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 participating individual 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

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

[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

operations is small. Another, relevant term for study of such heaps is *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 *makeheap* and finding the *min* pointer to get the minimum value requires constant number of steps. It is not intuitive to see why *insert* takes just constant amortized time. Inserting a value is interpreted as 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 i.e. the minimum between the two roots becomes the root of the combined tree. This make the average cost over n operations to be $\mathrm{O}(1)$ for *insert*. Similarly, the combining of roots of trees can be extended 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. 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 heaps is $rkt{-}heaps$ is an array-based purely functional implementation. Specifically using 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 (linear)* signifies the divisor(amortized cost) for the dividend(average cost) is linear in n, and therefore the curve has slope zero. Alternatively, taking the divisor as constant, the average 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[4]. Modifying the structuring of the heap to a doubly linked list(DDL) 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.

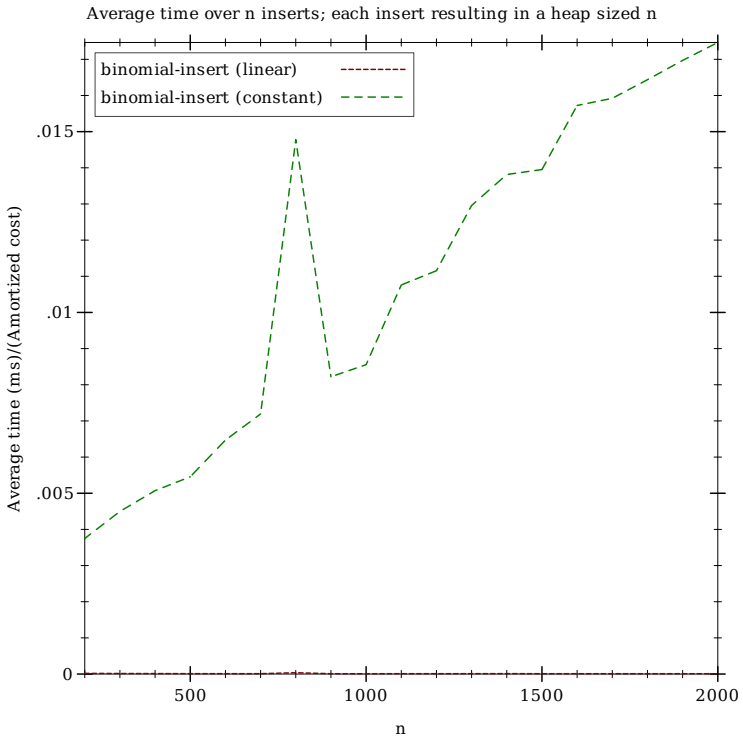

Fig. 1: Anomaly in *insert* operations for Binomial heaps. The y-axis shows the (average runtime cost)/(amortized cost). For one curve it is linear in n, and for the other it is just a constant. By average, it means over n insert operations and maintaining the heap state in-between inserts

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

[4] Racket's vector API provides *(vector-append v_1 v_2 ...)*, used for cloning, makes fresh copy of all elements of the given vectors

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	$O(1)$
findmin	$O(1)$
insert	$O(1)$
deletemin	$O(\log n)$
meld (lazy)	$O(1)$
decrement	$O(1)$
delete	$O(\log n)$

Table 2: Amortized costs for Fibonacci Heaps

Fibonacci heaps are implemented in $rkt - heaps$ using a DDL based on the learnings from the array based implementation of Binomial heaps. As stated earlier that in a Fibonacci heap only *deletemin* and *meld* operations are consdiered, then every tree becomes a binomial tree, thus ensuring the size of tree rooted at r to be exponential in $rank(r)$. This is also valid for Fibonacci heaps with operations like *decrement* and *delete* [2].

decrement operation after chaning the value of an element, check whether the heap propoerty is violated or not. If it is, then the sub-tree rooted at that element, is added to the DDDL of root of the heap. Ever element has a flag marked as true if one of its children have been cut or deleted. This element if makred and another child is cut, the lement is also added to the DDL of roots of theheap.

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.