Handout 3
Verified Pairing Heaps

Construction and Verification of Software

Nova School of Science and Technology
Mário Pereira mjp.pereira@fct.unl.pt

Version of November 19, 2025

The goal of this handout is to verify, using Separation Logic in VeriFast, a Java implementation of a Pairing Heaps, *i.e.*, a priority queue data structure. A good description of the behavior of a Pairing Heaps can be found on the Wikipedia page: `https://en.wikipedia.org/wiki/Pairing_heap`. An even better presentation of such data structure is given by Sleator and Tarjan in their important paper *"The Pairing Heap: a new Form of Self-Adjusting Heaps"* [1]. Pairing Heaps are normally represented as *multi-way* trees, *i.e.*, each node can have multiple child nodes. However, binary trees are a much more convenient way to represent trees, and in fact there is a systematic way to translate a multi-way tree into an equivalent binary tree. Shortly, taking a node of a multi-way tree, its leftmost child becomes its left child in a binary tree; on the other hand, the right-sibling of a node becomes, in a multi-way tree, becomes the right child of that node in a binary tree. This is mechanism is very well-explained in the Wikipedia page: `https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree`. For reference, we provide a complete Java implementation of a Pairing Heaps in Appendix A.

During this handout you are supposed to complete the specification and proof given in the companion `.java` file. This includes:

- definition of predicates
- pre-conditions (*i.e.*, `requires` clauses)
- post-conditions (*i.e.*, `ensures` clauses)
- use of `open`/`close` primitives

The `Heap` and `PairingHeap` data types and corresponding operations are already implemented. **Do not change such implementations**.

1 Data Type Implementation

The given implementation of heaps is efficient, both in space and time. Internally, a pairing heap is just a heap-allocated binary tree of integer values. The `Heap` implementation uses the `Node` class to represent a heap-allocated node of the data structure. Each `Node` contains fields `left` and `right` that represent the left and right sub-trees, respectively. Field `val` stores an integer associated with each node. The representation invariant of class `Node` is already provided in the companion file. An object of class `PairingHeap` contains a single field: `head`, which points to the entry pointer on the heap.

2 Heap Representation Invariant

In this handout, we **do not care about the logical model** of the data structure. You are only supposed to specify the memory footprint of the data structure.

Exercise 1. Complete the predicate `tree` that captures the *representation invariant* for a well-formed, **potentially empty** binary tree. Such a predicate has the following signature:

```
//@ predicate tree (Node h;) = ...
```

where `h` stands for the entry pointer to the tree. □

Exercise 2. Complete the predicate `non_empty_tree` that captures the *representation invariant* for a well-formed, **non-empty** binary tree. Such a predicate has the following signature:

```
//@ predicate non_empty_tree (Node h;) = ...
```

where `h` stands for the entry pointer to the tree. □

Exercise 3. Complete the predicate `heap` that captures the *representation invariant* for a well-formed, **potentially empty** heap. Such a predicate has the following signature:

```
//@ predicate heap (Heap h;) = ...
```

where `h` stands for the entry pointer to the heap. A well-formed skew heap is simply a pointer into a well-formed binary tree. □

Exercise 4. Complete the predicate `non_empty_heap` that captures the *representation invariant* for a well-formed, **non-empty** heap. Such a predicate has the following signature:

```
//@ predicate non_empty_heap (Heap h;) = ...
```

where `h` stands for the entry pointer to the heap. A well-formed skew heap is simply a pointer into a well-formed binary tree. □

Exercise 5. Complete the predicate `pairing_heap` that captures the *representation invariant* for a well-formed, **potentially empty** Pairing Heap. Such a predicate has the following signature:

```
//@ predicate pairing_heap (Heap h;) = ...
```

where `h` stands for the entry pointer to the heap. A well-formed skew heap is simply a pointer into a well-formed binary tree. □

Exercise 6. Complete the predicate `non_empty_pairing_heap` that captures the *representation invariant* for a well-formed, **non-empty** Pairing Heap. Such a predicate has the following signature:

```
//@ predicate non_empty_pairing_heap (Heap h;) = ...
```

where `h` stands for the entry pointer to the heap. A well-formed skew heap is simply a pointer into a well-formed binary tree. □

3 Verified Heap Operations

3.1 Constructors

The constructors for this class are fairly simple: the first one simply initializes field `head` with `null` pointer and field `val` with the given argument, hence creating a singleton heap; the second one initializes fields `head` and `val` with the given arguments.

Exercise 7. Complete the specification of the `Heap` constructors. □

3.2 Emptiness Checking

Exercise 8. Complete the specification of method `isEmpty`, which checks whether the current heap is empty. □

3.3 Get the Minimum Element

Exercise 9. Complete the specification of method `getMin`, which returns the minimum element in the heap, *i.e.*, the root of the tree. This method is *read-only*, it is not supposed to modify the structure. □

3.4 Merging Two Heaps

Exercise 10. Complete the specification of the private method `merge`, which recursively merges two trees, returning a new valid tree. □

3.5 Add a New Element

Exercise 11. Method `add` is the public method used to insert a new element `x` into the heap. It starts by creating a singleton heap that contains only `v` and then calls `merge`.

Complete the specification of method `add`. □

3.6 Merging Pairs of Heaps

Exercise 12. The Pairing Heap data structure gets its name from the core, recursive procedure of merging a list of heaps by consecutively merging pairs of heaps, until only one is left. In our binary tree-based implementation, this method is made much simpler.

Complete the specification of the private method `mergePairs`, which recursively merges all the trees contained under argument `t`. □

3.7 Removing the Minimum Element

Exercise 13. Method `removeMin` is the public method used to remove the minimum element of the heap. It simply calls method `mergePairs` to return the new heap resulting from merging the descendants of the root node.

Complete the specification of method `removeMin`. □

4 Verified Pairing Heap Sequential Operations

4.1 Constructors

This ADT presents three constructors:

1. the default constructor, which initializes an *empty* Pairing Heap
2. the constructor that excepts an object of class `Heap` as argument, and it simply initializes field `head`
3. finally, the constructor that takes an integer value to build a new singleton Pairing Heap.

Exercise 14. Complete the specification of the `PairingHeap` constructors. □

4.2 Emptiness Checking

Exercise 15. Complete the specification of method `isEmpty`, which checks whether the current Pairing Heap is empty. □

4.3 Get the Minimum Element

Exercise 16. Complete the specification of method `getMin`, which returns the minimum element in the Pairing Heap, *i.e.*, the root of the tree stored in the `head` field. This method is *read-only*, it is not supposed to modify the structure. □

4.4 Add a New Element

Exercise 17. Method `add` is the public method used to insert a new element `x` into a Pairing Heap. It simply calls method `add` from the `Heap` class and then updates the `head` field of `PairingHeap`.

Complete the specification of method `add`. □

4.5 Removing the Minimum Element

Exercise 18. Method `removeMin` is the public method used to remove the minimum element of a Pairing Heap. It simply calls method `removeMin` from the `Heap` class and then updates the `head` field of `PairingHeap`.

Complete the specification of method `removeMin`. □

5 Verified Concurrent Heap Operations

Appendix B provides a skeleton implementation of a Concurrent Wrapper around the `PairingHeap` class.

Exercise 19. Complete `CPairingHeap` specification and implementation to build a verified Concurrent Wrapper around the sequential `PairingHeap`. Operations of the `CPairingHeap` class must be implemented using monitors and conditions. Your task is to:

- complete `CPairingHeap` implementation with the needed fields to represent monitor and conditions;
- define *Predicate Constructors* for the shared state and each condition;
- define a *Representation Invariant* for class `CPairingHeap`;
- provide pre- and post-conditions for every public method in `CPairingHeap`;
- provide an implementation for every public method in `CPairingHeap`;
- provide the necessary `open/close` primitives to state, for instance, which predicates hold after acquiring/releasing a lock or waiting for a condition.

Important note (1): methods `getMin` and `removeMin` are critical operations from class `PairingHeap`, since these require the data structure to be non-empty. Hence, this is the only monitor condition that you need to capture in the concurrent wrapper `CPairingHeap`.

Important note (2): it is likely that you might need to change the specification for some methods of `PairingHeap` and `Heap`, in order to have more fine-grained information to explore in the concurrent wrapper `CPairingHeap`. Think, for instance, about method `add`: can the updated heap be the empty heap?

Important note (3): when inserting a new element via method `add` from `CPairingHeap`, you are supposed to signal that, for sure, the heap is no longer empty.

Important note (4): when refining the specification of some methods of classes `PairingHeap` and `Heap`, you might have post-conditions of the form $\mathsf{B}\ ?\ \mathsf{H}_1\ :\ \mathsf{H}_2$, where B is some Boolean condition and H_1 and H_2 are heap predicates. For instance, the method `merge` might return an empty tree if both arguments are the empty tree; otherwise, it returns for sure a non-empty tree. Under such scenarios, you might need to reason by case analysis. hence introduce *arbitrary pieces of ghost code* in the middle of your method definitions. `VeriFast` allows you to do so using the following block, for instance:

```
/*@

  if (Bb) {
    open Hh1;
  }
  else {
    open Hh2;
  }

@*/
```

□

References

[1] Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

A Java Implementation of Pairing Heaps

```
class Node {
    private int val;
    private Node left, right;

    public Node (int val)
    {
        this.val = val;
        this.left = null;
        this.right = null;
    }

    public Node (Node left, int val, Node right)
    {
        this.val = val;
        this.left = left;
        this.right = right;
    }

    public int getVal ()
    {
        return this.val;
    }

    public Node getLeft ()
    {
        return this.left;
    }

    public Node getRight ()
    {
        return this.right;
    }

    public void setLeft (Node left)
    {
        this.left = left;
    }

    public void setRight (Node right)
    {
        this.right = right;
    }
}

class Heap {
    private int val;
    private Node tree;

    public Heap (int val)
    {
        this.val = val;
        this.tree = null;
    }
```

```
public Heap (int val, Node t)
{
    this.val = val;
    this.tree = t;
}

public static boolean isEmpty (Heap h)
{
    return h == null;
}

public static int getMin (Heap h)
{
    return h.val;
}

private static Heap merge (Heap h1, Heap h2)
{
    if (h1 == null) {
        return h2;
    }
    if (h2 == null) {
        return h1;
    }

    if (h1.val <= h2.val) {
        Node t = new Node(h2.tree, h2.val, h1.tree);
        Heap r = new Heap(h1.val, t);
        return r;
    }
    else {
        Node t = new Node(h1.tree, h1.val, h2.tree);
        Heap r = new Heap(h2.val, t);
        return r;
    }
}

public static Heap add (int v, Heap h)
{
    Heap tmp = new Heap(v);

    return merge(h, tmp);
}

private static Heap mergePairs (Node t)
{
    if (t == null) return null;
    if (t.getRight() == null) return new Heap(t.getVal(), t.getLeft());

    Node right = t.getRight();
    Node rightRight = right.getRight();

    Heap r = mergePairs(rightRight);

    Heap h1 = new Heap(t.getVal(), t.getLeft());
```

```
        Heap h2 = new Heap(right.getVal(), right.getLeft());

        Heap h = merge(h1, h2);

        return merge(h, r);
    }

    public static Heap removeMin (Heap h)
    {
        Heap r = mergePairs(h.tree);

        return r;
    }
}

public class PairingHeap {
    Heap head;

    public PairingHeap ()
    {
        head = null;
    }

    public PairingHeap (Heap head)
    {
        this.head = head;
    }

    public PairingHeap (int v)
    {
        this.head = new Heap(v);
    }

    public boolean isEmpty ()
    {
        return this.head == null;
    }

    public int getMin ()
    {
        return Heap.getMin(this.head);
    }

    public void add (int v)
    {
        Heap tmp = Heap.add(v, head);
        head = tmp;
    }

    public void removeMin ()
    {
        Heap tmp = Heap.removeMin(head);
        head = tmp;
    }
}
```

B Concurrent Pairing Heaps

```
  class CPairingHeap {
    private PairingHeap heap;
    // Other relevant fields: to be completed

    public CPairingHeap () { ... }

    public CPairingHeap (int x) { ... }

    public boolean isEmpty () { ... }

    public int getMin () { ... }

    public void removeMin () { ... }

    public void add (int x) { ... }
}
```