Binomial and Fibonacci Heaps in Racket (rkt-heaps)

Abhinav Jauhri
abhinavjauhri@gmail.com

Abstract. Library[1] providing data structures viz. Binomial heap [6] and Fibonacci heap[4] are described with primitive operations, performance and usage.

1 Introduction

$rkt-heaps$ was made for demonstration in a contest, *Lisp In Summer Projects*[2]. Any participant was expected to develop one or many applications, or libraries, using any LISP-based technology. $rkt-heaps$ uses Racket, a dialect of the Lisp language and based on the scheme branch of the Lisp family [1].

Section 1 & 2 describe Binomial and Fibonacci heaps respectively. *Section 3* has performance results to ascertain $rkt-heaps$ correspond to run-time bounds as shown in literature. Comparison with an existing library for Binary heaps[2] in Racket is added to observe any runtime improvements in practice for operations which have similar run-time bounds in theory. *Section 4* has conclusions and author's learnings. *Appendix* includes syntactics of using this library in Racket.

2 Binomial Heaps

Binomial heaps are a collection of heap-ordered Binomial trees with a pointer *min* to the root of a tree having the minimum value amongst all elements in the heap. Primitive operations described in [5] are as follows:

1. *makeheap(i)* Makes a new heap with one element i
2. *findmin(h)* Returns the minimum value in the heap h
3. *insert(h, i)* Adds element i to heap h
4. *deletemin(h)* Deletes the element with minimum value from heap h
5. *meld(h, h')* Combines two heaps h and h' into one

Before studying costs and implementation of the above mentioned operations, explanations for certain technical terms may be helpful for the reader.

[1] A software package and will be referred as $rkt-heaps$ throughout the paper
[2] http://lispinsummerprojects.org/

Operation	Amortized Cost
makeheap	$\mathrm{O}(1)$
findmin	$\mathrm{O}(1)$
insert	$\mathrm{O}(1)$
deletemin	$\mathrm{O}(\log n)$
meld (eager)	$\mathrm{O}(\log n)$
meld (lazy)	$\mathrm{O}(1)$

Table 1: Amortized costs for Binomial Heaps

1. *Amortized cost* - Amortization analysis includes cost of a sequence of operations spread over the entire sequence[5]. Amortized costs may not always give the exact cost of an operation. An operation may be expensive or cheaper, but the average cost over a sequence of operations is small and defined as amortized cost.
2. *rank* - States the number of children of an element. For example, a Binomial tree, the tree size is 2^k with root at some element e having rank k.

Making a new heap with one element - *makeheap*, and using the *min* pointer to get the minimum value - *findmin*, both require constant number of steps.

Insertion of a value is equivalent to making a heap, h_1, with a single element and then melding h_1 with an existing heap, h_2. At each step of a meld, two trees having the same rank are considered. It could be the case that either h_1 or h_2 have a tree of size 2^k (rank k), or neither have one. When two trees are combined, the resulting tree is of rank $(k+1)$. If a tree with rank $(k+1)$ already exists in either of the heaps - h_1 or h_2, then a tree of rank $(k+2)$ is made. This happens continuously till there is single tree of some rank $(> k)$ in the resulting heap, $h_{1,2}$. At the end of meld, $h_{1,2}$ has only one or no tree of size 2^i, such that $0 \leq i \leq \log_2 s$, where $size(h_{1,2}) = s$ The heap property[3] is imposed each time roots of two trees are combined. For intuition on why *insert* is $\mathrm{O}(1)$ amortized cost consider a Binomial heap having only one tree with rank 2. There will be no requirement to combine another tree of rank 2 until 3 elements have been inserted in trees with sizes 2^0 and 2^1.

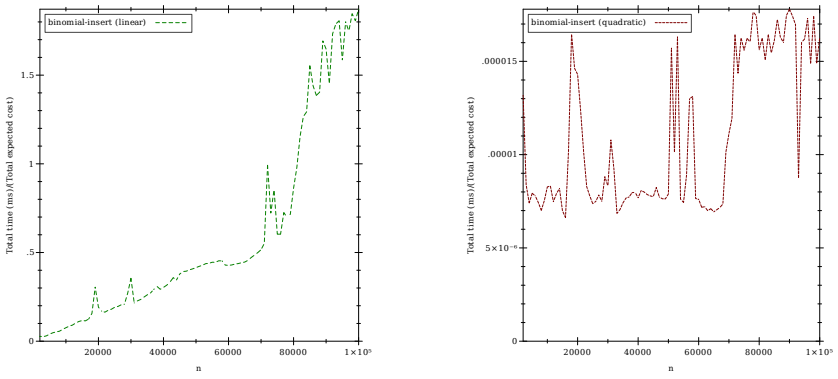
The technique of meld can be extended for *deletemin* operation also; take all children of the min element and meld them with the remaining trees in the heap, and update the min pointer, costing $\mathrm{O}(\log n)$. For a more detailed amortized analysis, refer [5].

meld eager version works in the manner as described above but for the lazy version, all roots of one heap are added to the set of roots of the other heap. This may result in more than one binomial tree of rank k in the resulting heap.

[3] Heap property ensures that all children of a tree rotted at r have their values ordered with its parent's value and that the same ordering applies across the heap

Any subsequent call to a *deletemin* operation will correct the heap such that only one binomial tree of rank k exists. This costs $\mathrm{O}(\log n)$.

Binomial heap in $rkt - heaps$ is an array-based purely functional implementation leveraging Racket's *vector* library. Purely functional ensures there are no mutations of previously created data structures. Although, this leads to anomalies in runtime costs for certain operations like *insert*. With reference to Table 1, the total cost for n inserts is linear - $\mathrm{O}(n)$; but in a purely functional implementation, the insert operation cost for n inserts aggregates to $\mathrm{O}(n^2)$. The reason for such an anomaly is due to cloning of heap vector at every insert operation. This is validated in Fig:1. The curve *binomial-insert (quadratic)* signifies the divisor(total expected cost) for the dividend(total time) is quadratic, and hence the slope of the curve does not linearly increase with n. Since, it is amortized cost the curve will have high and low peaks instead of slope being almost zero[4]. Alternatively, taking the divisor as linear, the total run-time cost depicts a curve with slope due to increase cost of cloning the vector, which is to say that nth insertion step clones the $(n-1)th$ heap vector[5]. Modifying the structuring of the heap to a doubly linked list(DDL) wherein all roots of trees have pointers to its right and left root elements is imperative to ensure constant number of steps for each insert (used for analyzing run-time costs in [5]). The DDL implementation and cost shall be discussed in later sections of this paper.



(a) Total expected cost = $\mathrm{O}(n)$ (b) Total expected cost = $\mathrm{O}(n^2)$

Fig. 1: Anomaly with pure functional impelementation of Binomial heaps

[4] The ratio of the actual cost to the value expected is constant. In other words, if the cost of doing *blah* with foo data structure is $\mathrm{O}(n^2)$, then the ratio is $Cost_{Actual}/n^2$ should be constant

[5] Racket's vector API provides *(vector-append* $v_1 v_2 \ldots$*)*, used for cloning, makes fresh copy of all elements of given vectors

3 Fibonacci Heaps

Fibonacci heaps are a generalization of Binomial heaps. In addition to the operations provided by Binomial heaps, Fibonacci heaps provide a couple more operations. Specifically, they allow deletion of any element from the heap, and modification of value of any element. The prototypes for additional operations are as follows:

1. *decrement(h, i, δ)* Decrements the value of element i by δ in h
2. *delete(h, i)* Deletes element i from the heap h

Operation	Amortized Cost
makeheap	$\mathrm{O}(1)$
findmin	$\mathrm{O}(1)$
insert	$\mathrm{O}(1)$
deletemin	$\mathrm{O}(\log n)$
meld (lazy)	$\mathrm{O}(1)$
decrement	$\mathrm{O}(1)$
delete	$\mathrm{O}(\log n)$

Table 2: Amortized costs for Fibonacci Heaps

Fibonacci heaps are implemented in $rkt - heaps$ using a DDL for its roots. Findings from array and functional based implementation of Binomial heaps, highlight the downsides of not having mutations which leads to much higher costs for performing primitive operations in contrast to what has been proved in literature. For this reason, Fibonacci heaps do not have a purely functional implementation in $rkt - heaps$.

A Fibonacci heap after creation, if operated with only *deletemin* and *meld* operations, becomes a Binomial heap. Thus the corollary that size of tree rooted at r is exponential in $rank(r)$. The exponential size of a tree is also valid when the Fibonacci heap is operated upon with *decrement* and *delete* operations cutting sub-tress in a controlled fashion[5]. This fact is essential to analyze amortized costs to be $\mathrm{O}(1)$ and $\mathrm{O}(\log n)$ for *decrement* and *delete* operations respectively.

decrement operation mutates the value of the element, e, followed by check with its parent, p_e, on whether the heap property is violated or not. If it is, then the sub-tree rooted at element e, is added to the DDL of the heap. Every parent, starting at p_e is also added to DLL if a boolean flag is marked. This happens till either a root element or an element which does not have the flag marked is reached while traversing up the tree from e. This flag is maintained for every element e_i, except root elements, and marked as true if one of its children have been removed or false otherwise. A sub-tree rooted at e_i shall only be cut from its

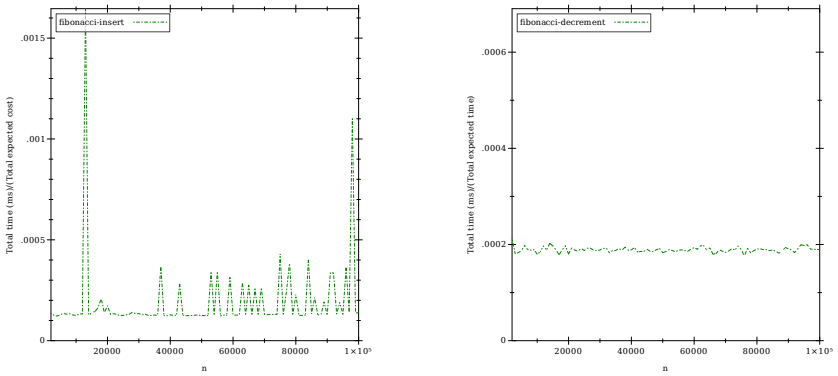
parent and added to DLL if a second child of e_i is removed i.e. the parent element is marked and a decrement operation on a child violates the heap property or a child is deleted.

delete makes the value of the element to be deleted less than the *min* value of the heap by means of *decrement* and then calls *deletemin*. This costs $\mathrm{O}(\log n)$.

4 Evaluation

In this section, comparisons are omitted between Binomial and Fibonacci heaps since one is purely functional and other is not. The functional technique was inefficient due to cloning of vectors as shown in Section 2. It is trivial to build on the Fibonacci heap implementation by excluding the *decrement* and *deletemin* operations to get a Binomial heap with lazy melds[5] at the same cost as shown for Fibonacci heaps in this section.

Fibonacci heap performance for operatios viz. *insert* and *meld* are shown in Figure 2. The curves have mountainous terrain as it the amortized total expected cost.



(a) Total expected cost = $\mathrm{O}(n)$ (b) Total expected cost = $\mathrm{O}(n)$

Fig. 2: Fibonacci Heaps, *insert* and *meld* operations

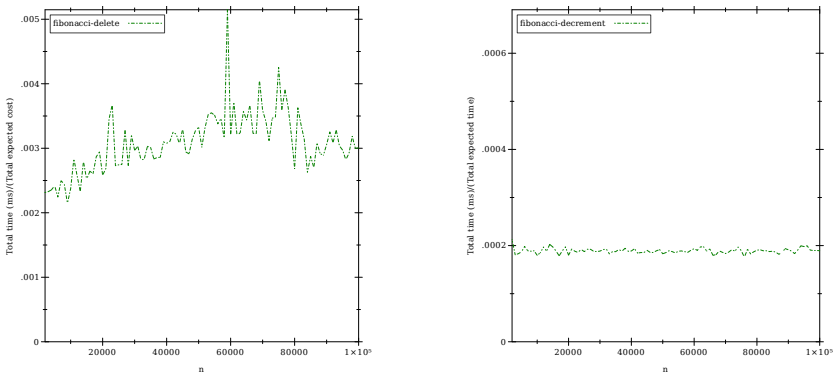
As stated before, *delete* involves a call to *decrement* to mutate the value of the node less than the *min* value in the heap, and subsequently to *deletemin*. In Figure 3a, there is a gradual increase in the slope of the curve. This is probably due to the increase in the constant, $\mathrm{O}(1)$, number of steps by *decrement* operation exceeding the $\mathrm{O}(\log n)$ number of steps by *deletemin*. It should be noted that for benchmarking a heap h of size n, the *delete* operation is called n times, and state of h is maintained in-between all *delete* operations. To find an element to be deleted, a simple random walk is performed on h. Algorithm 1 states the random walk procedure. Therefore, for decrement operation which

recursively removes parents while traversing up the graph, there is a high chance that the length of path traversed is positively correlated to number of calls to *delete* operation h. As a consequence, constant cost of *decrement* increases and thereby making the ratio of total time over total expected cost to increase with n for *delete* operation.

Algorithm 1: Random Walk for Fibonacci heap H_f

Require: A valid fibonacci heap H_f of size S_{H_f}

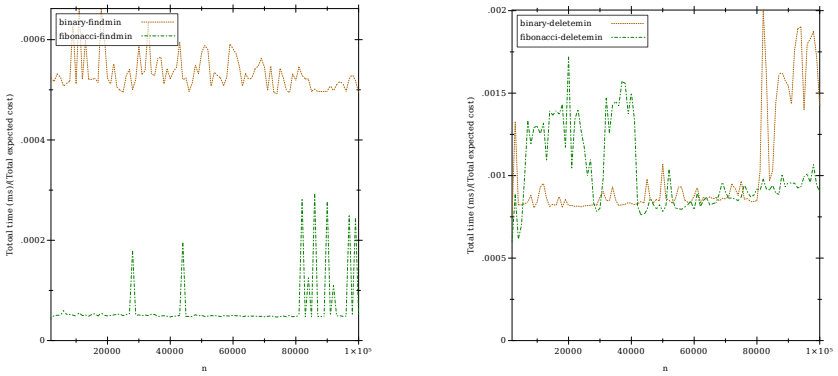
1: Using the fact that the rank of $H_f \leq \log_\phi S_{H_f}$, where ϕ is Golden Ratio[3], uniformly select a root element, $e_{current}$
2: $r := rank(e_{current})$
3: $d := F_{r+2}$ where F_i is the ith Fibonacci number
4: **while** $not\_leaf\_node(e_{current})$ and $depth(e_{current}) \leq d$ **do**
5: Select uniformly a child c, from the set of children of $e_{current}$
6: $e_{current} := c$
7: **end while**
8: **return** $e_{current}$



(a) Total expected cost = $\mathrm{O}(n \log n)$ (b) Total expected cost = $\mathrm{O}(n)$

Fig. 3: Fibonacci Heaps, *delete* and *decrement* operations.

Pre-packaged library for Binary heap in *Racket* is compared with Fibonacci implementation of $rkt - heaps$ on operations which have equal expected time bounds viz. $findmin$(Figure 4a) and $deletemin$(Figure 4b). Fibonacci heaps outperforms Binary heaps in both operations. The equivalent operations in Binary heap library are called as $heap - min$ and $heap - remove - min$.



(a) Total expected cost = $\mathrm{O}(n)$ (b) Total expected cost = $\mathrm{O}(n \log n)$

Fig. 4: Binary Heaps vs Fibonacci Heaps, *findmin* and *deletemin* operations.

5 Conclusions

In this paper:

- Binomial heaps included in $rkt - heaps$ do not run in expected time since they have been implemented in a functional style
- $rkt - heaps$ provide an efficient implementation of Fibonacci heaps using doubly linked lists(DDL)
- Fibonacci heaps perform better than the existing Racket library implementation of Binary heaps for comparable operations
- Fibonacci heaps can be easily extended to work as Binomial heaps although it will not be purely functional

5.1 Author's learnings

- Initially, this work was started using mit-scheme but later migrated to Racket since there was better support for generation of graphs, testing and debugging
- Understanding functional style of implementing heaps
- Run-time graphs[6] are essential to ascertain the algorithms were performing correctly as stated in literature
- Pure functional implementations can be slow
- A mutation can dependent on other mutations, hence sequence of mutations is important for correctness of an operation

[6] Thanks to Dhruv Matani (http://dhruvbird.com) for his inputs in making graphs more meaningful

References

1. Racket. `http://docs.racket-lang.org/guide/dialects.html`. [Accessed October 9, 2013].
2. Racket Binary Heaps. `http://docs.racket-lang.org/data/Binary_Heaps.html`. [Accessed October 9, 2013].
3. Richard A Dunlap. *The golden ratio and Fibonacci numbers*. World Scientific, 1997.
4. Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.
5. Dexter Kozen. *The design and analysis of algorithms*. Springer, 1992.
6. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

Binomial heaps

Version 5.3.4

October 21, 2013

```
(require binomial)
```

Binomial Heaps

A *Binomial heap* is a data structure for maintaining a collection of elements, such that new elements can be added and the element of minimum value extracted efficiently. This implmentation is purely functional hence *immutable*. *Binomial heap* allow only numbers to be stored in them.

heap-lazy? is just a check if the given argument complies with the binomial heap structure adopted in this implementation. *Binomial heaps* have a array-based implementation. All values of the heap are stored in a vector which is pointed by `car` of a pair. The `cdr` is the count/size of the heap. This pair is embedded within another pair's `car`. The `cdr` of the outer pair stores the min value of the heap.

```
(bino-makeheap val) → heap-lazy?
  val : number?
```

Returns a newly allocated heap with only one element `val`.

Examples:

```
> (define h (bino-makeheap 1))

> h
'((#(1) . 1) . 1)
```

```
(bino-findmin h) → number?
  h : heap-lazy?
```

Returns a minimum value in the heap `h`.

Examples:

```
> (define h (bino-meld (bino-makeheap 1) (bino-makeheap 2)))

> (bino-findmin h)
1
```

```
(bino-insert h val) → heap-lazy?
  h : heap-lazy?
  val : number?
```

Returns a newly allocated heap which is a copy of *h* along with *val*.

Examples:

```
> (define h (bino-makeheap 1))

> (bino-insert h 2)
'((#(#f 1 2) . 2) . 1)
```

```
(bino-deletemin h) → heap?
  h : heap?
```

Returns a newly allocated heap with the min value of the given heap *h* removed.

Examples:

```
> (define h (bino-makeheap 1))

> (bino-deletemin h)
'((#() . 0) . #f)
```

```
(bino-meld h1 h2) → heap?
  h1 : heap-lazy?
  h2 : heap-lazy?
```

Returns a newly allocated heap by coupling *h1* and *h2*.

Examples:

```
> (define h (bino-meld (bino-makeheap 1) (bino-makeheap 2)))

> h
'((#(#f 1 2) . 2) . 1)
```

```
(bino-count h) → exact-nonnegative-integer?
  h : heap-lazy?
```

Returns the count of the elements in the heap *h*

Examples:

```
> (define h (bino-meld (bino-makeheap 1) (bino-makeheap 2)))

> (bino-count h)
2
```

Fibonacci heaps

Version 5.3.4

October 21, 2013

```
(require fibonacci)
```

Fibonacci Heaps

A *Fibonacci heap* is a data structure for maintaining a collection of elements. In addition to the binomial heap operations, Fibonacci heaps provide two additional operations viz. *decrement* and *delete* exist. Although, it should be noted that that trees in *Fibonacci heaps* are not binomial trees as the implementation cuts subtrees out of them in a controlled way. The rank of a tree is the number of children of the root, and as with binomial heaps we only link two trees if they have the same rank.

Roots of binomial tress in the heap are stored in the form of a doubly linked list; each node has a reference to a left and right node.

```
(fi-makeheap val) → fi-heap?
  val : number?
```

Returns a newly allocated heap with only one element `val`.

Examples:

```
> (define h (fi-makeheap 3))

> h
#<fi-heap>
```

```
(fi-findmin h) → number?
  h : fi-heap?
```

Returns a minimum value in the heap `h`.

Examples:

```
> (define h (fi-meld! (fi-makeheap 1) (fi-makeheap 2)))

> (fi-findmin h)
1
```

```
(fi-insert! h val) → void?
  h : fi-heap?
  val : number?
```

Updates the left right pointers of the min node to accommodate the new node with `val`, `h`, with a new node having `val`.

Examples:

```
> (define h (fi-makeheap 1))

> (set! h (fi-insert! h 2))

> (fi-heap-size h)
2
```

```
(fi-deletemin! h) → fi-heap?
  h : fi-heap?
```

Updates the given heap `h` by removing the node with the minimum value and changing the reference to the new min node.

Examples:

```
> (define h (fi-meld! (fi-makeheap 1) (fi-makeheap 2)))

> (fi-deletemin! h)

> (fi-findmin h)
2
```

```
(fi-meld! h1 h2) → fi-heap?
  h1 : fi-heap?
  h2 : fi-heap?
```

Returns a newly allocated heap by coupling `h1` and `h2`.

Examples:

```
> (define h (fi-meld! (fi-makeheap 1) (fi-makeheap 2)))
```

```
> h
#<fi-heap>
```

```
(fi-decrement! h noderef delta) → void?
  h : fi-heap?
  noderef : node?
  delta : number?
```

Updates the value of *noderef* and if the heap condition is violated, then parent of the *noderef* is checked and if already marked, it is removed. This happens recursively until the root of the tree is reached or a parent which is not marked.

Examples:

```
> (define h (fi-meld! (fi-makeheap 1) (fi-makeheap 2)))

> (fi-decrement! h (fi-heap-minref h) 2)

> (fi-findmin h)
-1
```

```
(fi-delete! h noderef) → void?
  h : fi-heap?
  noderef : node?
```

Updates the heap *h* by deleting the *noderef* and updating its parent and children. Here also if the parent is marked, then it is also removed from the tree and added as a root. Happens until the root of the tree is reached or a parent is not marked.

Examples:

```
> (define h (fi-meld! (fi-makeheap 1) (fi-makeheap 2)))

> (fi-delete! h (fi-heap-minref h))

> (fi-findmin h)
2
> (fi-heap-size h)
1
```

```
(fi-heap-minref h) → node?
  h : fi-heap?
```

Returns the reference of the node which has the minimum value in *h*.

```
(fi-heap-size h) → exect-nonnegative-integer?
  h : fi-heap?
```

Returns the size of the heap *h*.

```
(fi-node-val n) → number?
  n : node?
```

Returns the value stored in the node *n*.

```
(fi-node-children n) → vector?
  n : node?
```

Returns a vector of all children of node *n*. If *n* does not have any children then a empty vector will be returned.

```
(fi-node-parent n) → node?
  n : node?
```

Returns the parent node of *n* if there exists one or #f.

```
(fi-node-left n) → node?
  n : node?
```

Returns *n*'s left node in the circular linked list. If *n* is the only node in list then a reference of *n* will be returned.

```
(fi-node-right n) → node?
  n : node?
```

Returns *n*'s right node in the circular linked list.