Binomial and Fibonacci Heaps in Racket (rkt-heaps)

Abhinav Jauhri
abhinavjauhri@gmail.com

No Institute Given

Abstract. Library[1] providing data structures viz. Binomial heap [3] and Fibonacci heap[1] are described along with primitive operations, performance and usage.

1 Introduction

rkt − heaps library is made for demonstration in a contest, *Lisp In Summer Projects*[2]. The goals of any participant in the contest were to build and demonstrate a project using any LISP-based technology. *rkt − heaps* uses Racket language belonging to the Lisp/Scheme family.

Section 1 & 2 describe Binomial and Fibonacci heaps respectively. *Section 3* has performance results to complement the theoretical run-time bounds. Comparison with an existing library for Binary heaps[3] in Racket is also added for evaluation purposes. *Section 4* talks about 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. They allow the following operations:

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

Before studying costs and implementation of the mentioned operations, some terms need brief explanations for reader's convenience.

[1] Will be referred as *rkt − heaps* throughout the paper
[2] http://lispinsummerprojects.org/
[3] http://docs.racket-lang.org/data/Binary_Heaps.html

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* - 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* - It states the number of children of a node. For Binomial and Fibonacci heaps, only when no *decrement* and *delete* operations are performed, the rank of a root node is k with 2^k nodes.

Making a new heap with one element with *makeheap* and finding the *min* pointer to get the minimum value with $findmin$ requires constant number of steps. Although, it may not be intuitive to see why *insert* takes just constant amortized cost. Inserting a value is equivalent to making a heap with a single element and then merging it with the existing heap. In melding, combining two trees with same rank k leads to a tree of rank $k+1$. If one already exists in either of the heaps, then a tree of rank $k+2$ is made. This happens recursively till there is single tree of some rank ($> k$) in the resulting heap. Each time trees are combined, the heap property is imposed[4]. This make the average cost over n operations to be $\mathrm{O}(1)$ for *insert*. Similarly, the same idea of combining of roots of trees can be used for *deletemin* operation. Take all children of the min root and meld them with the remaining trees in the heap, and update the min pointer, costing $\mathrm{O}(\log n)$.

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 having more than one binomial tree of rank k in the resulting 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 $vectors$ library. Being purely functional ensures there is no mutation 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

[4] 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

the divisor(total cost) for the dividend(total time) is quadratic, and therefore the curve has slope zero[5]. Alternatively, taking the divisor as linear, the total run-time cost depicts a linear increase in cloning the vector, which is to say that nth insertion step clones the $(n-1)th$ heap vector[6]. 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 nodes is imperative to ensure constant number of steps for each insert (used for analyzing run-time costs in [2]). The DDL implementation and cost shall be discussed in later sections of this paper. The array implementation was done out of pedagogical curiosity of the author.

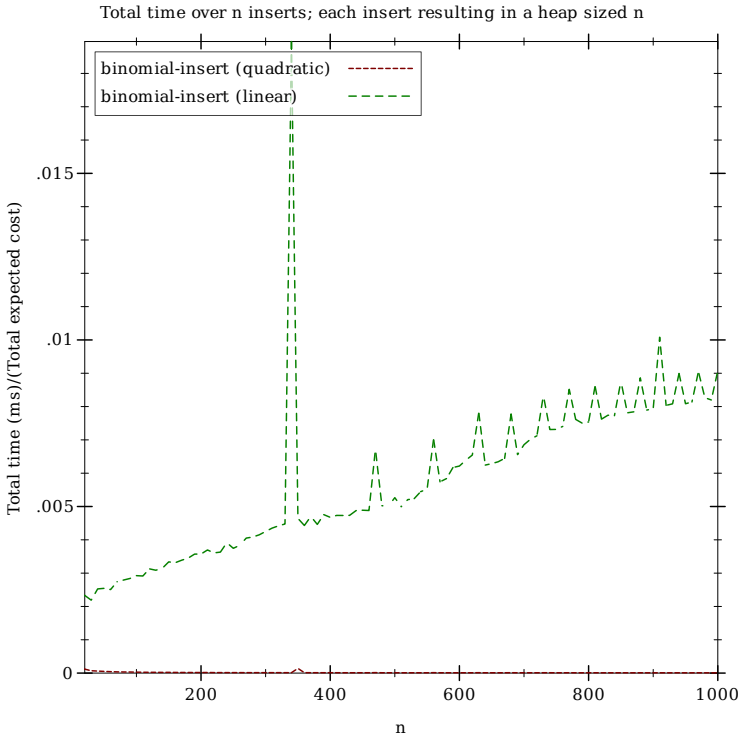

Fig. 1: Anomaly in *insert* operations for Binomial heaps. The y-axis shows the (total runtime cost)/(total expected cost). For one curve, the total expected cost is linear in n, and for the other it is quadratic in n^2. Total time is time for n insert operations and maintaining the heap state in-between each insert

[5] The ratio of total cost and n^2 will be constant

[6] 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 allowing additional operations other than those in Binomial heaps. Specifically, they allow deletion of an element from the heap, and modification of the value of an element. The prototypes for additional operations are as follows:

1. *decrement(h, i, δ)* Decrements the value of 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 based and functional 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.

As stated earlier that in a Fibonacci heap if only *deletemin* and *meld* operations were to be considered, then every tree becomes a binomial tree, thus ensuring the size of tree rooted at r to be exponential in $rank(r)$. The exponential size of a tree is also valid for Fibonacci heaps with operations like *decrement* and *delete* [2], and essential to ensure cutting of trees cost $\mathrm{O}(1)$ and $\mathrm{O}(\log n)$ for *decrement* and *delete* operations respectively.

decrement operation mutates the value of the element, and followed by check on whether the heap property is violated or not. If it is, then the sub-tree rooted at that element, is added to the DDL of the heap. Ever element e has a flag marked as true if one of its children have been removed and false otherwise. The sub-tree rooted at e shall also cut from its parent if a second child of e is removed i.e. the element is marked.

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

4 Evaluation

5 Usage

6 Conclusion

References

1. 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.
2. Dexter Kozen. *The design and analysis of algorithms*. Springer, 1992.
3. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.