Lazy Pairing Heap Functional Data Structure

Group 9:
Angelique Crawley
Chris Panican
Gerry Xu

Table of Contents

Introduction

Functional data structures in computer science, are described as data structures that can be implemented with a functional language such as Haskell. Functional language or functional programming are based on mathematical functions that use conditional expression and recursion to perform computation. When functional programming is combined with data structures, functional data structures takes form. The main difference between non-functional data structures and functional data structures is that, functional data structure are immutable, which means objects whose state cannot be modified after it has been created.

Since functional data structures are immutable, they possess several advantages such as persistency, thread safety, and quick copy of objects. Persistency is the ability to preserve the previous version of itself when it is modified. Thread safety prevents multiple threads that are operating on the state to be modified or corrupted. Quick copy of objects allows objects to be shared by references. The functional data structure that will be discuss and implement is pairing heap which will incorporate lazy evaluation, as well as the use of streams.

Brief History & Definition of Pairing Heap

A pairing heap is a heap data structure that was introduced in 1986 by Michael Fredman, Robert Sedgewick, Daniel Sleator, and Robert Tarjan. This heap data structure was originally inspired by splay trees, which is a self-adjusting binary search tree that rearranges the tree structure after every access in order to move the accessed element to the top of the tree. Pairing heap's nodes do not store any information aside from the key value and pointer references needed to traverse the structure. Generally the pointer to the leftmost child, and two sibling nodes, are stored, but there are other implementations, that store only the left and right child, or parent child if it is not a full tree. An additional bit would then be required to identify whether the child is left or right. Pairing heaps were also motivated by and introduced due to the need for a simpler version of Fibonacci heaps. It was conjectured that they have the same amortized complexity given how much resource such as time or memory it takes to execute each operations. Pairing heap has been observed to have excellent performance in practice where all of it operations are in *O(log n)* amortized time.

A pairing heap is a self adjusting min-heap ordered collection of linked binomial trees. It maintains the following properties: each node has zero or more children, that are listed from left to right, and the child's key value is always larger than its parent's. Pairing heap has a basic pairing operation, which combines two pairing heaps into one by attaching the root with the larger key value to the other root as its leftmost child. Pairing heap also have the following operations; `make-heap`, `find-min`, `insert`, `merge`, and `delete-min`. All of the pairing heap operation take constant time, except for delete-min which takes linear time in terms of the number of children of the root.

Lazy Evaluation & Streams

Lazy Evaluation was first proposed by Christopher Wadsworth as an optimization of normal-order reduction in the lambda calculus. Lazy Evaluation avoids repeated evaluation, the arguments that are passed in are initially unevaluated, they are only evaluated when the computation requires the results to continue. After the arguments are evaluated, the value of the evaluation of the argument are stored and cached, the value can then be retrieved at any time. This process is known as memoization. Lazy evaluation supports two primitives: one to suspend the evaluation of an expression which is known as `delay`, and one to resume the evaluation of a suspended expression, which is known as `force`.

An example of lazy evaluation is streams. Streams from an abstract point of view, is viewed simply as a sequence. Streams allows the introduction of delayed evaluation, which enables us to represent very large or even infinite sequences as streams. With the help of streams, it helps us model systems that have state without ever using assignment or mutable data. As a result, we can build models that avoid all the drawbacks that is inherent in introducing assignments. Streams can also manipulate sequences without incurring the costs of manipulating sequences as lists. Programs can be formulated elegantly with sequence manipulations, while retaining the benefits of incremental computation.

Streams are the same as list and they are implemented as lists. However, the difference is the time at which the elements are evaluated. With ordinary list, `car` and `cdr` are evaluated at construction time, while with streams, `cdr` isn't evaluated until selection time, therefore, streams are classified as delayed lists. Streams have a constructor `cons-stream`, and two selectors, `stream-car` and `stream-cdr` that are shown below:

```
(define-syntax cons-stream
     (syntax-rules ()
          ((_ a b) (cons a (delay b)))))

(define (stream-car stream) (car stream))
(define (stream-car stream) (force (cdr stream)))
```

Based off the implementation of the constructor, `cons-stream` and the two selectors, `stream-car` and `stream-cdr` in scheme, stream uses a special form called `delay` and `force`. Delay as the name suggest, delays the evaluation of the expression and returns a delayed object which is a promise to evaluate at some point in the future. Force takes the delayed object and performs the evaluation, which forces the delay to fulfill its promise. We will now incorporate lazy evaluation with streams into our pairing heap functional data structure.

Pairing Heap Pseudo Code

A pairing heap is based on a tree-like structure where each parent has a certain relationship to its children. Pairing heaps maintain a min-heap property that all parent nodes always have a smaller value than their children. We will be implementing the minimum functionalities such as; `make-heap`, `find-min`, `insert`, `merge`, and `delete-min`.

make-heap

A simple heap can be accomplished by using lists. The first element on the list will contain the root/parent. The heap can have: parent node with a child; a single root node; or it can be empty. Each node keep track of a pointer to its leftmost child and pointers to its sibling nodes.

```
PairingTree[Elem] = Heap(elem: Elem, subheaps: List[PairingTree[Elem]])
PairingHeap[Elem] = Empty | PairingTree[Elem]
```

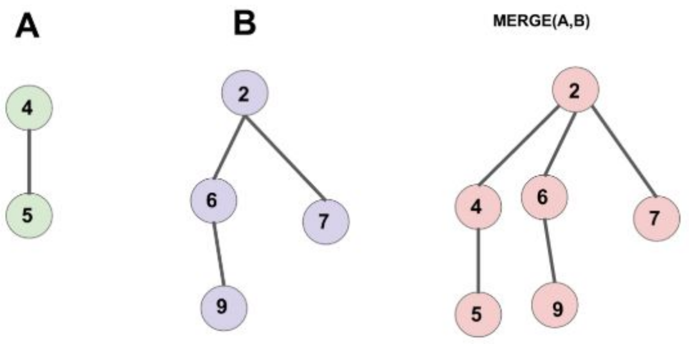
insert(heap, value)

This function will add a new value to an existing heap. If a value smaller than the parent node is inserted, then that value will be the new root node. This function will use another auxiliary function which will merge the existing heap with a new heap only containing the new value.

```
func insert(elem: Elem, heap: PairingHeap[Elem]) -> PairingHeap[Elem]
  return merge(Heap(elem, []), heap)
```

merge(heap1, heap2)

This function will accept two different heaps. If both pairing heaps are non-empty, the function will return a new heap where the smallest value between two heaps will be the new root and adds the other heap to the list of sub-heaps.



```
func merge(heap1, heap2: PairingHeap[Elem]) -> PairingHeap[Elem]
  if heap1 is Empty
    return heap2
  elsif heap2 is Empty
    return heap1
  elsif heap1.elem < heap2.elem
    return Heap(heap1.elem, heap2 :: heap1.subheaps)
  else
    return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

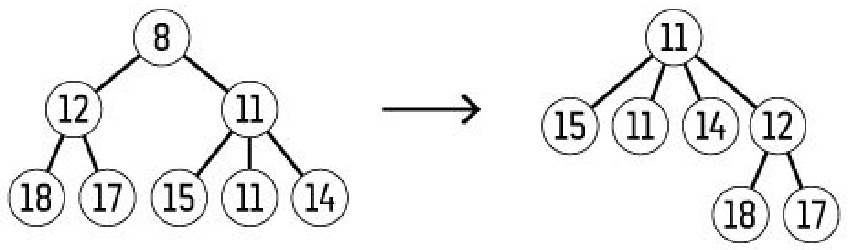
find-min(heap)

Get the smallest element in the heap. Since we are maintaining a min-heap property, we just need return the top element of the heap.

```
func find-min(heap: PairingHeap[Elem]) -> Elem
  if heap is Empty
    error
  else
    return heap.elem
```

delete-min(heap)

Remove the smallest object which is the root. When the root is removed, we are left with zero or more max trees. Therefore, after deleting the root node we must merge all these trees together. An auxiliary function `merge-pairs` will do the merging operation.



```
func delete-min(heap: PairingHeap[Elem]) -> PairingHeap[Elem]
  if heap is Empty
    error
  else
    return merge-pairs(heap.subheaps)
```

merge-pairs(heap)

This is an auxiliary function to recursively merge the subheaps from left to right then merges the resulting list of heaps from right to left.

```
func merge-pairs(list: List[PairingTree[Elem]]) -> PairingHeap[Elem]
  if length(list) == 0
    return Empty
  elsif length(list) == 1
    return list[0]
  else
    return merge(merge(list[0], list[1]), merge-pairs(list[2..]))
```

Lazy/Stream Implementation

The pseudo code in earlier section only showed a basic pairing heap. This section will focus on how laziness can be implemented into pairing heaps. Lazy pairing heaps have an advantage of being very fast by having a worst case running time of *O(1)* for operations `insert`, `find-min`, and `merge` since through lazy evaluation, the child nodes are only being processed when needed. In our implementation, this is true because the parent node is a regular element while the child returns a promise. The `delete-min` function has an amortized running time of *O(log n)* as it evaluates the child nodes. There are six main operations for lazy pairing heap: `make-heap`, `isHeapEmpty?`, `insert`, `merge`, `find-min` and `delete-min`. Each of these operations will use stream in one form or the other.

We start off by initializing some auxiliary function that will help us implement our lazy/stream implementation of our pairing heap functional data structure:

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
(define (stream-cadr stream) (cadr stream))
(define (stream-cddr stream) (force (cdr (force (cdr stream)))))
(define (empty-heap)
   '())
```

Some of the auxiliary function that uses `force` allows us to take the delayed object as an argument and use `force` to perform the evaluation of the result. The `stream-cadr` function will be used in our `merge` function and `stream-cddr` will be used in our `delete-min` function.

Our first step is to implement a function that allow us to create a heap that utilizes the `cons-stream` function, which is a special form of `cons` that uses a pair of `cars` and delayed `cdrs`, provided in earlier section.

```
(define (make-heap root trees)
    (cons-stream root (cons-stream trees '())))
```

`make-heap` takes in two parameters, root and trees, the root will be the parent node of the pairing heap and the tree elements will be the children of the parent node. Using `cons-stream`, our `make-heap` will always return the `car` of the root followed by a promise which contains the tree node elements. To display all the nodes in the pairing heap that was created by `make-heap`, the `display-stream` and `stream-for-each` function are used which allow one to traverse through the heap and force the remaining element to complete the evaluation and display each nodes in the heap to the console.

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (stream-for-each proc s)
  (if (stream-null? s)
      (newline)
      (begin
     (proc (stream-car s))
     (stream-for-each proc (stream-cdr s)))))
```

After a heap is created, we need to check if the heap is empty or not.

```
(define (isHeapEmpty? pheap)
  (if (equal? (empty-heap) pheap)
      #t
      #f
      ))
```

`isHeapEmpty?` function takes in a heap as a parameter and check if the heap equals an empty heap, if it is indeed empty, returns true, otherwise it returns false.

Our next implementation is to merge two different heaps together into a single pairing heap.

```
(define (merge pheap1 pheap2)
  (cond ((isHeapEmpty? pheap1) pheap2)
        ((isHeapEmpty? pheap2) pheap1)
        ((< (stream-car pheap1) (stream-car pheap2))
         (make-heap (stream-car pheap1) (cons-stream pheap2
(stream-cadr pheap1))))
        (else (make-heap (stream-car pheap2) (cons-stream pheap1
(stream-cadr pheap2))))))
```

The `merge` function takes in two heaps, and check whether one of the heap is empty or not. If either heap is empty, return the other heap. If both heaps are not empty, check which heap's parent node is the smallest by compare their first element using `stream-car`. If the first heap's parent node element is smaller than the second heap's parent node element, then make a new heap, with the new parent node element being the first heap's parent node and the tree of the heap being the con-stream of the second heap combining with the first heap's subtree node elements. If the second heap's parent node element is smaller than the first heap's parent node element, then make a new heap, with the new parent node element being the second heap's parent node and the tree of the heap being the con-stream of the first heap combining with the second heap's subtree node elements.

The `insert` function allows one to insert an element into the heap:

```
(define (insert pheap val)
  (merge pheap (make-heap val '())))
```

The `insert` function takes in two parameters, one being the heap and the second one being a value that will be inserted into the heap. Inside the insert function, it calls the merge function which takes in the heap and the new value that will be added to the heap. The `merge` function will then check, by comparing the value that will be inserted into the heap and the first element, in this case, the parent node element in the heap. If the value that will be inserted into the heap is smaller than the parent node element in the heap, then new heap will be created with the inserted value being the new parent node of the new heap.

The `find-min` function returns the smallest element in the pairing heap:

```
(define (find-min pheap)
  (if (isHeapEmpty? pheap)
      (delay (display "the heap is empty"))
      (stream-car pheap)))
```

`find-min` function takes in a heap as a parameter and checks if the heap is empty or not. If the heap is empty, it will be return a `delay` which contains a promise with the message “the heap is empty.” In order to display this promise, a `force` call is required for it to display the message on the console. If the heap is not empty, it will just return the top element of the heap, which is the smallest element in a pairing heap.

Our last implementation of the basic pairing heap operation is the `delete-min` function.

```
(define (delete-min pheap)
  (define (merge-pairs pheap-pair)
    (cond ((isHeapEmpty? pheap-pair) '())
          ((isHeapEmpty? (stream-cdr pheap-pair))
                         (stream-car pheap-pair))
          (else (merge (merge (stream-car pheap-pair)
                              (stream-cadr pheap-pair))
                       (merge-pairs (stream-cddr pheap-pair))))))
  (merge-pairs (stream-cadr pheap)))
```

`delete-min` is the most difficult operation to implement. The function takes in a heap, and defines a helper auxiliary/helper function called `merge-pairs` which takes in the pair of remaining trees; the pair heap does not contain the parent root node. It checks to see if the pair heap is empty and also checks if the `stream-cdr` element is empty as well. If the pair heap `stream-cdr` element is empty then the pair heap must originally contain just two elements—the parent and the child node element—so, it should return the child element with `car`. If the heap is not empty, recursively merge the pair heaps from left to right then merge the resulting heaps from right to left until the new parent node is the smallest node element follow by the child node elements.

Conclusion

Functional data structures are unique because of their persistence or immutability of data. This provides various advantage such as improvements in concurrency, memory usage, and most notably, by using functional data structures we can implement lazy persistent data structures. Lazy evaluation has the advantage of delaying an expression until its value is needed which reduces running time. One data structure that can take advantage of these attributes is Pairing Heap. Pairing Heap by itself is already a fast algorithm with *O(1)* access/insert, and *O(log n)* delete amortized times. By adding laziness on Pairing Heap, we were able to make it even more efficient by improving its persistency. Overall, we learned how to make a functional data structure and taking advantage of lazy evaluation - which is one of the features of Scheme. After learning the concepts of delay and force, it is important that we consider promises when improving the efficiency of our algorithms.

References

1. Purely Functional Data Structures

   By: Chris Okasaki, September 1996

   https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

2. A Linear Potential Function for Pairing Heaps

   By: John Iacono and Mark Yagnatinsky, June 28, 2016

   https://arxiv.org/pdf/1606.06389.pdf

3. Pairing Heap: The Forward Variant

   By: Dani Doftman, Haim Kaplan, Laszlo Kozma, and Uri Zwick, Jun 21, 2018

   https://arxiv.org/pdf/1709.01152.pdf

4. The Pairing Heap: A New Form of Self-Adjusting Heap

   By: Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator and Robert E. Tarjan, 1986

   https://www.cs.cmu.edu/~sleator/papers/pairing-heaps.pdf

Appendix

```
; Create a cons-stream function
(define-syntax cons-stream
  (syntax-rules ()
    ((_ a b) (cons a (delay b)))))

; Auxiliary functions
; Add a new stream-cadr for our merge function
; and a new stream-cddr for our delete function
(define stream-null? null?)

(define (stream-car stream) (car stream))

(define (stream-cdr stream) (force (cdr stream)))

(define (stream-cadr stream) (cadr stream))

(define (stream-cddr stream) (force (cdr (force (cdr stream)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      (newline)
      (begin
       (proc (stream-car s))
       (stream-for-each proc (stream-cdr s)))))

; displays the stream of the pairing heap
(define (display-stream s)
  (stream-for-each display-line s))

; prints it to the console
(define (display-line x)
  (display x))


; Definition of Pairing heap
(define (make-heap root trees)
  (cons-stream root (cons-stream trees '())))


; Creates an empty heap
(define (empty-heap)
  '())
```

```scheme
; Check if the heap is empty
(define (isHeapEmpty? pheap)
  (if (equal? (empty-heap) pheap)
      #t
      #f
      ))


; Merge
(define (merge pheap1 pheap2)
  (cond ((isHeapEmpty? pheap1) pheap2)
        ((isHeapEmpty? pheap2) pheap1)
        ((< (stream-car pheap1) (stream-car pheap2))
         (make-heap (stream-car pheap1) (cons-stream pheap2 (stream-cadr pheap1))))
        (else (make-heap (stream-car pheap2) (cons-stream pheap1 (stream-cadr pheap2))))))


; Insert
(define (insert pheap val)
  (merge pheap (make-heap val '())))


; Find min
(define (find-min pheap)
  (if (isHeapEmpty? pheap)
      (delay (display "the heap is empty"))
      (stream-car pheap)))


; Delete min
(define (delete-min pheap)
  (define (merge-pairs pheap-pair)
    (cond ((isHeapEmpty? pheap-pair) '())
          ((isHeapEmpty? (stream-cdr pheap-pair)) (stream-car pheap-pair))
          (else (merge (merge (stream-car pheap-pair) (stream-cadr pheap-pair))
                       (merge-pairs (stream-cddr pheap-pair))))))
  (merge-pairs (stream-cadr pheap)))


; Tests
; create 2 pairing heaps
(define pheap1 (make-heap 1 (make-heap 3 4)))
(define pheap2 (make-heap 2 (make-heap 5 6)))

; merge the two pairing heaps and return the parent root node being the smallest element,
; in this case 1 will be the parent root node since 1 < 2.
```

```
(merge pheap1 pheap2)

; insert 1 into pheap2 and create a new heap by calling merge in the
; insert function and display the min element of the new heap created
(find-min (insert pheap2 1))
; for this one it will return 1 for the min element since 1 is < 2 since it insert 1 into pheap2

; check if the heap is empty
(isHeapEmpty? pheap1)

; return the top element of the heap
; this returns 2 because the top element of the heap for pheap2 which is 2.
(find-min pheap2)
```