Development of heap structure for data storage and output in Clojure

Jiarui Zhu, *Nov 2021 - Mar 2022*

1. Introduction to heap

In computer science, a heap is a specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property: in a max heap, for any given node C, if P is a parent node of C, then the key (the value) of P is greater than or equal to the key of C. In a min-heap, the key of P is less than or equal to the key of C. The node at the "top" of the heap (with no parents) is called the root node.

2. Why need heap in clojure

During the development of Clojask, we faced the problem that we didn't have an efficient model for the sorted data storage and output. Also, there are no Clojure open-source packages supporting this task.

3. Design ideas

Since Clojure is a modern, dynamic and functional dialect of the Lisp programming language on the Java platform, our first idea is to find a mature Java library and provide the interface for Clojure application.

Therefore, Java PriorityQueue, an unbounded priority queue based on a priority heap is chosen as the base. All the related works and examples are shown in the ***heap from java*** directory.

The second idea is to see whether a pure Clojure package can handle this job. Clojure record is used for the development of data structure. All the related work is in ***heap from clj*** and the detailed program structure is shown in part 4.

4. Program structure

For the code in ***heap from clj***, there are several main functions. Their performance and interdependent relationship are shown below.

Data Structures:

Heaptree
root: Heapnode order: "ASC"/"DESC"; default ASC

Heapnode
lc: Heapnode rc: Heapnode data: any category matches compare function

Public Functions:

make-heaptree
Construction function of Heaptree input: [root] or [root order] output: a Heaptree structure

heap-compare
Comparative function input: [x1 x2] which are two data output: True if x1 should be the parent of x2. False otherwise.

heap-push
Push a new data to the tree input: [tree data] output: a new Heaptree structure functional dependency: heap-sort

heap-pop
Pop out the root and construct a new heap input: [tree] output: [data new-tree] When the tree is empty after popping, the function will return a tree with nil data in the root node. functional dependency: heap-sort, find-leave

Also, there are two private functions heap-sort and find-leave for package internal use.

5. Performance and Results

Both paths support push and pop operations, and ***heap from clj*** also supports personalized compare.

For ***heap from java***, it has the same running performance as Java Priority Queue. And for ***heap from clj***, it takes O(log n) steps for both pop and push operations.

Since Clojure does not have an efficient method for partially modifying objects, the current version can only complete heap updates by constantly constructing new objects and replacing old ones, which might be a huge cost of storage space.

6. Limitation and Future Work

In this edition of ***heap from clj***, we lack a search function, a get root function, and the initialization cannot add a number of data in one operation, which can be further developed in future editions.

Also, the current version does not implement complete encapsulation, which may cause tree imbalance and have a higher delay than estimated.

Supervised by
Dr. Matthias Buehlmaier

香 港 大 學
THE UNIVERSITY OF HONG KONG