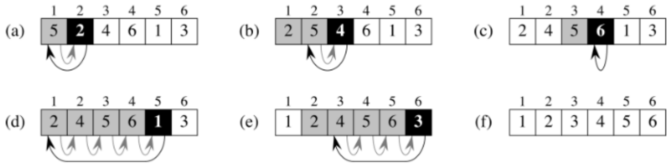
Invariants

Invariants

Invariant: A condition that is maintained by the algorithm throughout (parts of) its execution. Often forms the core idea behind the algorithm.

Example: Insertion sort:


(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

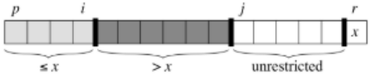
Invariant: Everything to the left of the black field is sorted.

When the loop stops: the entire array is to the left of the black field.

From the invariant it follows: the entire array is sorted. That is, the algorithm is correct.

Invariants

Example: Partition from Quicksort:



Invariant:

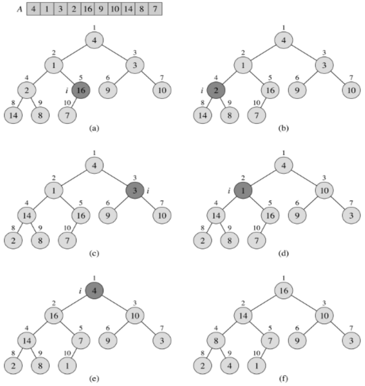
$$\boxed{\text{Light gray part} \le x < \text{dark gray part.}}$$

When the loop stops: Only x is white, the rest either light gray or dark gray.

From the invariant it follows that: the array is divided into three parts: “≤x”, “>x" and x . That is the algorithm is correct.

Invariants

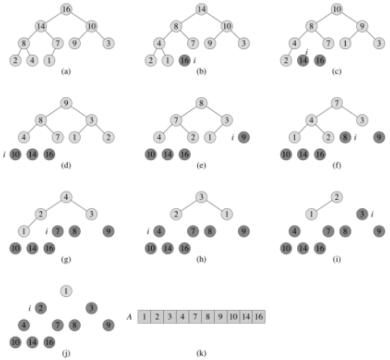
Example: Build-Heap:



Invariant: Subtrees whose root has index greater than the dark node obey heaporder.
When the loop stops: the root of the entire tree has an index greater than the dark node. The invariant then follows: The entire tree obeys heaporder. That is, the algorithm is correct.

Invariants

Example: Heapsort (and any Selectionsort-based sort):



Invariant:

The dark is sorted, and everything in the light is ≤ the dark.

When the loop stops: the entire array is dark. From the invariant it follows: the entire array is sorted. That is, the algorithm is correct.

Invariants

Example: searching in binary search trees.

Invariant:

> If the searched element k is found, it is in the subtree we have reached.

The algorithm must stop because we are looking at smaller and smaller subtrees.

When the algorithm stops: either k is found or we have ended up in an empty subtree.

In the latter case, the invariant then follows: k is not found in the tree. That is, the algorithm is correct (in both cases).

Invariants

Invariant under rebalancing after insertion in a red-black tree:

> There can be at most two consecutive red nodes on a root-to-leaf path in one place in the tree, but apart from this, the red-black requirements are met. After k iterations, there are k fewer black nodes between the problem and the root than at the start.

Invariant under rebalancing after deletion in a red-black tree:

> There may be a single blackened node somewhere in the tree, and if this blackening is counted, the red-black requirements are met. After k iterations, there are k fewer black nodes between the problem and the root than at the start.

The invariant demonstrates several things:

► The same case analysis works every time (always covers all possibilities, so the algorithm cannot get stuck as long as the problem is not solved).

► The algorithm must stop, either when the problem has disappeared or when it has reached the root (where it is easily solved).

Thus, the invariant shows that the algorithm is correct.

Invariants, more formally

Invariant for algorithm:

- A statement about the contents of memory (variables, arrays,. . .) that is true after all steps.
- At the end of the algorithm, correctness can be inferred from the statement (as well as the circumstances that caused the algorithm to stop).

Induction

That an invariant holds after every step is shown using induction:

$$\boxed{\begin{array}{l}\text{1) The invariant holds at the start.}\\ \text{2) If the invariant holds before a step}\\ \Rightarrow \text{the invariant holds after the step.}\end{array}} \Rightarrow \boxed{\text{Invariant always respected}}$$

(where "step" is often an iteration of a loop). That is: Show 1) and 2).

Induction ~ "Domino principle"

$$\boxed{\begin{array}{l}\text{1) Piece 1 falls}\\ \text{2) Piece k falls} \Rightarrow \text{piece k+1 falls}\end{array}} \Rightarrow \boxed{\text{All the pieces fall}}$$

Using invariants

Invariants can be used at two different levels of detail (with a smooth transition between them):

1. As a tool to develop algorithmic ideas: With the correct invariant, the essence of the method is captured, and the algorithm should "simply" be written based on maintaining this invariant.

2. As a tool to write code (or detailed pseudocode) and correctly demonstrate this concrete code.

In the first use case, softer descriptions (text, figures) are appropriate, as seen in previous examples. In the second use case, one must write down the invariant precisely in terms of concrete variables from the code, and argue based on the concrete code's changes to these.

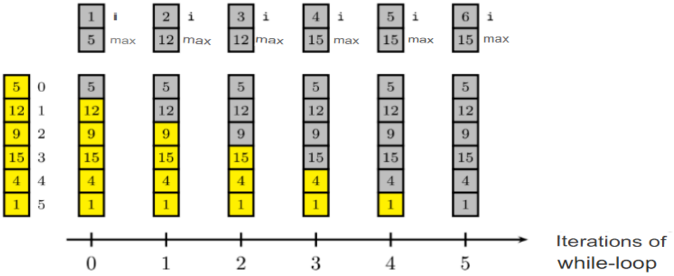
The example below of finding the largest element in an array illustrates this.

Example

Find largest element in array:

```
max = A[0]
i = 1
while i < A.length
  max = maximum(max,A[i])
  i++
```

Invariant: "After the k th iteration of the while loop, max contains the largest value of A[0..(i − 1)]"
Shown by induction on k.



1 i 5 max
2 i 12 max
3 i 12 max
4 i 15 max
5 i 15 max
6 i 15 max
5 12 9 15 4 1
0 1 2 3 4 5
Iterations of while-loop