Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Vojtech Bardiovský

Implementation of operations in double-ended heaps

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Václav Koubek, DrSc.
Study programme:	Computer Science
Specialization:	Discrete Models and Algorithms

Prague 2012

I would like to give special thanks to prof. Václav Koubek for advices of both technical and non-technical nature, for inspiration and for the enormous quantity of a very detailed and helpful feedback.

This work is dedicated to my parents, who made every single part of my education possible.

Název práce: Implementácia operácií vo dvojkoncových haldách

Autor: Vojtech Bardiovský

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Václav Koubek, DrSc., Katedra teoretické informatiky a matematické logiky

Abstrakt: Existuje viacero spôsobov ako vytvoriť dvojkoncovú haldu z dvoch klasických háld. V tejto práci rozšírime dvojkoncovú haldu založenú na prepojení listov a vytvoríme novú schému nazvanú L-korešpondencia. Táto schéma rozšíri triedu možných klasických háld použiteľných pre vytvorenie dvojkoncovej haldy (napr. Fibonacci halda, Rank-pairing halda). Ďalej umožní operácie "Zníž prioritu" a "Zvýš prioritu". Tento prístup ukážeme na troch konkrétnych haldách a odhadneme časovú zložitosť pre všetky operácie. Ďalším výsledkom je, že pre tieto tri konkrétne haldy, očakávaný čas operácií "Zníž prioritu" a "Zvýš prioritu" je obmedzený konštantou.

Klíčová slova: dvojkoncová halda, decrease, zložitosť

Title: Implementation of operations in double-ended heaps

Author: Vojtech Bardiovský

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Václav Koubek, DrSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: There are several approaches for creating double-ended heaps from the single-ended heaps. We build on one of them, the leaf correspondence heap, to create a generic double ended heap scheme called L-correspondence heap. This will broaden the class of eligible base single-ended heaps (e.g. by Fibonacci heap, Rank-pairing heap) and make the operations Decrease and Increase possible. We show this approach on specific examples for three different single-ended base heaps and give time complexity bounds for all operations. Another result is that for these three examples, the expected amortized time for Decrease and Increase operations in the L-correspondence heap is bounded by a constant.

Keywords: double-ended priority queue, decrease, leaf correspondence, complexity, rank-paring heap

Contents

Preface

This work builds on the leaf correspondence method for the generic creation of double ended priority queues from two single-ended priority queues of Chong and Sahni[4]. The original method was an improvement of the total correspondence heaps employing the fact that if not all nodes have to be interconnected, a great deal of unnecessary operations can be eliminated.

We define a new type of a double ended heap, the L-correspondence heap. The L-correspondence heap is a generalization of the previous approach and admits a broader area of base heaps (for example the Fibonacci heap or the Rank-pairing heap).

The new approach relies on the fact, that in many known and practically eligible heaps, the operation of deleting a leaf (or a similar special type of a node that we will denote L-node) is simple and fast. We define a new method **Delete-L-node**, that is a restriction of the **Delete** operation on the L-nodes. This procedure is generally no different from the standard **Delete** operation, but the analysis of its time bound is more precise.

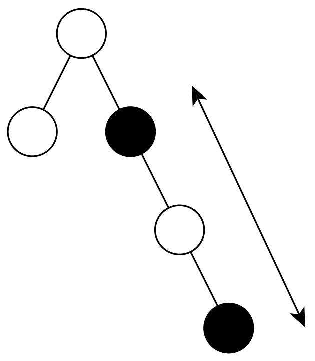

Figure 1: A chain of leaves in Fibonacci heap - it will be shown that its disassembly runs in amortized constant time. This is mainly because such chains are very rare and to achieve them, many cuts have to be made - these cuts then pay for the operation.

In the L-correspondence heap, the strict assumption for the priority of the parent (relative to the child's priority) is relaxed, which allows structures that do not satisfy the min-heap ordering (such as the Rank-pairing heap) to be eligible as base heaps.

The previous approach as it was originally defined did not allow heaps where the deletion of a node can result in a cascaded deletion of other nodes, to be used as the base heap (for example the Fibonacci heap). In our new approach, such heaps become eligible.

Very importantly, the **Delete-L-node** operation allows the operations **Decrease** and **Increase**. Since these operations use the base heaps's **Decrease** and **Increase**, a pure amortized time bound is not always possible. A simple probabilistic bound will be made instead to show that for some heaps, their expected amortized time bound is $O(1)$.

In the second chapter, the L-correspondence heap is defined and the **Delete-L-node** operation of the base heap is declared along with its use in **Delete-min** and **Decrease** and their symmetrical counterparts. In the third chapter, specific examples for the L-correspondence heap are shown for three different base heaps.

1. Introduction

1.1 Priority queue

1.1.1 Definition

Priority queue is an abstract data type containing elements with associated priority. Elements may be of a primitive or a structured type. The priority of an element x is denoted $\mathbf{Pr}(x)$. The data type supports the following operations:

Insert(x) - Inserts an element x, that is not represented by the priority queue, into the priority queue.

Find-min - Returns an element with the smallest priority.

Delete-min - Deletes and returns an element with the smallest priority.

Meld(Q_1, Q_2) - Returns a priority queue that represents the set $S = S_1 \cup S_2$, where S_1 and S_2 are sets represented by Q_1 and Q_2 respectively and $|S_1 \cap S_2| = \emptyset$.

In this work we will refer to a specific realization of the priority queue as **implementation**. The set of basic operations can be extended by following operations in most implementations:

Delete(x) - Deletes element x.

Decrease(x, a) - Sets a priority of element x, such that $\mathbf{Pr}(x) \geq a$, to a.

Increase(x, b) - Sets a priority of element x, such that $\mathbf{Pr}(x) \leq b$, to b.

Most of the time, to call these three operations, it is necessary to have some additional structural information. In case of a tree representation it may be a reference to a tree node representing the given element. In case of an array representation it can be an index of the array where the element is stored. This is necessary to avoid searching for the element in the structure and also not too limiting since many algorithms using priority queues can extend the object they are working with by a reference into the priority queue.

The argument x in these operations is a formal argument for the abstract data structure. In practice it will be information about the element's position in the data structure.

The defined priority queue is also called a **min** priority queue as it supports the **Delete-min** operation. Symmetrically equivalent data type is a **max** priority queue that supports the **Delete-max** operation. The implementations of both priority queue types are generally equivalent and do not require any additional work once the basic min priority queue is established.

1.1.2 Heap

The most typical way of representing a priority queue is by using a tree or a collection of trees. A conventional name for such structure is heap. Heap is a collection of trees where nodes represent elements from a priority queue and the following condition, also called min-heap ordering condition, holds:

(Min-heap ordering) For every non-root node v, if v represents element x from a priority queue and $\mathbf{Parent}(v)$ represents element y, then $\mathbf{Pr}(y) \leq \mathbf{Pr}(x)$.

To represent a max priority queue, the symmetrical max-heap ordering condition with reverse ordering will be used. For the rest of the work we will assume that min priority queue and max priority queue are represented by min-heap and max-heap, respectively, if not specified otherwise.

Priority queue can be implemented by a heap in two ways. Either the nodes can change the element they represent, thus they serve only as a container. Or they can be bound together with one single element. In this case the node is created when the element is inserted and is deleted whenever element is deleted. In this work we will consider only the second type of the implementation, therefore the node, and the element it is representing are **considered equivalent**.

This allows us to define a priority of the node v representing element x, as $\mathbf{Pr}(v) = \mathbf{Pr}(x)$.

1.2 Double-ended priority queue

1.2.1 Definition

Double-ended priority queue is an abstract data type that is both a min priority queue and a max priority queue, thus supports the following operations:

Insert(x) - Inserts an element x, that is not represented by the priority queue, into the priority queue.

Find-min - Returns an element with the smallest priority.

Find-max - Returns an element with the largest priority.

Delete-min - Deletes and returns an element with largest priority.

Delete-max - Deletes and returns an element with largest priority.

Delete(x) - Deletes element x.

Decrease$(x,\ a)$ - Sets a priority of element x, such that $\mathbf{Pr}(x) \geq a$, to a.

Increase$(x,\ b)$ - Sets a priority of element x, such that $\mathbf{Pr}(x) \leq b$, to b.

Meld$(Q_1,\ Q_2)$ - Returns a priority queue that represents the set $S = S_1 \cup S_2$, where S_1 and S_2 are sets represented by Q_1 and Q_2 respectively and $|S_1 \cap S_2| = \emptyset$.

1.2.2 Previous works

Many efficient double ended priority queue representations have been proposed. The *Symmetric min-max heap* of Arvind and Rangan[1] can be represented by an array. Its **Find-min** and **Find-max** operations have $O(1)$ running time and all other operations have $O(\log n)$ running time.

The *Interval heap* of [7] is equivalent with the *Interval heap* of [19], the *Twin heap*[18], the *Min-max pair heap* of [15], and the *Diamond deque* of [3]. The *Twin heaps* will be shown later. The **Find-min** and **Find-max** operations have $O(1)$ running time and all other operations have $O(\log n)$ running time.

All these approaches have one in common: they are in some way inspired by a basic 2-regular heap and they are static, meaning that the performance of their operations can not be changed to favor one specific operation or a set of operations.

The generic double ended heap approach, that we describe later, on the other hand is dynamic. For every approach (dual heap, total correspondence heap, leaf correspondence heap, our L-correspondence heap) there is a class of single-ended priority queues that can be used as the base heap to build the double ended priority queue.

1.2.3 Applications

One of the applications of the double ended priority queue is the modification of the sorting algorithm **Quicksort for external sorting**. External sorting is used when the elements to be sorted do not fit into the (fast) memory.

In the internal Quicksort, we split elements into two groups L and R and the pivot. All elements from L are smaller than the pivot, all elements from R are larger than the pivot. Then we recursively sort L and R and output the result: sorted L, pivot, sorted R.

With a double ended priority queue Q, we can sort elements externally. Assume that the priority of the element is equal to the key that we want to sort by. First read as much elements as fit into the memory and insert them into Q. The elements represented by Q will be our pivot set. Read the remaining elements one by one, let x be the current element:

- If $\mathbf{Pr}(x) \leq \mathbf{Pr}(Q.\mathbf{Find\text{-}min})$, output x into the file for the set L.
- If $\mathbf{Pr}(x) \geq \mathbf{Pr}(Q.\mathbf{Find\text{-}max})$, output x into the file for the set R.
- Else perform

  $y \leftarrow Q.$**Delete-min** and output y into the file for the set L

  $y \leftarrow Q.$**Delete-max** and output y into the file for the set R

  Choice can be made arbitrarily or randomly. Perform $Q.\mathbf{Insert}(x)$ to include x into the pivot set.

When all elements have been read, the elements from Q are output in the sorted order. Then sets L and R are sorted recursively and the algorithm can output the result.

The following application is described in [14]. Assume we want to search in a set of records by a given criterion and we want the found records to be ordered decreasingly by their relevance for this search. The number of output records will be limited by m. This can be done using a double ended priority queue.

Let the priority of the record be its relevance for the search. First we insert first m records into the priority queue Q. Then as we go through the remaining records, if for the record x, $\mathbf{Pr}(x) \leq \mathbf{Pr}(Q\mathbf{.Find\text{-}min})$, we can skip to the next record since the relevance of x is too small. If $\mathbf{Pr}(x) > \mathbf{Pr}(Q\mathbf{.Find\text{-}min})$, we perform Q**.Delete-min** and Q**.Insert**(x).

When all the records have been considered, the set represented by Q is the set of m records with the largest relevance among all records. Since Q is a double ended queue, it supports **Find-max** and **Delete-max** and the records can be output one by one in a decreasing order.

1.3 Generic methods for double ended heaps

There exist several implementations of double ended priority queues with varying performances. It has become apparent that a very logical approach is to reuse the implementations of single-ended priority queues, where they are used as a sort of a black-box. This way, whenever there is a need for some special property belonging to some type of a heap, then this heap can be used as the base heap. The advantage is that the structure is already implemented, analyzed theoretically and/or experimentally, so only a reasonable amount of time and resources are necessary.

The generic double ended heap is a scheme that, given compatible heaps as base heaps, can be turned into a particular implementation. There are three notable generic approaches[13]: *Dual heaps*, *Total correspondence heaps* and *Leaf correspondence heaps*. This work builds on and generalizes the leaf correspondence approach.

1.3.1 Dual heap

Possibly the most straightforward type of a generic double ended heap is a dual heap. To represent a set S of elements, a min-heap and max-heap are used, **both** representing the **whole** set S. For every element x, there is a pointer correspondence between node u representing x in min-heap and node v representing x in max-heap. This way, it is possible to get a reference on v knowing u and vice versa. Consequently, for every element it is sufficient to only have a reference to a node representing it in any of the heaps.

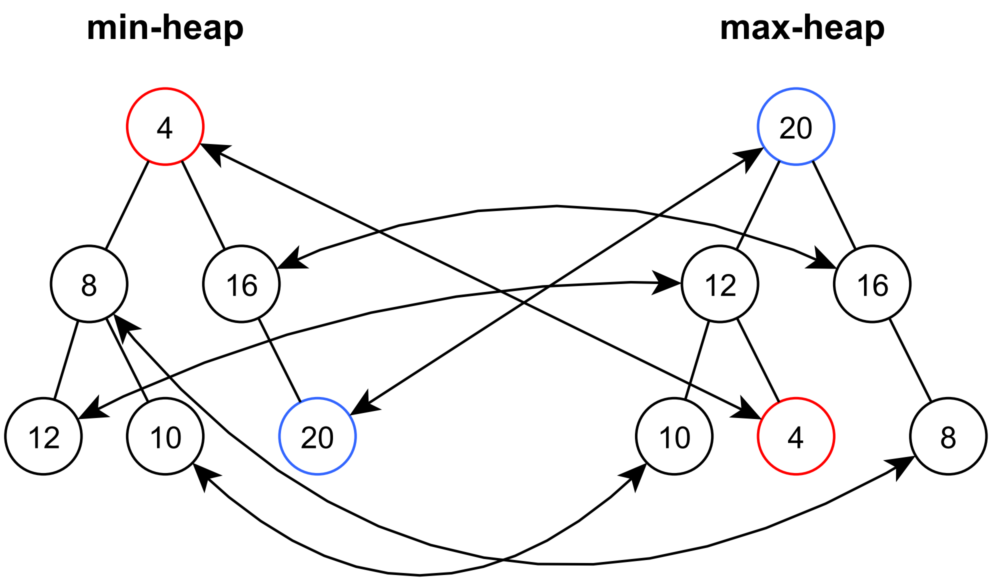

Figure 1.1: Dual heap - minimal element is represented by red nodes, maximal by blue nodes.

Under such conditions, the operations are very simple. Let Q_{min} and Q_{max} be base structures of the dual heap, the operations follow:

```
**Insert**($x$)
    $Q_{min}$.**Insert**($x$)
    $Q_{max}$.**Insert**($x$)
```

```
**Delete**($x$)
    $Q_{min}$.**Delete**($x$)
    $Q_{max}$.**Delete**($x$)
```

```
**Find-min**
    **return** $Q_{min}$.**Find-min**
```

```
**Find-max**
    **return** $Q_{max}$.**Find-max**
```

```
**Delete-min**
    $x \leftarrow Q_{min}$.**Find-min**
    $Q_{min}$.**Delete-min**
    $Q_{max}$.**Delete**($x$)
```

Delete-max

```
$x \leftarrow Q_{max}$.**Find-max**
$Q_{max}$.**Delete-max**
$Q_{min}$.**Delete**$(x)$
```

Decrease$(x,\ a)$

```
$Q_{min}$.**Decrease**$(x,\ a)$
$Q_{max}$.**Decrease**$(x,\ a)$
```

Increase$(x,\ a)$

```
$Q_{min}$.**Increase**$(x,\ a)$
$Q_{max}$.**Increase**$(x,\ a)$
```

Meld$(Q_1,\ Q_2)$

```
$Q.Q_{min} \leftarrow$ **Meld**$(Q_1.Q_{min},\ Q_2.Q_{min})$
$Q.Q_{max} \leftarrow$ **Meld**$(Q_1.Q_{max},\ Q_2.Q_{max})$
**return** $(Q.Q_{min}, Q.Q_{max})$
```

It is easy to see that algorithms for the dual heap are correct and straightforward to implement given the base heaps. It's most obvious disadvantages are that it takes twice as much memory as would be needed to represent each element once and that every operation is done twice. This makes it inferior in performance and effectivity to the next approach, the total correspondence heaps.

1.3.2 Total correspondence

In the dual heap, all elements were duplicated and they, along with their copies, formed two heaps linked by pointers. The total correspondence heap eliminates the need for duplicate elements and represents every element exactly once. It borrows its ideas from the **twin heap**[18]. In the twin heap, elements are represented by two binary (min- and max-) heaps in an array. Each of the min-heap and max-heap represent half of all elements - the possible odd element is stored in a **buffer**. Let **Element**$(buffer)$ be the element stored in the buffer, or *null* if the buffer is empty. The structure is a twin heap, if for every i, where $Q_{min}[i]$ (and $Q_{max}[i]$) is defined, $\mathbf{Pr}(Q_{min}[i]) \leq \mathbf{Pr}(Q_{max}[i])$.

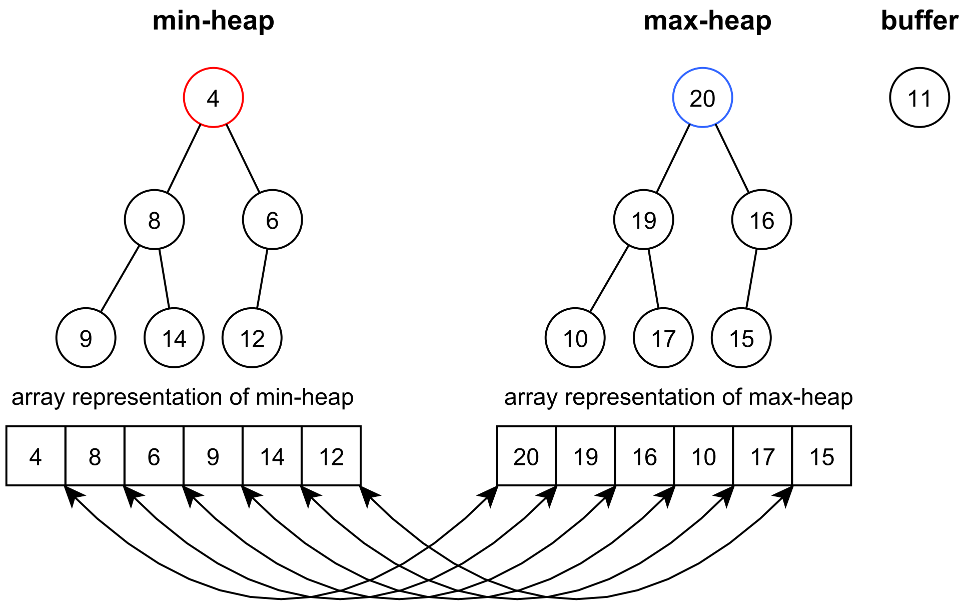
min-heap
max-heap
buffer
4
8
6
9
14
12
20
19
16
10
17
15
11
array representation of min-heap
array representation of max-heap
4
8
6
9
14
12
20
19
16
10
17
15

Figure 1.2: Twin heap - the minimal element is represented by a red node, maximal by a blue node.

This structure ensures that the minimal element is either in the buffer or is the minimal element of the min-heap and symmetrically for the maximal element.

The twin heap cuts space requirements of dual heaps to half and no longer has to be every operation performed twice. The problem is that this works only for heaps with an **array representation**.

The total correspondence heap is less restrictive and achieves the same results. The elements are represented by a min-heap and a max-heap and a buffer. Buffer stores a potential odd element, the remaining elements are uniformly split between both heaps. Since heaps may not be represented by an array, there are no implicit pairs of nodes that have to be in an ordering relation.

Instead, explicit pairs of nodes are formed: heaps are connected using additional pointers. Pair of nodes u and v can be connected if u is representing element x in min-heap and v is representing element y in the max-heap and $\mathbf{Pr}(x) \leq \mathbf{Pr}(y)$. Then v is a pointer pair node for u and u is a pointer pair node for v and we denote $\mathbf{Pointer}(u) = v$ and $\mathbf{Pointer}(v) = u$.

The structure consisting of a min-heap, max-heap and a buffer, is a **total correspondence heap** if every node from min-heap and max-heap has a pointer pair node from the other heap.

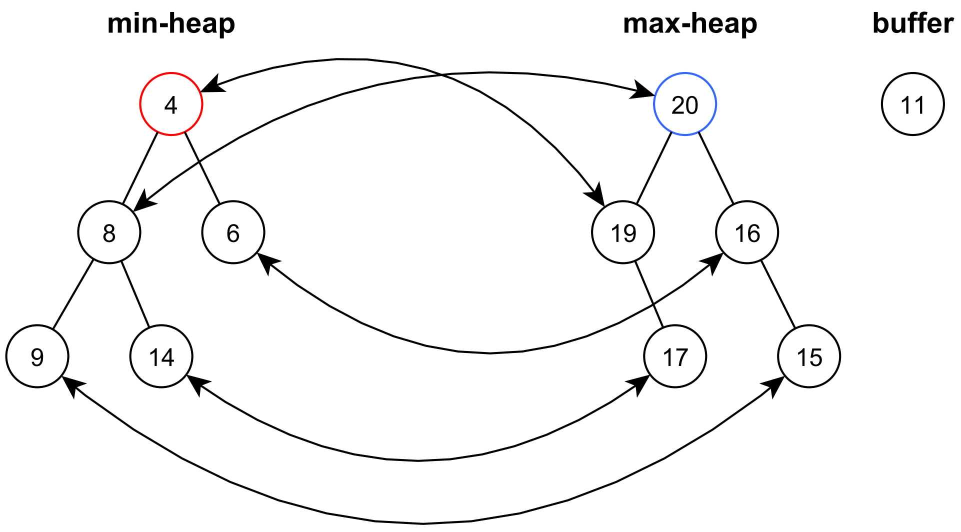

Figure 1.3: Total correspondence heap - minimal element is represented by a red node, maximal by a blue node. Every element has a corresponding and exclusive pair satisfying the ordering condition. There are no restrictions on the position of the nodes within the heap - unlike in the twin heap.

The operations must ensure that the total correspondence condition is satisfied at all times. The basic operations without their symmetrical counterparts follow.

Set-pointers(x, y) - establishes a pointer pair relation between two nodes

```
  **Pointer**($x$) $\leftarrow y$
  **Pointer**($y$) $\leftarrow x$
```

Insert(x)

```
  **if** $buffer$ is empty **then**
      $buffer \leftarrow x$
  **else**
      $small \leftarrow min\,\{$**Element**$(buffer), x\}$          ▷ Smaller element
      $large \leftarrow \{$**Element**$(buffer), x\} \setminus \{small\}$   ▷ Larger element
      $Q_{min}$.**Insert**($small$)
      $Q_{max}$.**Insert**($large$)
      **Set-pointers**($small$, $large$)
  **end if**
```

Find-min

```
  **if** $buffer$ is empty **then**
      **return** $Q_{min}$.**Find-min**
  **else**
      **return** $min\,\{Q_{min}$.**Find-min**, **Element**$(buffer)\}$
  **end if**
```

Delete-min - deletes the minimal element from the heap

```
if $buffer$ is empty then
    $y \leftarrow Q_{min}$.Find-min
    $z \leftarrow$ Pointer($y$)
    $Q_{min}$.Delete-min
    $buffer \leftarrow z$
else
    $b \leftarrow$ Element($buffer$)
    $y \leftarrow Q_{min}$.Find-min
    if Pr($b$) $\leq$ Pr($y$) then
        $buffer \leftarrow null$
    else
        $z \leftarrow$ Pointer($y$)
        $Q_{min}$.Delete($y$)                    ▷ $z$ lost its pointer pair $y$
        if Pr($b$) $\leq$ Pr($z$) then
            $Q_{min}$.Insert($b$)
            Set-pointers($b$, $z$)
        else
            $Q_{max}$.Delete($z$)
            $Q_{min}$.Insert($z$)
            $Q_{max}$.Insert($b$)
            Set-pointers($z$, $b$)
        end if
        $buffer \leftarrow null$
    end if
end if
```

Meld(Q_1, Q_2)

```
$Q.Q_{min} \leftarrow$ Meld($Q_1.Q_{min}$, $Q_2.Q_{min}$)
$Q.Q_{max} \leftarrow$ Meld($Q_1.Q_{max}$, $Q_2.Q_{max}$)
$Q = (Q.Q_{min}, Q.Q_{max})$
$B = \{$Element($Q_1.buffer$), Element($Q_2.buffer$)$\} \setminus \{null\}$
if $|B| = 1$ then
    let $B = \{b\}$
    $Q.buffer \leftarrow b$
else if $|B| = 2$ then
    let $B = \{b_1, b_2\}$ where Pr($b_1$) $\leq$ Pr($b_2$)
    $Q_{min}$.Insert($b_1$)
    $Q_{max}$.Insert($b_2$)
    Set-pointers($b_1$, $b_2$)
end if
return $Q$
```

In a same way that twin heap does, the total correspondence heap performs only a half of operations and uses half of space of the dual heap. It no longer requires an array representation and can use any heap as the base heap - as long as it is represented by a collection of trees with the min-heap ordering property.

Even better approach is however possible. It is not necessary that every node has a pointer pair to ensure that minimal and maximal elements are at correct locations. The pointer pair condition can be constricted to leaf nodes as will be shown in the next section.

1.3.3 Leaf correspondence

Our heap is based on the leaf correspondence heap of Chong and Sahni[4]. The double ended heap is created by interconnecting two base heaps: a min-heap and a max-heap, as defined in the previous chapter. The authors illustrate leaf correspondence on the Height biased leftist trees[6], Pairing heaps[8][16] and the Fast meldable priority queues of Brodal[2].

The leaf correspondence heap consists of two interconnected min- and max-heaps. Heaps are connected using additional pointers. A pair of nodes u and v can be connected if u is representing element x in the min-heap and v is representing element y in the max-heap and $\mathbf{Pr}(x) \leq \mathbf{Pr}(y)$. Then v is a pointer pair node for u and u is a pointer pair node for v and we denote $\mathbf{Pointer}(u) = v$ and $\mathbf{Pointer}(v) = u$. If node u has no pointer pair node, then $\mathbf{Pointer}(u)$ has *null* value.

The main requirement for the **leaf correspondence** is that each leaf u in the min-heap has a pointer pair node v in the max-heap that may or may not be a leaf and symmetrically, every leaf v in the max-heap has a pointer pair u in the min-heap that may or may not be a leaf.

This definition is not sufficient, because if there is only one element in the whole heap, then it is certainly a leaf and can not have a pointer pair, therefore there are some sets that are not representable by this heap. A separate container, denoted **buffer** is thus included in the heap. The buffer can hold one element or be empty. The second and probably more important use of the buffer is in most of the operations, because it helps build the leaf correspondence itself.

For example, to insert an element x, it must be ensured that in case it becomes a leaf, it has a pointer pair. This is easy to do with a buffer. If it is empty, the element is inserted there. If not, it must be inserted into one of the base heaps. If it becomes a leaf, the element from the buffer will serve as a pointer pair node.

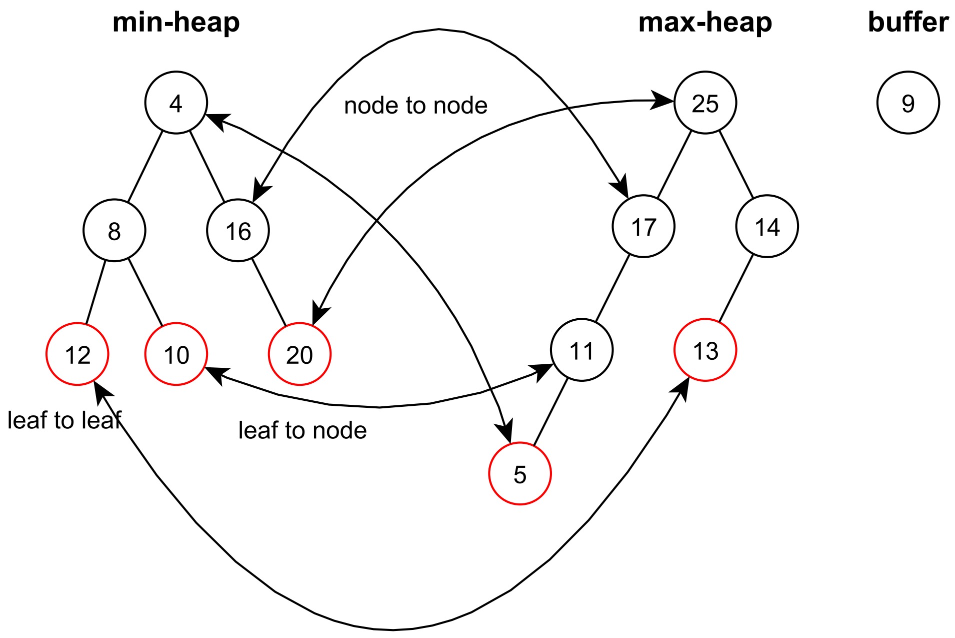

Figure 1.4: Leaf correspondence heap - red nodes are leaves and all such nodes must have a pointer pair. Every node is allowed to have a pointer pair however (note nodes with priorities 16 and 17 in the figure).

Let a priority queue be **meldable** if it supports the **Meld**(Q_1, Q_2) operation. The leaf correspondence heap of Chong and Sahni can be created from heaps that fulfill the following four conditions:

1. The (meldable) base heap supports the operation **Delete** efficiently.
2. When an element is inserted into the (meldable) base heap, no non-leaf node becomes a leaf node, except possibly the node for the newly inserted item.
3. When an element is deleted from the (meldable) base heap, no non-leaf node becomes a leaf node, except possibly the parent of the deleted node.
4. The **Meld** operation of the base heap (if supported) should not create new leaf nodes.

The fourth condition is only required if the new structure must support melding.

Basic operations

The operations of the leaf correspondence heap must ensure that the leaf correspondence condition holds at all times.

Set-pointers(x, y) - establishes a pointer pair relation between two nodes

```
    Pointer($x$) ← $y$
    Pointer($y$) ← $x$
```

Meld(Q_1, Q_2)

```
$Q.Q_{min} \leftarrow$ Meld($Q_1.Q_{min}$, $Q_2.Q_{min}$)
$Q.Q_{max} \leftarrow$ Meld($Q_1.Q_{max}$, $Q_2.Q_{max}$)
$Q = (Q.Q_{min}, Q.Q_{max})$
$B = \{\textbf{Element}(Q_1.buffer), \textbf{Element}(Q_2.buffer)\} \setminus \{null\}$
if $|B| = 1$ then
    let $B = \{b\}$
    $Q.buffer \leftarrow b$
else if $|B| = 2$ then
    let $B = \{b_1, b_2\}$ where $\textbf{Pr}(b_1) \leq \textbf{Pr}(b_2)$
    $Q_{min}$.Insert($b_1$)
    $Q_{max}$.Insert($b_2$)
    Set-pointers($b_1$, $b_2$)
end if
return $Q$
```

To meld two double ended heaps (if it is supported by base heap), the **Meld** operation of the base heap is used. First both min-heaps and both max-heaps are melded. Afterwards the potential elements from the buffer are reinserted and linked. The resulting heap is guaranteed to satisfy the pointer pair node condition due to requirement (4).

Insert(x) - inserts a new element into DEPQ

```
if $buffer$ is empty then
    $buffer \leftarrow x$
else
    $small \leftarrow min\{\textbf{Element}(buffer), x\}$                ▷ Smaller element
    $large \leftarrow \{\textbf{Element}(buffer), x\} \setminus \{small\}$     ▷ Larger element
    $Q_{min}$.Insert($small$)
    if $small$ is a leaf then
        $Q_{max}$.Insert($large$)
        Set-pointers($small$, $large$)
        $buffer \leftarrow null$
    else
        $buffer \leftarrow large$
    end if
end if
```

The **Insert** is straightforward. If the buffer is empty, the element is inserted into the buffer. If it is not, the smaller of the inserted and the element from the buffer is found. It is then inserted into the min-heap and in case it becomes a leaf, the larger will serve as a pointer pair node after insertion into the max-heap. In case the smaller element does not become a leaf, the larger element is left in the buffer.

The **Delete-min** operation of the leaf correspondence heap follows. Note that the following algorithm excludes the use of base heaps where a form of a cascaded cut is possible, such as the Fibonacci heaps. After a deletion other than of a node with the minimum priority, only the parent of a deleted node is considered to become a new leaf, while in fact, more than one nodes could have become leaves.

Delete-min - deletes the minimal element from the heap

```
if $buffer$ is empty then
    $y \leftarrow Q_{min}$.Find-min
    $z \leftarrow$ Pointer($y$)
    $Q_{min}$.Delete($y$)                              ▷ deleting the minimum
    if $z \neq null$ then
        if $z$ is not a leaf then
            Pointer($z$) $\leftarrow null$
        else
            $p \leftarrow$ Parent($z$)
            $Q_{max}$.Delete($z$)
            if $p$ is a leaf and Pointer($p$) $= null$ then
                $Q_{min}$.Insert($z$)
                Set-pointers($p$, $z$)
            else
                $buffer \leftarrow z$
            end if
        end if
    end if
else
```

Delete-min (continued - the buffer is not empty)

```
    $b \leftarrow$ **Element**($buffer$)
    $y \leftarrow Q_{min}$.**Find-min**
    $z \leftarrow$ **Pointer**($y$)
    **if Pr**($b$) $\leq$ **Pr**($y$) **then**
        $buffer \leftarrow null$
    **else**
        $Q_{min}$.**Delete**($y$)                                ▷ deleting the minimum
        **if** $z \neq null$ **then**
            **if** $z$ is not a leaf **then**
                **Pointer**($z$) $\leftarrow null$
            **else**
                **if Pr**($b$) $\leq$ **Pr**($z$) **then**
                    $Q_{min}$.**Insert**($b$)
                    **Set-pointers**($b$, $z$)
                    $buffer \leftarrow null$
                **else**
                    $p \leftarrow$ **Parent**($z$)
                    $Q_{max}$.**Delete**($z$)
                    $Q_{min}$.**Insert**($z$)
                    **if** either $p$ or $z$ became a leaf and **Pointer**($p$) $= null$ **then**
                        **Set-pointers**($z$, $p$)
                    **else**                                     ▷ $z \neq null$
                        **if** $z$ became a leaf **then**
                            $Q_{max}$.**Insert**($b$)
                            **Set-pointers**($z$, $b$)
                            $buffer \leftarrow null$
                        **end if**
                    **end if**
                **end if**
            **end if**
        **end if**
    **end if**
**end if**
```

The **Delete-min** is easier when the buffer is empty. The minimum is simply deleted from Q_{min}. If it had no pointer pair node, we are finished. If it had a pointer pair node z that is not a leaf, we simply cancel the pointer for this node. If the pointer pair node z is a leaf, we delete it from Q_{max}. If its parent p did not become a leaf, we simply put z into the buffer. If its parent p did become a leaf and has no pointer pair node, we put z into Q_{min} and make a pointer pair out of z and p.

If the buffer is not empty, we must also check the buffer for it may contain the smallest element. If it does not, we delete the node with minimum element from Q_{min}. If it has no pointer pair node, we are finished. If it has a pointer pair node z that is not a leaf, we cancel the pointer for this node. If the pointer pair node z is a leaf, we must fix the pointer for this node. If it represents element that is larger than one in the buffer, we can insert the element from buffer into Q_{min} and make them pointer pairs. Otherwise we delete z from Q_{max} and reinsert into Q_{min}. Now we must look at its previous parent p in Q_{max}. If none of them became a leaf after deletion and reinsertion, we are done. If any of them became a leaf, we will need to find a pointer pair. If p does not have a pointer pair, we can simply connect them and finish because z has just been inserted and has no pointer pair. If however p has a pointer pair and z became a leaf, we insert into Q_{max} a buffer element (that we know represents element larger than one represented by z) and make them a pointer pair.

After this operation is finished, every leaf has a correct pointer pair. The symmetrical procedure **Delete-max** is similar up to reversed inequalities and base heap references.

The complexity of **Insert** and **Meld** is equal to at most two times the complexity of **Insert** and **Meld** in the base heaps. The complexity of **Delete-min** is at most twice the complexity of **Delete** plus twice the complexity of **Insert** in base heaps.

2. L-correspondence heap

2.1 Definition

2.1.1 Conditions

Let **L-property** be a property of a node in the heap. For a given L-property we denote a node having this property **L-node**. As an example the L-property could be that a node is a leaf. In this case a leaf would be an L-node. If we want to emphasize that a node does not have the desired property, we denote it **non-L-node**.

The restriction on the base heaps can be **relaxed**. Instead of heaps that are collections of trees satisfying the min-heap ordering, we will allow that the relaxed min-heap is any collection of trees where in every tree, the **root has minimal priority** and for any node x in a tree, **there exists an L-node** y in the same tree such that $\mathbf{Pr}(x) \leq \mathbf{Pr}(y)$. Note that it is not necessary that $x \neq y$. From now on we will call this relaxed min-heap simply min-heap. The same applies to max-heap symmetrically.

Similarly as for the leaf correspondence, we define **L-correspondence**. The main requirement for the L-correspondence is that each L-node u in the min-heap has a pointer pair node v in the max-heap that may or may not be an L-node and symmetrically, every L-node v in the max-heap has a pointer pair u in the min-heap that may or may not be an L-node.

Lemma 2.1. *The L-correspondence requirement ensures that in a double ended heap made of a relaxed min-heap and max-heap, either the buffer contains an element with minimal priority, or one of the roots in min-heap represents an element with minimal priority. The same applies symmetrically to the element with maximal priority.*

Proof. Let x be any element with minimal priority. If it is in the buffer or the min-heap, we are finished, because the priority of a root is at most as large as of every other element in the tree. Let u be node from max-heap representing x. There exists an L-node v representing y such that $\mathbf{Pr}(y) \leq \mathbf{Pr}(x)$. L-correspondence ensures that v has a pointer pair node w representing element z in min-heap such that $\mathbf{Pr}(z) \leq \mathbf{Pr}(y)$. The root of tree containing z represents the element with minimal priority in the tree and since this is at most $\mathbf{Pr}(x)$, the statement holds. □

Let **singleton trees** be trees that consist of a single node.

We base the L-correspondence approach on a restricted deletion operation. Let **Delete-L-node**(x) be an operation that deletes an L-node x from the heap and if any new singleton trees are created during the process, the insertion of such nodes into the root list is deferred and a reference to these nodes is easily retrievable. The same will apply for the **Decrease** and **Increase** operations.

The conditions for the base min- and max- heaps of the L-correspondence heap are the following:

1. The (meldable) base heap supports the operation **Delete-L-node** efficiently.
2. When an L-node is **deleted** from the (meldable) base heap, or a priority of any node is **decreased** or **increased**, then
   - if any new singleton trees are created during the operation, they can not be put into the root list, but instead kept aside in a temporary data structure (such as a linked list) and it must be possible to retrieve this list in at most $O(1)$ additional time
   - apart from these nodes there is at most one new **arbitrary** L-node and reference to this L-node can be determined in $O(1)$ time
3. When an element is inserted into the (meldable) base heap, no non-L-node becomes an L-node, except possibly the node for the newly inserted element. Insertion of a singleton tree takes $O(1)$ time.
4. The **Meld** operation of the base heap (if supported) does not create new L-nodes.

For the operations **Delete-L-node**, **Decrease** and **Increase**, we will denote symbolically $(List, w) \leftarrow$ **Operation**(x) the fact that this operation provides a reference to the list of singleton trees and to one arbitrary L-node. Note that the retrieval of this pair is an implementation detail and even in the next chapter (specific implementations of base heaps for L-correspondence), its implementation will be omitted for clarity.

The conditions for the L-correspondence are similar to those for the leaf correspondence with a few notable differences:

- We generalize on the leaf correspondence approach to broaden the set of the eligible base heaps (for example Rank-pairing heap).
- We do not require the existence of a general **Delete** operation, but a specialized **Delete-L-node** operation is needed instead. Note that this operation may be implemented in the same way in the base heaps, but its analysis may differ and provide a better time complexity bound.
- The restrictions on **Delete-L-node** are different than on **Delete** in the leaf correspondence heaps. The reason is that the leaf correspondence conditions did not accommodate cases when several nodes are removed due to a **Delete** operation. For example this could happen in Fibonacci heap after deletion of a node x in case the path from **Parent**(x) to the root contains a continuous segment of marked nodes with exactly one child. All nodes from the path would become singleton trees and it is not guaranteed in any way that they would end up with the required pointer pair node. Our approach will therefore enable the use of Fibonacci heap and other heaps where deleting one node can induce deletion of other nodes.
- The conditions enable the **Decrease** and **Increase** operations.

2.1.2 Basic operations

The operations **Set-pointers** and **Insert** are the same as in the leaf correspondence heap and are shown for clarity, the **Meld** operation is slightly optimized.

Set-pointers(x, y) - establishes a pointer pair relation between two nodes

```
**Pointer**$(x) \leftarrow y$
**Pointer**$(y) \leftarrow x$
```

Insert(x) - inserts a new element into DEPQ

```
 1: **if** $buffer$ is empty **then**
 2:     $buffer \leftarrow x$
 3: **else**
 4:     $small \leftarrow min\,\{\textbf{Element}(buffer), x\}$                ▷ Smaller element
 5:     $large \leftarrow \{\textbf{Element}(buffer), x\} \setminus \{small\}$      ▷ Larger element
 6:     $Q_{min}$.**Insert**$(small)$
 7:     **if** $small$ is an L-node **then**
 8:         $Q_{max}$.**Insert**$(large)$
 9:         **Set-pointers**$(small, large)$
10:         $buffer \leftarrow null$
11:     **else**
12:         $buffer \leftarrow large$
13:     **end if**
14: **end if**
```

Meld(Q_1, Q_2)

```
$Q.Q_{min} \leftarrow$ **Meld**$(Q_1.Q_{min}, Q_2.Q_{min})$
$Q.Q_{max} \leftarrow$ **Meld**$(Q_1.Q_{max}, Q_2.Q_{max})$
$Q = (Q.Q_{min}, Q.Q_{max})$
$B = \{\textbf{Element}(Q_1.buffer), \textbf{Element}(Q_2.buffer)\} \setminus \{null\}$
**if** $|B| = 1$ **then**
    let $B = \{b\}$
    $Q.buffer \leftarrow b$
**else if** $|B| = 2$ **then**
    let $B = \{b_1, b_2\}$ where $\textbf{Pr}(b_1) \leq \textbf{Pr}(b_2)$
    $Q_{min}$.**Insert**$(b_1)$
    **if** $b_1$ is an L-node **then**
        $Q_{max}$.**Insert**$(b_2)$
        **Set-pointers**$(b_1, b_2)$
        $buffer \leftarrow null$
    **else**
        $buffer \leftarrow b_2$
    **end if**
**end if**
**return** $Q$
```

To meld two L-correspondence heaps (if it is supported by base heaps), the **Meld** operation of the base heap is used. First both min-heaps and both max-heaps are melded. Afterwards, the element with the smaller priority is inserted into the min-heap and if it is a leaf after the operation, the element with the larger priority is inserted into the max-heap and both nodes are linked. The resulting heap is guaranteed to satisfy the pointer pair node condition due to requirement (4).

2.2 Delete-min and Decrease

2.2.1 Fix-L-node

In the L-correspondence conditions, we allow certain base operations to create new L-nodes. We define a procedure **Fix-L-node** that handles such nodes to ensure the L-correspondence. Let x be a node from the L-correspondence. Then **Heap**(x) denotes the heap in which x is located (the min-heap or the max-heap).

Fix-L-node(x)

```
1:  $Q \leftarrow$ Heap($x$)
2:  if $x \neq null$ and Pointer($x$) $= null$ then
3:      $(L, w) \leftarrow Q$.Delete-L-node($x$)
4:      Insert($x$)
5:      if $L$ is not empty then
6:          let $L_s$ be the set of all singleton trees with a pointer pair in $L$
7:          for all $s \in L_s$ do
8:              $Q$.Insert($s$)
9:          end for
10:         split $L \setminus L_s$ into subsets $L_i$ of size 2 and let $l$ be a possible odd node
11:         if $l \neq null$ then
12:             Insert($l$)
13:         end if
14:         for all $L_i = (l_1, l_2)$ do
15:             if $l_1 > l_2$ then
16:                 swap $l_1$ and $l_2$
17:             end if
18:             $Q_{min}$.Insert($l_1$)
19:             $Q_{max}$.Insert($l_2$)
20:             Set-pointers($l_1$, $l_2$)
21:         end for
22:     end if
23:     if $w \neq null$ then
24:         Fix-L-node($w$)
25:     end if
26: end if
```

The **Fix-L-node** procedure ensures that the L-correspondence condition holds by forming pairs of singleton trees that had no pointer pairs (result of the **Delete-L-node** operation) and reinserting them into both heaps. The **Delete-L-node** procedure can create one new arbitrary L-node by definition, in which case a recursive call must be made. It is easy to see that **Fix-L-node** always terminates (in every recursive call it removes and reinserts at least one L-node with no pointer pair and never breaks a pointer pair connection). The **Delete-L-node** operation must be designed in a way that ensures that **Fix-L-node** performs fast. **Fix-L-node** accepts *null* node argument for the ease of use.

It is not enough if the bound for **Delete-L-node** or **Insert** is given. It must be possible to give a time bound for the whole **Fix-L-node** operation including all recursive calls and all calls to **Delete-L-node**. It is therefore fully dependent on the **Delete-L-node** of the base heap.

This analysis is left to the specific implementations and no reasonable time bounds for our double ended heap can be made without it. The analysis of this operation for some well known base heaps will be shown in next chapter. We will show that for these heaps, the amortized cost of the **Fix-L-node** operation is $O(1)$.

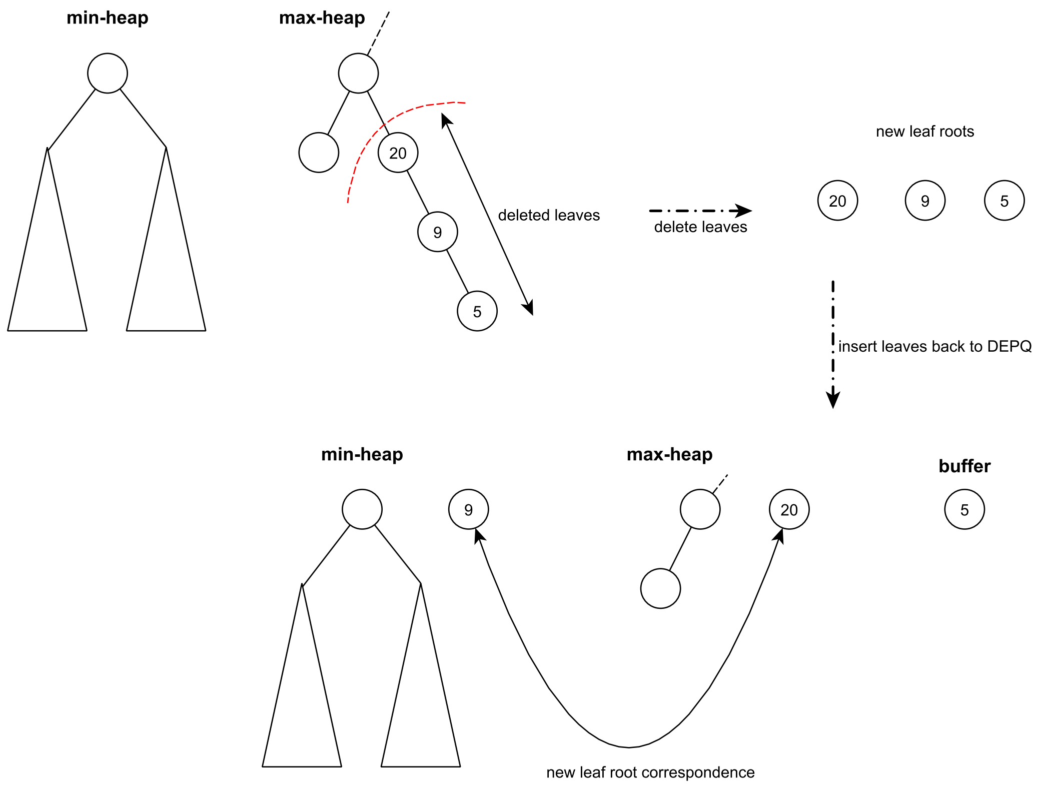

Figure 2.1: Example of a **Fix-L-node** operation (L-property ≡ leaf) - leaves are deleted until **Delete-L-node** creates no more leaves. Then they are reinserted using the **Insert** operation.

2.2.2 Delete-min

The new **Delete-min** operation no longer needs a general **Delete** in the base heaps and uses **Delete-L-node** instead.

Delete-min - deletes the minimal element from the heap

```
1: $y \leftarrow Q_{min}$.Find-min
2: $z \leftarrow$ Pointer($y$)
3: $Q_{min}$.Delete-min
4: if $z \neq null$ then
5:     if $z$ is not an L-node then
6:         Pointer($z$) $\leftarrow null$
7:     else
8:         Fix-L-node($z$)
9:     end if
10: end if
```

The **Delete-max** operation is defined symmetrically.

For the purposes of the time complexity analysis, let $c^A_{\text{operation}}(n)$ denote the amortized complexity of the operation in the base heap Q_{min} with n elements for the potential function $\Phi_{base}(Q_{min})$. Then we denote $C^A_{\text{operation}}(n)$ the amortized complexity of the operation in the L-correspondence Q for the potential function $\Phi(Q) = \Phi_{base}(Q_{min}) + \Phi_{base}(Q_{max})$.

The upper bound on the time complexity of **Delete-min** is then:

$$C^A_{\text{Delete-min}}(n) \leq c^A_{\text{Delete-min}}(n) + c^A_{\text{Fix-L-node}}(n)$$

2.2.3 Decrease

The **Fix-L-node** operation enables the operation **Decrease** for the L-correspondence heap.

```
Decrease($x$, $a$)
 1: if $x$ is in $Q_{min}$ then
 2:     $(L, w) \leftarrow Q_{min}$.Decrease($x$, $a$)
 3:     Fix-L-node($w$)
 4: else                                                  ▷ $x$ is in $Q_{max}$
 5:     if Pointer($x$) $= null$ or Pr(Pointer($x$)) $\leq a$ then
 6:         $(L, w) \leftarrow Q_{max}$.Decrease($x$, $a$)
 7:         Fix-L-node($w$)
 8:     else                                      ▷ Pr(Pointer($x$)) $> a$
 9:         $y \leftarrow$ Pointer($x$)                            ▷ $y \in Q_{min}$
10:         $b \leftarrow$ Pr($y$)
11:         swap elements $x$ and $y$ leaving the priorities in place
12:         $(L_1, w) \leftarrow Q_{min}$.Decrease($x$, $a$)
13:         $(L_2, z) \leftarrow Q_{max}$.Decrease($y$, $b$)
14:         Fix-L-node($w$)
15:         Fix-L-node($z$)
16:         $L \leftarrow L_1 \cup L_2$
17:     end if
18: end if
19: if $L$ is not empty then
20:     split $L$ into subsets $L_i$ of size 2 and let $l$ be a possible odd node
21:     if $l \neq null$ then
22:         Insert($l$)
23:     end if
24:     for all $L_i = (l_1, l_2)$ do
25:         if $l_1 > l_2$ then
26:             swap $l_1$ and $l_2$
27:         end if
28:         $Q_{min}$.Insert($l_1$)
29:         $Q_{max}$.Insert($l_2$)
30:         Set-pointers($l_1$, $l_2$)
31:     end for
32: end if
```

If node x to be decreased is in Q_{min}, it is decreased using the base heap **Decrease**. The conditions of the L-correspondence heap allow it to create any number of singleton trees and at most one arbitrary L-node. The singleton trees are reinserted and a possible L-node is fixed.

If node x is in Q_{max}, we have to check for violation of the L-correspondence condition. If x has no pointer pair node or the new decrease value is larger that the one represented by its pointer pair node, the **Decrease** in Q_{max} is performed which as in previous case can create singleton trees and at most one arbitrary L-node.

Otherwise we swap the elements leaving the priorities in place and then perform **Decrease** in the both heaps. The swapping of elements is correct, because each element was given the priority of the other element and the min-heap and max-heap condition were not affected. Both **Decrease** operations can create new singleton trees and at most one arbitrary L-node each. These must be all fixed to ensure the L-correspondence.

At the end of the algorithm, all singleton trees from the **Decrease** and **Increase** operations are reinserted and the L-correspondence is ensured.

The upper complexity bound for **Decrease** is then:

$$C^A_{\text{Decrease}}(n) \leq c^A_{\text{Decrease}}(n) + c^A_{\text{Increase}}(n) + 2c^A_{\text{Fix-L-node}}(n)$$

There is an iterated reinsertion of singleton trees at the end of the algorithm. The conditions of the L-correspondence heap ensure that insertion of the singleton trees into the base heaps takes $O(1)$, therefore the reinsertion can be accounted to the **Fix-L-node** operation.

Note that the **Increase** operation of the L-correspondence heap is symmetrical up to the reverse ordering relations.

2.3 Comparison with other structures

As we saw in the previous chapter, there are three important generic double ended priority queue approaches: Dual Heaps, Total Correspondence Heaps and Leaf Correspondence Heaps.

Dual heaps store all elements (or nodes representing them) twice. This makes them very memory consuming, but even worse, every operation is performed twice. This is obvious for **Delete** or **Insert**. The bigger problem is that the **Decrease** in min-heap is always paired with a corresponding **Decrease** in max-heap which is usually a much more expensive operation.

Total correspondence heaps, in the same way as the leaf correspondence heaps, store each element once, either in the min-heap, the max-heap or in the buffer. The difference is that every node in one heap must have a pointer pair node from the other heap. This constraint causes that for every removal of node x in one heap, the correspondence condition will have to be ensured for its pointer pair every time. In leaf correspondence heaps, the conditions have to be ensured only in cases when x has a pointer pair and only when this pointer pair is a leaf. This leads to belief that leaf correspondence heaps are faster than total correspondence heaps in general. It has also been shown experimentally[4] that this suggestion is probably legitimate.

It seems that the leaf correspondence is the most effective way of building a double ended priority queue. A more general correspondence approach is the L-correspondence that allows the use of a broader set of base heaps and makes use of the fact, that deleting a leaf (or generally an L-node) may be a very fast operation.

To support the **Increase** operation, instead of performing **Delete** and **Insert** or even worse a **Decrease**, **Delete-min** and **Insert**, it can be desirable if the

Increase operation of the base heap is used directly. The problem is that that **Decrease** and **Increase** of the base heap may create new L-nodes. We use the **Fix-L-node** operation to handle such cases.

In the next chapter we will show how to extend several well known heaps to support the **Delete-L-node** operation in a way that the amortized time bound for the **Fix-L-node** operation is $O(1)$. Using this time bound, the time bounds for the **Delete-min** and **Decrease** procedures of the L-correspondence heap will be derived.

A separate, but interesting result is that the expected amortized cost for **Increase** is $O(1)$ for some types of base heaps. We will be depending on the following general probabilistic bound for the number of children in a rooted tree.

Lemma 2.2. *The expected number of children in a node selected with uniform probability from a rooted tree is bounded by a constant. Same applies to a node selected with uniform probability from a collection of rooted trees.*

Proof. For uniform probabilistic distribution, the expected value is equal to average number of children. Let the number of nodes be n. Every edge represents one parent-children relation and there is exactly $n-1$ edges in a tree. Or equivalently, exactly $n-1$ nodes are children to some node. The average number of children is then $\dfrac{n-1}{n} = O(1)$. For a collection of rooted trees, the average number of children is even less. $\square$

3. L-correspondence implementations

A general approach for building a double ended priority queue from two (min and max) heaps has been shown. There are several conditions that have to be satisfied by a base heap in order to transform two instances of the base heap into a double ended heap. We will show how these conditions can be satisfied in some well known heaps.

3.1 Fibonacci heap

The Fibonacci heap has been proved useful in the area of graph algorithms, so it is natural to expect its double ended variant could perform well in a similar context. It's effectivity for some algorithms is based on very fast **Decrease** operation.

3.1.1 Description

The Fibonacci heap was first introduced in 1987 by Fredman and Tarjan[9] as an extension to the Binomial heap invented by Vuillemin [17]. The binomial queues are famous for their simplicity and intuitiveness and the fact that all priority queue operations have $O(\log n)$ worst-case time bound. Fibonacci heap is more complex to implement and analyze, but it achieves $O(1)$ amortized time for **Insert**, **Find-min**, **Meld** and **Decrease** and $O(\log n)$ amortized time for **Delete-min** and **Increase**.

The Fibonacci heap is a collection of trees satisfying the min-heap ordering property, that is, rooted trees where the priority of a child is greater or equal than the priority of its parent. The trees are stored in a circular list and there is a special minimum pointer that always points on a root with minimal priority. This list is called the *root list*.

Every node stores its *priority*, a pointer to its *parent*, a pointer to its *first child* and pointers to its *left sibling* and *right sibling*. With such structure it is possible to iteratively walk through any node's children list in time equal to the number of its children. It is also possible to cut out a node in constant time by removing it from the doubly linked list using references to left and right siblings.

Apart from pointers, every node keeps record of the number of its children (this is possible and straightforward to implement in constant time), the *rank*, and a boolean value indicating whether it is *marked*. Intuitively, a marked node is a node that had one child removed and has not been a root since, thus a number of its descendant decreased by a possibly large amount. Whenever a child is removed from its marked parent, the parent is removed too, is unmarked, added into the root list and possibly forms a new tree. This ensures that trees never have too little descendants with respect to its rank. It will be shown that number of descendants is always exponential in its rank.

Eligibility for the L-correspondence

First we define that L-property for a node x in the Fibonacci heap:

$$x \text{ is an L-node } \equiv x \text{ is a leaf}$$

Then the condition for relaxed min-heap (page 19) holds. This is also clear from the fact that the relaxed min-heap is a generalization of a usual min-heap satisfying the min-heap ordering property.

From the definition of the operations it will be clear that all L-correspondence conditions are satisfied. Notably the condition of instant retrieval of the $(List, w)$ pair after the **Delete-L-node**, **Decrease** and **Increase** operations is easily satisfied, but its implementation is **not included** in these operations for clarity.

Operations

The algorithms for the Fibonacci heap operations follow. Note that every time a root list is changed (a root is added or it is concatenated with another root list), it is ensured in constant time that a root with minimal priority is referenced by a minimum pointer.

Insert(x)

```
create a new node $v$ representing element $x$ and add $v$ into the root list
```

Meld(Q_1, Q_2)

```
concatenate the root lists of $Q_1$ and $Q_2$
```

Find-min

```
$x \leftarrow$ root referenced by the minimum pointer
return a reference to $x$
```

Link(x, y) - link two trees of equal rank rooted in x and y

```
if Pr($x$) > Pr($y$) then
    swap $x$ and $y$
end if
make $x$ the first child of $y$
return $x$
```

```
Delete-min
  $x \leftarrow$ first root in root list
  remove $x$ from root list
  for all child $y$ of $x$ do
      Parent($y$) $\leftarrow$ $null$
      add $y$ into root list
  end for
  $\forall i$ let $T_i$ be a set of all trees of rank $i$
  clear root list
  $i \leftarrow 0$
  while $\exists i \, |T_i| \geq 1$ do
      while $|T_i| > 1$ do
          remove any two trees $t_1$ and $t_2$ from $T_i$
          add Link($t_1$, $t_2$) to $T_{i+1}$
      end while
      if $|T_i| = 1$ then
          remove tree $t$ from $T_i$ and add to root list
      end if
      $i \leftarrow i + 1$
  end while
```

The **Cut**(x) procedure is used by the **Decrease**(x) and **Increase**(x) operation in case the min-heap ordering property has been breached. In this case, the whole subtree of x is cut out and added to the root list. If parent y of the node x is marked, recursively cut y.

```
Cut($x$)
  $y \leftarrow$ Parent($x$)
  if $y = null$ then
      return
  end if
  remove $x$ from its parent $y$ by setting appropriate pointers
  Parent($x$) $\leftarrow$ $null$
  add $x$ into root list
  Marked($x$) $\leftarrow$ $false$
  if Marked($y$) then
      Cut($y$)
  else if $y$ is not a root then
      Marked($y$) $\leftarrow$ $true$
  end if
```

Decrease(x, a)

```
$\mathbf{Pr}(x) \leftarrow a$
$y \leftarrow \mathbf{Parent}(x)$
if $y = null$ then
    return
end if
if $\mathbf{Pr}(x) \leq \mathbf{Pr}(y)$ then
    Cut($x$)
end if
```

The performance of the heap can be analyzed using a potential function defined as $\Phi_F = 2m + t$, where t is the number of trees in heap and m is number of marked nodes. This leads to the following (see [5]):

Lemma 3.1. *The number of descendants of any node with rank r is at least F_{r+2}, where F_k is the kth Fibonacci number.*

Theorem 3.2. *The amortized time bound (with Φ_F as the potential function) in Fibonacci heap for operations **Insert**, **Meld**, **Find-min** and **Decrease** is $O(1)$ and amortized time bound for **Delete-min** is $O(\log n)$.* □

We will use a different approach however, because we are interested in the time complexity of **Fix-L-node** (page 22). We will define a new potential function for the modification of the Fibonacci heap and show that using this potential function, all basic operations have the same amortized time cost as in the original Fibonacci heap using Φ_F. Then we will show that using the new potential, the operation **Fix-L-node** of the L-correspondence heap has $O(1)$ amortized time bound and from the operation **Increase** of Fibonacci min-heap we will derive a bound for the L-correspondence **Decrease** operation.

Let

$$\Phi = 7m + 4s + 2t$$

be a potential function for the L-correspondence modification of the Fibonacci heap, where m is the number of marked leaves, s is the number of nodes that have only one child and t is the number of trees in root list.

For every operation, let Δm, Δs, Δt be the difference between the number of marked nodes, nodes with one child, and trees, respectively, after the operation, and this number before the operation.

For the operations **Increase** and **Delete-min** let c be the number of children of the argument of the operation. From Lemma 3.1, $c \leq O(\log n)$.

We can then compute the amortized time complexity for the potential Φ. Note that all of the following operations are for the Fibonacci **min**-heap.

Insert - the time of the operation is $O(1)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t \\ &= 0 + 0 + 2 \\ &= 2\end{aligned}$$

Meld - the time of the operation is $O(1)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t\\ &= 0 + 0 + 0\\ &= 0\end{aligned}$$

Find-min - the time of the operation is $O(1)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t\\ &= 0 + 0 + 0\\ &= 0\end{aligned}$$

Decrease - let k be the number of **Cut** operations, i.e. the number of nodes that ceased to be marked plus one. One node may have been marked. There are k new trees. Each one of the k cuts may have created one node with only one child (see Fig. 3.1). The time of the operation is $O(k)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t\\ &= -7(k-2) + 4k + 2k\\ &= -k + 14\end{aligned}$$

Delete-min - let T be the set of trees in the root list right before the operation. After the linking, there is at most one tree of each rank. The number of trees after the linking is therefore at most $\log n$ (Lemma 3.1). No nodes are marked, some may become unmarked. During the linking, only nodes of rank 0 can create new nodes with only one child and it takes two such nodes to make one with only one child. Furthermore roots of rank 1 (that have exactly one child) will be linked together in pairs, so only at most a half of these trees will have exactly one child after the linking. The possible increase in nodes with only one child is therefore $\Delta s \leq \frac{|T|}{4}$. The time of the operation is $O(|T| + \log n)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t\\ &\leq 0 + 4\frac{1}{4}(|T| + c) + 2(\log n - |T|)\\ &\leq -|T| + 3\log n\end{aligned}$$

Theorem 3.3. *The amortized time bound (with Φ as the potential function) in the Fibonacci min-heap for operations **Insert**, **Meld**, **Find-min** and **Decrease** is $O(1)$ and amortized time bound for **Delete-min** is $O(\log n)$.* □

We now show that using the potential function Φ we get a desired amortized time bound for the **Fix-L-node** operation (page 22) of the L-correspondence heap using a very natural **Delete-L-node** operation. Then a probabilistic time bound for the **Decrease** operation of the L-correspondence heap will be given.

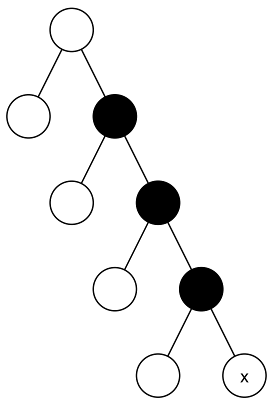

Figure 3.1: Cutting can lead to nodes with one child: cutting out node x causes three more cuts, unmarks 3 nodes, marks one node and creates 3 nodes with exactly one child.

Delete-L-node and Increase

Delete-L-node(x)

```
$y \leftarrow$ Parent($x$)
if $y = null$ then
    remove $x$ from root list
    return
end if
remove $x$ from its parent $y$ by setting appropriate pointers
if Marked($y$) then
    Cut($y$)
else if $y$ is not a root then
    Marked($y$) $\leftarrow$ $true$
end if
```

During **Delete-L-node**, the leaf is removed from its parent or from root list in case it has no parent. If it had a parent, this gets marked if it was unmarked. If it was marked however, we must call the **Cut** procedure to ensure that crucial properties of the Fibonacci heap are not broken.

The implication of this is that the **Fix-L-node** operation (page 22) cuts out a sequence of leaves one by one until an unmarked node that is either non-leaf or a leaf with pointer pair, is reached.

Increase(x, a)

```
Pr($x$) ← $a$
$y$ ← Parent($x$)
if $y \neq null$ then
    Cut($x$)
end if
for all child $y$ of $x$ do
    remove $y$ from its parent $x$ by setting appropriate pointers
    add $y$ into root list
end for
```

3.1.2 Fibonacci heap in the L-correspondence

We will now analyze the **Fix-L-node** operation (page 22) of the L-correspondence heap with Fibonacci heap as the base heap. We are interested in the **total change of potential** in both heaps.

Fix-L-node

The time of the operation is clearly $O(k)$, where k is the number of nodes cut out either because they were marked, or because they became leaves after their only child had been cut out.

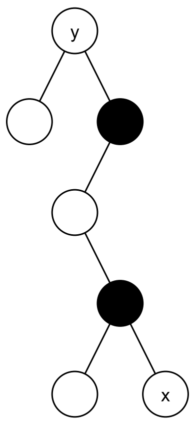

Figure 3.2: **Fix-L-node**(x) stops when y is reached, because it is unmarked and a non-leaf.

One node can become marked and some other nodes can become unmarked, but there is no guarantee on the number of such nodes. Let c_m be the number of nodes cut out because they were marked and c_s be the number of nodes cut out because they became leaves during the operation (they had only one child before). We have $k = c_m + c_s$. The number of new trees in both heaps in total is $k + 1$. The time of the operation is $O(k)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t \\ &\leq -7(c_m - 1) - 4c_s + 2(k+1) \\ &= -3c_m - 2k + 9\end{aligned}$$

Decrease

To analyze the **Decrease** operation (page 24) of the L-correspondence heap, we must give amortized time bound for every sub-routine as in:

$$C^A_{\text{Decrease}}(n) \leq c^A_{\text{Decrease}}(n) + c^A_{\text{Increase}}(n) + 2c^A_{\text{Fix-L-node}}(n)$$

It remains to show the time bound for the min-heap **Increase** operation. Its time cost is dependent on the amount of children and this in the Fibonacci heap is bounded by $O(\log n)$. If we pick a node at random, the expected amortized cost is constant.

Lemma 3.4. *If the argument of* ***Increase*** *operation in Fibonacci min-heap is a node selected with uniform probability, the expected amortized time bound (with* Φ *as the potential function) is* $O(1)$.

Proof. Let k be the number of **Cut** operations, i.e. the number of nodes that ceased to be marked plus one. One node may have been marked. Each one of the k cuts may have created one node with only one child. Let c be the number of children of the increased node. There are $1 + k + c$ new trees. The time of the operation is $O(k + c)$, the change of potential:

$$\begin{aligned}\Delta\Phi &= 7\Delta m + 4\Delta s + 2\Delta t \\ &\leq -7(k-2) + 4k + 2(k+1+c) \\ &= -k + 2c + 16\end{aligned}$$

If we let c be a random variable from a probabilistic space where every node is selected with uniform probability, applying Lemma 2.2, we get the desired bound. □

All results together give us the following:

Theorem 3.5. *The expected amortized cost (with* Φ *as the potential function) of operation* ***Decrease*** *on a node selected with uniform probability in the L-correspondence heap, with Fibonacci heap as the base heap, is bounded by a constant.* □

We give the following overview for the L-correspondence heap using the Fibonacci heap as the base heap.

Operation	Worst case	Amortized	Expected amortized
Insert	$O(1)$	$O(1)$	$O(1)$
Meld	$O(1)$	$O(1)$	$O(1)$
Find-min and **Find-max**	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(n)$	$O(\log n)$	$O(\log n)$
Decrease and **Increase**	$O(n)$	$O(\log n)$	$O(1)$

3.2 Thin heap

Thin and Thick Heaps are inspired by the Fibonacci heap, but are less restrictive. The relaxation is sufficient to give amortized efficiency of the Fibonacci heap. They require less space than the Fibonacci heap and are expected to perform better in practice because the number of cuts is minimized.

3.2.1 Description

The Thin and Thick heaps were introduced in 2008 by Kaplan and Tarjan[11]. The authors gave two variants of a Thin heap and analogous structure Thick heap. Both Thin heap variants are asymptotically more efficient than the Thick heap and the second variant is more promising in practical use compared to the first. We will describe and build upon the second variant of the Thin heap.

Thin trees

A **thin tree** is a tree where priorities of nodes satisfy the min-heap ordering condition and where every node has a non-negative integer $rank_T$ with the following properties:

1. The children of a node with k children have $rank_T$ from $k-1$ to 0 and are ordered decreasingly in the children list: $k-1$, $k-2$, …, 0.
2. A node with $rank_T$ k has k or $k-1$ children. It is denoted **normal** if it has k children, otherwise it is **thin**.
3. The root is always normal.

Note that a thin tree whose all nodes are normal is a binomial tree and every binomial tree is a thin tree with all nodes normal.

Lemma 3.6. *The number of descendants of any node (including itself) in a thin tree with r children is at least F_{r+1}, where F_k is the kth Fibonacci number.*

Proof. We will prove by induction by the number of children. Nodes with 0 and 1 children have $1 \geq F_1$ and $2 \geq F_2$ descendants respectively. For $k \geq 2$, let x be a node with k children, thus its children have the following $rank_T$s: $k-1, k-2, \ldots, 0$. A child with $rank_T$ l has at least $l-1$ children. By the induction hypothesis and a well known equation for the sum of a continuous sequence of Fibonacci numbers, the number of descendants of x including itself is at least $1 + F_{k-1} + F_{k-2} + \ldots + F_0 = F_{k+1}$. □

Structure

The structure consists of a circular linked root list, where it is ensured that a root with minimal priority is always pointed on by a special minimum pointer. There are three pointers per node: to the first child; to the right sibling (or next root if node is root); and to the left sibling or the parent, if node is a first child. Every node stores a non-negative integer $rank_T$. A singleton tree (in root list) has $rank_T$ 0.

Eligibility for the L-correspondence

As in the Fibonacci heap we can define that L-property for a node x in the Thin heap is:

$$x \text{ is an L-node } \equiv x \text{ is a leaf}$$

Since Thin heap satisfies the min-heap ordering property, it is also a relaxed min-heap (page 19).

From the definition of the operations it will be clear that all L-correspondence conditions are satisfied. Notably the condition of instant retrieval of the $(List, w)$ pair after the **Delete-L-node**, **Decrease** and **Increase** operations is easily satisfied, but its implementation is **not included** in these operations for clarity.

Basic operations

The algorithms for the Thin heap operations follow. Every time a root list is changed (a root is added or it is concatenated with another root list), it is ensured in constant time that a root with minimal priority is referenced by a minimum pointer.

In the following algorithms, to "make node x normal" means decrementing its $rank_T$ by one in case x was thin. This has to be done everytime a node with its subtree is added into the root list to ensure the condition (3).

Insert(x)

```
create a new node $v$ representing element $x$ and add $v$ into the root list
$rank_T(v) \leftarrow 0$
```

Find-min

```
$x \leftarrow$ root referenced by the minimum pointer
return a reference to $x$
```

Meld(Q_1, Q_2)

```
concatenate the root lists of $Q_1$ and $Q_2$
```

Link(x, y) - link two normal trees of equal $rank_T$ rooted in x and y

```
if Pr($x$) > Pr($y$) then
    swap $x$ and $y$
end if
make $x$ the first child of $y$
$rank_T(y) \leftarrow rank_T(y) + 1$
return $x$
```

Delete-min

```
$x \leftarrow$ root referenced by the minimum pointer
remove $x$ from the root list
**for all** child $y$ of $x$ **do**
    **Parent**$(y) \leftarrow null$
    add $y$ into the root list and make it normal
**end for**
$\forall i$ let $T_i$ be a set of all trees of $rank_T$ $i$
clear the root list
$i \leftarrow 0$
**while** $\exists i \, |T_i| \geq 1$ **do**
    **while** $|T_i| > 1$ **do**
        remove any two trees $t_1$ and $t_2$ from $T_i$
        add **Link**$(t_1, t_2)$ to $T_{i+1}$
    **end while**
    **if** $|T_i| = 1$ **then**
        remove tree $t$ from $T_i$ and add into the root list
    **end if**
    $i \leftarrow i + 1$
**end while**
```

In the Fibonacci heap, if the min-heap ordering condition is violated, a (possibly cascaded) cut is made. In the Thin heap, only one cut followed by reinsertion into the root list is made. To ensure that the thin tree conditions are satisfied, this cut is followed by restructuring operations during which $rank_T$ of some nodes may be changed and some nodes can become children of their siblings.

After the initial cut of a node x, let y be its first left sibling or parent if it has no left siblings. There are three ways of violating the main conditions at node y:

Violation of condition 1 - Node y has $rank_T$ two greater than that of its first right sibling, or has $rank_T$ 1 and no right siblings.

Violation of condition 2 - Node y has $rank_T$ three greater than that of its first child, or has $rank_T$ 2 and no children.

Violation of condition 3 - Node y is a thin root.

To cope with the violations the procedure **Fix-violation**(y) is defined. This procedure can create new violation in left sibling or parent of y in which case it calls itself recursively.

Fix-violation(x)

```
1: if Condition 1 is violated then
2:     if $x$ is thin then
3:         $y \leftarrow$ first left sibling of $x$ or parent if $x$ has no left siblings
4:         $rank_T(x) = rank_T(x) - 1$            ▷ May create a new violation at $y$.
5:         Fix-violation($y$)
6:     else if $x$ is normal then
7:         remove first child $z$ of $x$ and make $z$ right sibling of $x$
8:     end if                                   ▷ Does not create a new violation.
9: else if Condition 2 is violated then
10:     $z \leftarrow$ right sibling of $x$
11:     if $z$ is normal then
12:         $rank_T(z) = rank_T(z) + 1$
13:         $rank_T(x) = rank_T(x) - 1$
14:         swap $x$ and $z$ in the children list   ▷ Does not create a new violation.
15:     else if $z$ is thin then
16:         $y \leftarrow$ first left sibling of $x$ or parent if $x$ has no left siblings
17:         if Pr($z$) $\leq$ Pr($x$) then
18:             $rank_T(x) = rank_T(x) - 2$
19:             make $x$ the first child of $z$
20:         else if Pr($x$) $\leq$ Pr($z$) then
21:             $rank_T(x) = rank_T(x) - 1$
22:             $rank_T(z) = rank_T(z) - 1$
23:             make $z$ the first child of $x$
24:         end if                               ▷ May create a new violation at $y$.
25:         Fix-violation($y$)
26:     end if
27: else if Condition 3 is violated then
28:     $rank_T(x) = rank_T(x) - 1$                          ▷ Makes root normal.
29: end if
```

The **Decrease** operation then follows:

Decrease(x, a)

```
Pr($x$) $\leftarrow a$
if $x$ is not root then
    $y \leftarrow$ first left sibling of $x$ or parent if $x$ has no left siblings
    remove $x$ from its parent by setting appropriate pointers
    put $x$ with its subtree into the root list and make it normal
    Fix-violation($y$)
end if
```

The performance of the heap can be analyzed using a potential function defined as $\Phi_T = h + t$, where h is the number of thin nodes in heap and t is number of trees.

Lemma 3.7. *The amortized time bound (with Φ_T as the potential function) of the **Fix-violation** operation in the Thin min-heap is $O(1)$.*

Theorem 3.8. *The amortized time bound (with Φ_T as the potential function) for the Thin min-heap operations* ***Insert****,* ***Meld****,* ***Find-min*** *and* ***Decrease*** *is $O(1)$ and amortized time bound for* ***Delete-min*** *is $O(\log n)$.*

Proof. See [11]. □

We now define a **Delete-L-node** operation and show the amortized time bound for the **Fix-L-node** operation (page 22) of the L-correspondence heap with Thin heap as the base heap. Finally, a probabilistic time bound for the **Decrease** operation of the L-correspondence heap will be given.

Delete-L-node and Increase

Delete-L-node(x)

```
if $x$ is root then
    remove $x$ from root list
    return
end if
$y \leftarrow$ first left sibling of $x$ or parent if $x$ has no left siblings
remove $x$ from its parent by setting appropriate pointers
Fix-violation($y$)
```

The **Delete-L-node** operation uses **Fix-violation** in the same way as **Decrease**. By definition it is allowed to create new leaves and as we will see, in some cases it will.

Increase(x, a)

```
Pr($x$) $\leftarrow a$
if $x$ is not root then
    $y \leftarrow$ first left sibling of $x$ or parent if $x$ has no left siblings
    remove $x$ from its parent by setting appropriate pointers
    put $x$ into root list and make it normal
    Fix-violation($y$)
end if
for all child $z$ of $x$ do
    remove $z$ from its parent $x$ by setting appropriate pointers
    add $z$ into root list and make it normal
end for
```

The **Increase** operation uses **Fix-violation** in the same way as **Decrease**, but to ensure the min-heap ordering property, its children must be removed.

3.2.2 Thin heap in the L-correspondence

We will now analyze the **Fix-L-node** operation (page 22) of the L-correspondence heap with the Thin heap as the base heap. We are interested in the **total change of potential** in both heaps, which in this case is the sum of the number of trees and the number of thin nodes in both heaps.

Fix-L-node

We will show that the **Fix-L-node** operation (page 22) of the L-correspondence heap using **Delete-L-node** of the Thin heap terminates after a constant number of steps that take constant time and increases the potential by a constant amount.

First we must determine whether **Fix-violation** procedure creates new leaves and under what conditions. First we look only at the body of the procedure, then we handle the recursive calls. Suppose there is a violation at node x.

Violation of condition 1 - If node x is thin, no structural change is done. If it is normal, it has its first child removed. It can therefore **become a leaf** if before the operation it had $rank_T$ 1.

Violation of condition 2 - Let z be the right sibling of x. If z is normal, the only structural change is a swap of children in the children list. If z is thin, either z becomes the child of x or vice versa, depending on the priorities of both nodes. No leaves are created.

Violation of condition 3 - No structural change is done.

We conclude that in the body of **Fix-violation**, the only case where a new leaf is created is when x violates condition 1, is normal and $rank_T(x) = 1$. Let a node with such characteristics be called **bad**. Now we analyze whether there is a possibility that a bad node becomes an argument of a **recursive call** of **Fix-violation** (lines 5 and 25 of the algorithm).

First consider the recursive call on line 5. If the argument y of the recursive call is bad, then $rank_T(y) = 1$. Before $rank_T$ adjusting, y was either the first left sibling or the parent of x. In any case, at this time $rank_T(x) \leq 0$. A node with $rank_T$ 0 can not violate any condition, which gives us a contradiction.

Second, the recursive call on line 25. Again, before adjustment of $rank_T$, y was either the first left sibling or the parent of x, so at this time $rank_T(x) \leq 0$, therefore the assumption of the recursive call is incorrect.

It is clear that no recursive call of **Fix-violation** can cause creation of a new leaf.

Lemma 3.9. *The **Fix-violation** of the Thin heap with argument x creates a new leaf if and only if x is normal of $rank_T$ 1 and violates condition 1 of the thin tree.* □

Now we can finally show that the **Fix-L-node** operation (page 22) with **Delete-L-node** in Thin heaps terminates after a constant time.

Consider node x, the argument of the **Fix-L-node** operation. We must determine in which cases a new leaf is created after calling **Delete-L-node** on x.

The only two cases when **Delete-L-node** is called on x is when $rank_T(x) = 1$ or $rank_T(x) = 0$, thus it is thin or normal, respectively. First assume $rank_T(x) = 1$ (Fig. 3.3). The x is deleted, but it has a right sibling of $rank_T$ 0, therefore its possible parent does not become a leaf. Afterwards the **Fix-violation** procedure is called on y, the first left sibling of x or the parent if is has no left siblings. In any case $rank_T(y) \geq 2$. **Fix-violation**(y) does not create new leaves if $rank_T(y) \neq 1$ (Lemma 3.9).

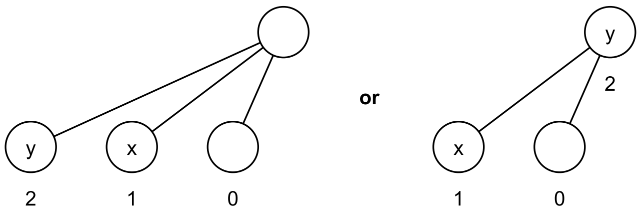

Figure 3.3: x has $rank_T$ 1

The second case is when $rank_T(x) = 0$. Here we must distinguish whether x has a left sibling or not. First if it does not have a left sibling. Then its parent y has either $rank_T$ 1 (Fig. 3.4), in which case after deletion a new leaf y is created of $rank_T$ 1 and **Fix-violation** is not required because there is no violation. We already know that the deletion of a leaf of $rank_T$ 1 does not create new leaves so the **Fix-violation** operation stops after a constant number of steps.

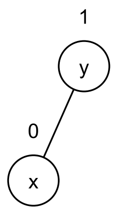

Figure 3.4: x has $rank_T$ 0, no left sibling and its parent y is normal

If $rank_T(y) = 2$, a violation of condition 2 is created at y and **Fix-violation**(y) is called. Let w be the right sibling of y. If w is normal (Fig. 3.5), a swap of y and w in the children list is done, therefore y does not acquire any new children and a call to **Delete-L-node**(y) may be performed (if y has no pointer pair). But new $rank_T$ of y is 1 and deletion of such leaf does not create new leaves.

If w is thin (Fig. 3.6), y may acquire new child if its priority is smaller than that of y. If its priority is greater than that of y, it will become a child of w and remain a leaf of $rank_T$ 0, but its new parent w will have a $rank_T$ 1, and the deletion of such leaf does not lead to the creation of more than one new leaf.

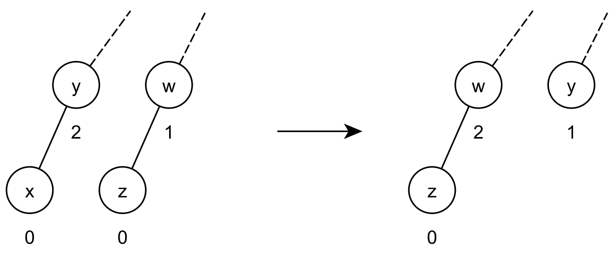

Figure 3.5: x has $rank_T$ 0, no left sibling, its parent y is thin, right sibling w of y is normal

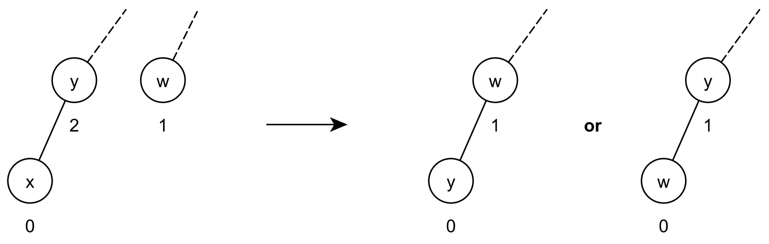

Figure 3.6: x has $rank_T$ 0, no left sibling, its parent y is thin, right sibling w of y is thin

The last case is when x has a left sibling y_s (with $rank_T$ 1). Then after the deletion of x, there is a violation of condition 1 at y_s. Now if y_s was thin (Fig. 3.7), its $rank_T$ is decreased and there is a new violation in z, the first left sibling of y_s or its parent if it has no left siblings. In any case $rank_T(z) \geq 2$. **Fix-violation**(z) does not create new leaves if $rank_T(z) \neq 1$.

If y_s was normal (Fig. 3.8), the **Fix-violation** removes the child w_c of y_s and makes it the right sibling of y_s. This will make a leaf out of y_s and may cause the **Delete-L-node**(y_s). But $rank_T(y_s) = 1$, so this case does not lead to the creation of more than one new leaf.

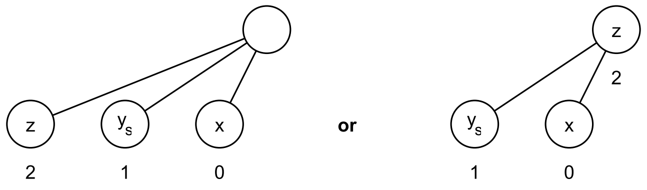

Figure 3.7: x has $rank_T$ 0 and its left sibling y_s is thin

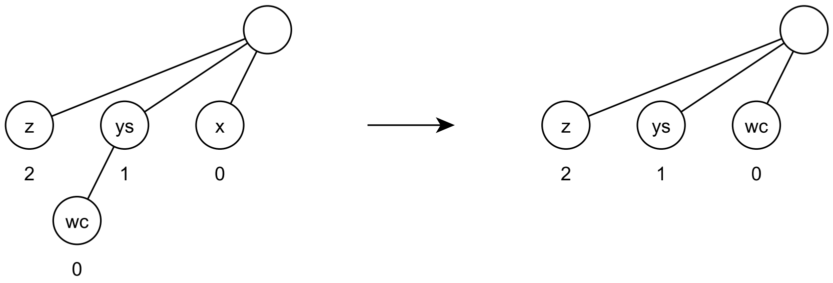

Figure 3.8: x has $rank_T$ 0 and its left sibling y_s is normal

Lemma 3.10. *The amortized time complexity (with Φ_T as the potential function) of the* ***Fix-L-node*** *operation in the L-correspondence heap with the Thin heap as base heap is $O(1)$.*

Proof. The number of **Delete-L-node** and **Fix-violation** calls is constant. The potential $\Phi_T = h + t$ is increased by one for every new tree and decreased for every thin node that becomes normal. The amortized time cost of **Fix-violation** is $O(1)$, therefore we consider only increase of potential due to calls of **Delete-L-node** operation. One call may decrease the number of thin nodes and it may increase the number of trees by one, which concludes the proof. □

Decrease

To analyze the **Decrease** operation (page 24) of the L-correspondence heap, we must give amortized time bound for every sub-routine as in:

$$C^A_{\text{Decrease}}(n) \leq c^A_{\text{Decrease}}(n) + c^A_{\text{Increase}}(n) + 2c^A_{\text{Fix-L-node}}(n)$$

It remains to show the time bound for the min-heap **Increase** operation. Its time cost is dependent on the amount of children and this in the Thin heap is bounded by $O(\log n)$. If we pick a node at random, the expected amortized cost is constant.

Lemma 3.11. *If the argument of* ***Increase*** *operation in the Thin min-heap is a node selected with uniform probability, the expected amortized time bound (with Φ_T as the potential function) is $O(1)$.*

Proof. Let c be the number of children of y, the argument of the operation. The potential may increase by 1 for every removed child and it may decrease if there were any thin nodes among the children of y. The increase in potential is $O(c)$, therefore the amortized time cost of the operation is $O(c)$, because the amortized time cost of **Fix-violation** is $O(1)$.

If we let c be a random variable from a probabilistic space where every node is selected with uniform probability, applying Lemma 2.2, we get the desired bound. □

All results together give us the following:

Theorem 3.12. *The expected amortized cost (with Φ_T as the potential function) of the* ***Decrease*** *operation on a node selected with uniform probability in the L-correspondence heap with the Thin heap as the base heap, is bounded by a constant.* □

The theoretical performance of the L-correspondence heap with the Thin heap as the base heap follows:

Operation	Worst case	Amortized	Expected amortized
Insert	$O(1)$	$O(1)$	$O(1)$
Meld	$O(1)$	$O(1)$	$O(1)$
Find-min and **Find-max**	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(n)$	$O(\log n)$	$O(\log n)$
Decrease and **Increase**	$O(\log n)$	$O(\log n)$	$O(1)$

3.3 Rank-pairing heap

The Rank-pairing heap is a recent result that shows that simplicity of Fibonacci heaps can be kept intact while at the same time, the number of cut operations can be minimized (to at most one cut per decrease operation).

3.3.1 Description

The Rank-pairing heap was introduced in 2011 by Haeupler, Sen and Tarjan[10] and inspired by the Fibonacci heap and the Pairing heap[8]. The operations of the data structure are very simple, but they are not easy to analyze. Similarly as the Fibonacci heap, it achieves $O(1)$ amortized time for **Insert**, **Find-min** and **Decrease** and $O(\log n)$ amortized time for **Delete-min**, but intuitively it has a potential to perform better in practice.

Half trees

A **half-tree** is a binary tree where root has no right child. A **half-ordered** binary tree is a binary tree such that for every node v representing element x, all elements represented by nodes from left subtree of v have larger priority than x. A half-tree is **perfect** if after removing the root, we are left with a full binary tree.

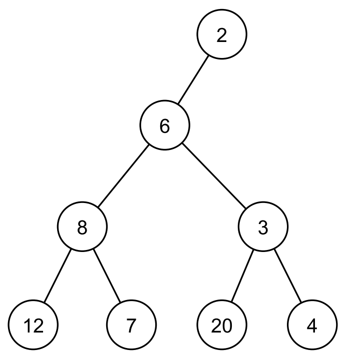

Figure 3.9: A perfect half-tree

The half-trees use three pointers per node, parent pointer and left and right child pointers. It is possible to use two pointers instead of three as a trade of space for additional constant time[8] (Fig. 3.10). The left pointer points on the left child, or the right child if it has no left child. The right pointer points on the right sibling, or the parent if it has no right sibling. The right pointer of the root is always null. To distinguish left and right nodes, every node must keep an additional bit indicating whether it is the left or the right child.

Similarly as with rank in the Fibonacci heap, nodes keep additional information. We will denote this as $rank_{RP}$ because it has a different meaning than rank in the Fibonacci heap. Since we are working with binary trees, $rank_{RP}$ does not describe the number of children, but serves as a lower bound on the size of the node's subtree. Trees have the $rank_{RP}$ of its root. Singleton trees have

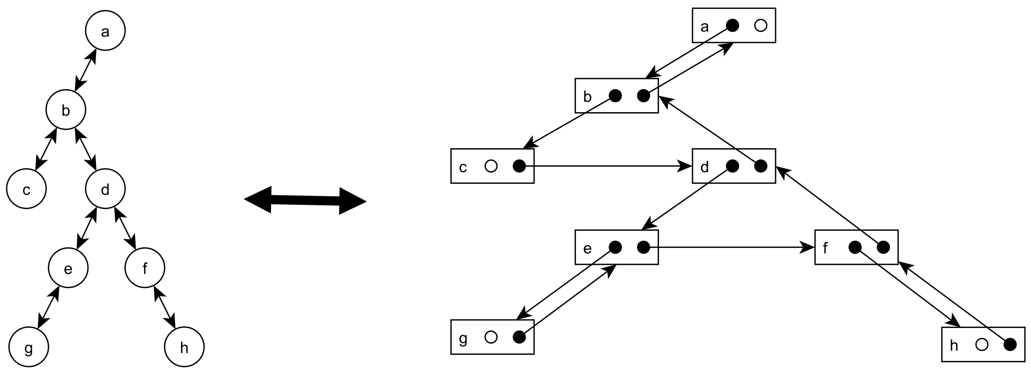

Figure 3.10: Binary tree representation using two pointers instead of three.

$rank_{RP}$ 0. A linking operation (Fig. 3.12) is defined such that only trees with equal $rank_{RP}$ can be linked and the $rank_{RP}$ of the new tree has $rank_{RP}$ equal to previous $rank_{RP}$ incremented by one.

The half-ordered half-trees are **fully equivalent** with the heap-ordered trees[12] and this will be used in our latter analysis. The reason why the Rank-pairing heap is analyzed as a half-ordered half-trees instead of considering its heap-ordered representation is that "[half-ordered half-tree representation] is closer to an actual implementation, and it unifies the treatment of key decrease"[10].

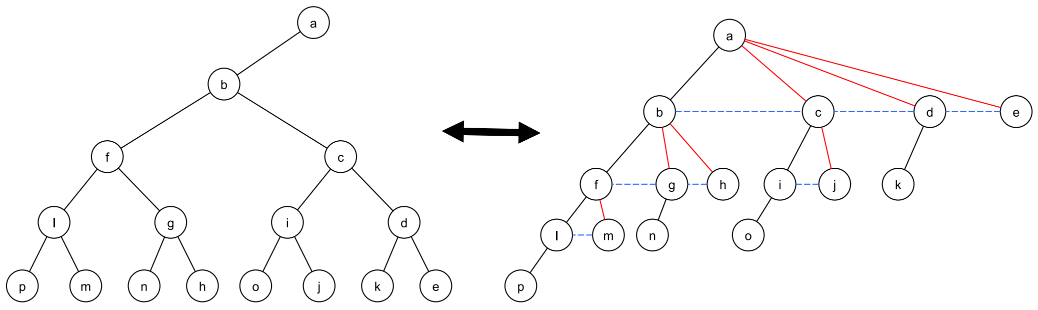

Figure 3.11: Equivalence of half-ordered half-tree to heap ordered tree

Equivalence of a perfect half-ordered half-tree with a heap-ordered (binomial) tree is shown on Fig. 3.11. Blue dashed lines represent the right child pointer that is replaced by right sibling pointer. Red lines represent (typically virtual) parent-children pointers. Note that ordering is equivalent too: for a node in heap-ordered representation, its descendants are exactly nodes from its left subtree in the half-tree representation.

Algorithms for the operations will be described and later analyzed from the view of the half-ordered half-trees for the simplicity of analysis. A single exception will be the analysis of the **Increase** operation for which the heap-ordered tree representation is preferred.

Rank-pairing heaps

For a non-root node v, let the **rank difference** of v be $rank_{RP}(\mathbf{Parent}(v))$ - $rank_{RP}(v)$. Node whose children have rank differences *i* and *j* is called an *i,j-node*. A missing child is considered having $rank_{RP}$ -1. In the original paper, two types of Rank-pairing heaps were defined. The type-1 Rank-pairing heap performs better than the type-2 Rank-pairing heap, but is more difficult to analyze. We will extend the type-1 Rank-pairing heap and for the rest of the work we denote it Rank-pairing heap.

A collection of (not necessarily perfect) half-ordered half-trees is a Rank-pairing heap if it satisfies the following constraints on $rank_{RP}$s: every non-root node is a *1,1-node* or a *0,i-node* for some $i > 0$. For a root node, the rank difference of its (left) child is 1 and it is denoted *1-node*.

The definition of the Rank-pairing heap leads to the following restrictions on the node $rank_{RP}$. Let x be a node of a Rank-pairing heap. Then for non-root x

$$rank_{RP}(x) = \begin{cases} 0 & \text{if } x \text{ is a leaf} \\ rank_{RP}(y) & \text{if } y \text{ is a child with the larger rank} \\ rank_{RP}(y) + 1 & \text{if both children } y \text{ and } z \text{ have the same rank} \end{cases}$$

The first case holds because the missing children of x have $rank_{RP}$ -1 by definition, therefore x is a *1,1-node* and its $rank_{RP}$ is larger by one than the $rank_{RP}$ of its children. The second and third cases correspond to the case when x is either a *0,r-node* for some $r > 0$ or a *1,1-node*, respectively.

The half-trees are stored in a circular list and there is a special minimum pointer that always points on a root with minimal priority. This list is called a root list.

Lemma 3.13. *In the Rank-pairing heap, a non-root node of $rank_{RP}$ k has at least $2^{k+1} - 1$ descendants (including itself), therefore any node of $rank_{RP}$ k has at least 2^k descendants (including itself).*

Proof. First part of the lemma implies the second part since in Rank-pairing heaps the rank difference of the left child of the root is 1. The first part is proved by induction on a height of a node. First a leaf has $rank_{RP}$ 0 and $1 = 2^1 - 1$ descendants. Let the statement be true for all nodes with height smaller than k. Let v be a node of $rank_{RP}$ k. If v is a *1,1-node*, then number of its descendants is at least $2(2^k - 1) + 1 = 2^{k+1} - 1$. If v is a *0,i-node* for some $i > 0$, then one of its children has $rank_{RP}$ k. By the induction hypothesis, it has at least $2^{k+1} \geq 2^{k+1} - 1$ descendants. □

Eligibility for the L-correspondence

For the Rank-pairing heap we define that L-property for a node x is:

$$x \text{ is an L-node } \equiv x \text{ has no left child}$$

The condition for the relaxed min-heap (page 19) holds, because for any node x there must be a node y in the same tree with no left child such that $\mathbf{Pr}(x) \leq \mathbf{Pr}(y)$.

This is very natural, because a node in a half-ordered half-tree has no left child if and only if in the min-heap representation this node is a leaf.

From the definition of the operations it will be clear that all L-correspondence conditions are satisfied. Notably the condition of instant retrieval of the $(List, w)$ pair after the **Delete-L-node**, **Decrease** and **Increase** operations is easily satisfied, but its implementation is **not included** in these operations for clarity.

Operations

The algorithms for the Rank-pairing heap operations follow. Every time a root list is changed (a root is added or it is concatenated with another root list), it is ensured in constant time that a root with minimal priority is referenced by a minimum pointer.

Insert(x)

```
  create a new node $v$ representing element $x$ and add $v$ into the root list
  $rank_{RP}(v) \leftarrow 0$
```

Meld(Q_1, Q_2)

```
  concatenate the root lists of $Q_1$ and $Q_2$
```

Find-min

```
  $x \leftarrow$ root referenced by the minimum pointer
  return a reference to $x$
```

Link(x, y) - link two trees of equal $rank_{RP}$ rooted in x and y (Fig. 3.12)

```
  if Pr($x$) > Pr($y$) then
      swap $x$ and $y$
  end if
  $z \leftarrow$ Left($x$)
  remove the subtree rooted in $z$ from $x$
  Right($y$) $\leftarrow z$
  Left($x$) $\leftarrow y$
  $rank_{RP}(x) \leftarrow rank_{RP}(x) + 1$
  return $x$
```

The **Delete-min** operation is defined in a different way than a typical multi-pass linking of the Fibonacci (or Thin) heap. First, let all trees before the operation be *old* trees. After the operation, let trees from the disassembly be *new* trees. Now the linking is performed in the following way: until there is a pair of *new* trees with the same $rank_{RP}$, link them together, let the resulting tree be *old* and put it into the root list with the rest of the *old* trees. When there is no such pair left, let all trees be *old* and perform a standard multi-pass linking.

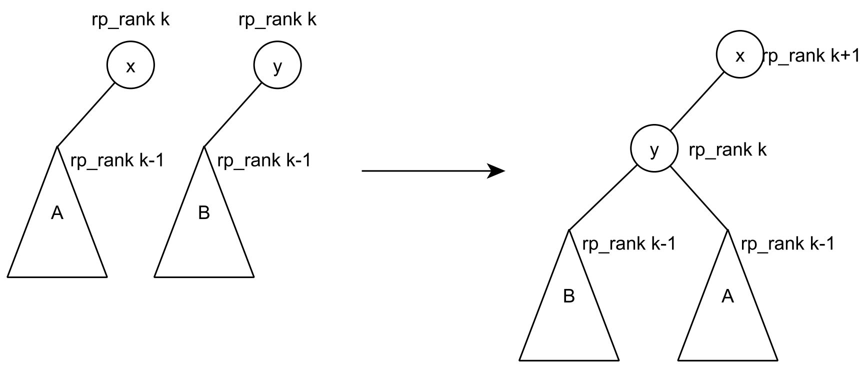

Figure 3.12: Link of half-ordered half-trees of equal $rank_{RP}$

Delete-min

```
(∀i ≤ maximum rank from the trees of the root list) $F_i \leftarrow \emptyset$
$x \leftarrow$ root referenced by the minimum pointer
remove $x$ from the root list
$y \leftarrow$ Left($x$)
while $y \neq null$ do                                    ▷ Disassemble the tree
    Parent($y$) $\leftarrow null$
    $z \leftarrow$ Right($y$)
    Right($y$) $\leftarrow null$
    if Left($y$) $= null$ then
        $rank_{RP}(y) \leftarrow 0$
    else
        $rank_{RP}(y) \leftarrow rank_{RP}$(Left($y$)) $+ 1$
    end if
    $r \leftarrow rank_{RP}(y)$
    if $F_r = \emptyset$ then
        $F_r \leftarrow \{y\}$
    else if $F_r = \{w\}$ then
        add Link($y$, $w$) into the root list                ▷ One-pass linking
        $F_r \leftarrow \emptyset$
    end if
    $y \leftarrow z$
end while
add all remaining trees from all $F_i$ into the root list
∀$i$ let $T_i$ be a set of all trees of $rank_{RP}$ $i$ from the root list
clear the root list
$i \leftarrow 0$
while $\exists i\, |T_i| \geq 1$ do                  ▷ Proceed with multi-pass linking
    while $|T_i| > 1$ do
        remove any two trees $t_1$ and $t_2$ from $T_i$
        $x \leftarrow$ Link($t_1$, $t_2$)
        add $x$ to $T_{i+1}$
    end while
    if $|T_i| = 1$ then
        remove last tree from $T_i$ rooted in $x$
        add $x$ into the root list
    end if
    $i \leftarrow i + 1$
end while
```

To delete the minimum, take the first tree from the root list, remove its root x and disassemble the tree (Fig. 3.13). Start with the left child y of x and after removing its right child either put y into F_i, where $i = rank_{RP}(y)$, or if F_i was not empty, link y with the tree of F_i and put the result into the root list (as an *old*) tree. Proceed by setting the former right child of y as the new y and repeat until there is no right child.

Note that the two-phase linking process is required for the analysis of the change of potential and the authors admit that it is not known whether "[our] analysis can be extended to arbitrary multipass linking".

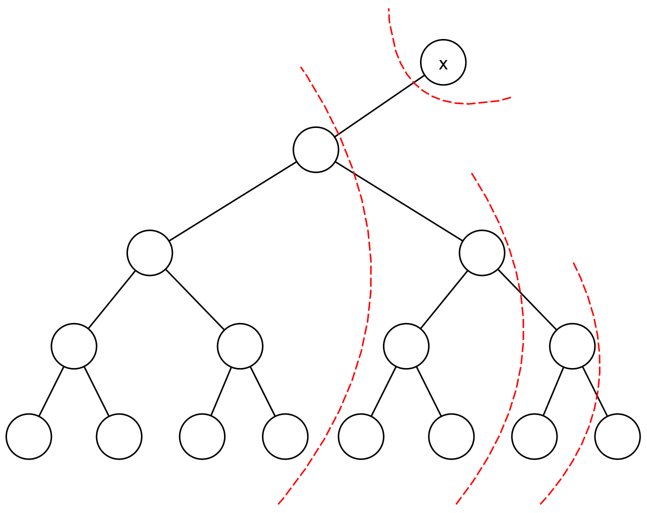

Figure 3.13: Disassembly of a perfect half-tree

Fix-rank(x)

```
if $x$ is a root then
    $rank_{RP}(x) \leftarrow rank_{RP}(\textbf{Left}(x)) + 1$
else
    if $rank_{RP}(\textbf{Left}(x)) = rank_{RP}(\textbf{Right}(x))$ then
        $k \leftarrow rank_{RP}(\textbf{Left}(x)) + 1$
    else
        $k \leftarrow \textbf{max}(rank_{RP}(\textbf{Left}(x)), rank_{RP}(\textbf{Right}(x)))$
    end if
    if $rank_{RP}(x) = k$ then
        return                                        ▷ no $rank_{RP}$ update needed
    else
        $rank_{RP}(x) \leftarrow k$
        Fix-rank(Parent$(x)$)
    end if
end if
```

The **Fix-rank** procedure is called to ensure that the $rank_{RP}$ invariants of the Rank-pairing heap are satisfied.

Decrease(x, a) - decrease a priority of an element (Fig. 3.14)

```
Pr($x$) ← $a$
if $x$ is a root then
    put $x$ in front if Pr($x$) is the new minimum
else
    $y$ ← Right($x$)
    cut $x$ and $y$ from the tree with their subtrees
    attach $y$ in place of $x$
    add $x$ into the root list
    if Left($x$) = $null$ then
        $rank_{RP}(x) \leftarrow 0$
    else
        $rank_{RP}(x) \leftarrow rank_{RP}(\textbf{Left}(x)) + 1$
    end if
    Fix-rank(Parent($y$))
end if
```

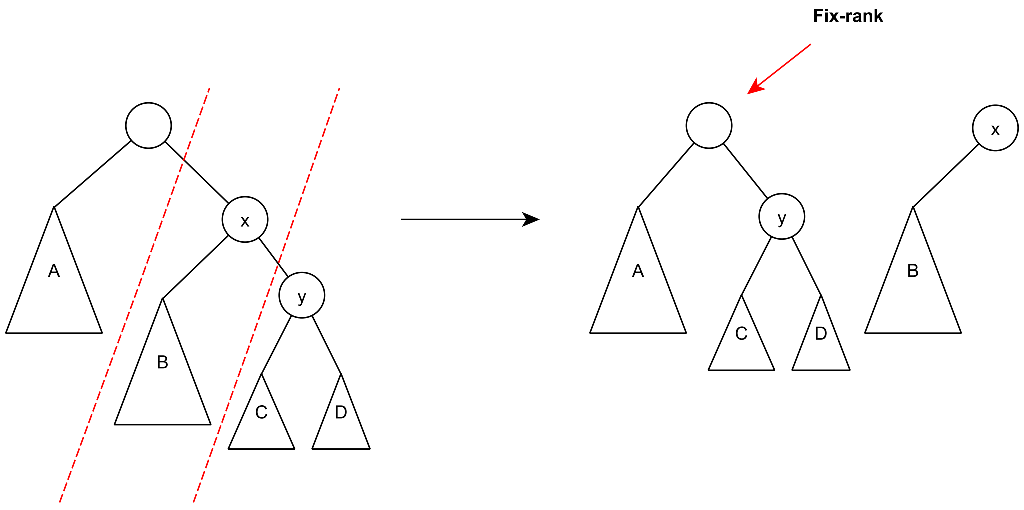

Figure 3.14: Decrease operation

The **Decrease** procedure makes one cut at most. After the cut, the former parent of x is a node with a new child y. The former parent and x are the only nodes where $rank_{RP}$ invariants may have been breached and must be fixed.

Delete-L-node and Increase

The **Delete-L-node** operation must be designed in a way that the L-correspondence conditions are met. It may create a set on new singleton trees during its operation and one arbitrary L-node. In the Rank-pairing heap, L-node is a node with no left child.

For the simplification of our analysis we will design **Delete-L-node** specifically for the L-correspondence heap. This will allow us to check whether a node has a pointer pair. Let's assume that **Fix-L-node** of the L-correspondence heap is the only operation that calls **Delete-L-node**. From the definition of **Fix-L-node** (page 22), the operation stops when **Delete-L-node** does not create a new L-node with no pointer pair (it may create new singleton trees however).

The main reason for this is the operation **Fix-rank**. This is required after deletion to ensure that the conditions on $rank_{RP}$ are satisfied. We cannot afford to call this procedure every time a node is deleted, therefore we call it only on the parent of the topmost deleted node, i.e. when no more **Delete-L-node** operations are going to be called. To see when this happens, we need to check whether a node has a pointer pair.

```
Delete-L-node($x$)
  if $x$ is a root then
      remove $x$ from the heap
      return
  end if
  $y \leftarrow$ Parent($x$)
  $z \leftarrow$ Right($x$)
  remove $x$ from its parent $y$ by setting appropriate pointers
  if $x$ was the right child of $y$ before the operation then
      Right($y$) $\leftarrow z$
  else if $x$ was the left child of $y$ before the operation then
      Left($y$) $\leftarrow z$
  end if
  if Pointer($y$) $\neq null$ then
      Fix-rank($y$)
  end if
```

In the **Delete-L-node** we remove node x from its parent y. If x was the right child of y before the operation, we put z, the former right child of x, in the place of x.

If x was the left child of y before the operation, again we put z, the former right child of x, in the place of x. Note that if z is *null* (x had no right child before the operation), a new L-node is created (at y).

```
Increase($x$, $a$)
    Pr($x$) $\leftarrow a$
    if $x$ is not root then
        $y \leftarrow$ Right($x$)
        cut $x$ and $y$ from the tree with their subtrees
        attach $y$ in place of $x$
        Fix-rank(Parent($y$))
    end if
    $z \leftarrow$ Left($x$)
    remove $z$ from $x$
    add $x$ into the root list
    $rank_{RP}(x) \leftarrow 0$
    while $z \neq null$ do                                  ▷ Disassemble the tree
        Parent($z$) $\leftarrow null$
        $w \leftarrow$ Right($z$)
        Right($z$) $\leftarrow null$
        if Left($z$) $= null$ then
            $rank_{RP}(z) \leftarrow 0$
        else
            $rank_{RP}(z) \leftarrow rank_{RP}$(Left($z$)) $+ 1$
        end if
        add $z$ into the root list
        $z \leftarrow w$
    end while
```

The **Increase** starts with a cut of x. If x is not a root, the former parent of x gets the right child of x instead of x. This is same as in **Decrease**. The subtree of x, stripped of its right child subtree, may not satisfy the half-ordering property, thus must be disassembled as in **Delete-min**.

3.3.2 Rank-pairing heap in the L-correspondence

We will show that the **Fix-L-node** operation (page 22) of the L-correspondence heap using **Delete-L-node** of the Rank-pairing heap terminates after a constant number of steps that take constant time and increases the potential by a constant amount.

Fix-L-node

We will show that the potential function of the original analysis can be reused to give good amortized time bounds on **Delete-L-node** and **Increase**. The original potential function is described in the following way:

Every node is either **good** or **bad**. A node is good if it is one of the following:

- singleton tree
- leaf
- root with a *1,1-node* left child
- *1,1-node* with both children that are *1,1-node*s
- *0,1-node* but its 0-child is a *1,1-node*

The potential of a node is defined as the **sum of the rank differences of its children** plus a constant depending on whether it is good or bad and whether it is a root:

- root that is good: +1
- root that is bad: +5
- non-root that is good: -1
- non-root that is bad: +3

Note that the missing right child of a root is not considered as a node of $rank_{RP}$ -1, therefore the $rank_{RP}$ of a root is defined as the **rank difference of its left child** plus a constant specified above. The potential Φ_{RP} of the heap is the sum of its node potentials. See [10] for the following results:

Theorem 3.14. *The amortized time bound (with Φ_{RP} as the potential function) for operations **Find-min**, **Meld**, **Insert** and **Decrease** is $O(1)$ and for operation **Delete-min** is $O(\log n)$ in rank pairing heaps.* □

Lemma 3.15. *The amortized time bound (with Φ_{RP} as the potential function) for the **Fix-rank** operation in rank pairing heaps is $O(1)$.* □

Lemma 3.16. *The amortized time bound (with Φ_{RP} as the potential function) for the **Fix-L-node** operation in the L-correspondence heap with Rank-pairing heap as the base heap is $O(1)$.*

Proof. Let x be the argument of the first **Delete-L-node** operation. **Delete-L-node** is called again by **Fix-L-node** if x has a parent and **Parent**(x) has no left child after the operation and no pointer pair. This is applied iteratively until we reach the topmost node that is either not an L-node after the operation or has a pointer pair or is a singleton tree.

Let k be the number of calls of the **Delete-L-node** operation. Then **Fix-L-node** takes $O(k)$ time. There are three possible cases for **Fix-L-node** to finish.

First assume that the **Fix-L-node** operation stops because it reaches the root (Fig. 3.15). Let x be the first deleted L-node, and y be its parent. All nodes on the path from x (including x) to the root have no right children (**Fix-L-node** would have terminated here otherwise), therefore they have $rank_{RP}$ 0. The root has $rank_{RP}$ 1. The only two good nodes on the path are x and y, all others

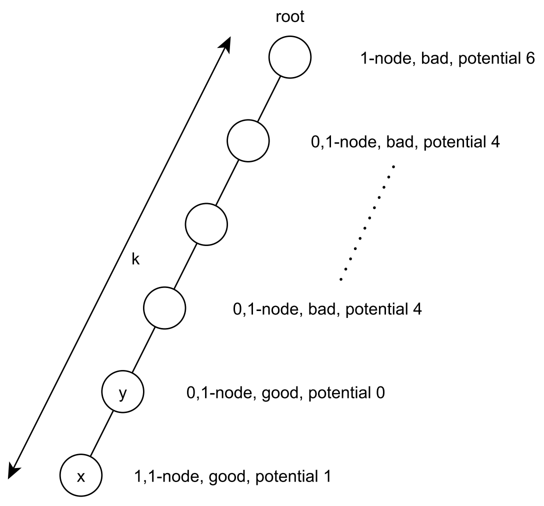
root
1-node, bad, potential 6
0,1-node, bad, potential 4
k
0,1-node, bad, potential 4
y
0,1-node, good, potential 0
x
1,1-node, good, potential 1

Figure 3.15: **Fix-L-node** stops at the root.

including the root are bad. The potential of good non-root *i,j-node*s is $i + j - 1$ and it is $i + j + 3$ for bad non-root nodes. The root is a *1-node* and its potential is defined as the sum of the rank difference of its left (and only) child plus a constant depending on whether it is good or bad. The root is bad, therefore its potential is $1 + 5 = 6$.

All nodes on the path will be deleted and reinserted as singleton trees. The root of a singleton tree has $rank_{RP}$ 0 and is a *1-node*. It is a good node, hence its potential is $1 + 1 = 2$. The change in potential is therefore:

$$\begin{aligned}\Delta\Phi_{RP} &= 2k - [1 + 0 + 4(k-3) + 6] \\ &= -2k + 5\end{aligned}$$

The second case is that the **Fix-L-node** operation stops because it deletes a node that is the right child of its parent, thus does not create a new L-node (Fig. 3.16).

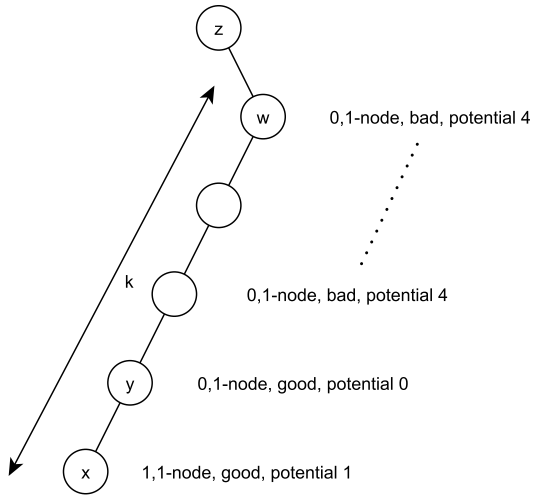
z
w
0,1-node, bad, potential 4
k
0,1-node, bad, potential 4
y
0,1-node, good, potential 0
x
1,1-node, good, potential 1

Figure 3.16: **Fix-L-node** stops at a right child.

This case is almost identical to the previous case except that the last deleted (and reinserted) node is not the root, but w, the first node on the path that is the right child of its parent. Node z will not be deleted, but its child will and this will change the potential of z. Assume node z was *i,j-node* before the operation. After its child is deleted, it becomes a *i,(j+1)-node* and its potential increases by one. Then the change in potential is:

$$\begin{aligned}\Delta\Phi_{RP} &= 2k - [1 + 0 + 4(k-2)] + 1\\ &= -2k + 8\end{aligned}$$

The last possible case is when the **Fix-L-node** operation stops because the last deleted node had a right child and this comes in place of the deleted node (Fig. 3.17).

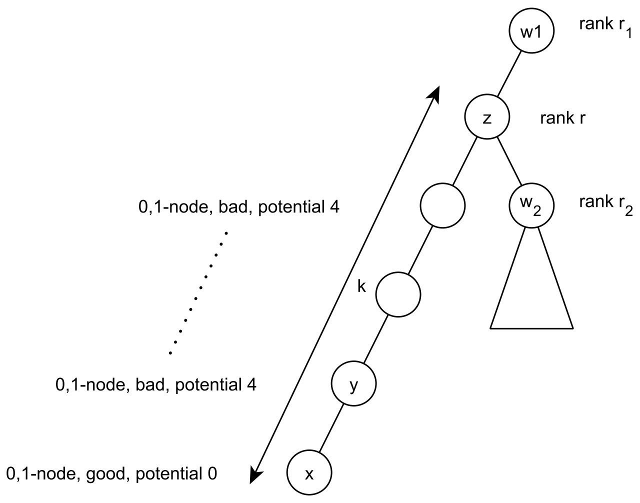
w1
rank r$_1$
z
rank r
w$_2$
rank r$_2$
0,1-node, bad, potential 4
k
0,1-node, bad, potential 4
y
0,1-node, good, potential 0
x

Figure 3.17: **Fix-L-node** stops at a node with a right child.

Let z be the last deleted node, w_1 be its parent, w_2 be its right child. The situation is similar to the previous case, but we must consider the change in the potential of z and w_1. Let the $rank_{RP}$ of z be r and the $rank_{RP}$ of w_2 be r_2. If z is a *1,1-node*, $r_2 = 0$ and $r = 1$. If z is a *0,i-node*, $r_2 = r > 0$.

The new left child of w_1 will be w_2. The potential of w_1 either does not change, or changes by $O(1)$, depending on whether w_1 is good or bad and whether w_2 is good or bad.

Node z is a bad *i,j-node*, so its potential is $i + j + 3$. It is either a *1,1-node* or a *0,i-node* for $i > 0$. Therefore $\Phi_{RP}(z) \geq 4$. The change in potential is:

$$\begin{aligned}\Delta\Phi_{RP} &\leq 2k - [1 + 0 + 4(k-2)] + O(1)\\ &= -2k + O(1)\end{aligned}$$

In any of the three cases, the decrease in the potential pays for the operation. The last step of the operation is **Fix-rank(p)**, where p is the parent of the last deleted node, in case the last deleted node was not a root. From Lemma 3.15, we know that **Fix-rank** has $O(1)$ amortized time bound which completes the proof. □

Decrease

To analyze the **Decrease** operation (page 24) of the L-correspondence heap, we must give amortized time bound for every sub-routine as in:

$$C^A_{\text{Decrease}}(n) \leq c^A_{\text{Decrease}}(n) + c^A_{\text{Increase}}(n) + 2c^A_{\text{Fix-L-node}}(n)$$

It remains to show the time bound for the min-heap **Increase** operation. Its time cost is dependent on the dissasembly. Let the **right path** of x be all nodes on the path from x to a node with no right child, following only the right children. Let y be the argument of the **Increase** operation and let x be its left child. We will show that the amortized time bound of **Increase** is $O(1 +$ length of the right path of $x)$ and that the expected length of the right path of x is $O(1)$. Note that in the Rank-pairing heaps, the length of the right path can be $\Omega(n)$.

Lemma 3.17. *If the argument of **Increase** operation in the Rank-pairing min-heap is a node selected with uniform probability, the expected amortized time bound (with Φ_{RP} as the potential function) is $O(1)$.*

Proof. The **Increase** operation can be split naturally into two parts. First if the node with increased priority is not a root, it is cut out and its left child is cut out and put in the former place of x. Then the **Fix-rank** procedure is performed on the former parent of x. It takes constant time to cut out x and there is only a constant change in potential: Let z be the parent of x and y be the right child of x. Since the potential of a *i,j-node* node is defined as $i + j + O(1)$, the only change in potential is with respect to the rank difference of z and x and the rank difference of x and y. The change of potential in z is

$$rank_{RP}(z) - rank_{RP}(y) - (rank_{RP}(z) - rank_{RP}(x)) + O(1)$$

The change of potential in x is

$$-(rank_{RP}(x) - rank_{RP}(y)) + O(1)$$

The total change of potential is then $O(1)$.

From Lemma 3.15, the **Fix-rank** takes $O(1)$ amortized time which concludes the time bound for the first part of the algorithm.

In the second part, a dissasembly of the tree is performed. During the disassembly, k new trees are created and added into the root list, which amounts for $O(k)$ work and an increase in potential. Every node of the k nodes on the disassembly path becomes a root. A root has a potential of 2 or 6 depending on whether it is good or bad, so the increase in potential is at most $6k$. The amortized time cost is therefore $O(k)$.

The number k of new trees is equal to the length of the right path of the left child of x. Since the half-ordered half-tree is equivalent to heap-ordered tree, the length of the right path is simply the number of children of x. From Lemma 2.2, if we let k be a random variable representing the number of children of a node picked with uniform probability, $E[k] = O(1)$ which concludes the proof. □

All results together give us the following:

Theorem 3.18. *The expected amortized cost (with Φ_{RP} as the potential function) of the **Decrease** operation on a node selected with uniform probability in the L-correspondence heap with the Rank-pairing heap as the base heap, is bounded by a constant.* □

The theoretical performance of the L-correspondence heap with the Rank-pairing heap as the base heap follows:

Operation	Worst case	Amortized	Expected amortized
Insert	$O(1)$	$O(1)$	$O(1)$
Meld	$O(1)$	$O(1)$	$O(1)$
Find-min and **Find-max**	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(n)$	$O(\log n)$	$O(\log n)$
Decrease and **Increase**	$O(n)$	$O(n)$	$O(1)$

Epilogue

We have shown that it is possible to generalize on the leaf correspondence heap to enable a larger class of heaps to serve as the base heap. We have shown how the **Delete-min**, **Decrease** and **Increase** operations can take advantage of a fast **Delete-L-node** operation.

Specific examples were given on how to extend and analyze some well known heaps. For these heaps, expected amortized time bound has been given for the operation **Increase** which led to expected amortized time bound of the **Decrease** operation in the L-correspondence heap. The expected time bound assumed uniform probability distribution when selecting a node as an argument of the operation. This assumption is very simplified, and it may be much more valuable to give an expected time bound on an arbitrary probabilistic distribution. Intuitively, if there are nodes that are selected with higher probability as an argument for the **Increase** operation, then this should not lower the performance. The reason is that once **Increase** is performed, the node loses all its children and the next **Increase** operation can be expected to be cheaper than the previous one. It seems that proving this conjecture is not going be as trivial as for the uniform probabilistic distribution.

Experiments in the previous work by Chong and Sahni on the leaf correspondence have shown that basic operations are faster in the leaf correspondence heaps than in the total correspondence heaps in practice. Intuitively, the L-correspondence heap should perform at least as well as the leaf correspondence heap. Since this approach admits a larger class of base heaps, some very effective heaps may be utilized.

Bibliography

[1] A. Arvind and C. Pandu Rangan, *Symmetric min-max heap: A simpler data structure for double-ended priority queue*, Inf. Process. Lett. (1999), 197–199.

[2] Gerth Stølting Brodal, *Fast meldable priority queues*, Algorithms and data structures (Kingston, ON, 1995), Lecture Notes in Comput. Sci., vol. 955, Springer, Berlin, 1995, pp. 282–290. MR 1465221

[3] S. C. Chang and M. W. Du, *Diamond deque: a simple data structure for priority deques*, Inf. Process. Lett. **46** (1993), no. 5, 231–237.

[4] Kyun-Rak Chong and Sartaj Sahni, *Correspondence-based data structures for double-ended priority queues*, ACM J. Exp. Algorithmics **5** (2000), 20 pp. (electronic). MR 1794935

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, third ed., MIT Press, Cambridge, MA, 2009. MR 2572804 (2010j:68001)

[6] Clark Allan Crane, *Linear lists and priority queues as balanced binary trees*, Ph.D. thesis, Stanford, CA, USA, 1972, AAI7220697.

[7] Yuzheng Ding and Mark Allen Weiss, *On the complexity of building an interval heap*, Inf. Process. Lett. **50** (1994), no. 3, 143–144.

[8] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan, *The pairing heap: a new form of self-adjusting heap*, Algorithmica **1** (1986), no. 1, 111–129. MR 833121 (87e:68011)

[9] Michael L. Fredman and Robert Endre Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach. **34** (1987), no. 3, 596–615. MR 904195 (90d:68012)

[10] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan, *Rank-pairing heaps*, SIAM J. Comput. **40** (2011), no. 6, 1463–1485. MR 2863183

[11] Haim Kaplan and Robert Endre Tarjan, *Thin heaps, thick heaps*, ACM Trans. Algorithms **4** (2008), no. 1, Art. 3, 14. MR 2398583 (2009e:68016)

[12] Donald E. Knuth, *The art of computer programming. Volume 1*, second ed., Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975, Fundamental algorithms, Addison-Wesley Series in Computer Science and Information Processing. MR 0378456 (51 #14624)

[13] Dinesh P. Mehta and Sartaj Sahni (eds.), *Handbook of data structures and applications*, Chapman & Hall/CRC Computer and Information Science Series, Chapman & Hall/CRC, Boca Raton, FL, 2005. MR 2122148 (2005m:68003)

[14] A. Chow M.W. Du, S.C. Chang, *Incremental sorting and selective sorting - two techniques to reduce response time in information retrieval*, Tech. Memo., GTE Laboratories, Waltham, MA.

[15] S. Olariu, C. M. Overstreet, and Z. Wen, *A mergeable double-ended priority queue*, Comput. J. **34** (1991), no. 5, 423–427.

[16] John T. Stasko and Jeffrey Scott Vitter, *Pairing heaps: experiments and analysis*, Comm. ACM **30** (1987), no. 3, 234–249. MR 886126

[17] Jean Vuillemin, *A data structure for manipulating priority queues*, Comm. ACM **21** (1978), no. 4, 309–315. MR 0478740 (57 #18215)

[18] J. Williams, *Algorithm 232*, Communications of the ACM **7** (1964), no. 1, 347–348.

[19] D. Wood, Vakgreep Informatica, Budapestlean Cd H, J. Van Leeuwen, and J. Van Leeuwen, *Interval heaps*, The Computer Journal **36** (1987), 209–216.