper Bound on $s(k)$
$t+8$	16
$t+7$	8
$t+6$	8
$t+5$	4
$t+4$	4
$t+3$	2
$t+2$	2
$t+1$	1
t	1
t - 1	1
t - 2	1

Table 2.1: The size of nodes with rank k relative to t, where t is even.

□

2.3.2 Bounds on The Number of Corrupted Items

Lemma 3. *Let the rank of an item be defined as the rank of the node it belongs to. Then the number of items with rank $k > t$ is at most εm.*

Proof. The number of items of any rank $k > t$ is

$$\sum_{k>t} \frac{m}{2^k} \cdot s(k)$$

Expanding the terms in the above summation and taking the sum to infinity gives

$$\begin{aligned} \sum_{k>t} \frac{m}{2^k} \cdot s(k) &\leq m\left[\frac{s(t+1)}{2^{t+1}} + \frac{s(t+2)}{2^{t+2}} + \frac{s(t+3)}{2^{t+3}} + ...\right] \\ &= \frac{m}{2^t}\left[\frac{s(t+1)}{2^1} + \frac{s(t+2)}{2^2} + \frac{s(t+3)}{2^3} + ...\right]. \end{aligned}$$

We can generalize this summation again, this time differentiating between items of odd and even ranks.

$$\frac{m}{2^t}\sum_{i\geq 1}\left(\frac{s(t+2i-1)}{2^{2i-1}} + \frac{s(t+2i)}{2^{2i}}\right)$$

We can simplify by calculating the size of even and odd ranks separately, since

$$s(t+2i-1) \leq 2^{\lceil (t+2i-1-t)/2 \rceil} = 2^{\lceil (2i-1)/2 \rceil} = 2^{(2i)/2} = 2^i$$

$$s(t+2i) \leq 2^{\lceil (t+2i-t)/2 \rceil} = 2^{\lceil (2i)/2 \rceil} = 2^{(2i)/2} = 2^i$$

The size of an even rank is just twice that of the previous odd rank, so we multiply the size of the odd rank by two to give the same denominator for both ranks, allowing us to combine them into a single quantity. So we get

$$\begin{aligned}
\sum_{k>t} \frac{m}{2^k} \cdot s(k) &\leq \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{2^i}{2^{2i-1}} + \frac{2^i}{2^{2i}} \right) \\
&= \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{1}{2^{2i-1}} + \frac{1}{2^{2i}} \right) \cdot 2^i \\
&= \frac{m}{2^t} \sum_{i \geq 1} \frac{2+1}{2^{2i}} \cdot 2^i \\
&= \frac{m}{2^t} \sum_{i \geq 1} \frac{3}{2^i} \\
&= \frac{3m}{2^t} \sum_{i \geq 1} \frac{1}{2^i}.
\end{aligned}$$

It is well known that the geometric series $\sum_{i \geq 1} \frac{1}{2^i}$ solves to 1. So, we have

$$\sum_{k>t} \frac{m}{2^k} \cdot s(k) \leq \frac{3m}{2^t}$$

Replacing t with $\log \frac{3}{\varepsilon}$, we get

$$\frac{3m}{2^t} = \frac{3m}{2^{\log 3/\varepsilon}} = \frac{3m}{\frac{3}{\varepsilon} \log 2} = \frac{3m}{\frac{3}{\varepsilon}} = \varepsilon m.$$

□

Theorem 4. *The total number of corrupted items in a soft heap is at most εm.*

Proof. From Lemma 3, the total number of items in a soft heap with ranks $k > t$ is at most εm. Since only items above the threshold t can ever be corrupted by double fillings, the total number of corrupted items is also at most εm. □

2.3.3 Analysis of Fillings in Soft Heap Operations

Since fillings are an integral part of the analysis of soft heap operations, we first discuss the total number of fillings performed. Since fillings performed at nodes with ranks $k \leq t$ differs from those performed at ranks above t, we analyze them separately. We call these *low fillings* and *high fillings* respectively.

In order to achieve an amortized runtime for soft heap operations, we charge extra units for low fillings to pay for both the cost of high fillings and the cost of deletions. When an item fills its parent, it is charged one additional unit. Once the item fills a node with rank t, it is charged one final additional unit so that it now has an additional t units of charge. Since we only charge extra for low fillings, any item has at most t units of charge, and an item with rank $k \leq t$ has k units of charge. This process is illustrated in Figure 2.10.

Lemma 5. *The total charge of all the low fillings performed is $O(td + m)$.*

Proof. To determine the total charge among all low fillings, we can take the number of nodes with rank $k \leq t$, the number of items a node has at rank k, and how much charge each item in a node has been charged. Since we are only considering low fillings, we know $s(k) = 1$, leaving us with a charge that amounts to

$$\sum_{k=0}^{t} \frac{m}{2^k} \cdot s(k) \cdot k = \sum_{k=0}^{t} \frac{m}{2^k} \cdot k = m \sum_{k=0}^{t} \frac{k}{2^k}.$$

The expression $\sum_{k=0}^{t} \frac{k}{2^k}$ can be added to infinite terms to get an upper bound. This is a famous arithmetico-geometric series that sums to 2. The Appendix contains proofs of this series, as well as sums of other common geometric series. So the total charge among all low fillings is at most $2m = O(m)$. Note that this only includes all charges of items currently in the heap, and excludes the charges of items that exited the heap through `delete-min` operations.

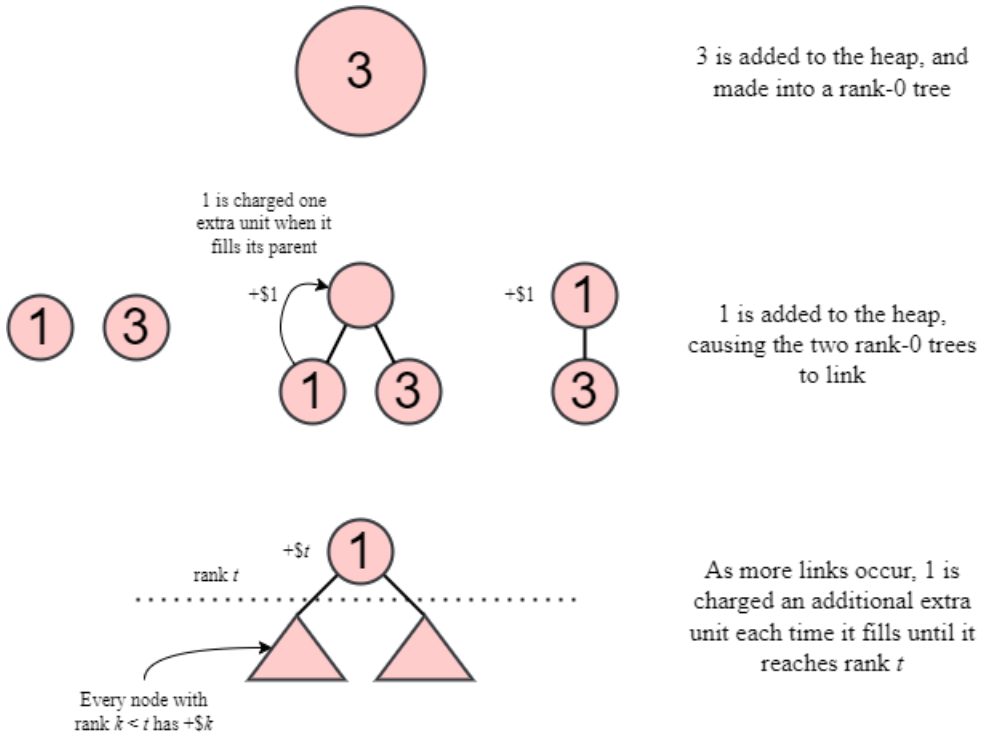


Figure 2.10: Distribution of Charges in a Soft Heap Tree

Since d items were deleted from the heap, they must have at most td total charge. Therefore, the total charge of all low fillings is $O(td + m)$. □

Note that the charge among low fillings directly corresponds to total work done among the low fillings.

Corollary 6. *The total number of (or the total work done among) all low fillings is* $O(td+m)$.

Lemma 7. *The total charge of all the nodes involved in high fillings is* $O(m)$.

Proof. From Lemma 3, the total number of items with rank $k > t$ is at most εm. Each of these has t units of charge. So the total charge is $t\varepsilon m$, which is at most $3m = O(m)$ since $0 \leq \varepsilon \leq 1$. □

Let $f(k)$ be the maximum number of fillings on a node of rank k. That is, among all nodes of rank k, $f(k)$ is the maximum number of times `defill` is called. We use the bounds on $f(k)$ to analyze the total number of high fillings. We also have $f(0) = 1$, which amounts

to the work done by calling `make-root` to create a new rank 0 tree. Note that each time a `fill` operation is called on a node x, exactly one node gets destroyed from x's subtree. So, the maximum number of times we can call `fill` on a node is upper bounded by the maximum number of nodes a tree of rank k can have. The last call to `fill` destroys node x, after all nodes in its subtree are destroyed.

Lemma 8. *If $k \geq 1$, we have $f(k) \leq 2f(k-1)$. If $k > t$, and k is even, then $f(k) \leq f(k-1)$.*

Proof. Let $g(k)$ be the maximum number of fillings on a node x of rank k, excluding fillings when $g(k)$ is the root. Since $f(k)$ also includes fillings when x is the root, we have $f(k) \geq g(k)$. Every time `defill` is called on x, we trigger a `defill` on one of the children of x. The number of times we can call `defill` on x is therefore bounded by the number of times we can call `defill` on both of the children of x. So $f(k) \leq 2g(k-1) \leq 2f(k-1)$. If $k > t$, and k is even, then one call to `defill` on x calls `fill` twice on x, which triggers two calls to `defill` on the children of x. The only time `defill` calls `fill` once is when node x gets destroyed before the second call. So $2f(k) - 1 \leq 2g(k-1) \leq 2f(k-1)$. Simplifying, we get $2f(k) \leq 2g(k-1) + 1 \leq 2f(k-1)$. So $f(k) \leq f(k-1)$. □

Lemma 9. *If $k \leq t$, then $f(k) \leq 2^k$. If $k > t$, then $f(k) \leq 2^{\lceil (k+t)/2 \rceil}$.*

Proof. It is easy to prove this by induction on k from Lemma 8, and we leave this as an exercise for the reader. Intuitively, there is an exponential increase in $f(k)$ for any $k \leq t$. If $k > t$, then there is an exponential increase in $f(k)$ between every two levels above t. At level t, we have $f(t) \leq 2^t$. If $k > t$, then there will be exponential increases $(k-t)/2$ times. So we have $f(k) \leq 2^t \cdot 2^{\frac{(k-t)}{2}} \leq 2^{\frac{(k+t)}{2}}$.

□

Theorem 10. *The total number of all high fillings is $O(m)$.*

Proof. From Lemmas 1 and 8, the total number of high fillings is

$$
\begin{aligned}
\sum_{k>t} \frac{m}{2^k} \cdot f(k) &= m \sum_{k>t} \frac{f(k)}{2^k} \\
&\leq m \left[\frac{f(t+1)}{2^{t+1}} + \frac{f(t+2)}{2^{t+2}} + \frac{f(t+3)}{2^{t+3}} + ... \right] \\
&= m \sum_{i \geq 1} \left(\frac{f(t+2i-1)}{2^{t+2i-1}} + \frac{f(t+2i)}{2^{t+2i}} \right) \\
&\leq m \sum_{i \geq 1} \left(\frac{2^{t+i}}{2^{t+2i-1}} + \frac{2^{t+i}}{2^{t+2i}} \right) \\
&= \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{2^{t+i}}{2^{2i-1}} + \frac{2^{t+i}}{2^{2i}} \right) \\
&= \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{1}{2^{2i-1}} + \frac{1}{2^{2i}} \right) \cdot 2^{t+i} \\
&= \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{2+1}{2^{2i}} \right) \cdot 2^{t+i} \\
&= \frac{2^t \cdot 3m}{2^t} \sum_{i \geq 1} \frac{1}{2^i \cdot 2^i} \cdot 2^i \\
&= 3m \sum_{i \geq 1} \frac{1}{2^i} \\
&= 3m \\
&= O(m).
\end{aligned}
$$

□

Note that the total work done over all high fillings does not depend on the number of delete operations performed, and is solely based on the number of elements inserted. This is because a node of rank k can only be filled $f(k)$ times regardless of whether the `insert` or the `delete-min` operation initiates the filling.

Theorem 11. *The total work done over all fillings is $O(td + m)$.*

Proof. This follows from Corollary 6 and Theorem 10. □

2.3.4 Analysis of Soft Heap Operations

It is easy to determine that `make-heap` and `find-min` take $O(1)$ in the worst case. For `delete-min`, if the root x with minimum key is not emptied by deletion, it takes $O(1)$ time in the worst case. If x is emptied, however, then it gets filled, and the root list must be reordered. If x has rank k, the time it takes to reorder the root list takes $O(k)$ time. We show that reordering the root list takes $O(m)$ time over a sequence of m `insert` operations, and therefore can be amortized.

Theorem 12. *Excluding filling, the time spent doing reorderings is $O(td + m)$. Therefore, excluding filling, the total time spent on deletions is $O(td + m)$.*

Proof. If a tree of rank k is reordered, then it takes $O(k)$ total time. More specifically, we need to swap $k+1$ trees in the root list during each reordering. Each `delete-min` operation pays to reorder t trees in the root list. This amounts to $O(td)$ time. The remaining total work over all `delete-min` operations for reordering can be computed using the bounds of $f(k)$ developed in Lemma 9 as well as Lemma 1;

$$\begin{aligned}
\sum_{k>t} \frac{m}{2^k} \cdot f(k) \cdot (k-t) &= m \sum_{k>t} \frac{f(k)}{2^k} \cdot (k-t) \\
&\leq m \sum_{i\geq 1} \left(\frac{2^{t+i} \cdot (2i-1)}{2^{t+2i-1}} + \frac{2^{t+i} \cdot (2i)}{2^{t+2i}} \right) \\
&= \frac{m}{2^t} \sum_{i\geq 1} \left(\frac{2i-1}{2^{2i-1}} + \frac{2i}{2^{2i}} \right) \cdot 2^{t+i} \\
&= m \sum_{i\geq 1} \left(\frac{4i-2+2i}{2^{2i}} \right) \cdot 2^i \\
&= m \sum_{i\geq 1} \frac{2(3i-1)}{2^i} \\
&\leq 6m \sum_{i\geq 1} \frac{i}{2^i} \\
&= 12m \\
&= O(m).
\end{aligned}$$

□

Theorem 13. *Excluding filling, the total time to perform insertions and melds is $O(1)$ amortized.*

Proof. To show this, we use a potential function argument popularized by [20]. The potential of a single heap is the number of trees plus the largest node rank, and the potential of multiple heaps is the sum of each heap's potential. Since `insert` is melding a heap with a single rank-0 tree, we analyze melding two heaps with h_1 trees and h_2 trees, with the largest rank trees r_1 and r_2, respectively. Without loss of generality, assume $r_1 \leq r_2$. To meld these two trees, we scan at most r_1 trees on both heaps, and link at most k trees, and spend one extra unit of time melding a null tree in the base case. So the total work done in melding two heaps is at most $r_1 + k + 1$, scaling constants appropriately. The amortized cost of the j^{th} operation, denoted a_j, is the actual cost of the j^{th} operation, denoted c_j, plus the change in potential. We can write this as

$$a_j = c_j + (\phi_j - \phi_{j-1})$$

where ϕ represents our potential function. Note that ϕ_j is the potential that exists in the data structure after performing the j^{th} operation, while ϕ_{j-1} is the potential before the j^{th} operation. Expanding c_j and ϕ_j, we get

$$\begin{aligned} a_j &= r_1 + k + 1 + (r_2 + 1 + h_1 + h_2 - k - (h_1 + h_2 + r_1 + r_2) \\ &= 2 \\ &= O(1). \end{aligned}$$

□

Theorem 14. *Starting with a empty heap, any sequence of soft heap operations that consist of m insertions, and d deletions takes $O(td + m)$ total time.*

Proof. This follows from Theorems 11, 12, and 13. □

Corollary 15. *The amortized cost of insert is $O(1)$, and the amortized cost of deletes is $O(t)$.*

Work	`insert`	`delete-min`
`make-root`	1	
low filling	1	t
high filling	1	
reordering during `delete-min`	1	t
total	$O(1)$	$O(t)$

Table 2.2: Table of Distribution of Work Between `insert` and `delete-min`

Scaling work differently, we can think of operations `insert` and `delete-min` contributing to the work in Table 2.2.

2.4 Comparison of Implementations

Chazelle's original implementation [3] of the soft heap uses a collection of binomial heaps [21]. This implementation was simplified to use a collection of heap-ordered binary trees in both KZ [10] and KZT [9]. However, where the KZT implementation keeps the trees in either findable or meldable order, both the Chazelle and KZ implementations order the trees with respect to rank, and augment each root with a `suffix-min` pointer. Given a tree x of rank k, `suffix-min` points to the tree with the minimum key amongst trees with rank $\geq k$. If this happens to be x, then x simply points to itself. This allows the trees to always be in a meldable state, though anytime the heap updates, all `suffix-min` pointers must be updated.

The root list in both the Chazelle and KZ implementations are doubly-linked, whereas the KZT implementation is singly-linked. Using a doubly-linked root list is helpful when updating `suffix-min` pointers, though it requires additional upkeep to maintain the root list when inserting or removing a tree.

The Chazelle and KZ implementations have a "rank invariant" and "target size" for nodes, respectively. Chazelle's rank invariant requires that the number of children of a root node is no smaller than half of its rank. When a rank invariant violation is detected, it "dismantles" the tree, and melds its children back into the heap. A node's target size in the KZ implementation is the number of items a node should contain. Should the number of items in a node drop below half of its target size, then items from its left child fill it, if possible.

These invariants are used to bound the number of corrupted items in a soft heap, as well as in analyzing the runtime complexities of the soft heap operations. Therefore, we feel that the KZT implementation is the simplest in terms of both its implementation and the analysis.

Chapter 3

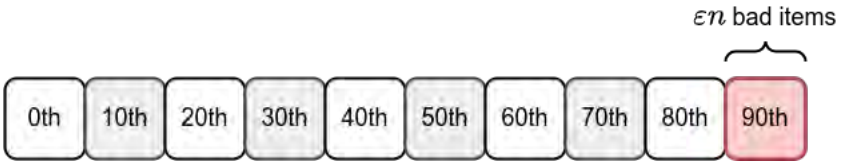
Soft Heap Applications

The main motivation for soft heaps is in computing a minimum spanning tree of a graph [2]. Given a connected graph with n nodes and m weighted edges, Chazelle's algorithm finds a minimum spanning tree of the graph in only $O(m\alpha(m,n))$ time, where α is the functional inverse of the Ackermann function. Where the Ackermann function grows extraordinarily quickly, its inverse grows very slowly over time. A discussion of this application is slightly beyond the scope of this thesis, and interested readers can refer to [2] for more details. In this chapter, we discuss other applications in which soft heaps can provide us with reasonably fast solutions.

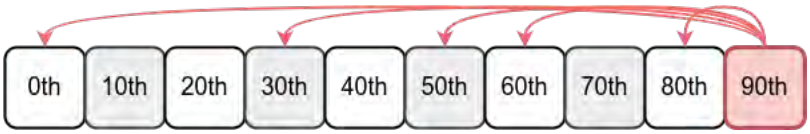
3.1 Dynamic Percentile Maintenance

The soft heap can be used to obtain an item within some percentile. Suppose we have some list of items, and we want to select an item within the 90^{th} percentile. We can use a max soft heap, in which the heap property is reversed, and we have a `delete-max` operation to pull out the largest key in the heap. Note that corruption in a max soft heap means that some keys may be artificially decreased. We set $\varepsilon = \frac{1}{10}$. After inserting all items into the heap, we can delete a single item and know that it is within the 90^{th} percentile.

Let us assume the worst case, where all items within the first εn items removed are corrupted. If we let $n = 100$, then only $\varepsilon n = 10$ items may be corrupted. Even if all ten items in the 90^{th} percentile have their keys decreased due to corruption, the largest item removed will only be smaller than εn items, and therefore lies in the 90^{th} percentile. (See Figure 3.1.)



(a) All Corrupted Items Within 90th Percentile



(b) 90th Percentile is At Most εn Items Away from True Rank

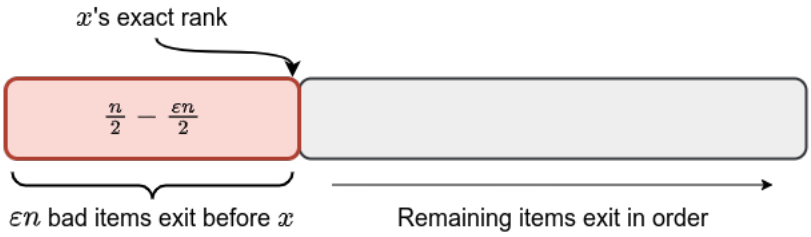
Figure 3.1: Approximate Median: Worst Case Corruption

Since $\varepsilon = O(1)$, both `insert` and `delete-max` operations run in $O(1)$ amortized time. So the entire algorithm takes only linear time.

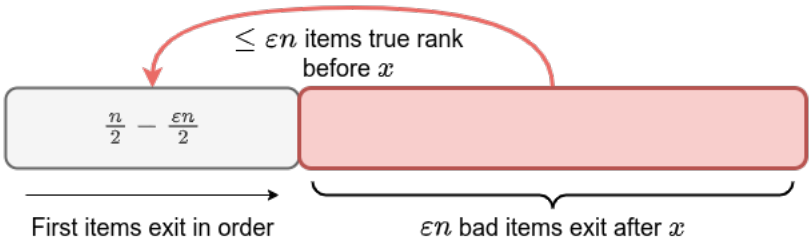
3.2 Approximate Median Finding

Finding the approximate median of a list of items can be done similarly to the percentile maintenance application. Using a min soft heap, suppose we set $\varepsilon = \frac{1}{10}$, then insert n items into the heap. We then perform $\frac{n}{2} - \frac{\varepsilon n}{2}$ deletions. Let x be the largest item amongst those removed.

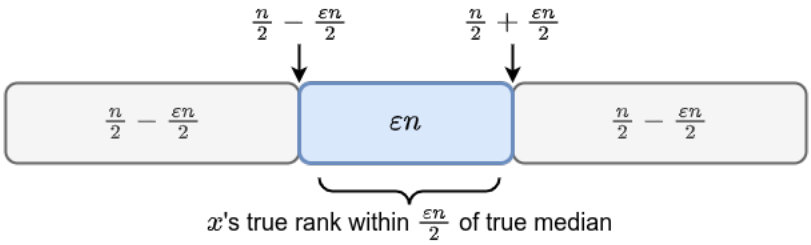
We know that x is larger than at least $\frac{n}{2} - \frac{\varepsilon n}{2}$ elements. If all the corrupted items were removed from the soft heap, then x's rank is exactly $\frac{n}{2} - \frac{\varepsilon n}{2}$, as shown in Figure 3.2(a). If none of the items removed were corrupted and still remained in the heap, then in the worst case, all of their original keys could be smaller than x. (See Figure 3.2(b).) Therefore, there are at most $\frac{n}{2} - \frac{\varepsilon n}{2} + \varepsilon n = \frac{n}{2} + \frac{\varepsilon n}{2}$ items smaller than x. So x's rank lies between $\frac{n}{2} - \frac{\varepsilon n}{2}$ to $\frac{n}{2} + \frac{\varepsilon n}{2}$. (See Figure 3.2(c).) By choosing an arbitrarily small constant for ε, we can find a value of x closer and closer to the true median. It can be seen that the entire algorithm takes linear time. This algorithm to find an approximate median is clearly simpler than Floyd's classic deterministic selection algorithm [6].



(a) Item x's True Rank is $\frac{n}{2} - \frac{\varepsilon n}{2}$ if All Corrupted Items Exit Before x



(b) Each Corrupted Item May Have a True Rank Before x



(c) Item x's True Rank Lies Within $\frac{n}{2} - \frac{\varepsilon n}{2}$ and $\frac{n}{2} + \frac{\varepsilon n}{2}$

Figure 3.2: Finding the Approximate Median.

3.3 Approximate Sorting

The application of approximate or near sorting is perhaps more obvious, given the nature of the soft heap. There are two different ways in which we can define "near-sortedness". One is based on inversions, and the other based on the bounds on the ranks of each element. We discuss these in this section.

3.3.1 Near-sortedness Based on Inversions

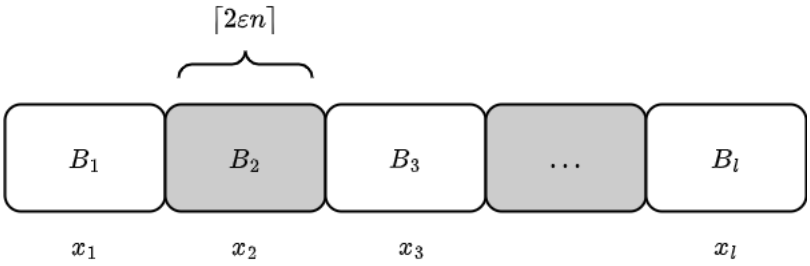
With some error rate ε, insert n items into the heap, then delete all n items. When an item x with rank k is deleted from the heap, there may be at most εn items with smaller keys corrupted, that are removed after x. So, item x can only have at most $I_k = \varepsilon n$ inversions. Adding up over all items x, the total number of inversions in the array is at most

$$I = \sum_{k>0}^{n} I_k = \sum_{k>0}^{n} \varepsilon n = \varepsilon n \sum_{k>0}^{n} 1 = \varepsilon n \cdot n = \varepsilon n^2.$$

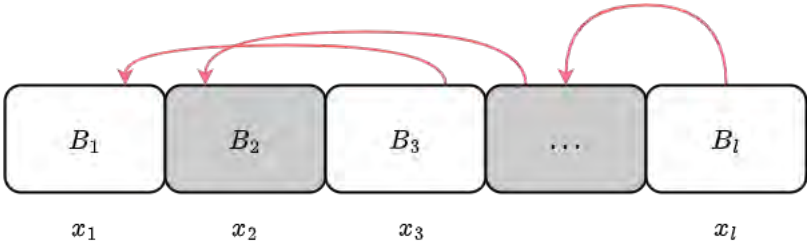
3.3.2 Near-sortedness Based on Ranks

Another version of near-sortedness ensures that each item x appears in the output array within certain bounds of its true rank. For any value of ε, we can perform n insertions followed by n deletions. Now, we divide the output array into a series of l blocks, each containing $\lceil 2\varepsilon n \rceil$, so that $n = \lceil 2\varepsilon n \rceil l$. The last block may contain fewer items. Let the blocks be labeled $B_1, \ldots, B_l$, and let x_i be the smallest original key amongst uncorrupted items in block B_i.

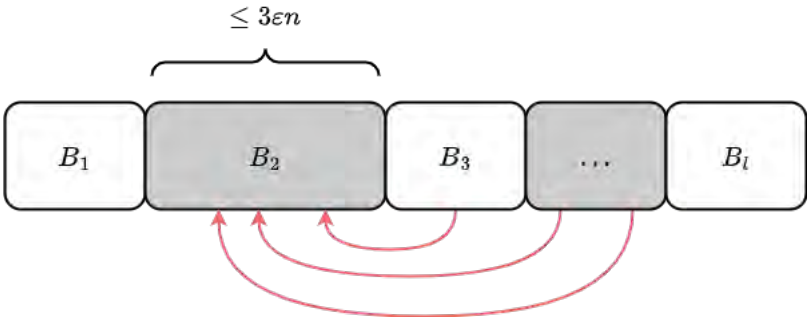
Since only at most εn items can be corrupted, each block contains at least εn uncorrupted items. Therefore, the x_i's, where $1 \leq i \leq l$, are in sorted order. (See Figure 3.3(a).) Now we scan through all the items in the array, and place each item e in its appropriate block. Note that we only have to move uncorrupted items to their correct blocks. Let e_i be an item in block B_i. If e_i is smaller than x_i, it is in the incorrect block. Scan the blocks before B_i until an x_j is found such that $e_i \geq x_j$, where $1 \leq j < i$. Then e_i belongs to block b_j. (See Figure 3.3(b).) Since only up to εn items may be corrupted, this entire process takes $O(\varepsilon n l) = O(n)$ time.



(a) Output of a Soft Heap Divided into l Blocks



(b) Each Corrupted Item Moves to Appropriate Block



(c) Each Item is at Most $3\varepsilon n$ Items from its True Rank

Figure 3.3: Moving Corrupted Items to their Correct Block

Once all items are in their correct block, we have our near-sorted array. We know that each block starts with $\lceil 2\varepsilon n \rceil$ items, and at most εn corrupted items could move into a block. So the largest block has at most $3\varepsilon n$ items. Therefore, each item in the list is now guaranteed to be within $3\varepsilon n$ items from its true rank. (See Figure 3.3(c).)

Chapter 4

Visualizing Soft Heaps

4.1 The Visual Tool

We have written a visualization library in TypeScript [15], which builds on top of the Cytoscape.js library [7]. Cytoscape.js allows developers to visualize graphs using JavaScript. Our visual tool provides functions that wrap some of Cytoscape's features, making graphs (and therefore data structures like the soft heap) easier to visualize. It also adds new functionality for a better animation experience. TypeScript adds type syntax to JavaScript, and it compiles into JavaScript. Our application is hosted on the servers of the Department of Computer Science at Appalachian State University [14], and the source code can be found on GitHub [13].

4.1.1 Cytoscape

Many data structures and algorithms can be visualized using graphs, which is why we choose to build on top of the Cytoscape graph visualization library. To visualize a graph, Cytoscape requires some container HTML element. It will then add the necessary components to visualize a graph in that element. Once a Cytoscape instance has been created, nodes and edges can then be added, manipulated, or removed.

A Cytoscape instance contains a list of elements, and a list of styles that can be used to modify how elements appear. Elements in Cytoscape are organized into a group of edges or a group of nodes. They also may contain a `data` attribute, which we use to hold element IDs, any label associated with the element, and any other data we may need. Elements may also

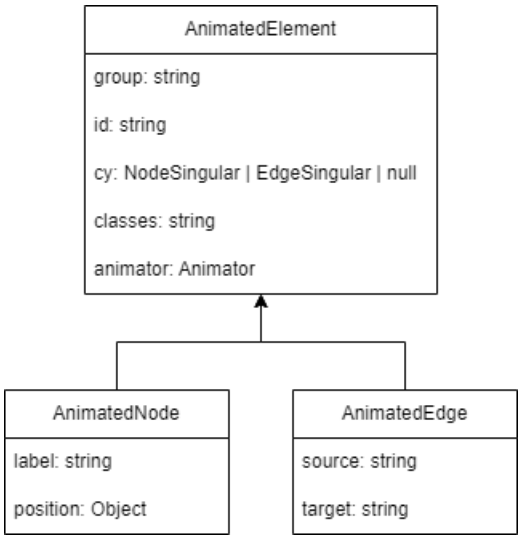


Figure 4.1: A Class Diagram of AnimatedElement, AnimatedNode, and AnimatedEdge

contain classes, which are used to denote whether they should be styled in a certain way. Node elements have positions, denoted by x and y coordinates, and edge elements have source and target nodes.

By default, Cytoscape does not perform any animations. Elements will simply appear, change, or disappear the instant the data model changes. It does, however, provide some animation features that we leverage in our visual tool, though we have found it easier in some cases to provide our own implementations of animation features. In order to make some of these implementations work, we choose to maintain a data model separate from that of a Cytoscape instance. We go into more detail about this in Section 4.1.3.

4.1.2 Animated Elements

Our data model contains several classes, the three primary ones being `AnimatedElement`, `AnimatedNode`, and `AnimatedEdge`. Both `AnimatedNode` and `AnimatedEdge` extend `AnimatedElement`. A basic class diagram containing key attributes of each class can be seen in Figure 4.1. Each element contains the group it belongs to, which helps separate nodes and edges in our data model. They also contain an ID that matches the corresponding Cytoscape element's ID. `AnimatedElements` have a reference to their corresponding element in the Cytoscape data model. This is primarily used to check if an element in our data model exists

in Cytoscape's data model. It also allows quick access to the Cytoscape element so that we may update it in the visualization.

`AnimatedNodes` contain a label which we use to represent some data about that node in the visualization, and its position in our data model. `AnimatedEdges` contain the IDs of the source and target nodes of the edge in our data model.

4.1.3 The Animator

The core component of our animations is the `Animator`. An `Animator` maintains a single Cytoscape instance by default, as well as its own data model of `AnimatedElements`. An animator simply initializes a Cytoscape instance with an empty list of elements and a predefined list of styles. `Animators` also globally maintain an animation duration and an animation delay, both in milliseconds. When an animation runs, its duration is not determined by the animation itself, but the global duration once the animation begins. The same is true for animation delay. These values may be changed at any time, allowing the visualization to be sped up or slowed down. Figure 4.2 may be referenced to help understand how the main components of `Animator` work together as we continue discussing its features.

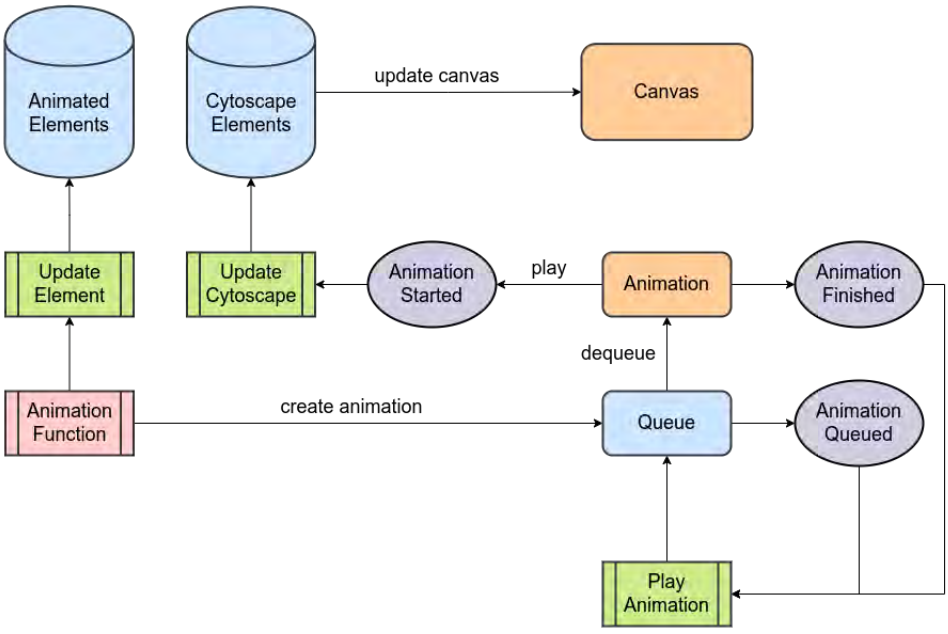


Figure 4.2: A Flow Diagram of the Visualization Tool

Animation Queue

One of the most notable aspects of `Animators` is the animation queue. Performing animations sequentially via Cytoscape directly is not user-friendly. We choose to implement a queue of animations for animators rather than using Cytoscape's animation queue. When an animation is queued, we first check if the queue is empty. If so, we simply begin playing it. Otherwise, we add it to the queue. Once an animation is finished, and the time for the animation delay has elapsed, the next animation (if there is one) is dequeued and played.

The implementation of the animation queue is why we implemented a separate data model from Cytoscape's data model. When an `AnimatedElement` is created via an animator, it queues an animation to add its corresponding element to Cytoscape so that it is only added once all prior animations have completed. Future animations may interact with that element, and they may do so before that element exists in Cytoscape. In order to prevent stale or non-existent data from being used for future animations, we keep the information up-to-date in `AnimatedElements` as animations are created and queued. We can then access up-to-date data in an `AnimatedElement` to create accurate animations. Once the animation queue is empty, the state of both the animator's data model and that of its Cytoscape instance will be equivalent.

Animator Events

The animation queue is managed via several event listeners, which are registered with an `AnimatorEventBus`. An `AnimatorEventBus` simply maintains a dictionary of events and event listeners, and provides functions to add or remove event listeners, and to emit events. `Animator` emits a few queue-related events. It listens for these events to control the flow of animations. These are listed in Table 4.1.

`Animator` also emits and handles a few playback-related events. The `pause()`, `resume()`, and `step()` functions are provided to affect animation playback. Allowing a user to manipulate playback gives them time to process animations and better understand what is being shown. When an `Animator` emits an event, it first checks if it is paused. If so, the `Animator` catches and stores the event emitted, rather than triggering it in its event bus.

Event	When Event is Triggered	How `Animator` Handles Event
animationQueued	When an animation is added to the queue.	If the newly queued animation can be played, play it.
animationStarted	When an animation begins playing.	Set a flag indicating the `Animator` is busy.
animationFinished	When an animation completes.	After the delay has elapsed, set a flag indicating the `Animator` is idling, and attempt to play another animation.
animationPaused	When the `pause()` function is called.	Set a flag indicating the `Animator` is paused.
animationResumed	When the `resume()` function is called.	Set a flag indicating the `Animator` is not paused.
animationStep	When the `step()` function is called and the `Animator` is paused.	`Animator` does not listen to this event.

Table 4.1: Animator Events, when they are thrown, and how `Animator` handles them

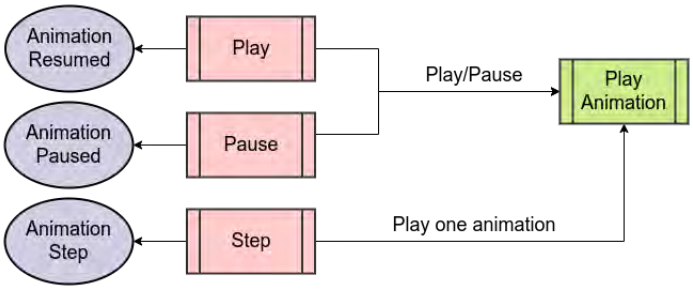


Figure 4.3: A Flow Diagram of `Animator`'s Playback Control

Since the flow of animations is maintained via events, this is necessary to pause the visualization. When the `Animator` resumes, it then releases the stored event, triggering it in the event bus, and resuming the animation. If the `Animator` is paused, then the `step()` function can be called to trigger the stored event, while keeping the `Animator` paused. This lets users play animations one step at a time. These events are also listed in Table 4.1, and Figure 4.3 shows a simple flow diagram of playback control.

Element Manipulation

`Animator` has many functions that allow for the addition, removal, and manipulation of elements. Some of these are straightforward, such as the addition and removal of elements. When adding an element, create an `AnimatedNode` or `AnimatedEdge` with the given data, then create and queue an `Animation` to add that element to the Cytoscape instance. The added element is returned so that it may be referenced by the developer when needed. Having access to element IDs is useful for animating already-existing elements. When removing an element, create and queue an `Animation` to remove it from both the `Animator` and the Cytoscape instance. The element is removed from both models during the animation to allow any prior animations to continue to reference the `AnimatedElement`. This is important when animations need access to the Cytoscape element reference stored in an `AnimatedElement`.

Elements can be manipulated by movement or style changes. `Animator` may move a single node or multiple nodes at a time, and it may do so by either moving them to given positions or by a given amount. The `Animator` can move multiple nodes to different positions by creating multiple Cytoscape animations, only emitting an `animationFinished` event once every Cytoscape animation has completed. `Animator` can also swap node positions, which we detail in Section 4.2. Style changes include color changes, as well as updating the label of a node or annotating a node. These update the `Animator`'s data model immediately, and queue an animation to update the visualization. Edges behave in largely the same way, though we do not label or annotate edges. When moving an edge, we simply change its source node and/or its target node.

Snapshots

An `Animator` may take snapshots of the visualization at any time. This can be used to show the visualization at various stages, which can be useful in understanding data structures and algorithms. A snapshot may be taken using the `takeSnapshot` function. An animation is then queued. When the animation is played, a copy of the Cytoscape data model is created, and stored in a list of snapshots maintained by the `Animator`. Queuing this function allows snapshots to be taken at various points during visualization. Snapshots are a special animation

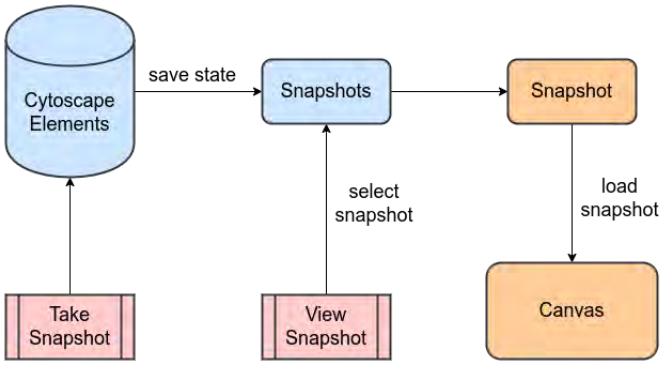


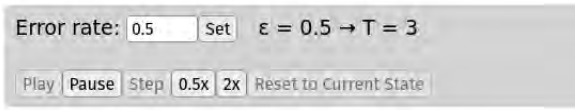
Figure 4.4: A Flow Diagram of `Animator`'s Snapshot Feature

in that there is no delay between a snapshot being taken and the next animation beginning. Calling `restoreSnapshot` will load a snapshot. It takes either an index in the snapshot list or a Snapshot object as an argument. It then loads the `Animator`'s history container with the Cytoscape elements. The history container is a container element in which a Cytoscape instance has been initialized. It is optionally provided to the `Animator` constructor. The history container may or may not be the same container element in which the main visualization occurs, but for stability, it is best to use a separate container element. A snapshot may also be taken instantly, rather than queued as an animation. The `getCurrentSnapshot` function takes a snapshot of the visualization at the moment it is called, and returns it. Figure 4.4 shows the flow of taking and viewing snapshots.

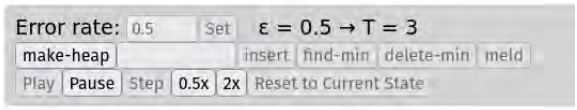
4.2 Visualizing the Soft Heap

4.2.1 The Webpage

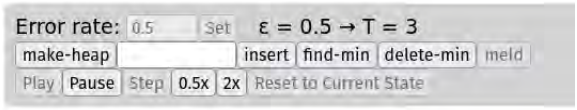
The soft heap visualization contains several key elements, which include the control panel, the sidebar, and the visualization. The control panel is where we provide buttons to input an error rate, perform soft heap operations, and manipulate playback. Initially, the buttons for operations are hidden. Once a valid error rate is provided, they appear. We provide buttons for the `insert`, `delete-min`, and `find-min` operations. We do to not provide `meld` visualizations due to constraints of the visual tool, but `meld` can still be understood to a degree by performing insertions. The visualization begins in a "play" state, although the user



(a) User May Enter an Error Rate



(b) Once Entered, User May Call `make-heap`



(c) The Remaining Operations Are Made Available

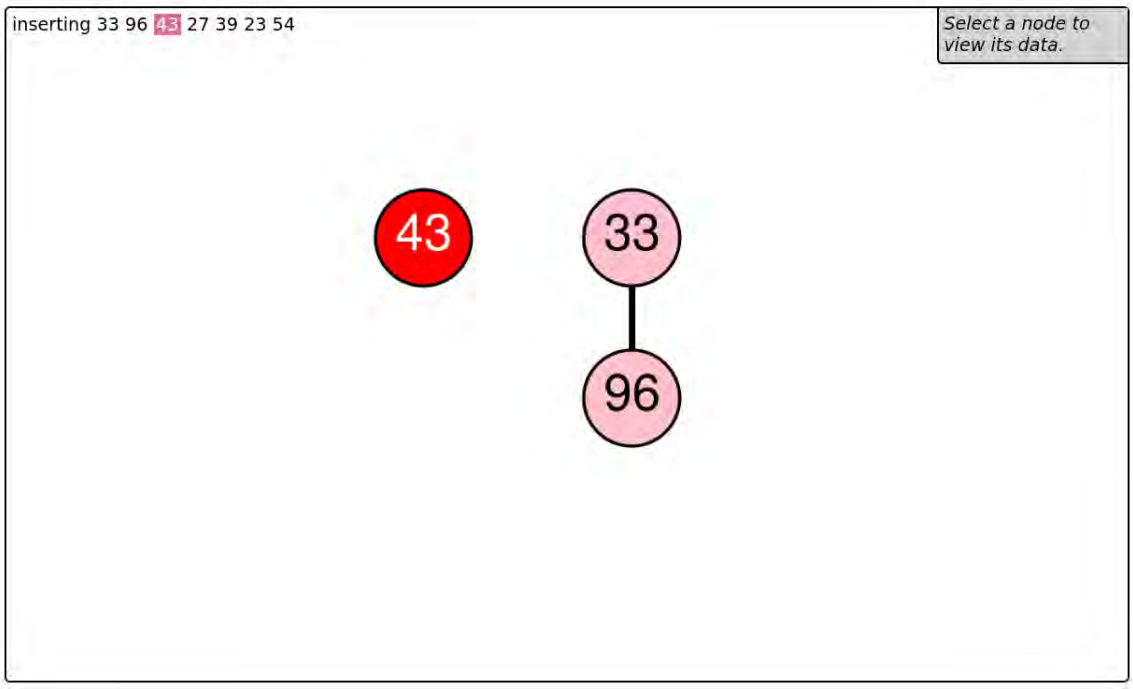
Figure 4.5: Control Panel Overview

may pause the animation at any time. When the visualization is paused, the user may step through the visualization one animation at a time. They may also slow the visualization by halving its speed, or speed it up by doubling its speed. Once given an error rate, an instance of `SoftHeap` is created and given an `Animator`.

4.2.2 The Soft Heap

To visualize the soft heap, we first need to be able to reference visual elements from within the soft heap. The `Vertex` class represents a node in the soft heap (`Node` is already a class in JavaScript). A `Vertex` contains all the same data that is mentioned in Chapter 2, however, we augment that with two things: a `corrupted` flag, denoting if the node contains corrupted elements; and an `elements` object. The `elements` object contains references to `AnimatedNode`, and `AnimatedEdges` for the left and right nodes, and for the next tree in the root list. Also contained in `elements` is an `AnimatedTree`, which provides functions to show nodes in a tree-like fashion, and to get the visual boundaries of a tree.

As items are inserted or deleted, we show the status of the operation within the visualization. (See Figure 4.6.) For example, in `insert`, when a specific item is being inserted, it is highlighted within the visualization status to make it easier to determine where the visualization is amongst all insertions.



(a) Item with Key 43 Being Inserted

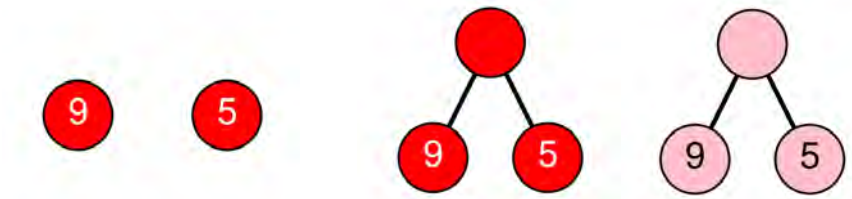
Figure 4.6: Visualization Status

When an item is inserted into the soft heap, it is first made into a root node. When creating a root node, we first shift all existing nodes in the visualization by some amount, roughly the width of a node. This creates room for the new node to appear. We add the node via the `Animator` with its key as the label, and at a position relative to the first tree in the root list. We also provide the node with relevant data that will be used in the visualization, like its key, rank, and set. We then highlight the node, which temporarily changes its style to make it more noticeable to the user. The highlight is an animation that gets queued in the `Animator`. Importantly, we add a class to this new node. The class is the node's ID. We discuss this more when discussing how we visualize `fill`. Lastly, we make an `AnimatedTree` out of the node. The animation process is shown in Figure 4.7.

When linking two trees, we place a new node above both trees, adding edges from the new node to the root of the trees, making them children of the new node. We associate the nodes in both trees with the new node by adding a class of the new node's ID. We then add both trees' `AnimatedTrees` as children of the new node's `AnimatedTree`. We apply a tree

(a) A Single-Node Heap (b) The Node Shifts Right (c) New Node is Added

Figure 4.7: The Process of Adding An Item to the Soft Heap Visualization

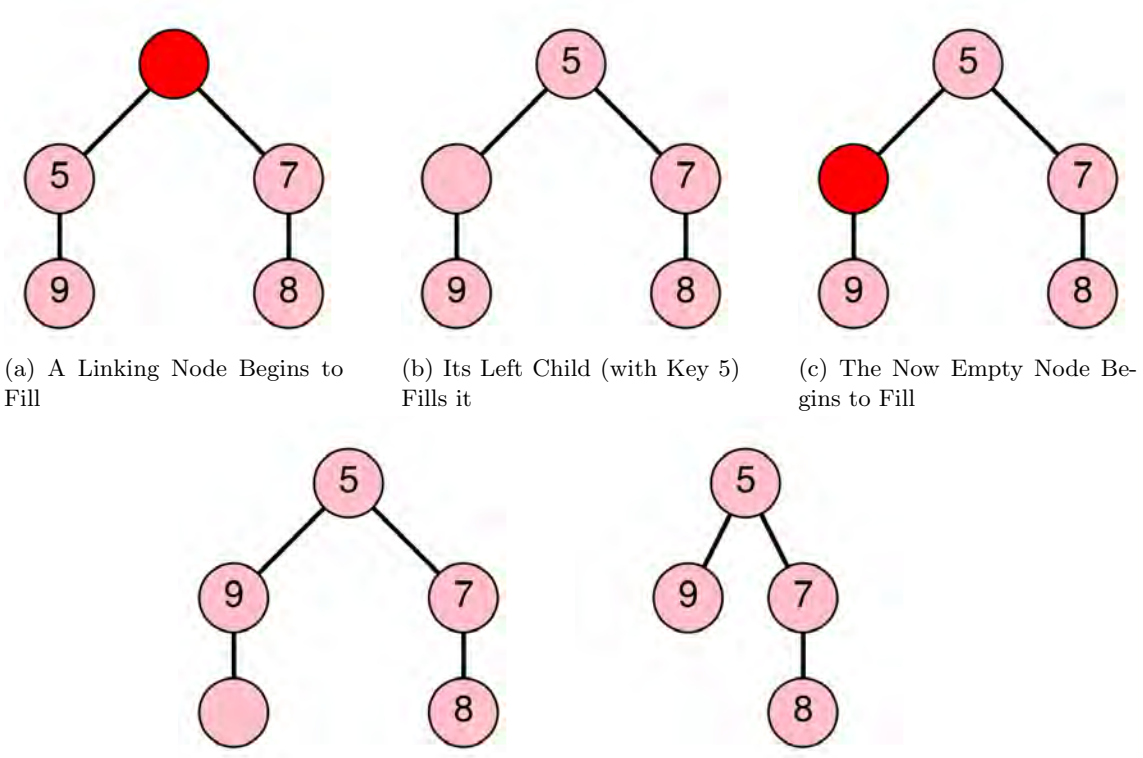


(a) Two Trees with Same Rank (b) A New Node Links Both Trees

Figure 4.8: The Process of Linking Trees in the Soft Heap Visualization

layout to the newly formed tree before calling `defill`. This linking process is shown in Figure 4.8. The linking node then gets filled. In `fill`, if the children need to be swapped, we call the `swapNodes()` function. Given two nodes x and y, `swapNodes()` first gets all nodes in both subtrees (all nodes with the class $x.id$ or $y.id$), calculates the difference in positions of x and y, and moves the nodes simultaneously (x nodes in one direction, y nodes in the opposite direction). This is the primary benefit of classing nodes with what tree they belong to; it is easier to find and manipulate all nodes that may be associated with a particular node. Once the nodes are swapped, we begin filling the linking node. We update the node's label to that of its left child. If the left child was a leaf, we remove it from the visualization. Otherwise, recursively fill the left child. This filling process is illustrated in Figure 4.9.

Once a tree is linked, the root list must be reordered. When inserting, we simply call `rank-swap` and `key-swap`. The visualizations for both operations are the same. We first remove any edges from the roots of the trees being swapped, then swap the trees. Once the trees are swapped, we add new edges to reflect the recently changed *next* pointers for the swapped roots. When swapping trees, it may be the case that they either overlap or have a



(a) A Linking Node Begins to Fill

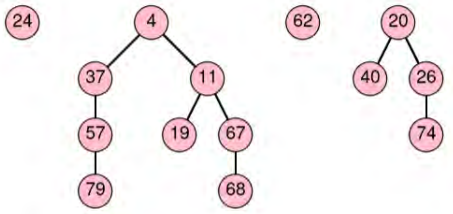
(b) Its Left Child (with Key 5) Fills it

(c) The Now Empty Node Begins to Fill

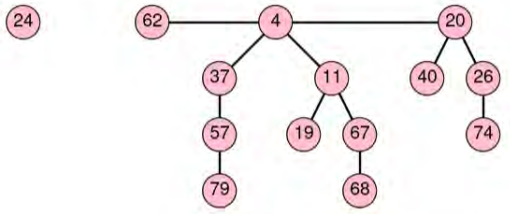
(d) The Node is Filled by its Child

(e) The Leaf Node is Removed, and the Layout Updated

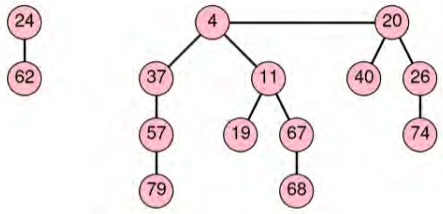
Figure 4.9: The Process of Filling Nodes in the Soft Heap Visualization



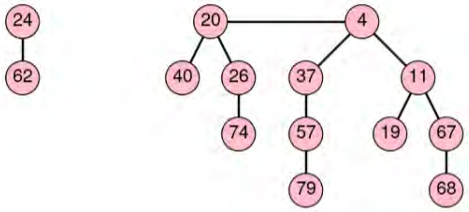
(a) Item with Key 24 is Inserted, Remove Edges from Tree with Conflicting Rank



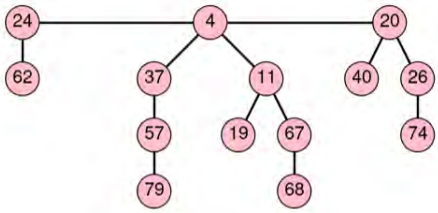
(b) First Two Trees Swap, and New Edges are Added



(c) Trees Link to Form a New Tree



(d) Rank Swap to Check For Rank Conflicts



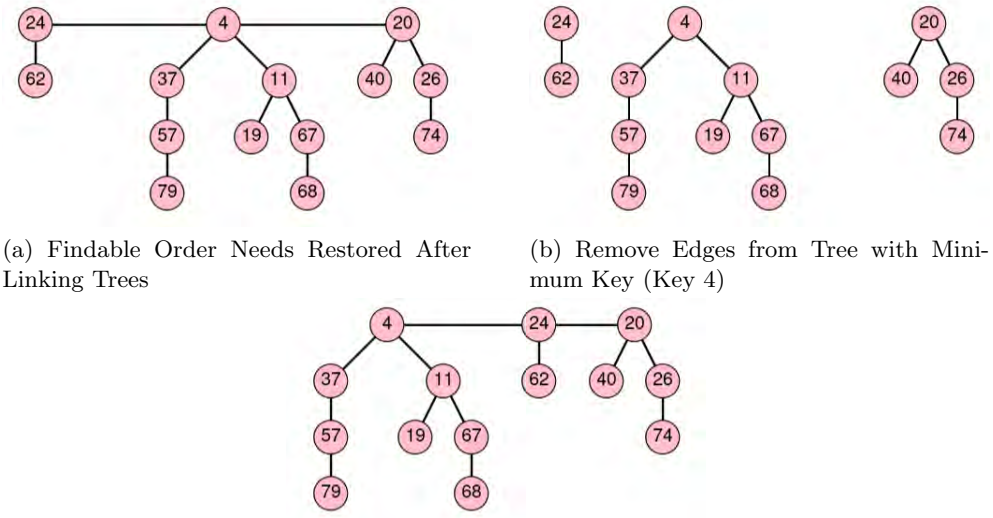
(e) Key Swap Since There Are No Rank Conflicts

Figure 4.10: Converting to Meldable Order to Insert an Item in the Soft Heap Visualization

large distance between them. To fix this, once the trees have been swapped, we use a function specific to animated soft heaps called `tightenTrees`, which simply calculates the bounds of the trees, and shifts them so that there is some short distance between the trees. The swapping process is shown in Figures 4.10 and 4.11. The process of reordering the entire root list is much the same, as it relies heavily on `rank-swap` and `key-swap`. The `reorder` function ensures tree edges are tightened, and once the recursive operation of `reorder` is complete, it updates edges in the visualization by removing existing edges, and then adding new ones when necessary.

Showing Corruption

To more easily and quickly determine if a node contains corrupted elements, each node is augmented with a `corrupted` flag. We set this flag when a double-even filling occurs in



(a) Findable Order Needs Restored After Linking Trees

(b) Remove Edges from Tree with Minimum Key (Key 4)

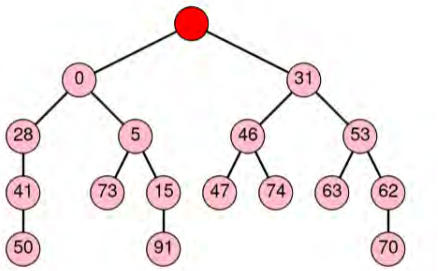
(c) Swap the Minimum Tree with the Newly Linked Tree, Adding Edges

Figure 4.11: Converting to Findable Order After Inserting an Item in the Soft Heap Visualization.

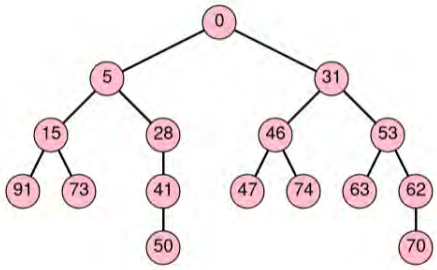
`defill`. If a node contains corrupted elements, we choose to show that by changing the color of the node to differentiate it. We also add an annotation to the node that shows its items. Since Cytoscape allows just a single label for each element, we add what Cytoscape calls a compound node. Such a node is meant to contain other nodes in the graph. We add the corrupted soft heap node to a compound node. This compound node has its own label containing a corrupted node's items. Anytime the set of a corrupted node changes, so does the annotation. This animation is shown in Figure 4.12. It may be the case that when a corrupted node empties and fills with the set of its left child, the left child contains no corrupted elements. We simply revert the node to its non-corrupted state by changing the node color back to the original color, and removing the set annotation.

Soft Heap Snapshots

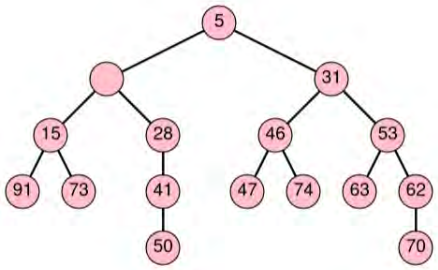
At the end of each heap update operation (`insert` and `delete-min`), we take a snapshot of the heap. Snapshots are shown under the "History" tab in the sidebar. We show the operation performed, and the key of the target item (the item inserted or deleted). This can be customized by optionally passing a string to the `takeSnapshot()` function. The user may double-click



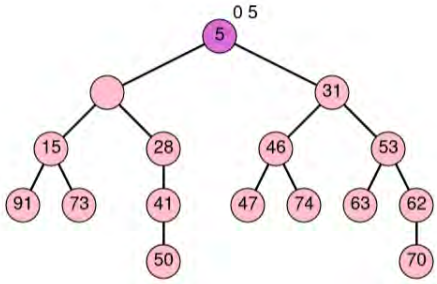
(a) Filling an Even Rank Node $> t$



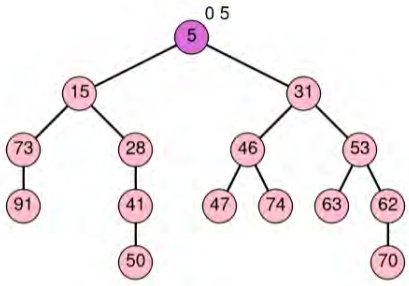
(b) The First Filling is Complete



(c) Node with Key 5 Fills the Root



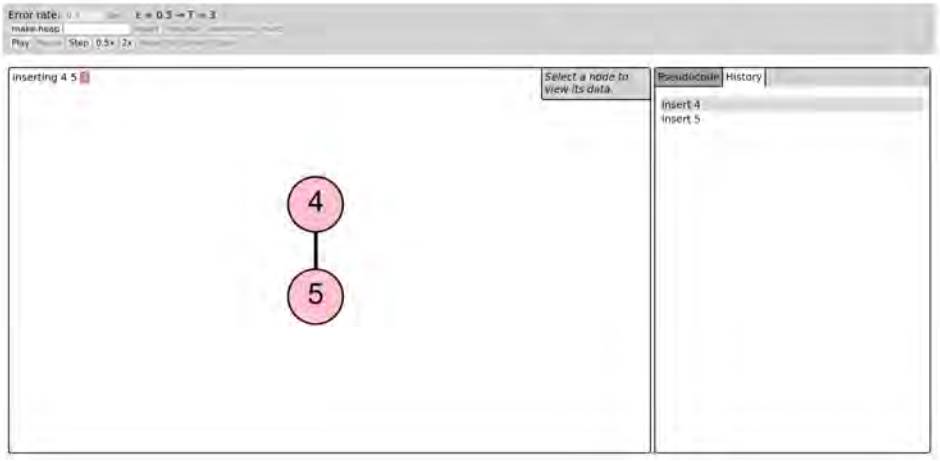
(d) The Root Changes Color and Annotates its Set



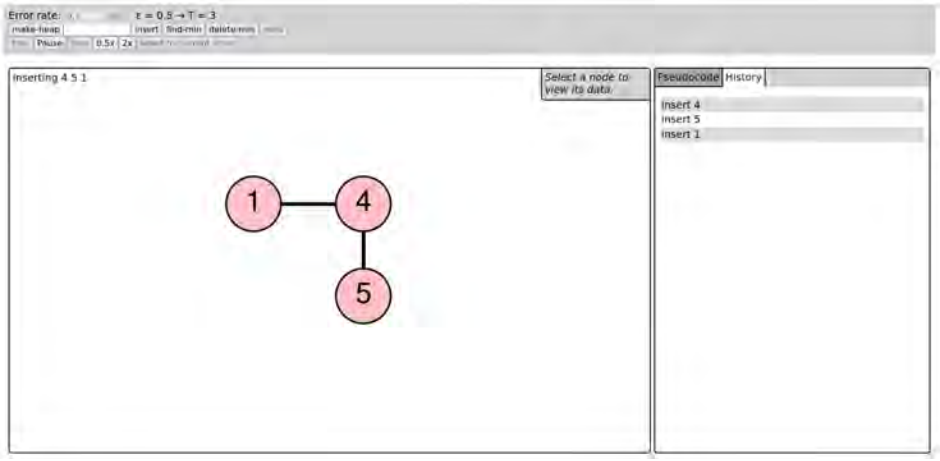
(e) The Second Filling Completes

Figure 4.12: Showing Corruption in the Soft Heap Visualization

a snapshot entry to view the state of the soft heap when the snapshot was taken. To show the snapshot, we hide the primary, interactive visualization container and show in its place the snapshot container. We create a second Cytoscape instance in this container and simply load the state of the snapshot into it. Doing this prevents any changes to the current visualization. We choose to disable any operations that could run animations until the user chooses to view the original visualization again. At that point we hide the snapshot container, show the original one, and re-enable operations. An example of viewing a snapshot is shown in Figure 4.13.

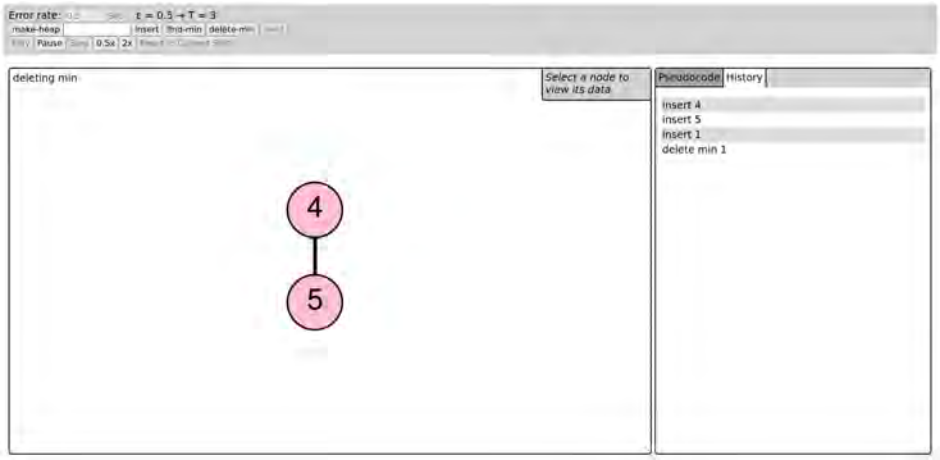


(a) Snapshots for Inserting 4 and 5, Before Inserting 1

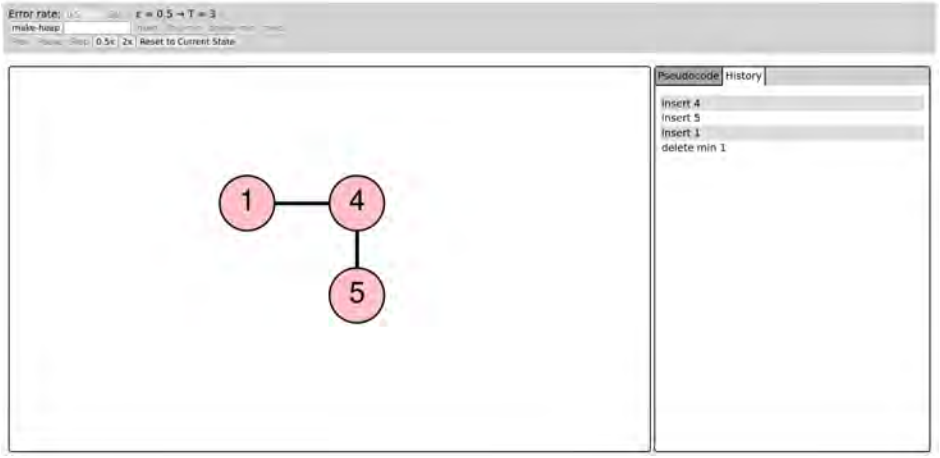


(b) Snapshots for Each Insertion, After Inserting 1

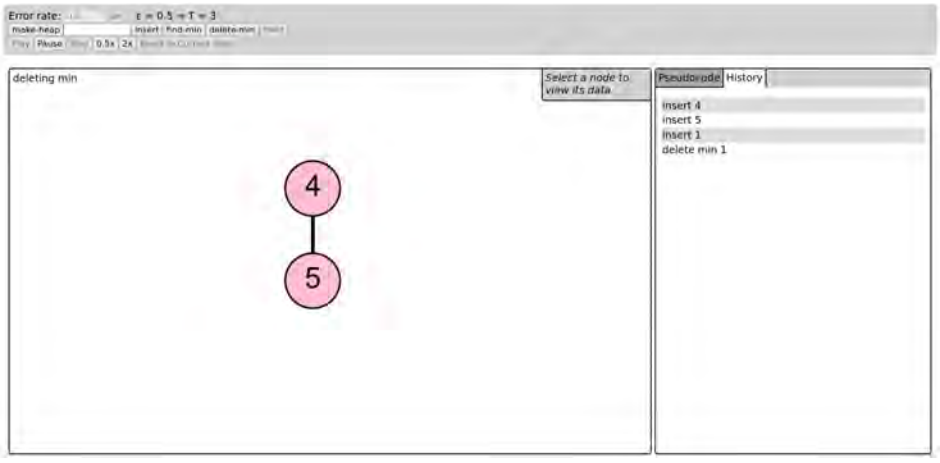
Figure 4.13: Viewing Snapshots in the Soft Heap Visualization



(c) A Snapshot is Added When 1 is Deleted



(d) Double-clicking "insert 5" Shows the State After Inserting 5



(e) Clicking "Reset to Current State" Hides the Snapshot

Figure 4.13: Viewing Snapshots in the Soft Heap Visualization *Cont.*

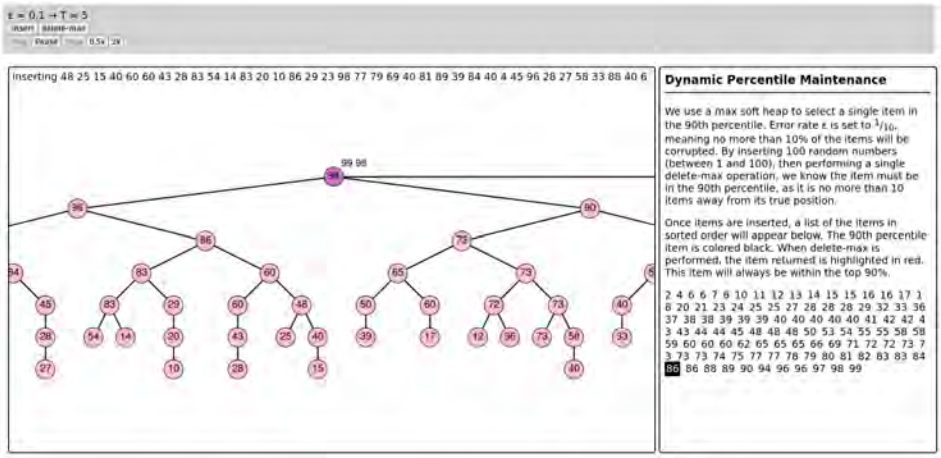
4.3 Visualizing Soft Heap Applications

We provide visualizations of three soft heap applications: dynamic percentile maintenance, approximate median finding, and approximate sorting. To visualize dynamic percentile maintenance, we first create a max soft heap. It is largely the same as described in the previous section, except we change `key-swap` to place the tree with the maximum key at the beginning of the root list, and we change `fill` to swap child nodes if the key of the right child is greater than the key of the left child.

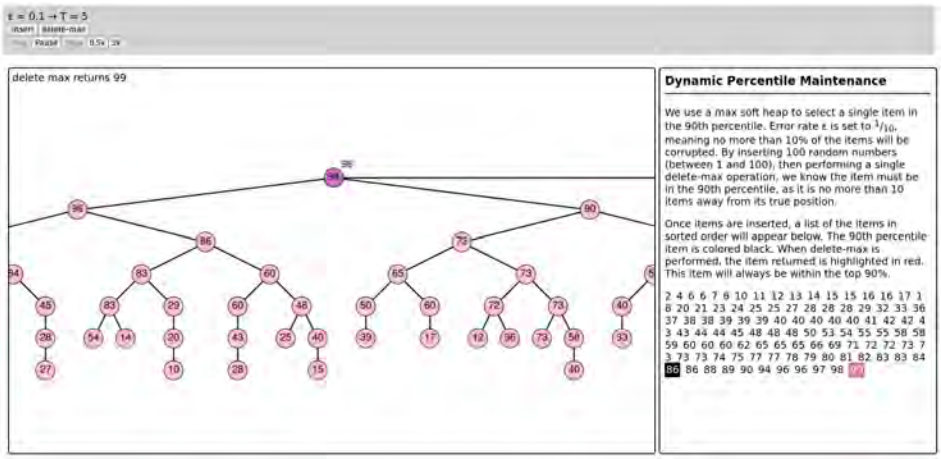
The webpage is very similar; it contains a control panel, a sidebar, and the visualization. In the control panel, we only show buttons for `insert` and `delete-max`, as well as playback buttons. We remove the ability to view snapshots and the ability to set an error rate. The error rate is set to $\frac{1}{10}$ when the page initializes. The "insert" button generates one-hundred random numbers between one and one-hundred, and inserts them in random order into the soft heap. In the sidebar, we show the items sorted, and highlight the 90th-percentile item. Once all the items are inserted, performing a single `delete-max` will return an item in the 90th percentile, highlighted in red. The visualization can be seen in Figure 4.14.

Visualizing approximate median finding is done in much the same way as percentile maintenance. The error rate is also set to $\frac{1}{10}$ upon initialization, and we only provide an "insert" and a "delete" button. Another list of one-hundred randomly generated items is inserted into the heap, and the sorted list appears in the sidebar with the inner 20% of items highlighted in gray. When "delete" is pressed, 45 items are removed. The largest removed item is the approximate median, and is highlighted in the sidebar. This visualization is shown in Figure 4.15.

To visualize approximate sorting (shown in Figure 4.16), the error rate is set to $\frac{1}{2}$. We can adequately show that items exit the heap nearly sorted with fewer items, so we increase the error rate, which in turn decreases the threshold at which corruption is allowed to occur. Once all items are inserted, all items are then deleted. The order in which items exit is shown above the visualization, and if an item is corrupted, we highlight it.

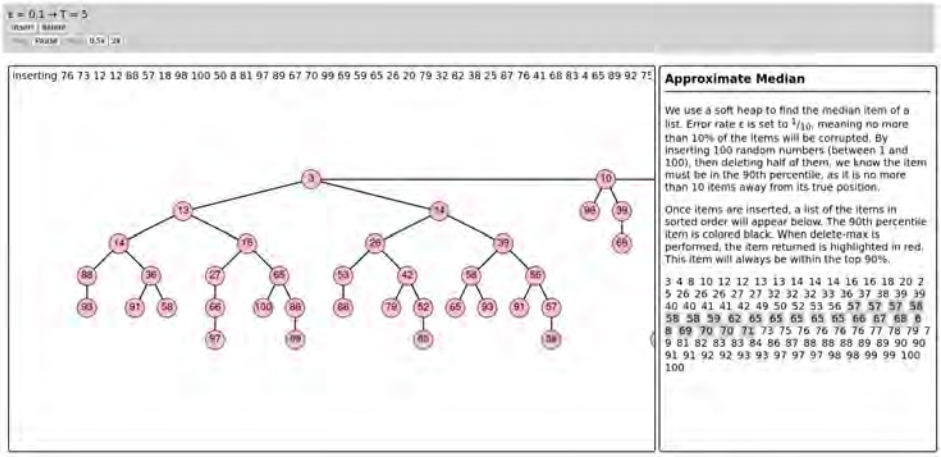


(f) All Items Inserted, 90th Percentile Highlighted in Black

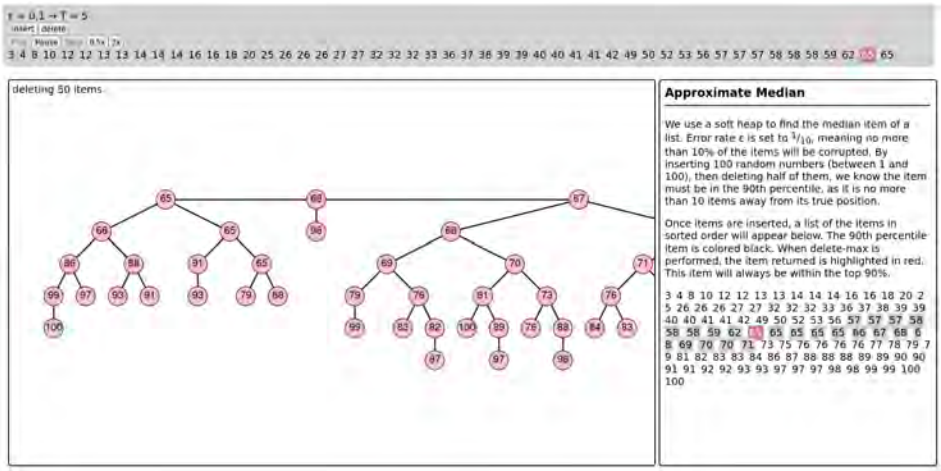


(g) Deleting the Maximum Key Returns an Item (with Key 99) in the 90th Percentile

Figure 4.14: Visualizing Dynamic Percentile Maintenance

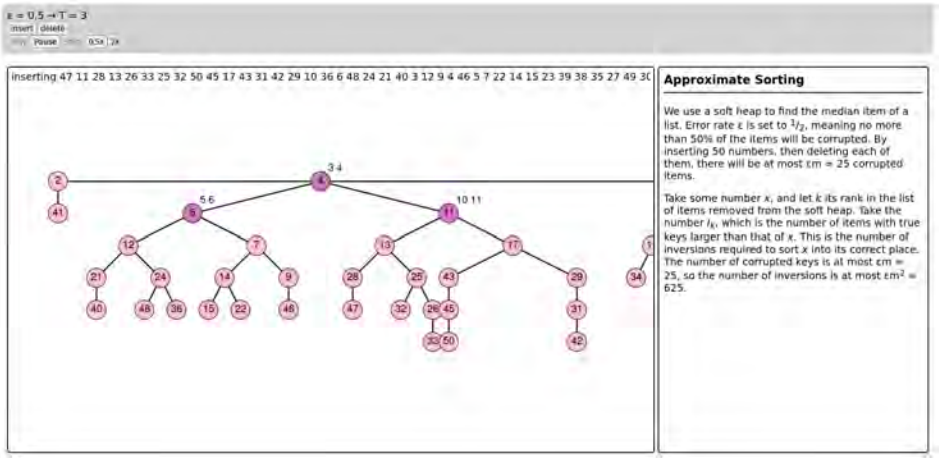


(a) All Items Inserted, Items within 10% of True Median Highlighted in Gray

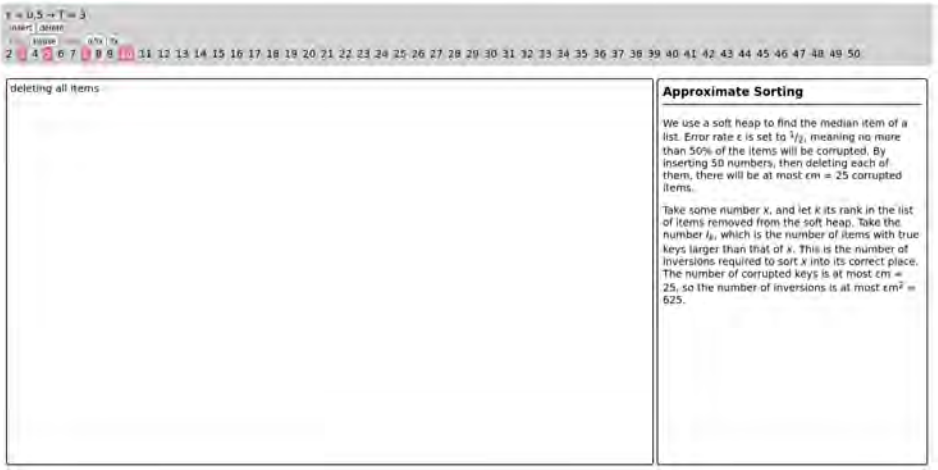


(b) Deleting Fourty-Five Items, the Largest Item Deleted (65) Highlighted

Figure 4.15: Visualizing Approximate Median Finding



(a) All Items Inserted



(b) After Deleting All Items, Each Corrupted Item is Highlighted

Figure 4.16: Visualizing Approximate Sorting

Chapter 5

Conclusion

The soft heap is a simple, yet powerful data structure that can be used in many applications. Specifically, they can be used in the fastest deterministic minimum spanning tree algorithm [2], and can be used in dynamic maintenance of percentiles, finding approximate medians, and approximate sorting [3]. Chazelle's original implementation involved a collection of binomial trees [3], and these were simplified to use a collection of binary trees [10] [9]. In this thesis, we provided a description of the implementation of Kaplan, Zwick, and Tarjan, as well as their analysis for completeness. We hope that our presentation is more intuitive and accessible. We also provide a visualization tool that can visualize the inner workings of the soft heap.

We propose a slight modification to the soft heap, to do only single fillings during `delete-min` operations. This ensures that no new items are corrupted during deletion, and allows us to use the soft heap as a black box, guaranteeing that no more than εn items may become corrupted. The modified soft heap is available in several different programming languages [12], so that it may be used by others as a black box. The visual tool also visualizes three applications of soft heaps. Our tool is publicly available [13], and may be extended to visualize other data structures and algorithms.

Our tool is in its first version, and can be improved in its performance, as well as adding in new features. First and foremost, removing the tool's reliance on the Cytoscape library, which would allow us to create and manage visual elements directly. This would eliminate the need for separate (yet nearly structurally equivalent) data models to show a single visualization. The tool could continue to use its animation queue to perform animations sequentially, though it

would be interesting to see how JavaScript Promises (or the like) could be used to achieve a similar effect. This would also give a developer more flexibility in allowing multiple animations to play asynchronously, if desired. We also like to explore having multiple data structures in the same visualization, and having the visualization. This would be very useful in visualizing the soft heap's `meld` operation.

If the visual tool were able to directly manage its visual elements, it would also be more effective at managing animation history, and loading elements directly into the visualization without animating them. This would allow us to "load", and "save" animations, in case we are working to showcase a more complex algorithm using them. It would also allow us to roll back the animation to a particular point, and perform different operations from that point forward. Due to the technical intricacies of using separate data models, we did not fully implement either feature. Though, the visual tool would feel more complete and effective with these features. Being able to load elements instantly would make some visualizations better, specifically those like the visualizations of soft heap applications. It would be natural to use the existing soft heap visualization as part of a larger visualization of Chazelle's minimum spanning tree algorithm [2]. It would also be interesting to see how we can easily extend this tool to visualize and animate other advanced data structures.

Bibliography

[1] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[2] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, nov 2000.

[3] Bernard Chazelle. The soft heap: An approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, nov 2000.

[4] Seonghun Cho and Sartaj Sahni. Weight-biased leftist trees and modified skip lists. *ACM J. Exp. Algorithmics*, 3:2–es, sep 1998.

[5] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. *Heapsort*. The MIT Press, 4th edition, 2022.

[6] Robert W. Floyd and Ronald L. Rivest. Algorithm 489: The algorithm select—for finding the ith smallest of n elements [m1]. *Commun. ACM*, 18(3):173, mar 1975.

[7] Max Franz, Christian T. Lopes, Dylan Fong, Mike Kucera, Manfred Cheung, Metin Can Siper, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. Cytoscape.js 2023 update: a graph theory library for visualization and analysis. *Bioinformatics*, 39(1):btad031, 01 2023.

[8] R.L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[9] Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Soft heaps simplified. *SIAM Journal on Computing*, 42(4):1660–1673, 2013.

[10] Haim Kaplan and Uri Zwick. A simpler implementation and analysis of chazelle's soft heaps. In *ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[11] Donald E. Knuth. Sorting and searching. *The Art of Computer Programming*, 3, 1998.

[12] Shane McCann. Soft heap implementations. `https://github.com/mccannsa/soft-heap-implementations`, 2023.

[13] Shane McCann. Soft heap visualizer. `https://github.com/mccannsa/soft-heap-visualizer`, 2023.

[14] Shane McCann. Visualizer. `https://cs.appstate.edu/mccannsa/vis/`, 2023.

[15] Microsoft. Typescript: Javascript with syntax for types. https://www.typescriptlang.org/.

[16] Robert Sedgewick and Kevin Wayne. *Sorting*. Pearson Education, 4th edition, 2011.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.

[18] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[19] Robert E. Tarjan and Caleb C. Levy. Zip trees. *CoRR*, abs/1806.06726, 2018.

[20] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[21] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, apr 1978.

Appendix

Proof of Lemma 3 and Theorem 10 use the following sum of an infintie geometric series.

Theorem A1. $\sum_{i\geq 1} \frac{1}{2^i} = 1.$

Proof. We can show that this geometric series sums to 1 by using the "shifting sums" technique. Let $x = \sum_{i\geq 1} \frac{1}{2^i} = 1.$

$$\begin{aligned} x &= \sum_{i\geq 1} \frac{1}{2^i} = 1 \\ &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \end{aligned} \tag{5.1}$$

Dividing both sides by 2, we get

$$\frac{x}{2} = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \tag{5.2}$$

Subtracting Equation 5.2 from 5.1 cancels all terms in the right hand side except $\frac{1}{2}$. So we get

$$\begin{aligned} x - \frac{x}{2} &= \frac{1}{2} \\ \implies \frac{x}{2} &= \frac{1}{2} \\ \implies x &= 1. \end{aligned}$$

□

Proof of Lemma 5 and Theorem 12 use the following sum of an infinite arithmetico-geometric series.

Theorem A2. $\sum_{i\geq 1} \frac{i}{2^i} = 2.$

Proof. We can prove this by applying the shifting sums technique twice. Let $x = \sum_{i\geq 1} \frac{i}{2^i}$.

$$
\begin{aligned}
x &= \sum_{i \geq 1} \frac{i}{2^i} \\
&= \frac{1}{2^1} + \frac{2}{2^2} + \cdots \\
\frac{x}{2} &= \frac{1}{2^2} + \frac{2}{2^3} + \cdots \\
x - \frac{x}{2} &= \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots \\
&= \sum_{i \geq 1} \frac{1}{2^i} \\
&= 1 \qquad \text{(from Theorem A1)} \\
\Longrightarrow x &= 2.
\end{aligned}
$$

□

Vita

Shane McCann was born in Ventura, California to Sandra Adrianne McCann and David McCann. They graduated from Cape Fear High School in 2015. Afterward, they attended Fayetteville Technical Community College from 2015 to 2019, earning an A.A.S. in Computer Programming in 2017, and an A.S. in 2019. Shane then transferred to Appalachian State University, earning a B.S. in Computer Science in 2021. Later that year, they began pursuing an M.S. in Computer Science at Appalachian State University, eventually concentrating in Theoretics. As a graduate student, they worked primarily as a Graduate Teaching Assistant, and as a part-time Junior Software Developer at Ferrous Design.

As of 2023, Shane lives in Boone, North Carolina, working as a Lecturer in the Computer Science Department at Appalachian State University.