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.