Demystifying Smoothsort

ACM Tech Talks
January 7, 2011

What is Smoothsort?

- Sorting algorithm invented by **Edsger Dijkstra**.
- **Adaptive** variant of heapsort:
  - **O(n)** best-case runtime.
  - **O(n lg n)** worst-case runtime.

```
smoothsort:
|[ q, r, p, b, c, r1, b1, c1: int
 ; q:= 1; r:= 0; p, b, c := 1, 1, 1 {inveriant: P3' ∧ P4'}
 ; do q ≠ N
     → r1:= r
     ; if p mod 8 = 3
         → b1, c1 := b, c; sift; p:= (p + 1)/4; up; up
        [] p mod 4 = 1
         → if q + c < N → b1, c1 := b, c; sift
            [] q + c ≥ N → trinkle
           fi; down; p:= 2 * p
         ; do b ≠ 1 → down; p:= 2 * p od; p:= p + 1
       fi; q:= q + 1; r:= r + 1
   od {P3' ∧ P4'}; r1:= r; trinkle {inveriant: P3 ∧ P4}
 ; do q ≠ 1
     → q:= q - 1
     ; if b = 1
         → r:= r - 1; p:= p - 1; do even(p) → p:= p/2; up od
        [] b ≥ 3
         → p:= p - 1; r:= r - b + c
         ; if p = 0 → skip [] p > 0 → semitrinkle fi
         ; down; p:= 2*p + 1; r:= r + c; semitrinkle
         ; down; p:= 2*p + 1
       fi
   od
]|
```

Goals of this Talk

- Understand **how smoothsort works**.
- Better appreciate the connection between **priority queues** and **sorting algorithms**.
- Explore **implicit data structures** and how they affect memory usage.

The Basics: Priority Queues

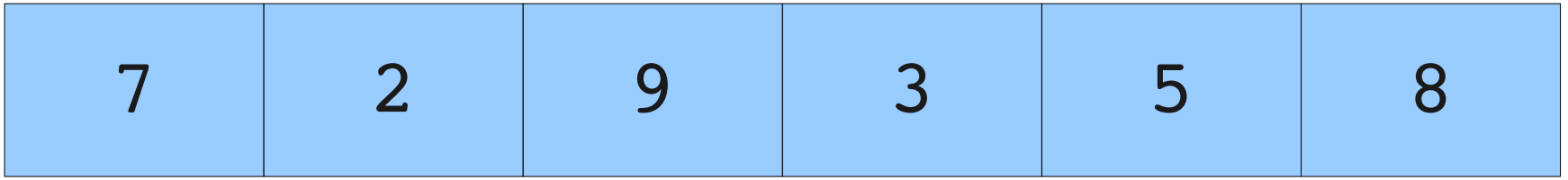
- A **priority queue** is a data structure supporting:
  - **insert**, which adds an element to the queue.
  - **dequeue-max**, which returns and removes the largest element from the queue.
- There are **many** implementations of priority queues.

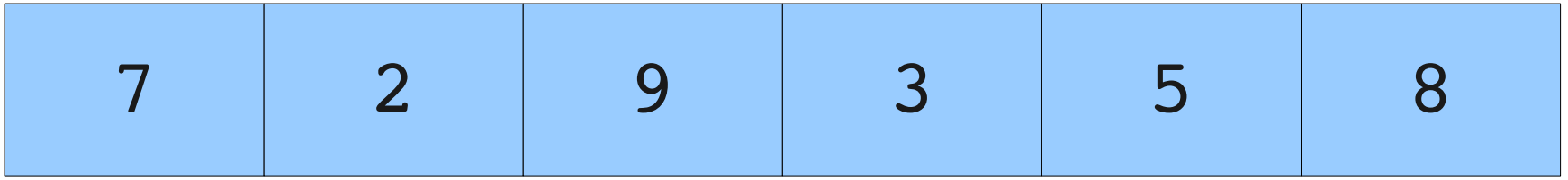
Priority Queue Sort

- Given an array **A**[1 .. n], sort array **A**.
- Algorithm:
  - Create a priority queue **PQ**.
  - For i = 1 .. n, **PQ.insert**(**A**[i])
  - For i = n .. 1, **A**[i] = **PQ.dequeue-max**()
- Runtime:
  - $O(n\ T_{insert} + n\ T_{dequeue\text{-}max})$

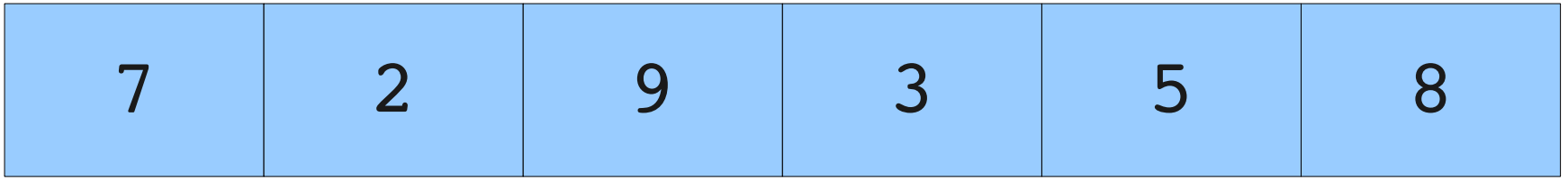
Our First Priority Queue

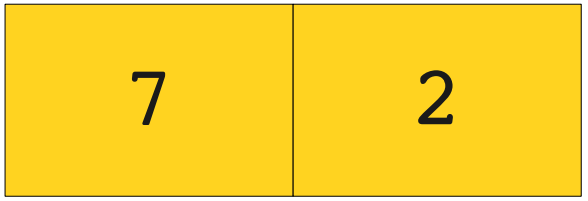
- Implement a priority queue as an **unsorted array**.
- To **insert** an element, append it to the array.
  - Runtime: **O(1)**
- To **dequeue-max**:
  - Locate the smallest element in the array.
  - Swap it to the last position in the array.
  - Remove the last element of the array.
  - Runtime: **Θ(n)**

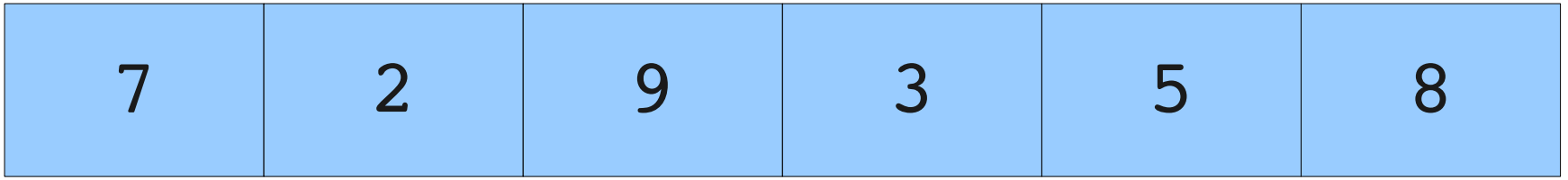

7
2
9
3
5
8


7
2
9
3
5
8

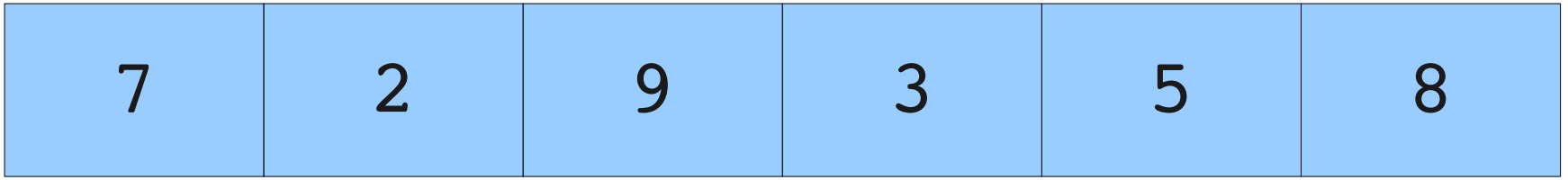

7

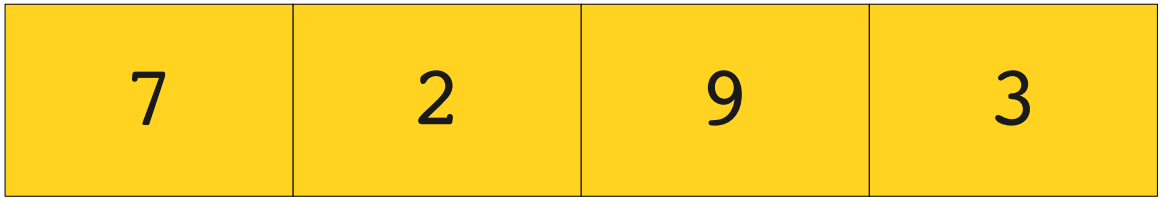

7
2
9
3
5
8

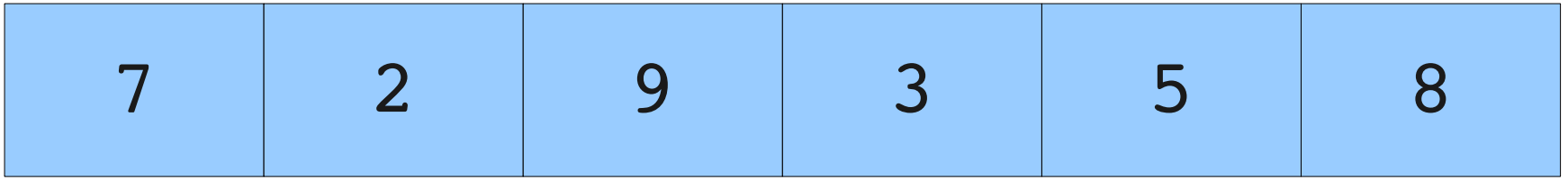

7
2


7
2
9
3
5
8

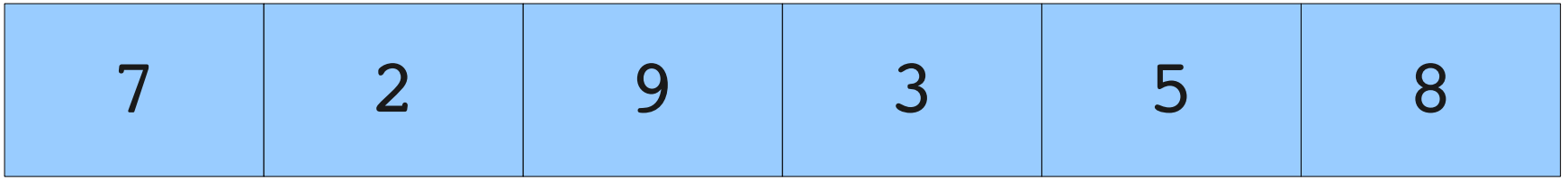

7
2
9

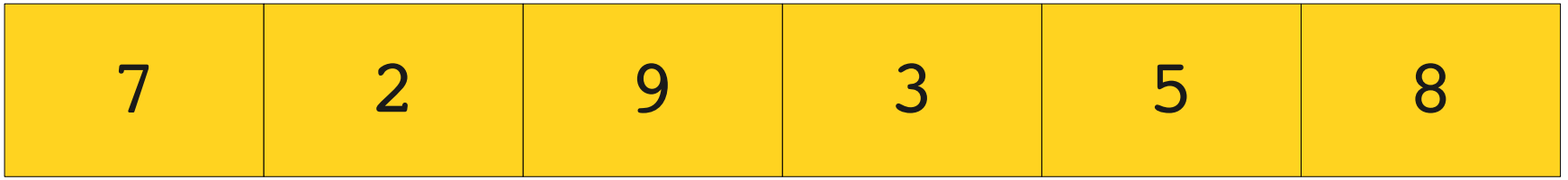

7
2
9
3
5
8

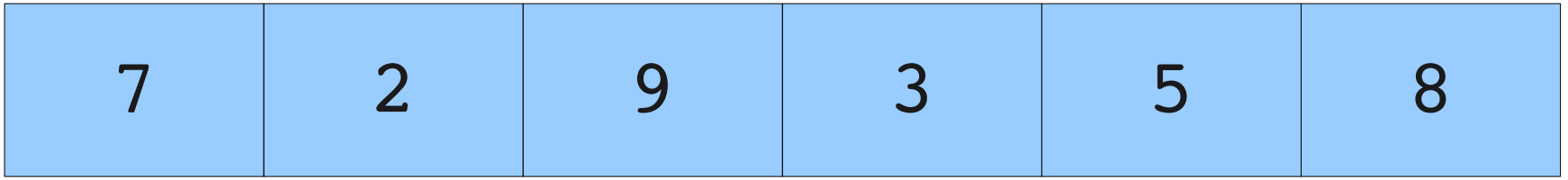

7
2
9
3

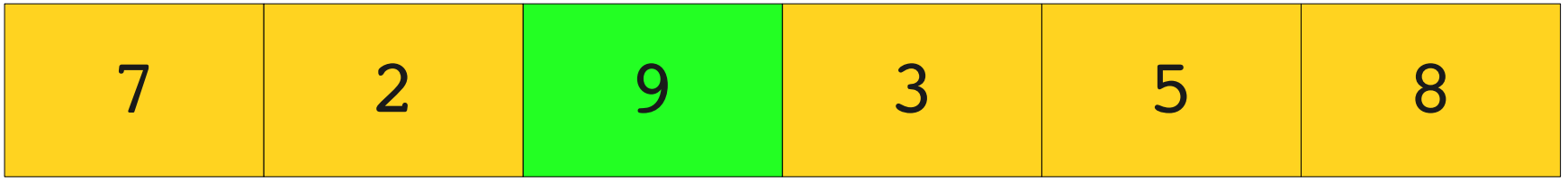

7
2
9
3
5
8

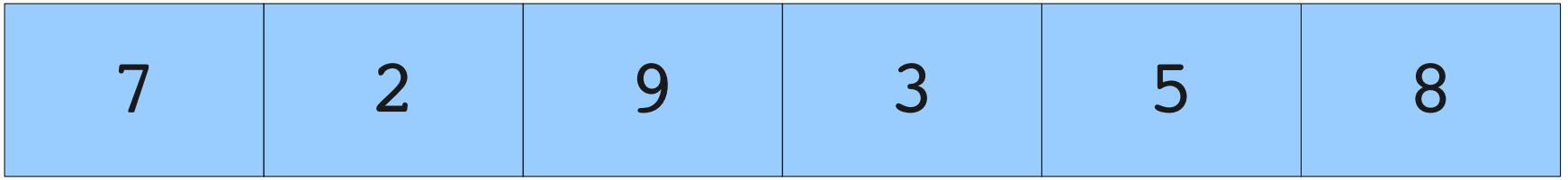

7
2
9
3
5

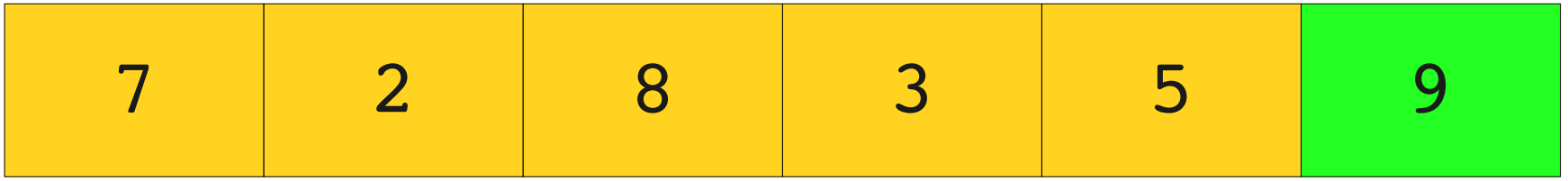

7
2
9
3
5
8

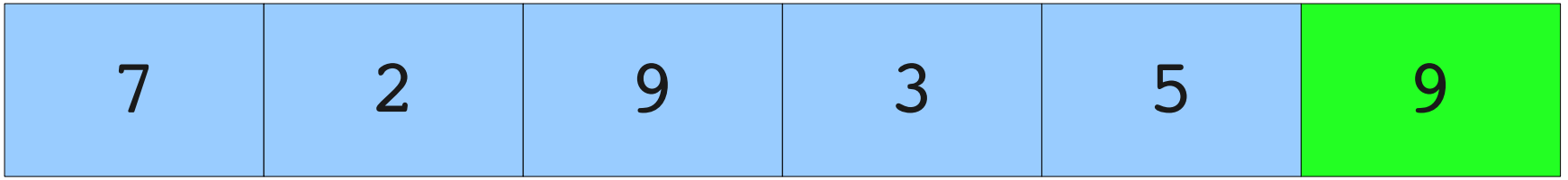

7
2
9
3
5
8


7
2
9
3
5
8

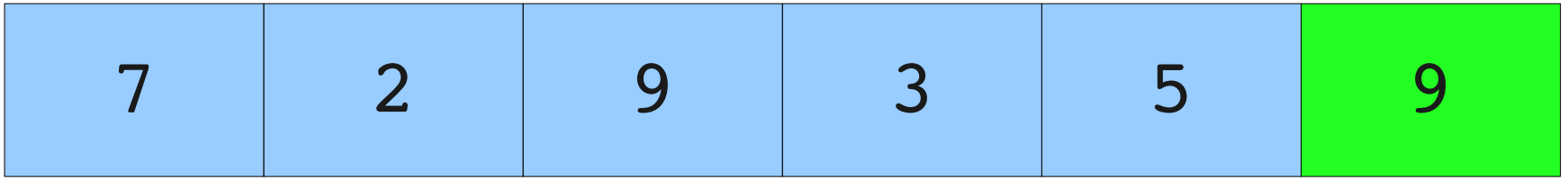

7
2
9
3
5
8

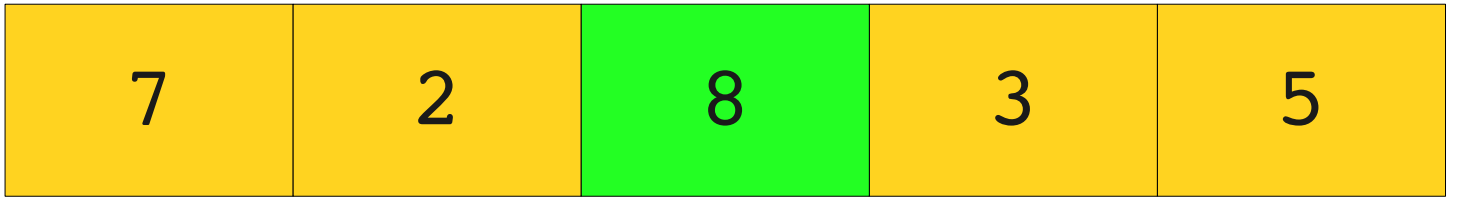

7
2
9
3
5
8

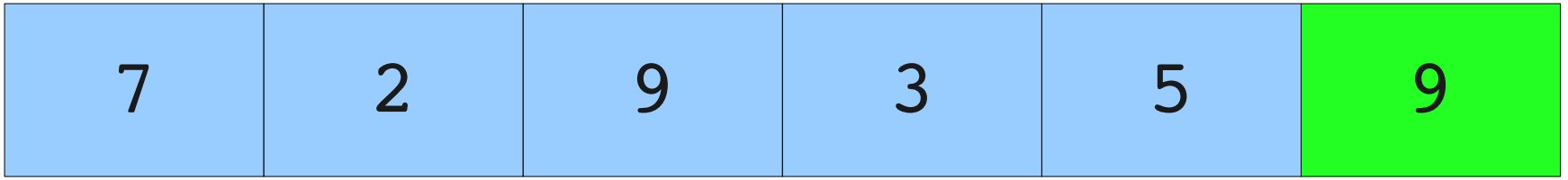

7
2
8
3
5
9


7
2
9
3
5
9

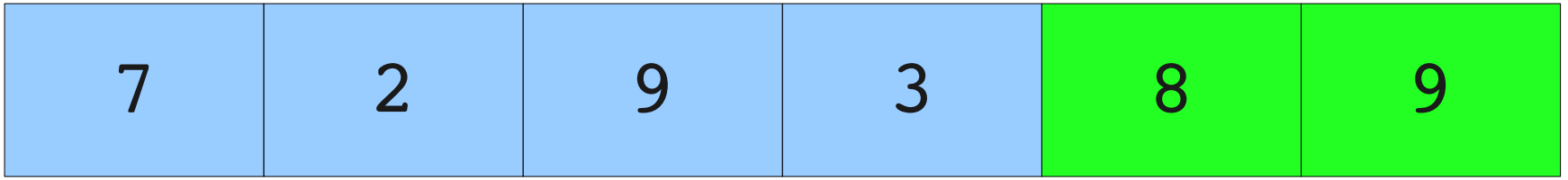

7
2
8
3
5

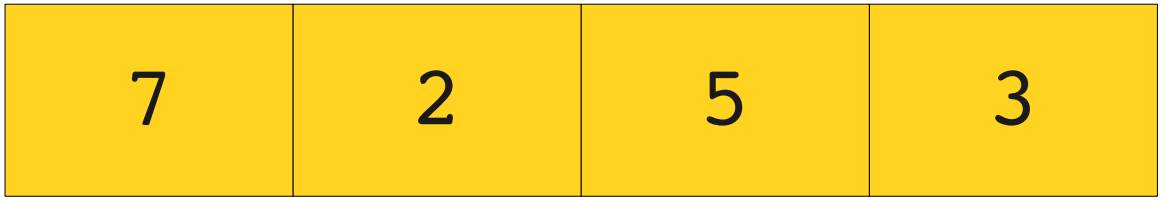

7
2
9
3
5
9

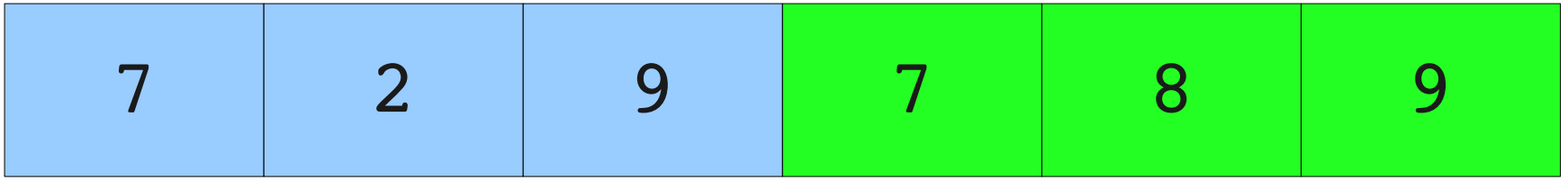

7
2
8
3
5


7
2
9
3
5
9

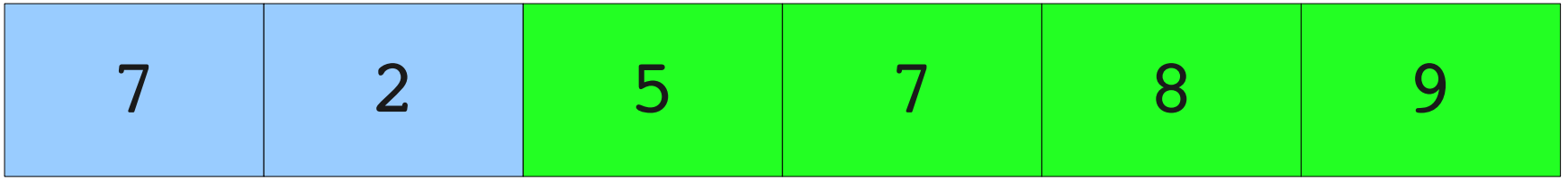

7
2
5
3
8

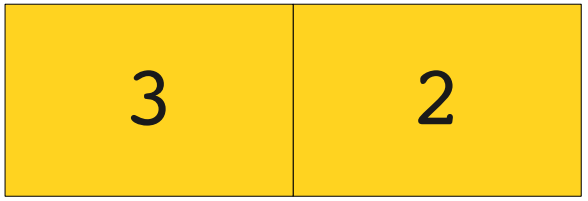

7
2
9
3
8
9

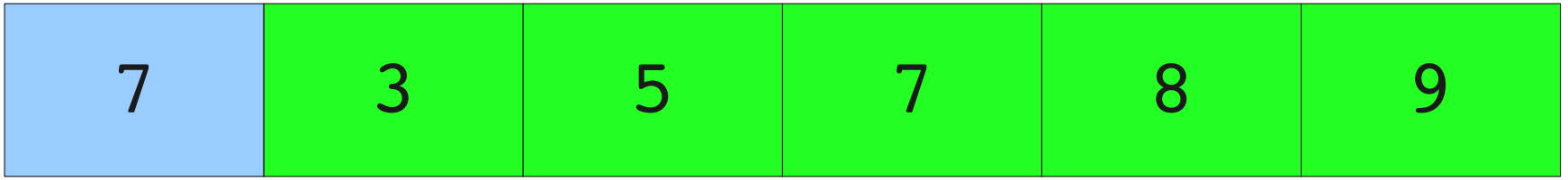

7
2
5
3


7
2
9
7
8
9

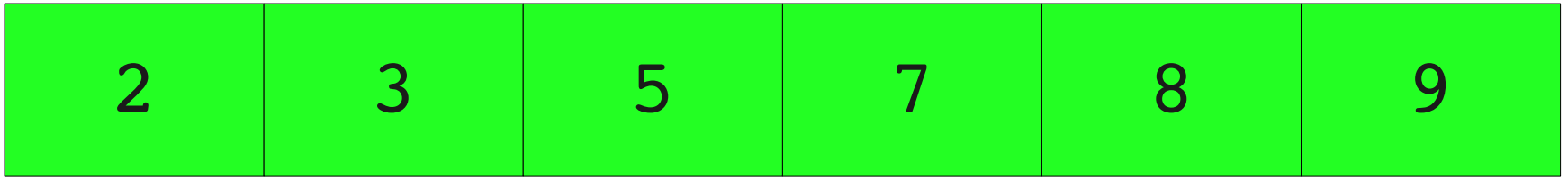

3
2
5


7
2
5
7
8
9

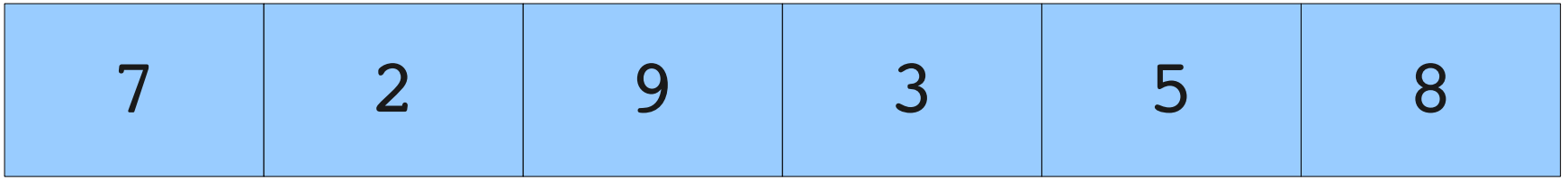

3
2

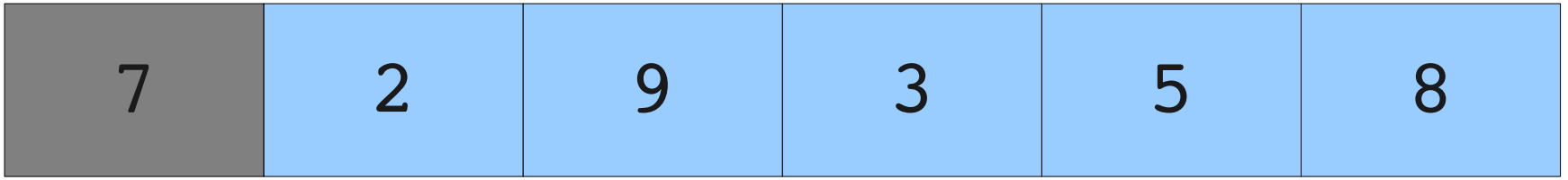

7
3
5
7
8
9


2

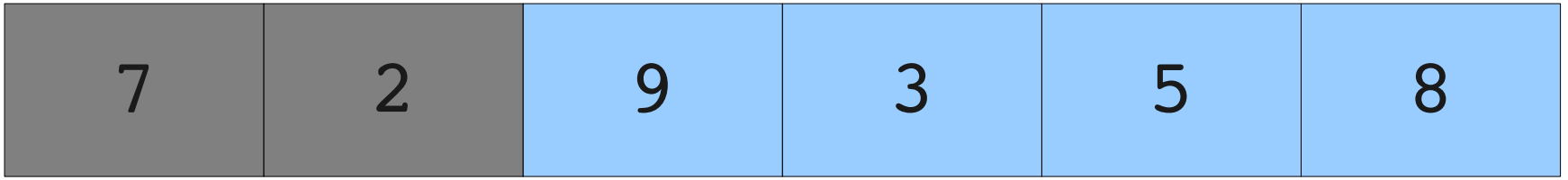

2
3
5
7
8
9

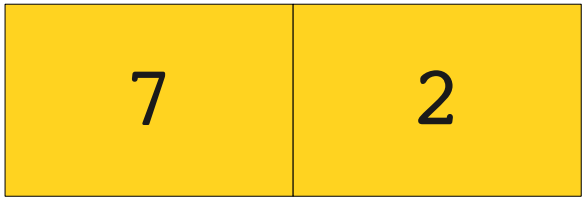
Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$

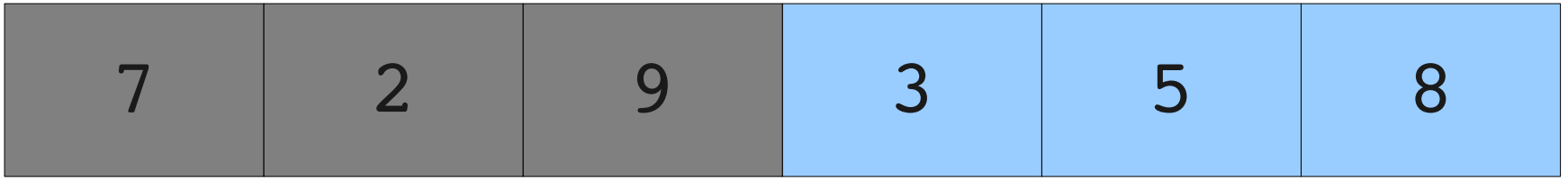

7
2
9
3
5
8


7
2
9
3
5
8

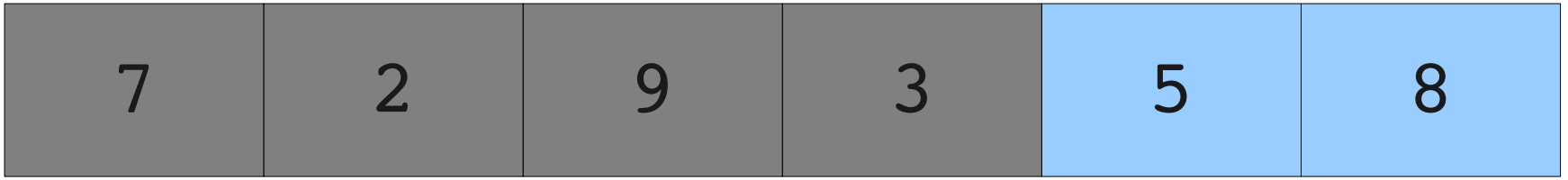

7

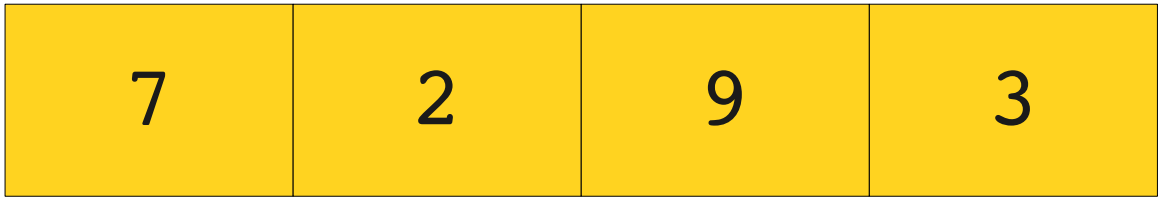

7
2
9
3
5
8

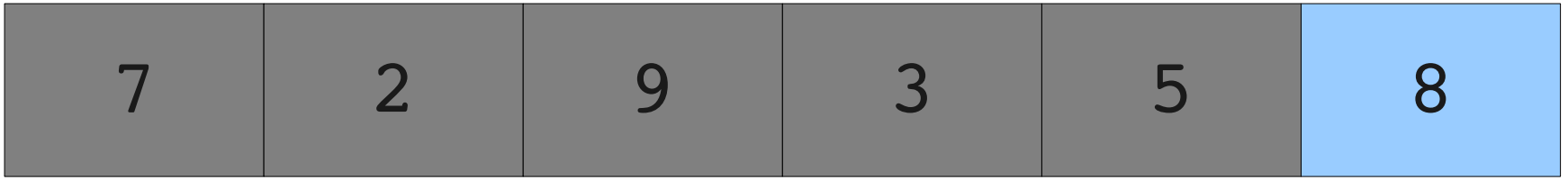

7
2


7
2
9
3
5
8

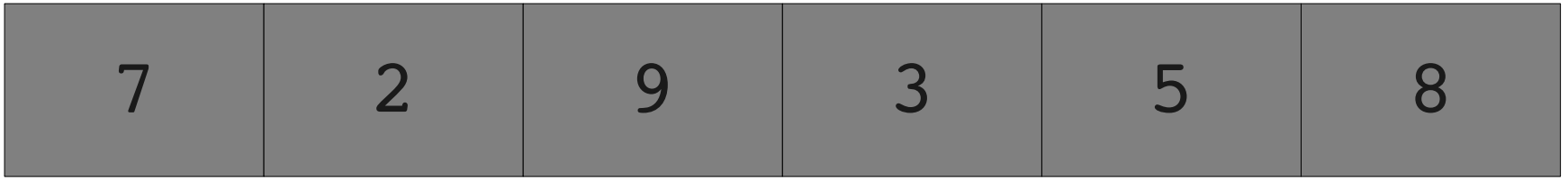

7
2
9

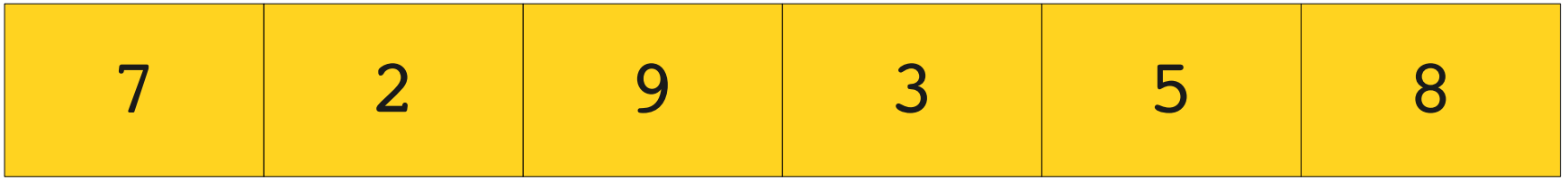

7
2
9
3
5
8

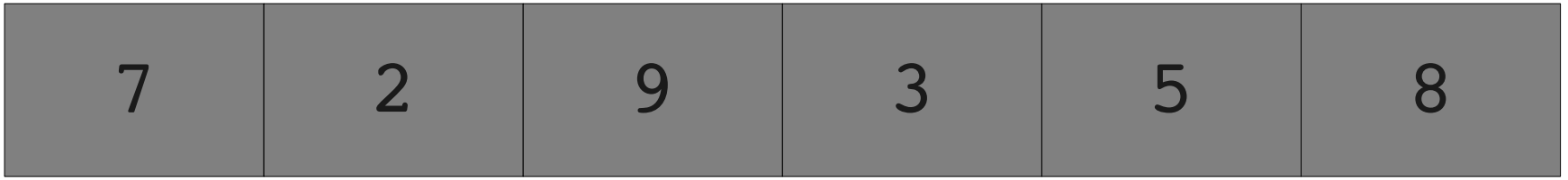

7
2
9
3

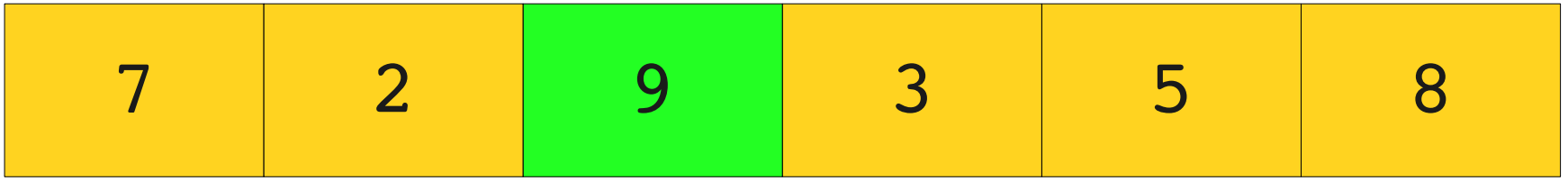

7
2
9
3
5
8

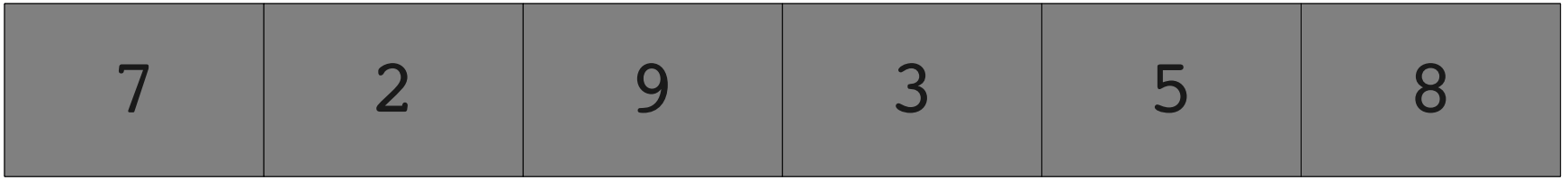

7
2
9
3
5

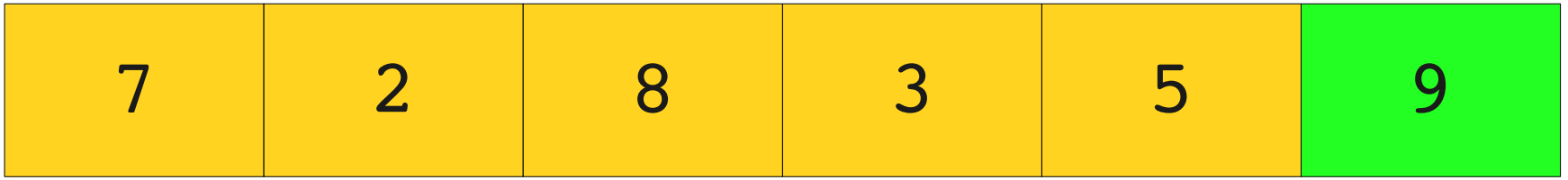

7
2
9
3
5
8

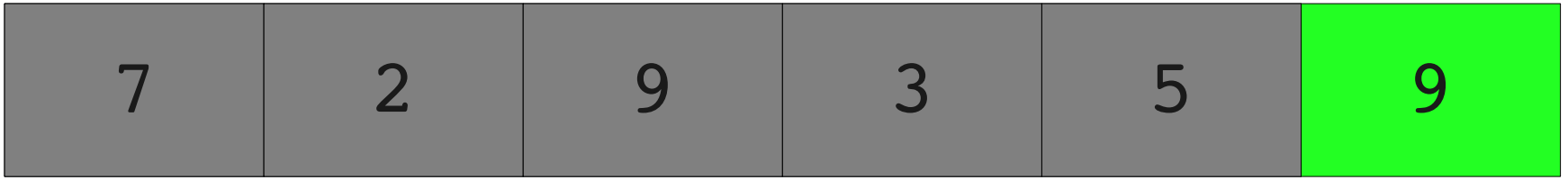

7
2
9
3
5
8


7
2
9
3
5
8

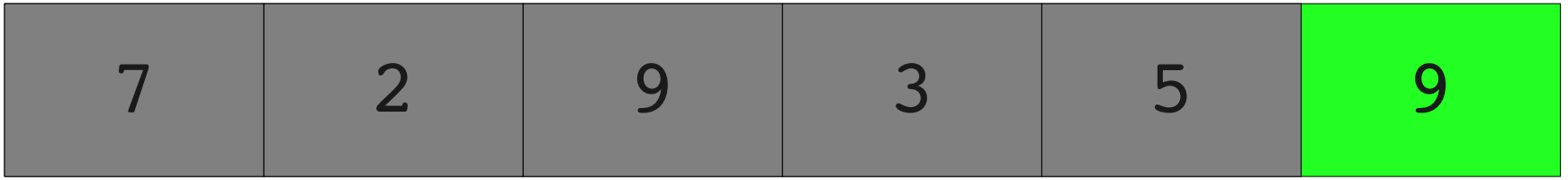

7
2
9
3
5
8

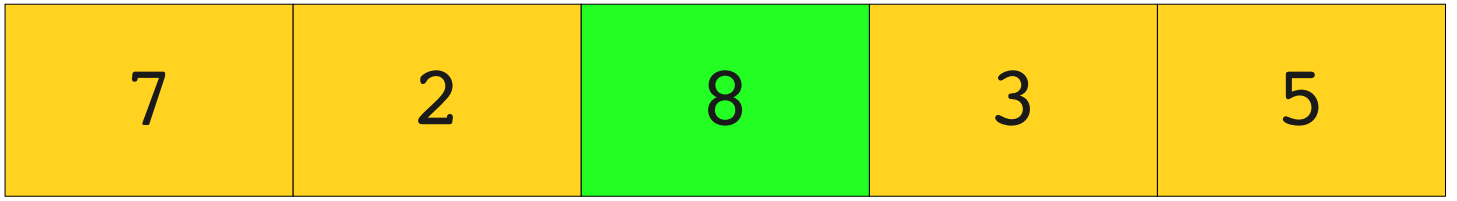

7
2
9
3
5
8

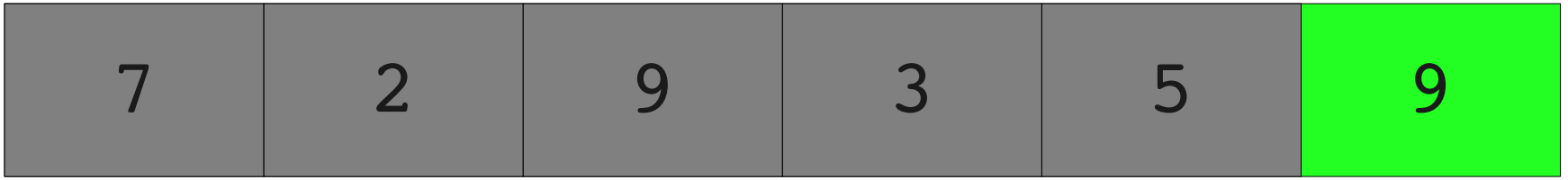

7
2
8
3
5
9


7
2
9
3
5
9

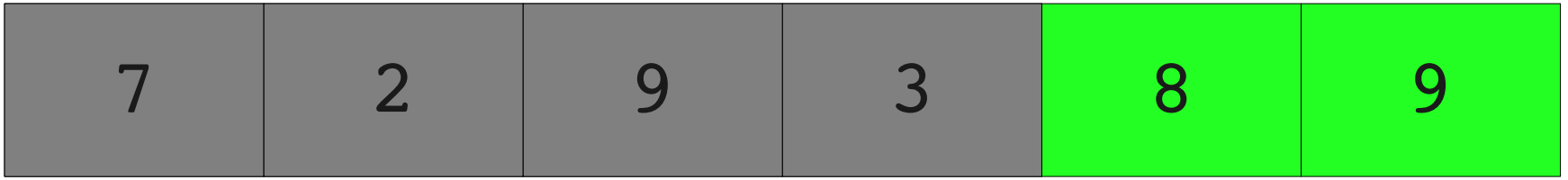

7
2
8
3
5

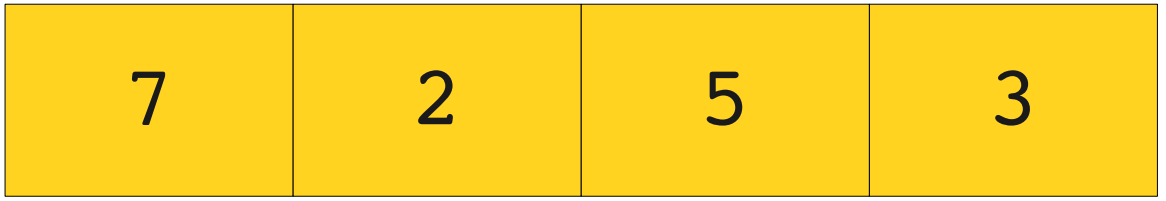

7
2
9
3
5
9

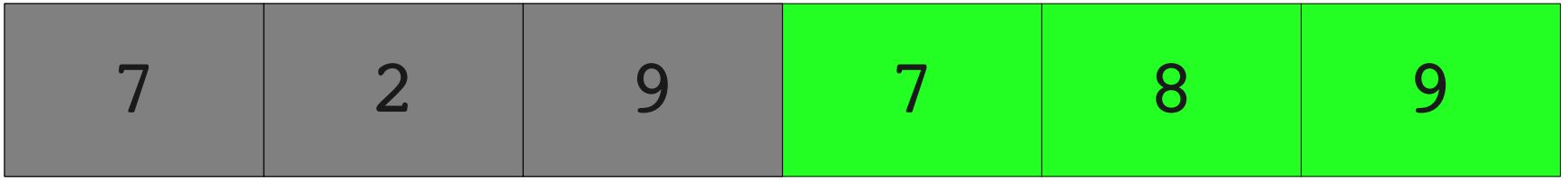

7
2
8
3
5


7
2
9
3
5
9

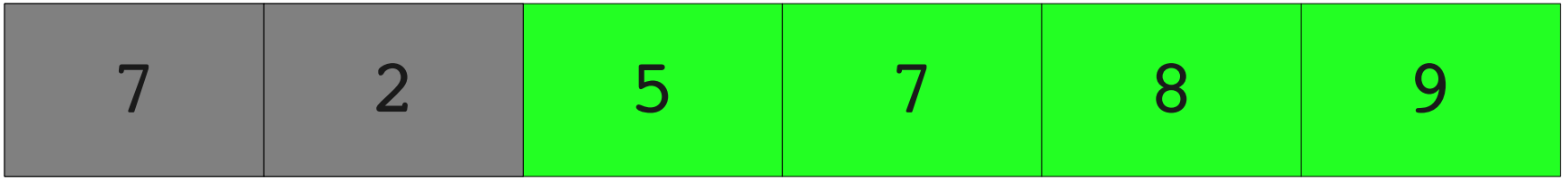

7
2
5
3
8

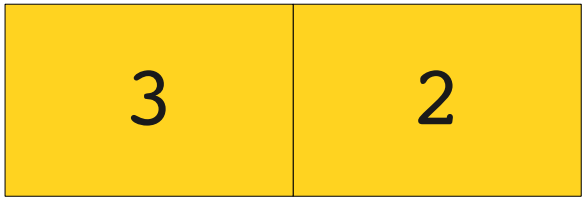

7
2
9
3
8
9

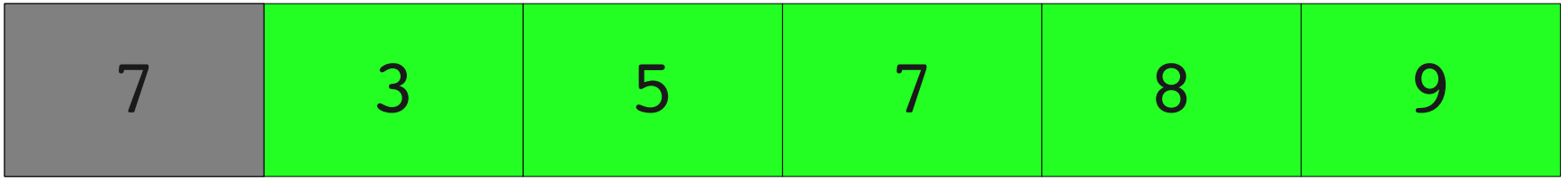

7
2
5
3


7
2
9
7
8
9

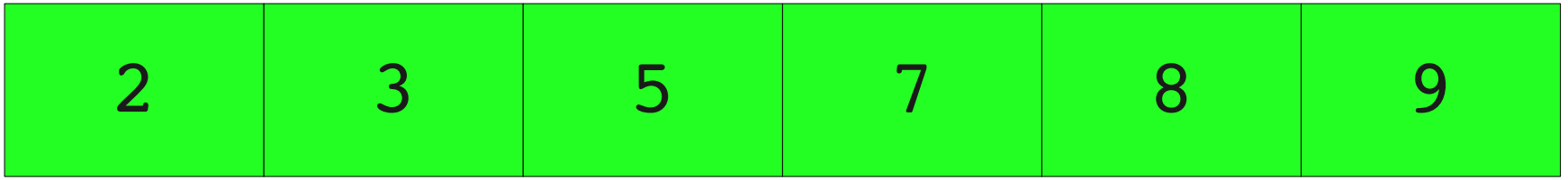

3
2
5


7
2
5
7
8
9

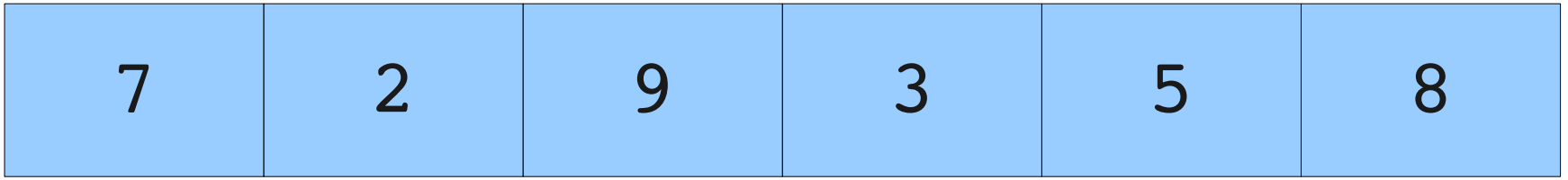

3
2

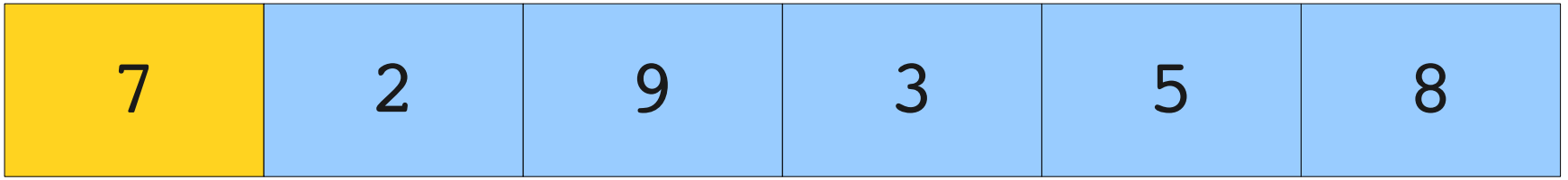

7
3
5
7
8
9

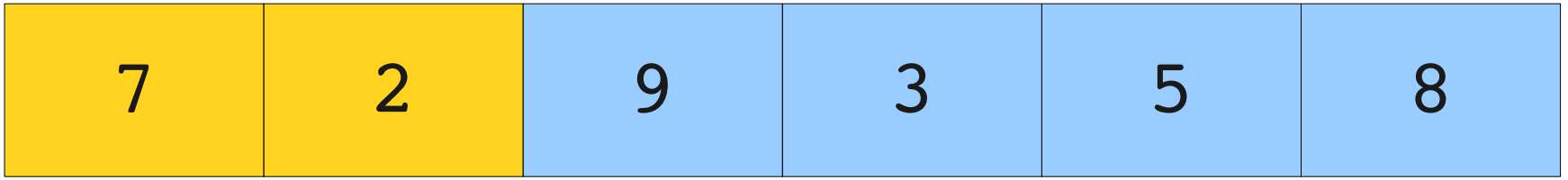

2

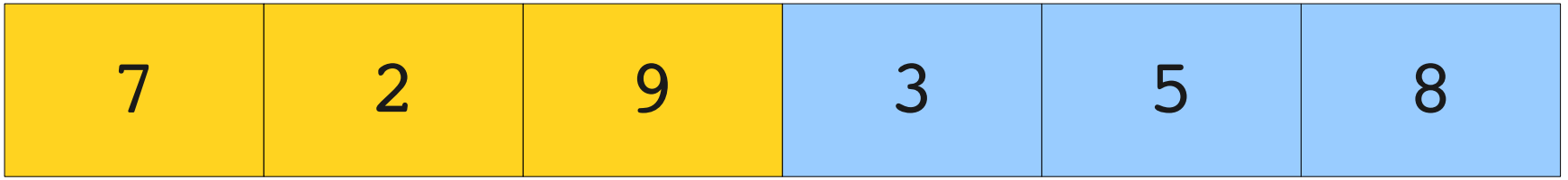

2
3
5
7
8
9

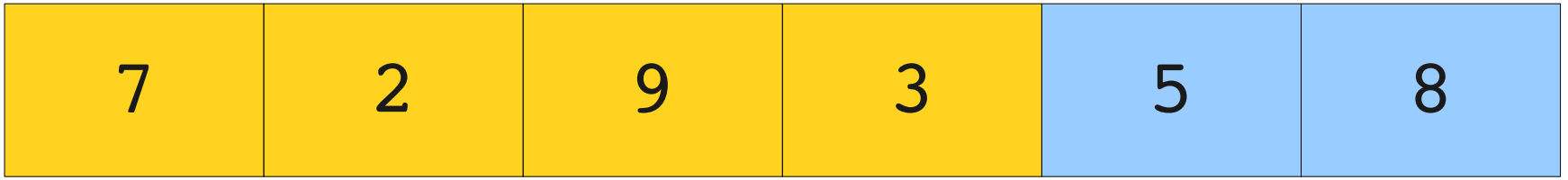
Key idea: **Make the priority queue implicit.**

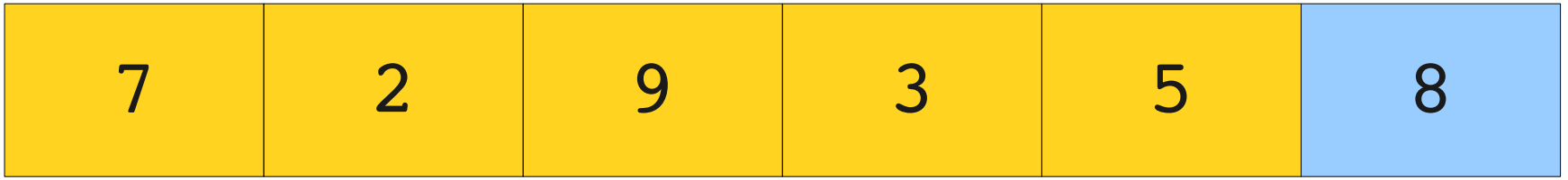

7
2
9
3
5
8

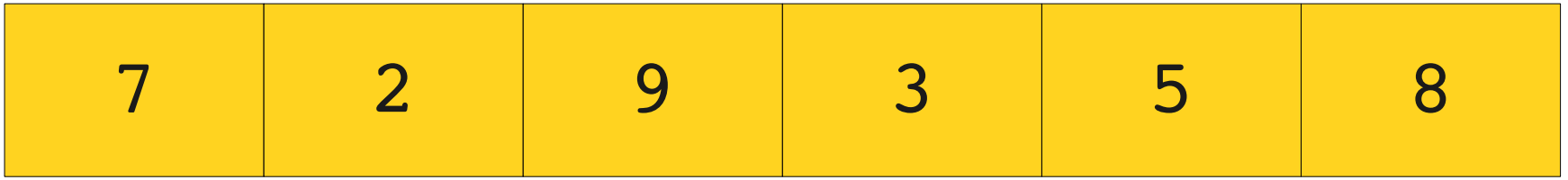

7
2
9
3
5
8

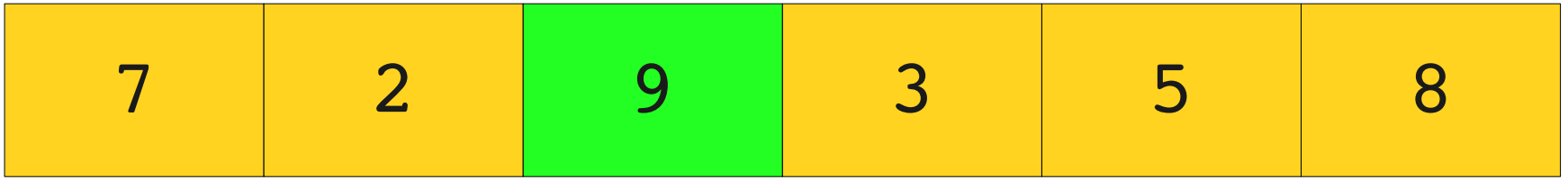

7
2
9
3
5
8

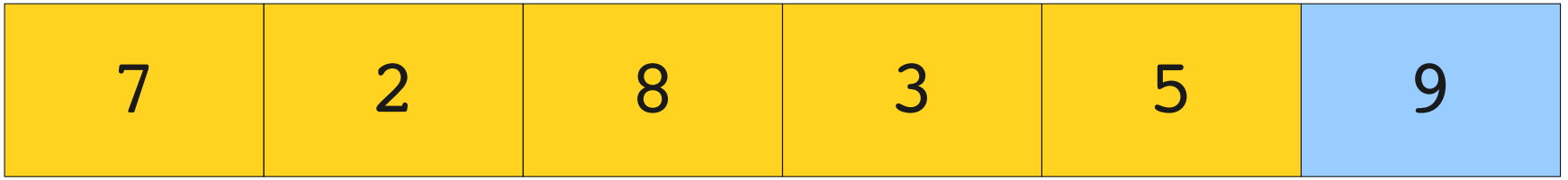

7
2
9
3
5
8

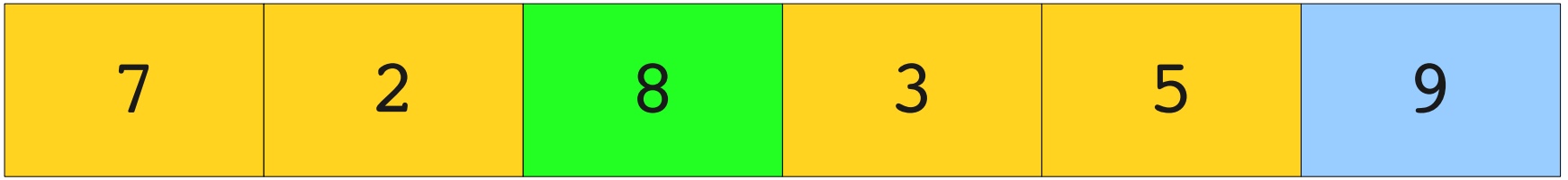

7
2
9
3
5
8

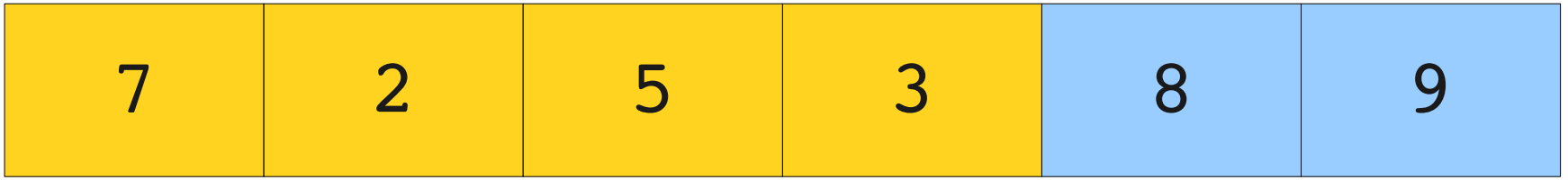

7
2
9
3
5
8

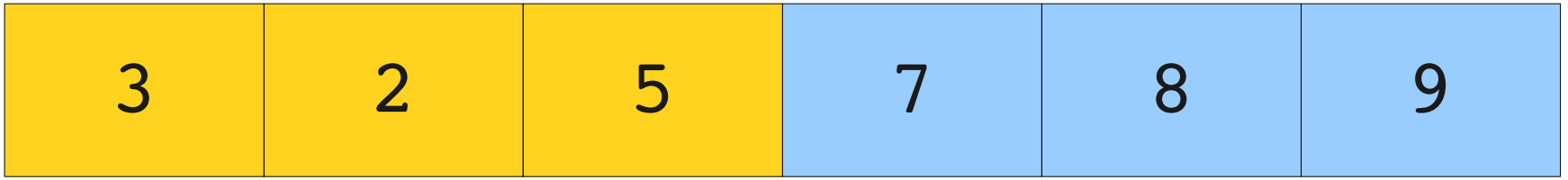

7
2
9
3
5
8

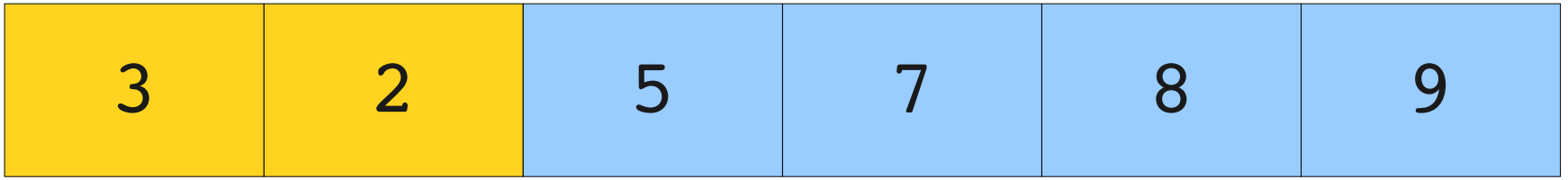

7
2
9
3
5
8

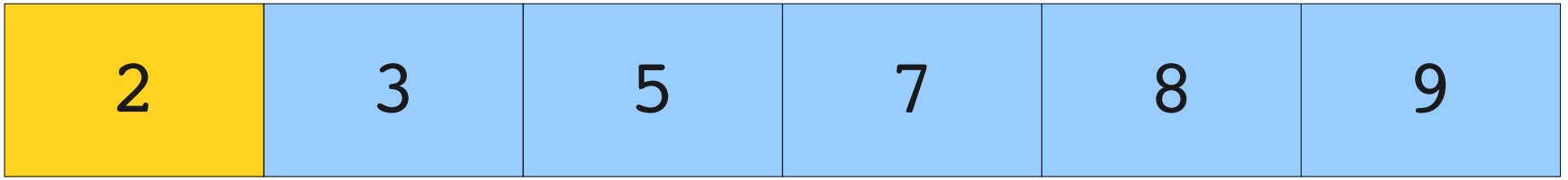

7
2
8
3
5
9

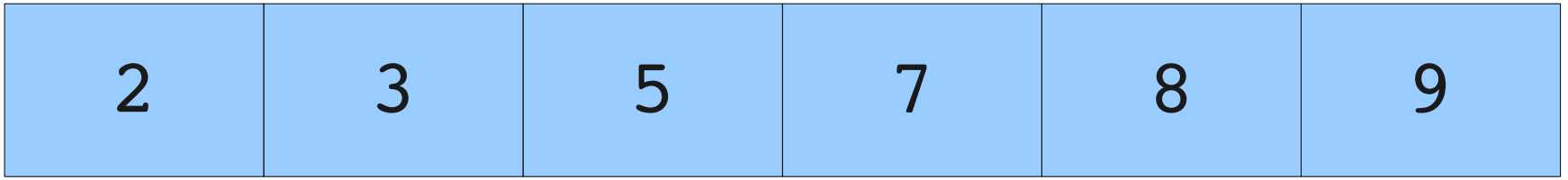

7
2
8
3
5
9


7
2
5
3
8
9


3
2
5
7
8
9


3
2
5
7
8
9

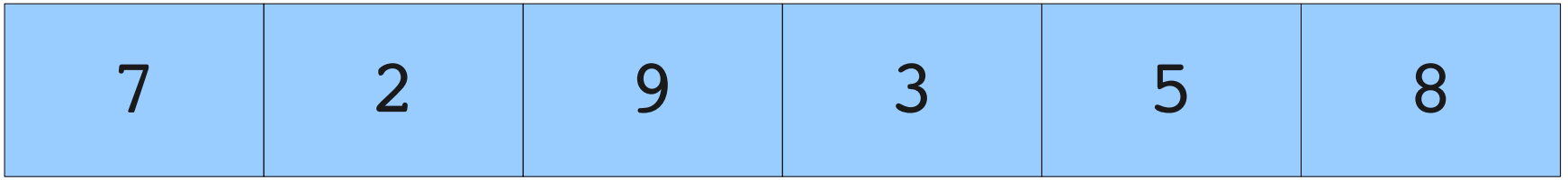

2
3
5
7
8
9

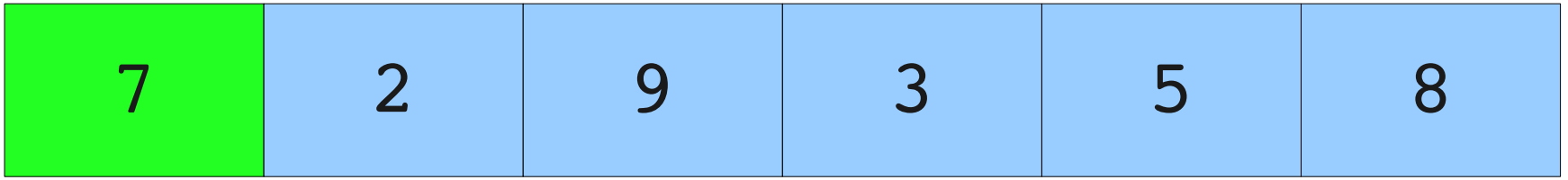

2
3
5
7
8
9

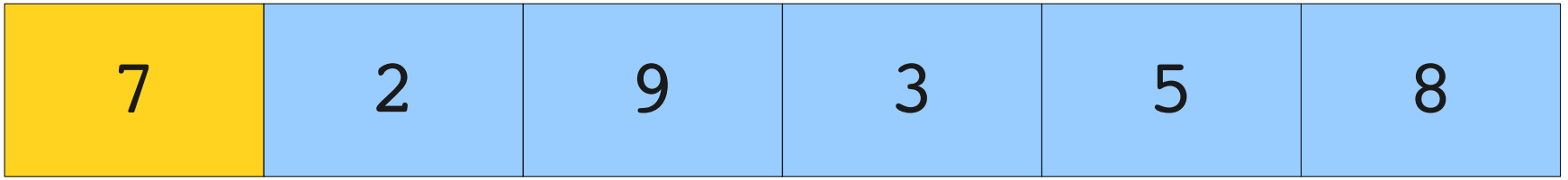
Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$

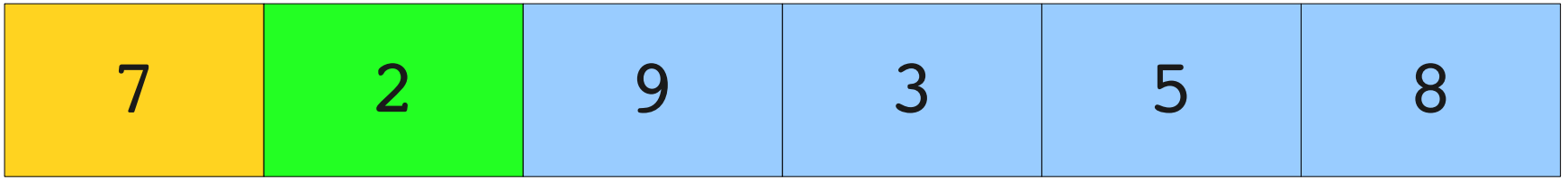
A Better Priority Queue

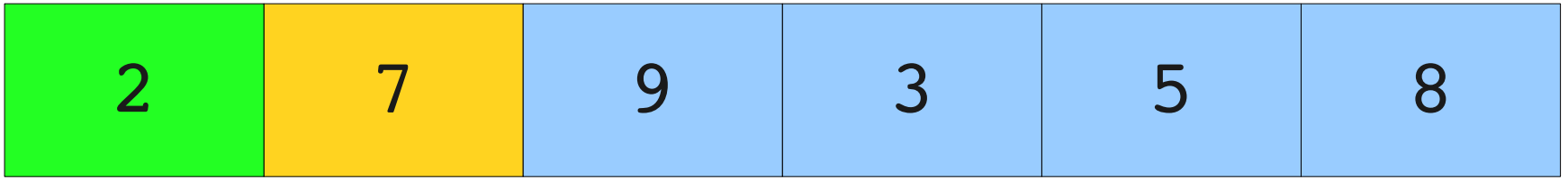
- Implement a priority queue as a **sorted array**:
- To **insert** an element:
  - Append it to the array.
  - “Bubble down” the element until it comes to rest.
  - Runtime: **O(1)** best-case, **O(n)** worst-case.
- To **dequeue-max**, remove the last element of the array.
  - Runtime: **O(1)**.

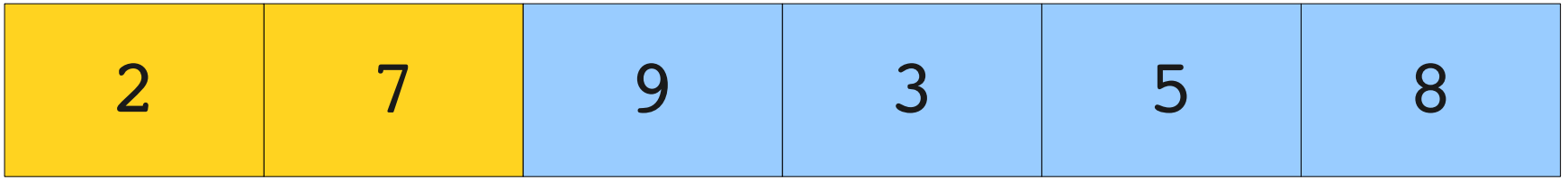

7
2
9
3
5
8

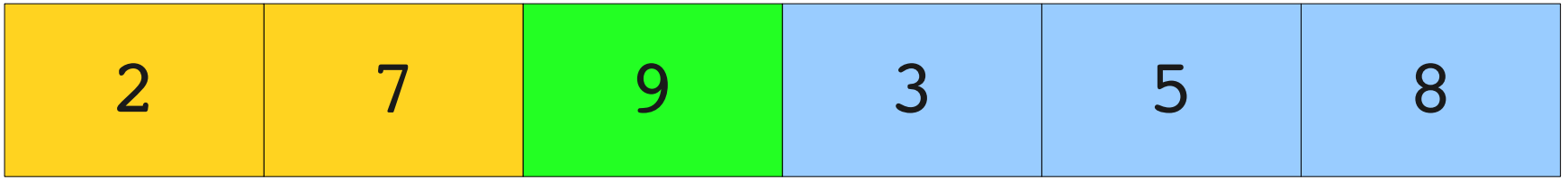

7
2
9
3
5
8

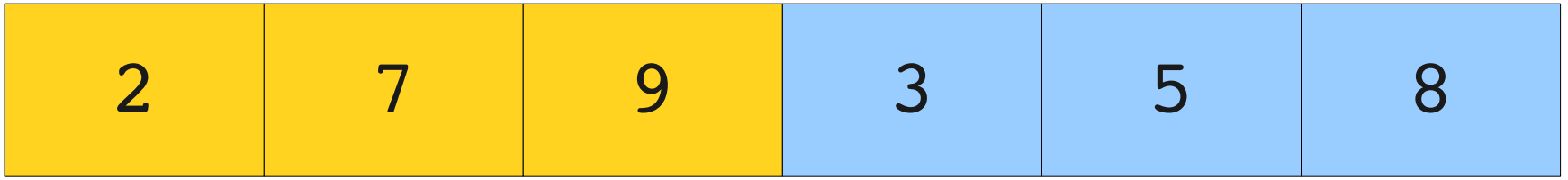

7
2
9
3
5
8

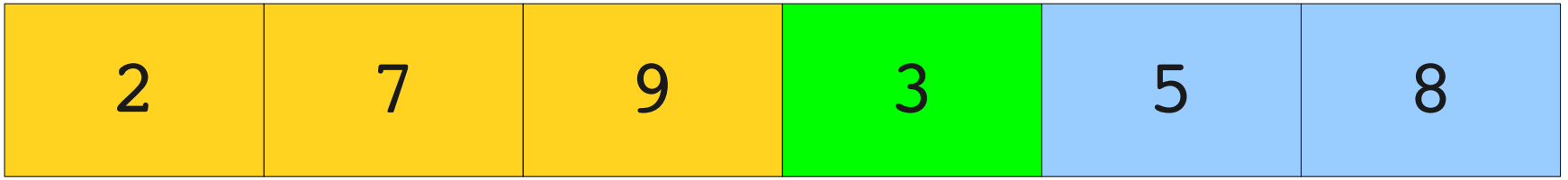

7
2
9
3
5
8

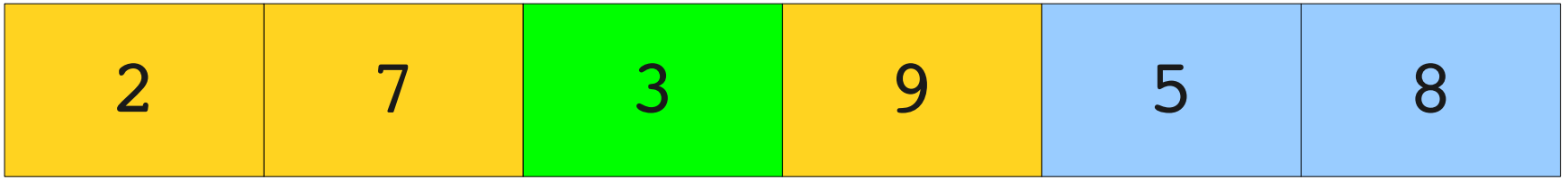

2
7
9
3
5
8

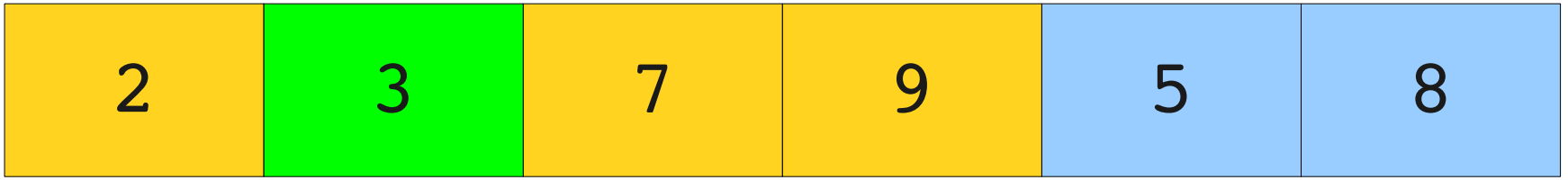

2
7
9
3
5
8

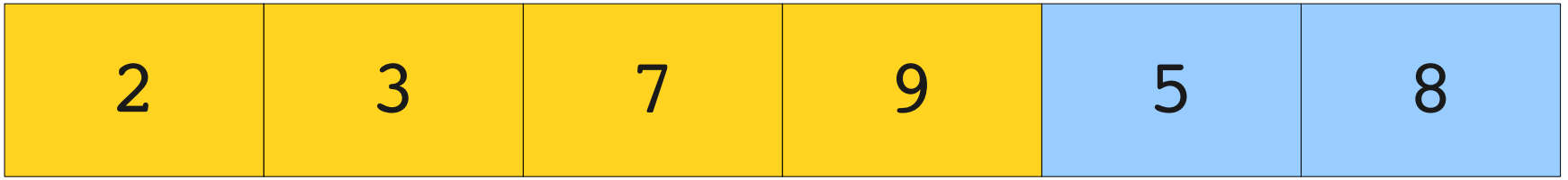

2
7
9
3
5
8

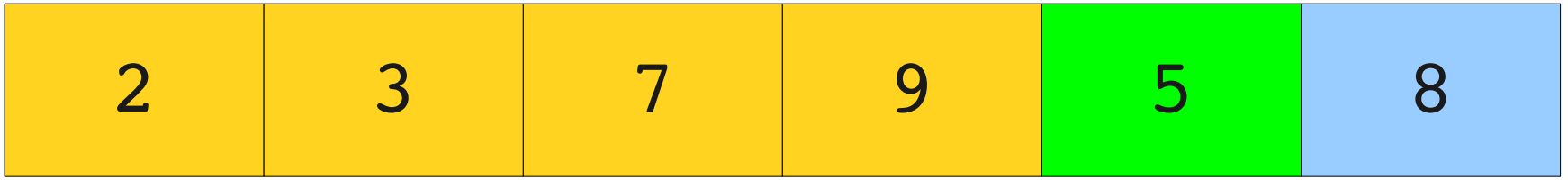

2
7
9
3
5
8

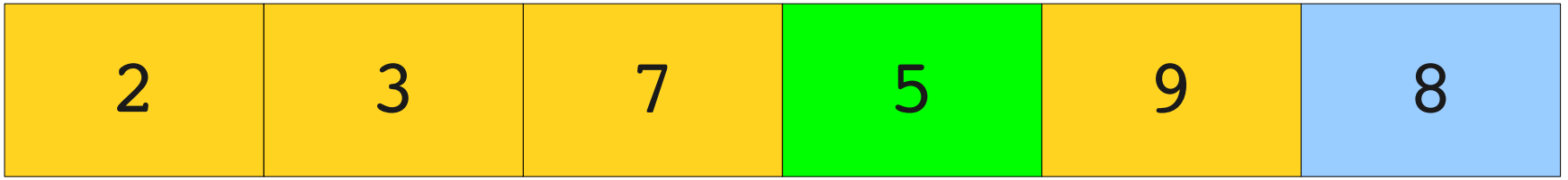

2
7
9
3
5
8

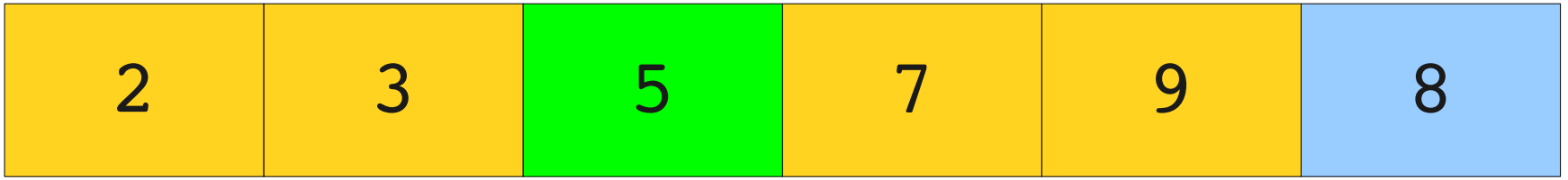

2
7
3
9
5
8

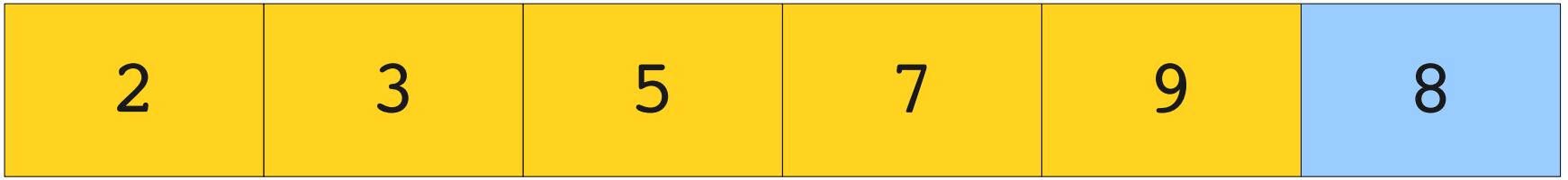

2
3
7
9
5
8

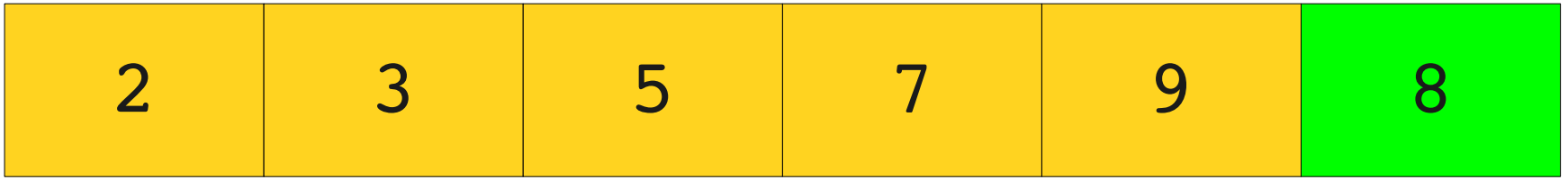

2
3
7
9
5
8

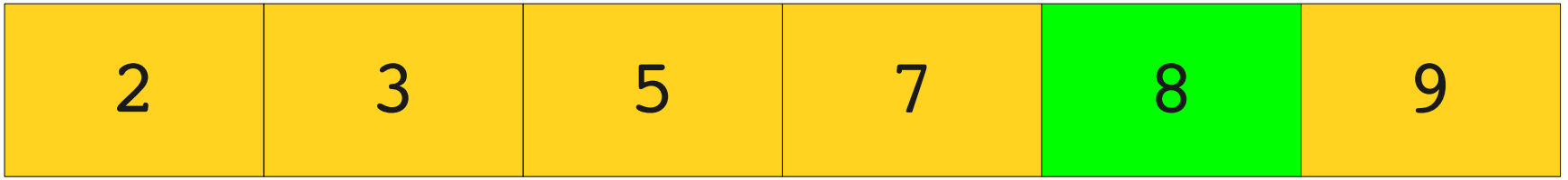

2
3
7
9
5
8

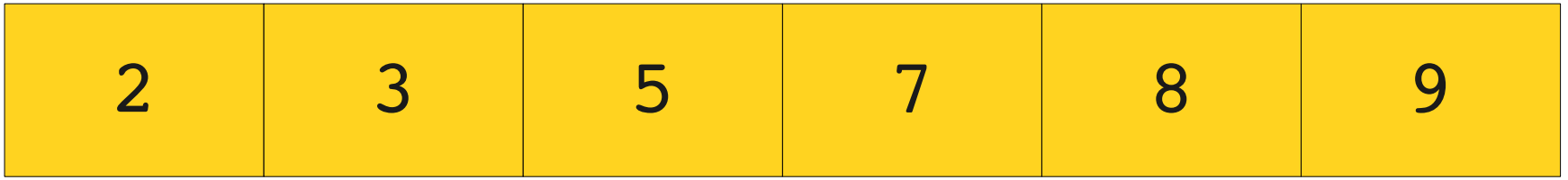

2
3
7
5
9
8

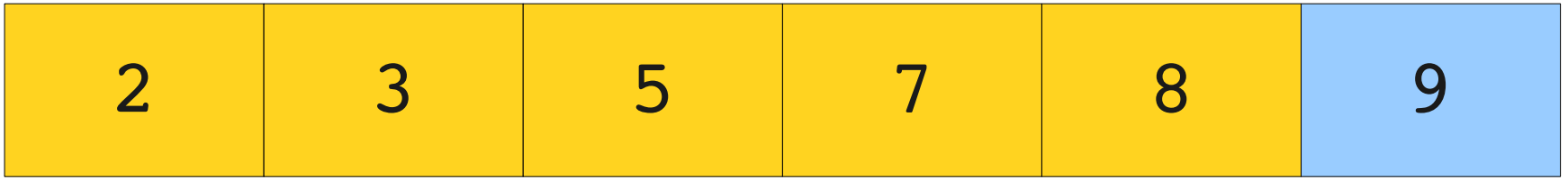

2
3
5
7
9
8

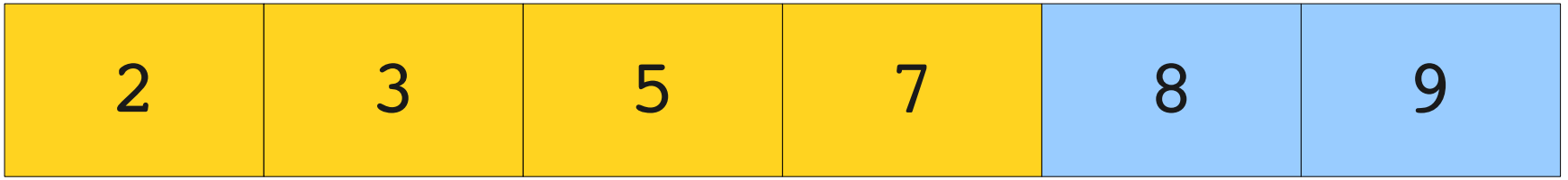

2
3
5
7
9
8

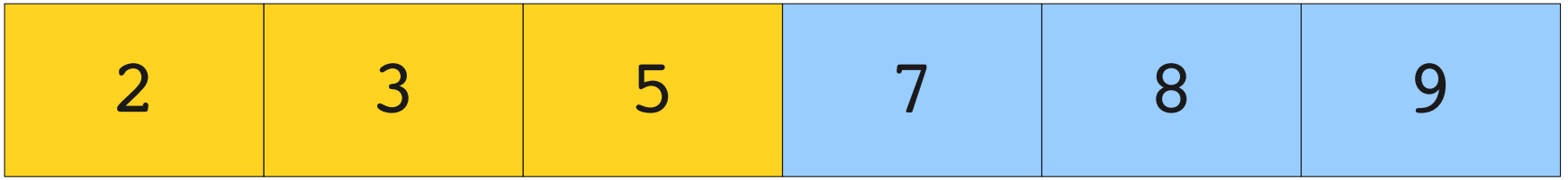

2
3
5
7
9
8

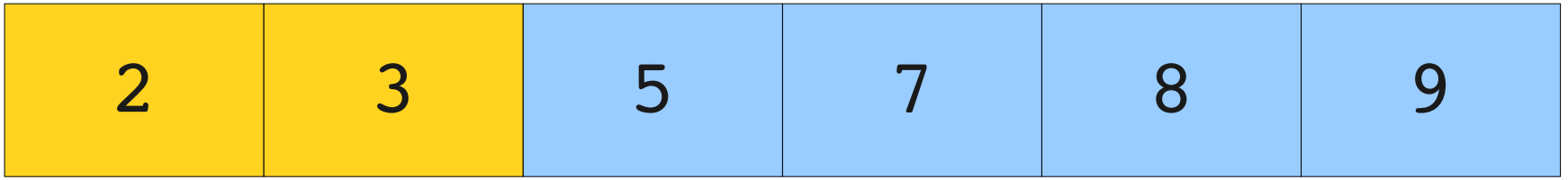

2
3
5
7
8
9

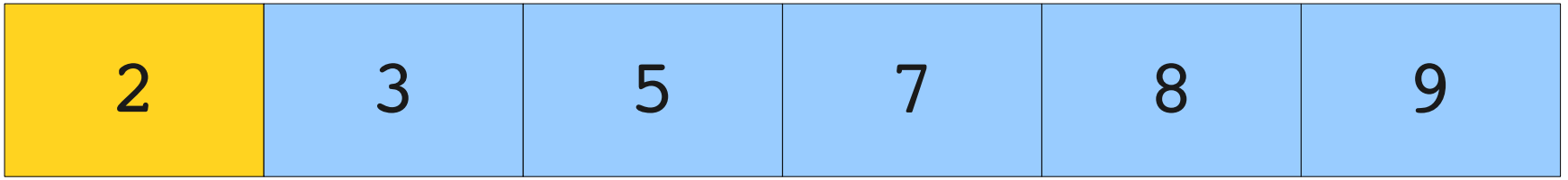

2
3
5
7
8
9

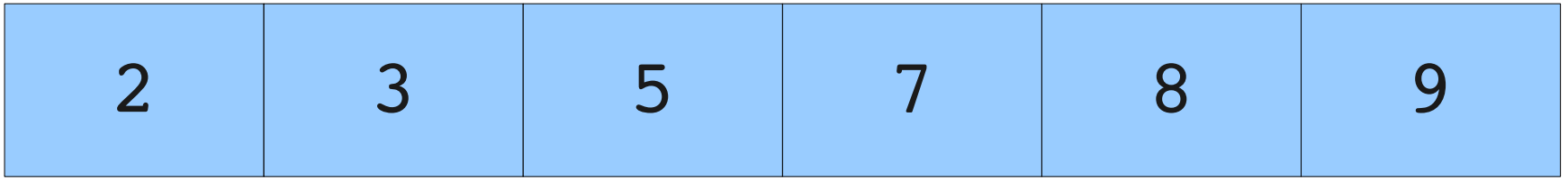

2
3
5
7
8
9


2
3
5
7
8
9


2
3
5
7
8
9

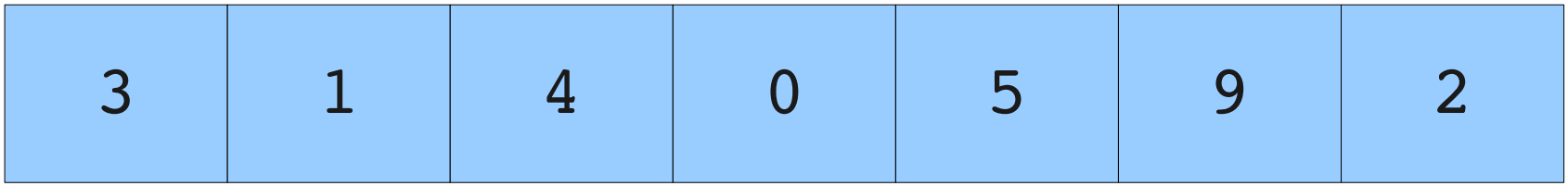

2
3
5
7
8
9


2
3
5
7
8
9

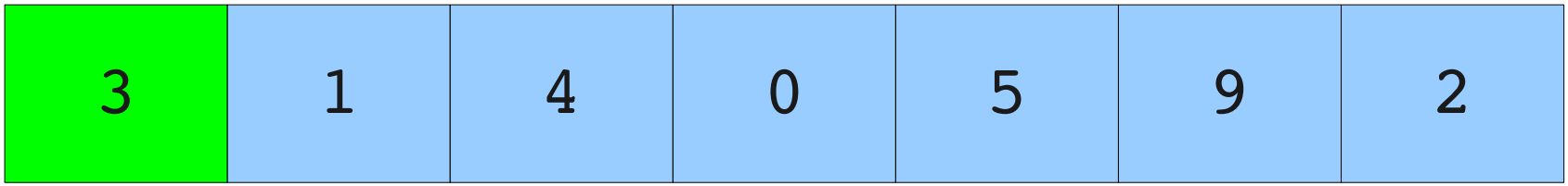

2
3
5
7
8
9

Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$

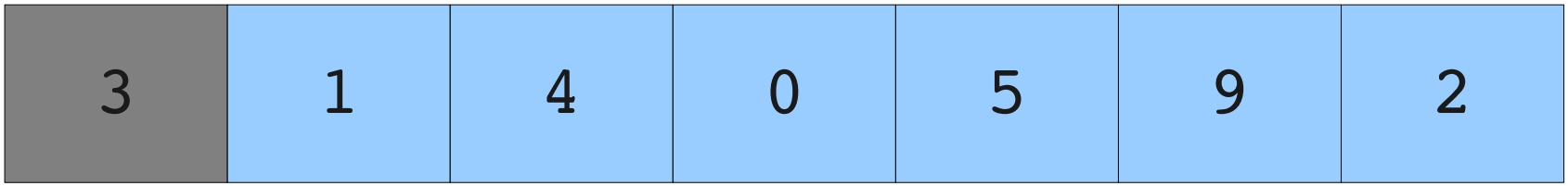
Sorting with Binary Heaps



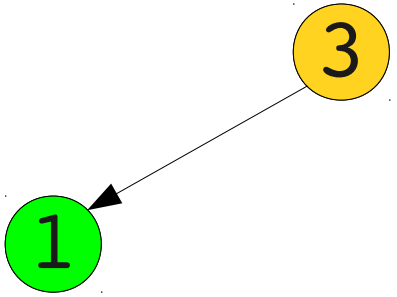
Sorting with Binary Heaps



Sorting with Binary Heaps

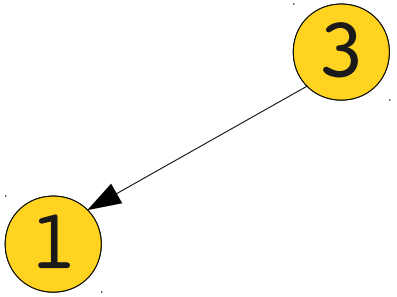


Sorting with Binary Heaps



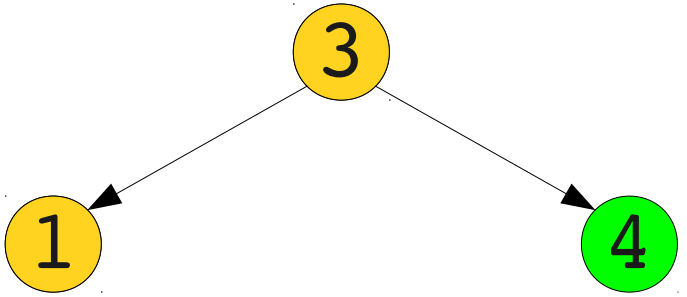
3	1	4	0	5	9	2

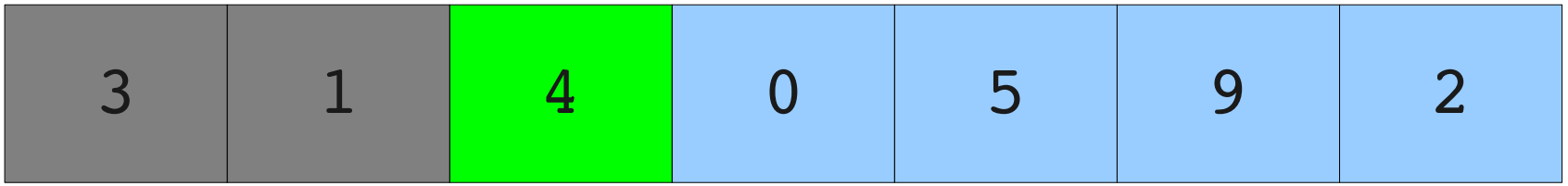
Sorting with Binary Heaps



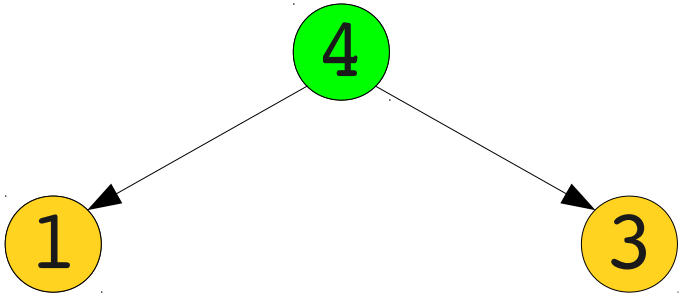
3	1	4	0	5	9	2

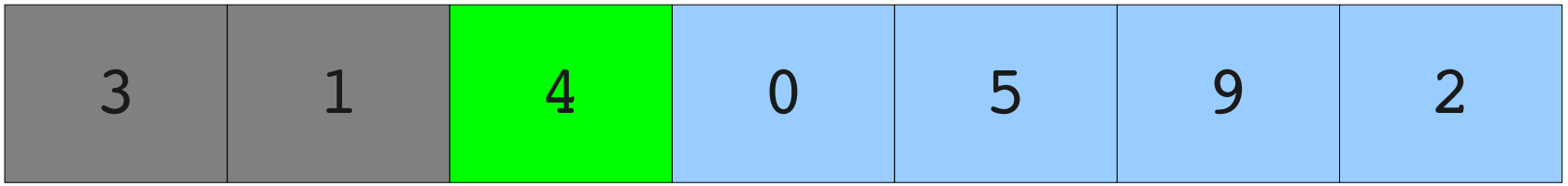
Sorting with Binary Heaps



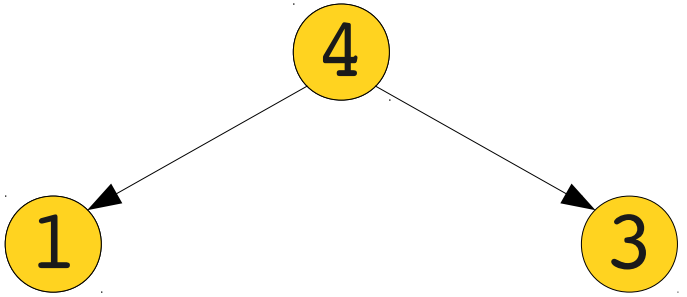


Sorting with Binary Heaps



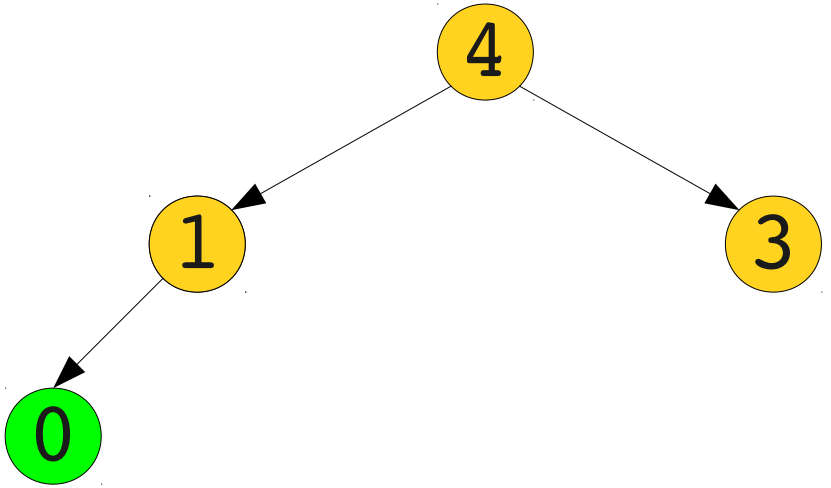


Sorting with Binary Heaps



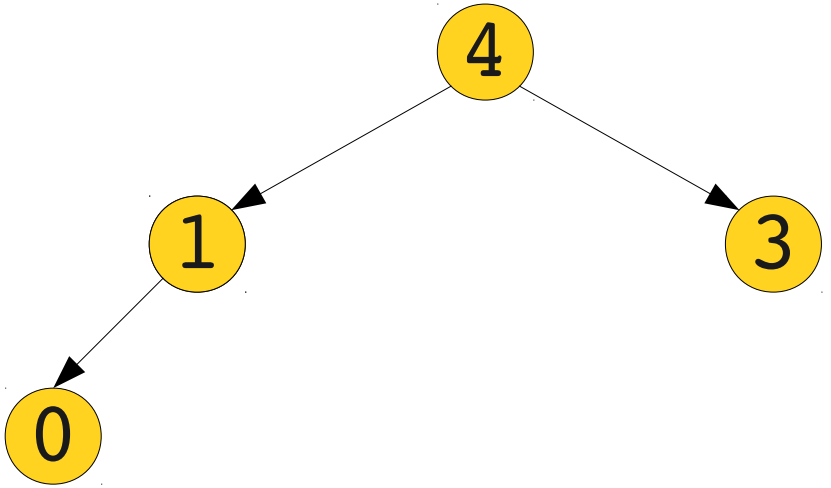
3	1	4	0	5	9	2

Sorting with Binary Heaps



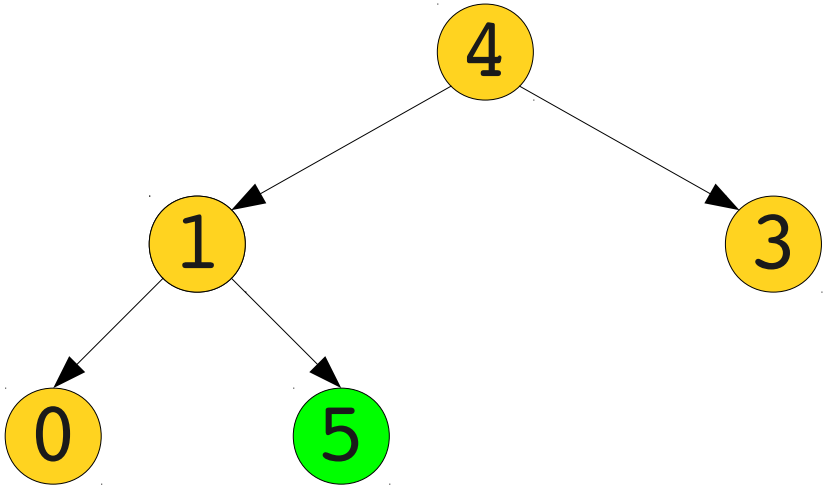
3	1	4	0	5	9	2

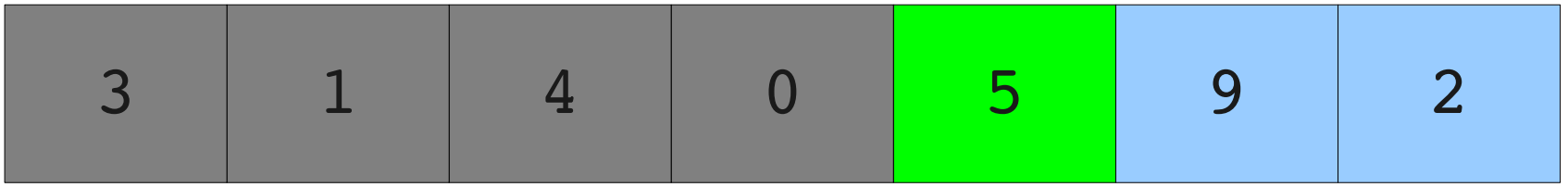
Sorting with Binary Heaps



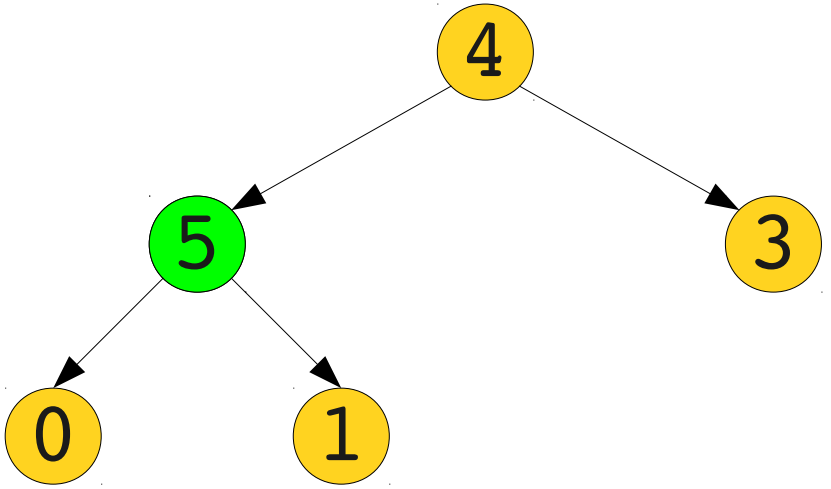
3	1	4	0	5	9	2

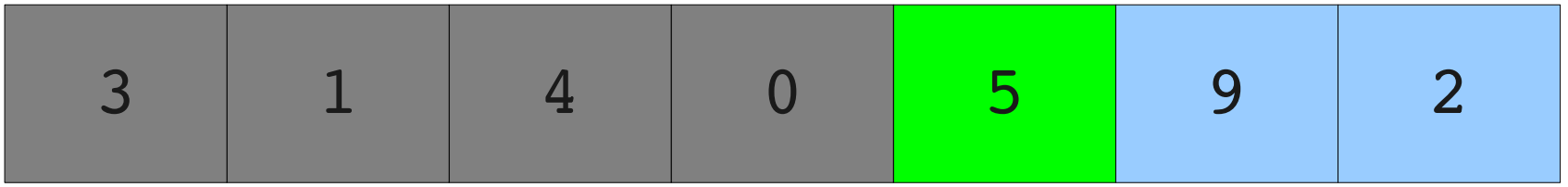
Sorting with Binary Heaps



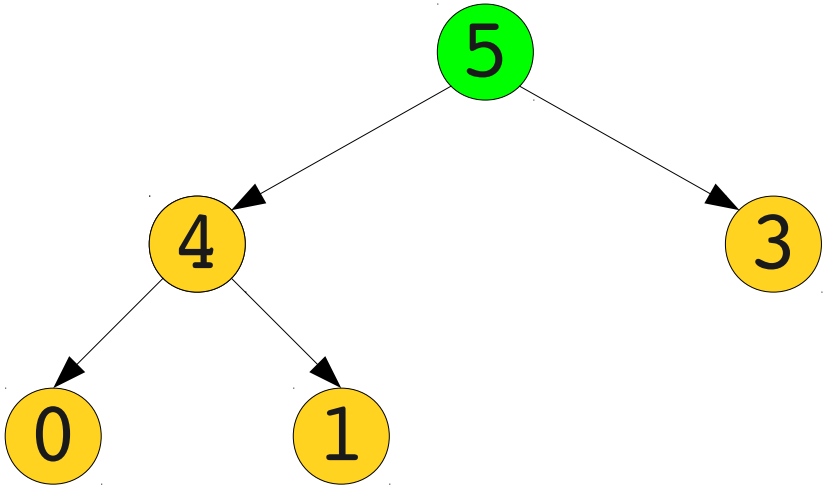


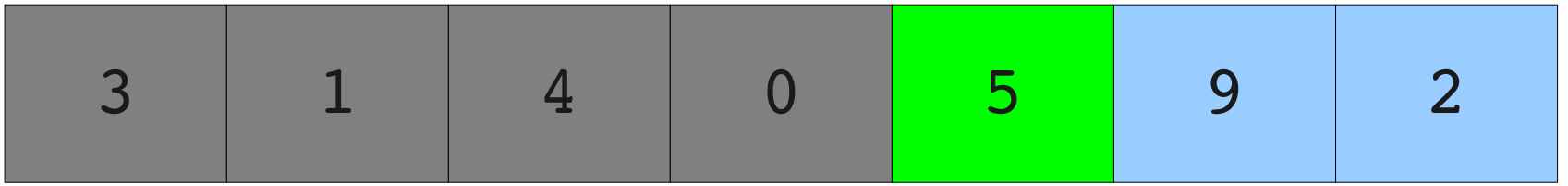
Sorting with Binary Heaps



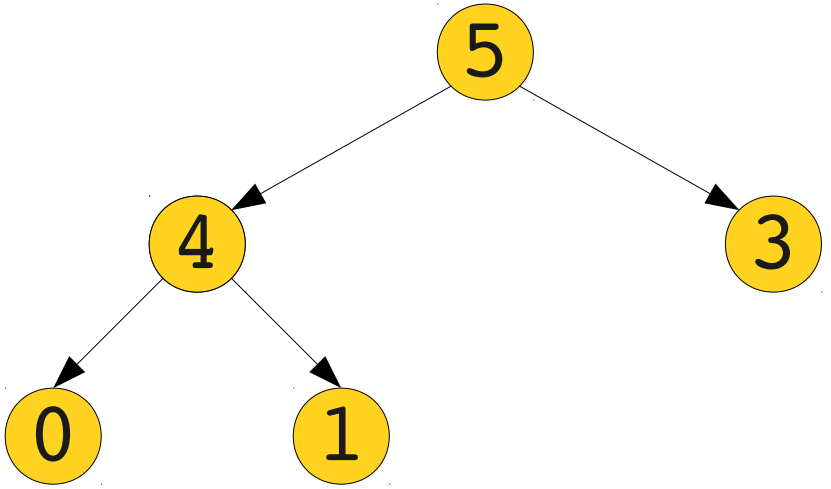


Sorting with Binary Heaps



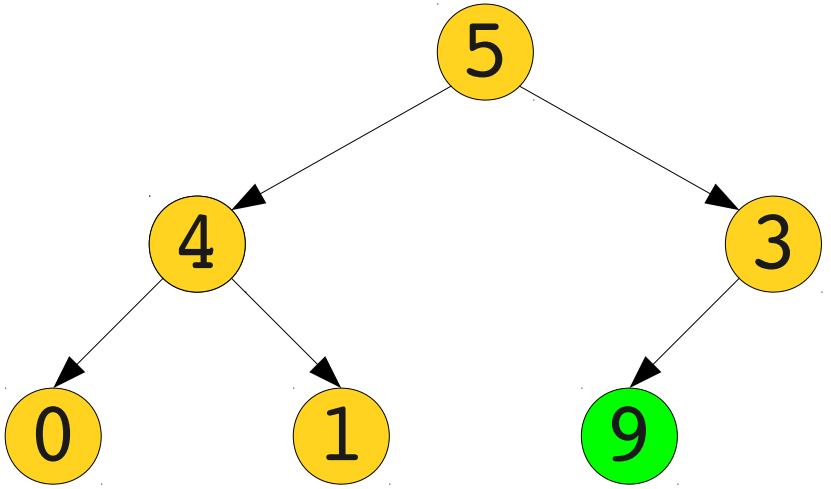


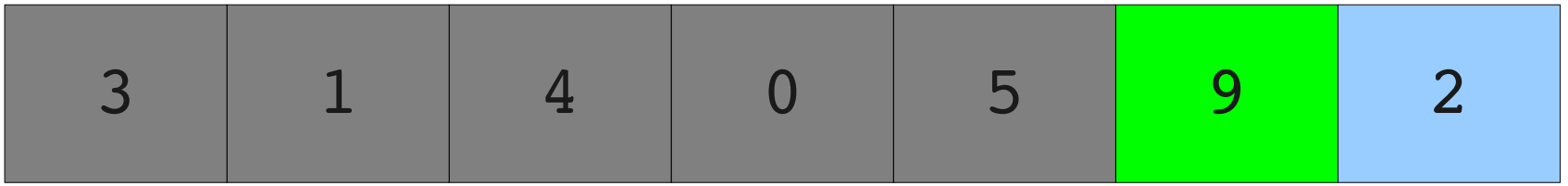
Sorting with Binary Heaps



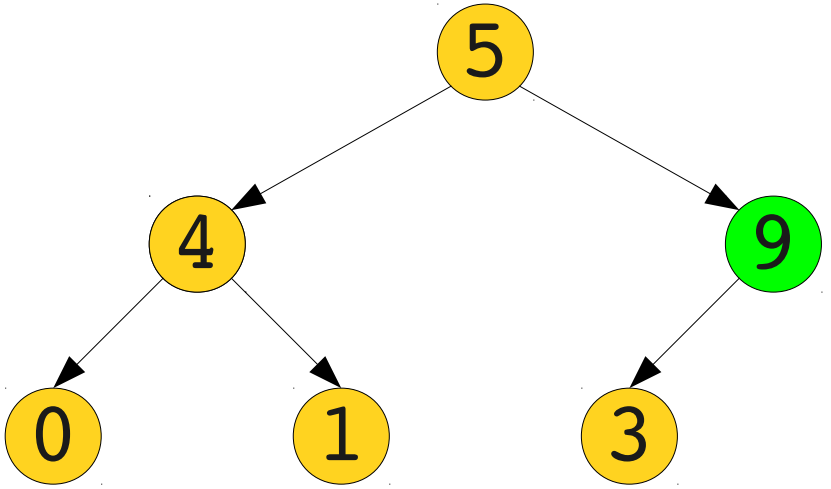
3	1	4	0	5	9	2

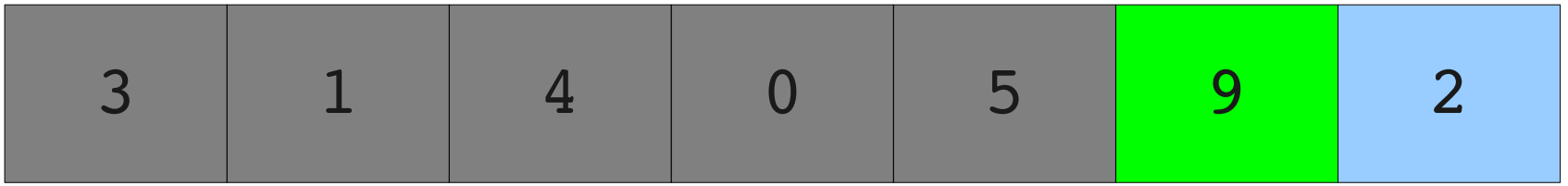
Sorting with Binary Heaps



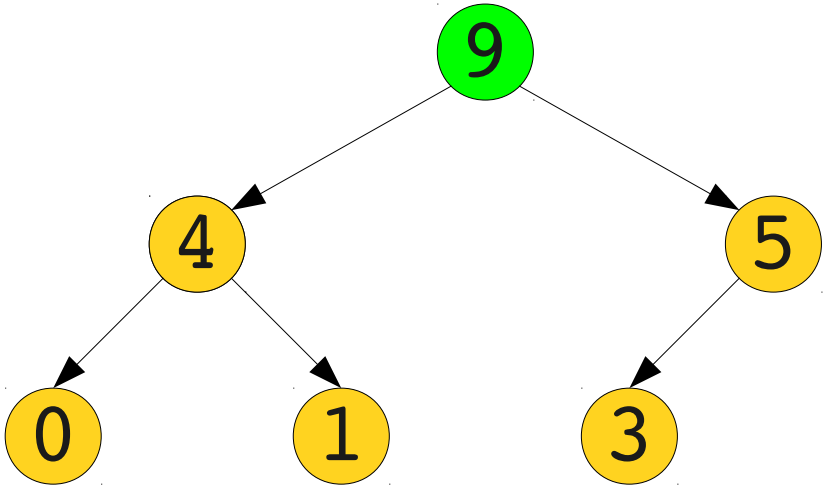


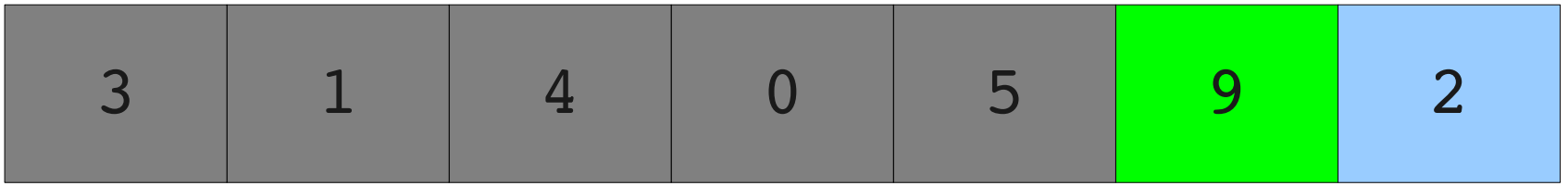
Sorting with Binary Heaps



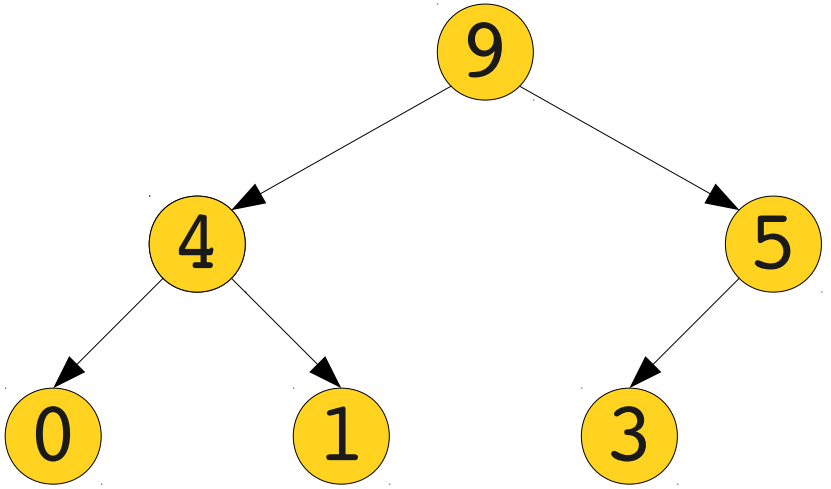


Sorting with Binary Heaps



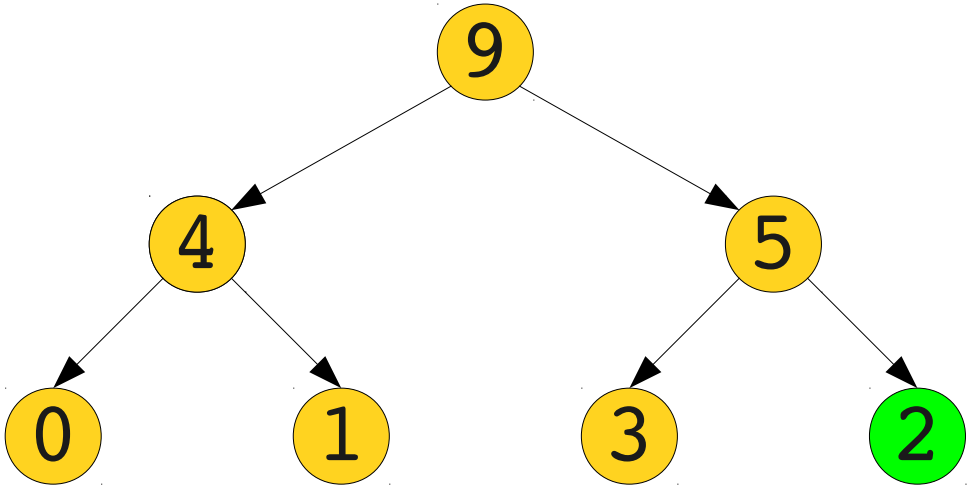


Sorting with Binary Heaps



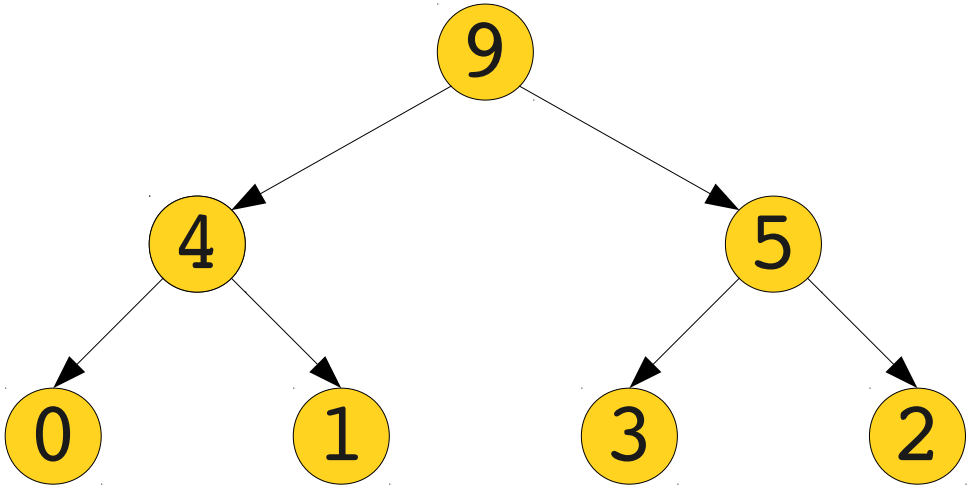
3	1	4	0	5	9	2

Sorting with Binary Heaps



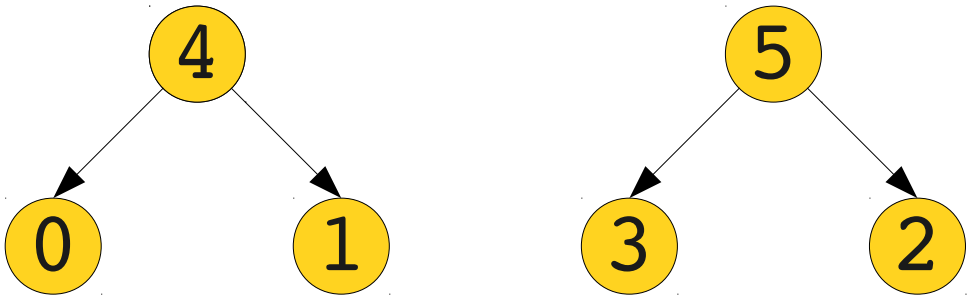
3	1	4	0	5	9	2

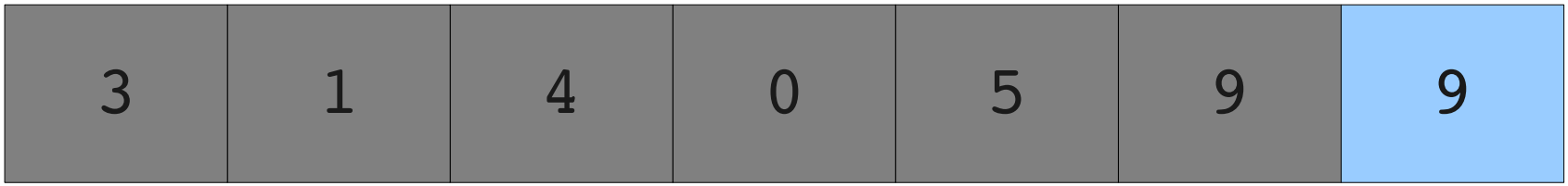
Sorting with Binary Heaps



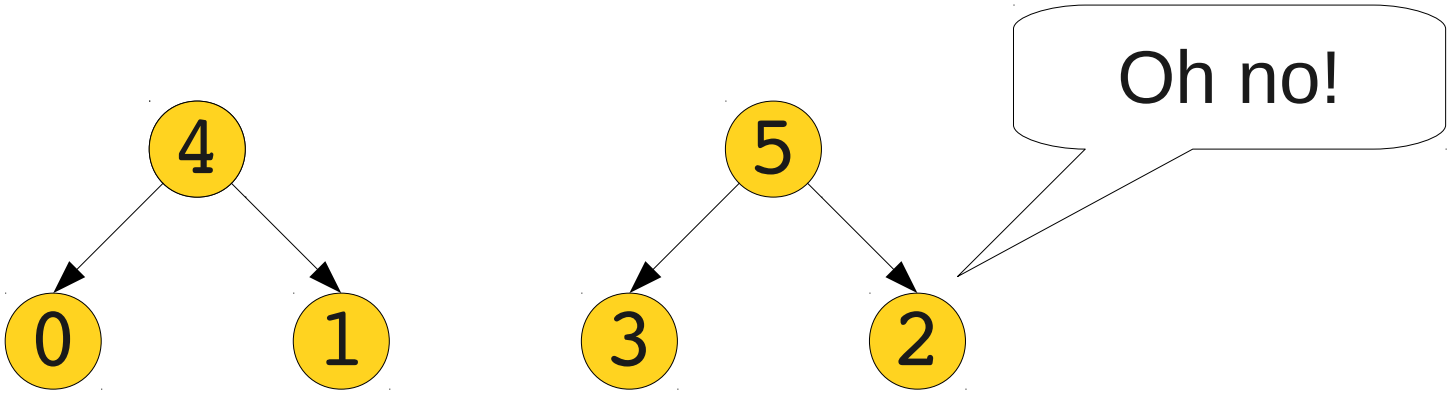
3	1	4	0	5	9	2

Sorting with Binary Heaps



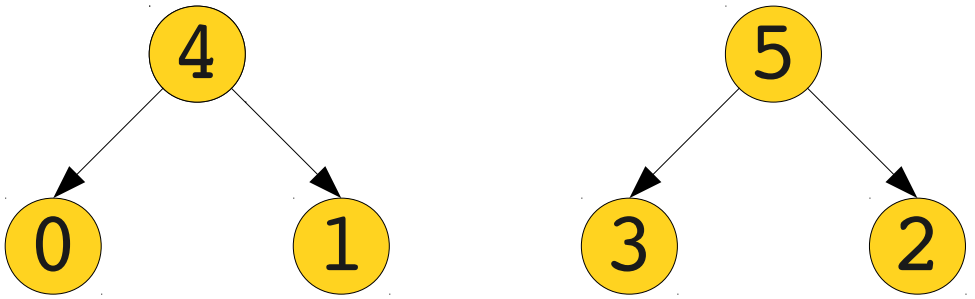


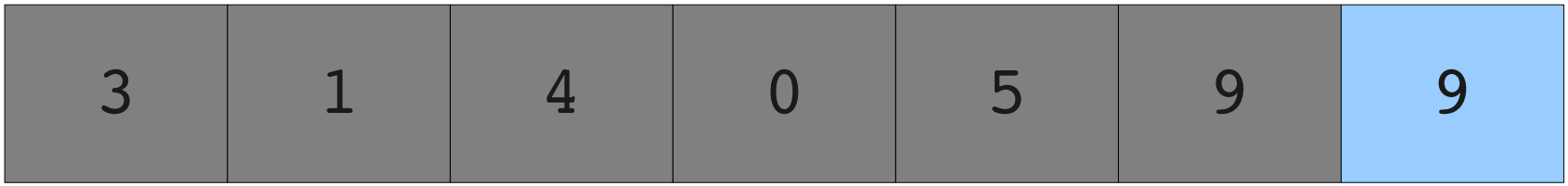
Sorting with Binary Heaps



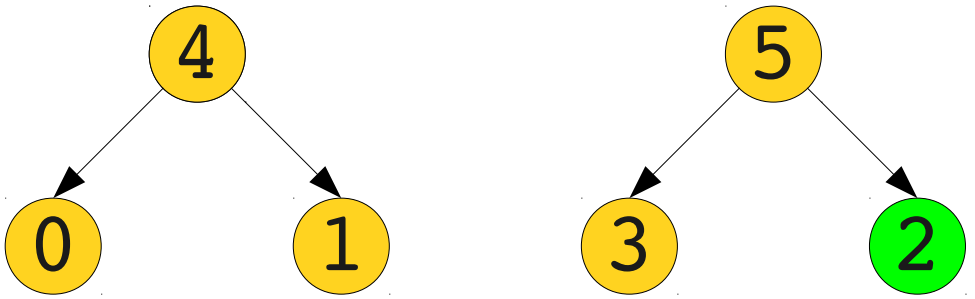
3	1	4	0	5	9	9

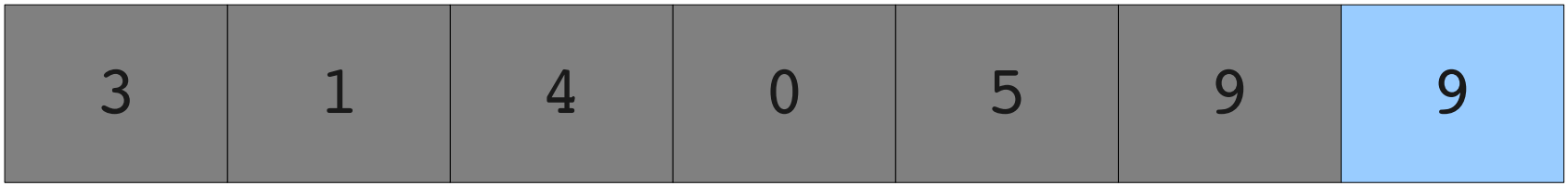
Sorting with Binary Heaps



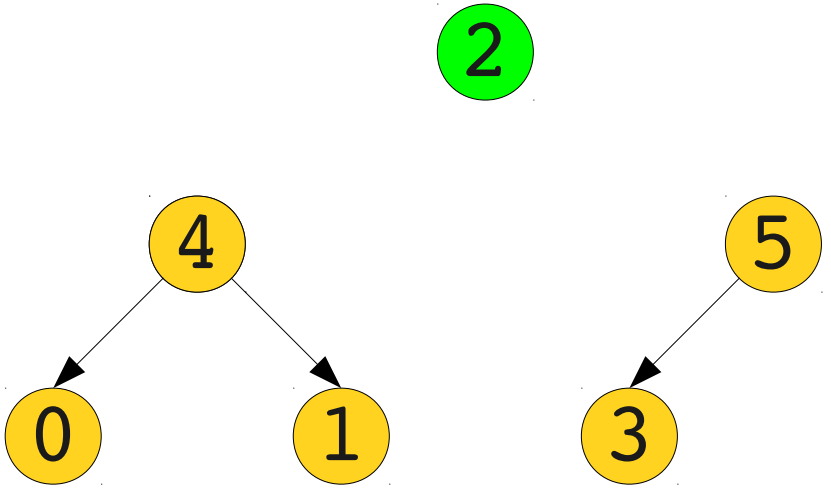


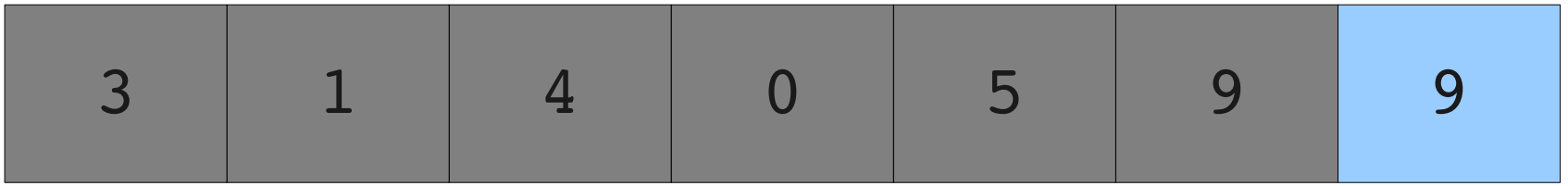
Sorting with Binary Heaps



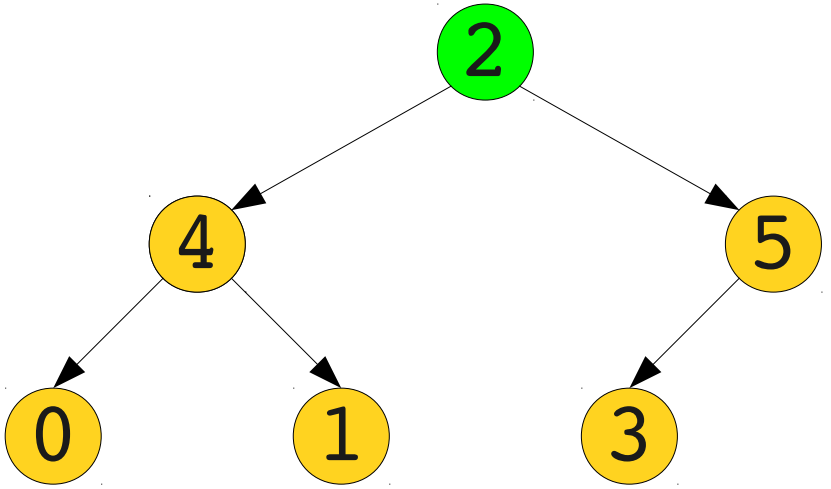


Sorting with Binary Heaps



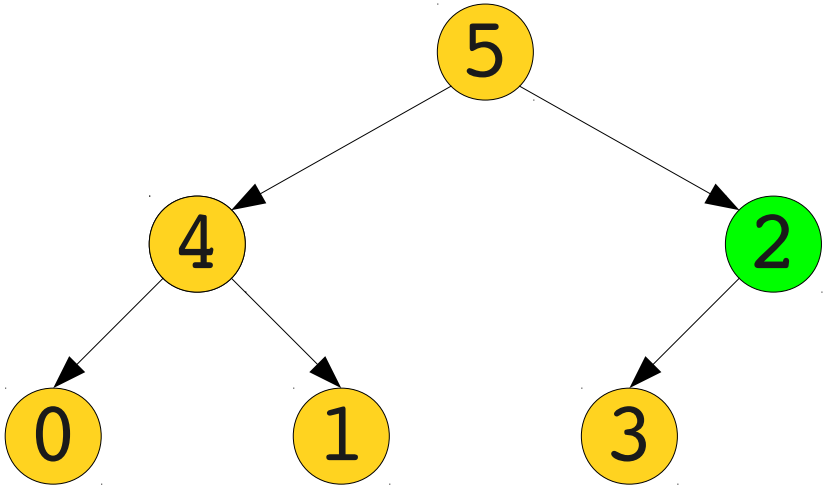


Sorting with Binary Heaps



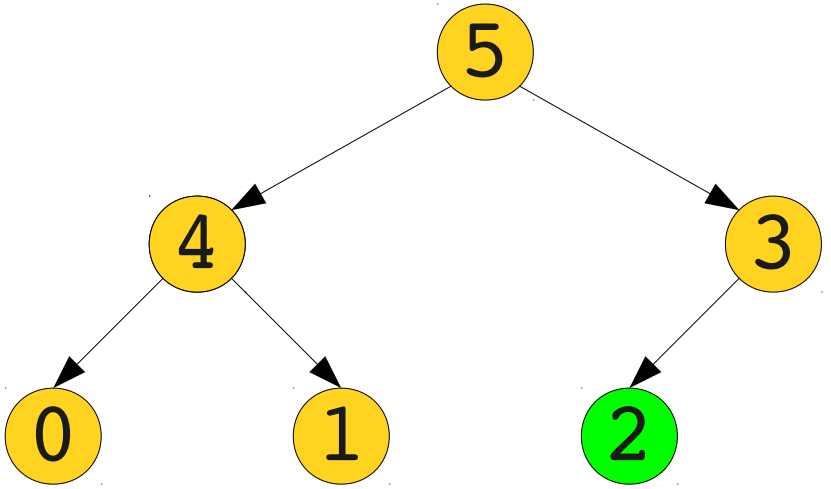
3	1	4	0	5	9	9

Sorting with Binary Heaps



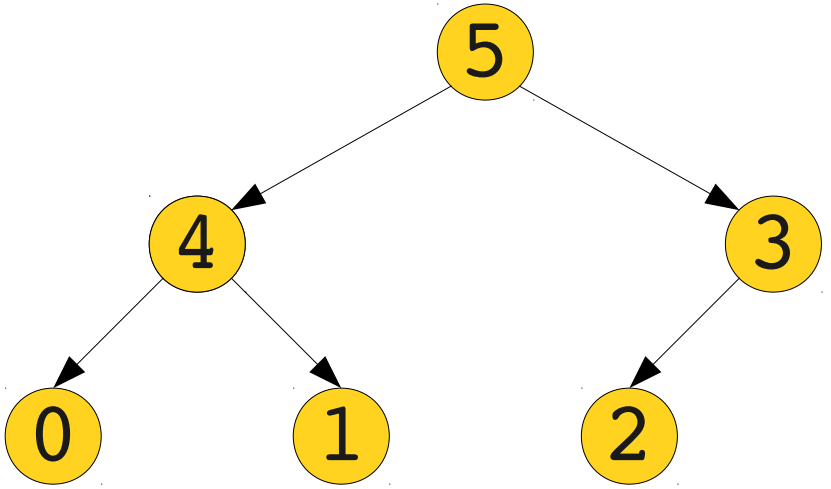
3	1	4	0	5	9	9

Sorting with Binary Heaps



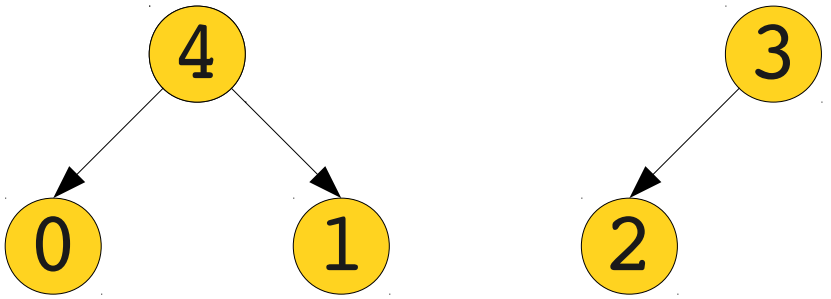
3	1	4	0	5	9	9

Sorting with Binary Heaps



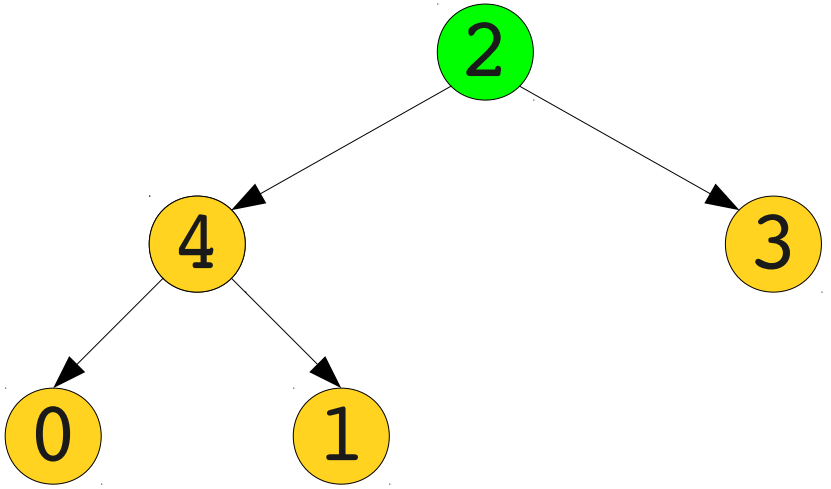
3	1	4	0	5	9	9

Sorting with Binary Heaps



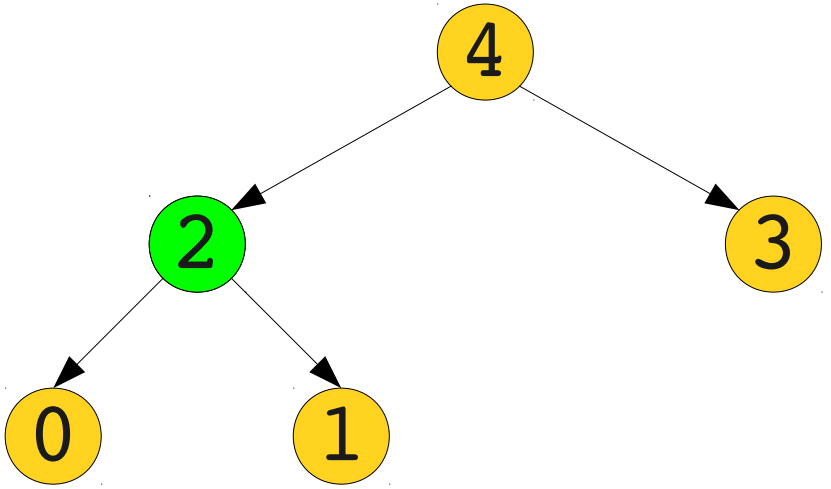
3	1	4	0	5	5	9

Sorting with Binary Heaps



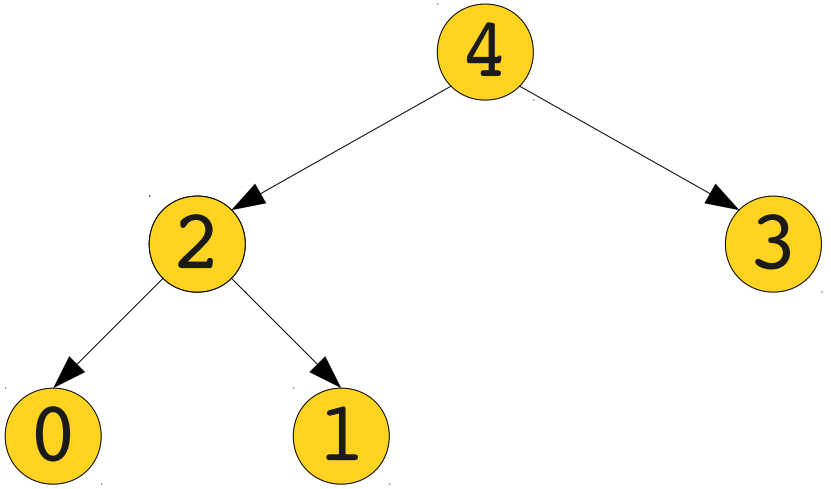
3	1	4	0	5	5	9

Sorting with Binary Heaps



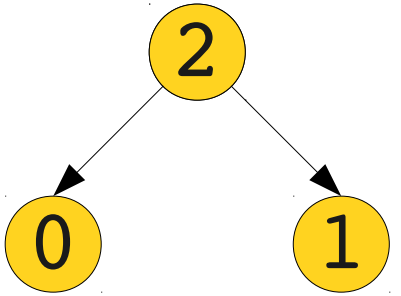
3	1	4	0	5	5	9

Sorting with Binary Heaps



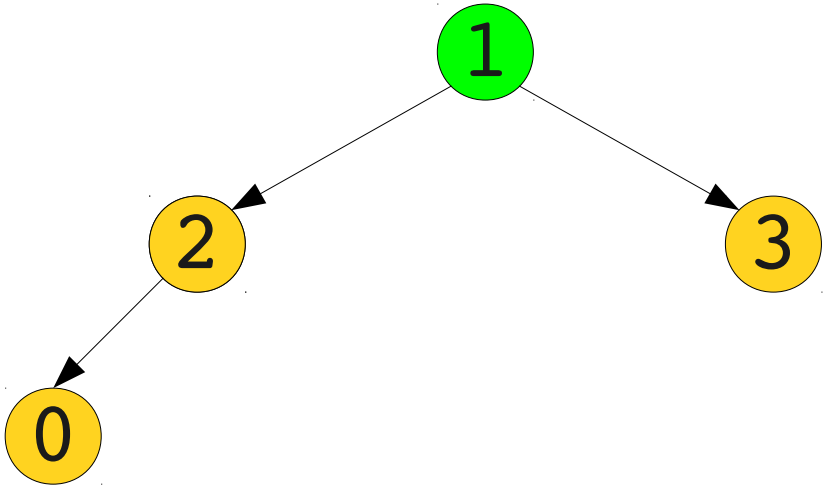
3	1	4	0	5	5	9

Sorting with Binary Heaps



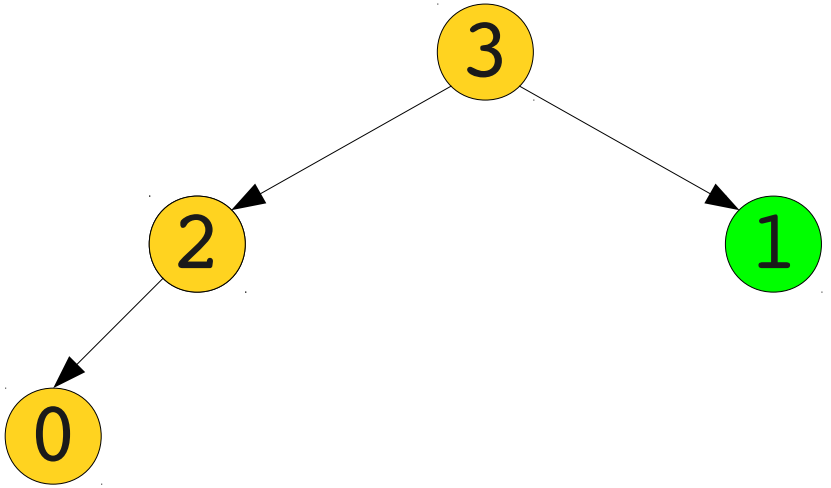
3	1	4	0	4	5	9

Sorting with Binary Heaps



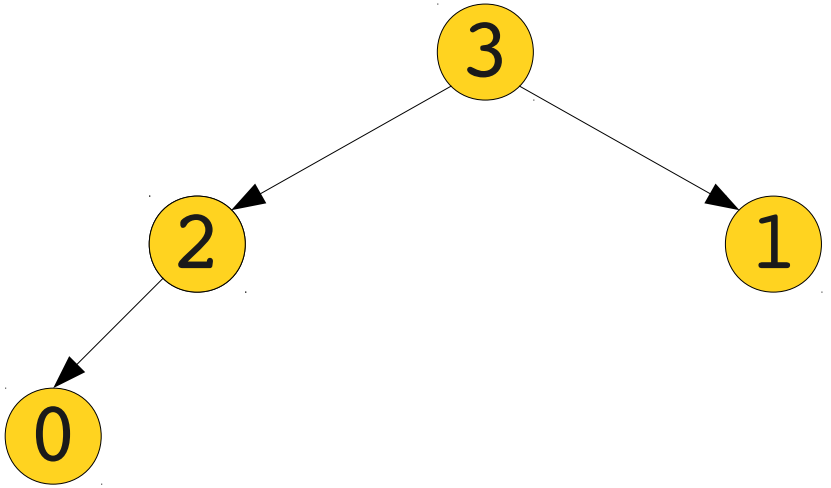
3	1	4	0	4	5	9

Sorting with Binary Heaps



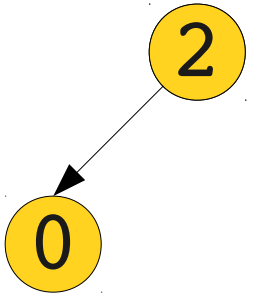
3	1	4	0	4	5	9

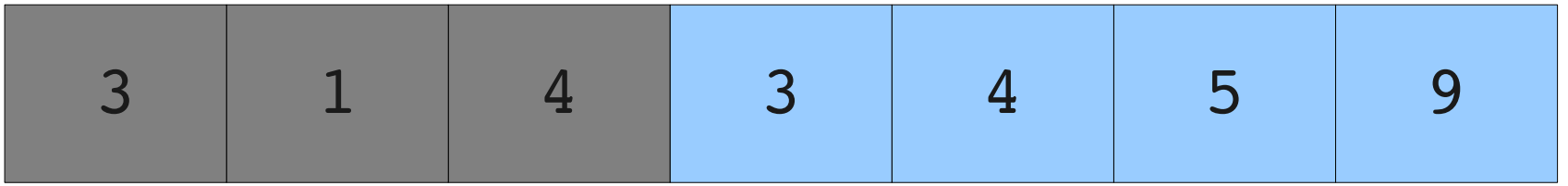
Sorting with Binary Heaps



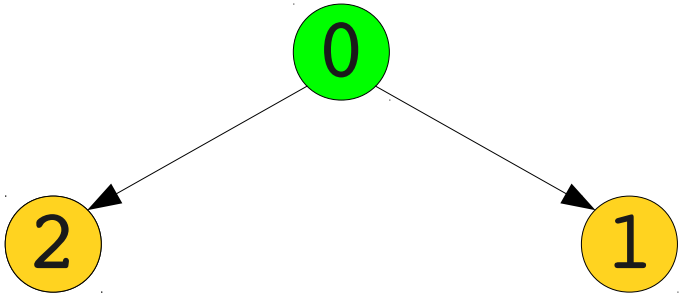
3	1	4	0	4	5	9

Sorting with Binary Heaps



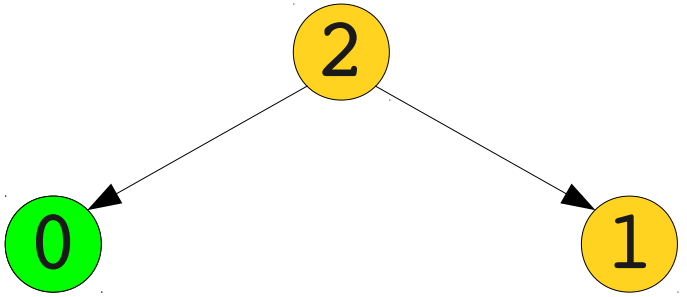


Sorting with Binary Heaps



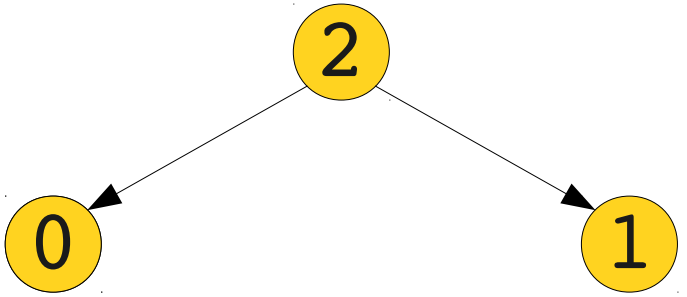
3	1	4	3	4	5	9

Sorting with Binary Heaps



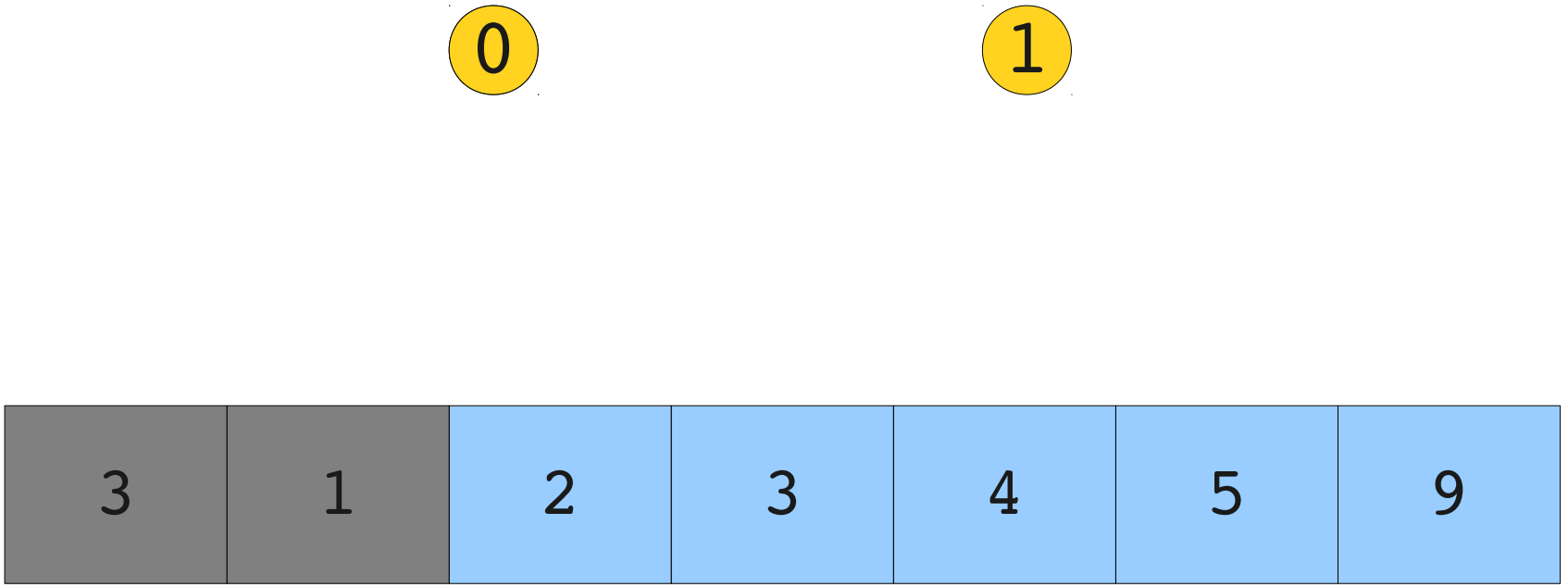
3	1	4	3	4	5	9

Sorting with Binary Heaps

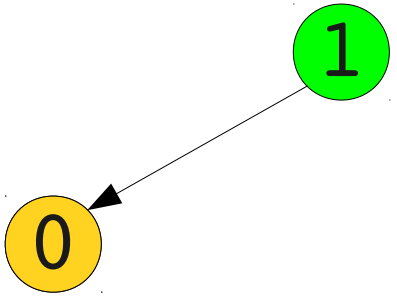


3	1	4	3	4	5	9

Sorting with Binary Heaps

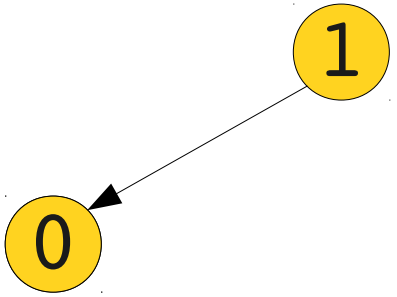


Sorting with Binary Heaps



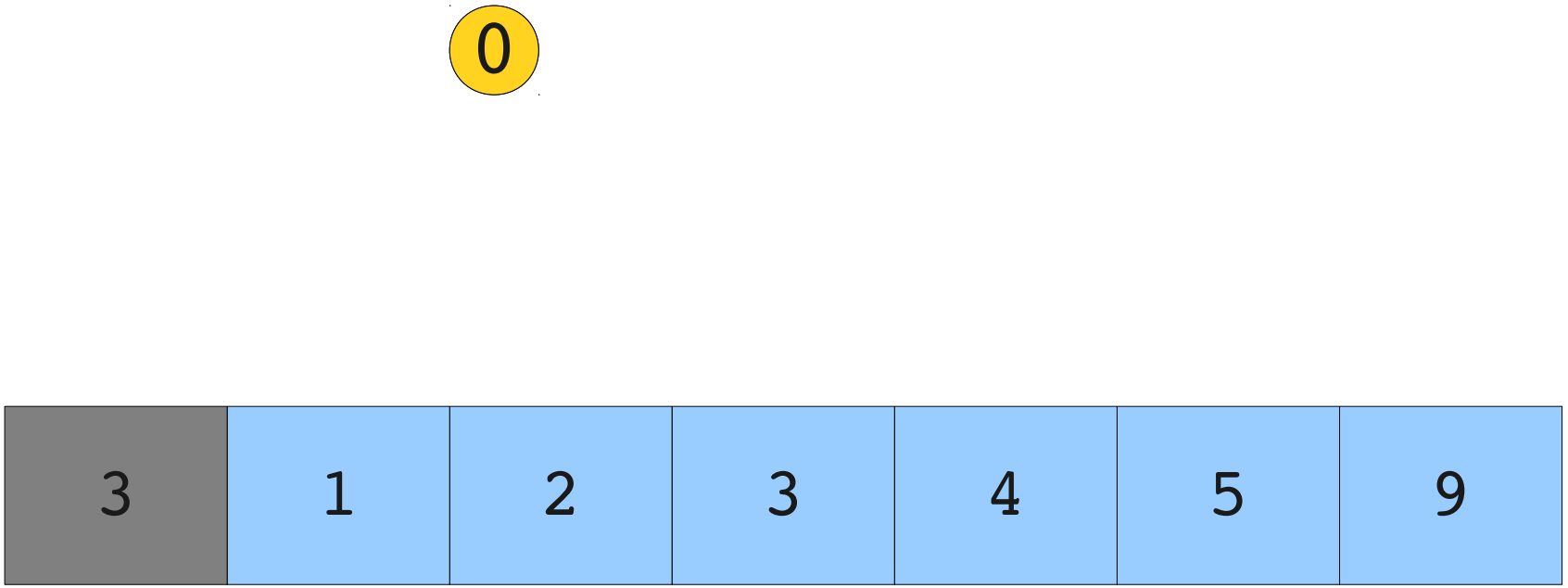
3	1	2	3	4	5	9

Sorting with Binary Heaps

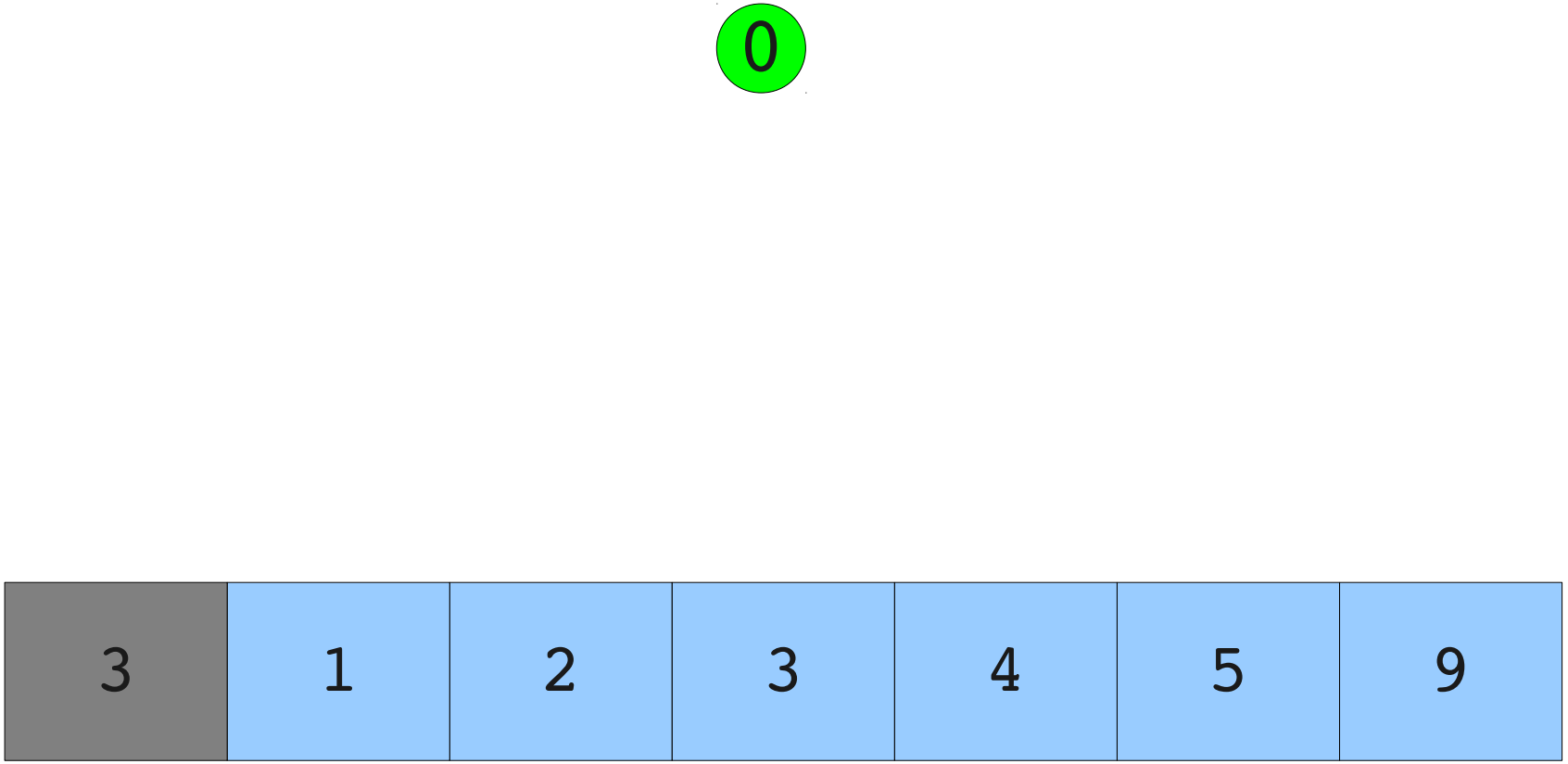


3	1	2	3	4	5	9

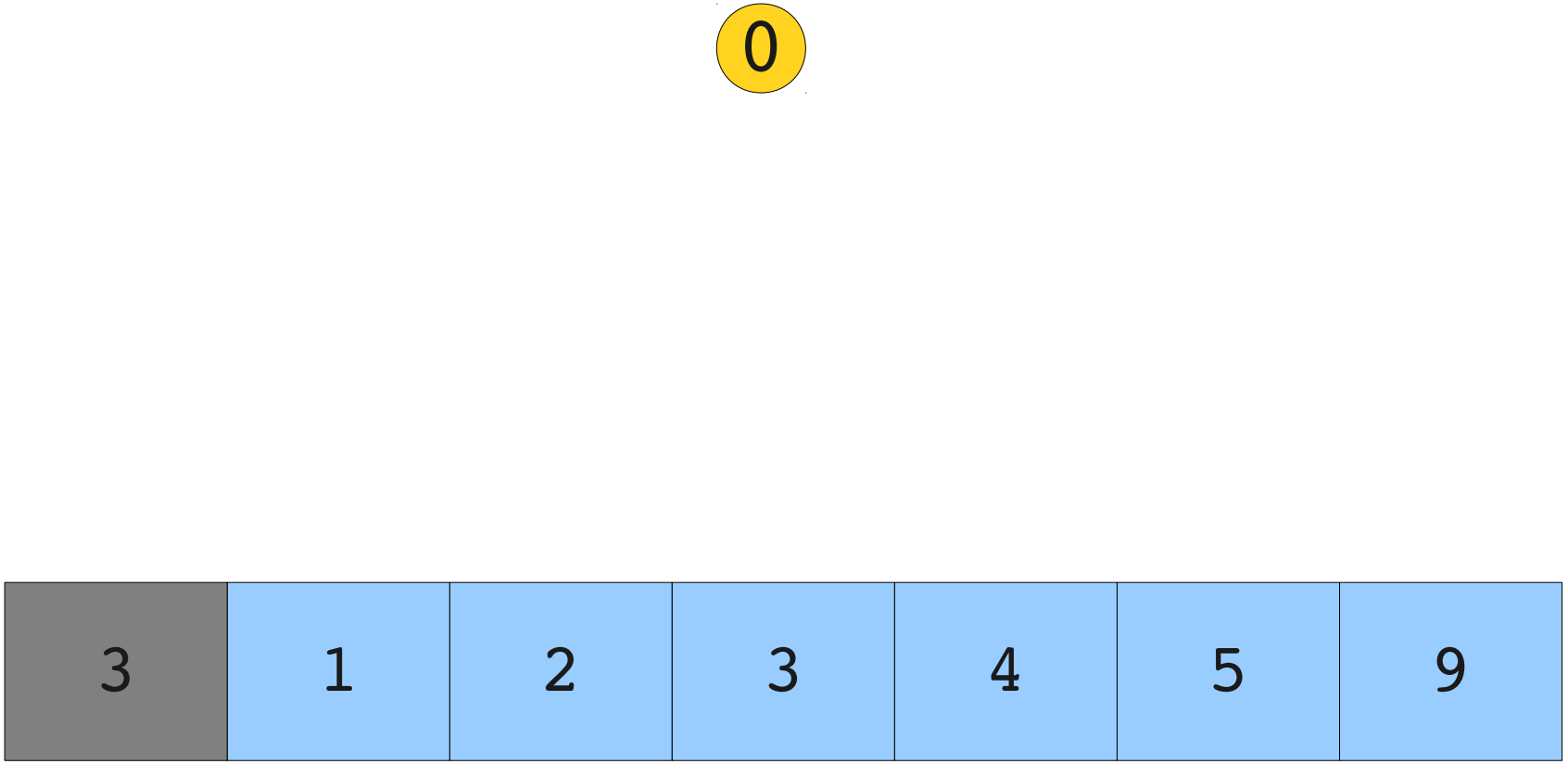
Sorting with Binary Heaps



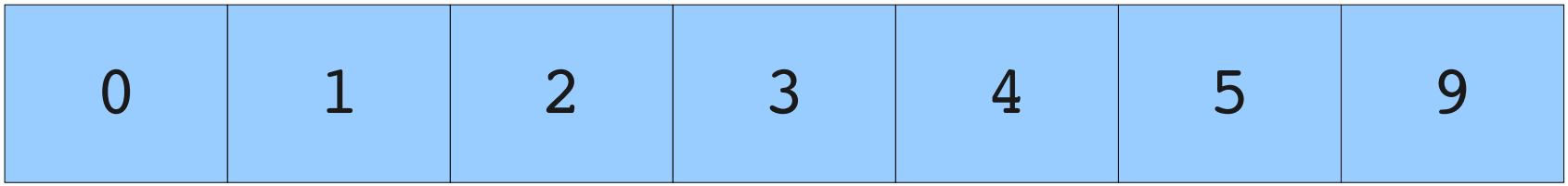
Sorting with Binary Heaps



Sorting with Binary Heaps

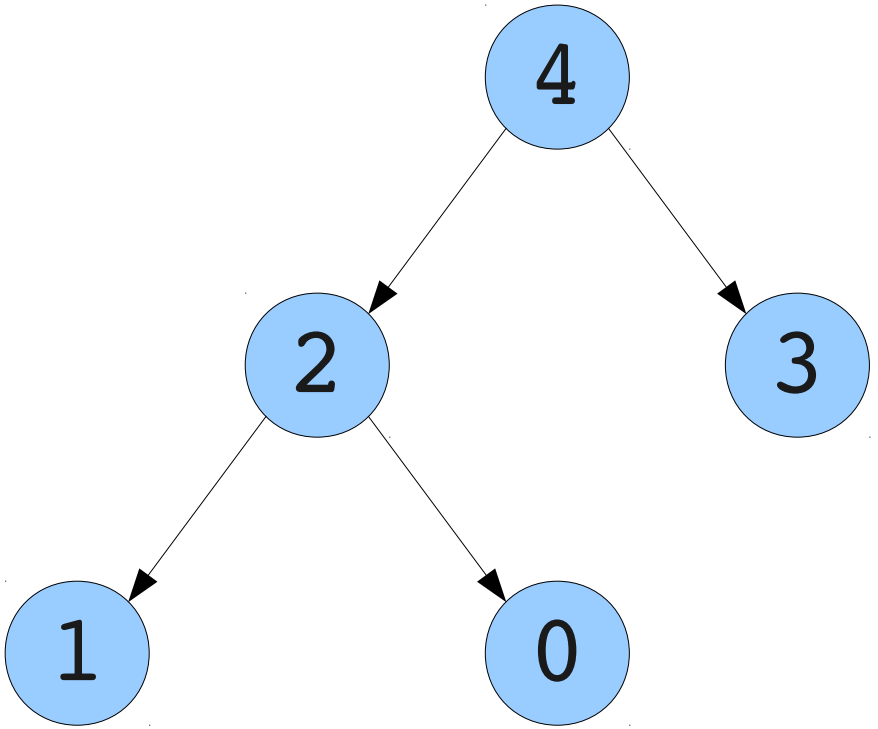


Sorting with Binary Heaps

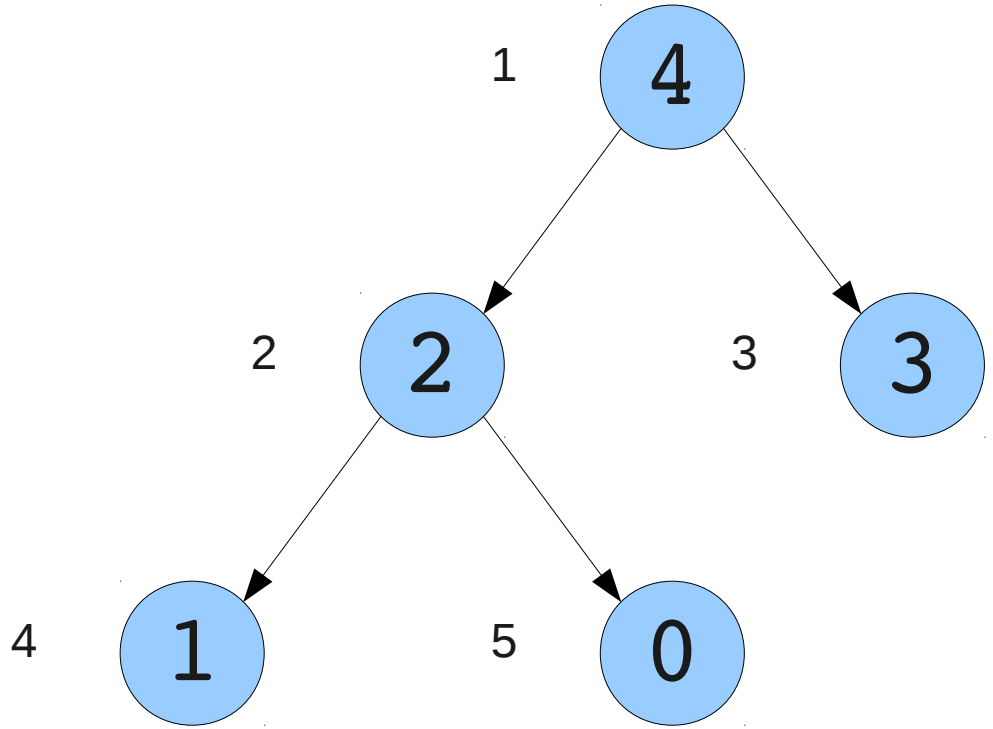


Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Binary Heap	Naïve Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$

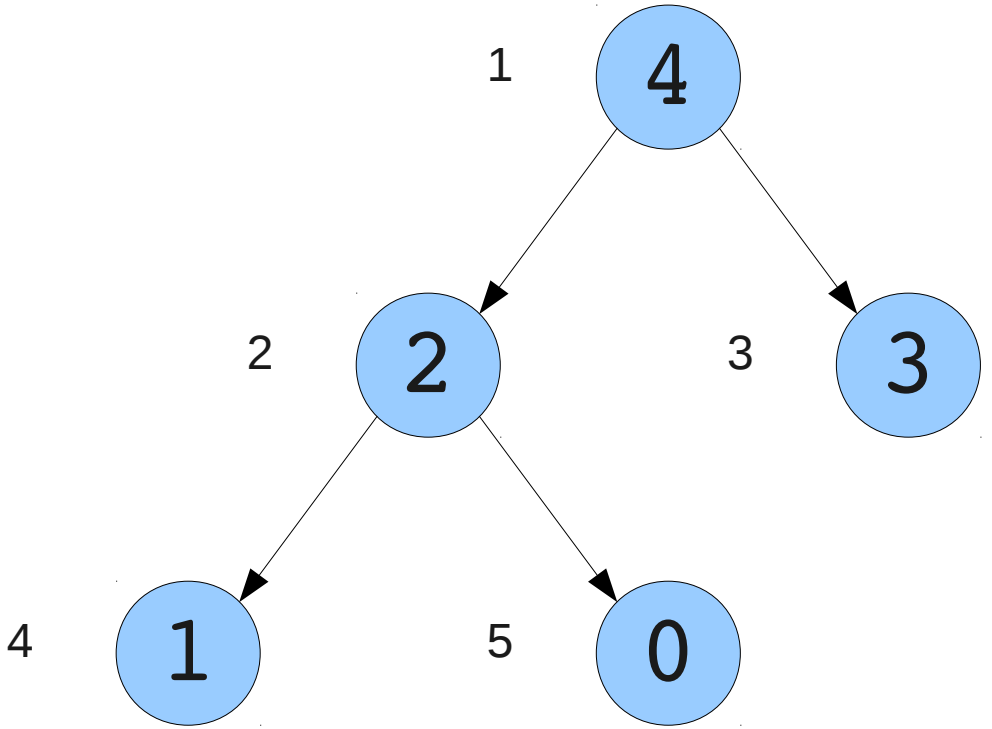
Implicit Binary Heaps



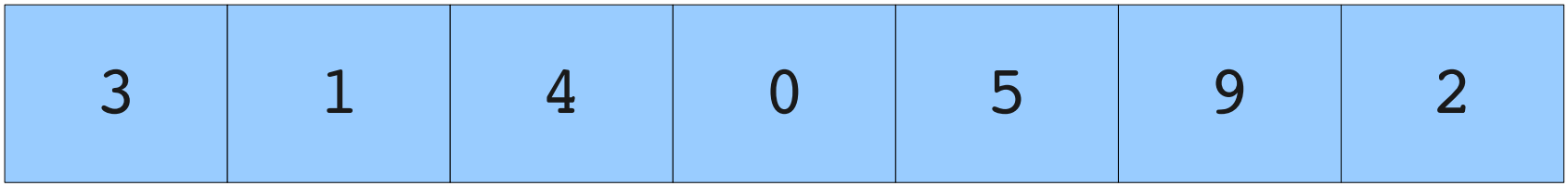
Implicit Binary Heaps



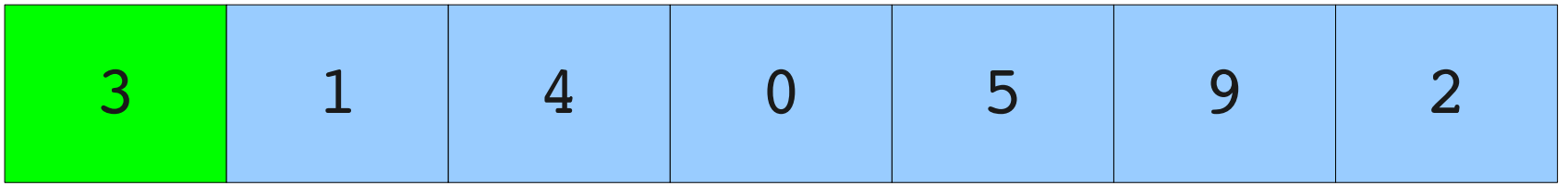
Implicit Binary Heaps



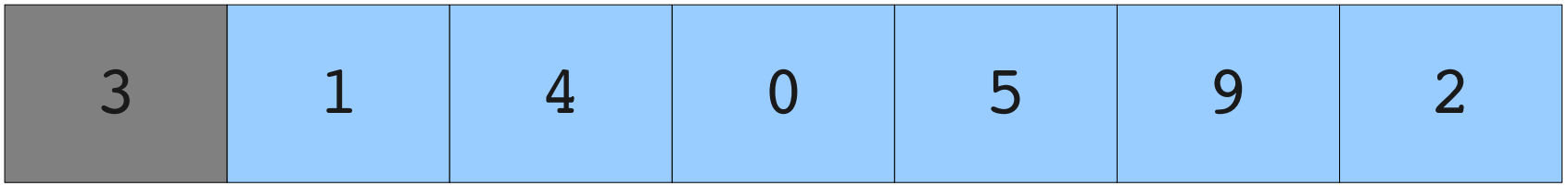
Sorting with Implicit Binary Heaps



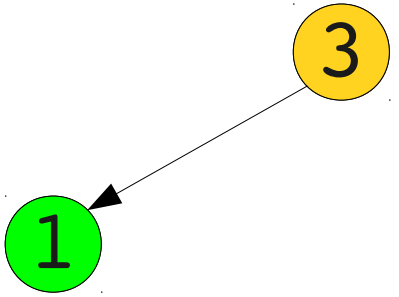
Sorting with Implicit Binary Heaps

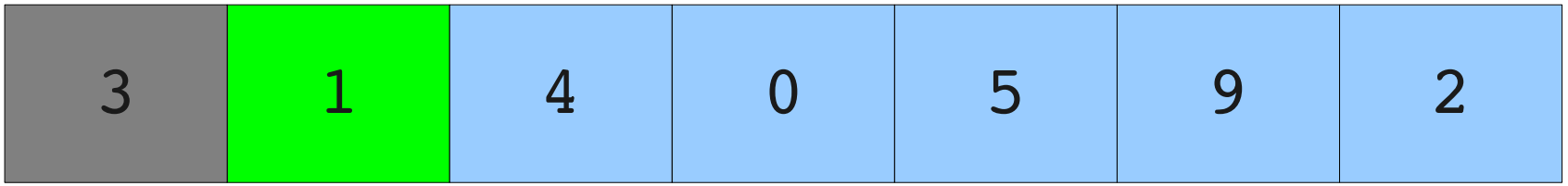


Sorting with Implicit Binary Heaps

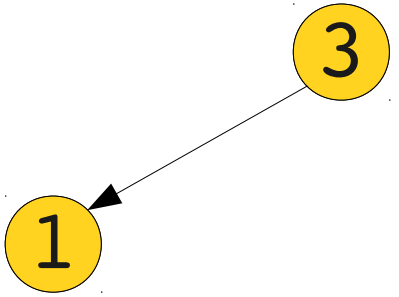


Sorting with Implicit Binary Heaps



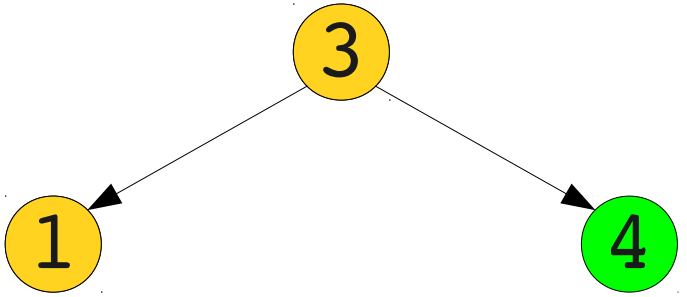


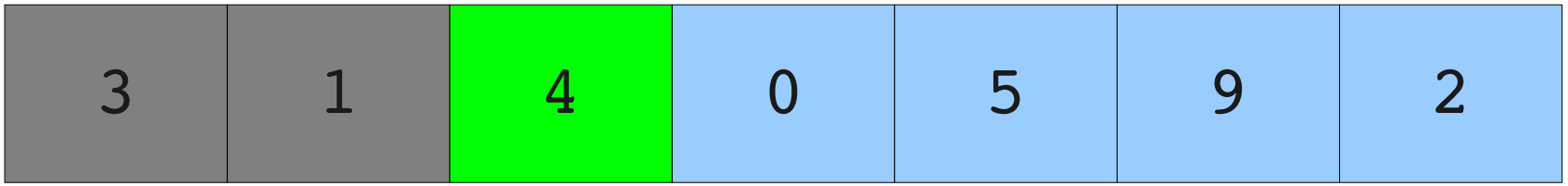
Sorting with Implicit Binary Heaps



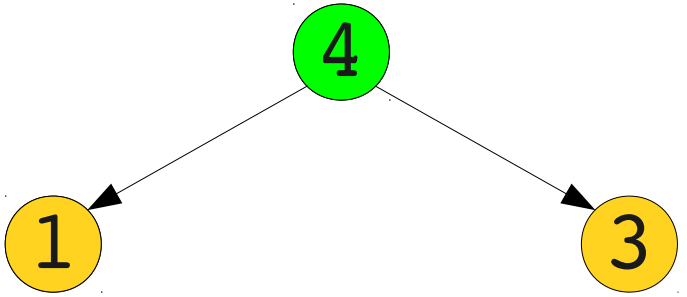
3	1	4	0	5	9	2

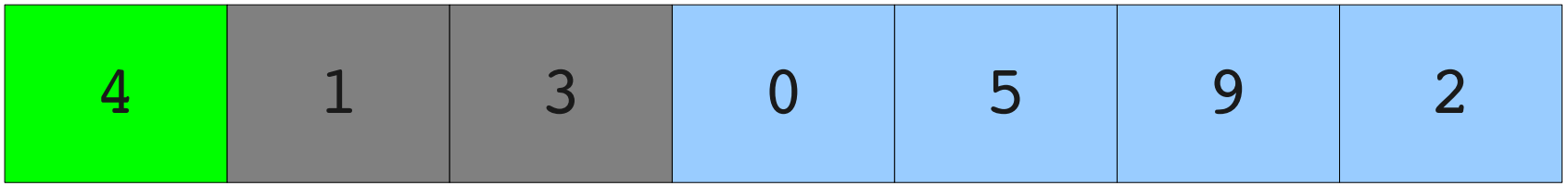
Sorting with Implicit Binary Heaps



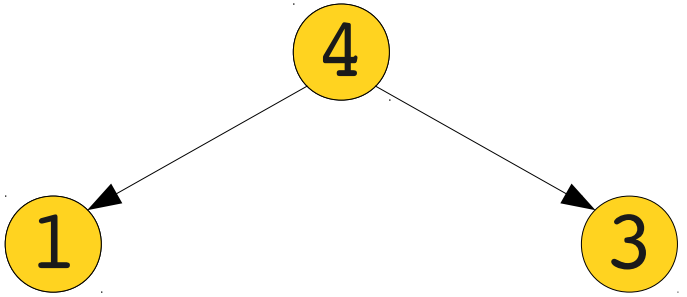


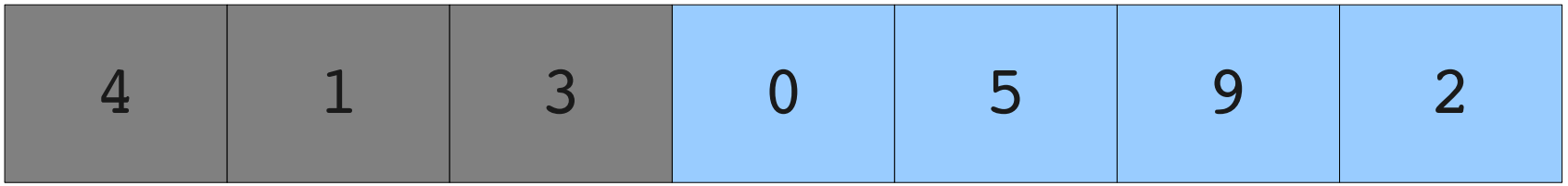
Sorting with Implicit Binary Heaps



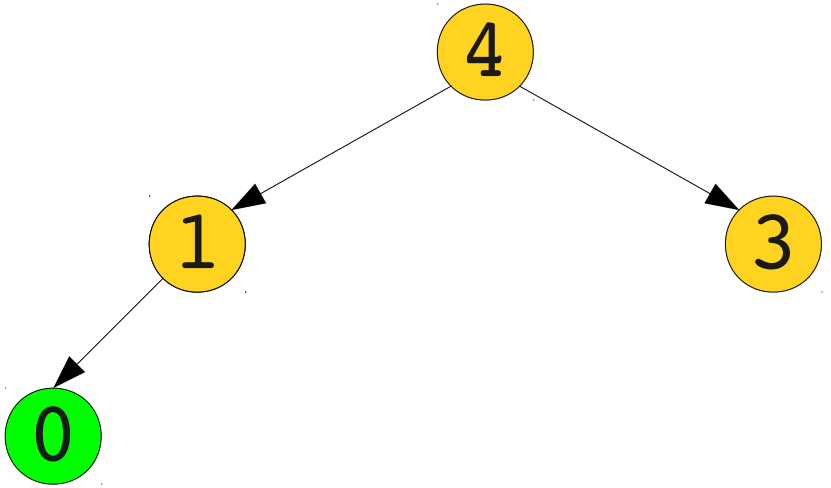


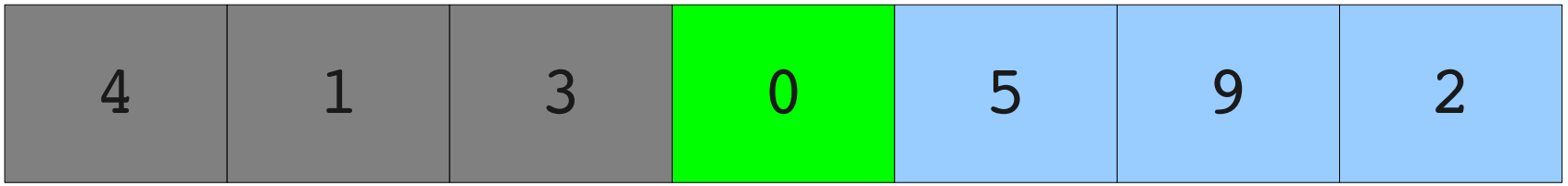
Sorting with Implicit Binary Heaps



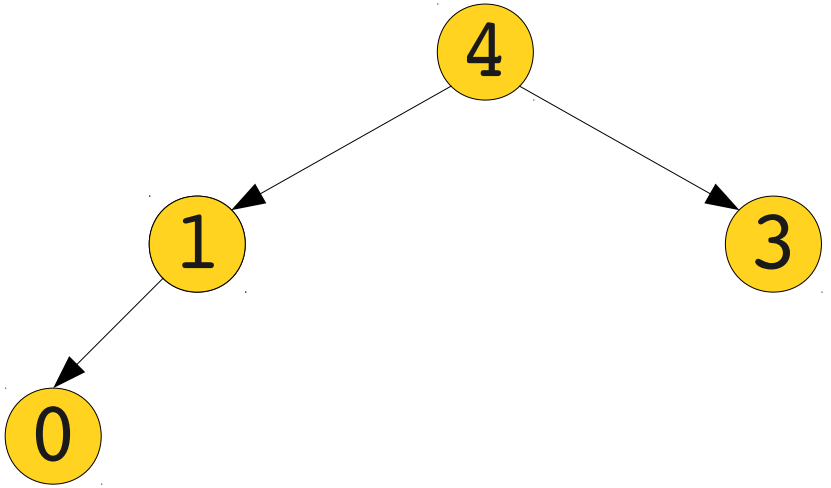


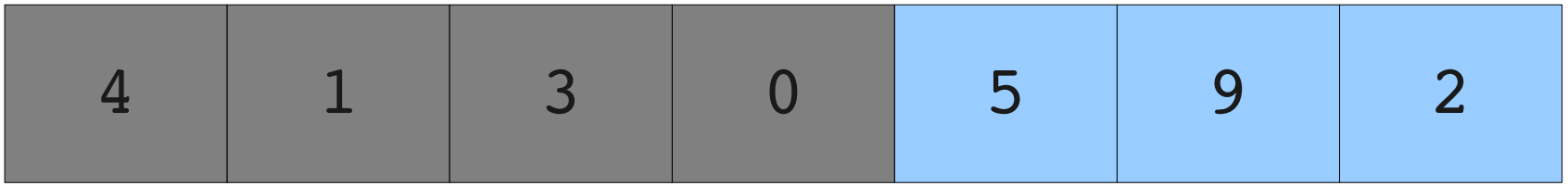
Sorting with Implicit Binary Heaps



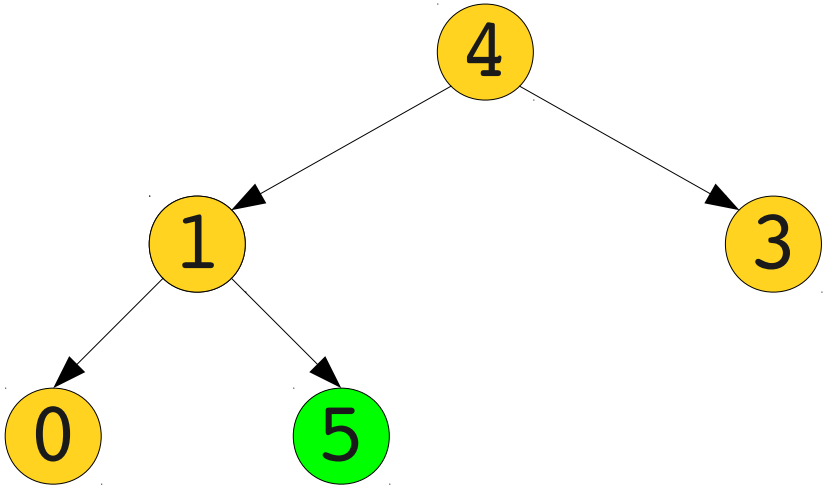


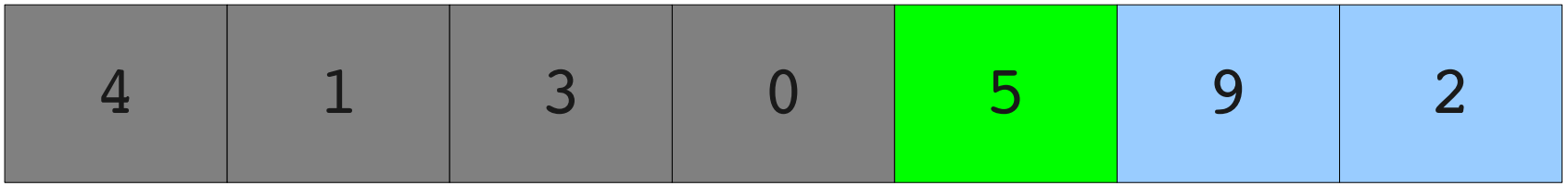
Sorting with Implicit Binary Heaps



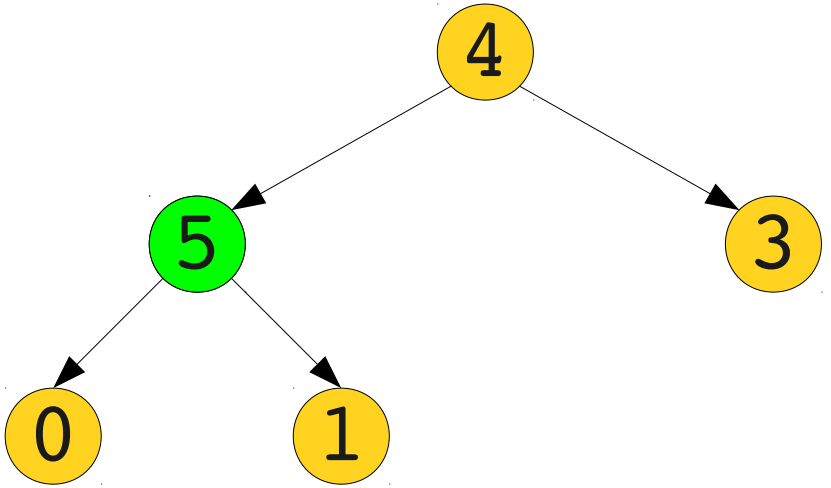


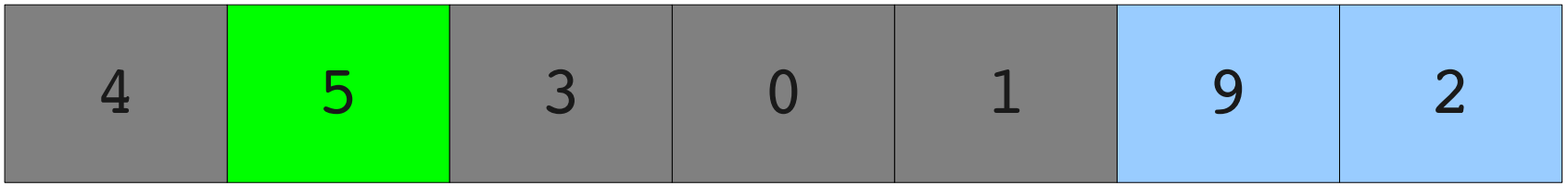
Sorting with Implicit Binary Heaps



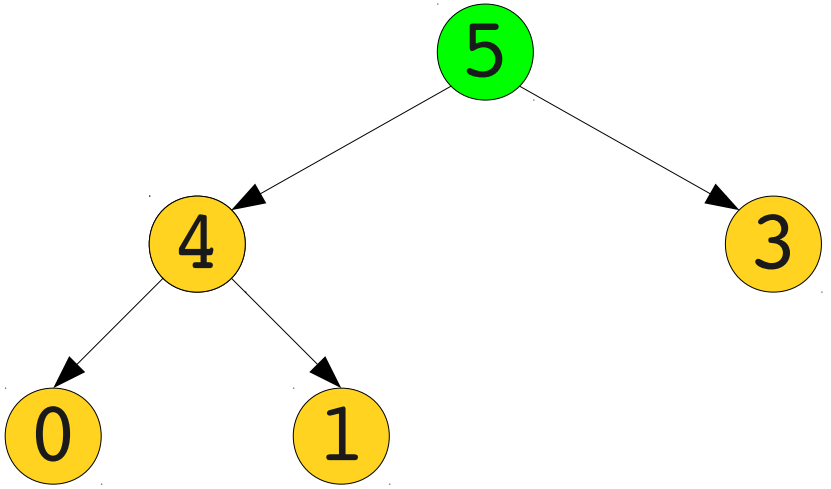


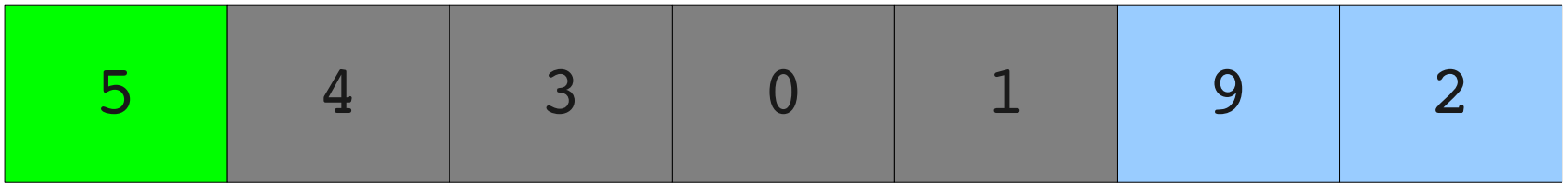
Sorting with Implicit Binary Heaps



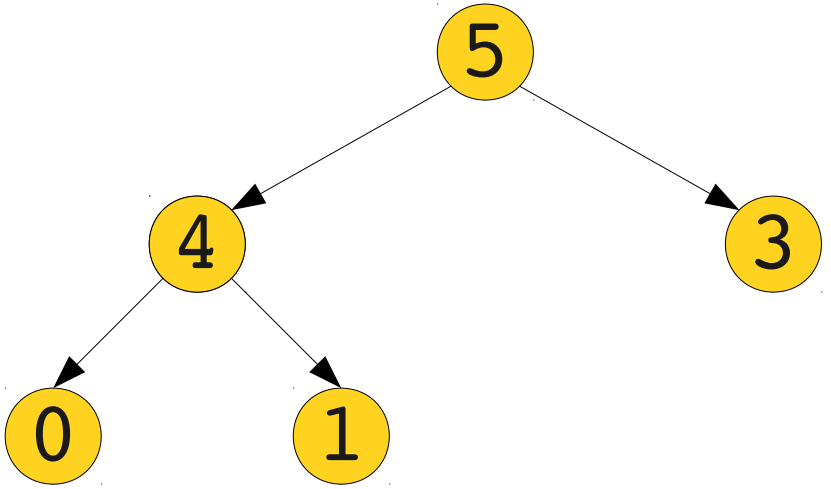


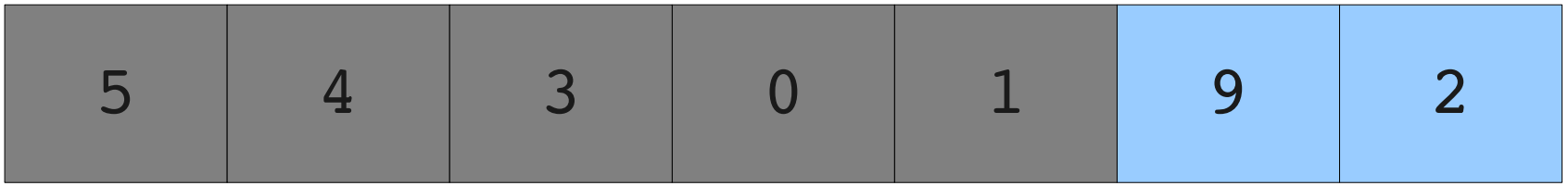
Sorting with Implicit Binary Heaps



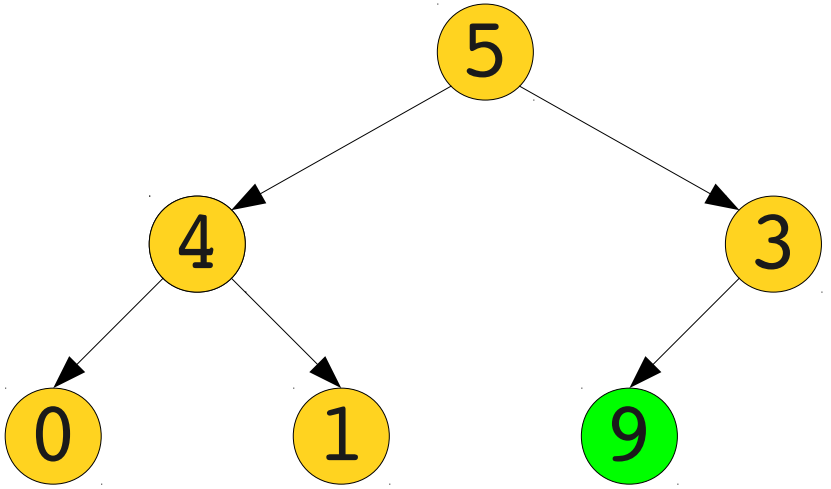


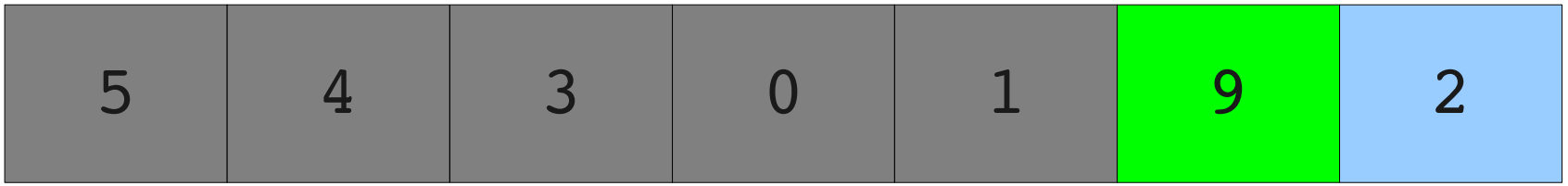
Sorting with Implicit Binary Heaps



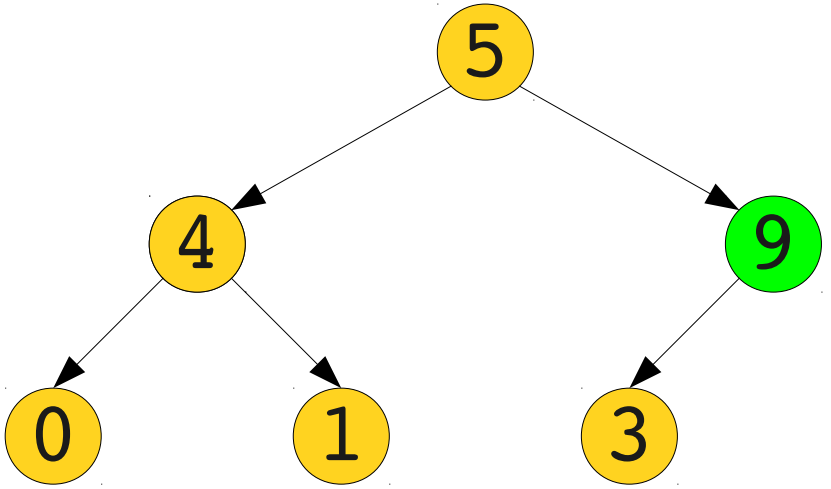


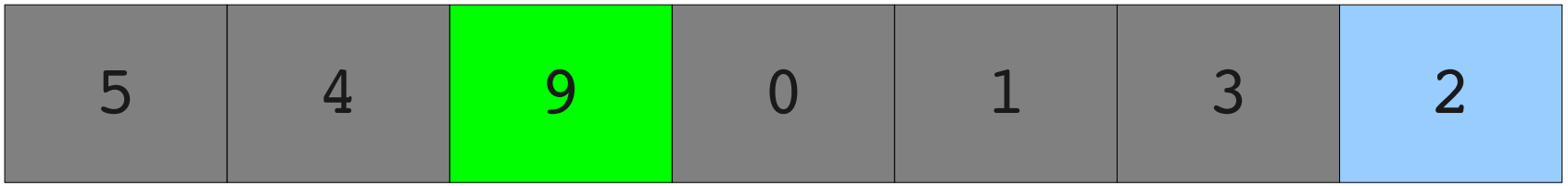
Sorting with Implicit Binary Heaps



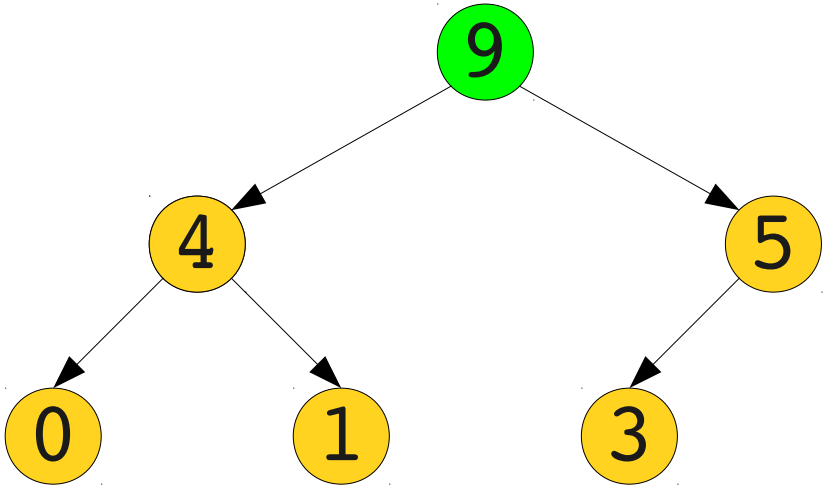


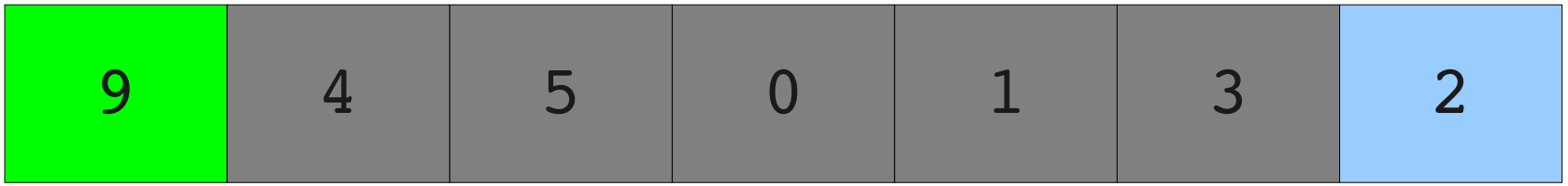
Sorting with Implicit Binary Heaps



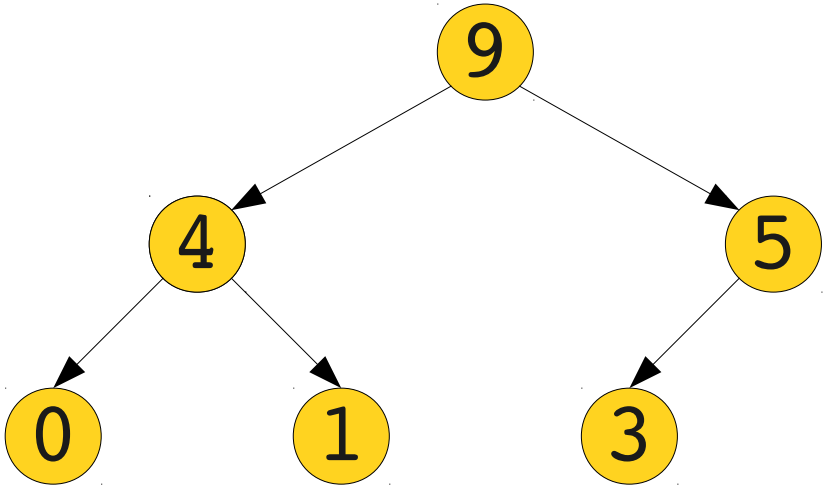


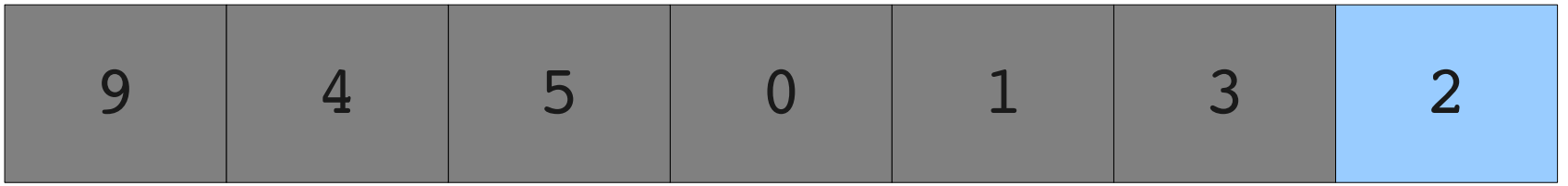
Sorting with Implicit Binary Heaps



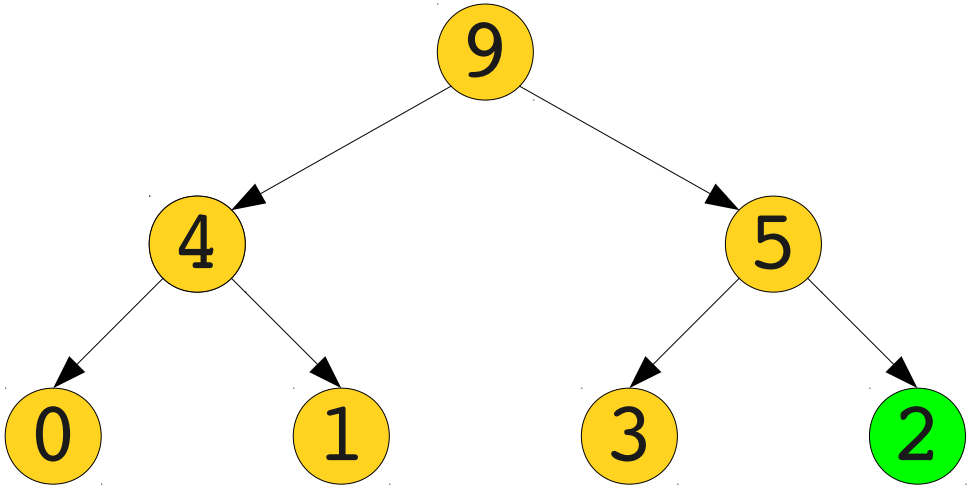


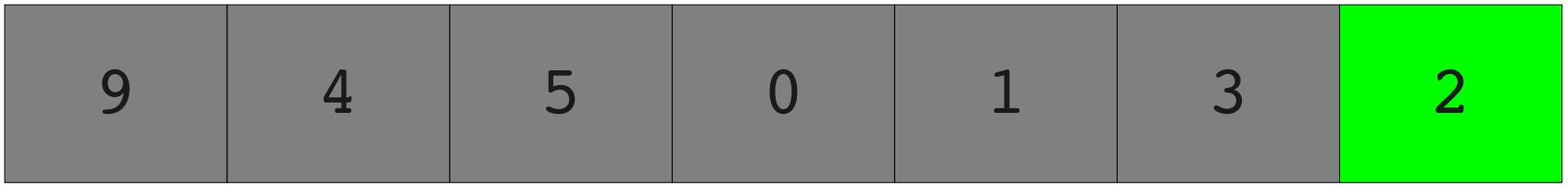
Sorting with Implicit Binary Heaps



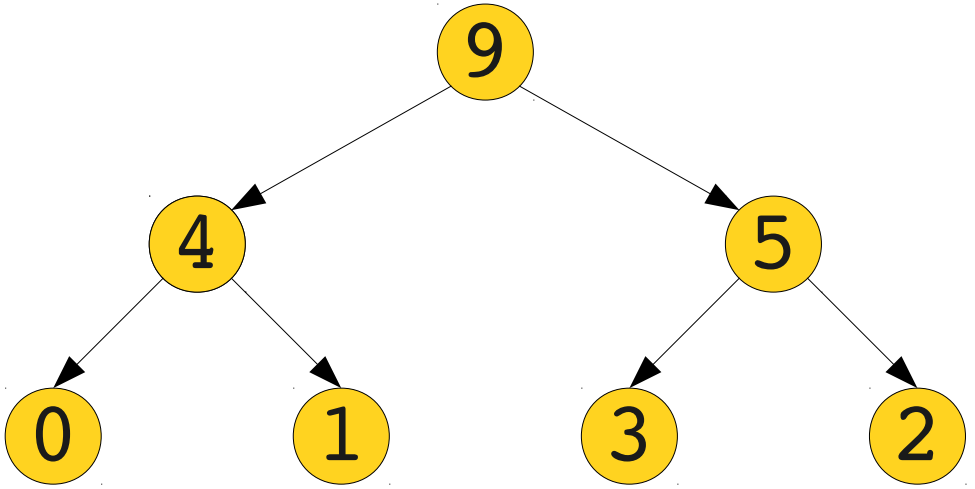


Sorting with Implicit Binary Heaps



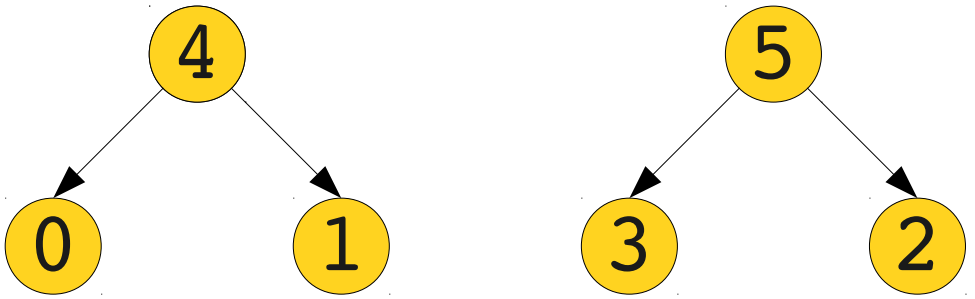


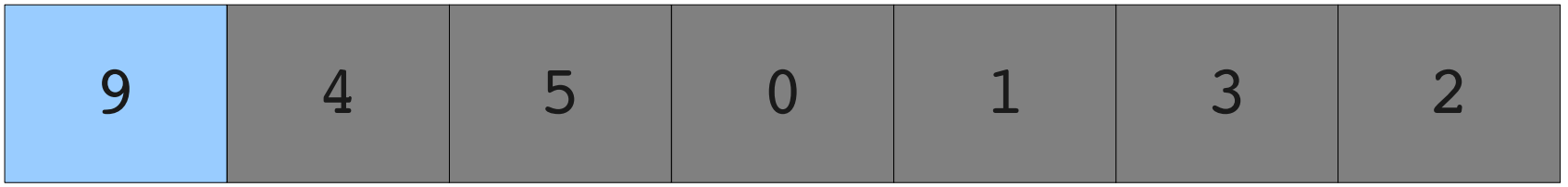
Sorting with Implicit Binary Heaps



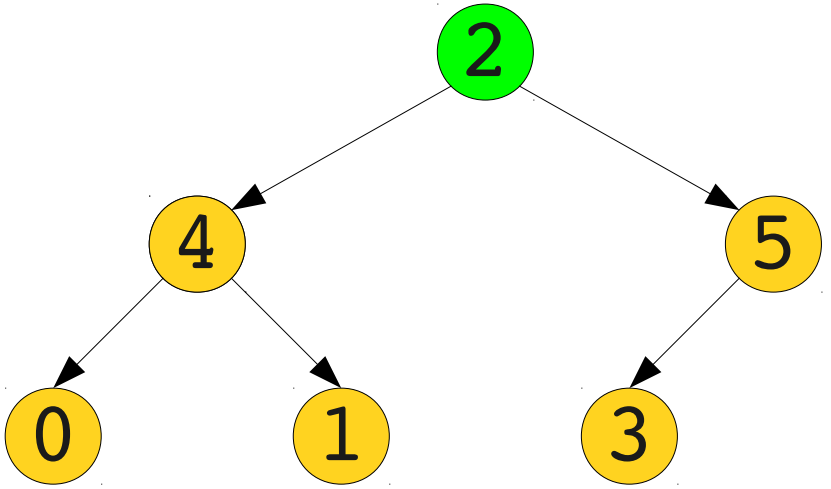
9	4	5	0	1	3	2

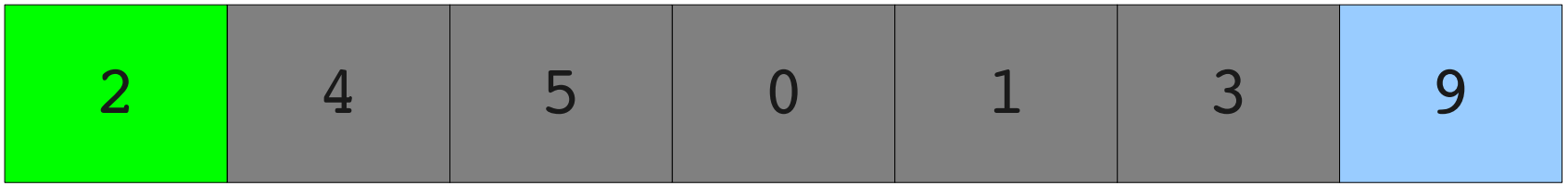
Sorting with Implicit Binary Heaps



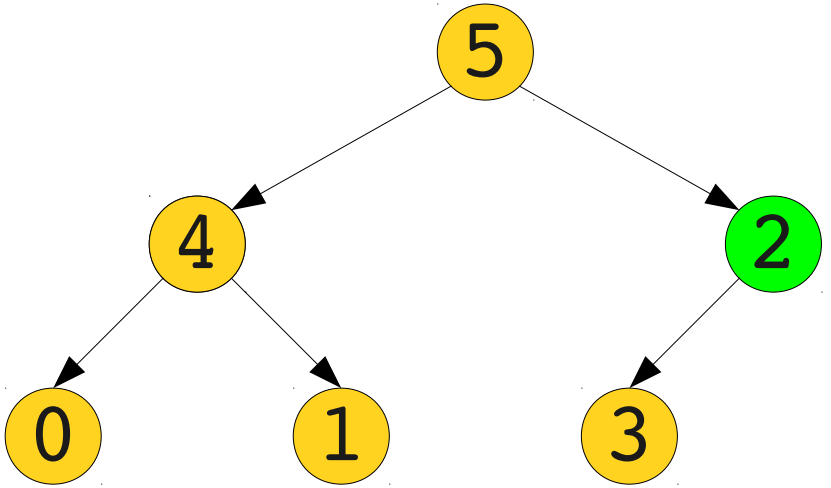


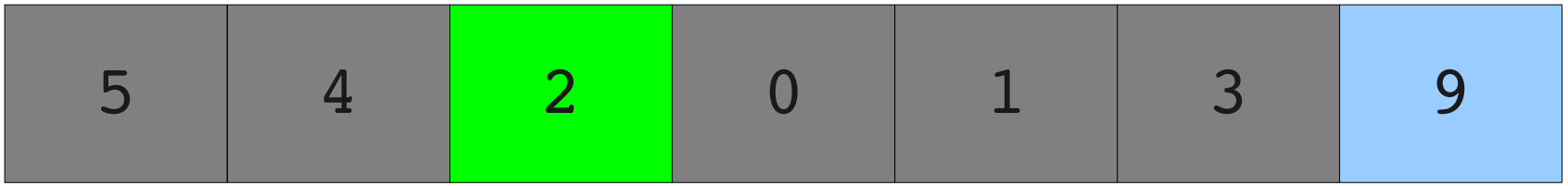
Sorting with Implicit Binary Heaps



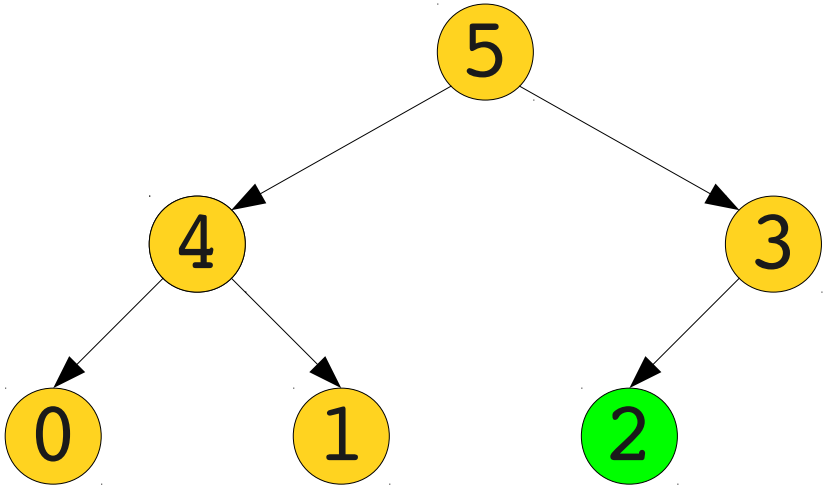


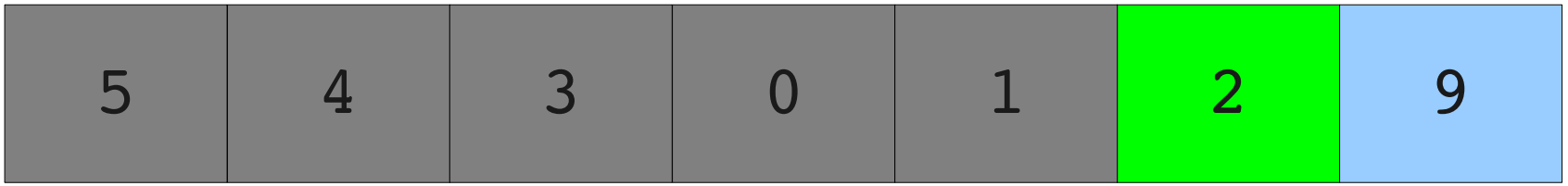
Sorting with Implicit Binary Heaps



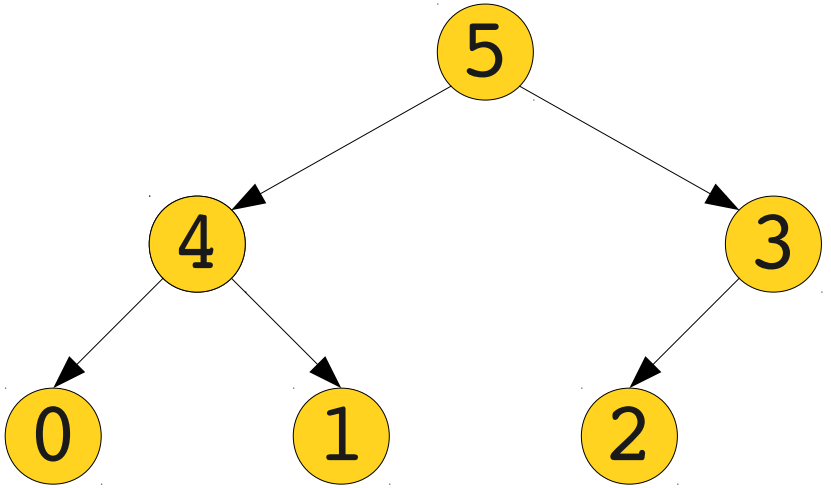


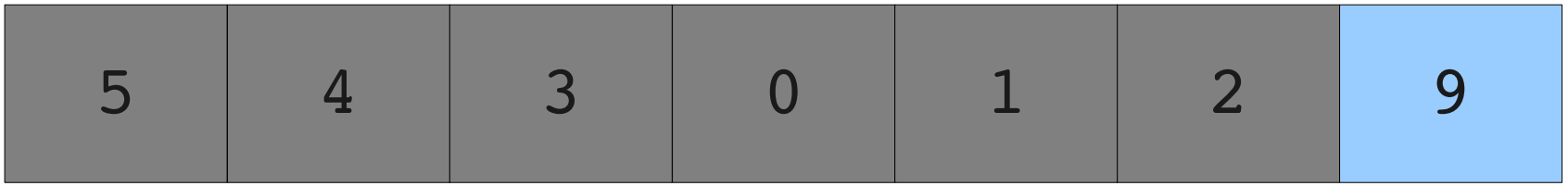
Sorting with Implicit Binary Heaps



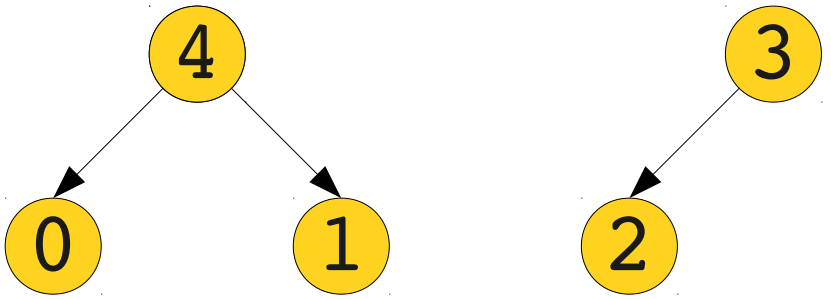


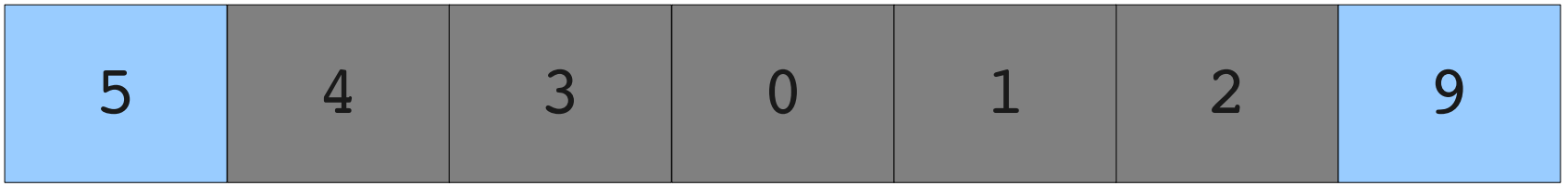
Sorting with Implicit Binary Heaps



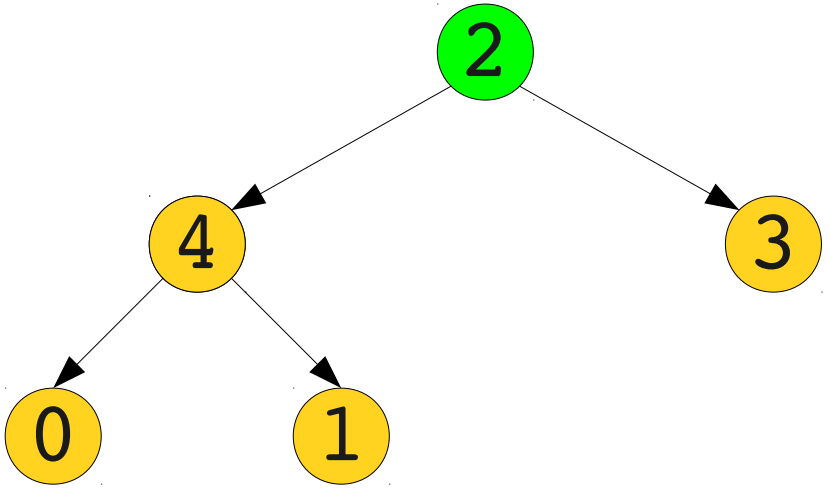


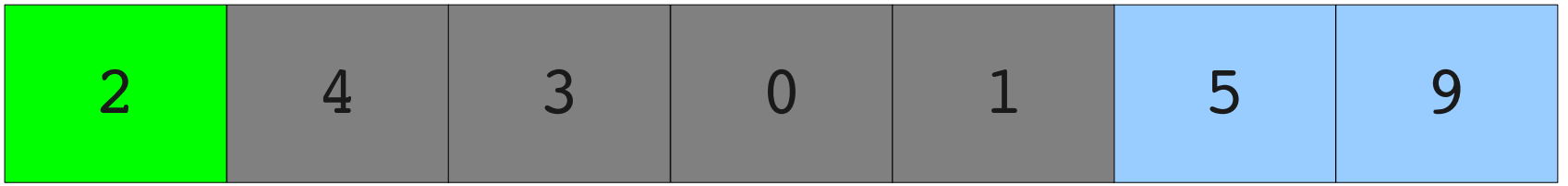
Sorting with Implicit Binary Heaps



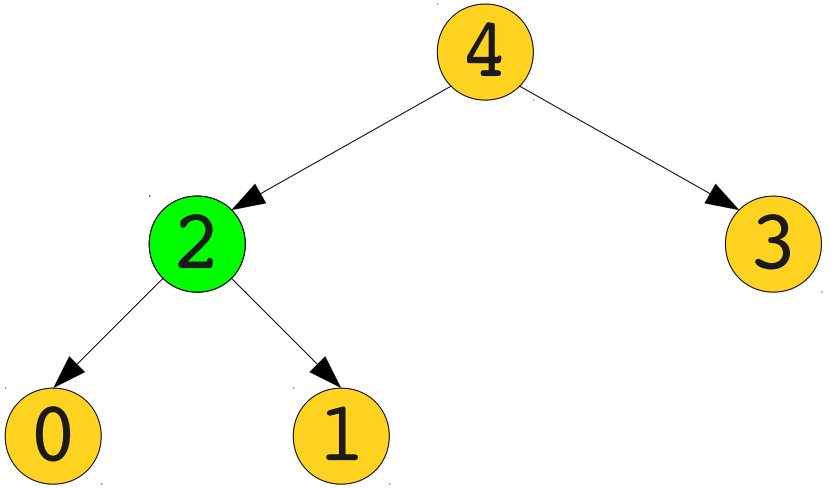


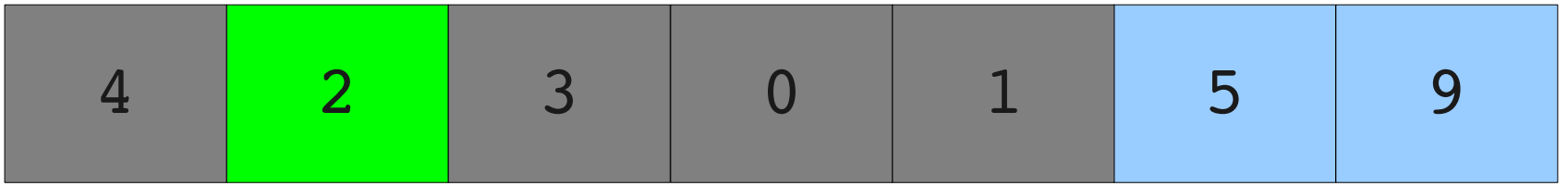
Sorting with Implicit Binary Heaps



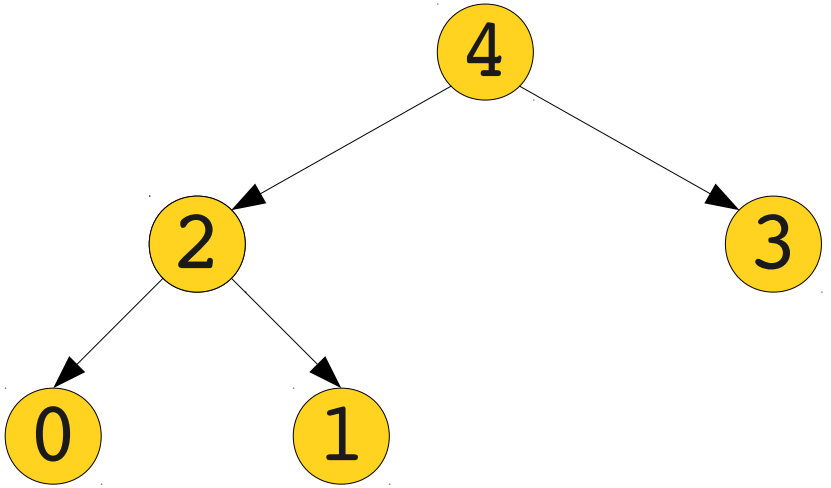


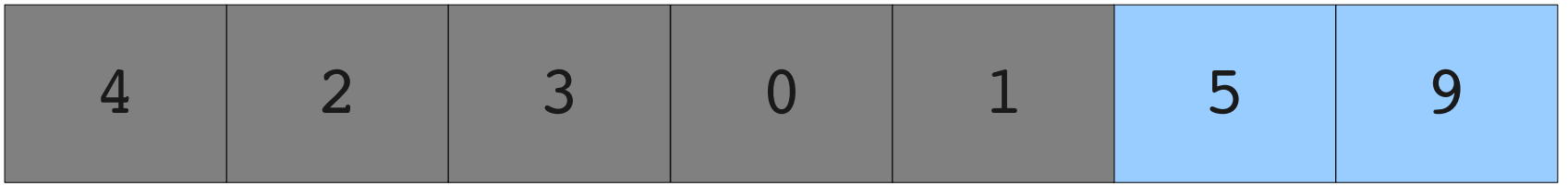
Sorting with Implicit Binary Heaps



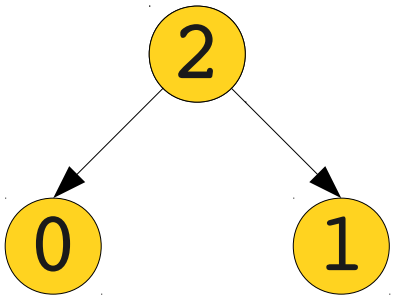


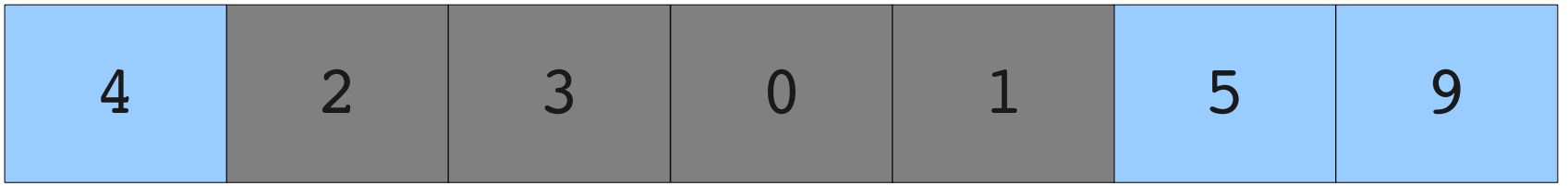
Sorting with Implicit Binary Heaps



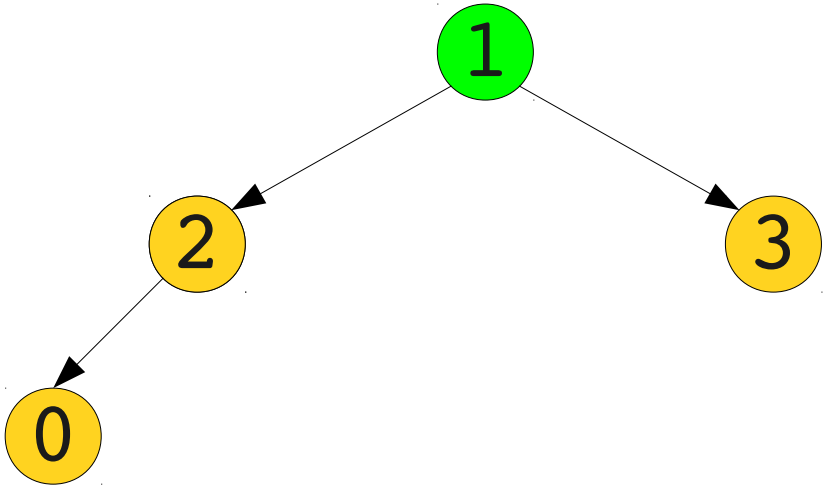


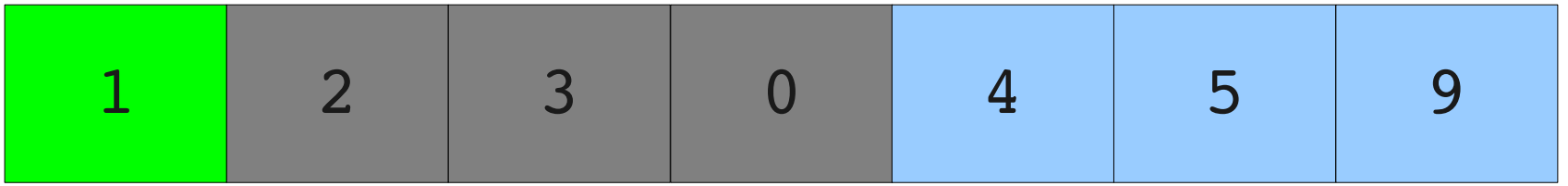
Sorting with Implicit Binary Heaps



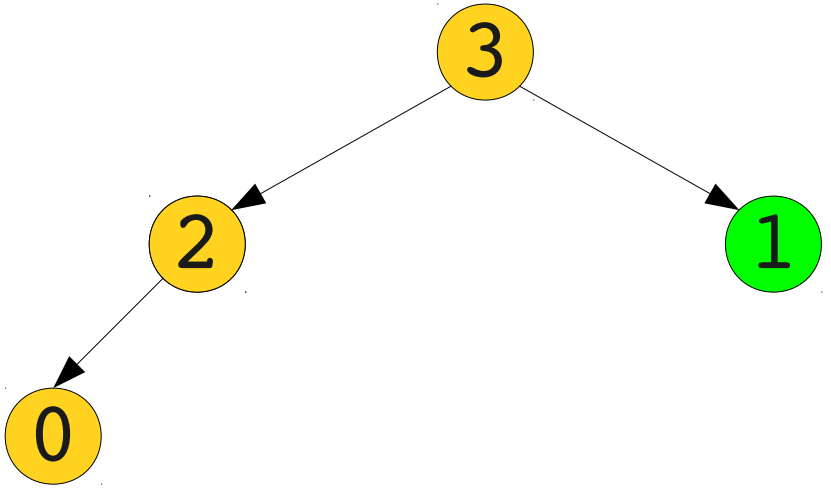


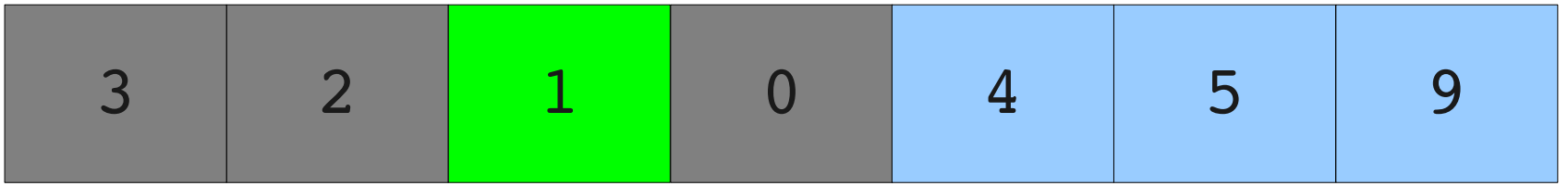
Sorting with Implicit Binary Heaps



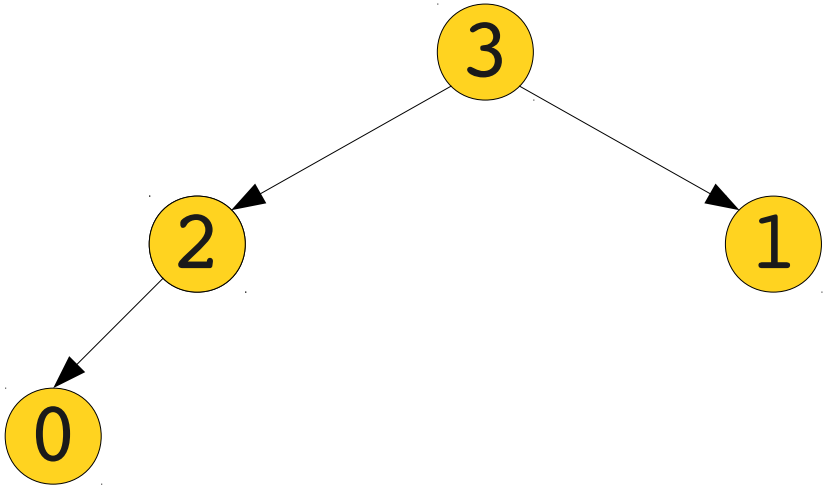


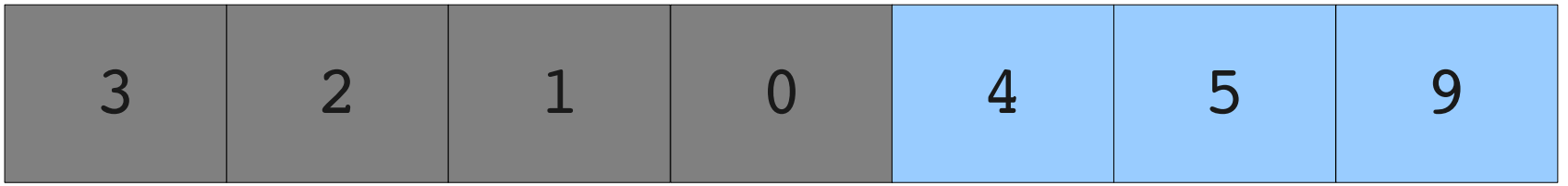
Sorting with Implicit Binary Heaps



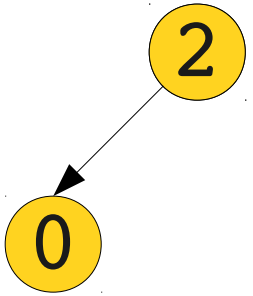


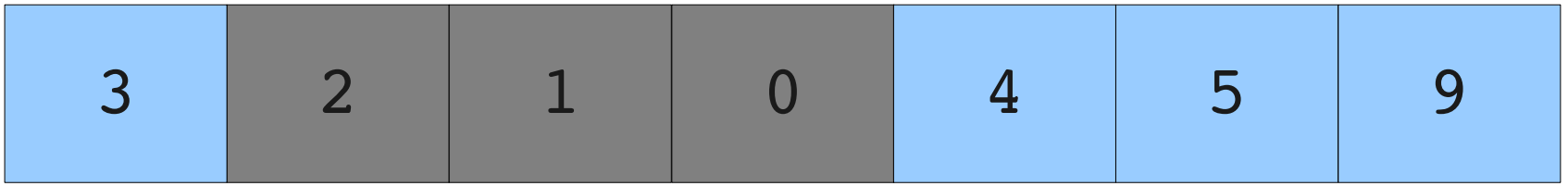
Sorting with Implicit Binary Heaps



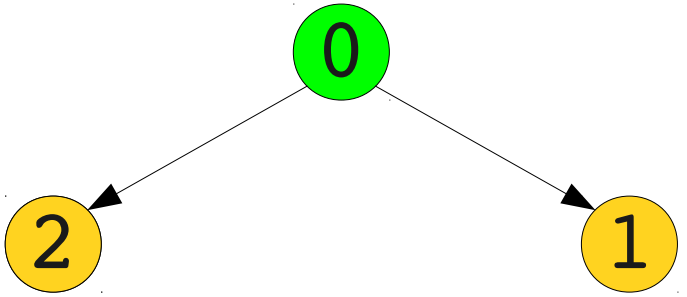


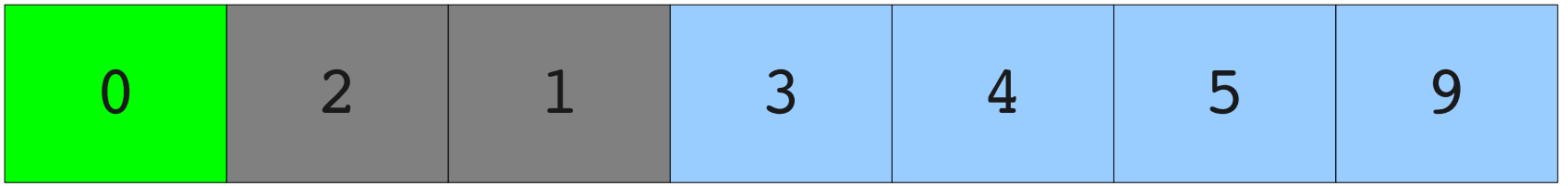
Sorting with Implicit Binary Heaps



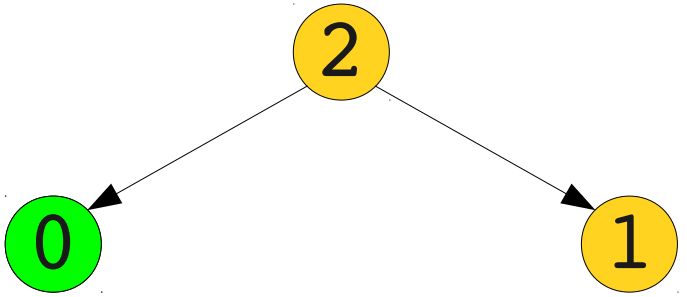


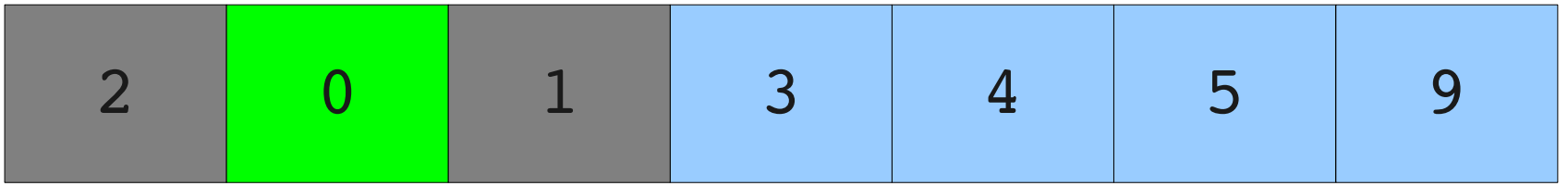
Sorting with Implicit Binary Heaps



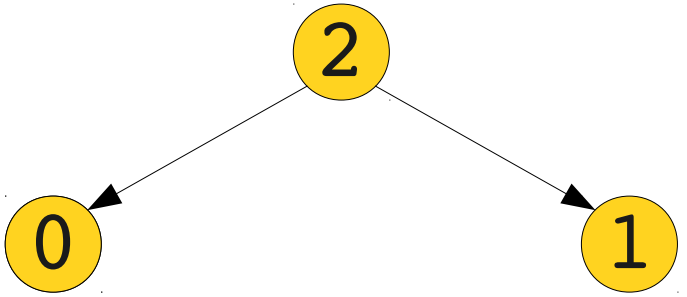


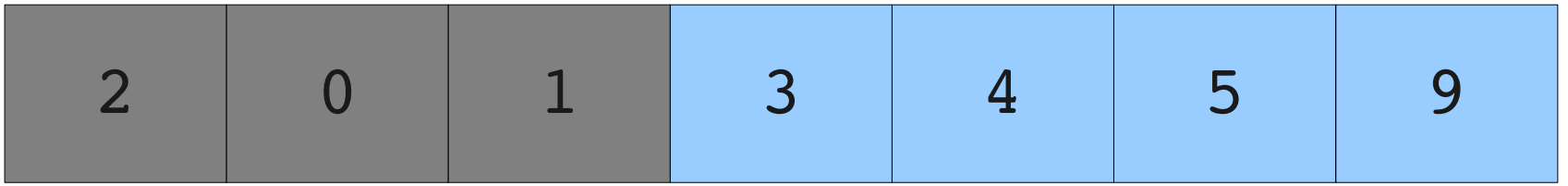
Sorting with Implicit Binary Heaps



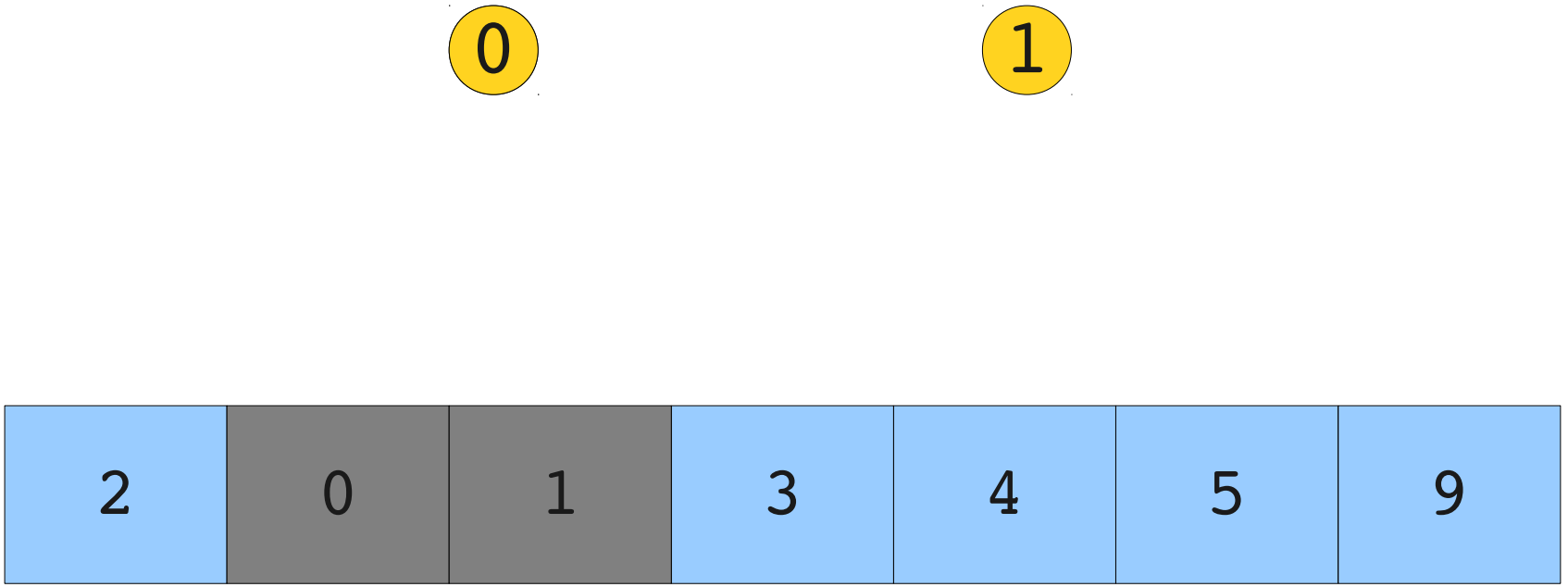


Sorting with Implicit Binary Heaps

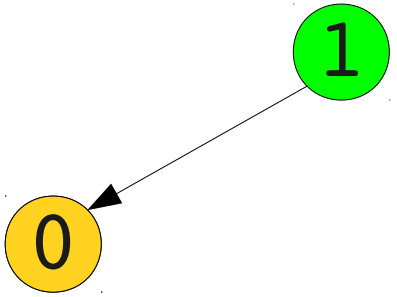


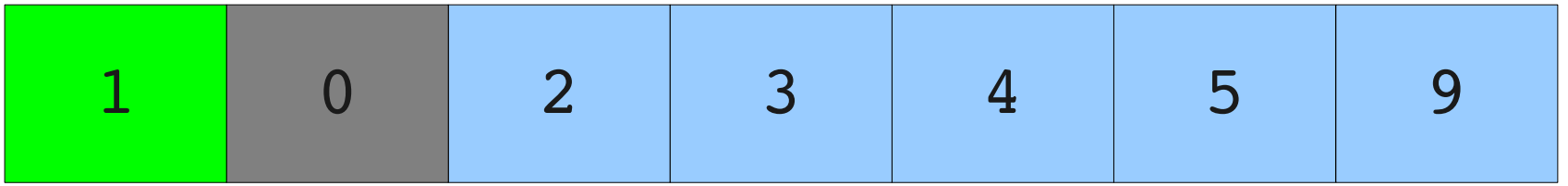


Sorting with Implicit Binary Heaps

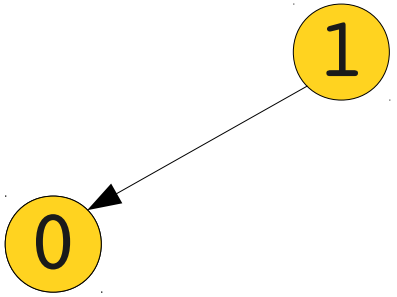


Sorting with Implicit Binary Heaps



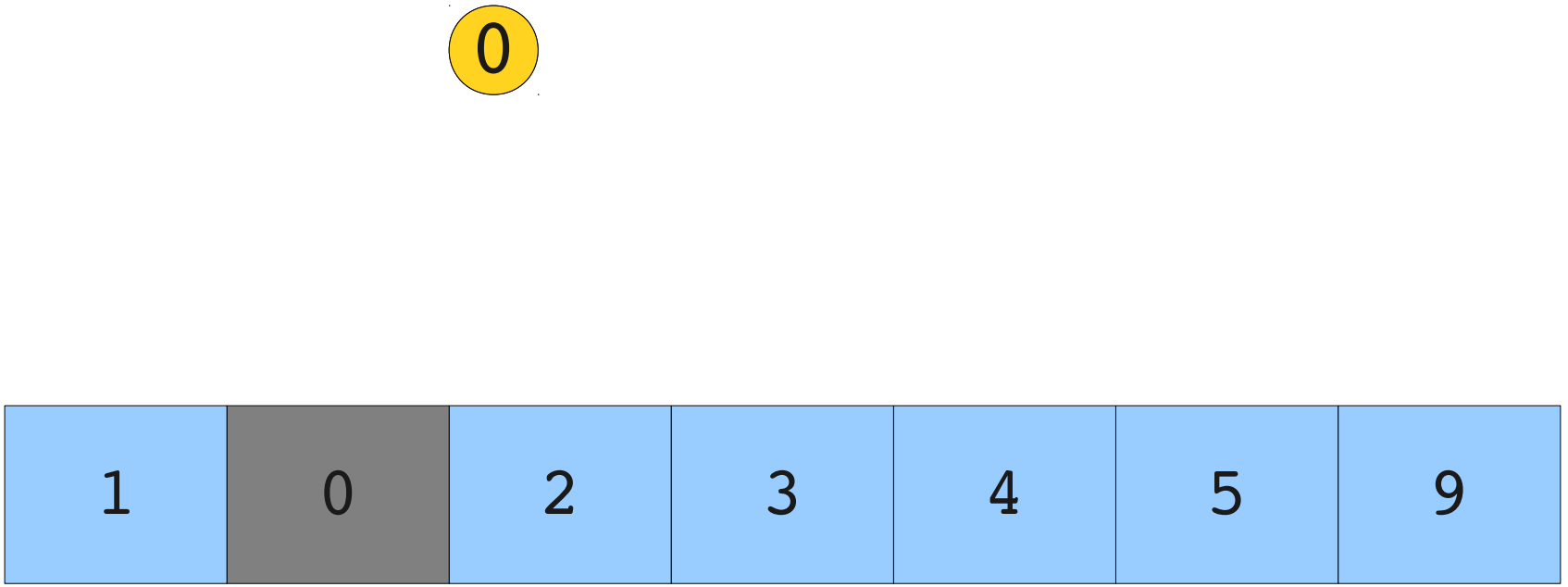


Sorting with Implicit Binary Heaps

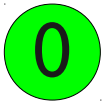


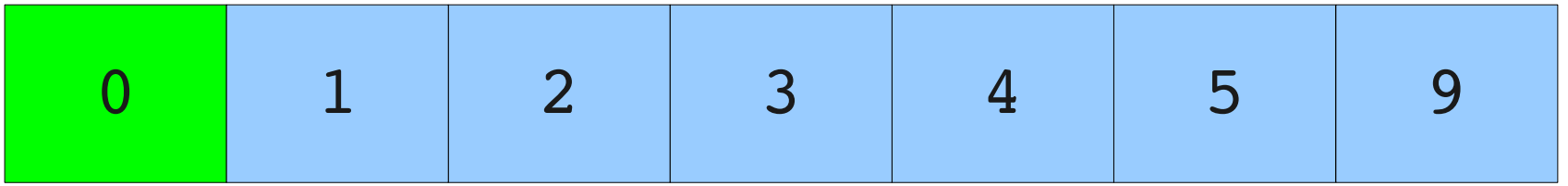
1	0	2	3	4	5	9

Sorting with Implicit Binary Heaps

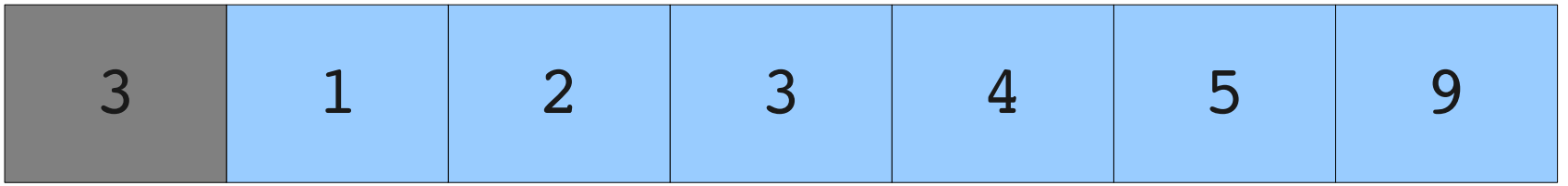


Sorting with Implicit Binary Heaps

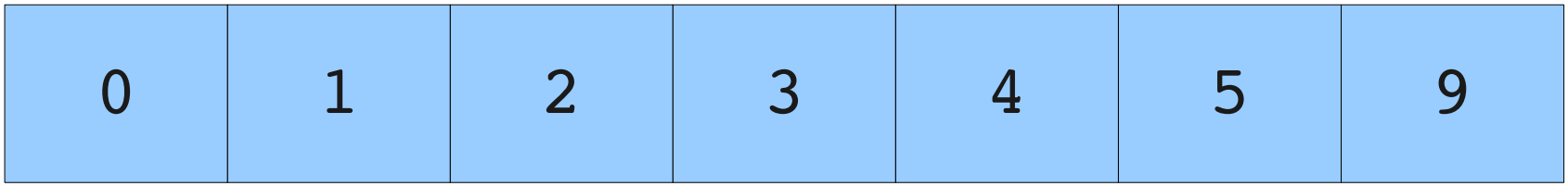




Sorting with Implicit Binary Heaps



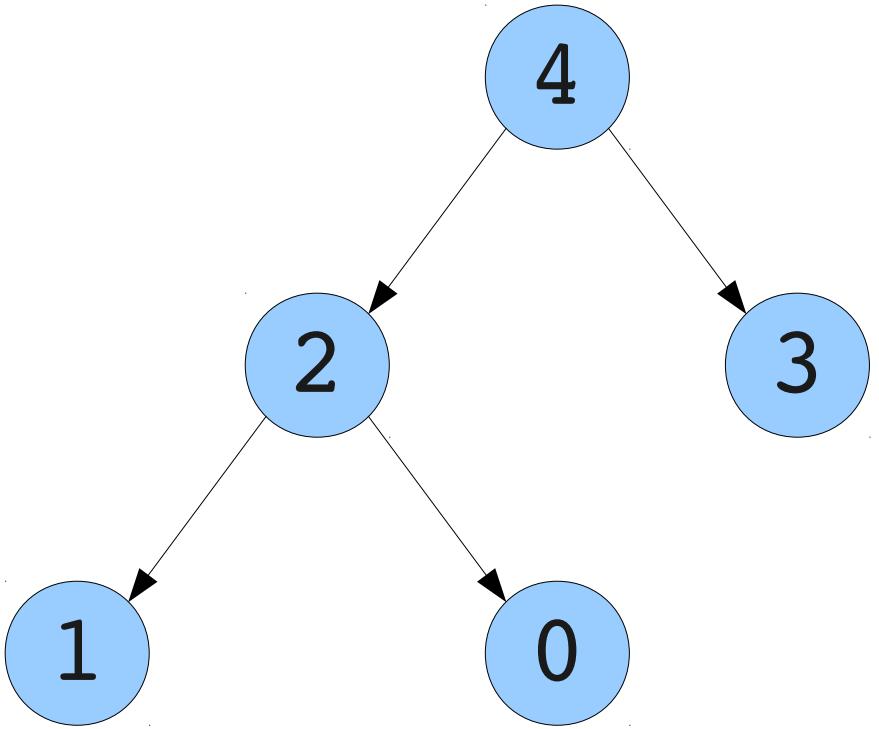
Sorting with Implicit Binary Heaps



Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Binary Heap	Naïve Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Implicit Binary Heap	Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$

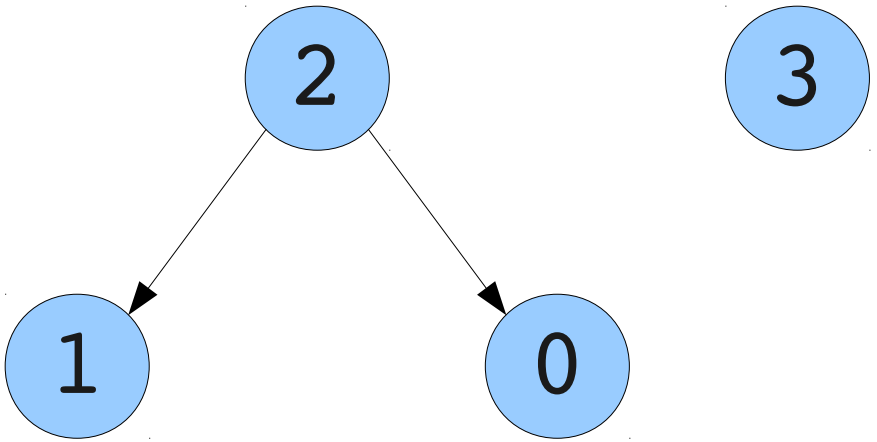
Why is Heapsort Ω(n lg n)?

Need to repair the heap at each step...



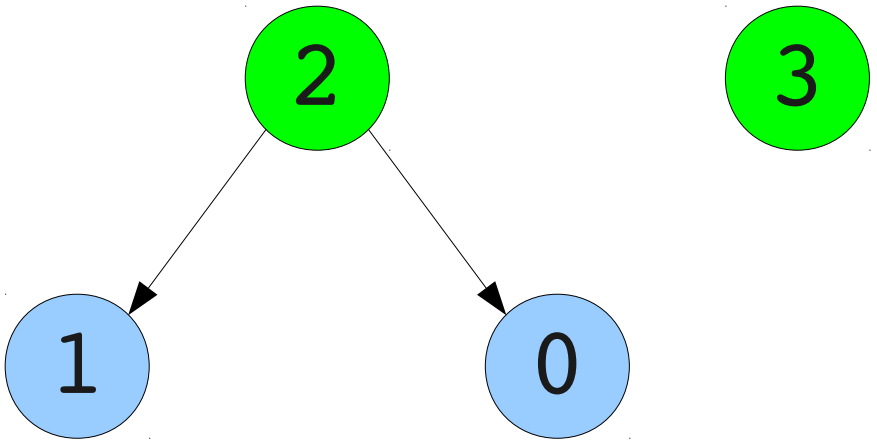
Why is Heapsort Ω(n lg n)?

Need to repair the heap at each step...



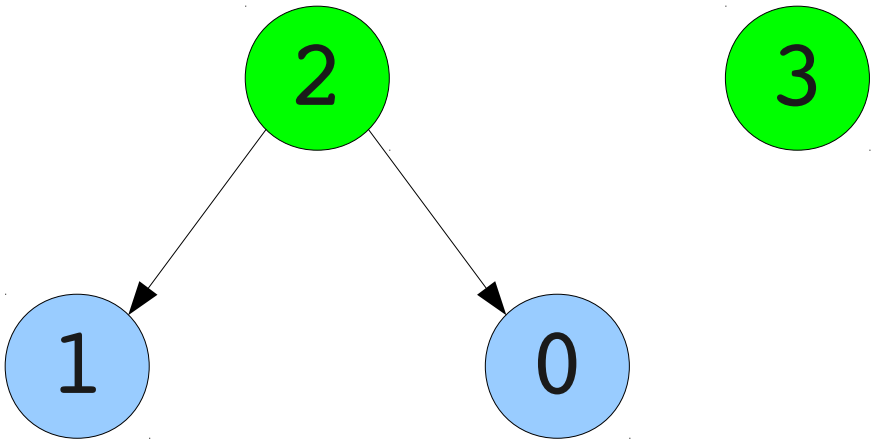
Why is Heapsort Ω(n lg n)?

Need to repair the heap at each step...



Why is Heapsort Ω(n lg n)?

Need to repair the heap at each step...



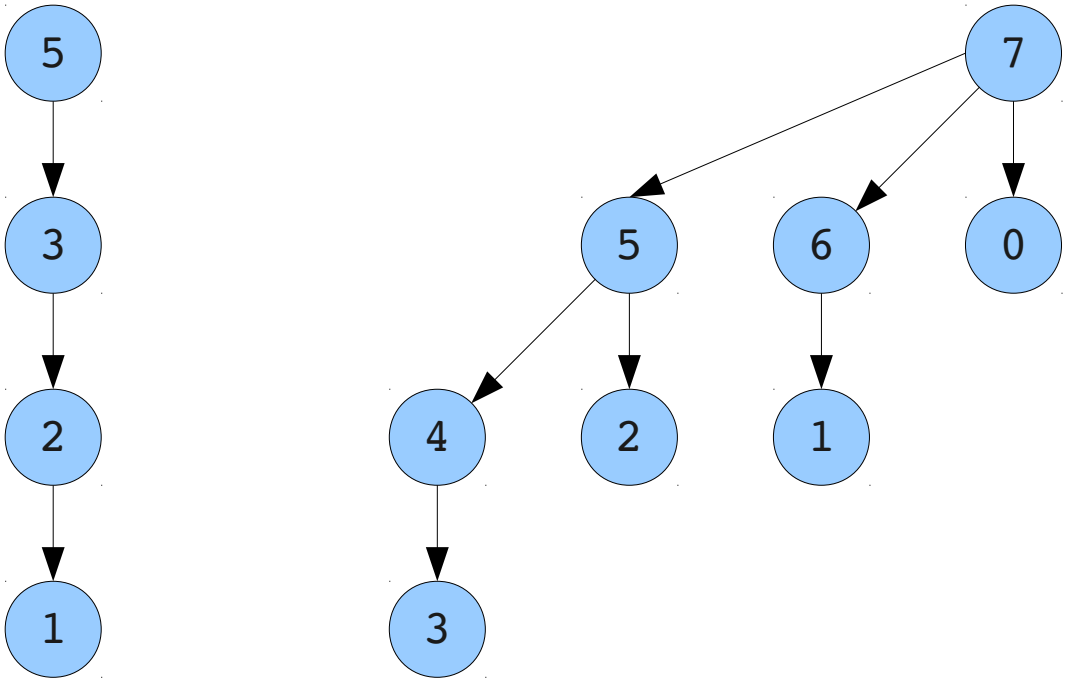
...or we can't find the max fast enough.

Idea: A Family of Heaps

- Rather than using a single max-heap, have a **family** of max-heaps.
- Order heaps by their max value.
- Keep number of heaps low (say, O(lg n)).
- How can we do this?

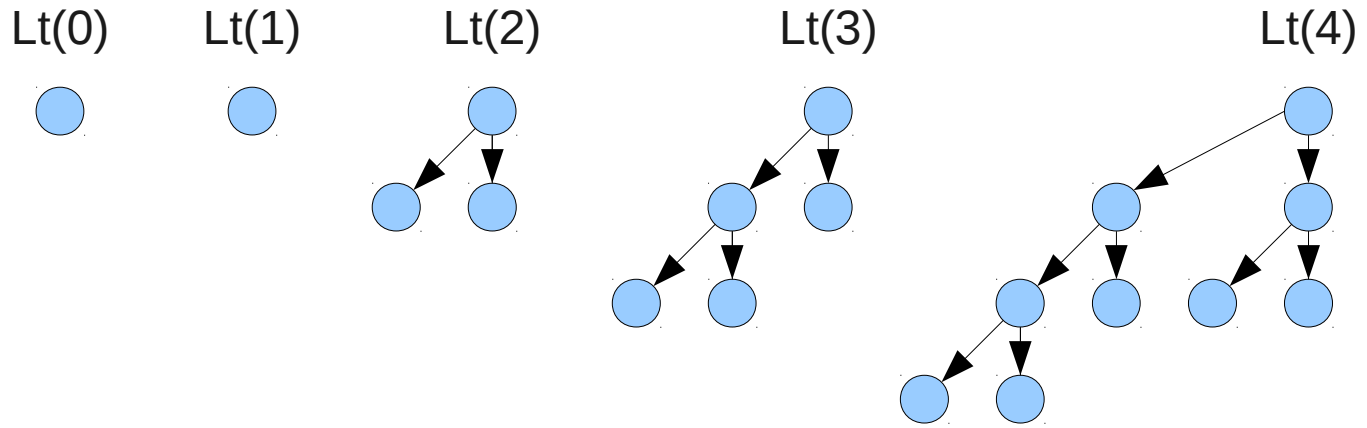
Changing the Heap Shape

- Binary heap is a complete binary tree obeying the **heap property**.
- Other types of trees can also be heaps:



Leonardo Trees

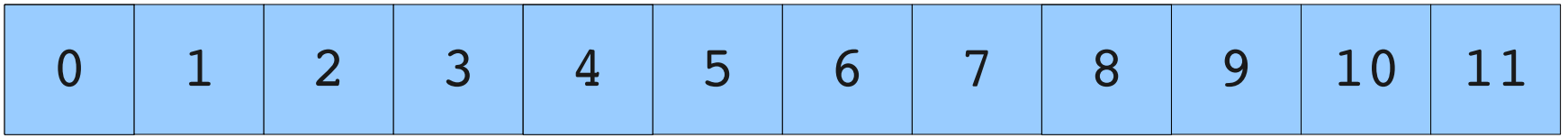
- Defined recursively:
  - Lt(0) is a single node
  - Lt(1) is a single node.
  - Lt(n + 2) is a node whose left tree is Lt(n + 1) and whose right tree is Lt(n)



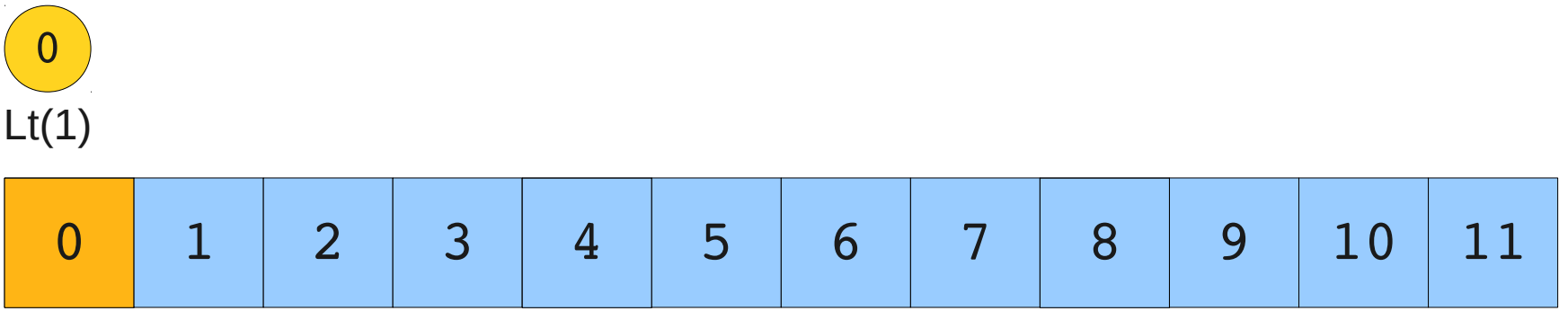
Why Leonardo Trees?

- Any sequence can be stored in a forest of Leonardo trees such that:
  - Each tree has **unique** order.
  - There are **at most** two trees of consecutive order.
- Examples:
  - 6 = 5 + 1 = Lt(3) + Lt(1)
  - 13 = 9 + 3 + 1 = Lt(4) + Lt(2) + Lt(1)
- Do this with **greedy algorithm**.

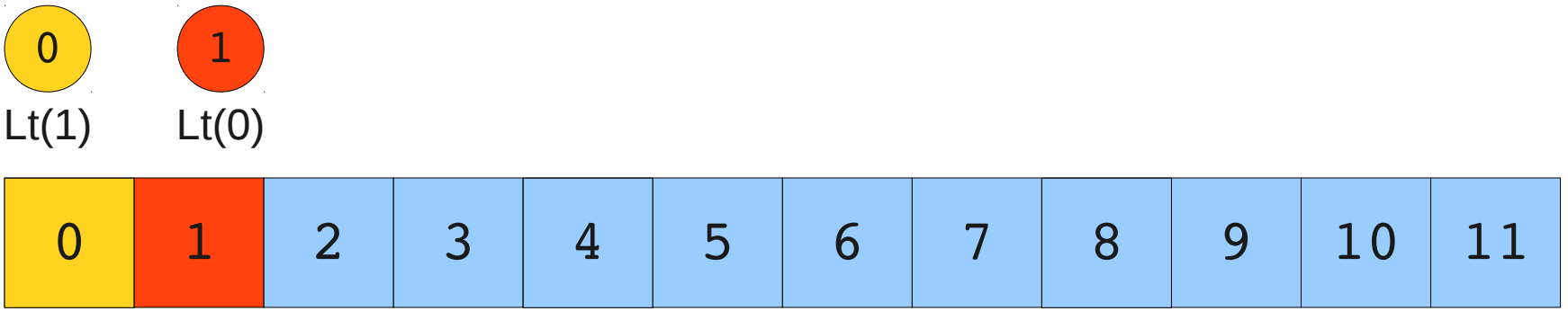
Partitioning a Sequence



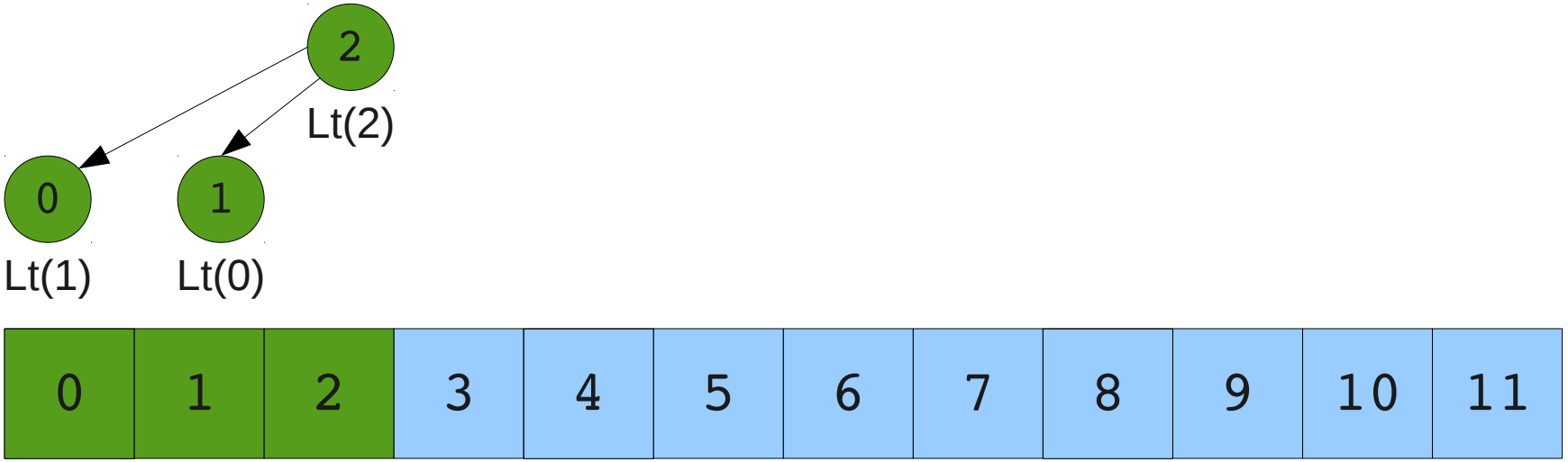
Partitioning a Sequence



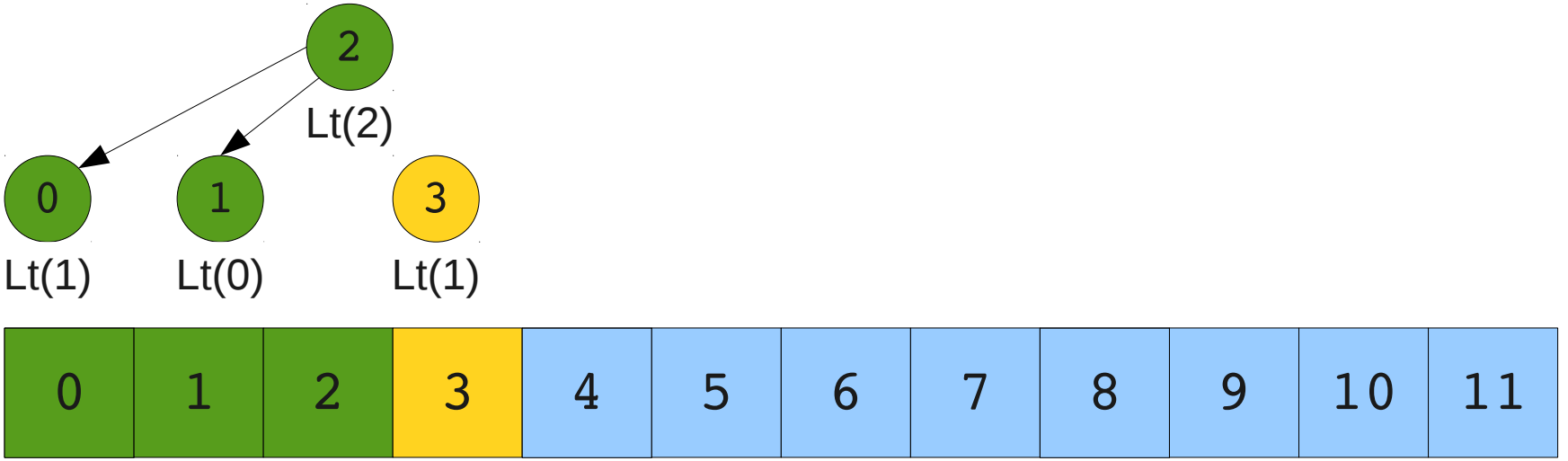
Partitioning a Sequence



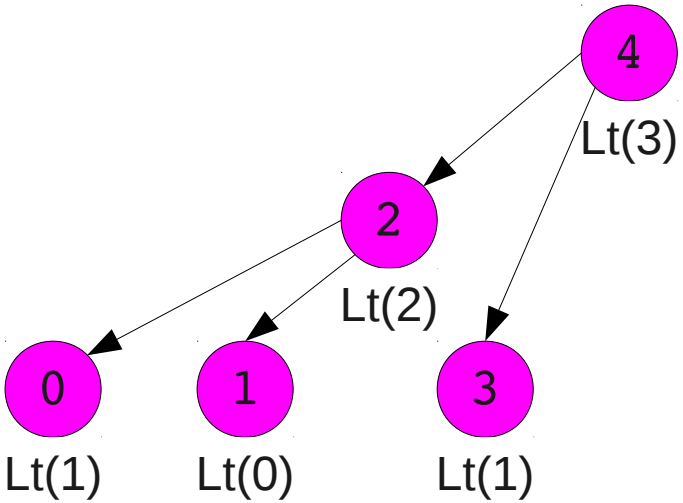
Partitioning a Sequence

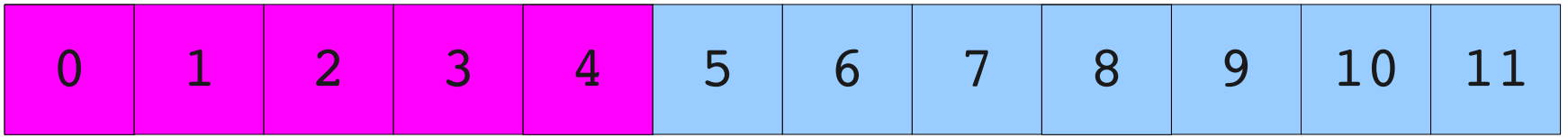


Partitioning a Sequence

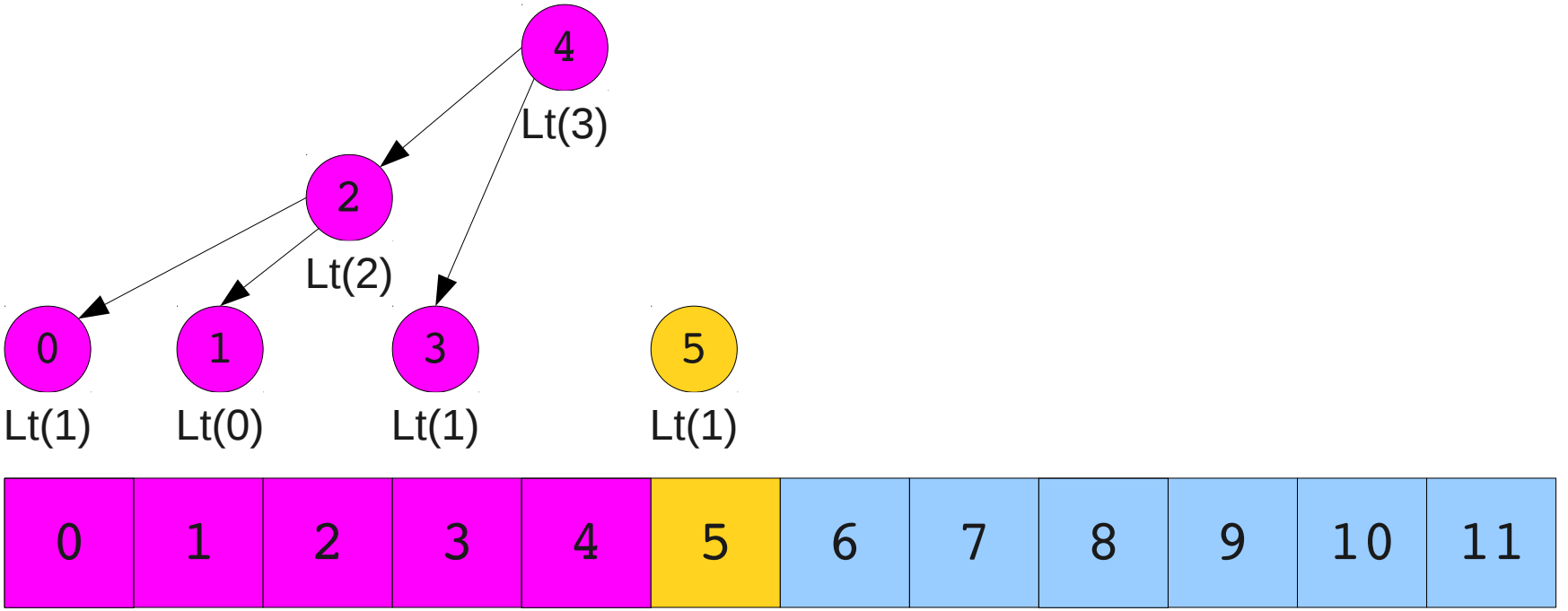


Partitioning a Sequence

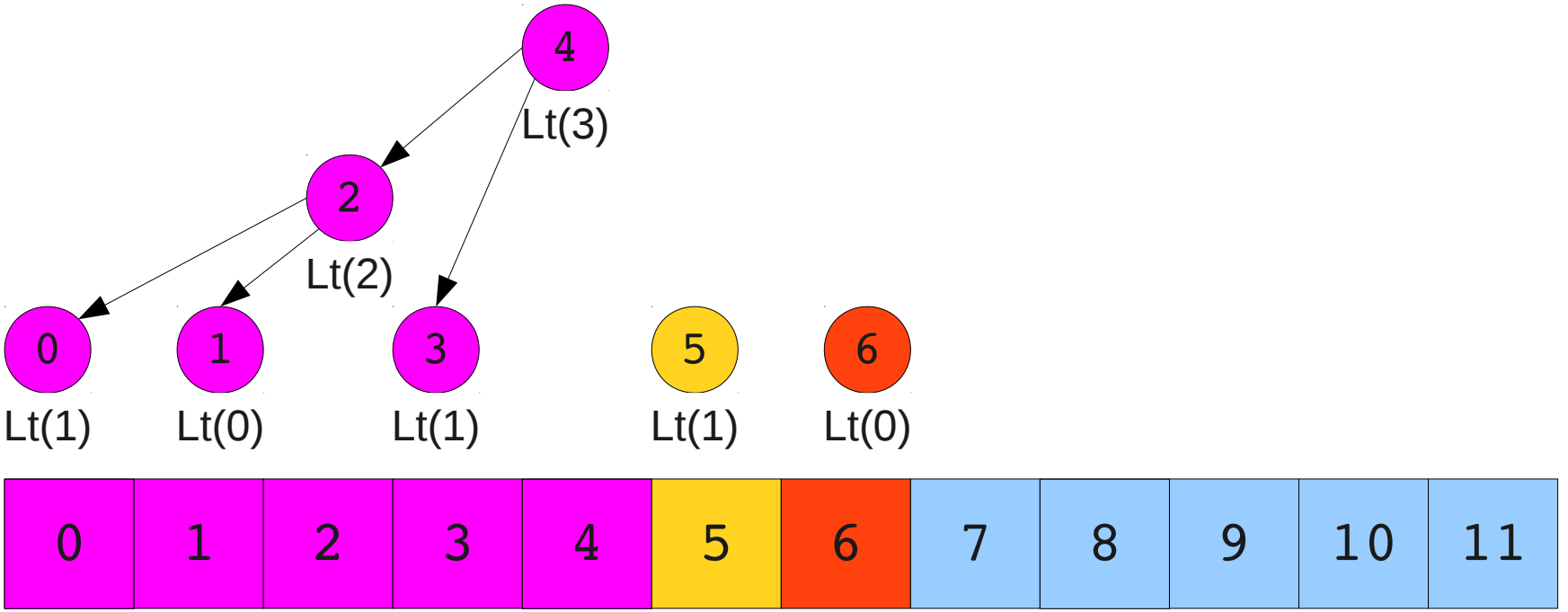




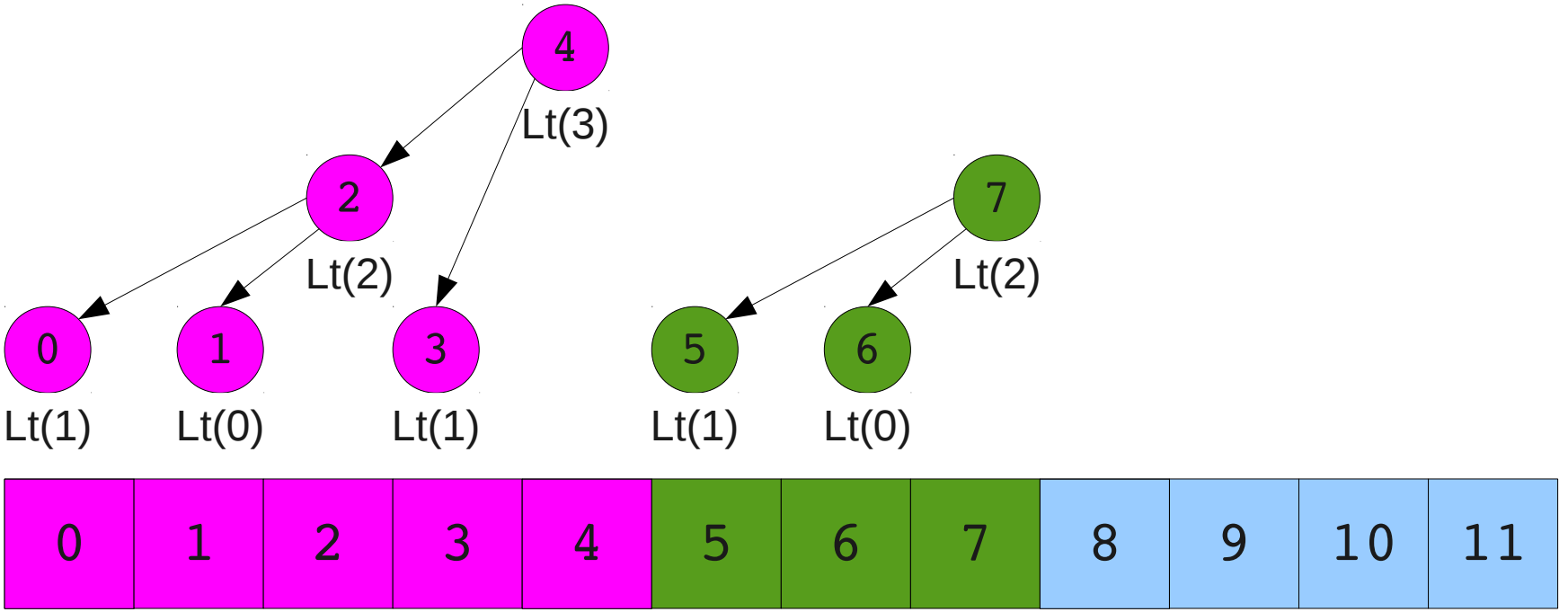
Partitioning a Sequence



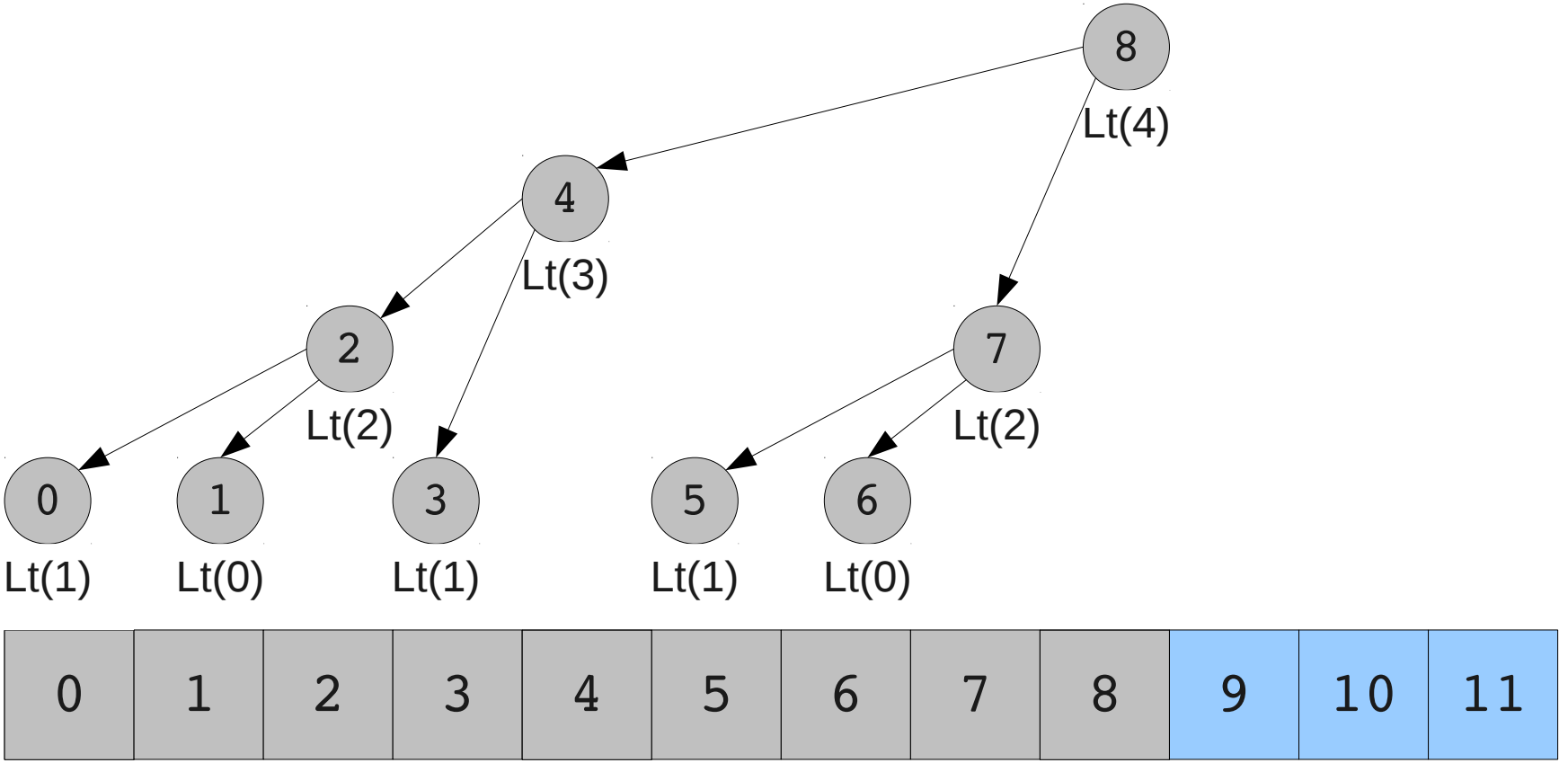
Partitioning a Sequence



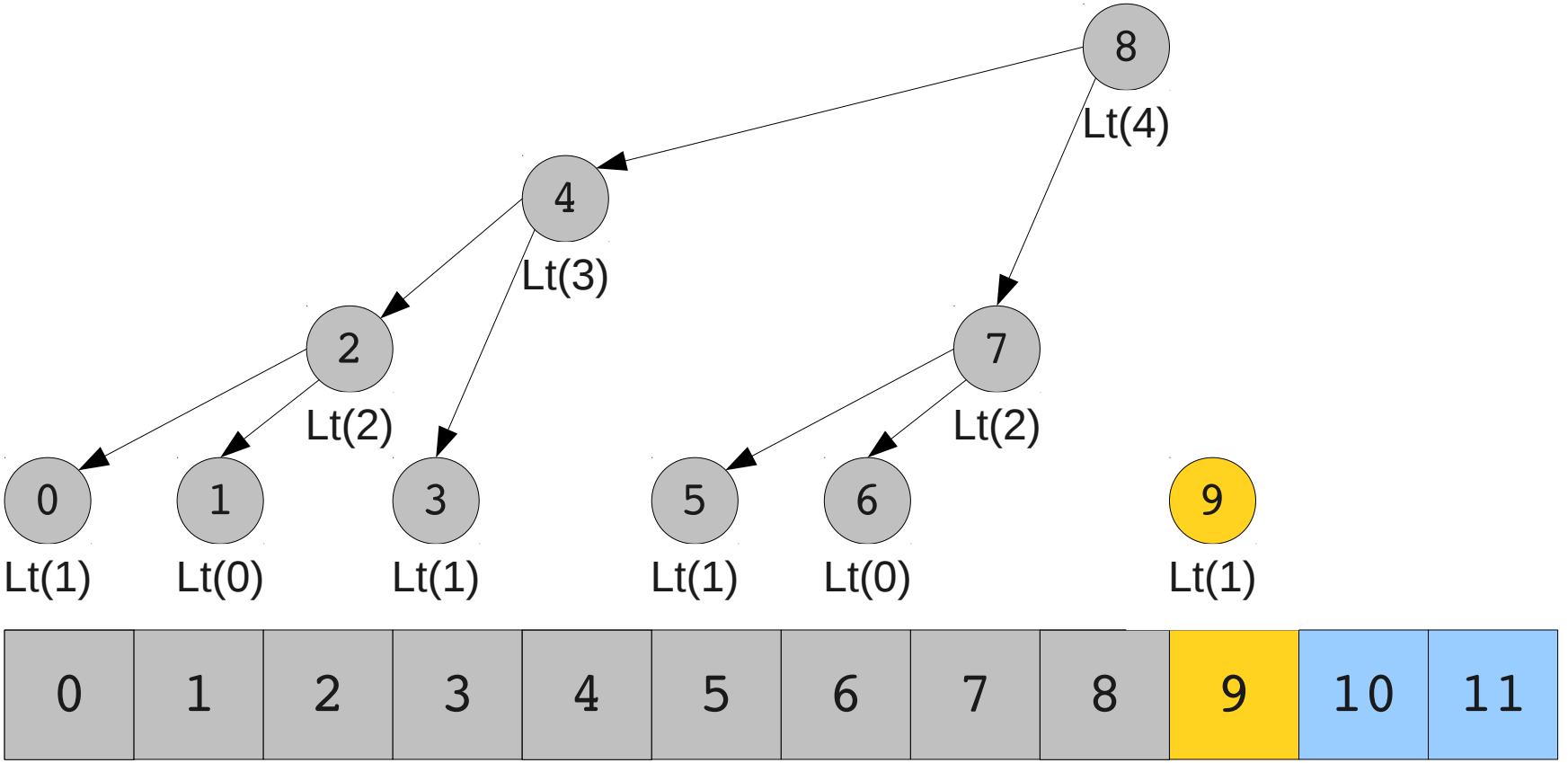
Partitioning a Sequence



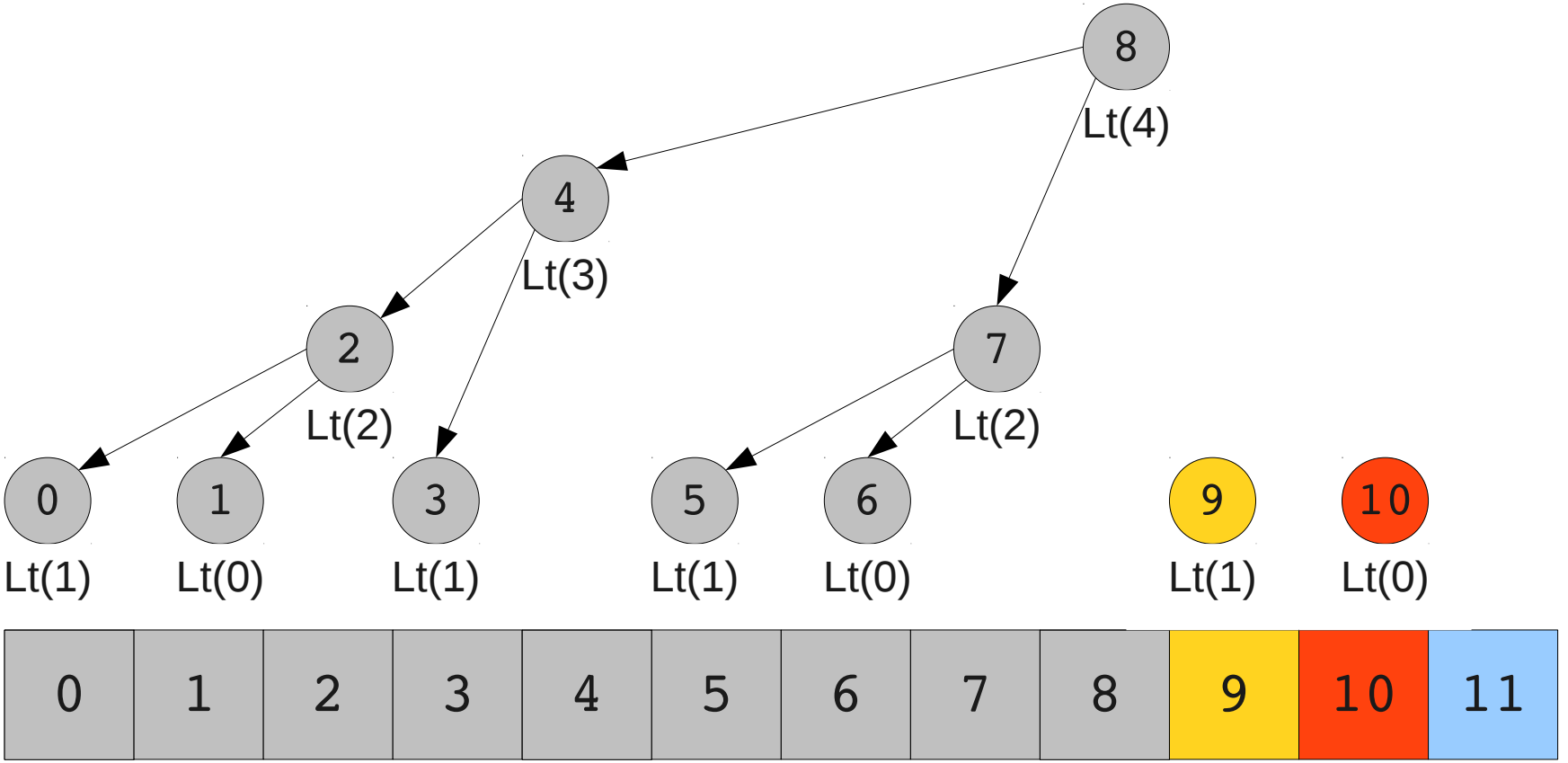
Partitioning a Sequence



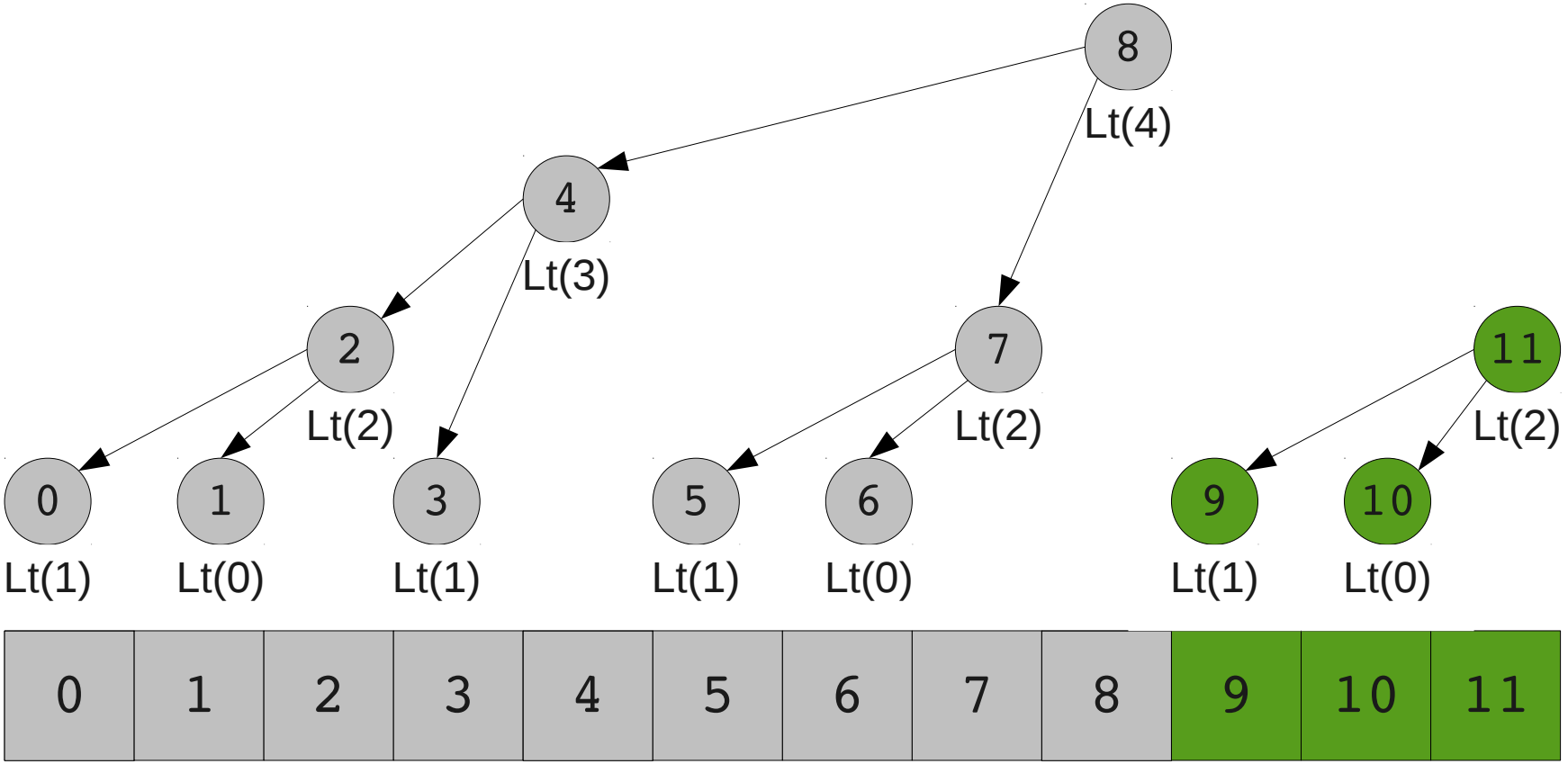
Partitioning a Sequence



Partitioning a Sequence



Partitioning a Sequence



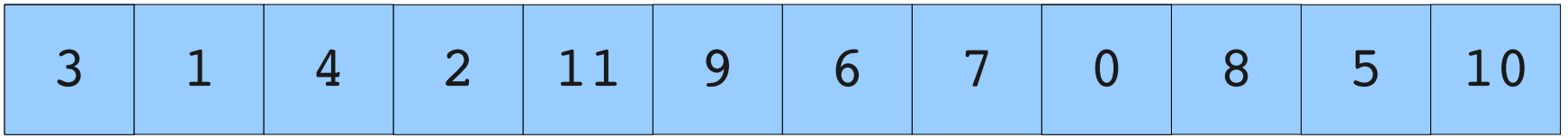
Partitioning Into Leonardo Trees

- If last two trees have adjacent order, merge them into one new tree.
- If last tree isn't order 1, add a tree of order 1.
- Otherwise, add a tree of order 0.

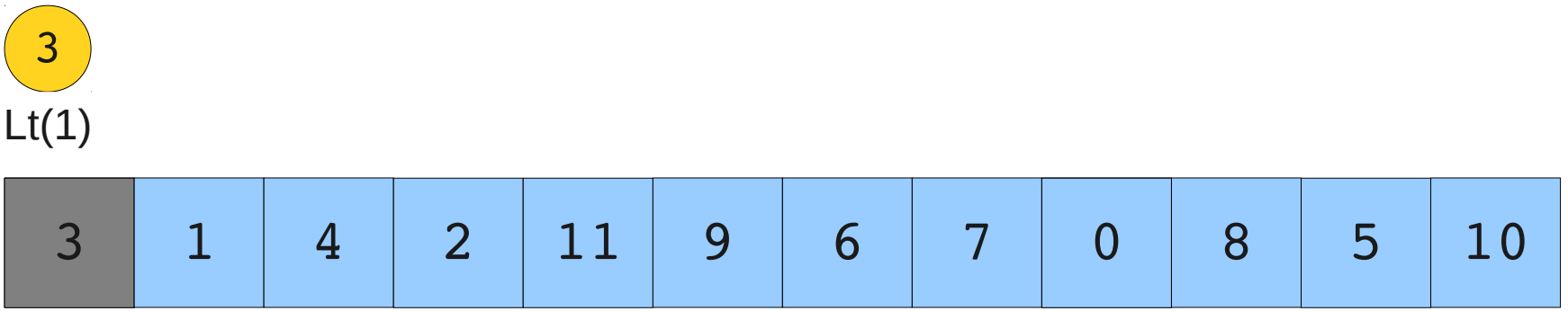
From Trees to Priority Queues

- Question – How to convert a family of trees into a priority queue?
- Two ideas:
  - Make each tree obey **max-heap** property.
  - Sort the roots in **ascending order**.
- Max element always the root of the rightmost (smallest) heap.

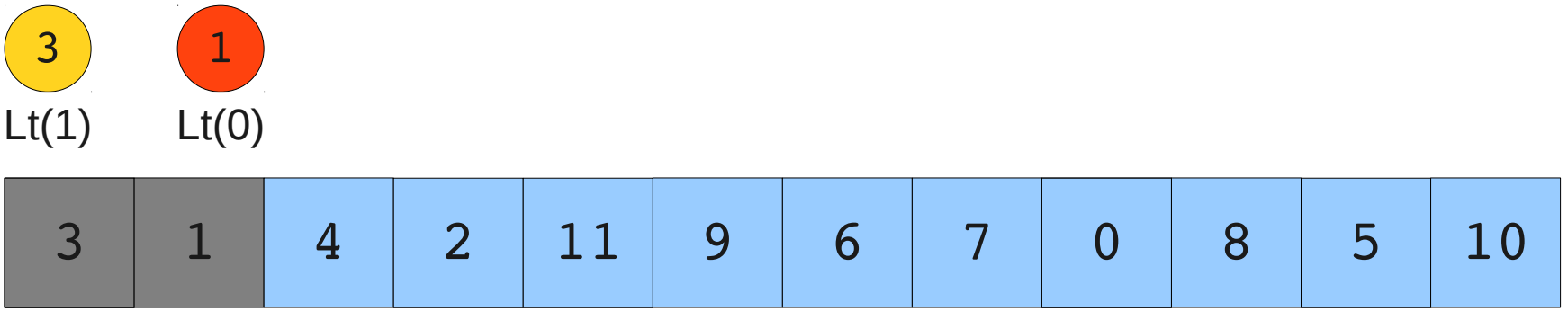
Building a Leonardo Heap



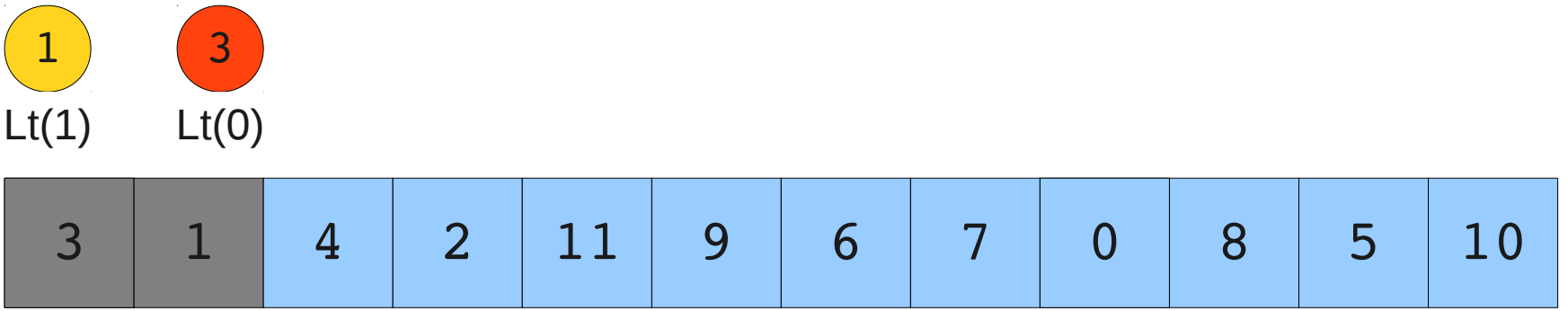
Building a Leonardo Heap



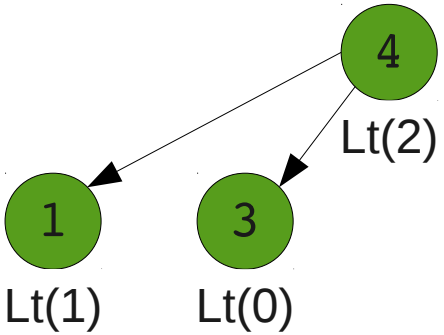
Building a Leonardo Heap



Building a Leonardo Heap

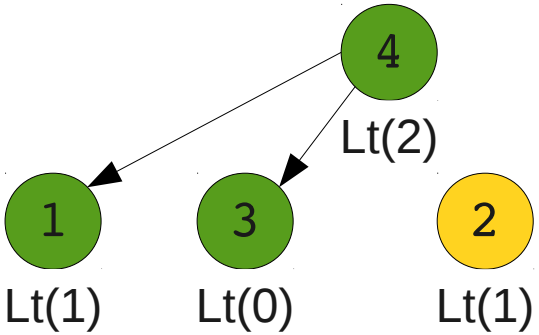


Building a Leonardo Heap



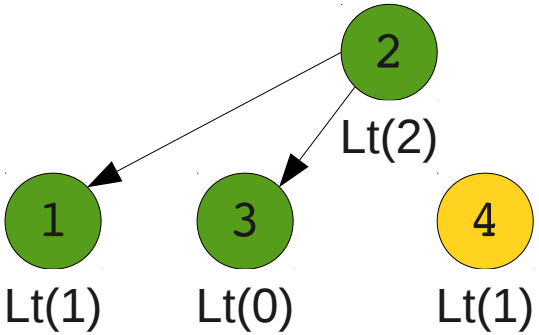
3	1	4	2	11	9	6	7	0	8	5	10

Building a Leonardo Heap



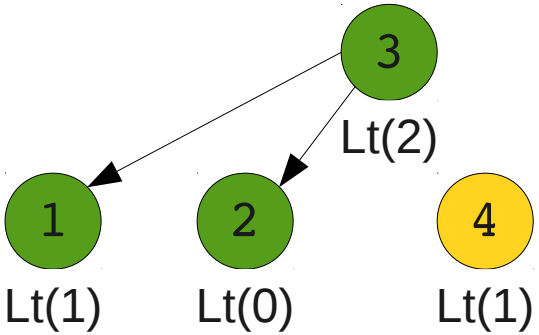
3	1	4	2	11	9	6	7	0	8	5	10

Building a Leonardo Heap



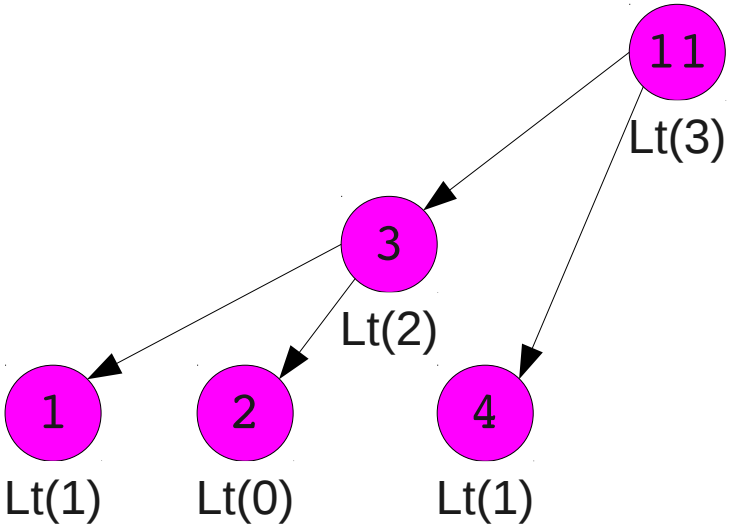
3	1	4	2	11	9	6	7	0	8	5	10

Building a Leonardo Heap



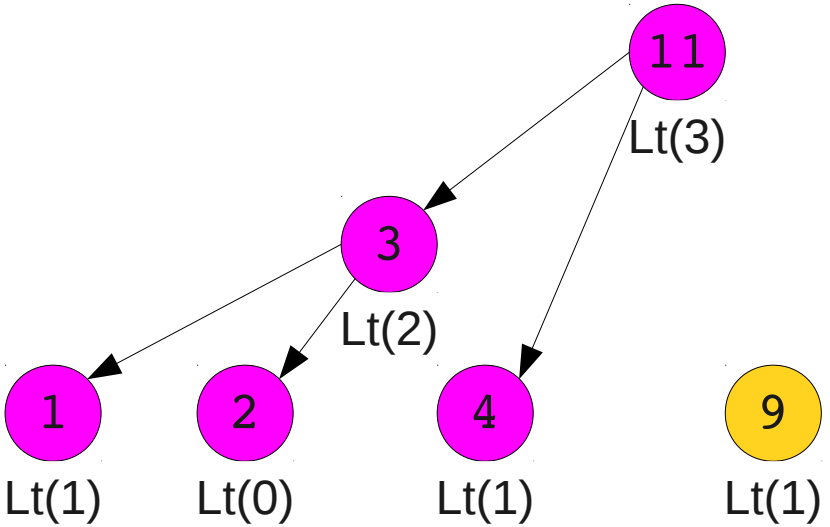
3	1	4	2	11	9	6	7	0	8	5	10

Building a Leonardo Heap



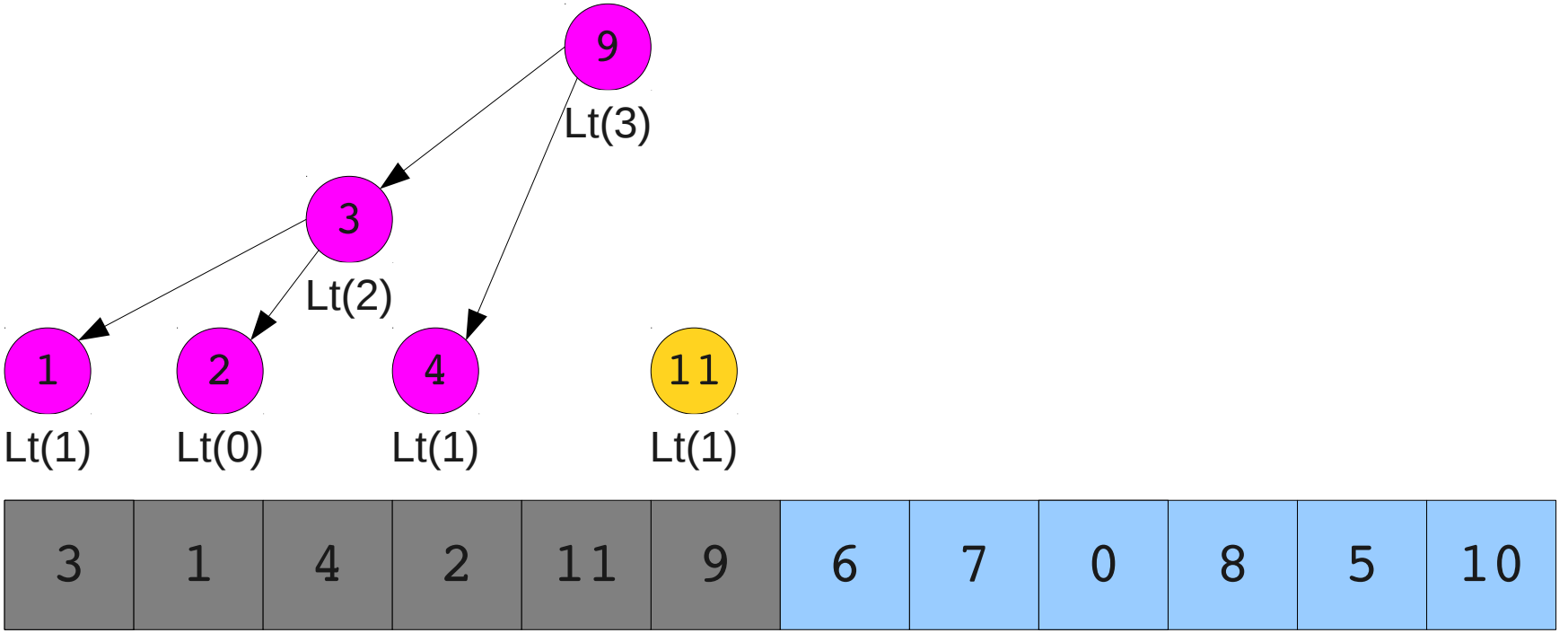
3	1	4	2	11	9	6	7	0	8	5	10

Building a Leonardo Heap

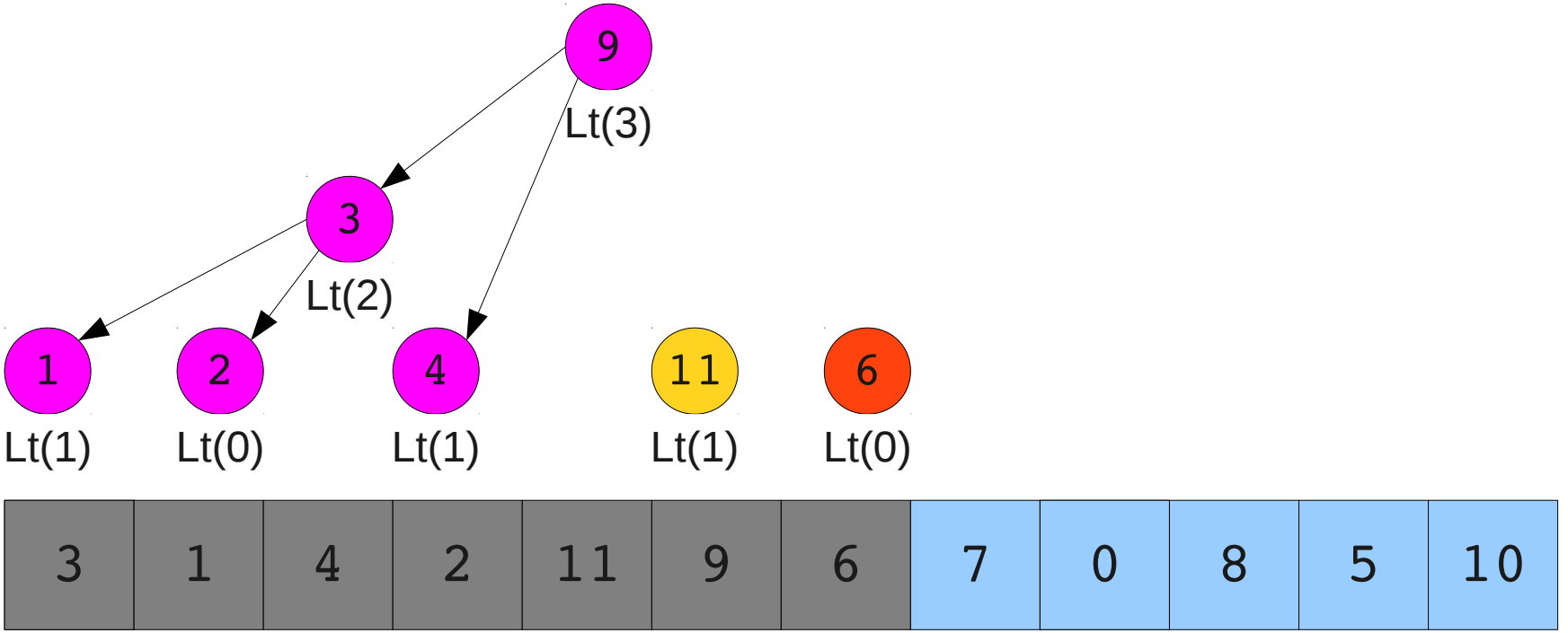


3	1	4	2	11	9	6	7	0	8	5	10

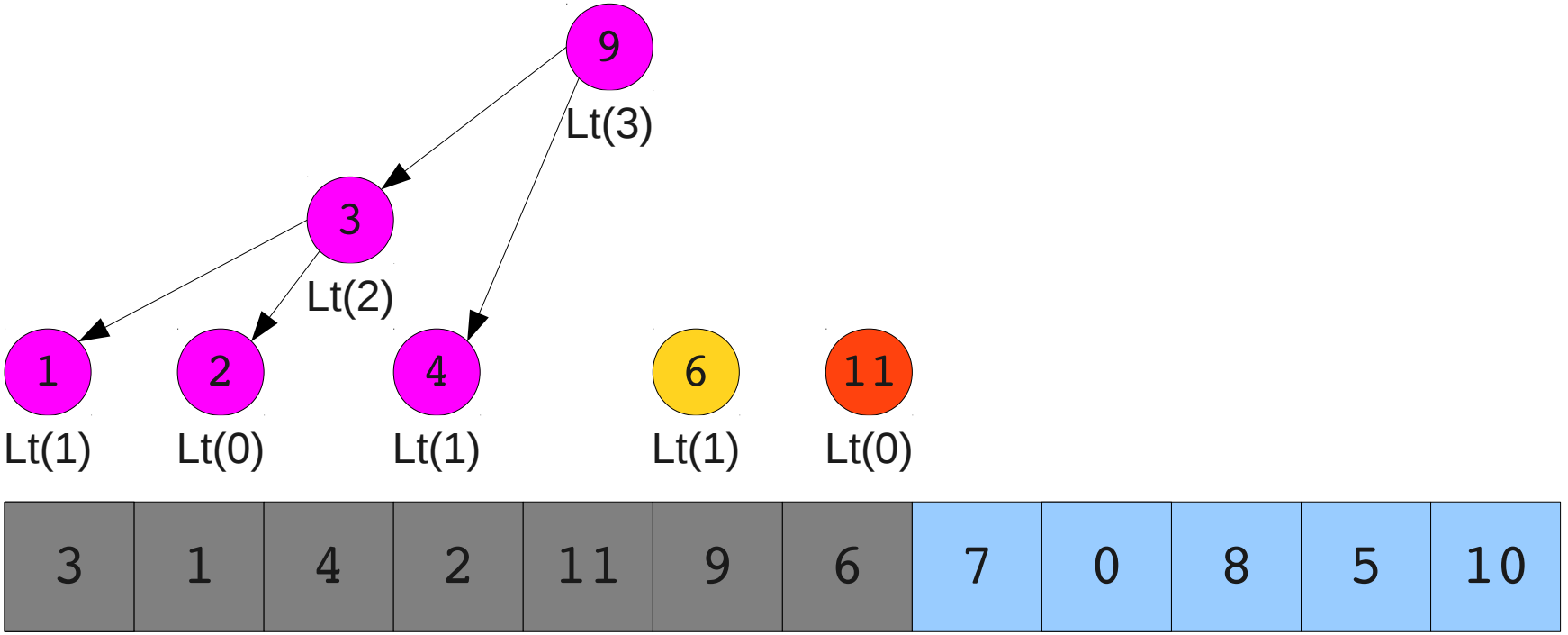
Building a Leonardo Heap



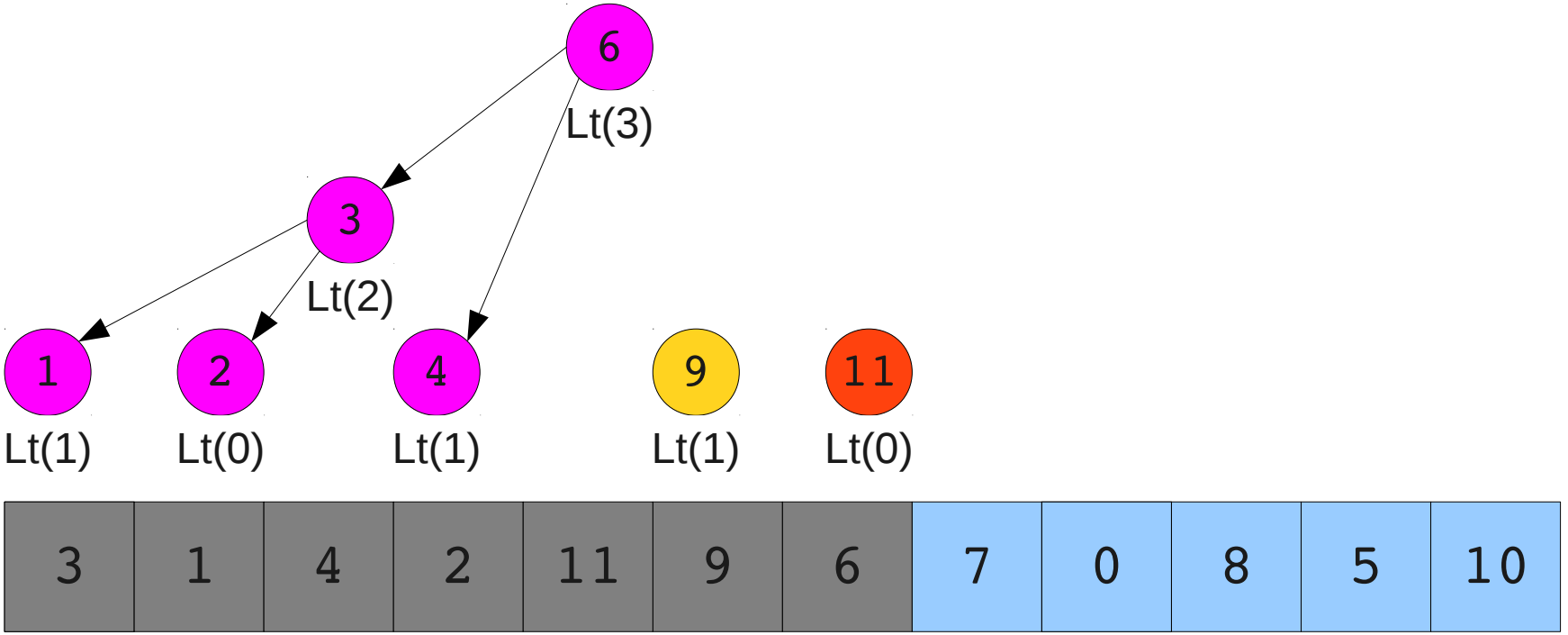
Building a Leonardo Heap



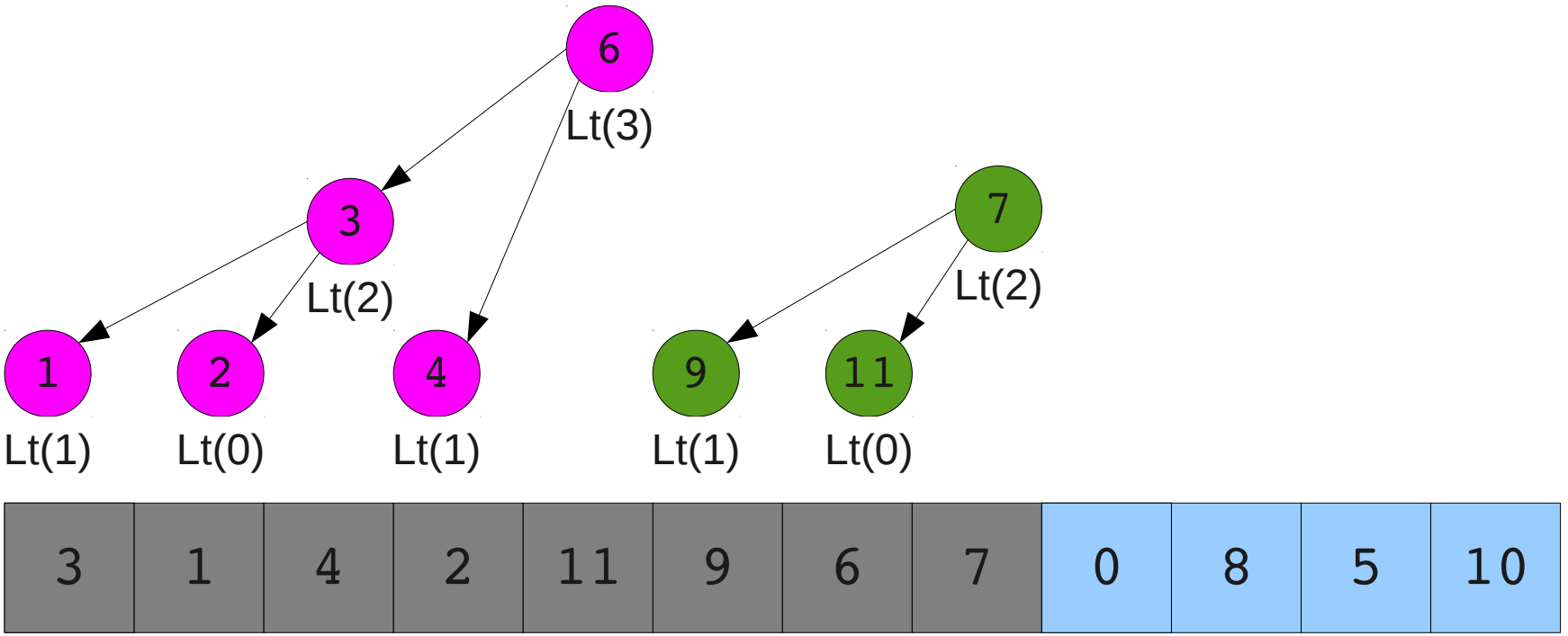
Building a Leonardo Heap



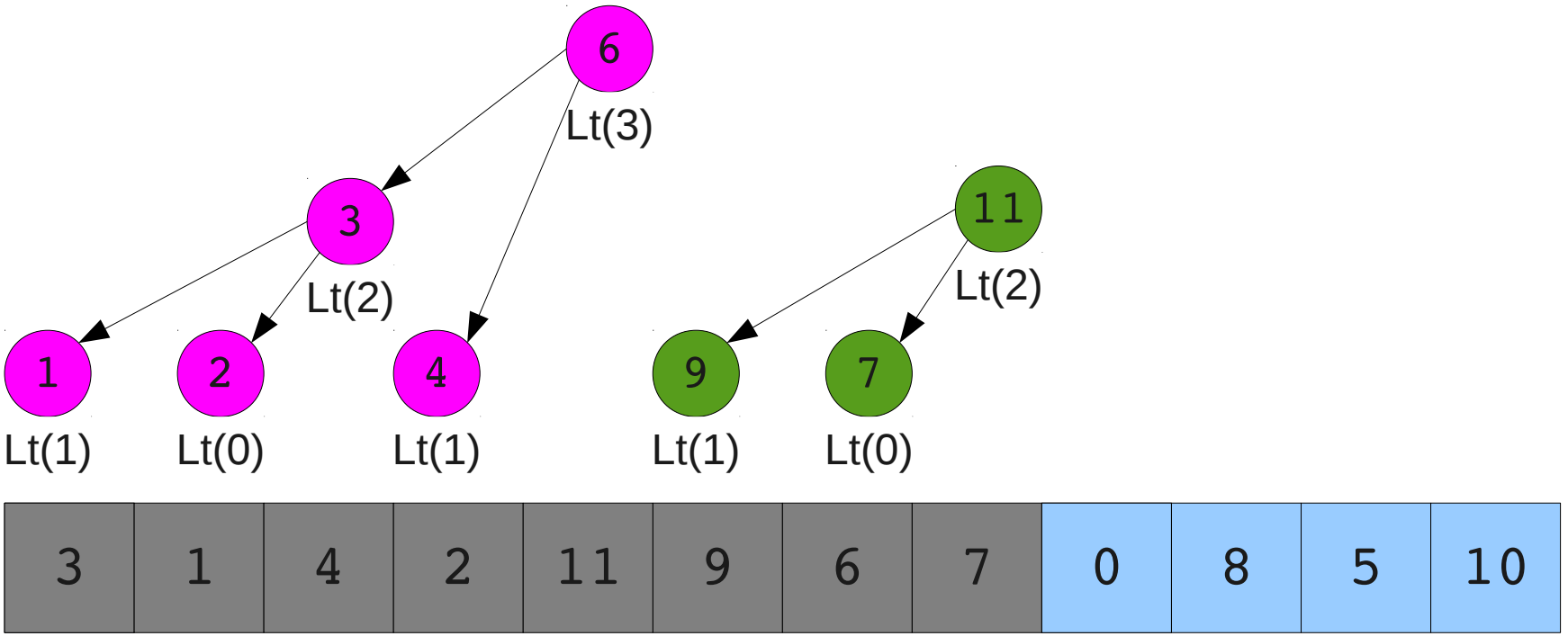
Building a Leonardo Heap



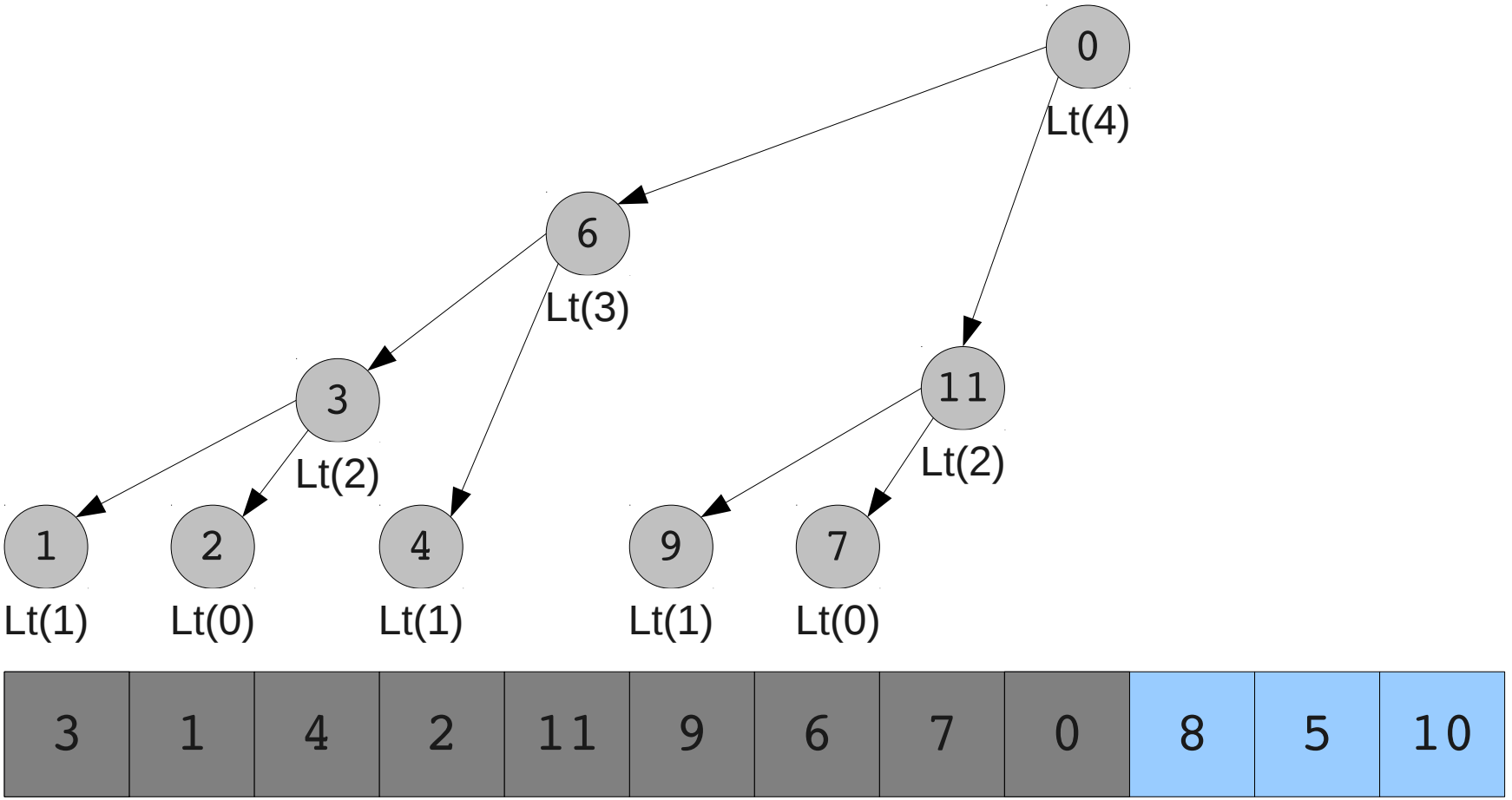
Building a Leonardo Heap



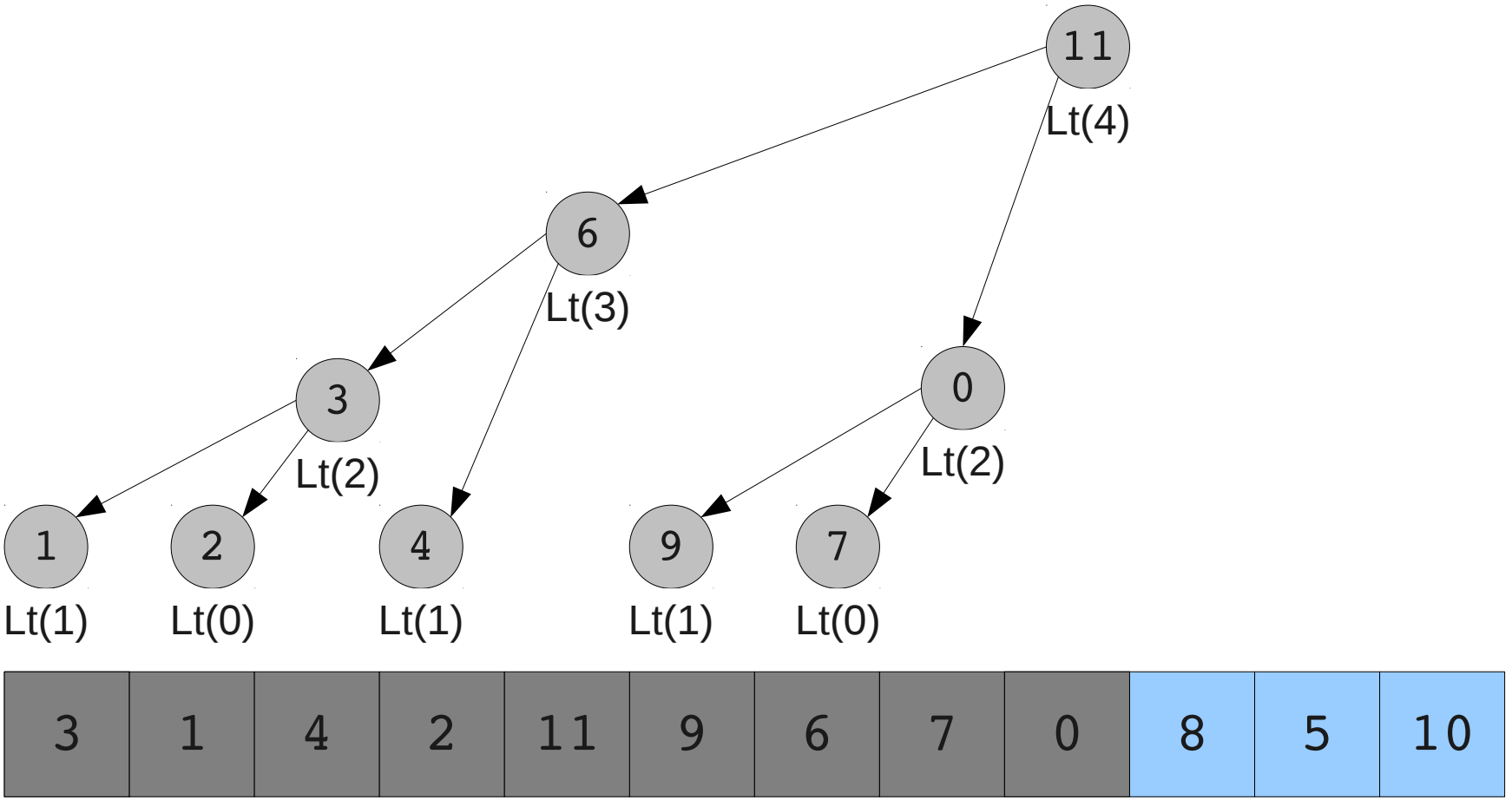
Building a Leonardo Heap



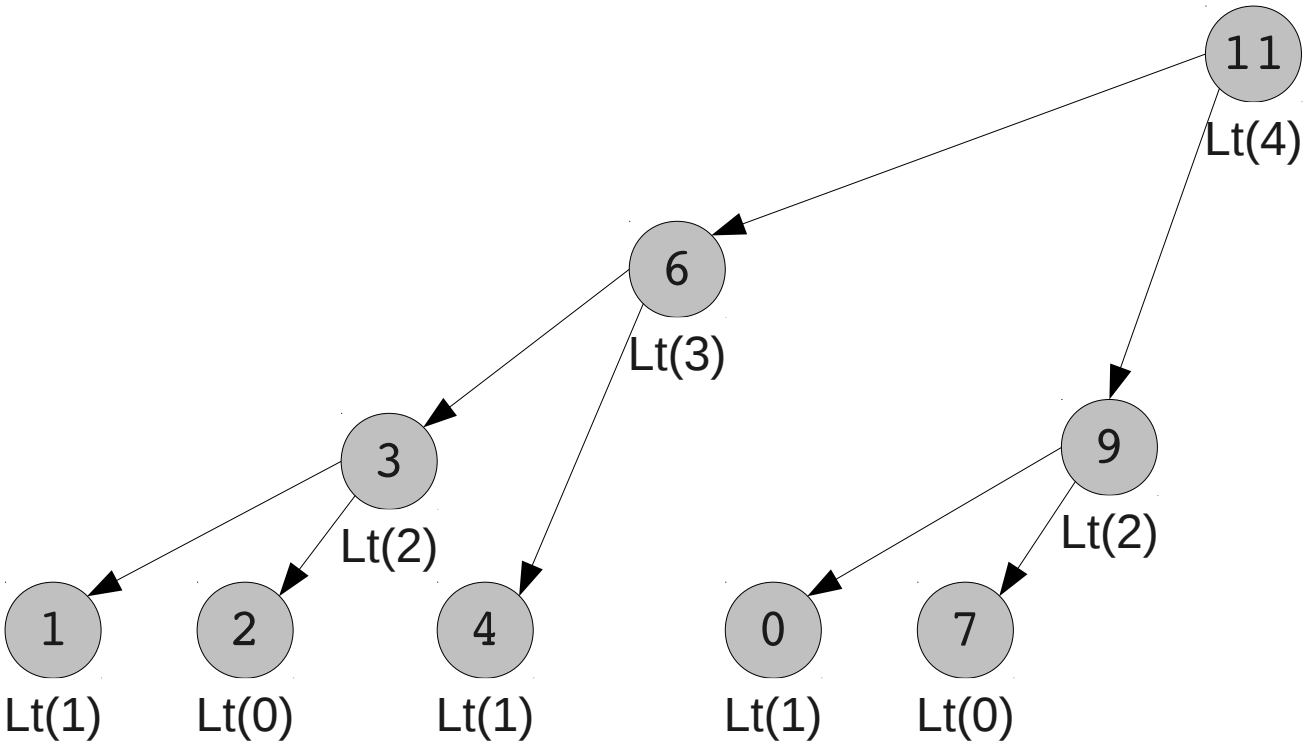
Building a Leonardo Heap



Building a Leonardo Heap

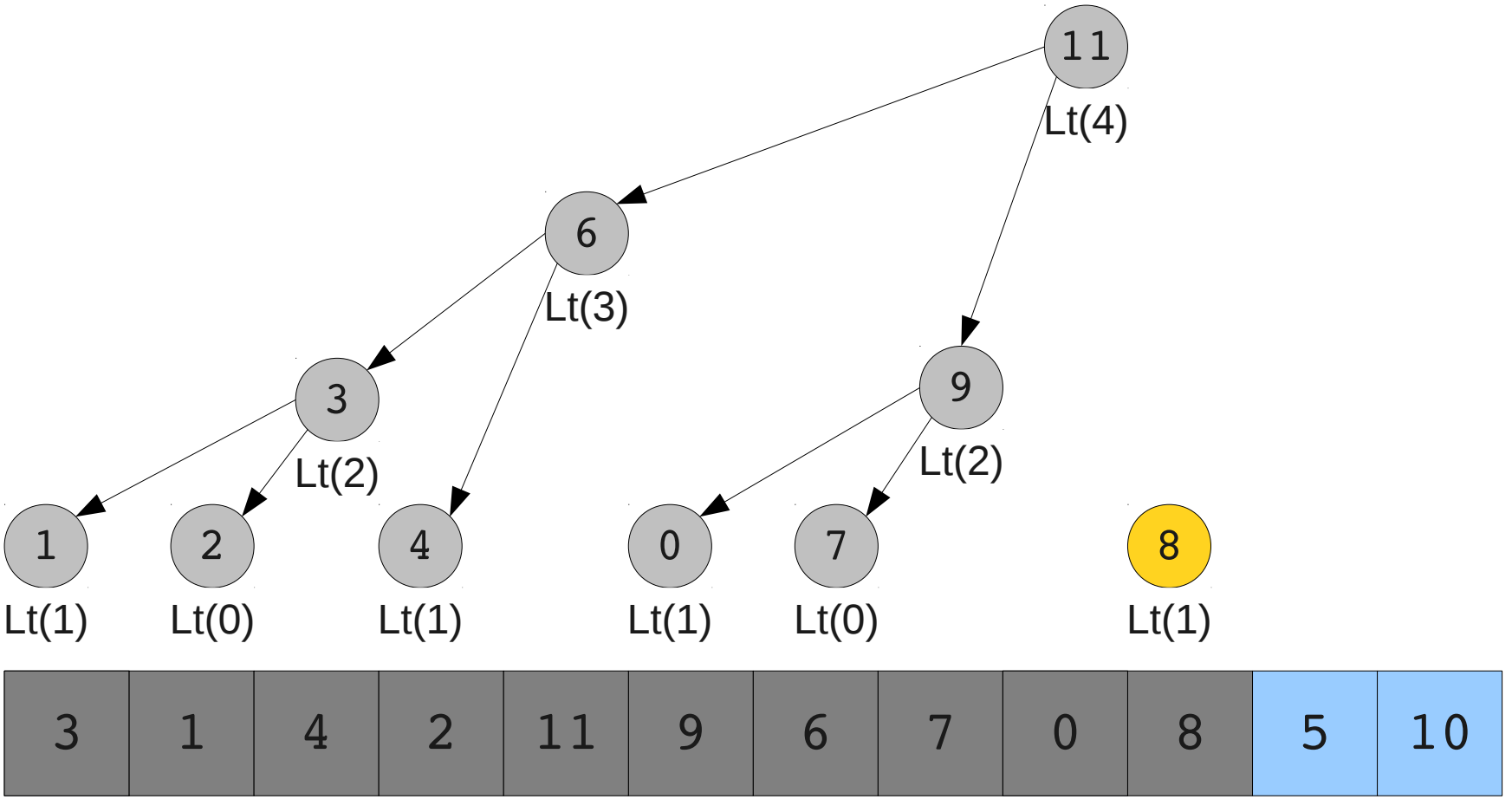


Building a Leonardo Heap

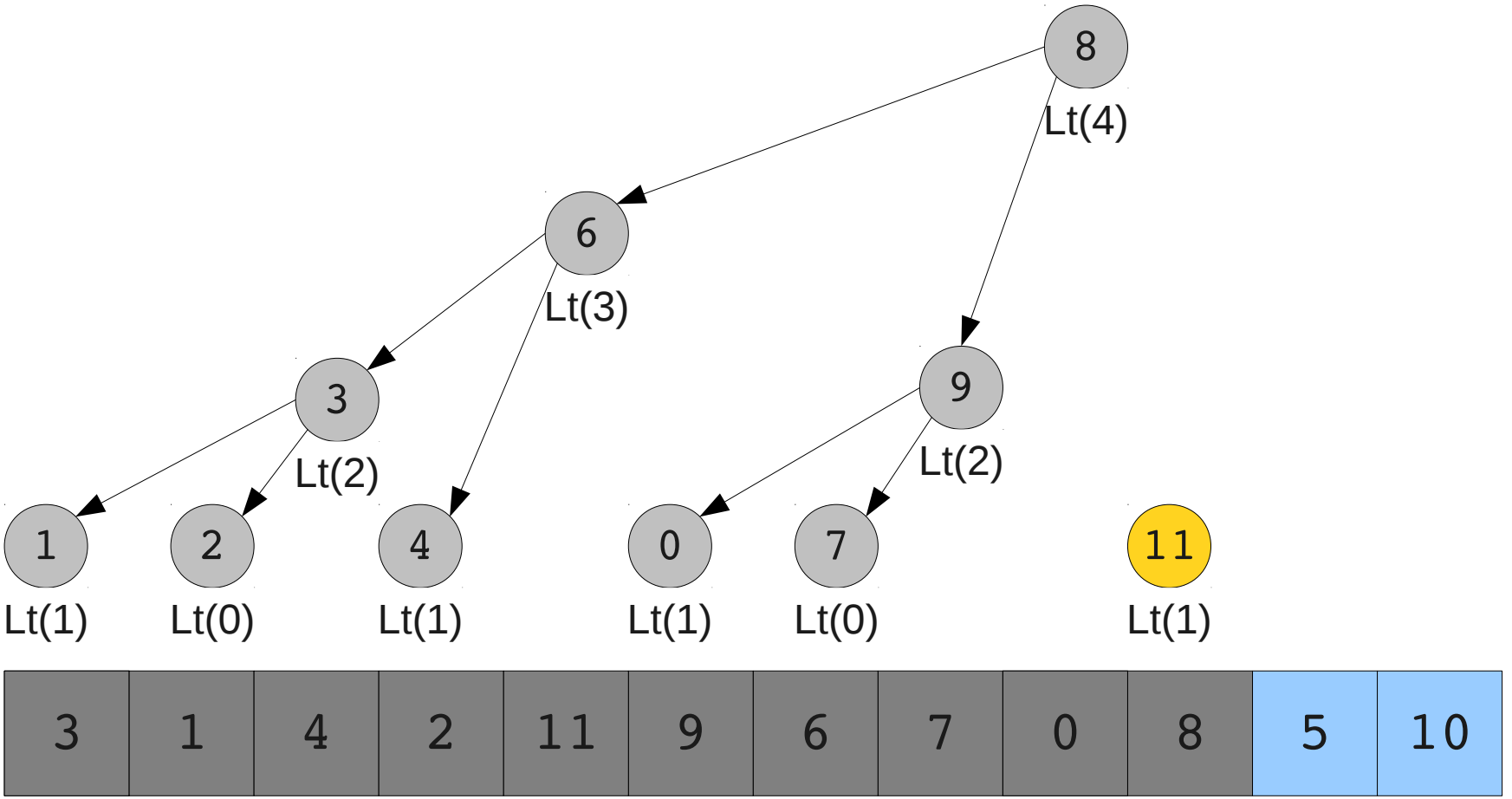


3	1	4	2	11	9	6	7	0	8	5	10

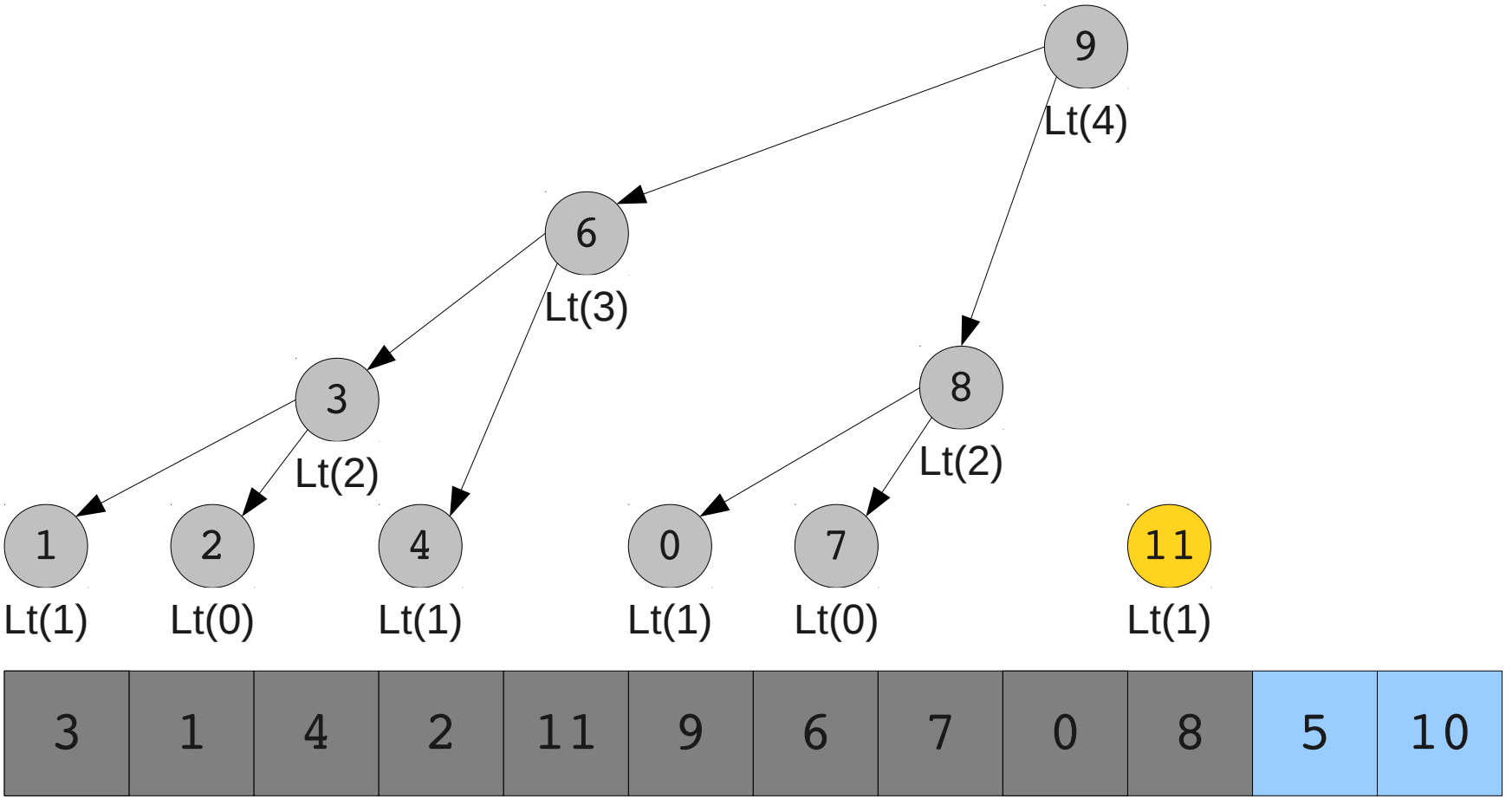
Building a Leonardo Heap



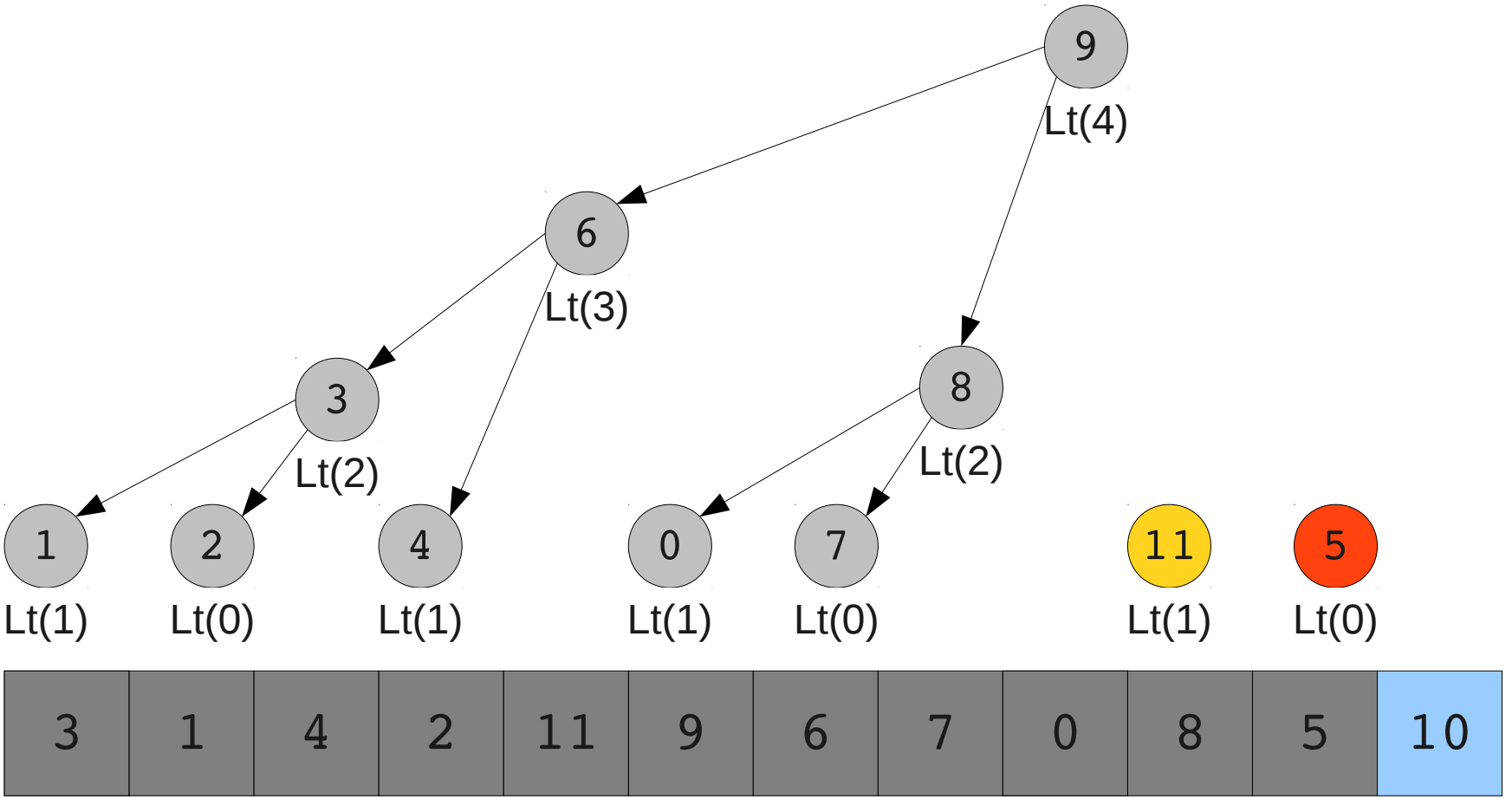
Building a Leonardo Heap



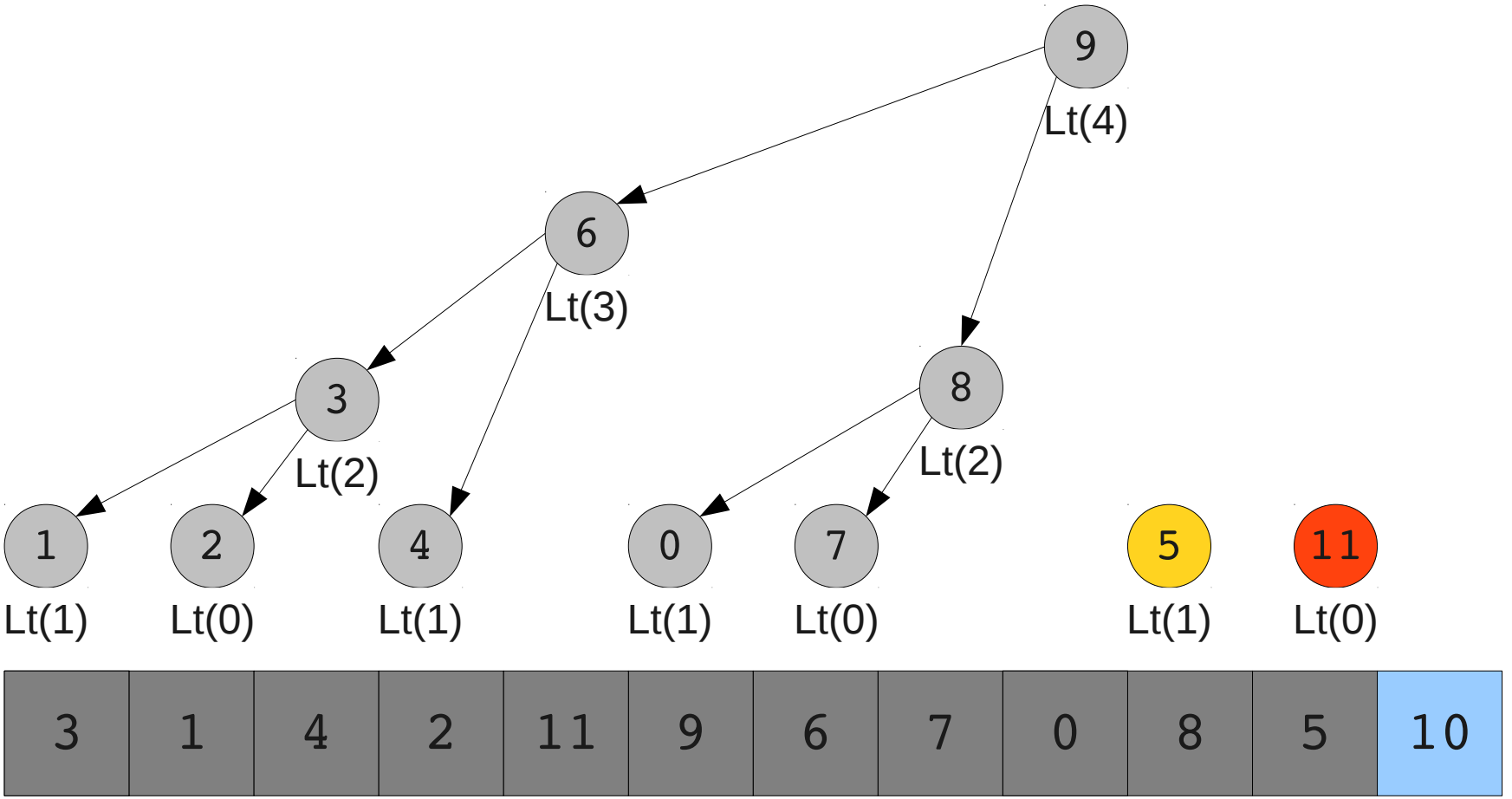
Building a Leonardo Heap



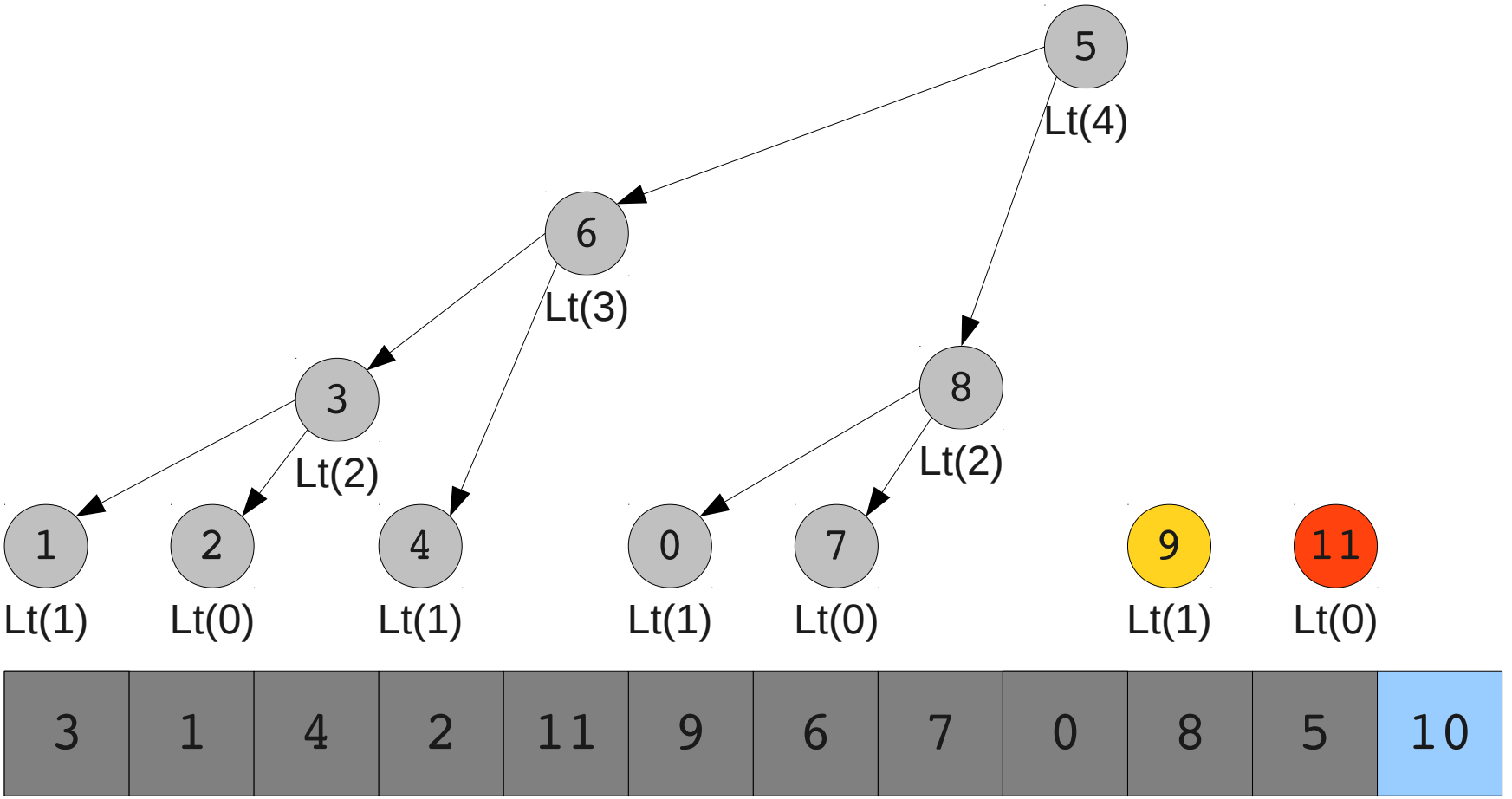
Building a Leonardo Heap



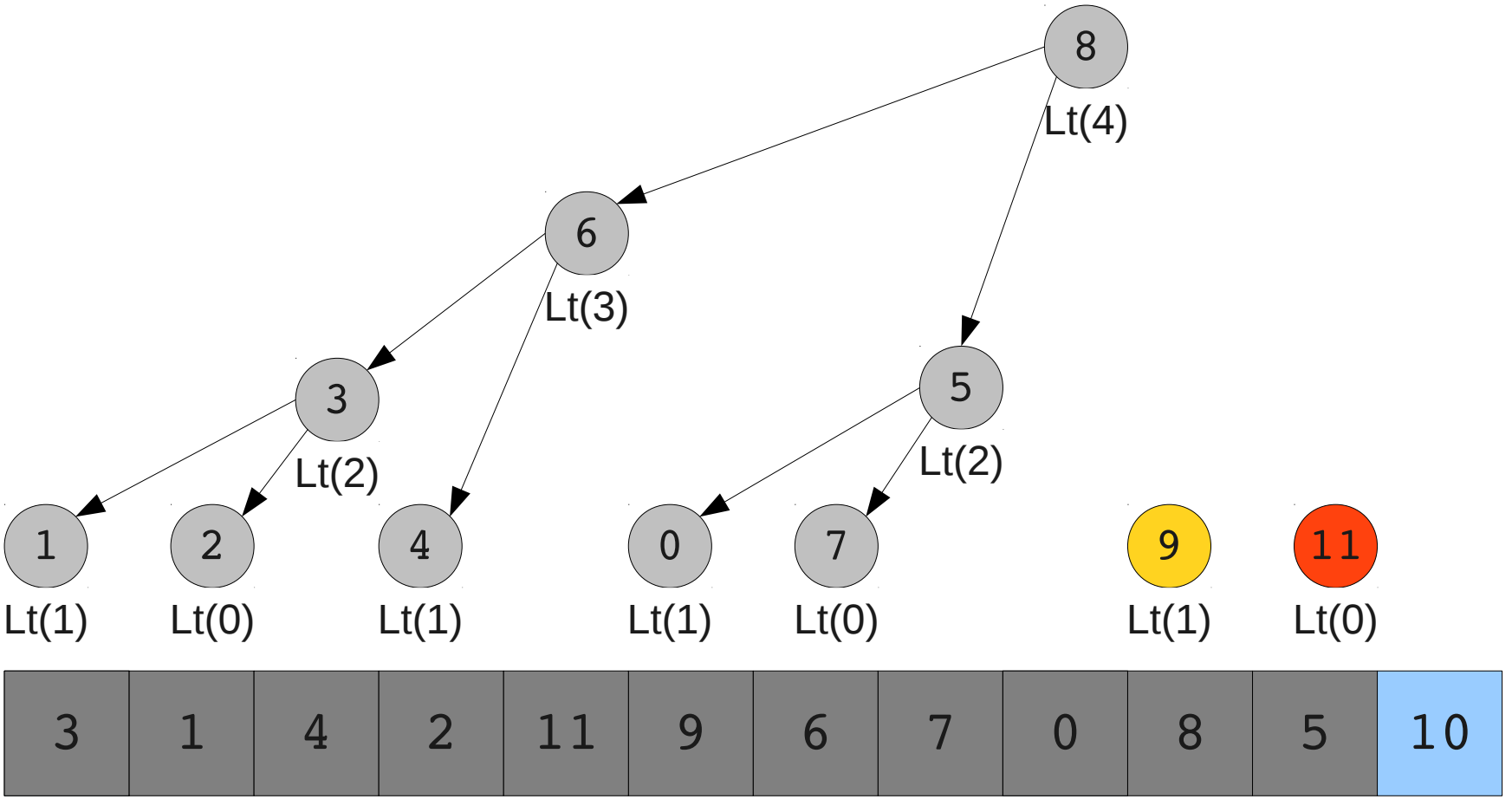
Building a Leonardo Heap



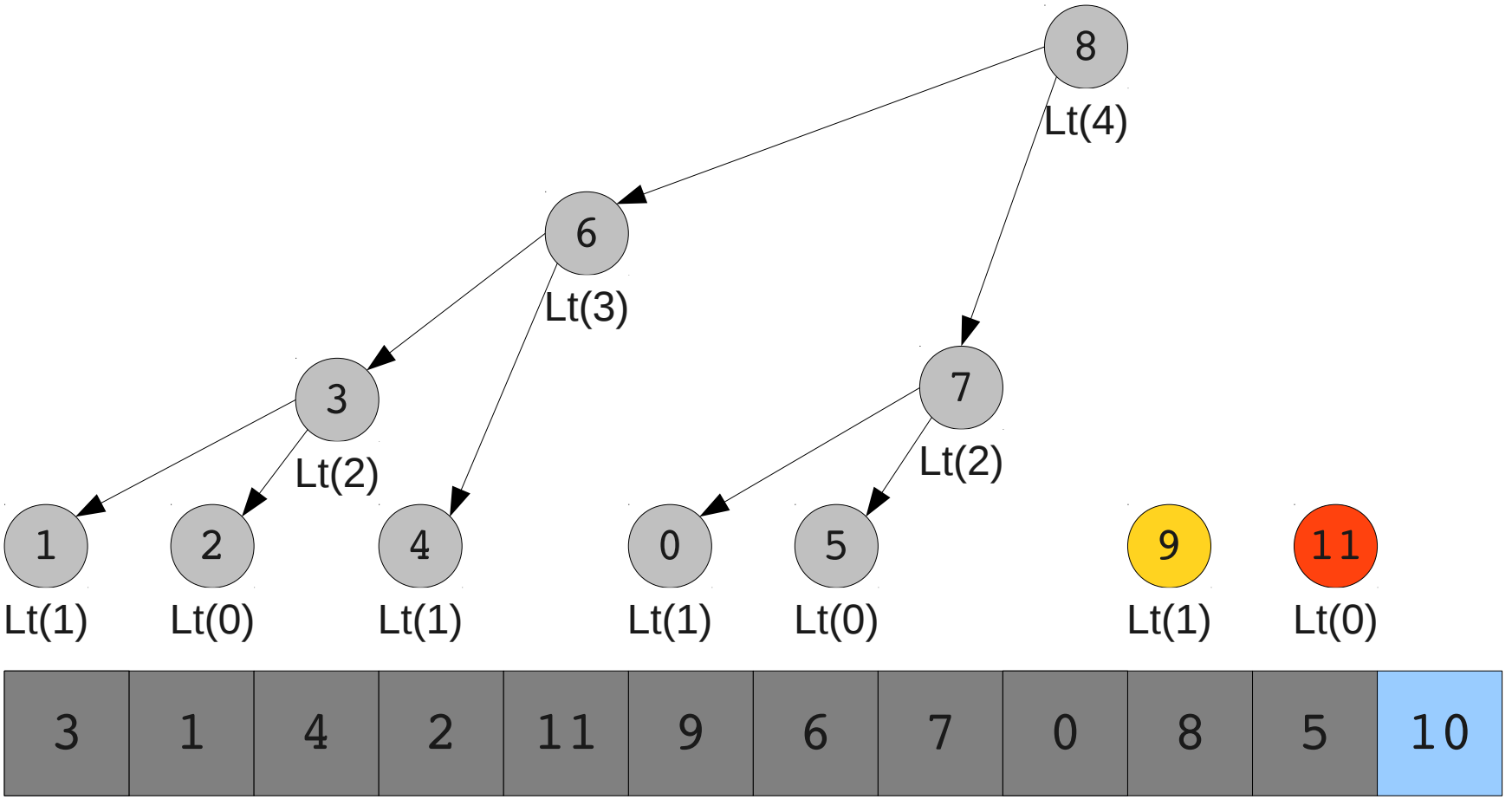
Building a Leonardo Heap



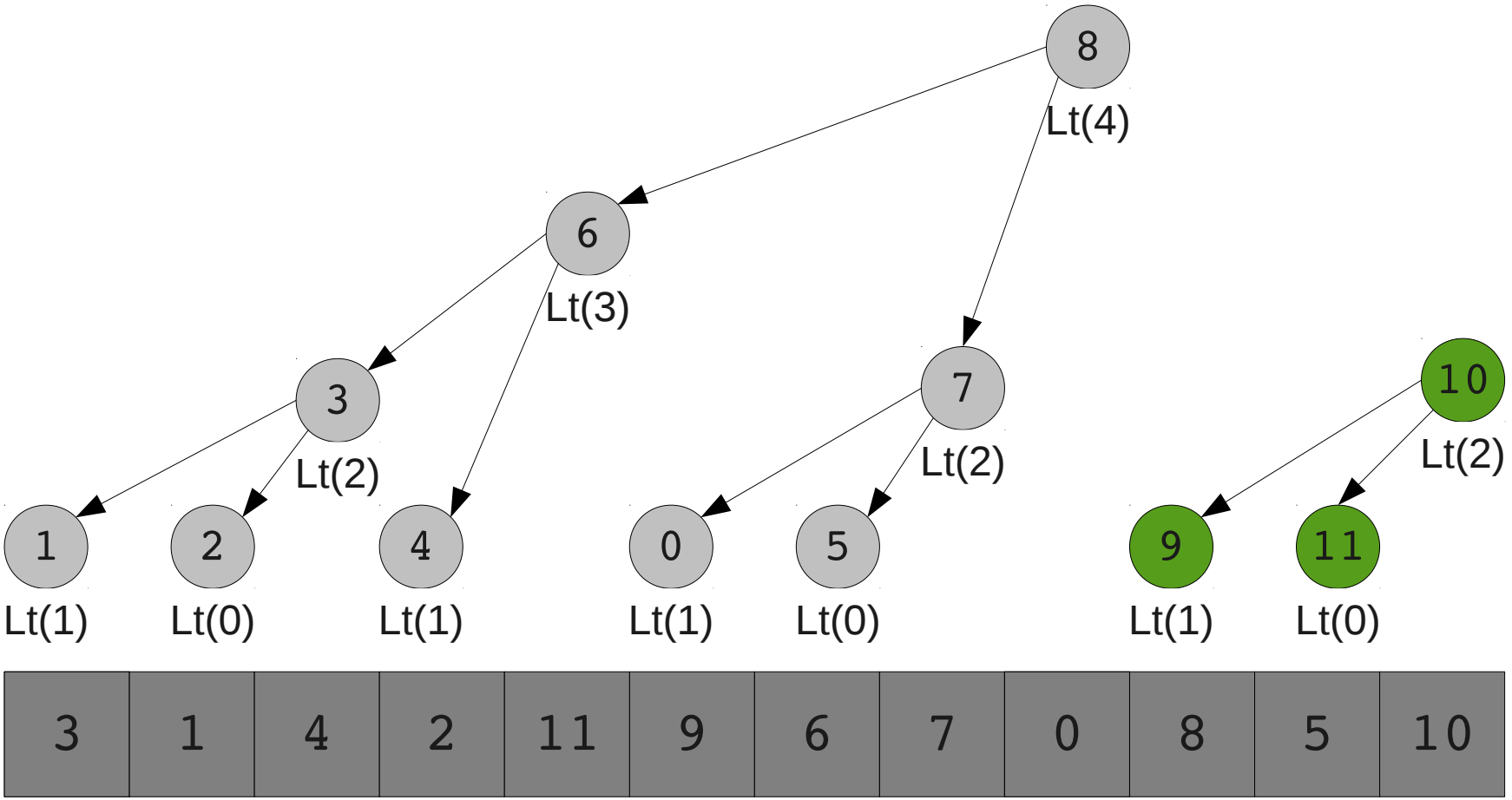
Building a Leonardo Heap



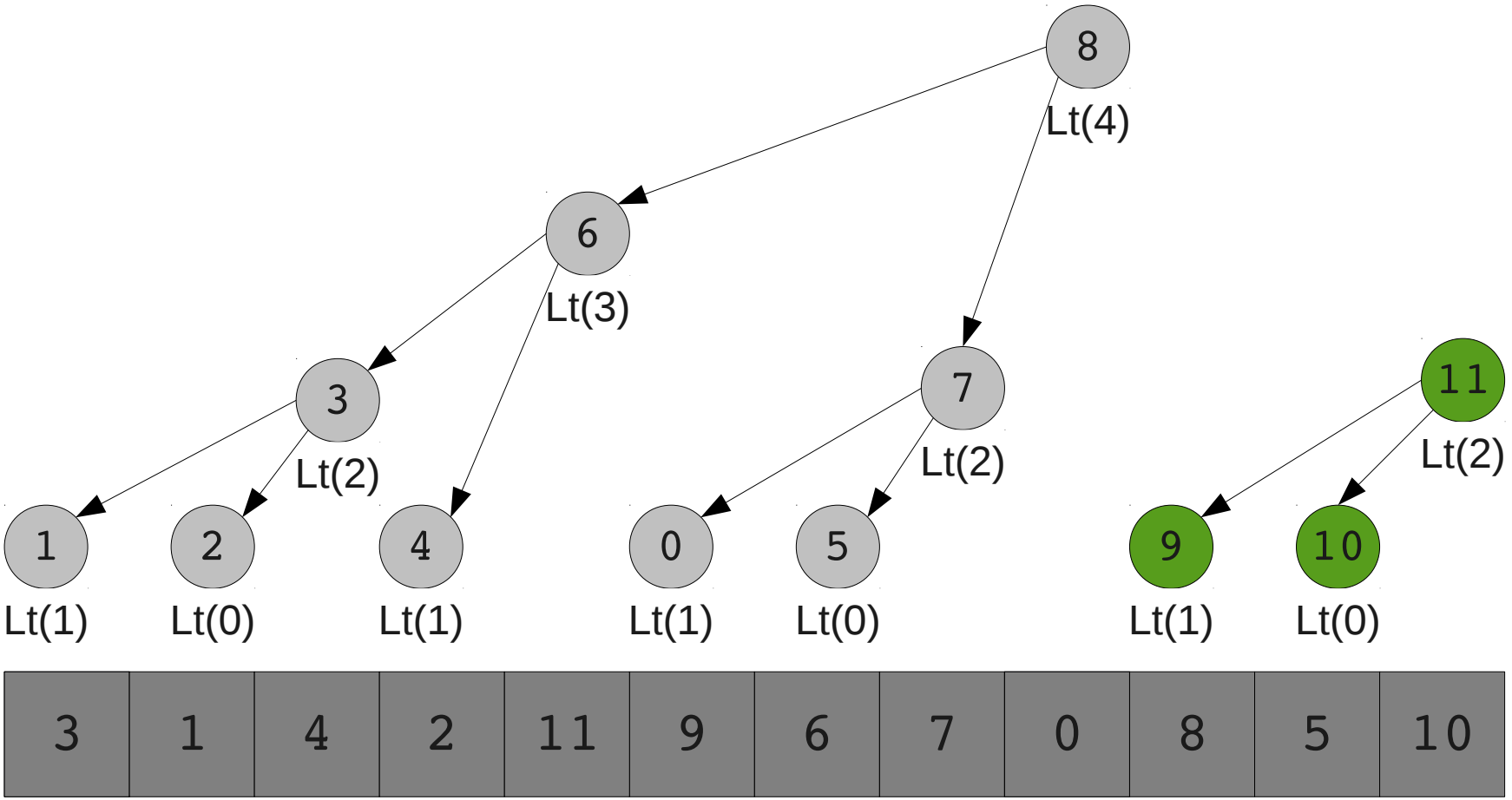
Building a Leonardo Heap



Building a Leonardo Heap



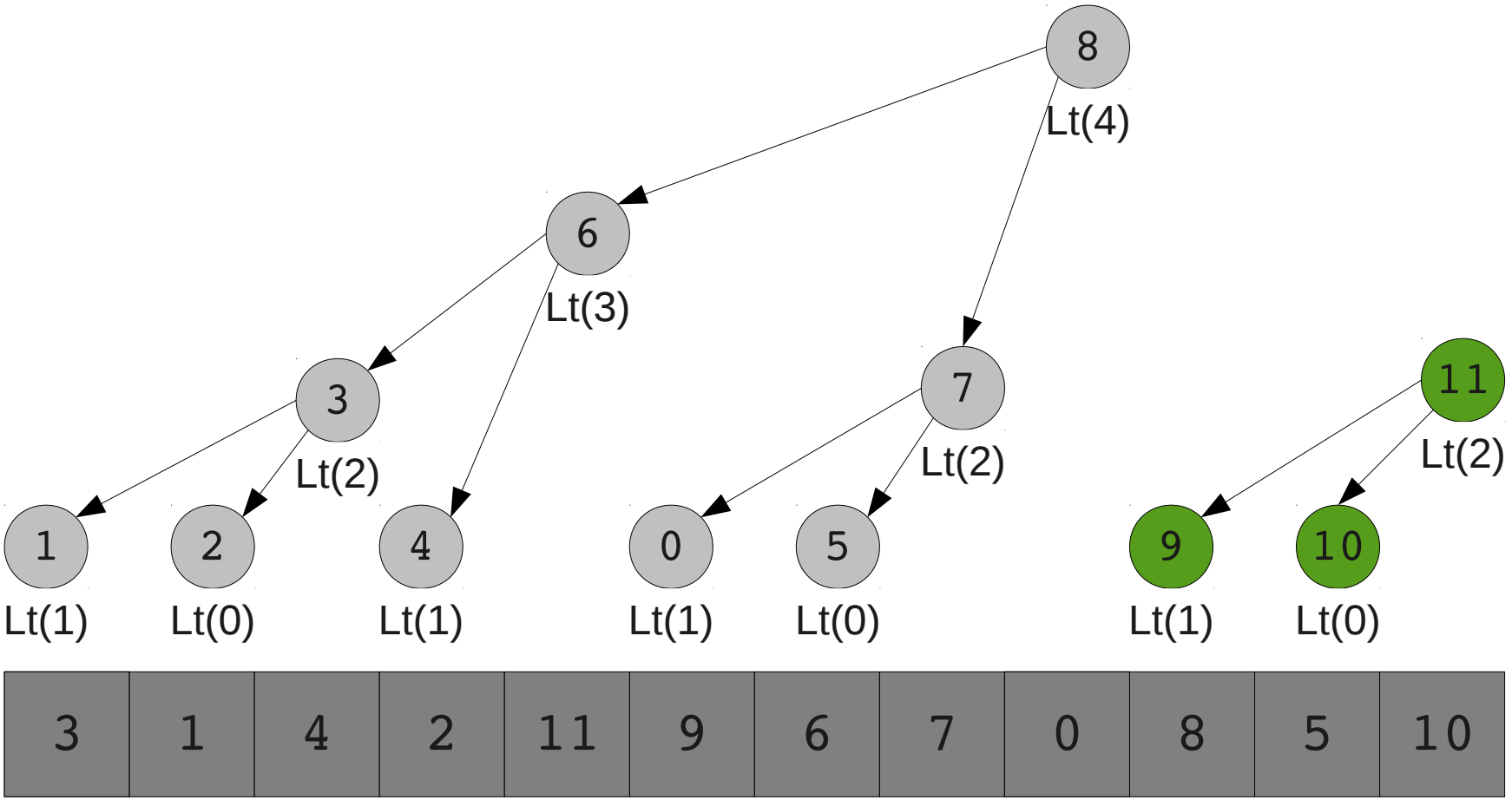
Building a Leonardo Heap



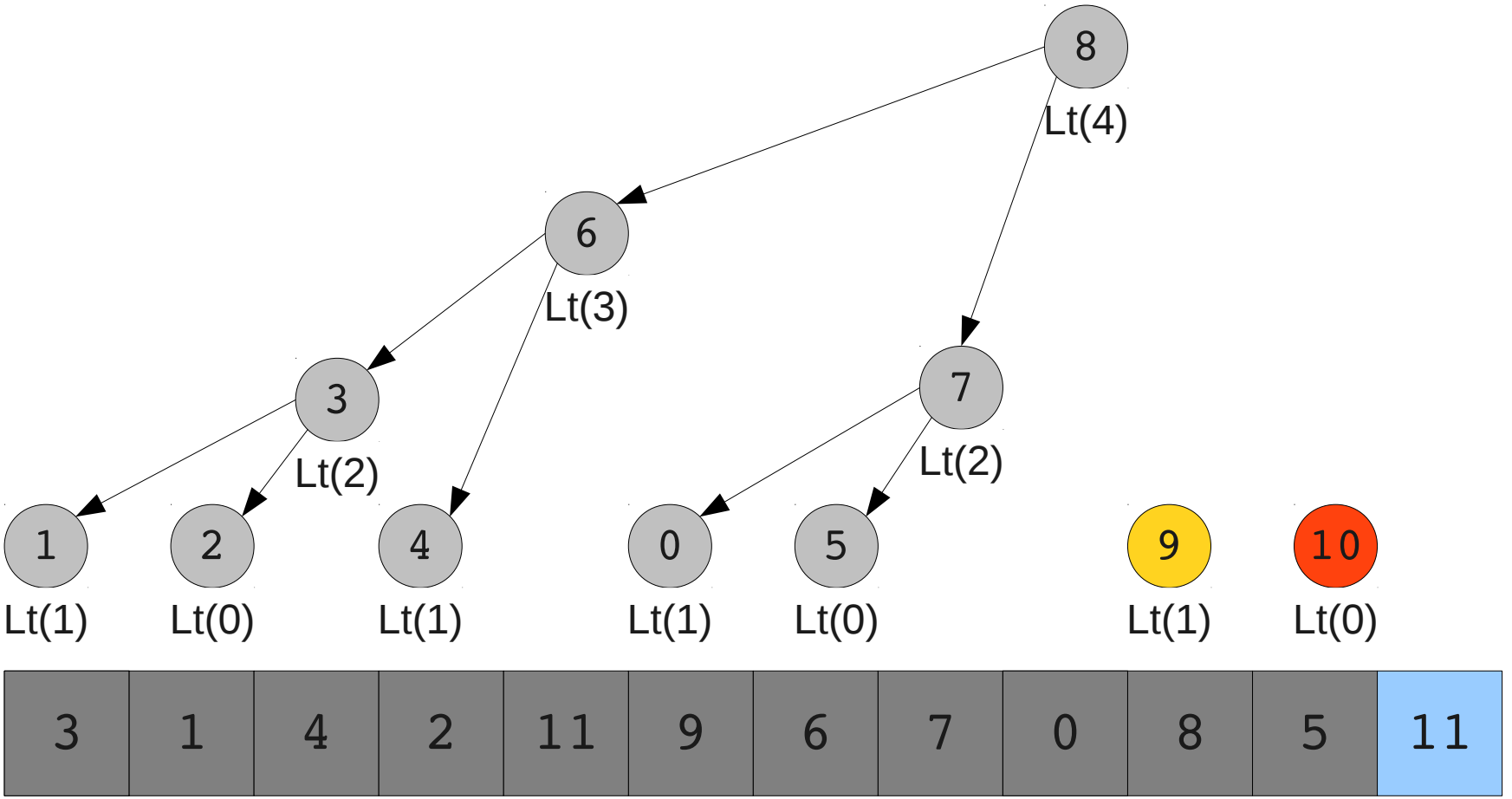
Summary of Insert

- **Add** new element as in unsorted case.
- **Move** new element atop the correct heap.
- **Sift** the element to its final position.

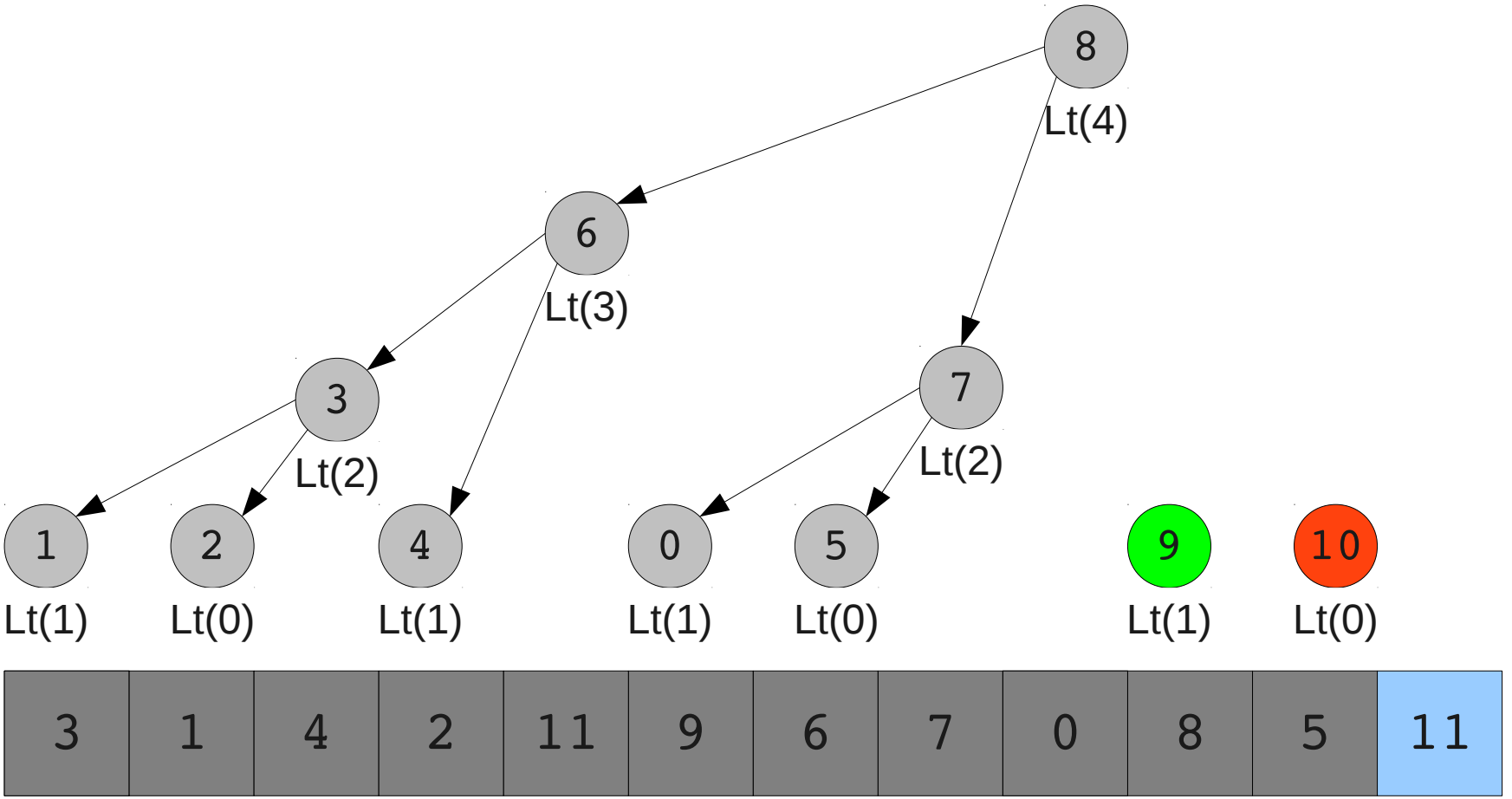
Dequeuing from a Leonardo Heap



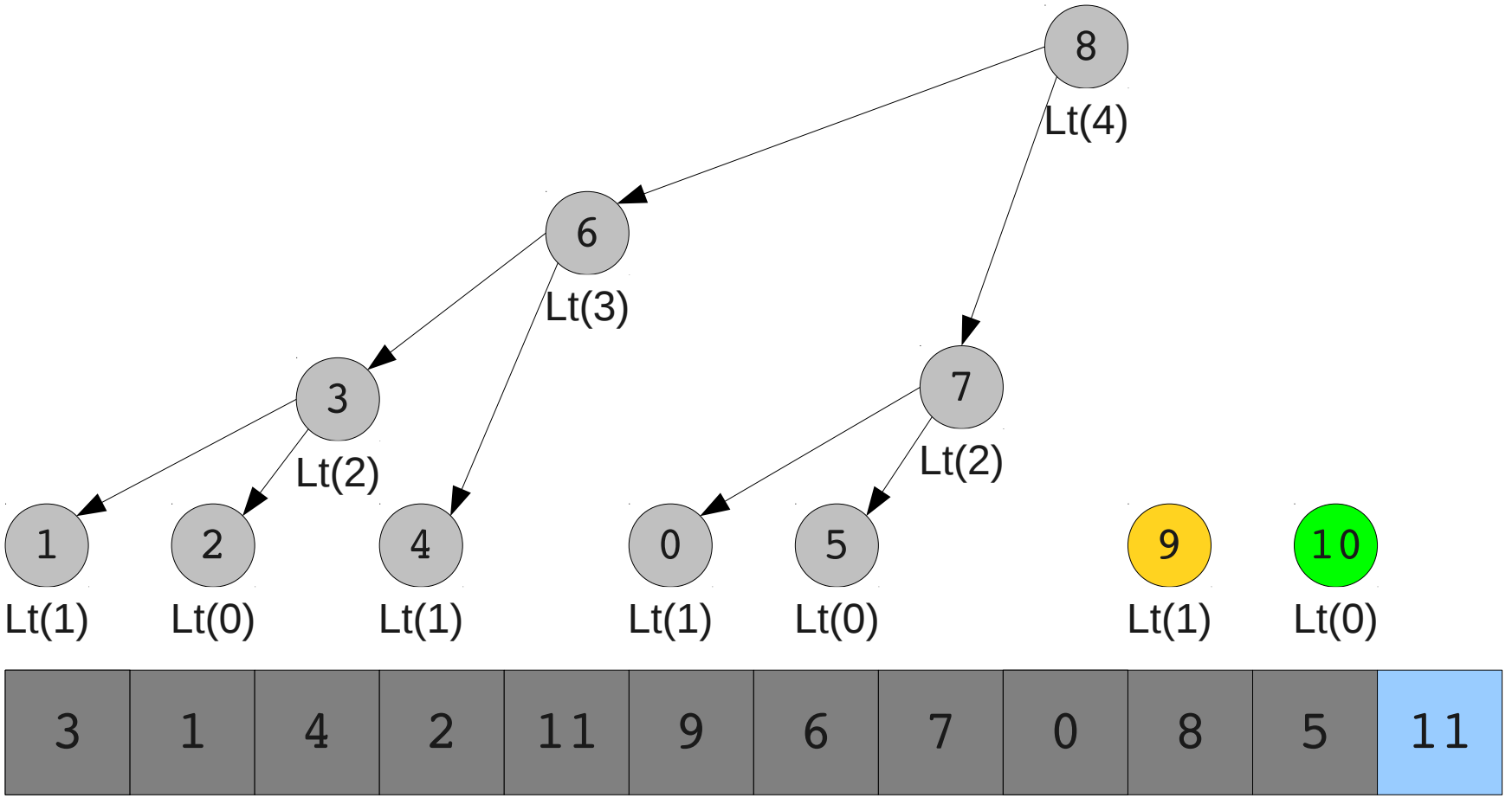
Dequeuing from a Leonardo Heap



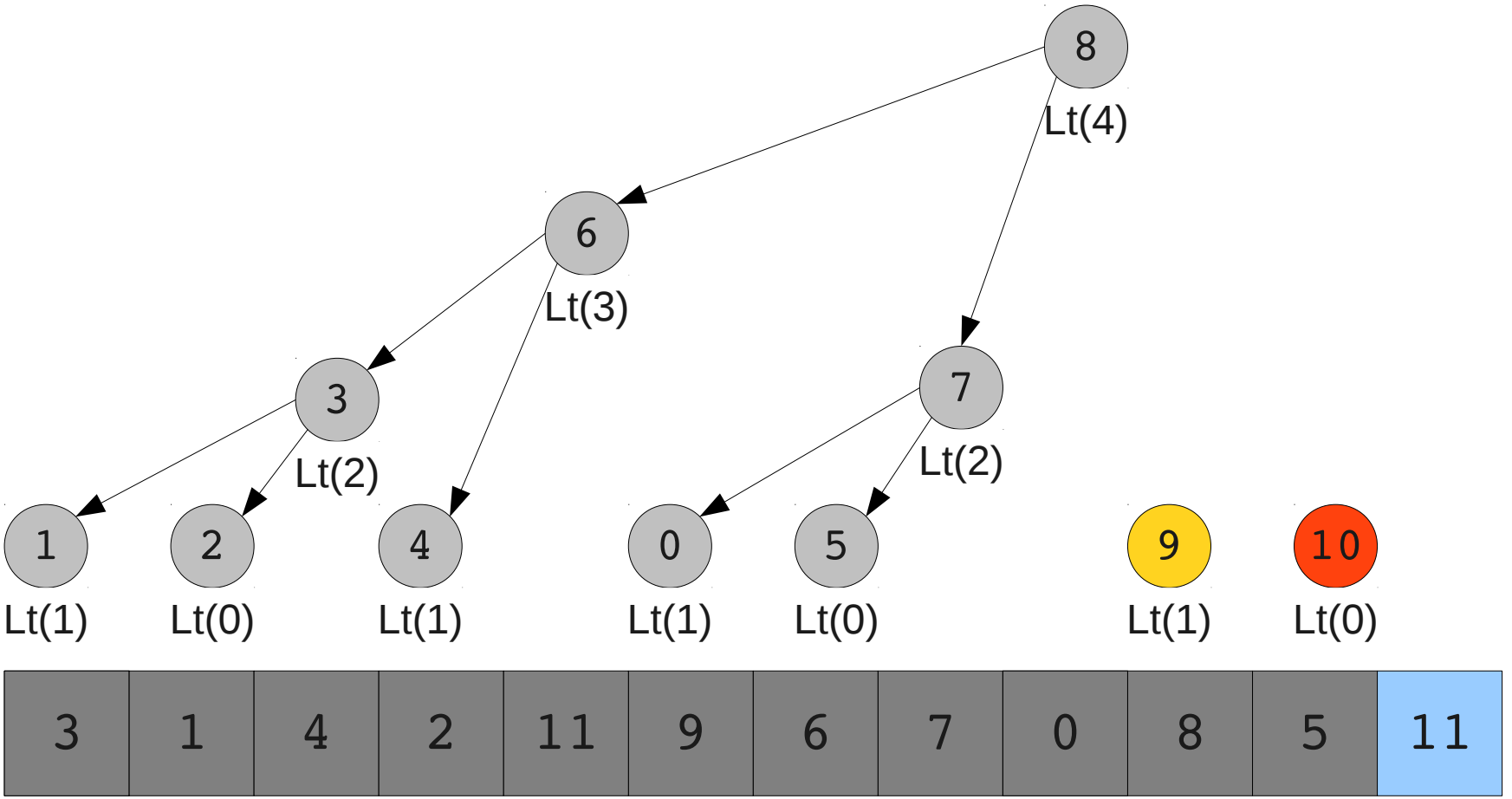
Dequeuing from a Leonardo Heap



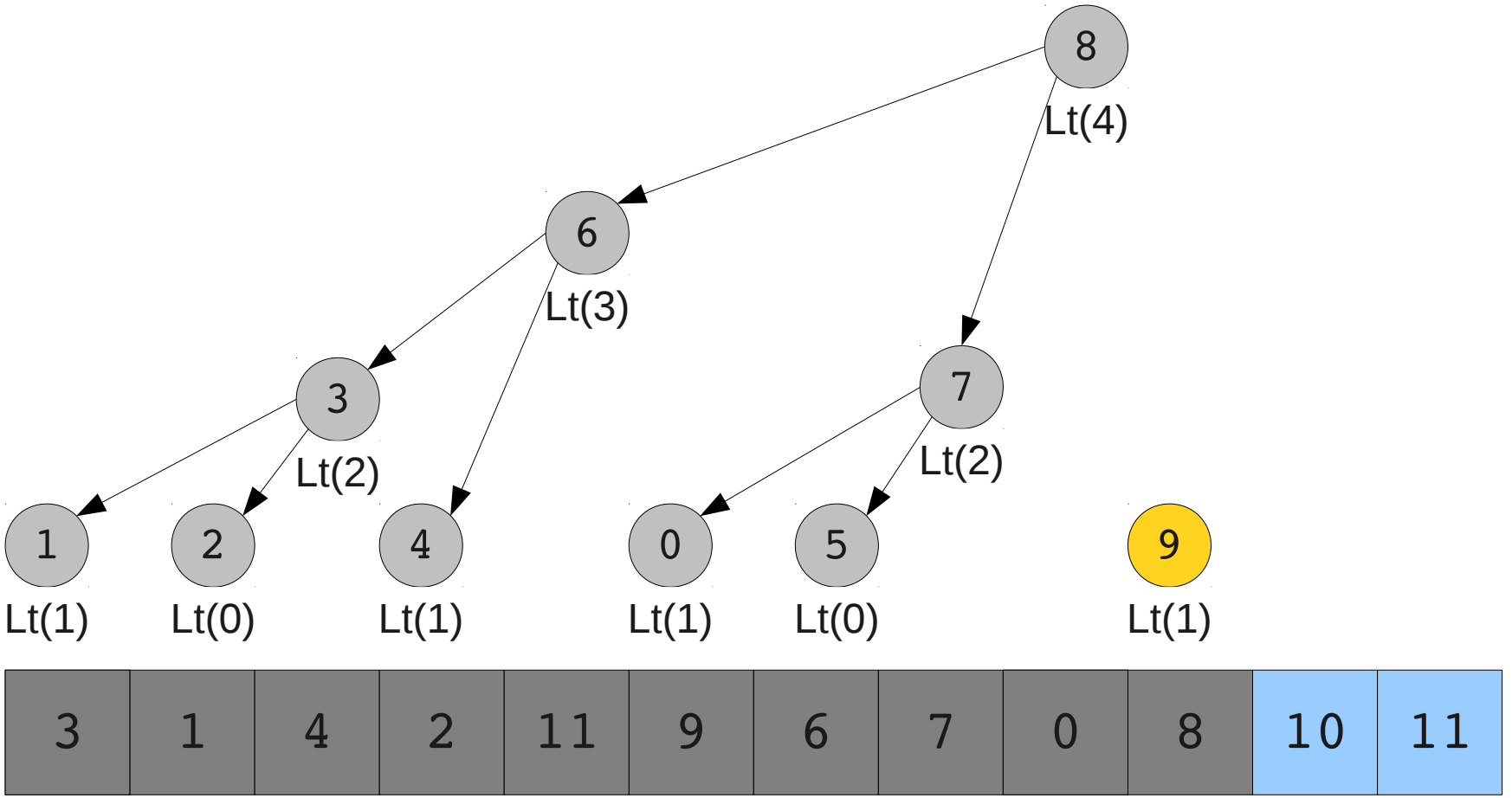
Dequeuing from a Leonardo Heap



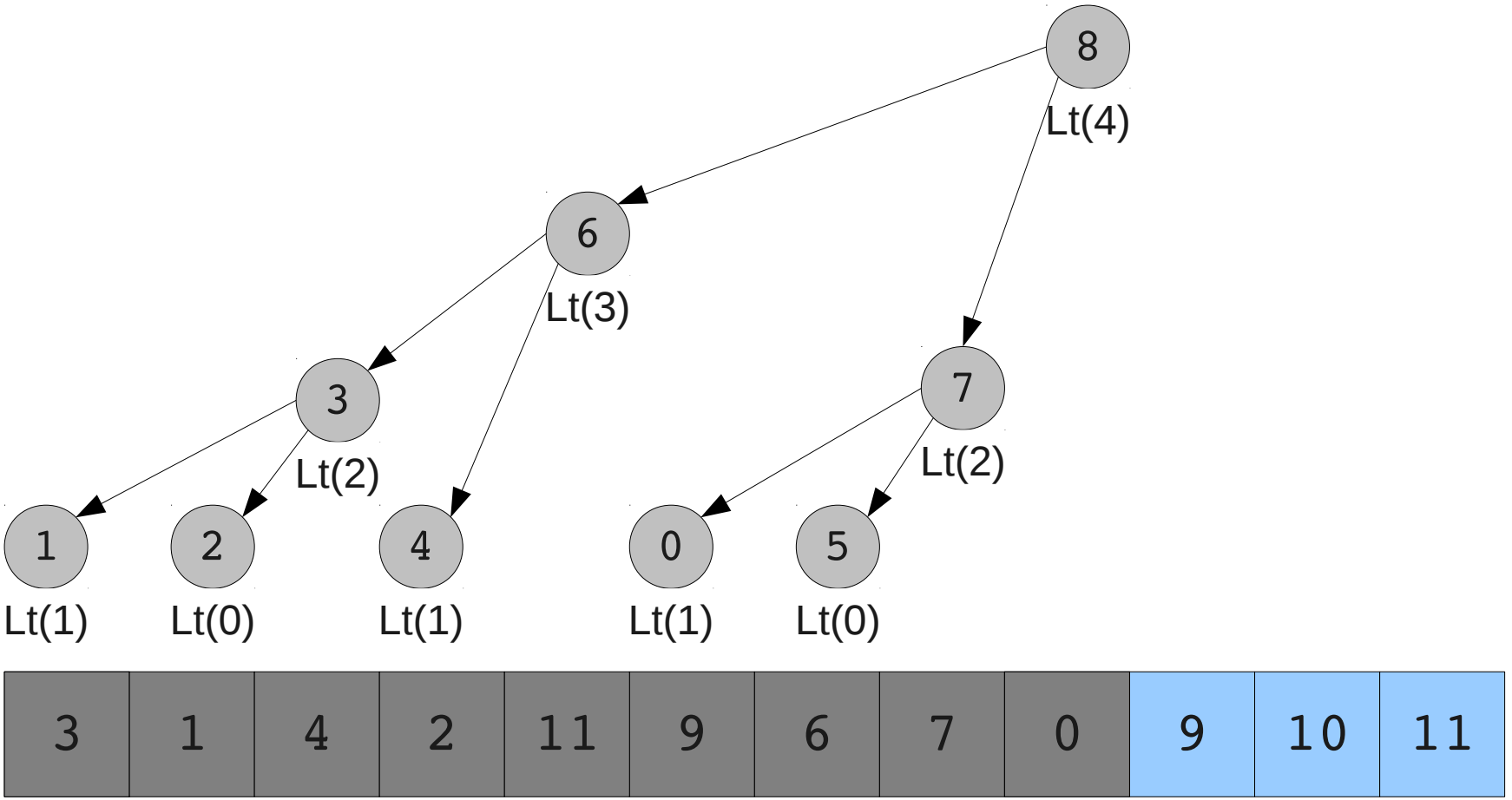
Dequeuing from a Leonardo Heap



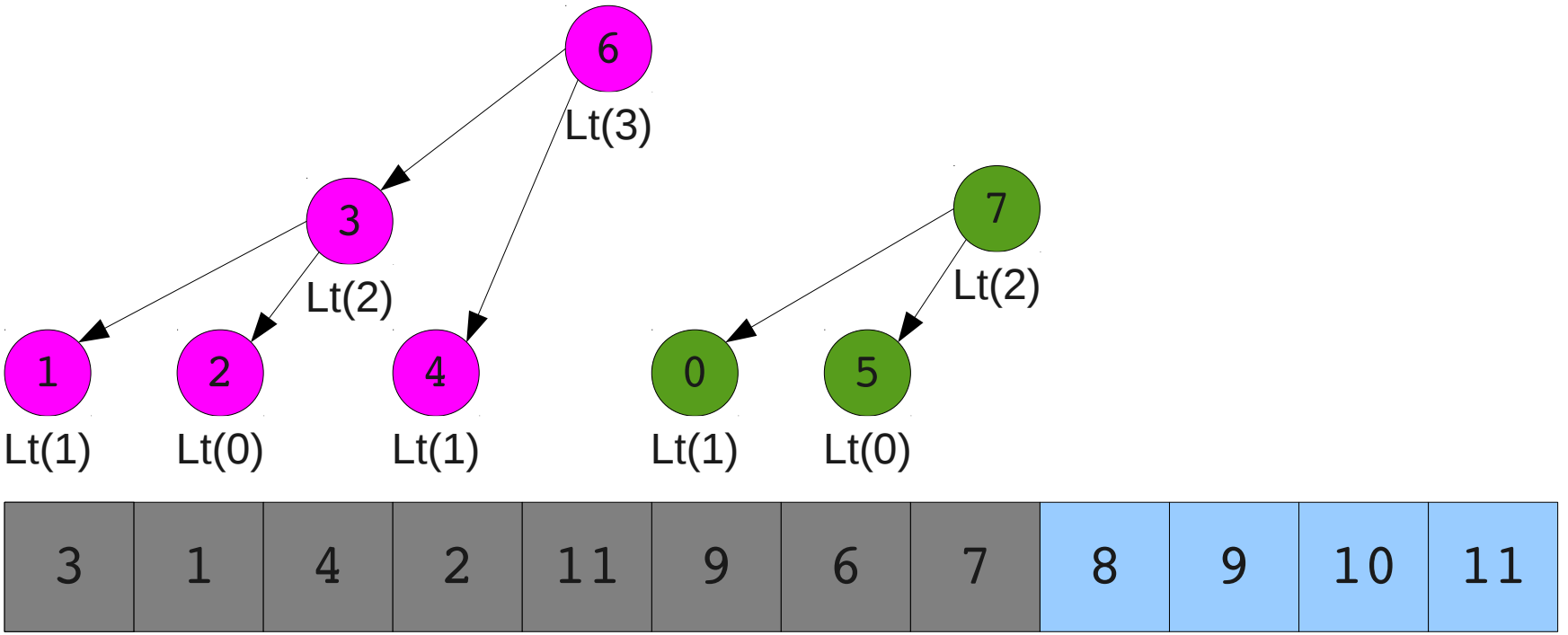
Dequeuing from a Leonardo Heap



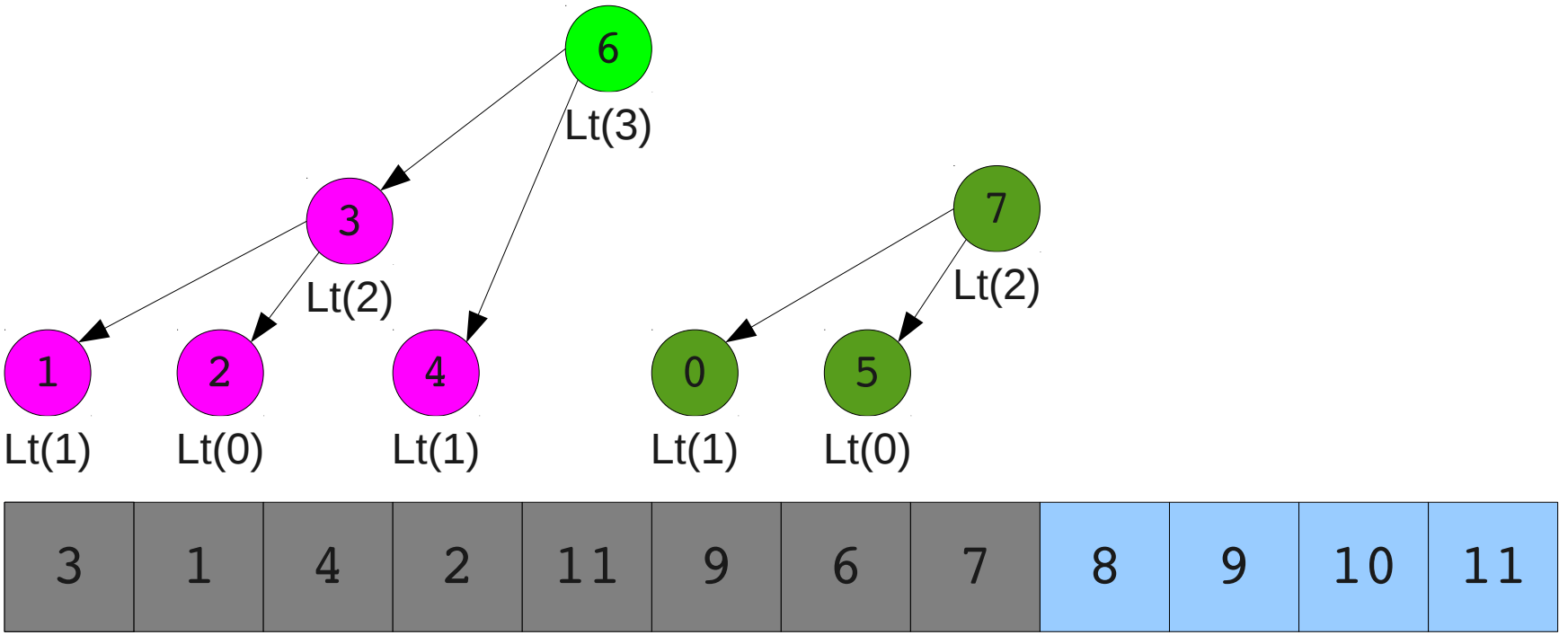
Dequeuing from a Leonardo Heap



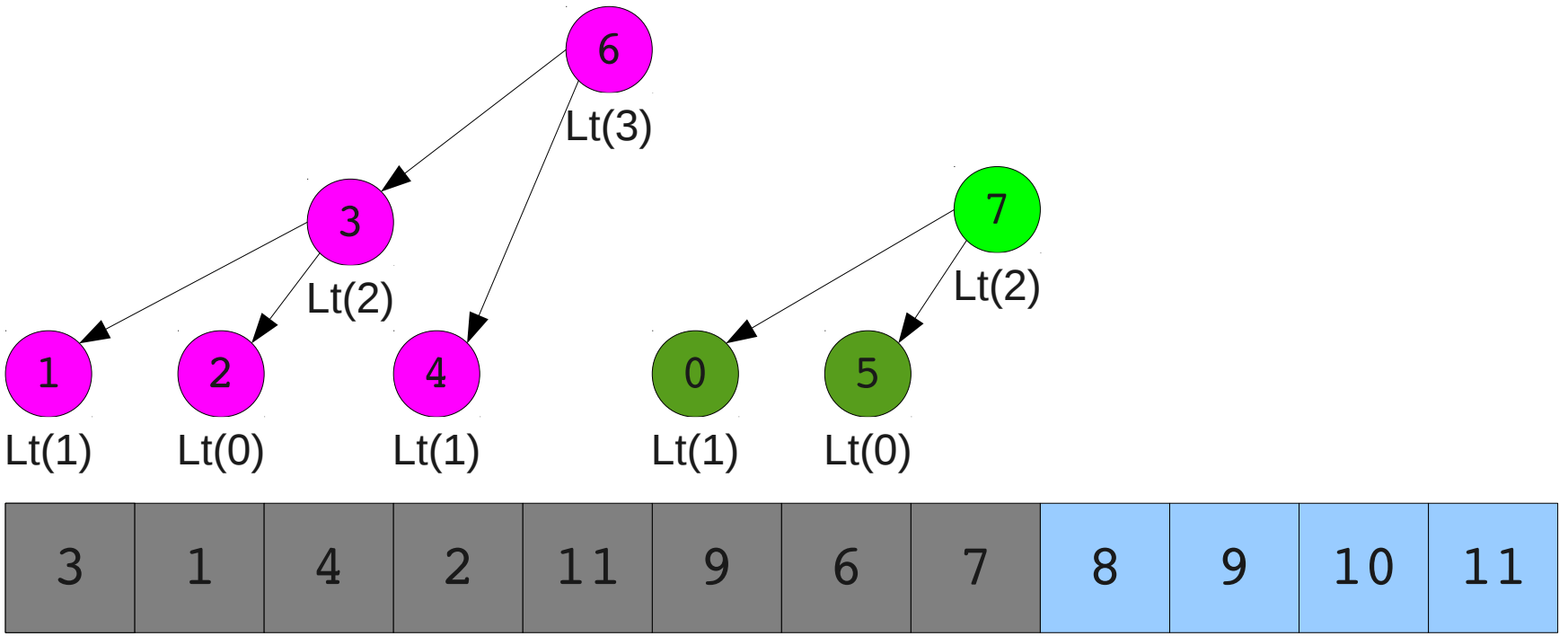
Dequeuing from a Leonardo Heap



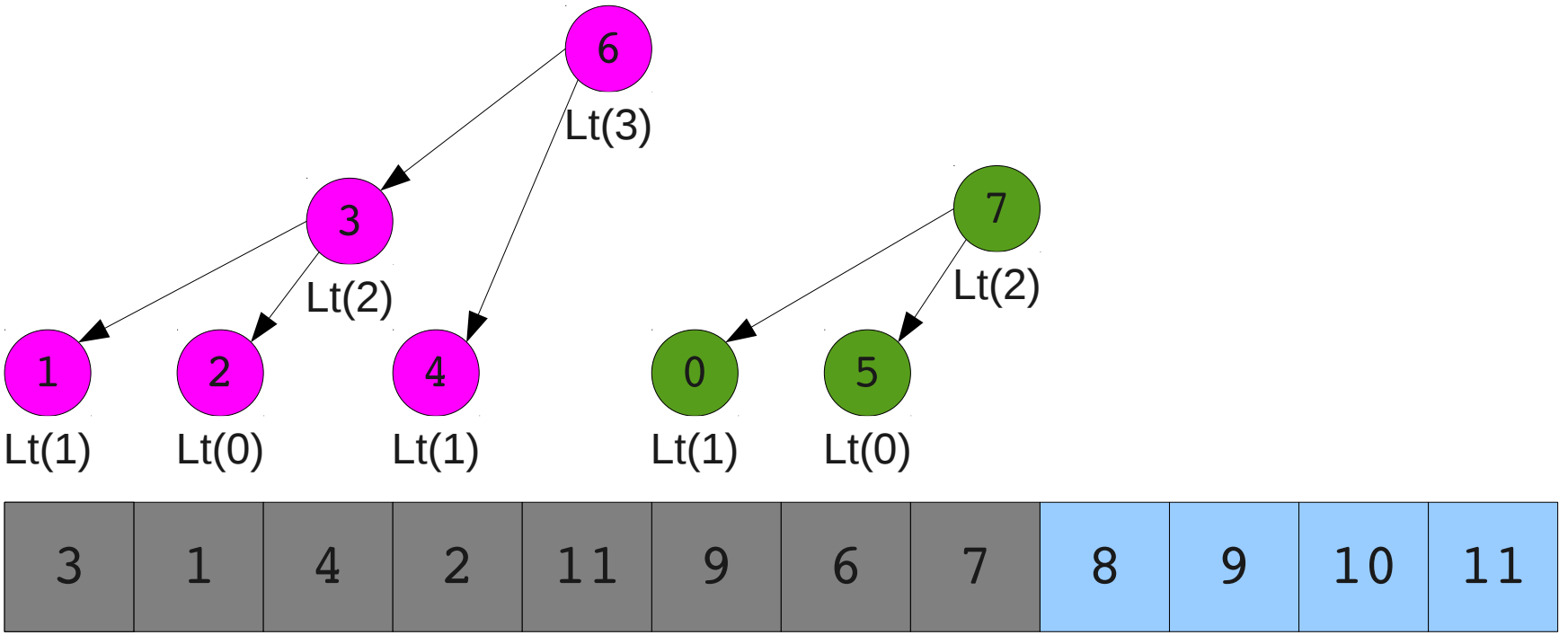
Dequeuing from a Leonardo Heap



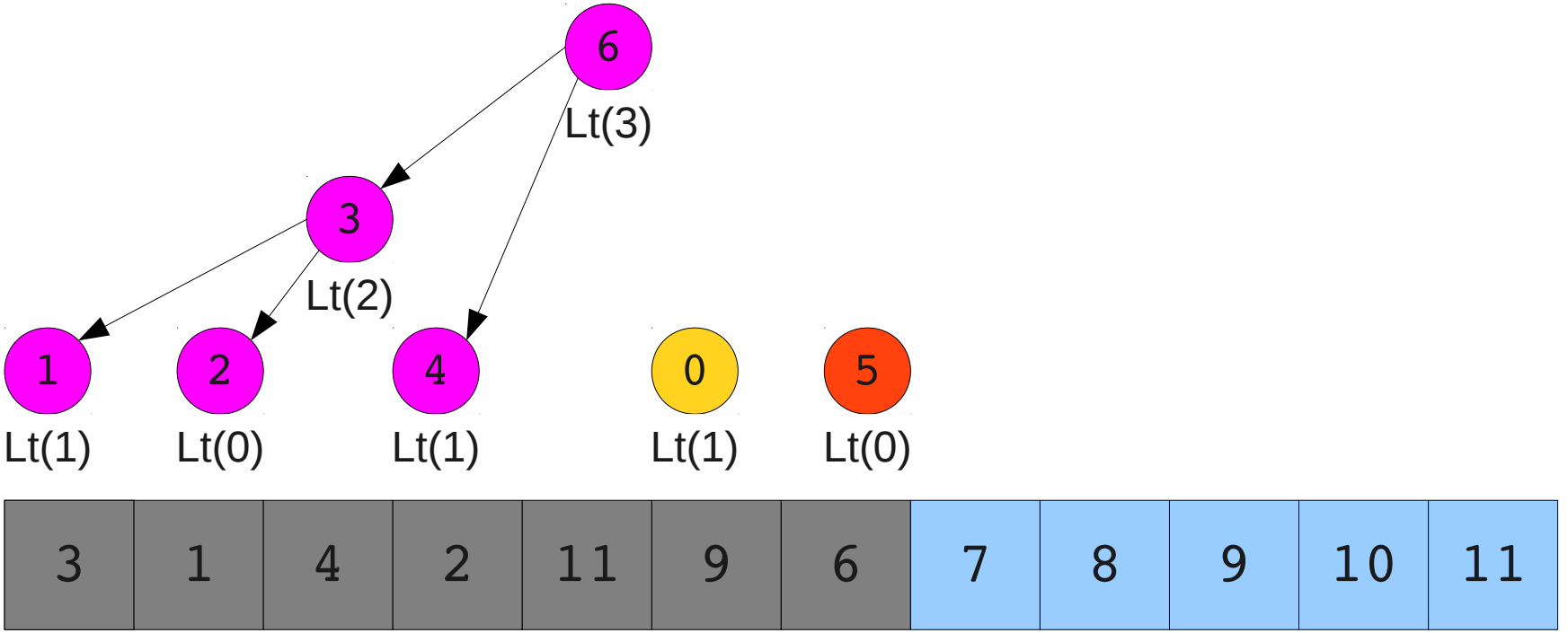
Dequeuing from a Leonardo Heap



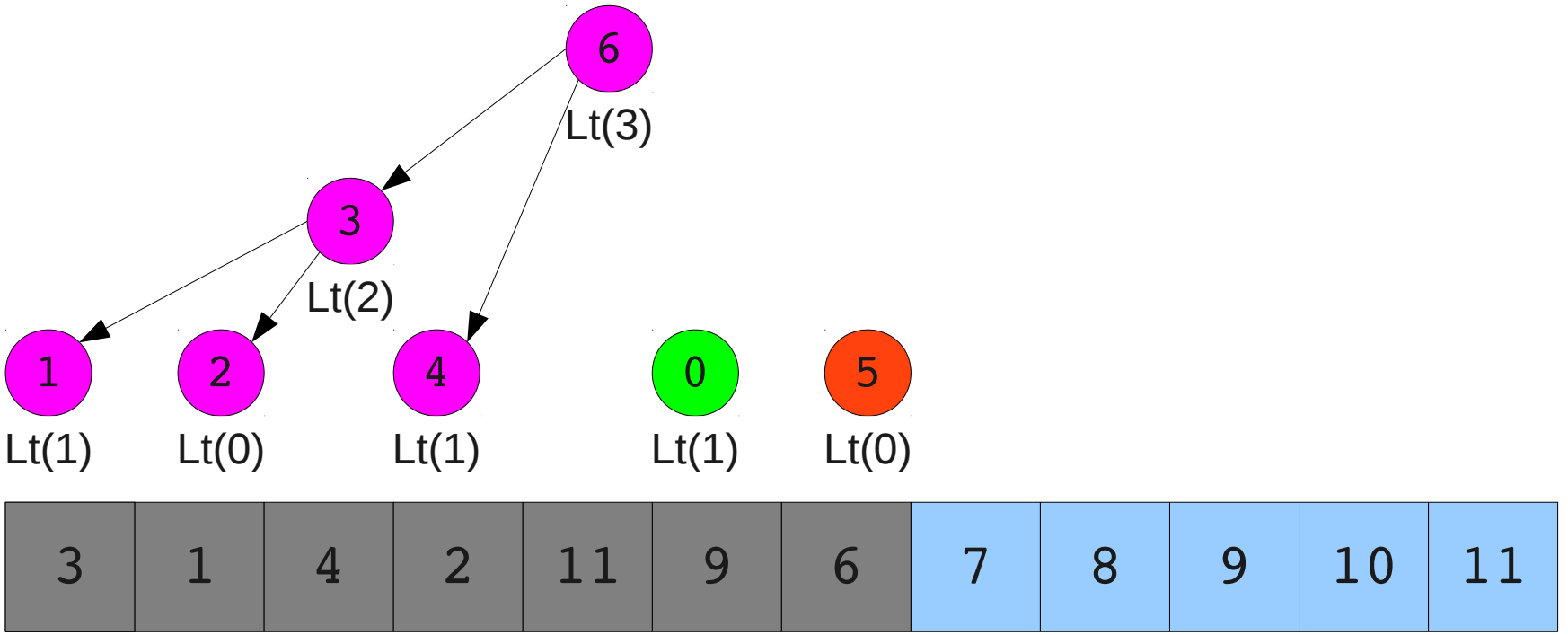
Dequeuing from a Leonardo Heap



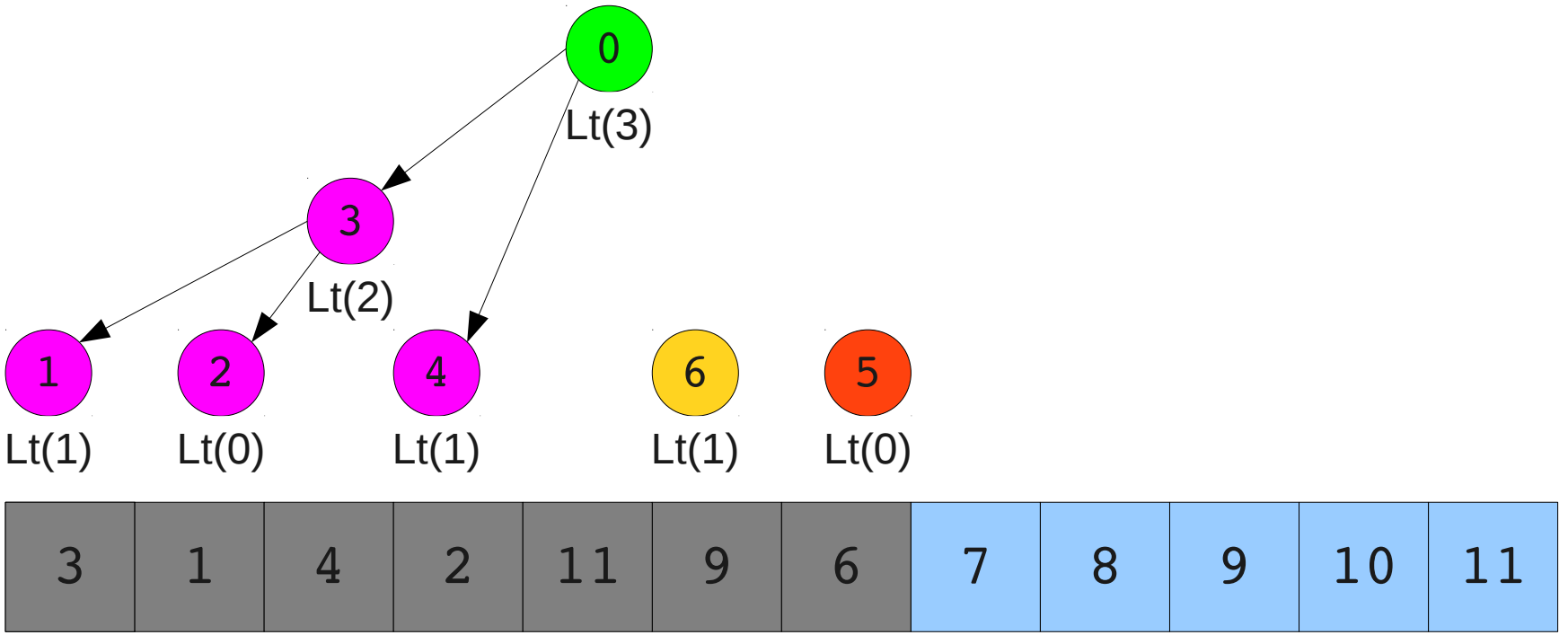
Dequeuing from a Leonardo Heap



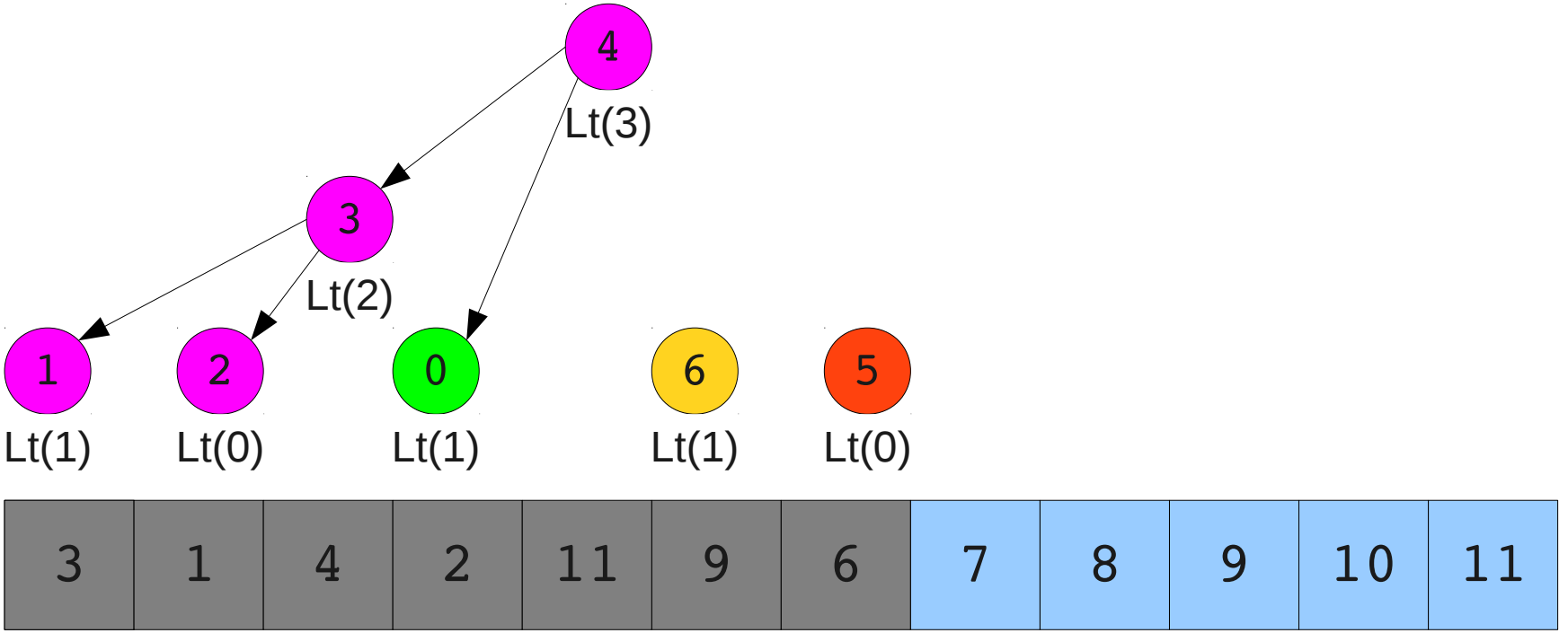
Dequeuing from a Leonardo Heap



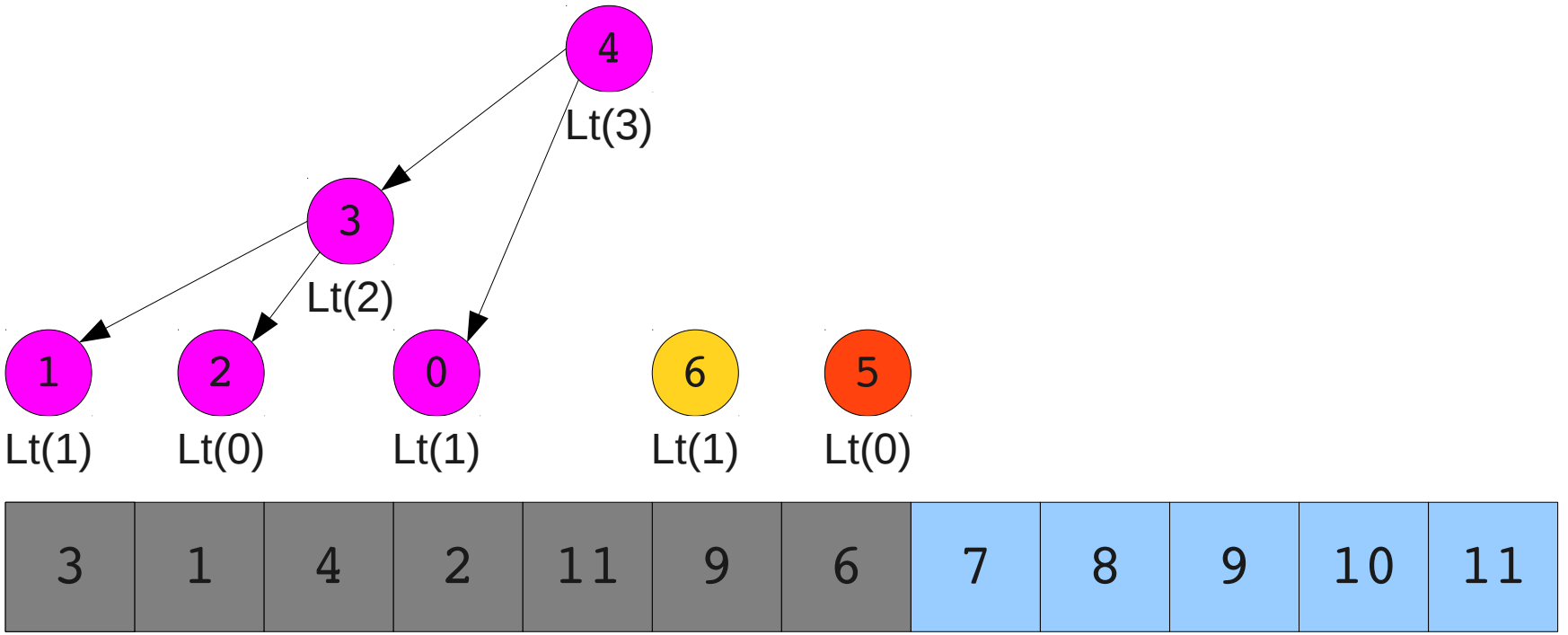
Dequeuing from a Leonardo Heap



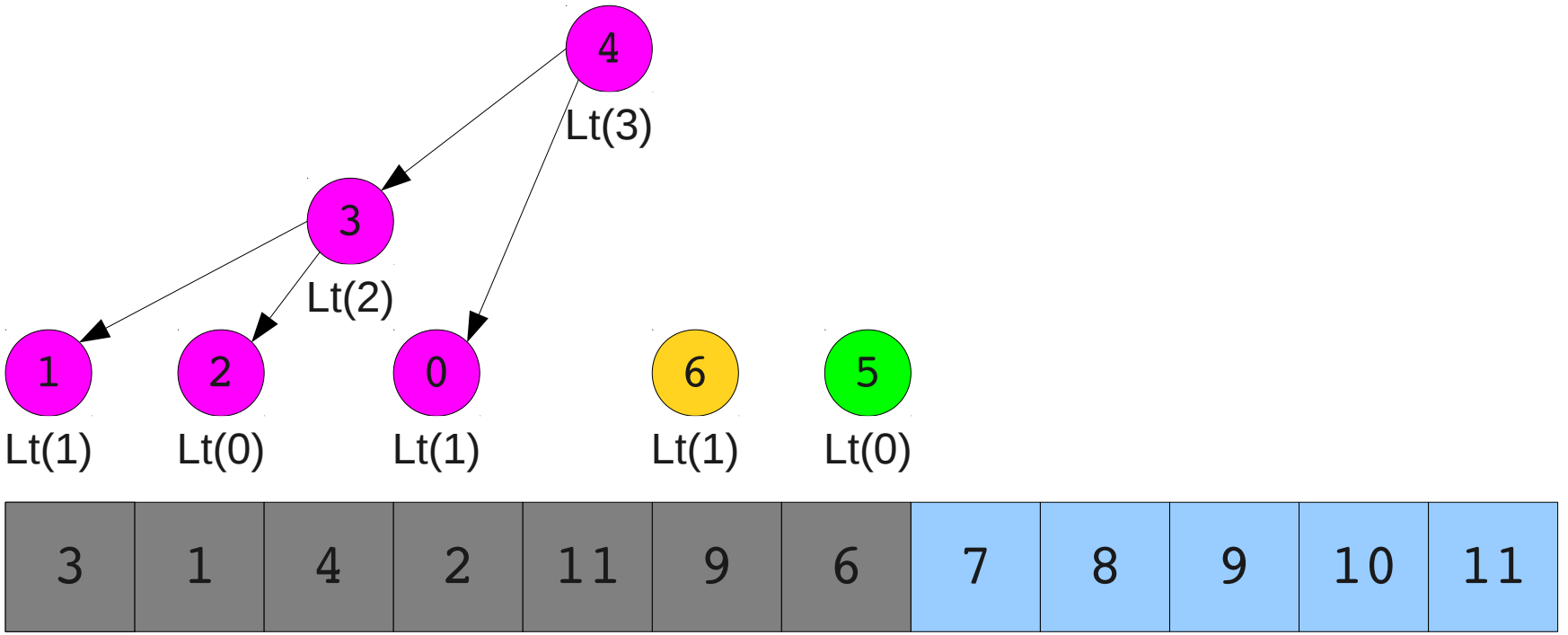
Dequeuing from a Leonardo Heap



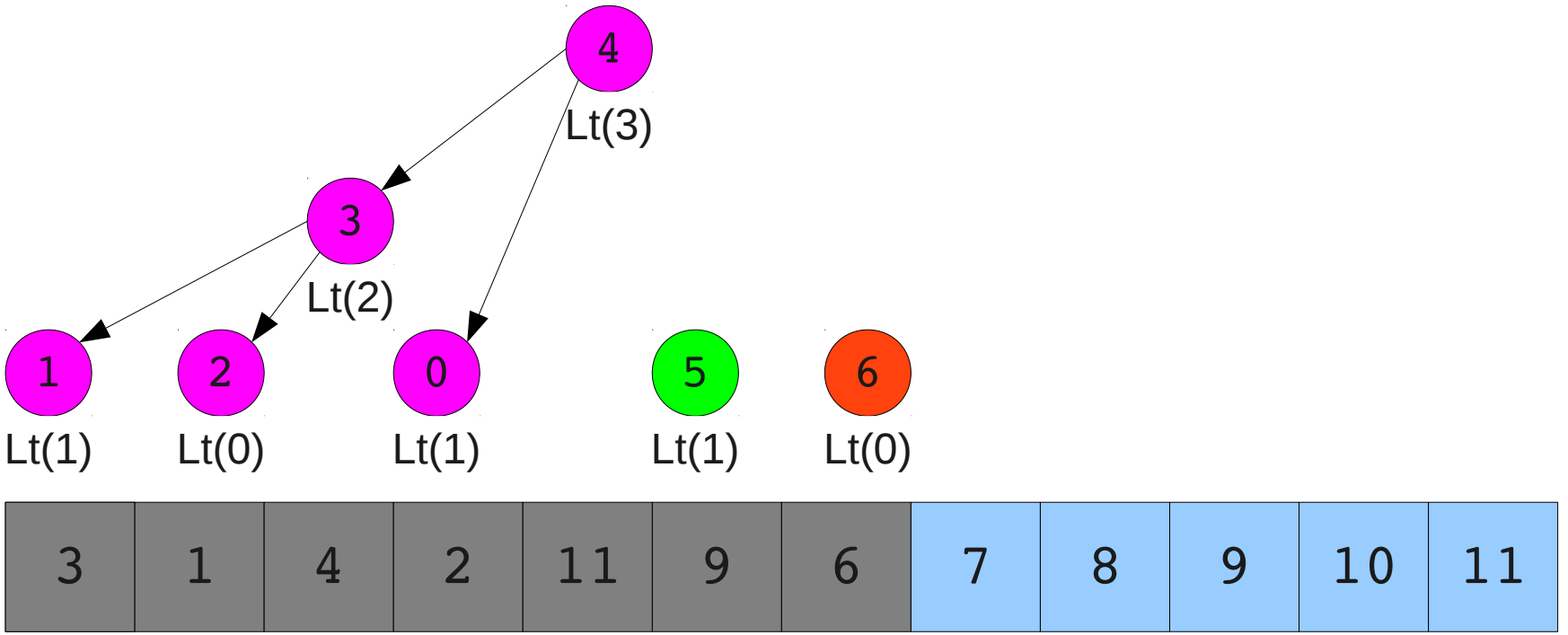
Dequeuing from a Leonardo Heap



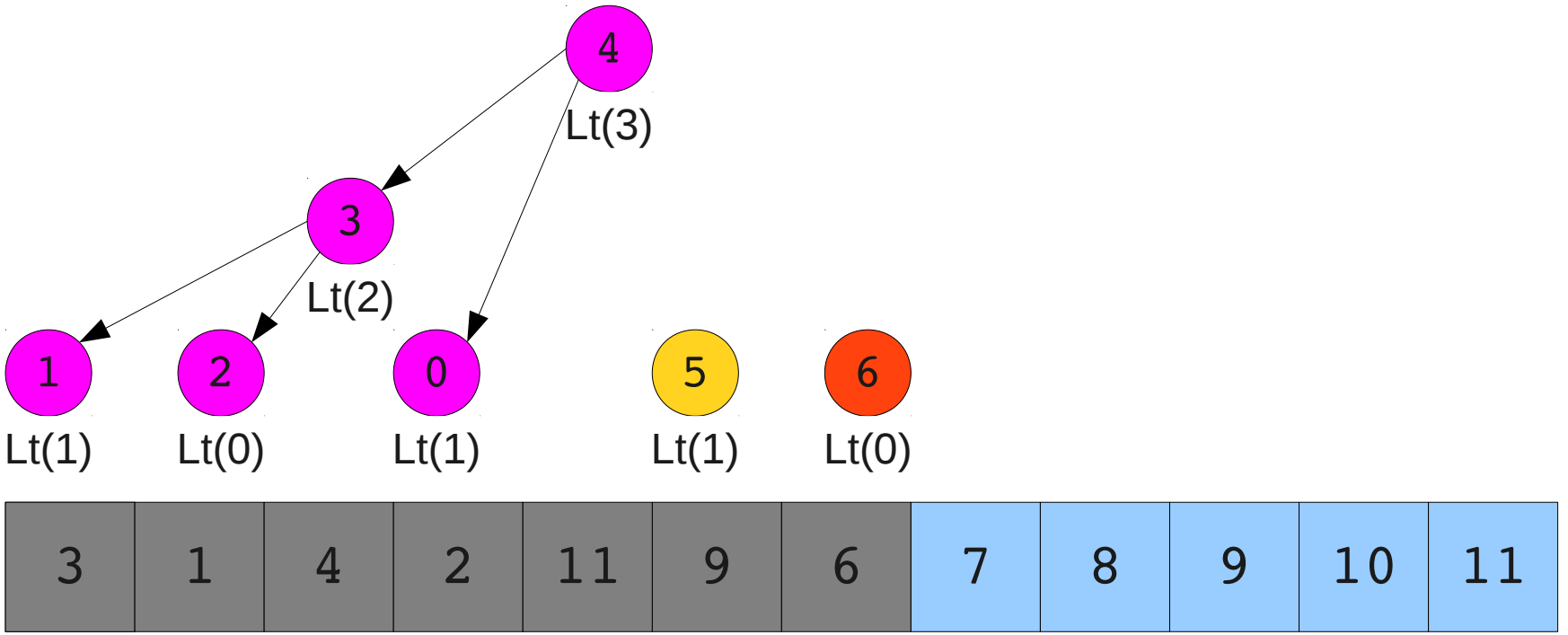
Dequeuing from a Leonardo Heap



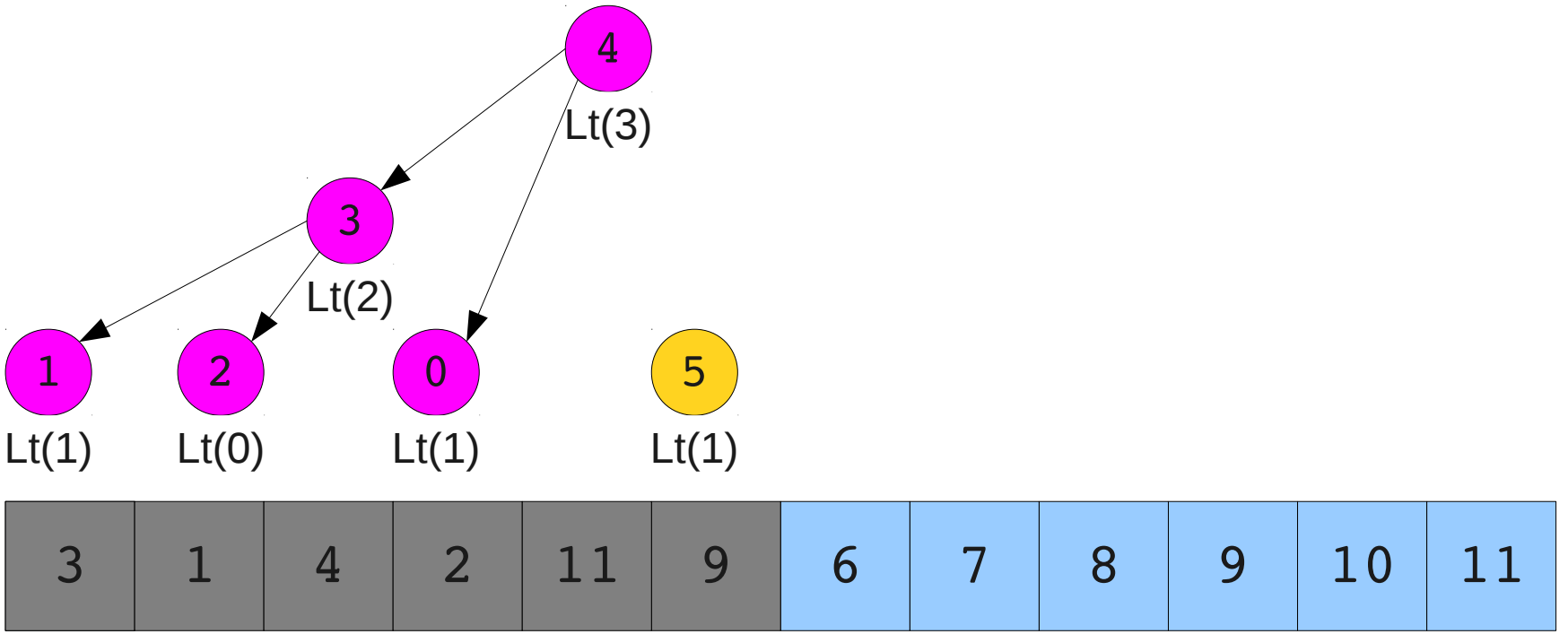
Dequeuing from a Leonardo Heap



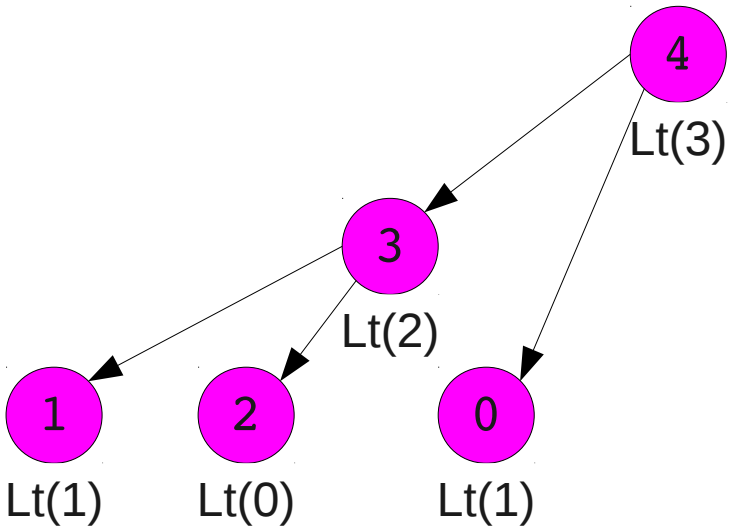
Dequeuing from a Leonardo Heap



Dequeuing from a Leonardo Heap

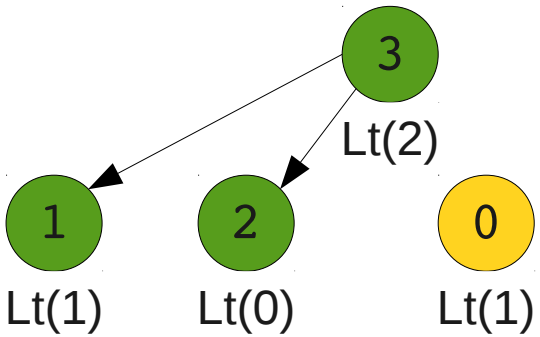


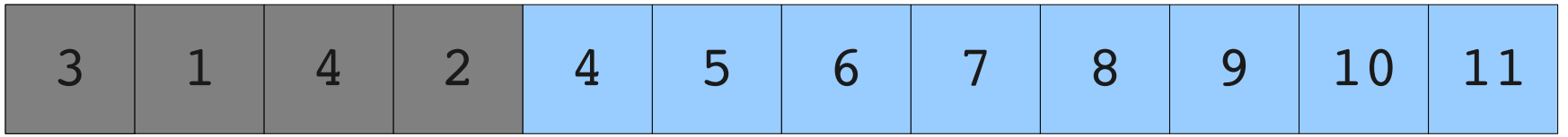
Dequeuing from a Leonardo Heap



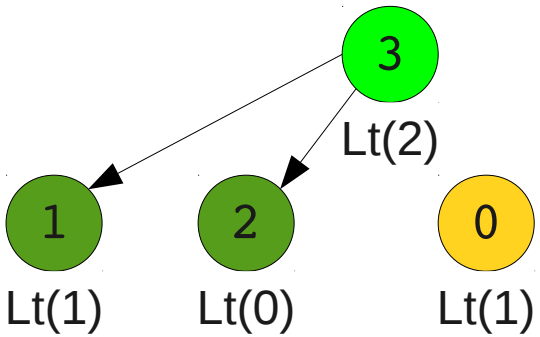
3	1	4	2	11	5	6	7	8	9	10	11

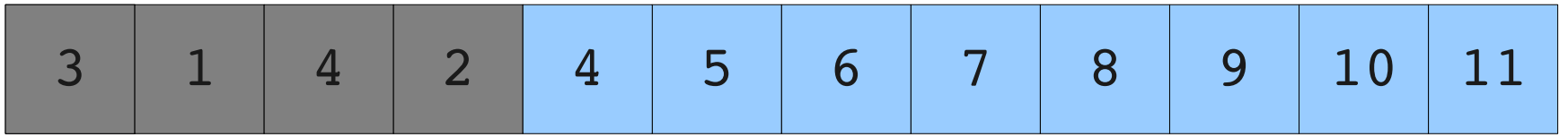
Dequeuing from a Leonardo Heap



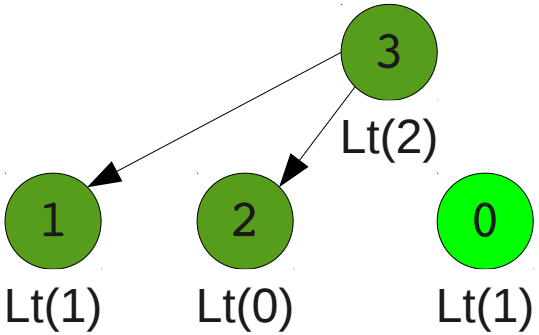


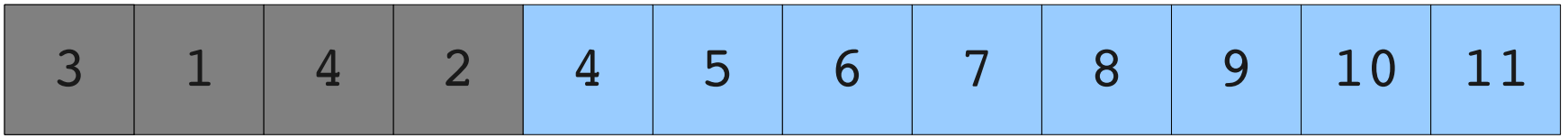
Dequeuing from a Leonardo Heap



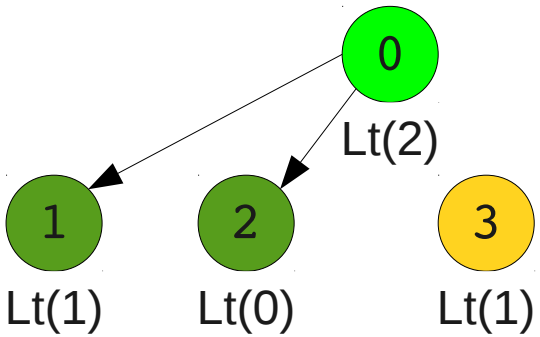


Dequeuing from a Leonardo Heap



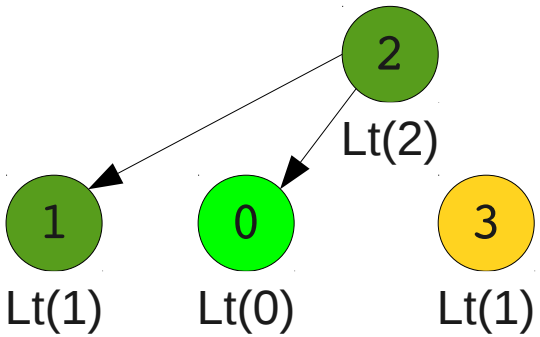


Dequeuing from a Leonardo Heap



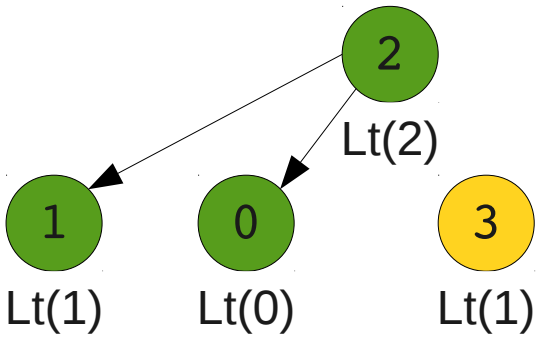
3	1	4	2	4	5	6	7	8	9	10	11

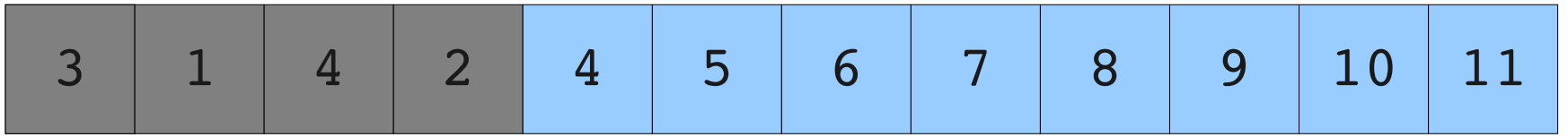
Dequeuing from a Leonardo Heap



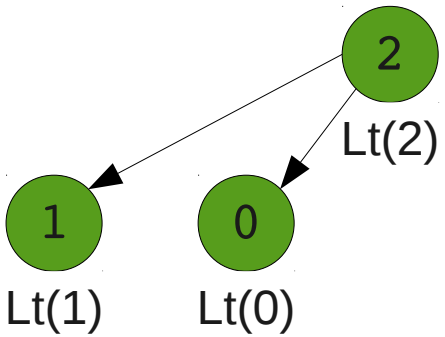
3	1	4	2	4	5	6	7	8	9	10	11

Dequeuing from a Leonardo Heap



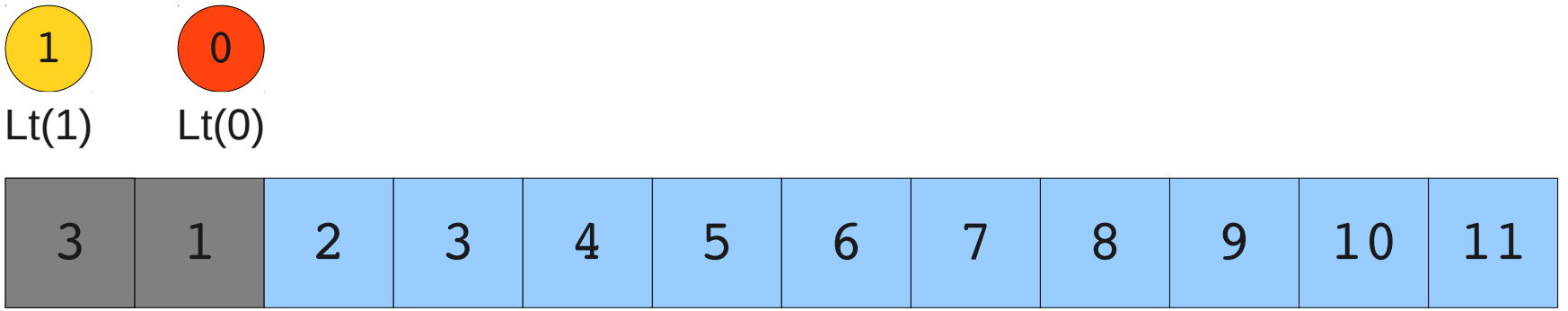


Dequeuing from a Leonardo Heap

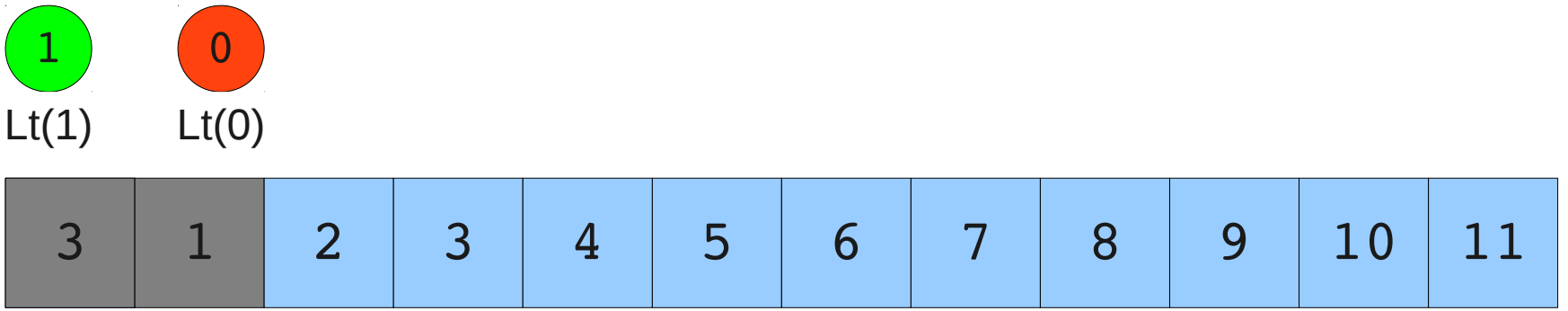


3	1	4	3	4	5	6	7	8	9	10	11

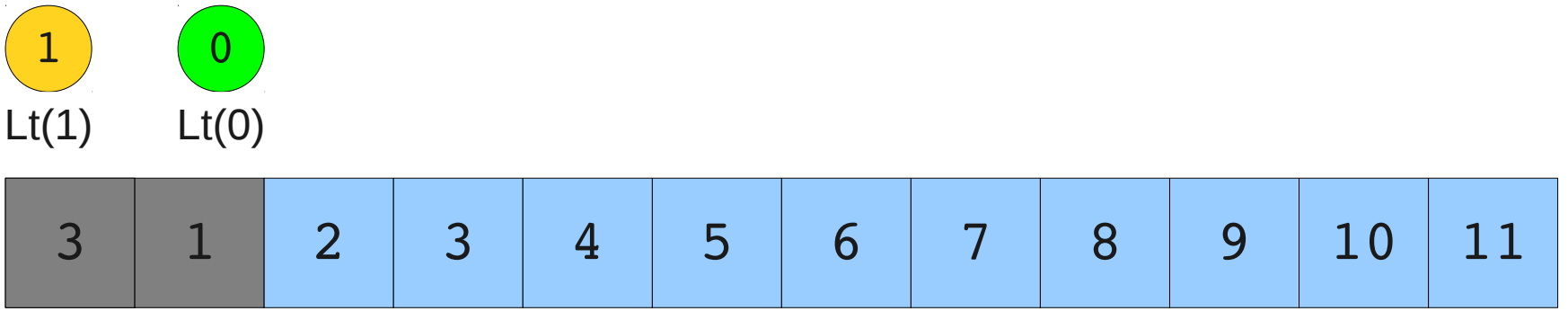
Dequeuing from a Leonardo Heap



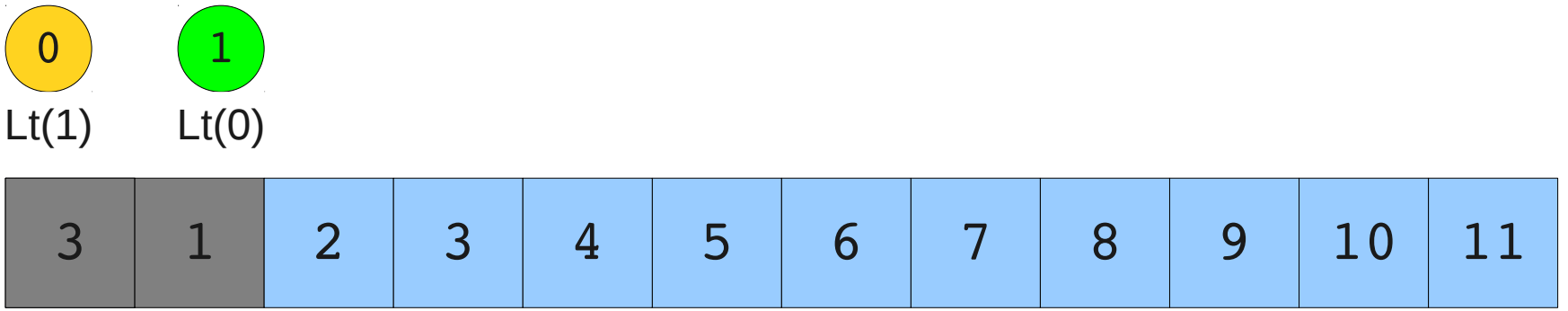
Dequeuing from a Leonardo Heap



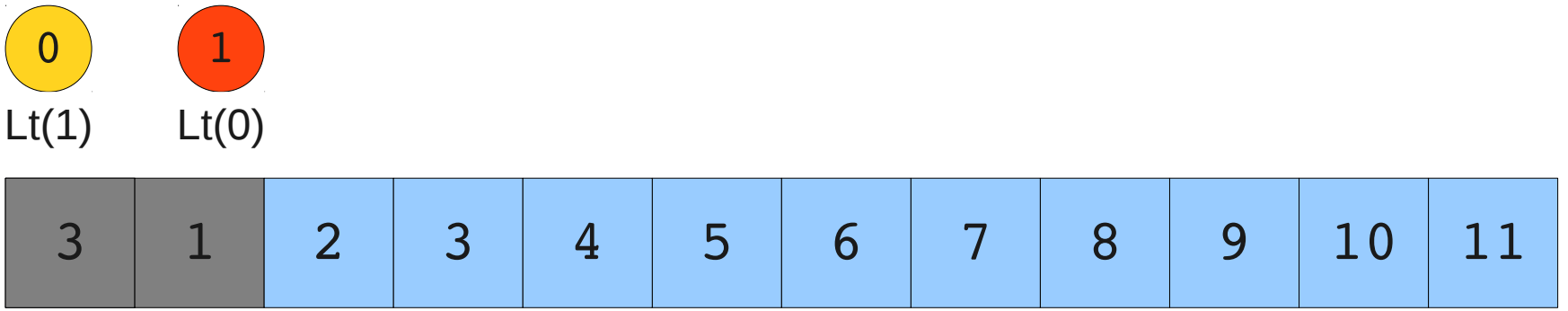
Dequeuing from a Leonardo Heap



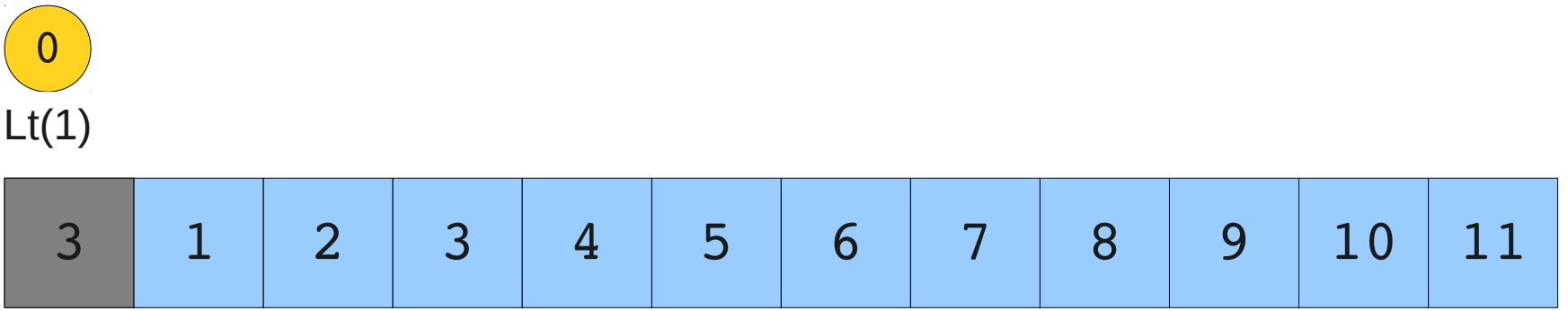
Dequeuing from a Leonardo Heap



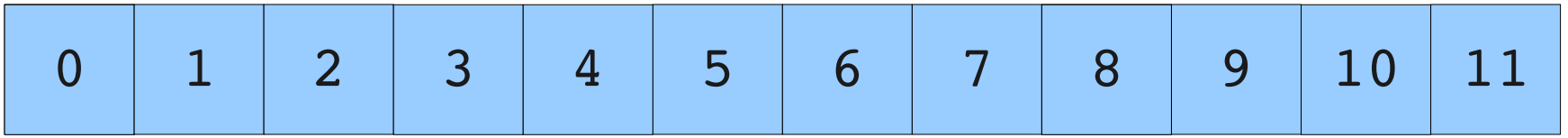
Dequeuing from a Leonardo Heap



Dequeuing from a Leonardo Heap



Dequeuing from a Leonardo Heap



Summary of Dequeue

- Remove topmost node of rightmost heap.
- If it has no children, we're done.
- Otherwise:
  - Fix up the left of the two heaps.
  - Then fix up the right of the two heaps.

Runtime Analysis

- Each insertion does a scan over (potentially) all the heaps, then does a bubble-down.
- Each deletion does up to two scans over the heaps and two bubble-downs.
- Runtime per operation: **O(#heaps + heapsize)**
- **What are #heaps and heapsize?**

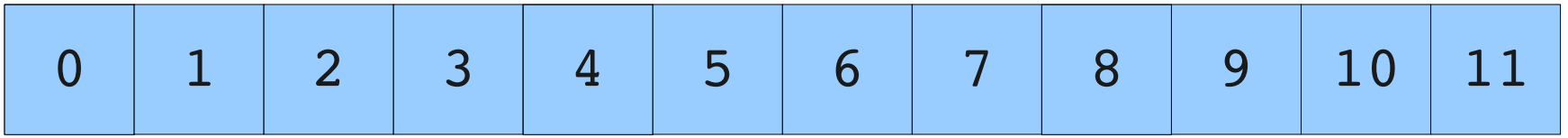
Leonardo Numbers

- How many nodes are in Lt(n)?
- Answer: the nth **Leonardo number**:
  - L(0) = 1
  - L(1) = 1
  - L(n + 2) = 1 + L(n) + L(n + 1)
- First few terms are 1, 1, 3, 5, 9, 15, 25, 41, …

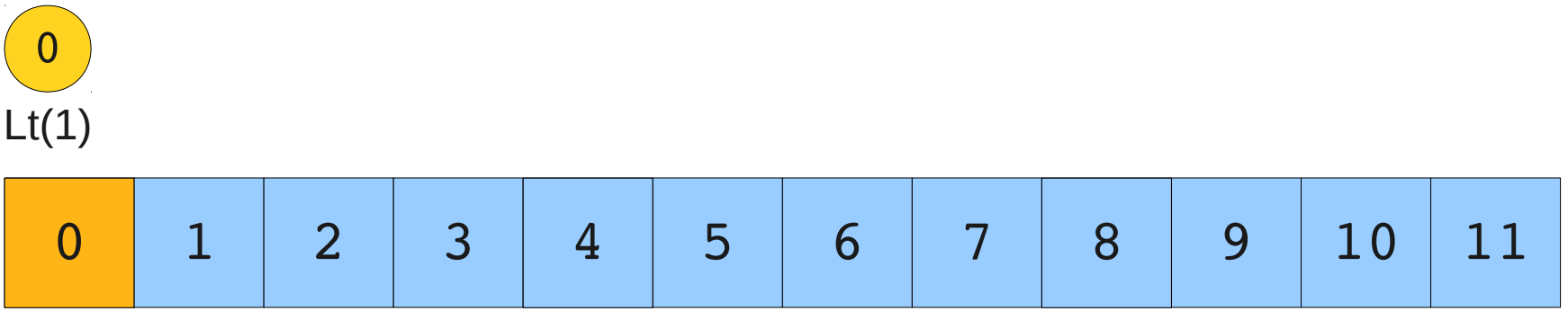
Leonardo Numbers

- Important fact: **L(k) = 2F(k + 1) – 1**
  - Can prove by induction.
- Leonardo numbers grow **exponentially fast**; only **O(lg n)** Leonardo numbers smaller than n.
- Maximum number of trees: **O(lg n)**
- Maximum tree depth: **O(lg n)**
- **Runtime is O(n lg n)**.

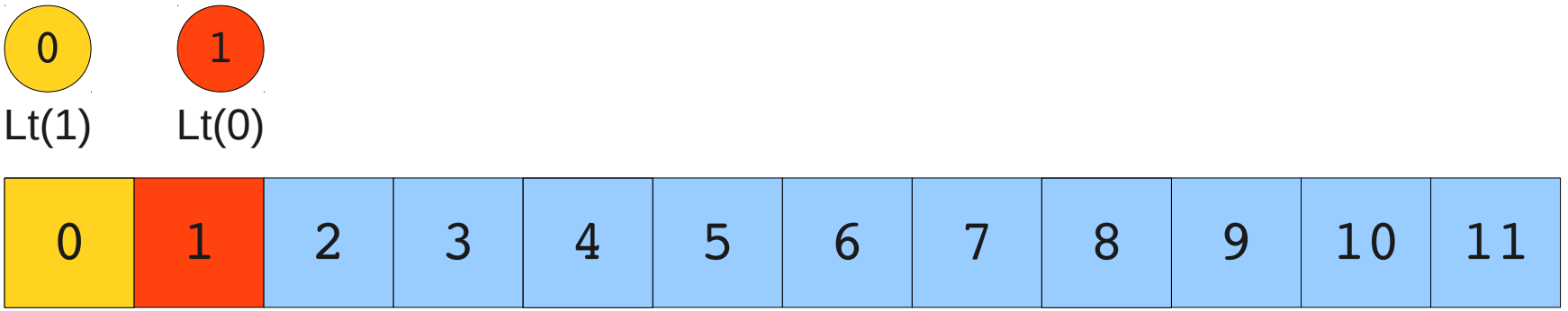
Sorting a Sorted Sequence



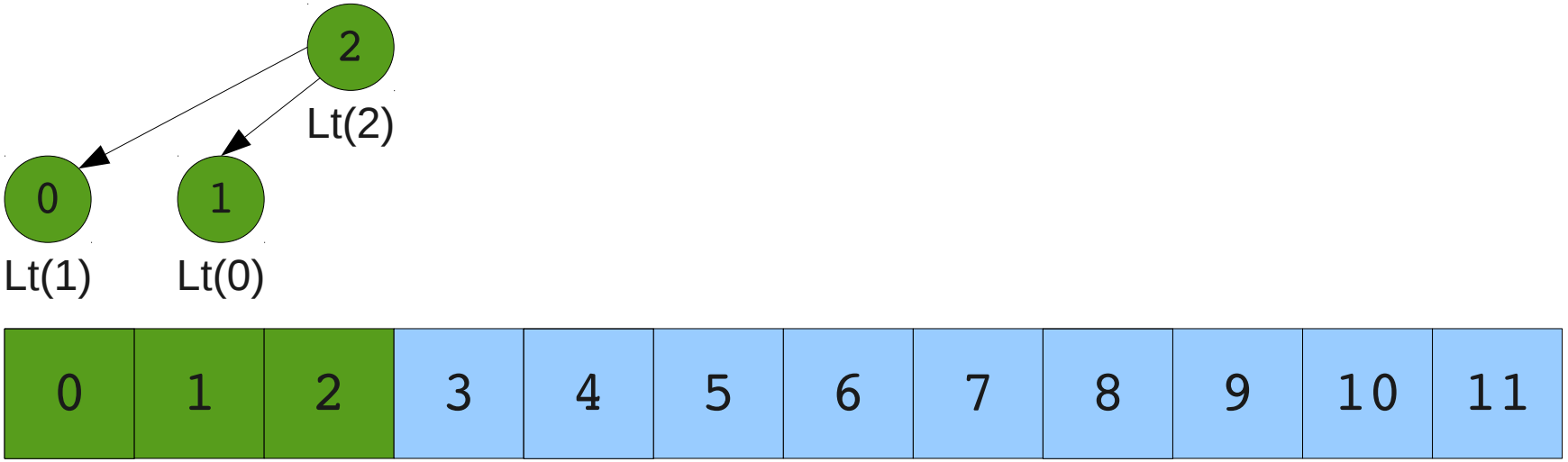
Sorting a Sorted Sequence



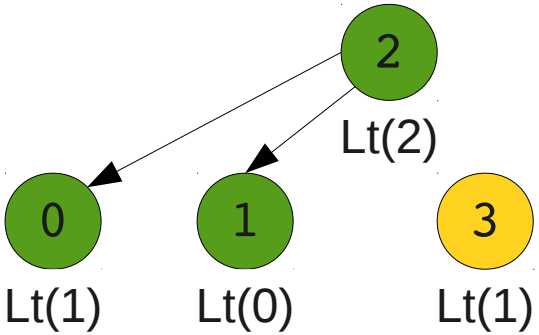
Sorting a Sorted Sequence

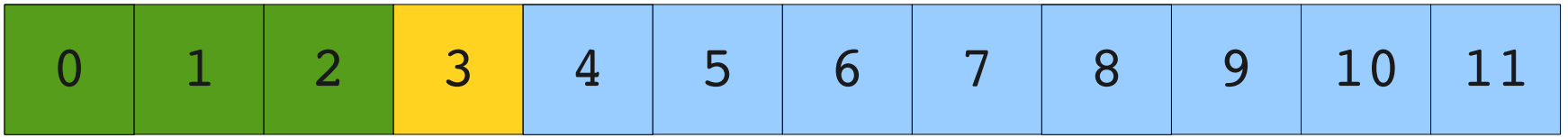


Sorting a Sorted Sequence

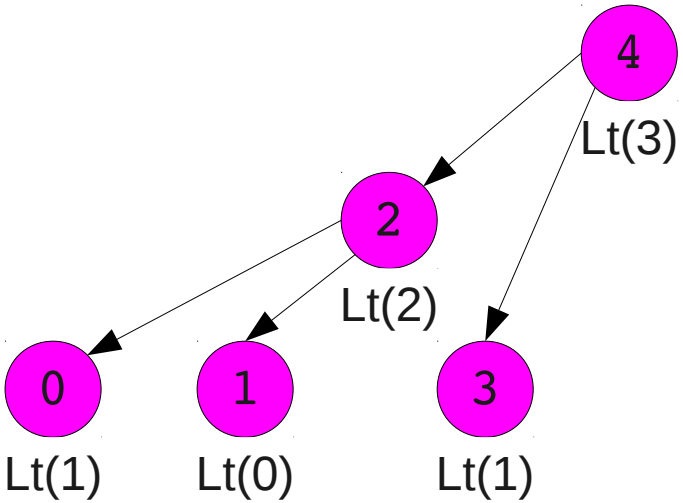


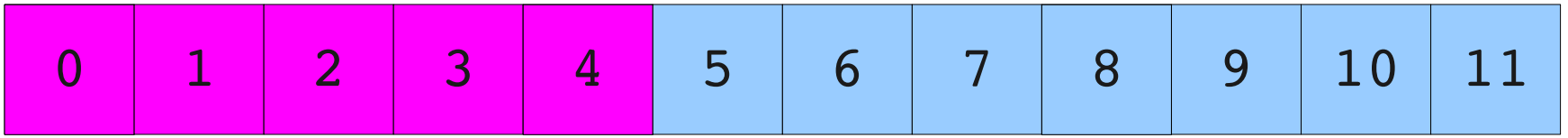
Sorting a Sorted Sequence



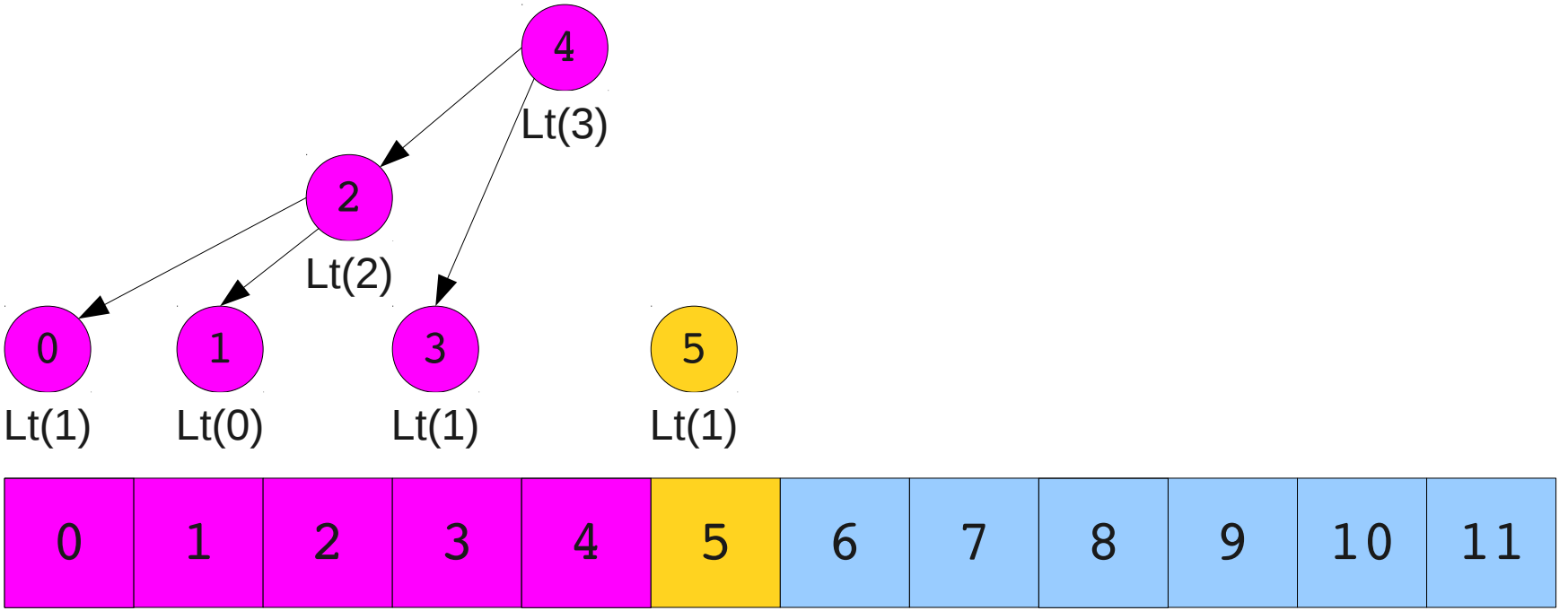


Sorting a Sorted Sequence

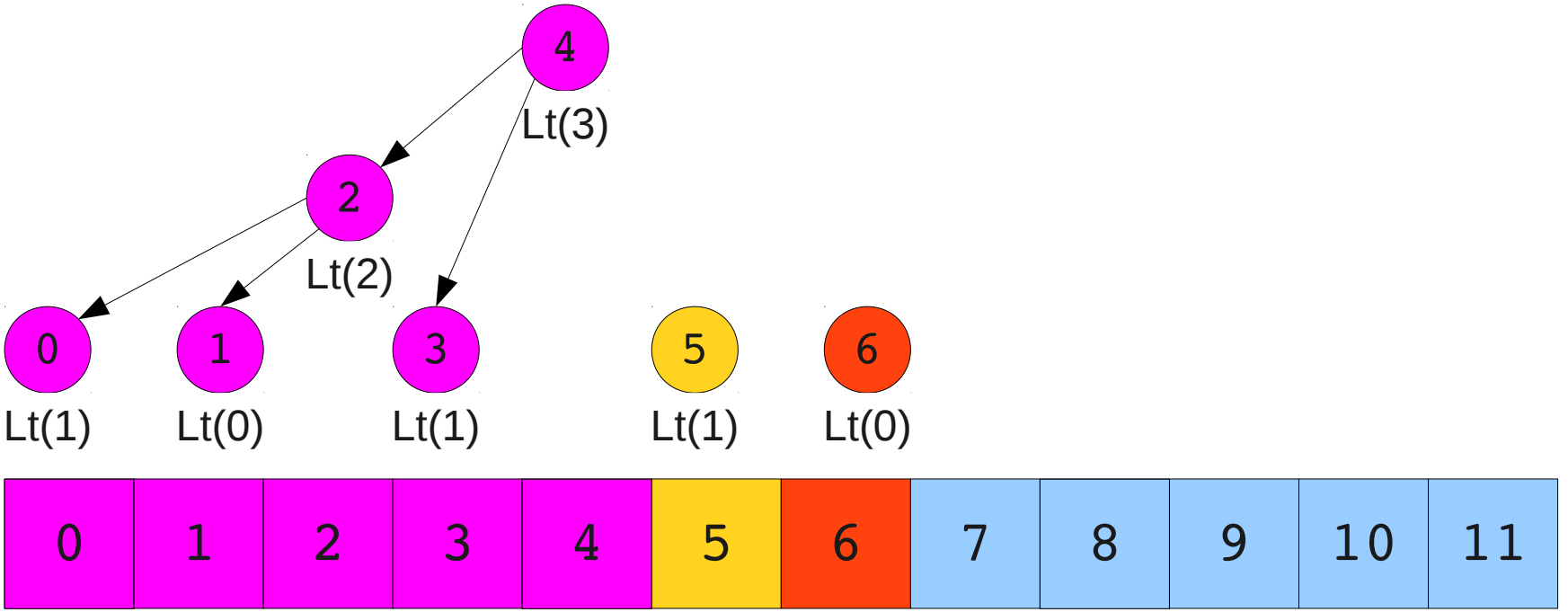




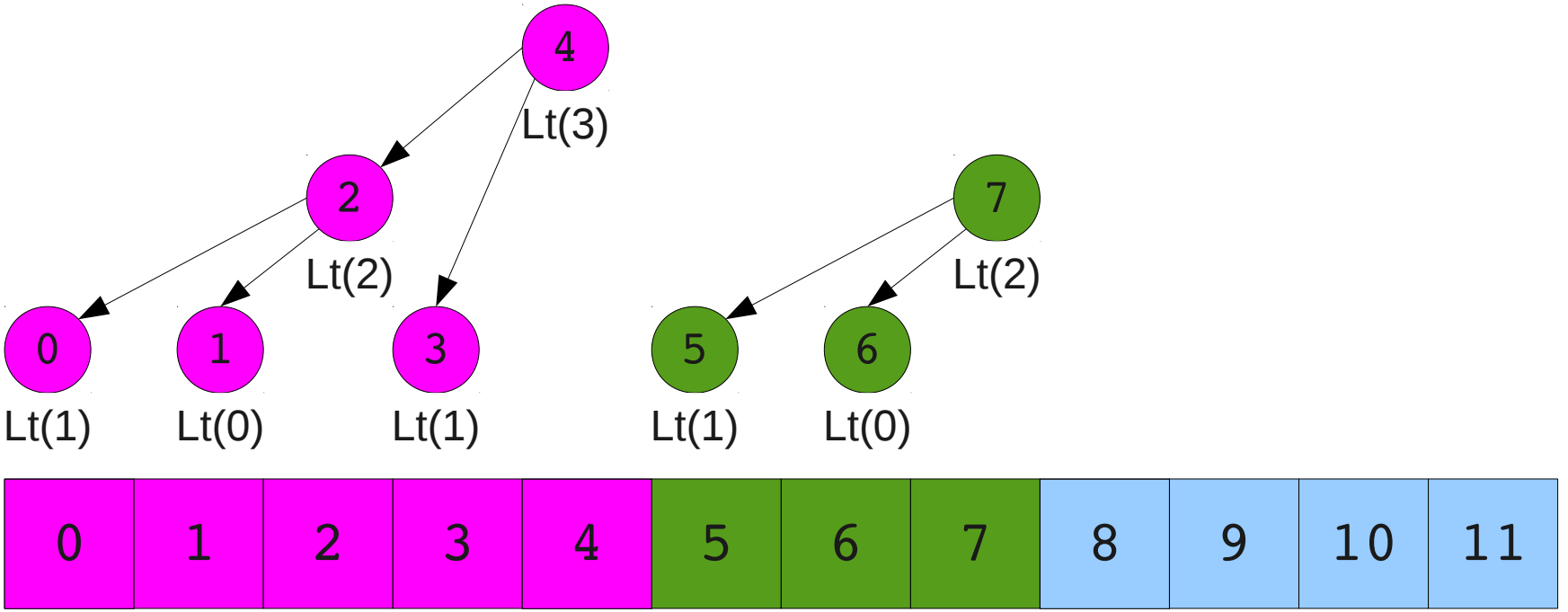
Sorting a Sorted Sequence



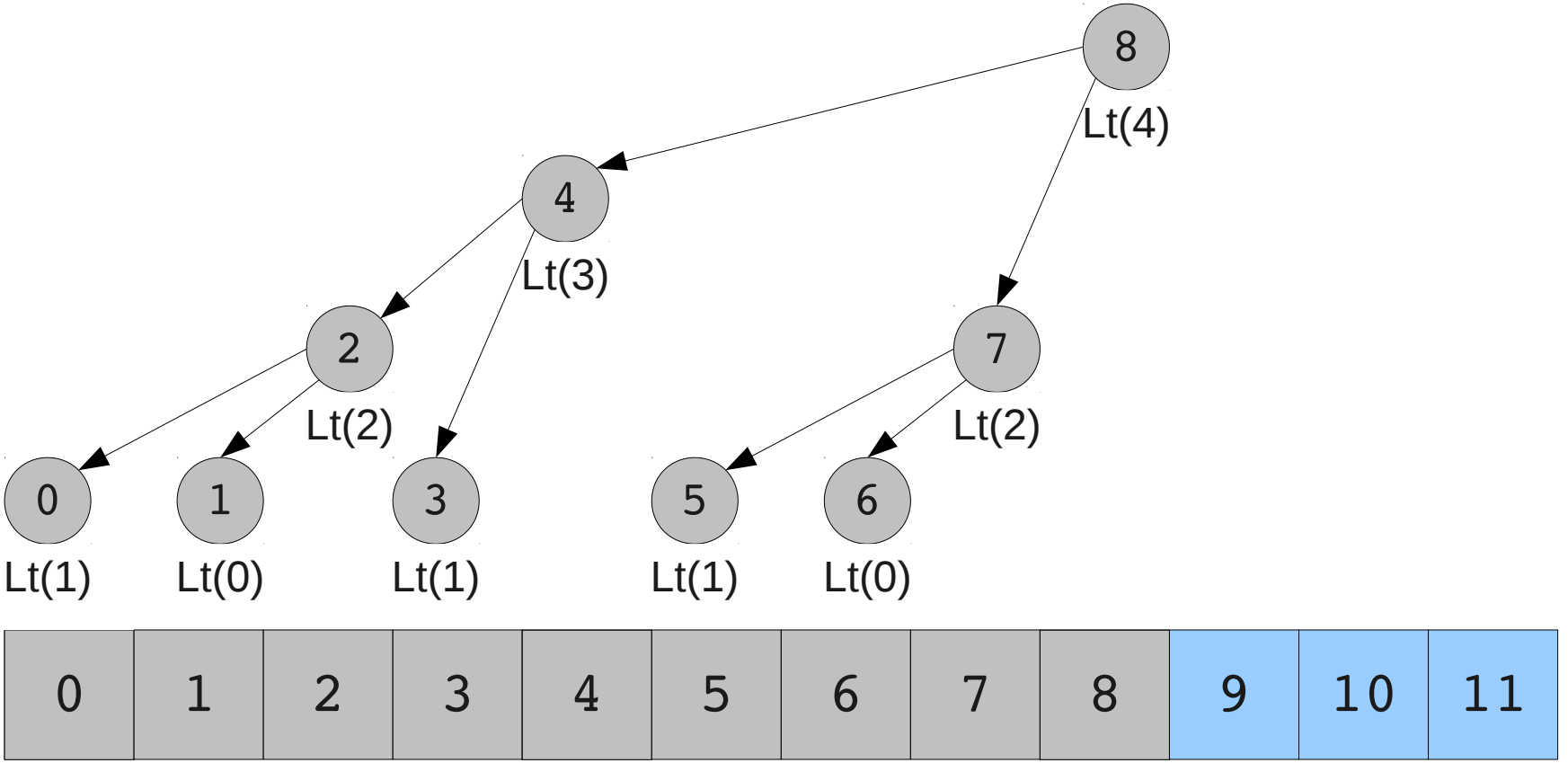
Sorting a Sorted Sequence



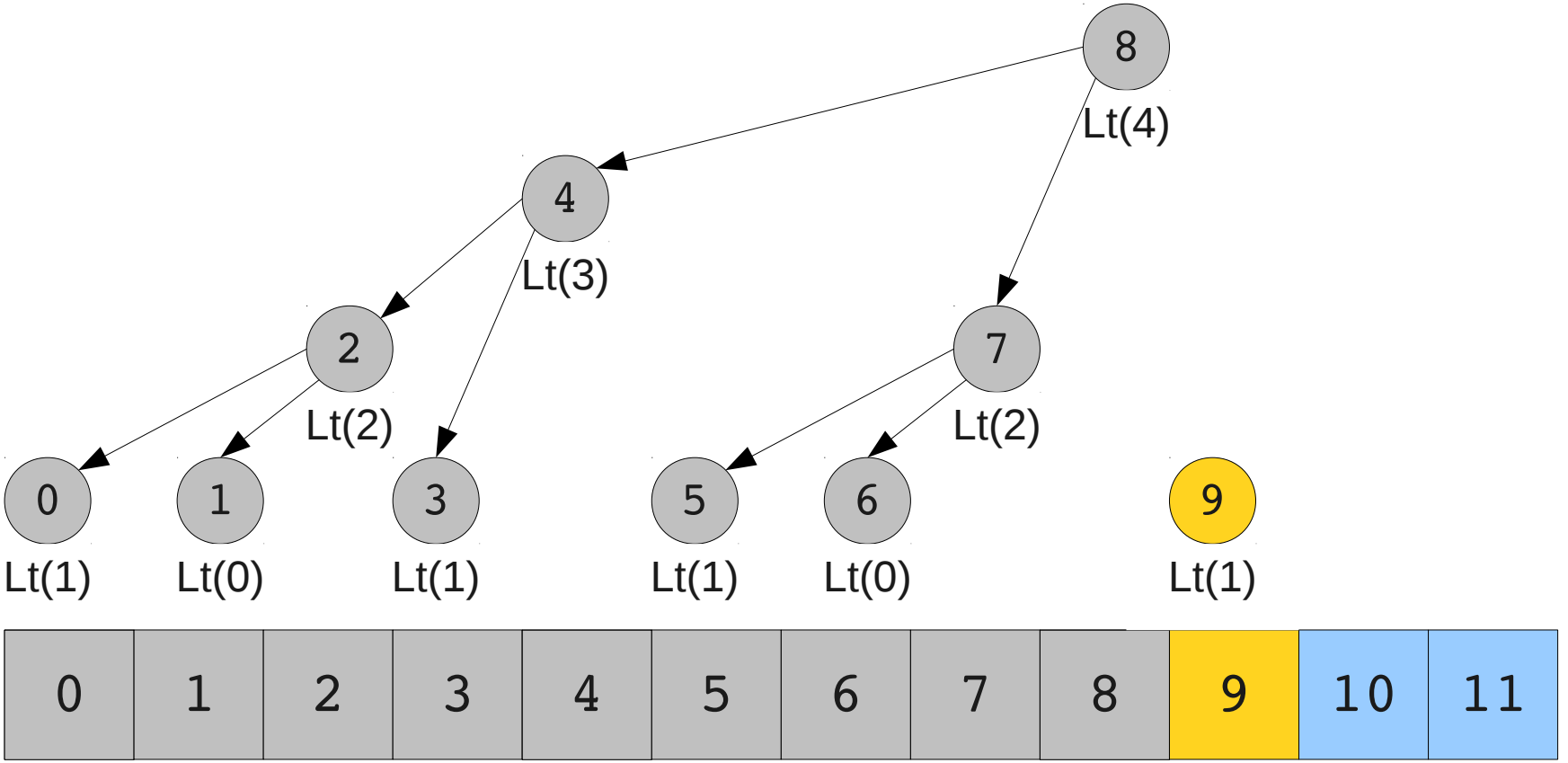
Sorting a Sorted Sequence



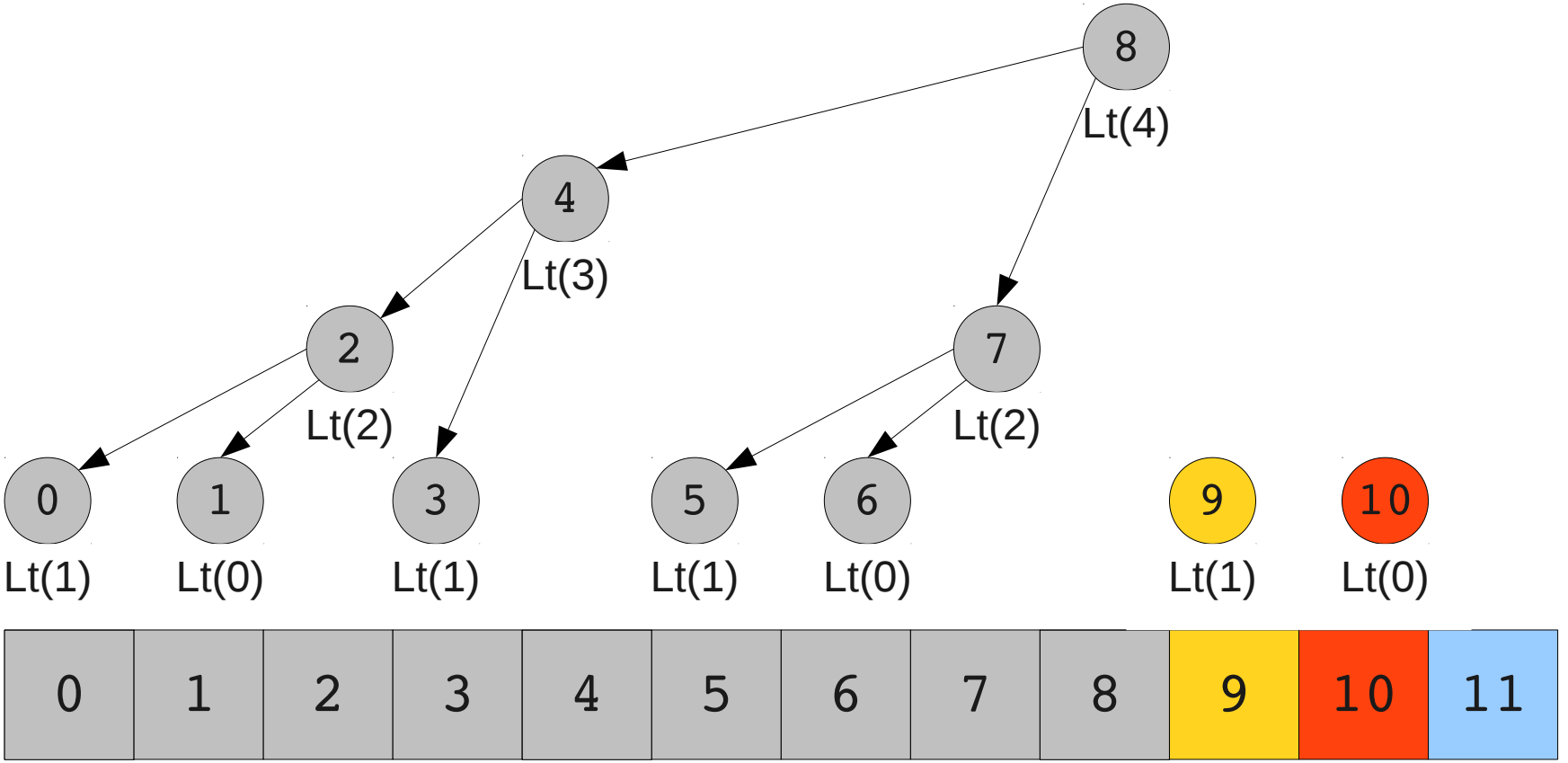
Sorting a Sorted Sequence



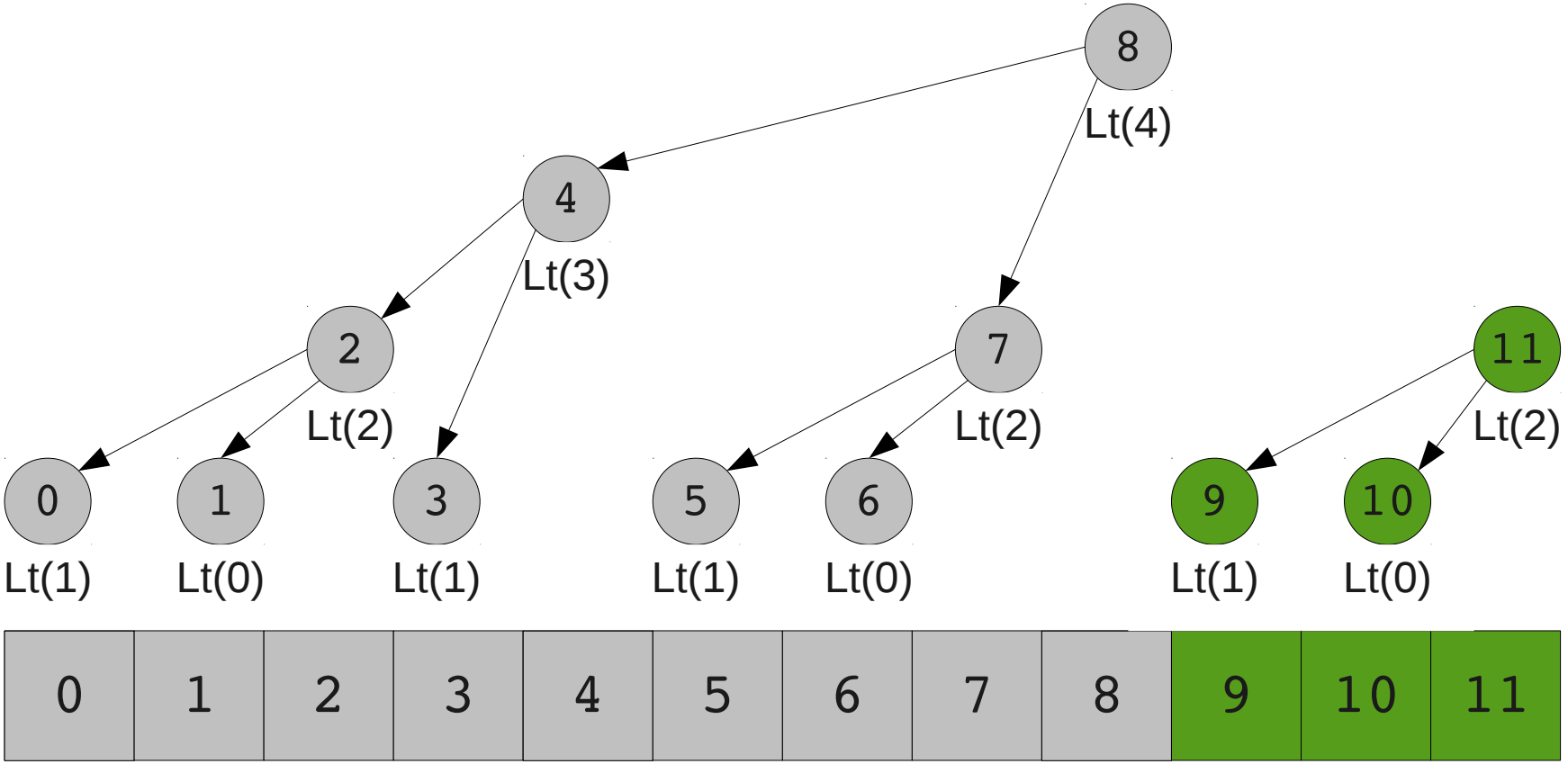
Sorting a Sorted Sequence



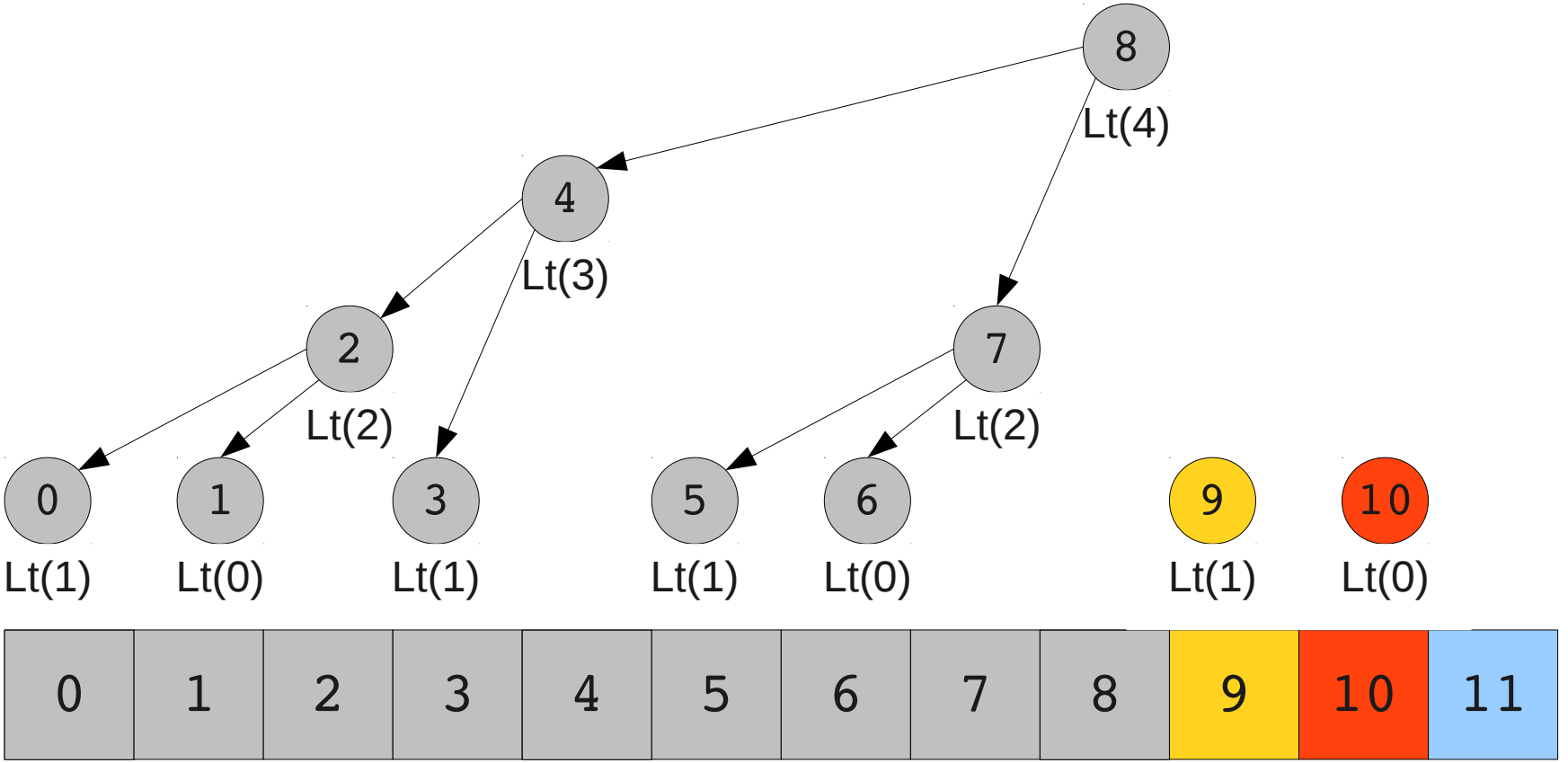
Sorting a Sorted Sequence



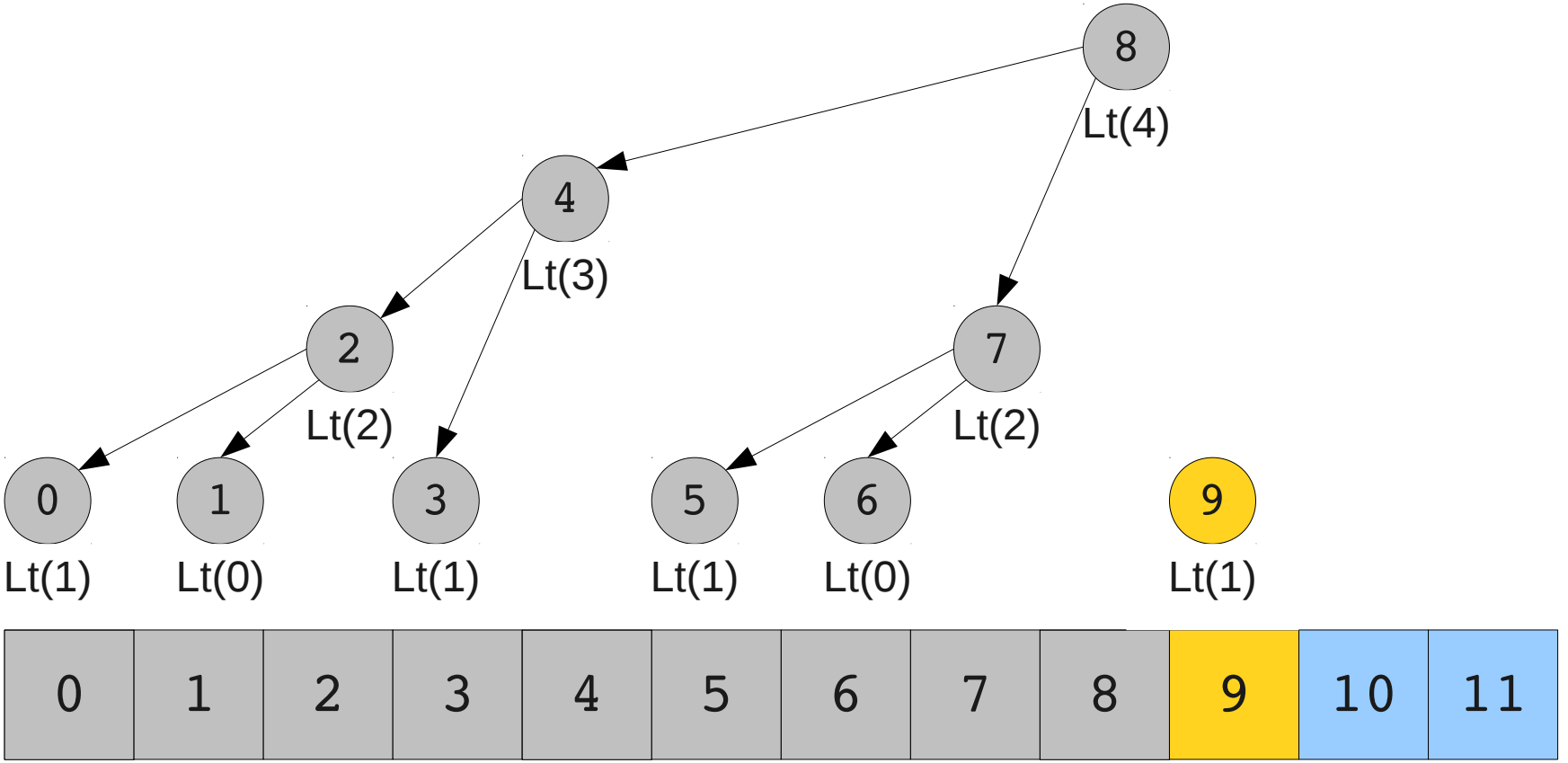
Sorting a Sorted Sequence



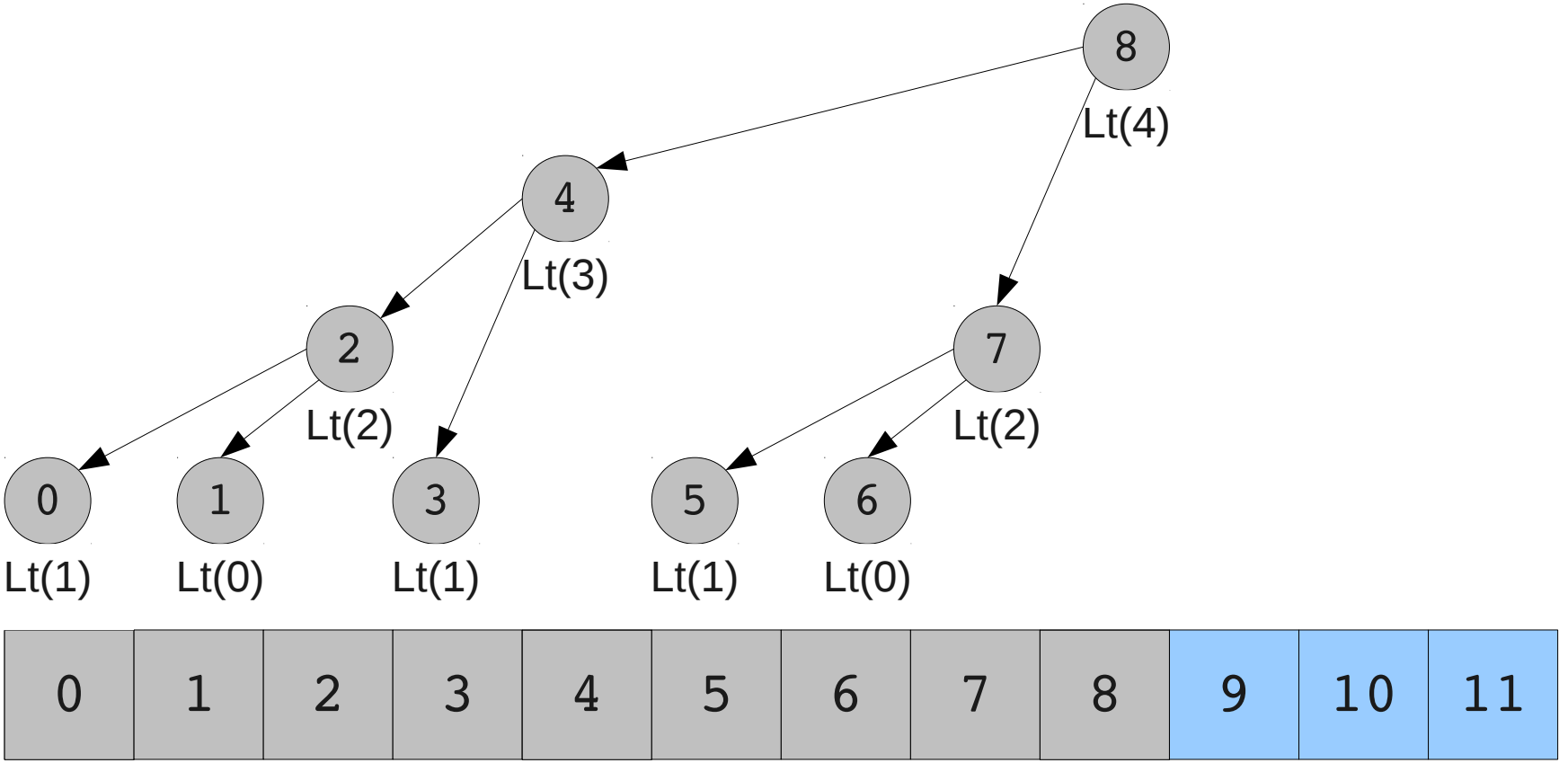
Sorting a Sorted Sequence



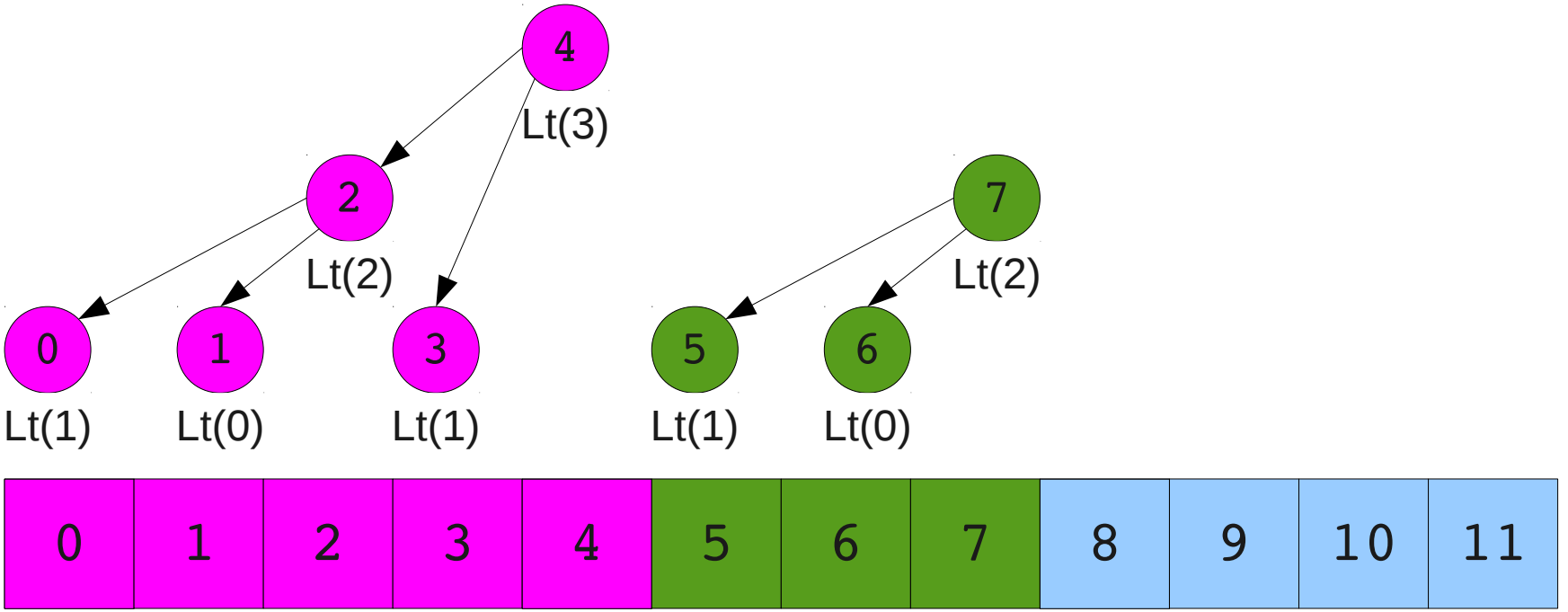
Sorting a Sorted Sequence



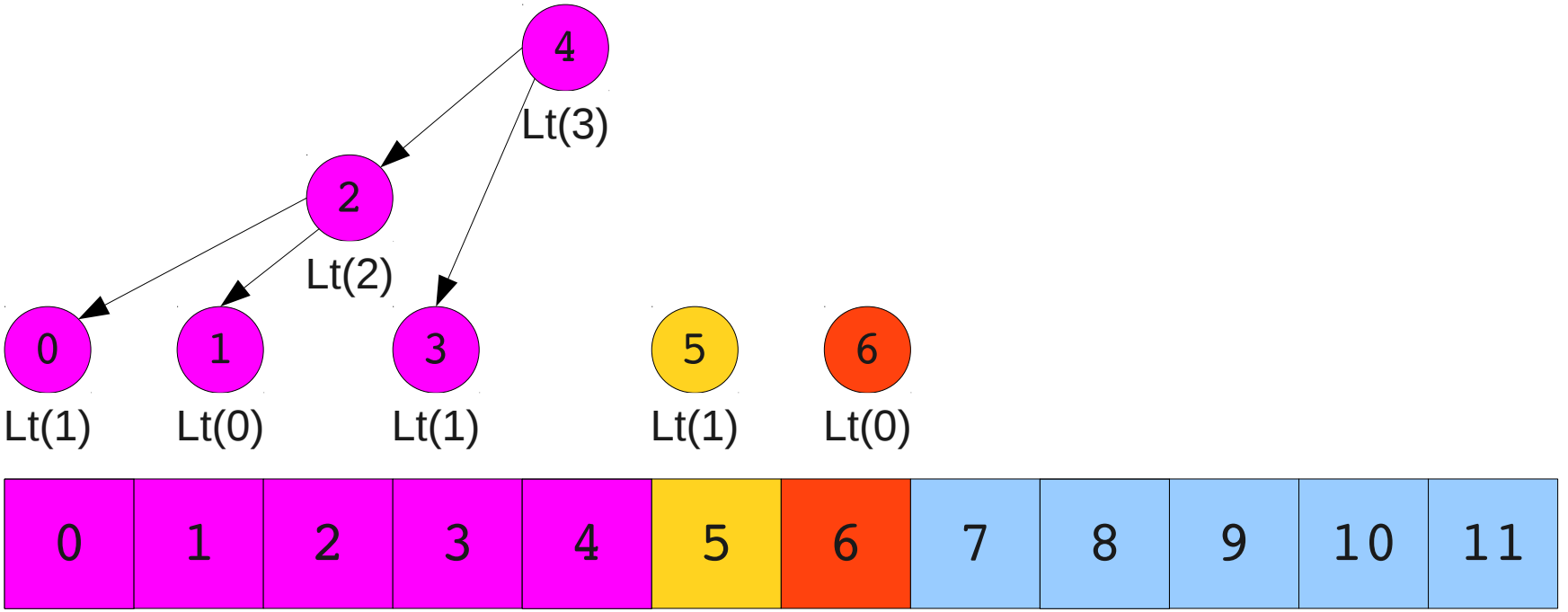
Sorting a Sorted Sequence



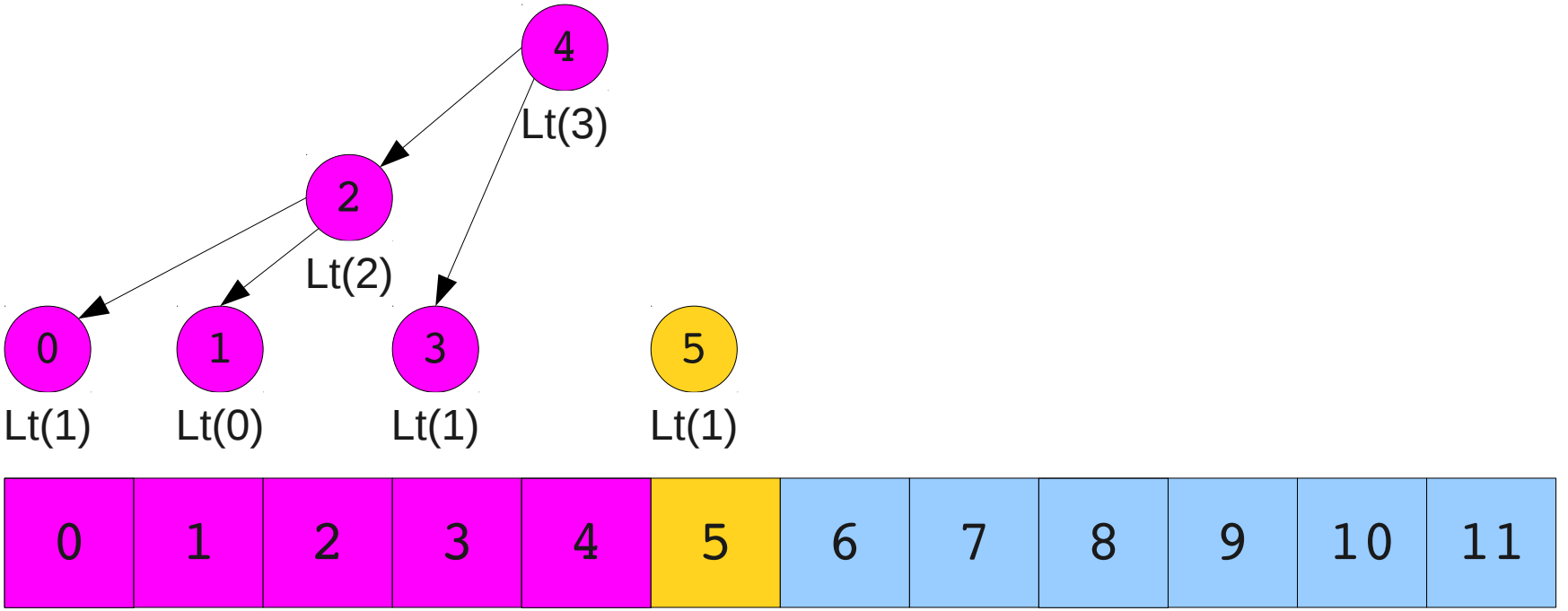
Sorting a Sorted Sequence



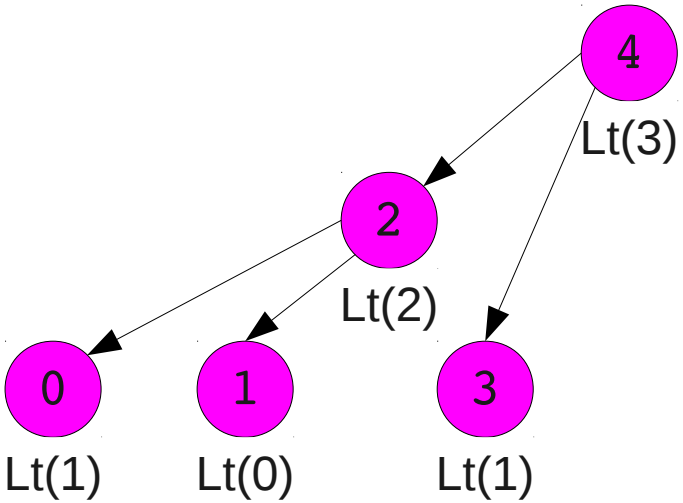
Sorting a Sorted Sequence

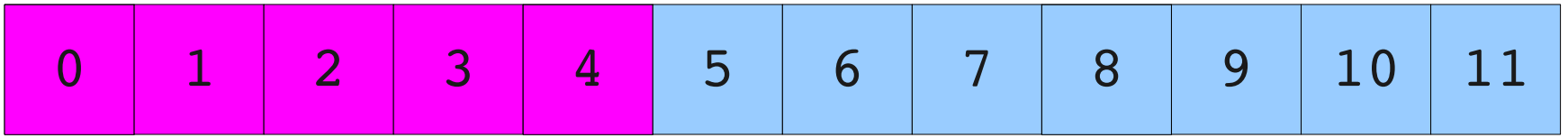


Sorting a Sorted Sequence

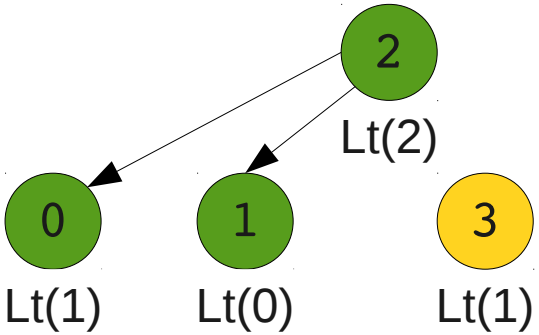


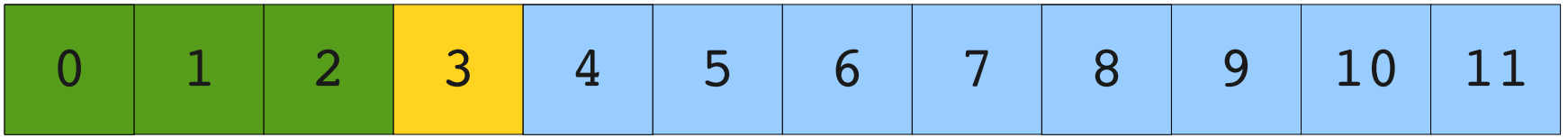
Sorting a Sorted Sequence



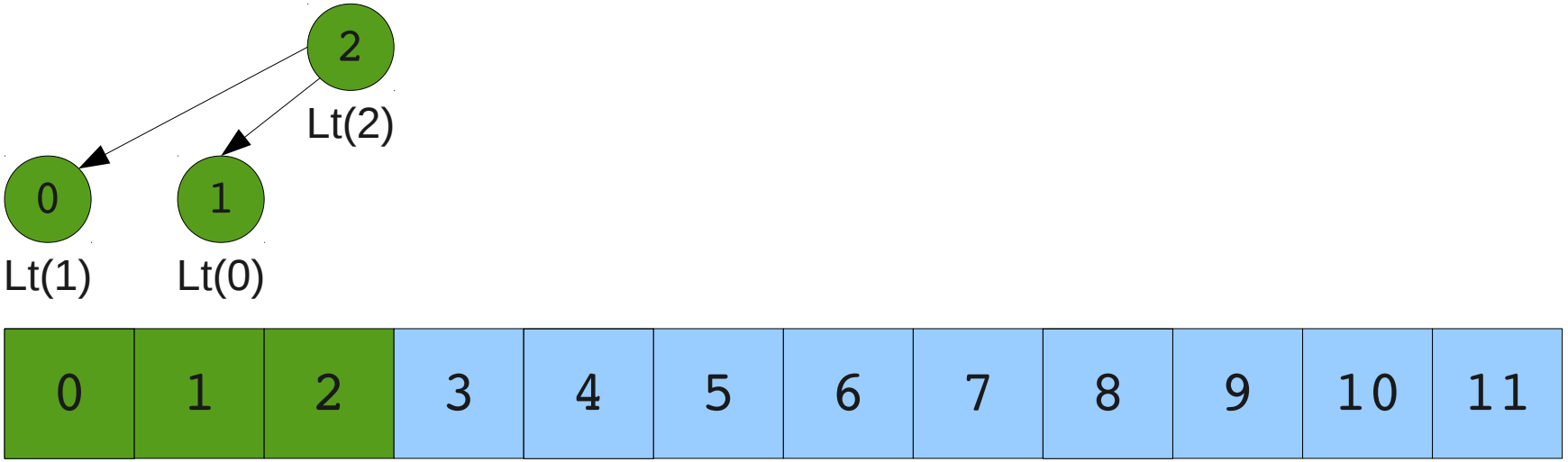


Sorting a Sorted Sequence

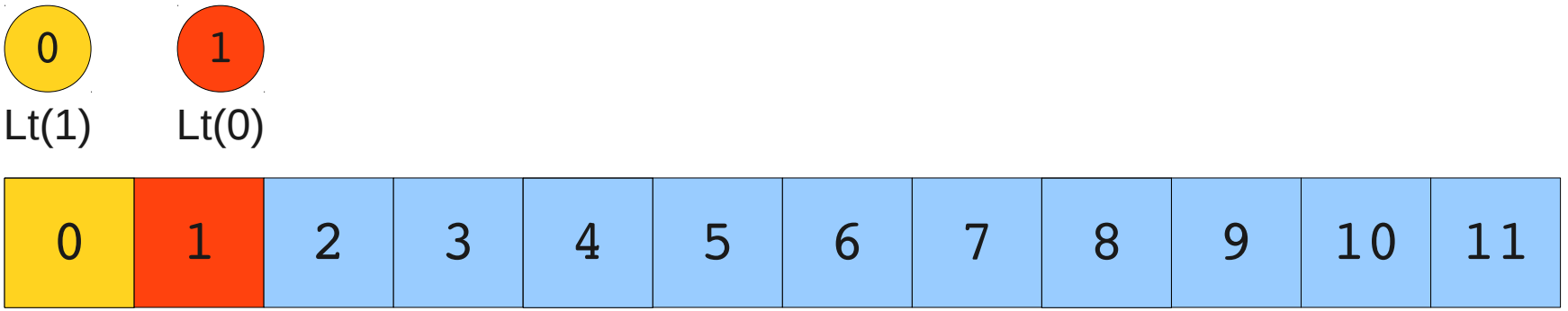




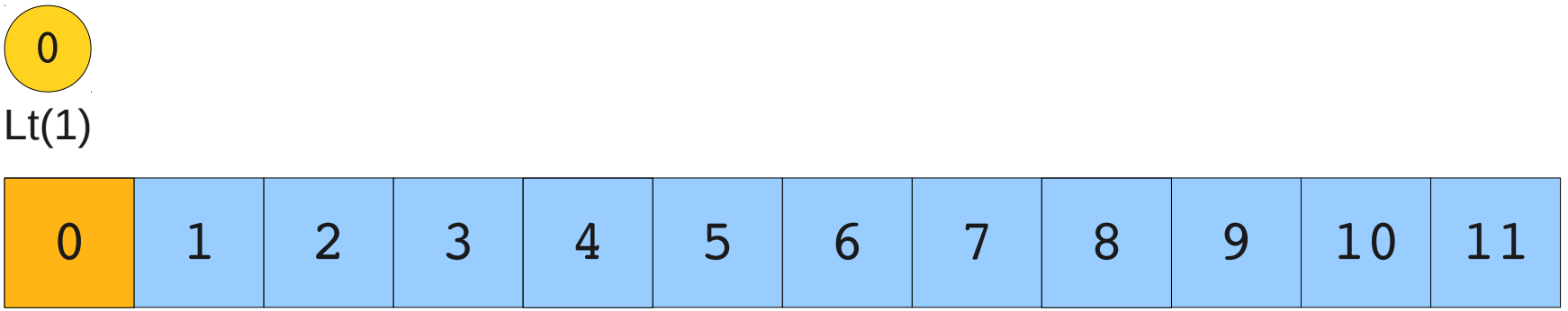
Sorting a Sorted Sequence



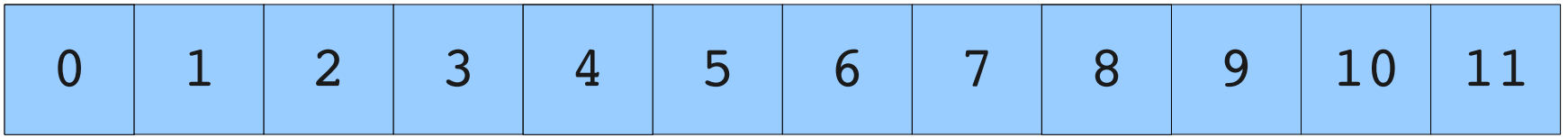
Sorting a Sorted Sequence



Sorting a Sorted Sequence



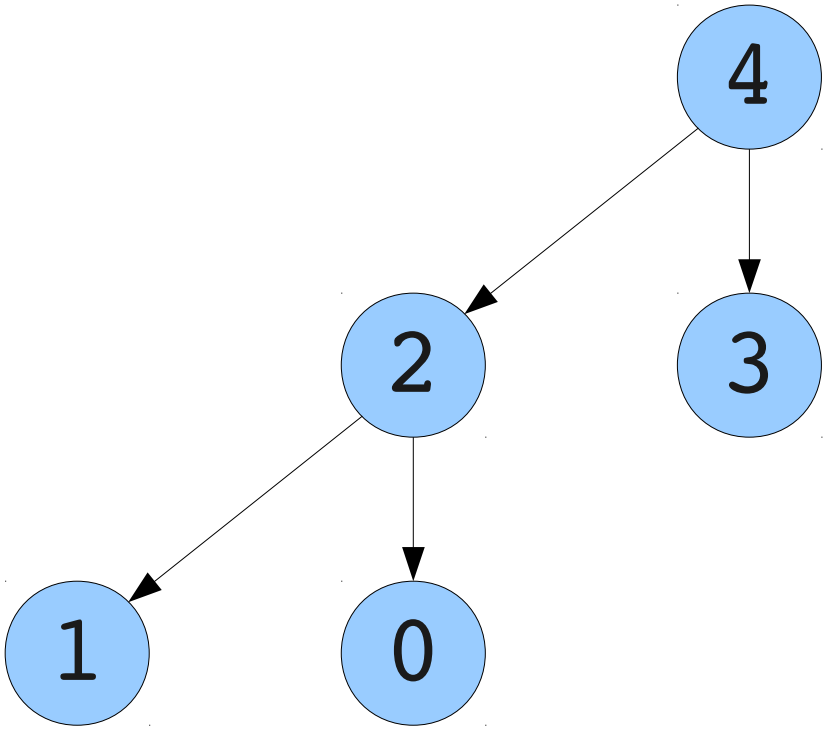
Sorting a Sorted Sequence



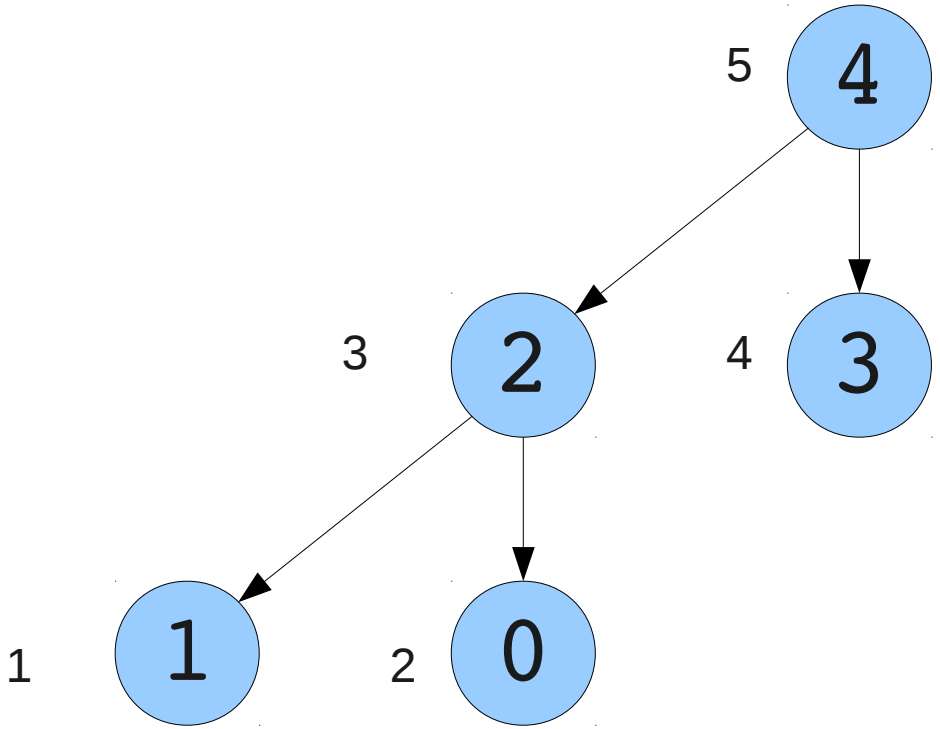
Sorting an already-sorted sequence with a Leonardo heap takes time **O(n)**.

Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Binary Heap	Naïve Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Implicit Binary Heap	Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
Leonardo Heap	Very Naïve Smoothsort	$O(n)$	$O(n \lg n)$	$O(n)$

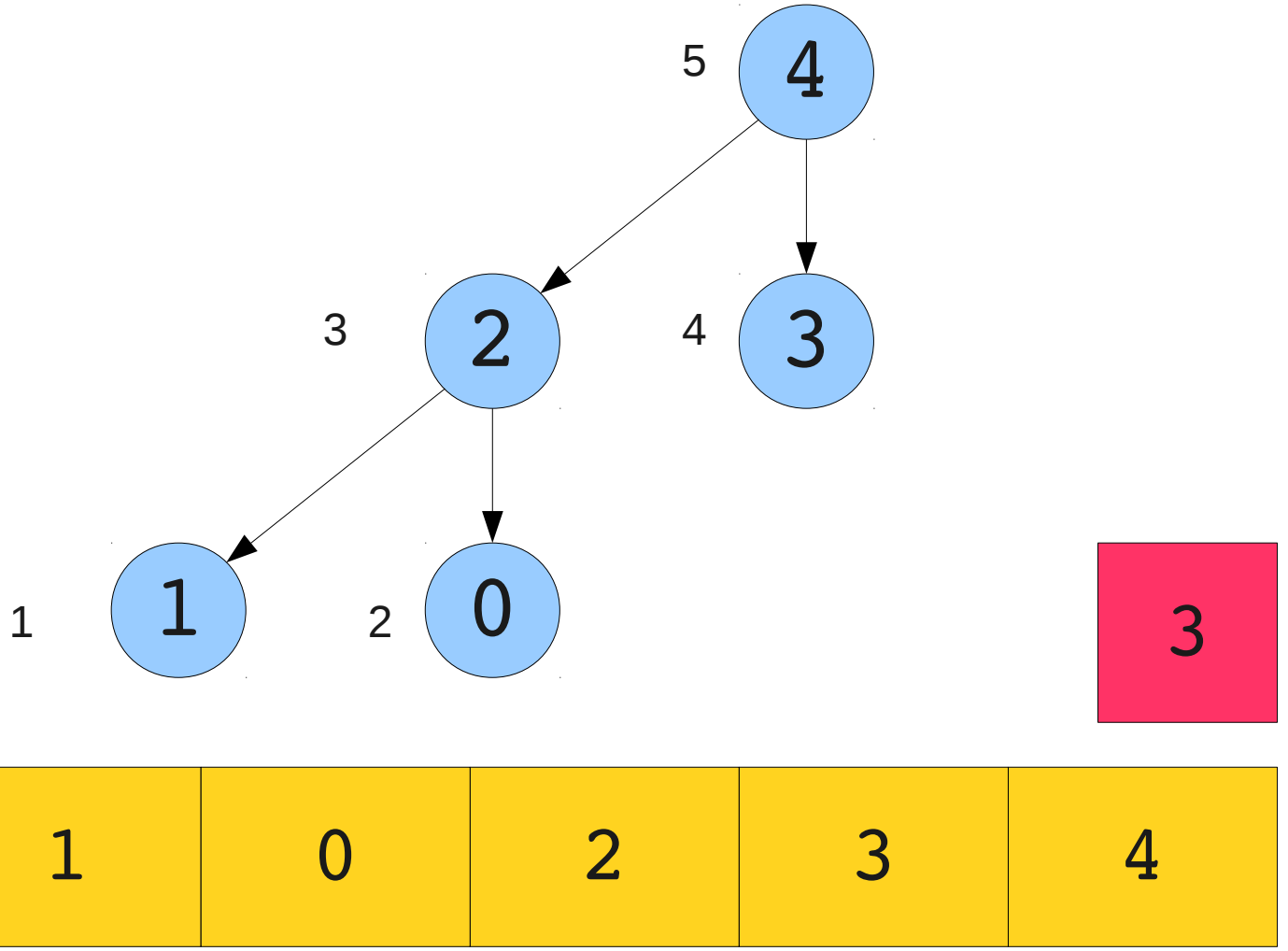
Implicit Leonardo Trees



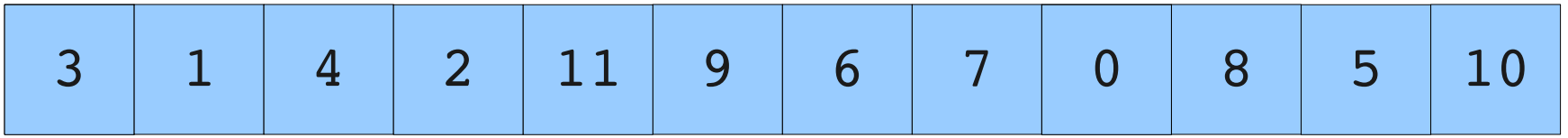
Implicit Leonardo Trees



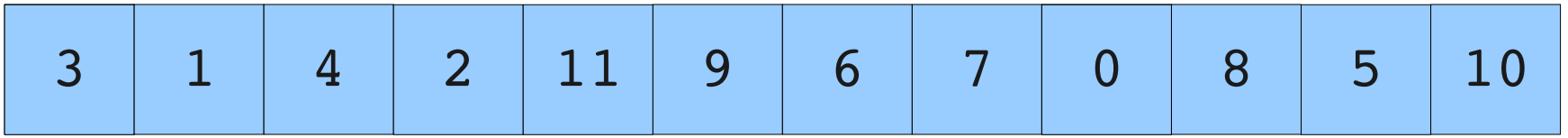
Implicit Leonardo Trees



Reducing Memory Usage

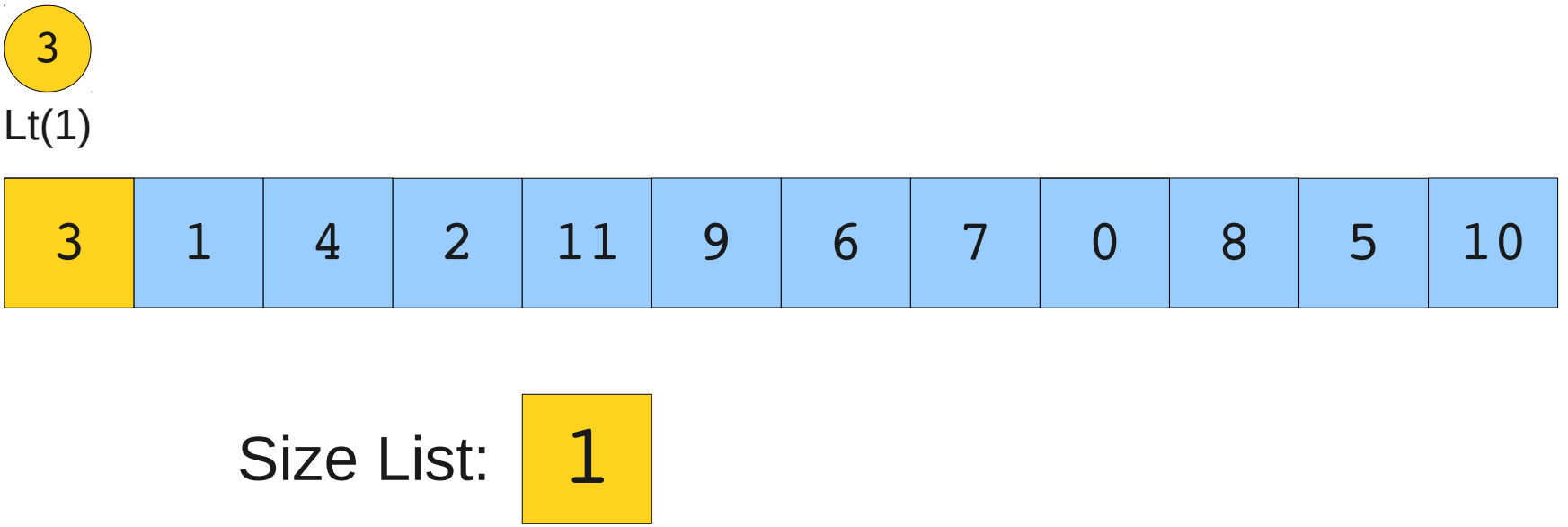


Reducing Memory Usage

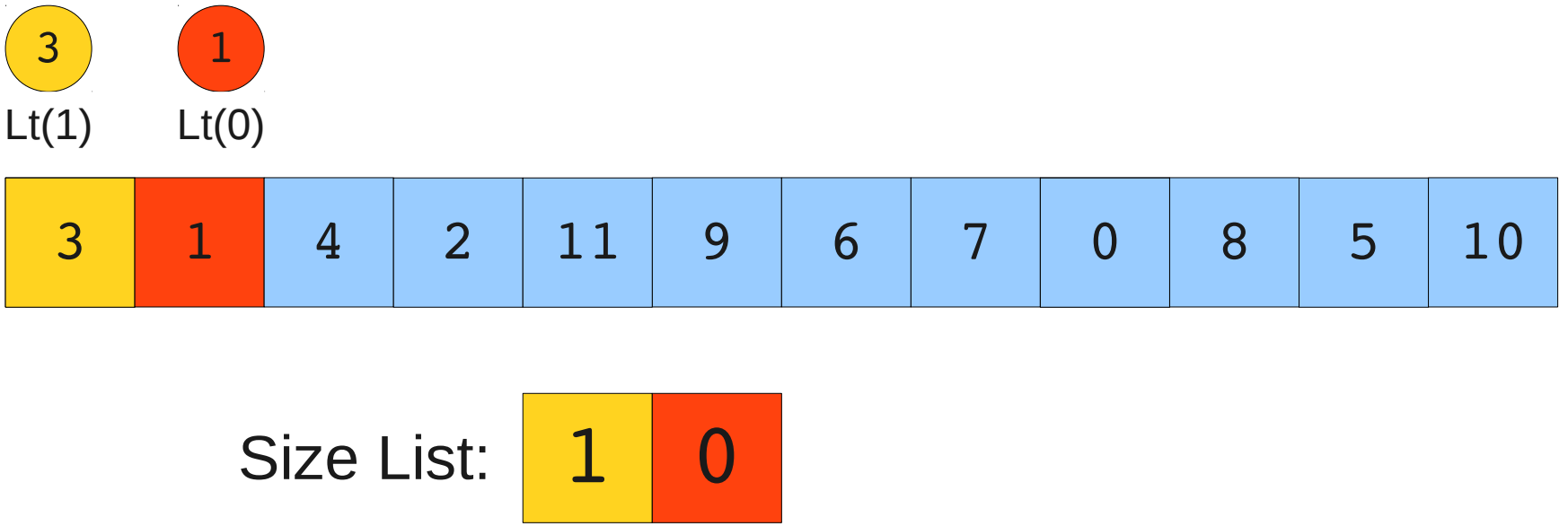


Size List:

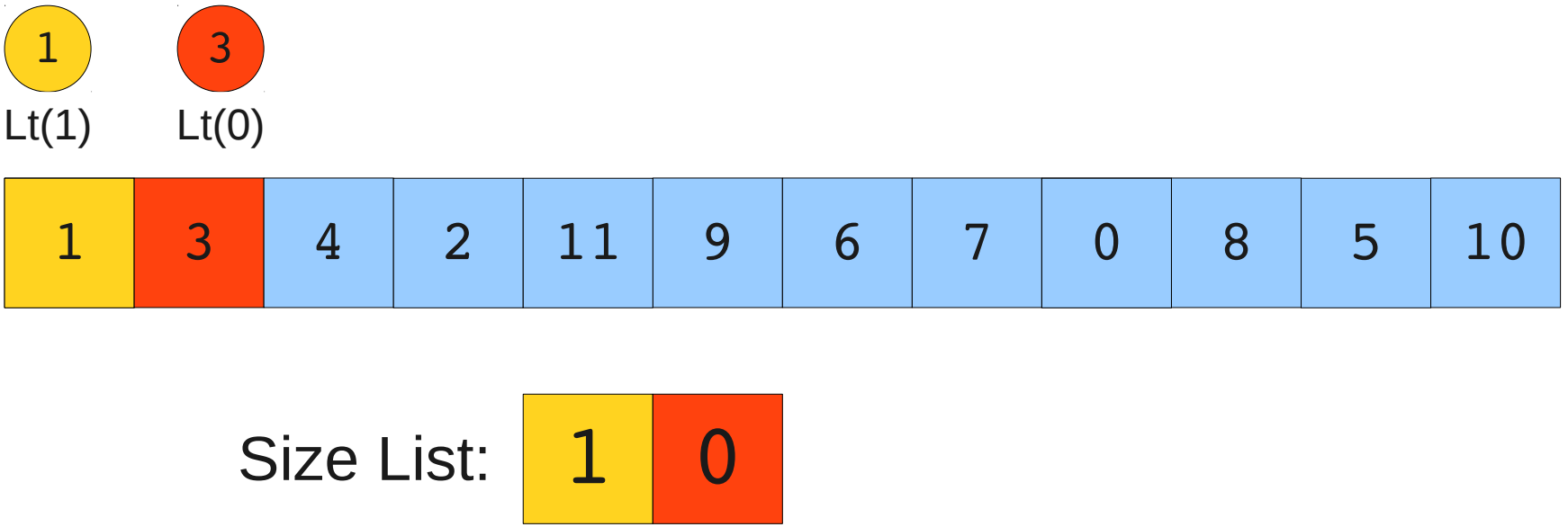
Reducing Memory Usage



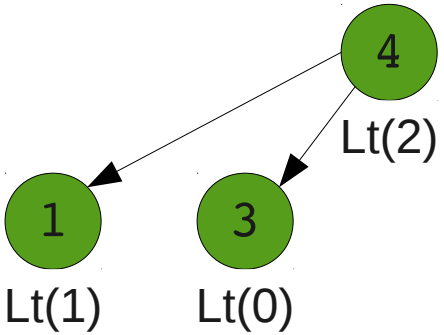
Reducing Memory Usage



Reducing Memory Usage



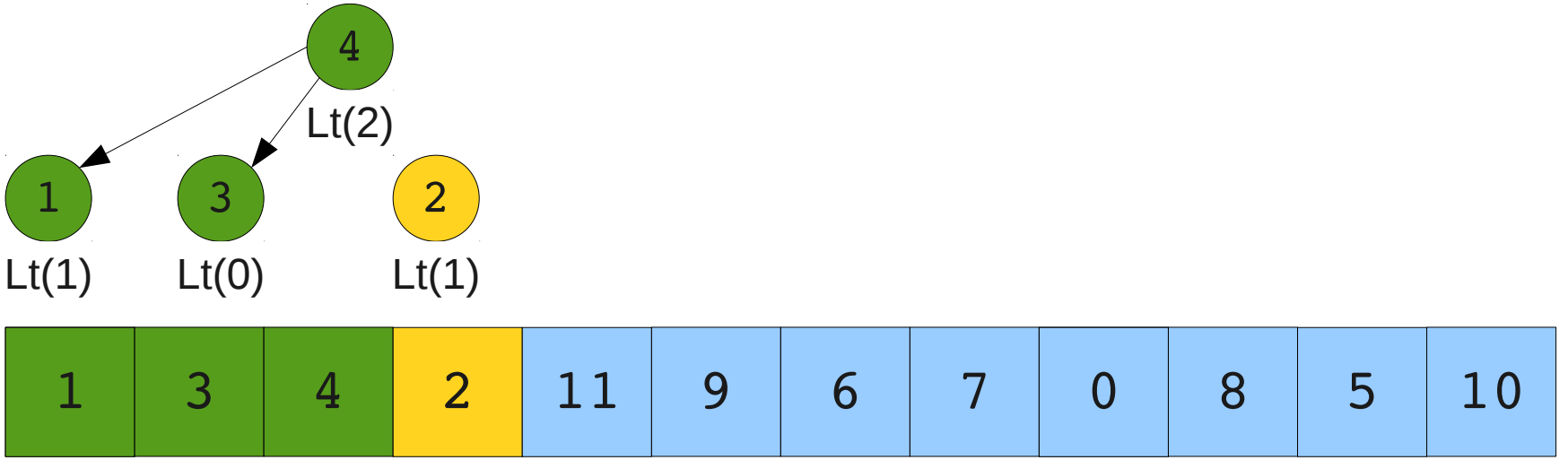
Reducing Memory Usage



1	3	4	2	11	9	6	7	0	8	5	10

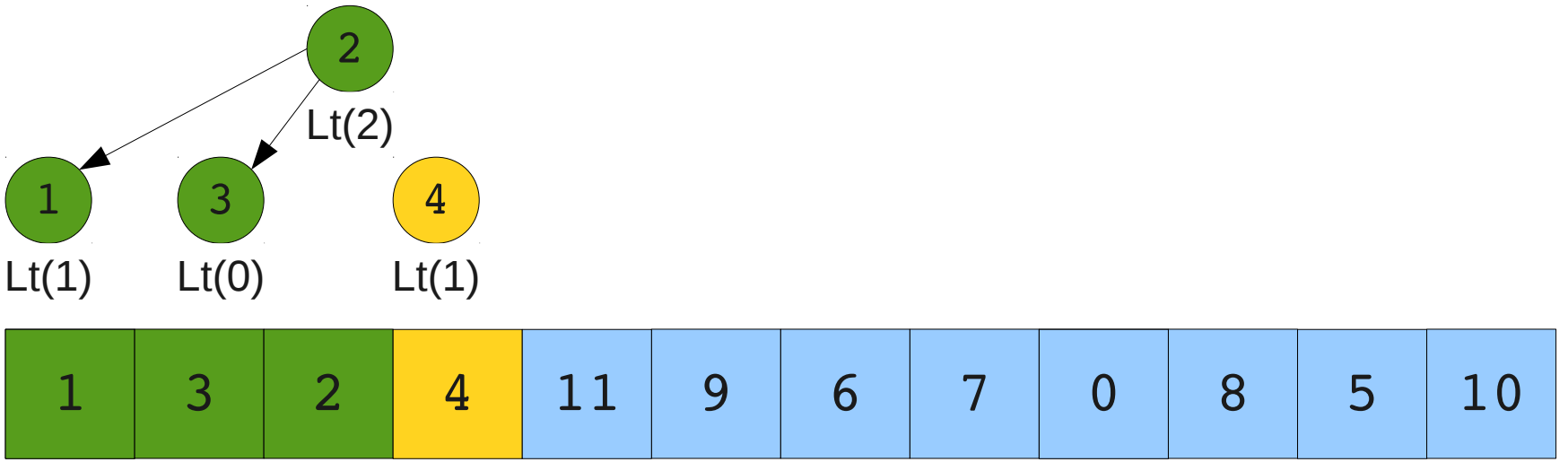
Size List: 2

Reducing Memory Usage



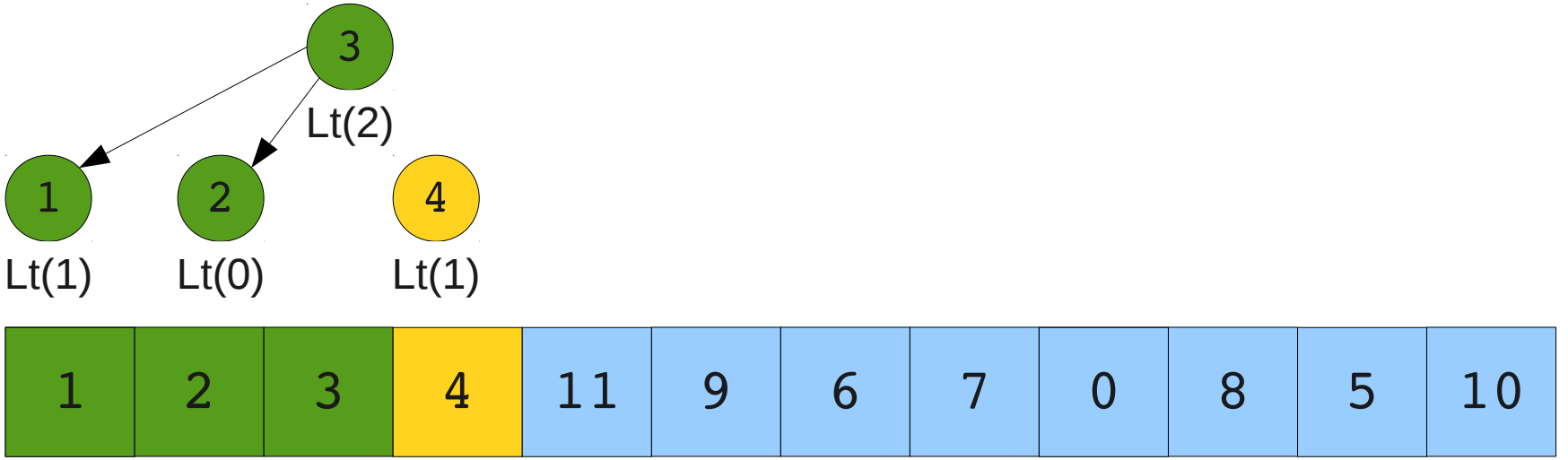
Size List: 2 1

Reducing Memory Usage



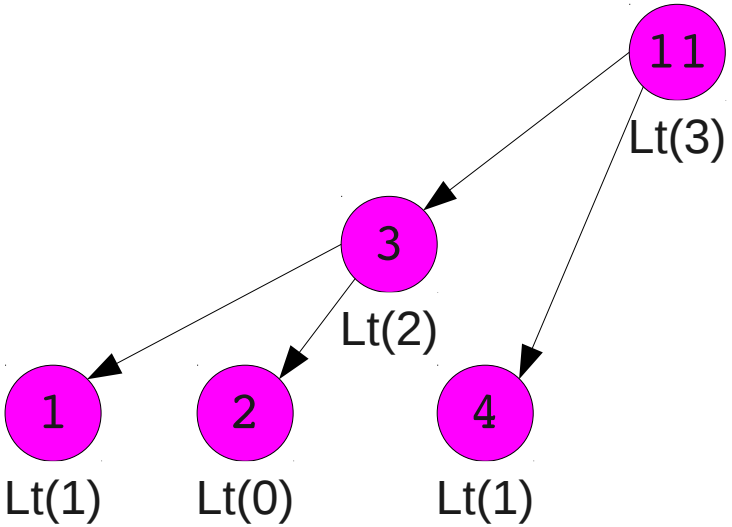
Size List: 2 1

Reducing Memory Usage



Size List: 2 1

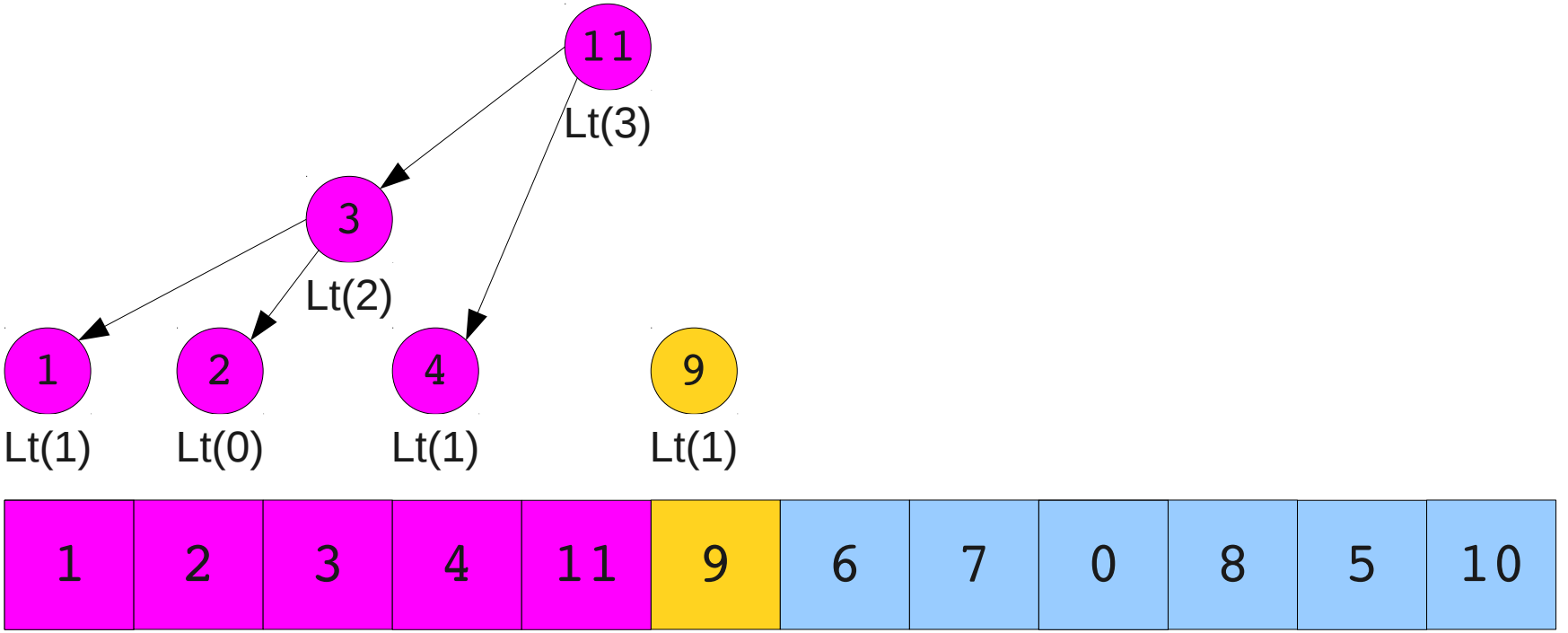
Reducing Memory Usage



1	2	3	4	11	9	6	7	0	8	5	10

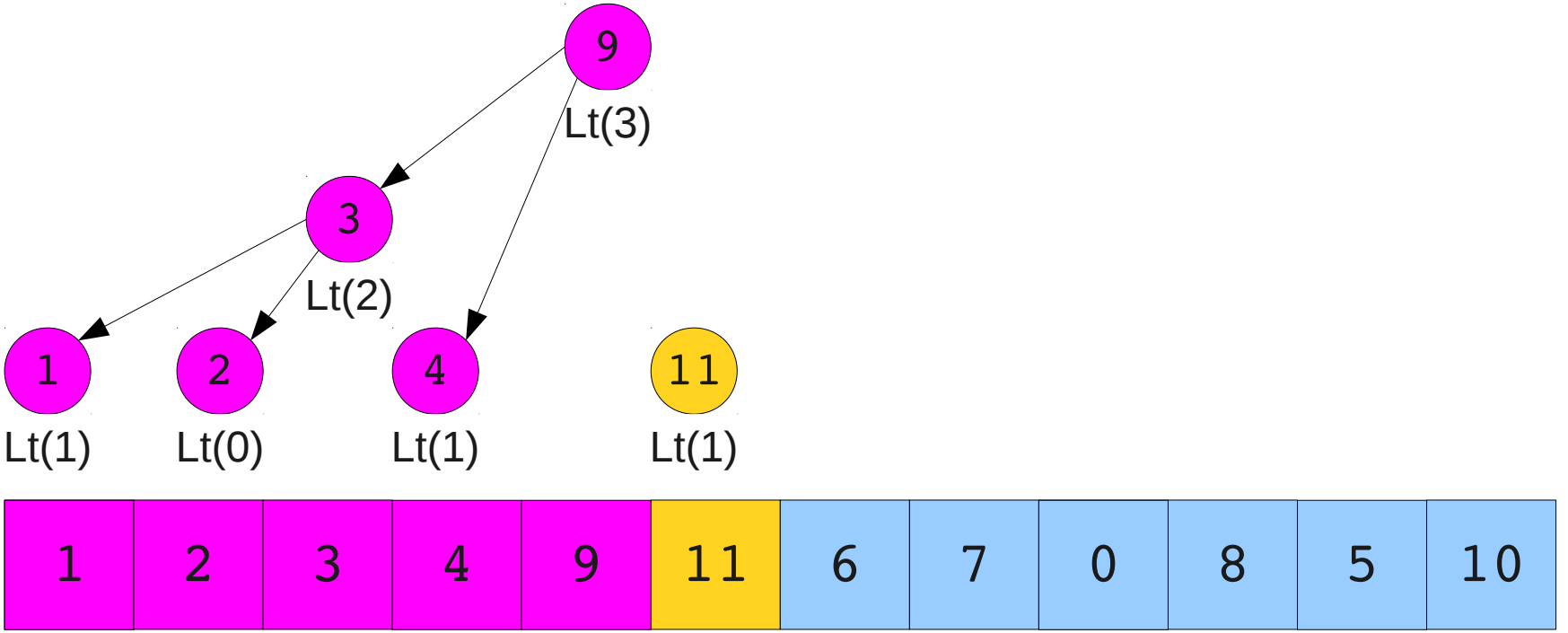
Size List: 3

Reducing Memory Usage



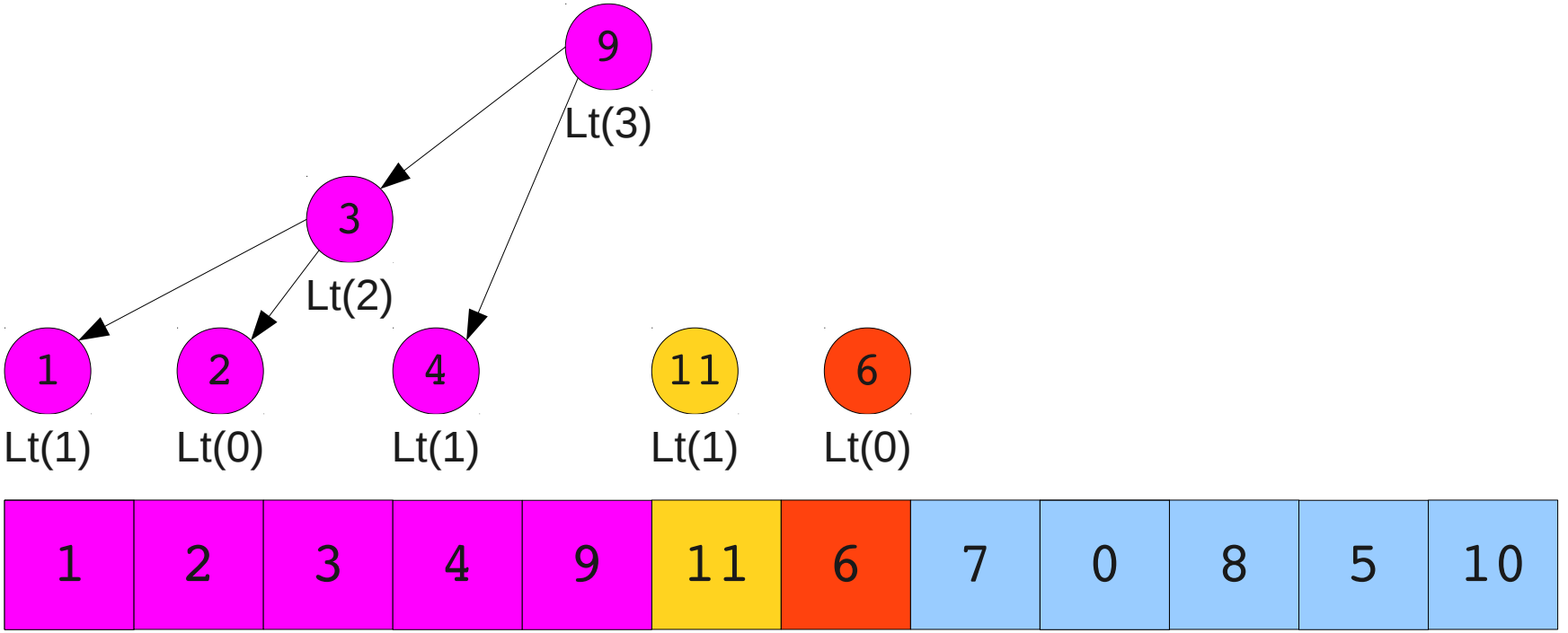
Size List: 3 1

Reducing Memory Usage

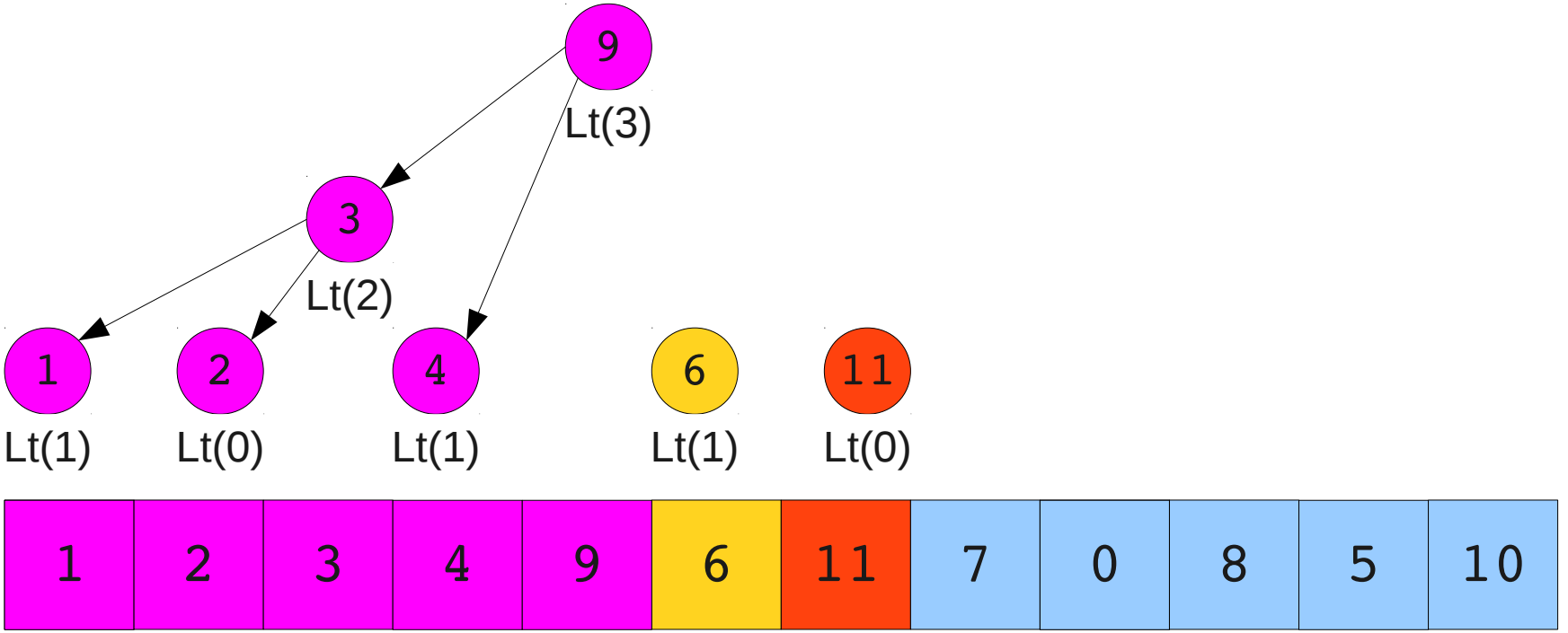


Size List: 3 1

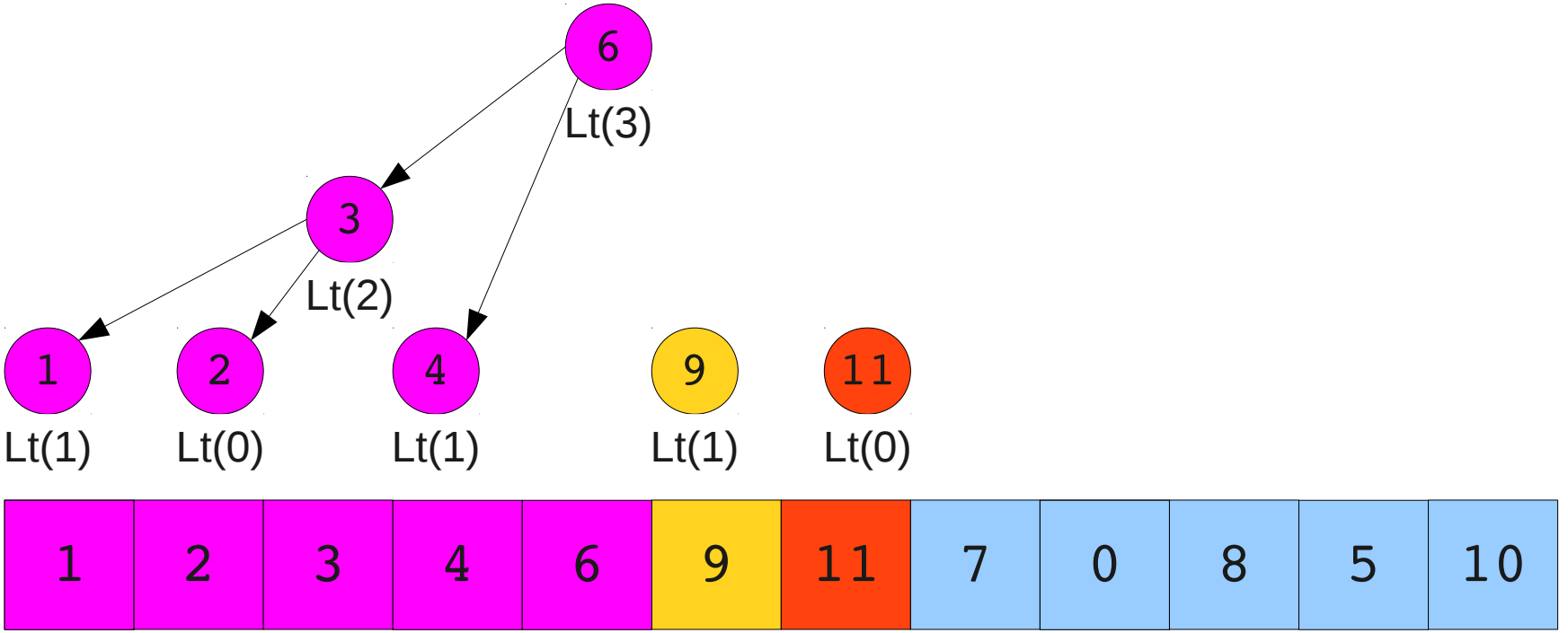
Reducing Memory Usage



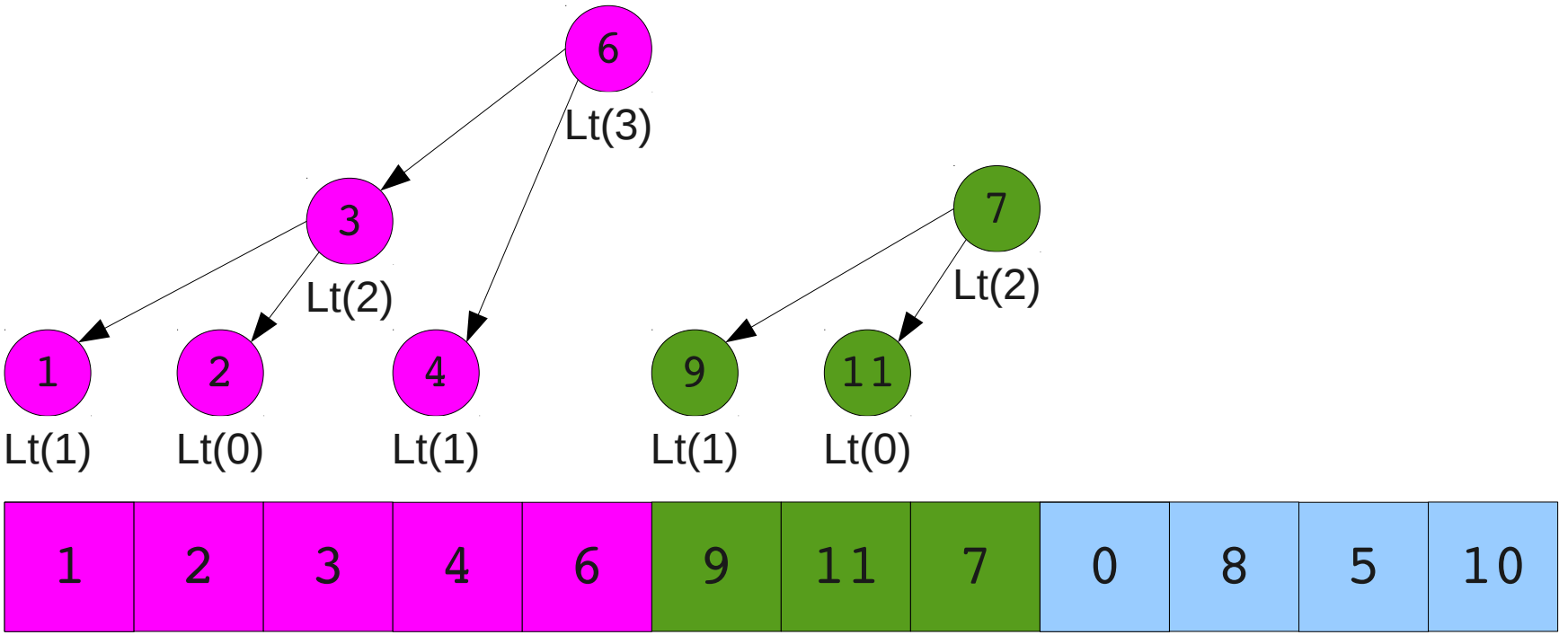
Reducing Memory Usage



Reducing Memory Usage

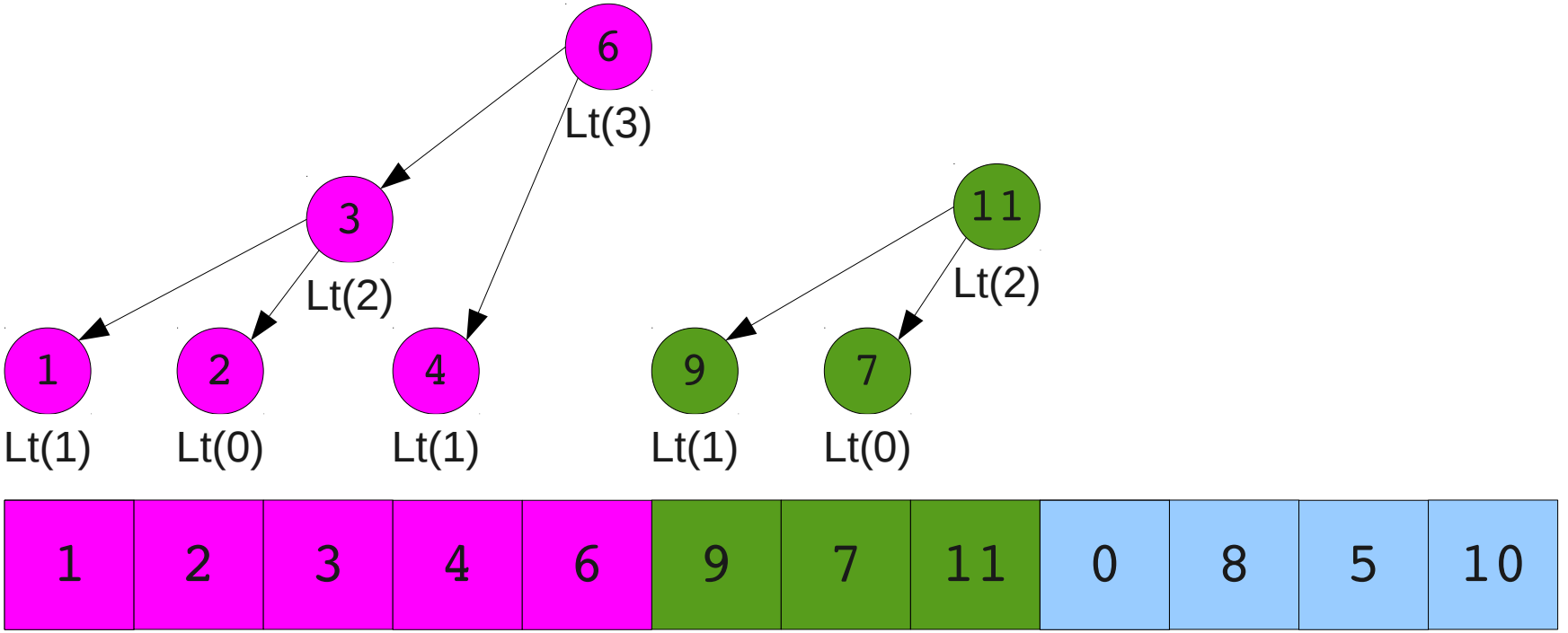


Reducing Memory Usage



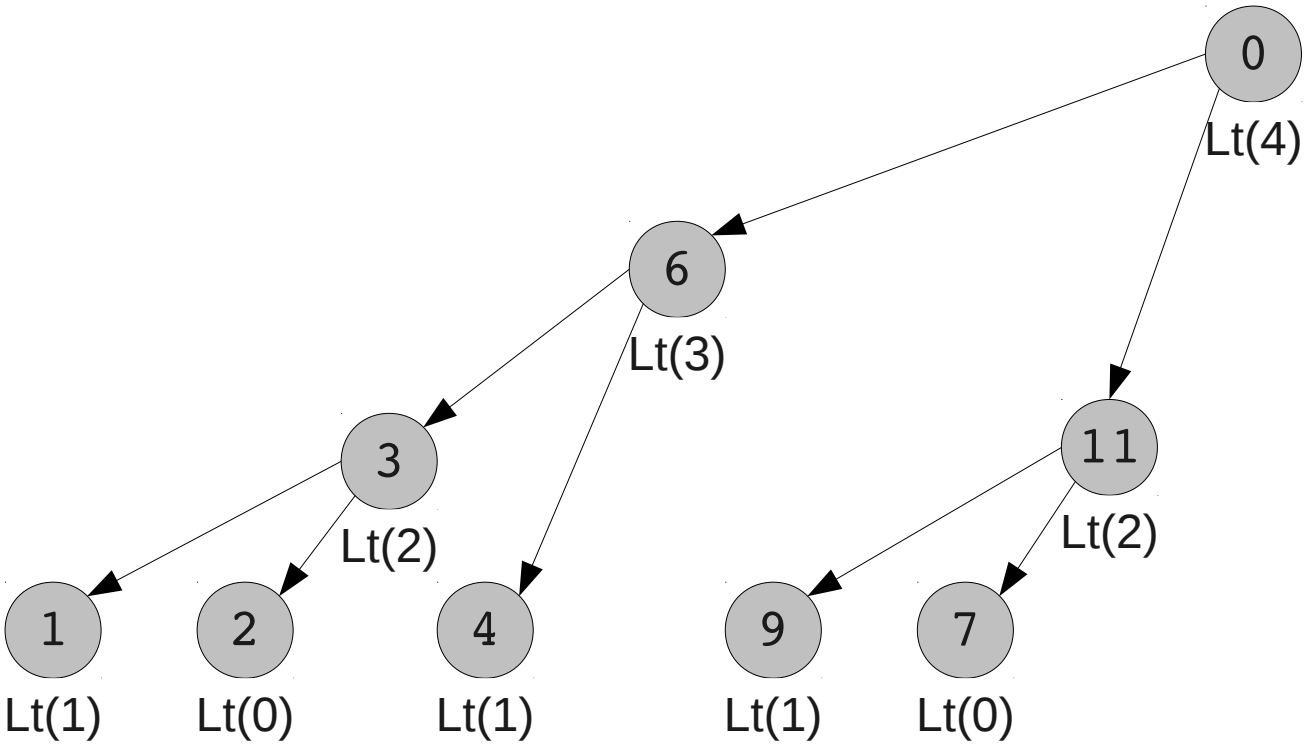
Size List: 3 2

Reducing Memory Usage



Size List: 3 2

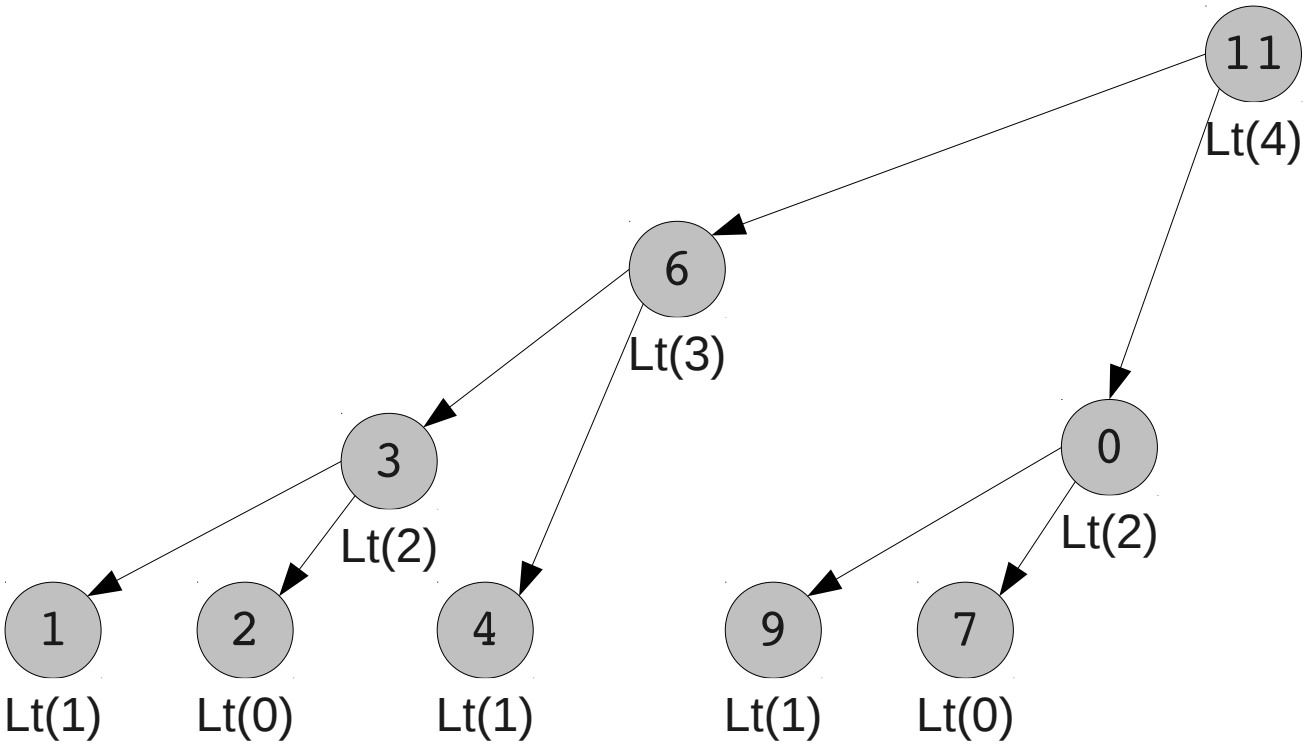
Reducing Memory Usage



1	2	3	4	6	9	7	11	0	8	5	10

Size List: 4

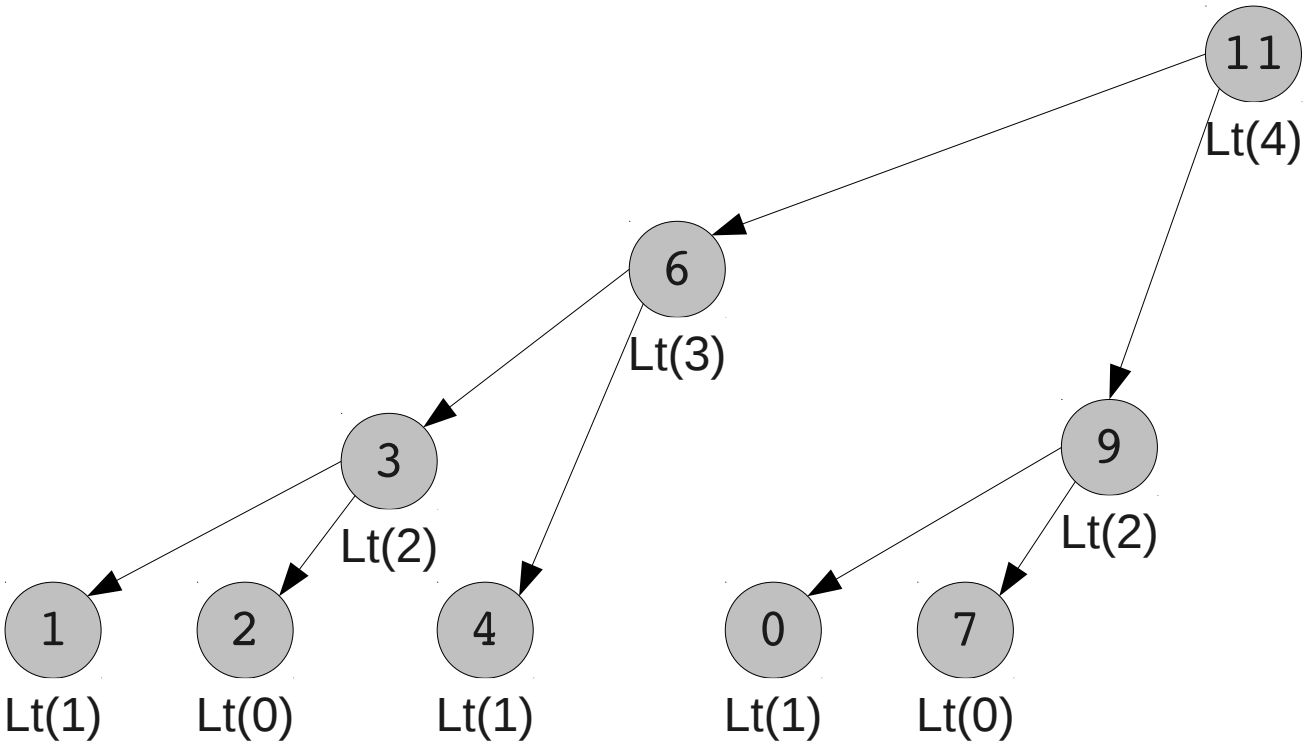
Reducing Memory Usage



1	2	3	4	6	9	7	0	11	8	5	10

Size List: 4

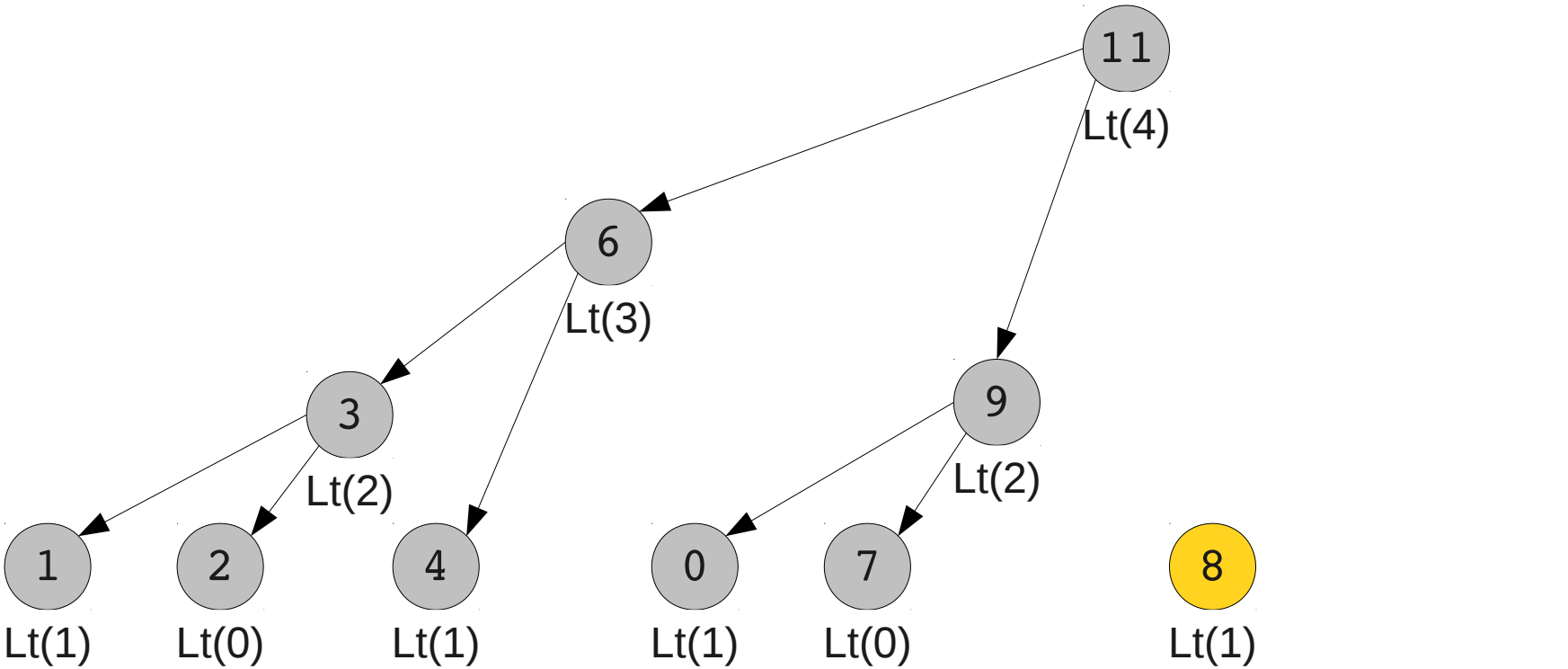
Reducing Memory Usage



1	2	3	4	6	0	7	9	11	8	5	10

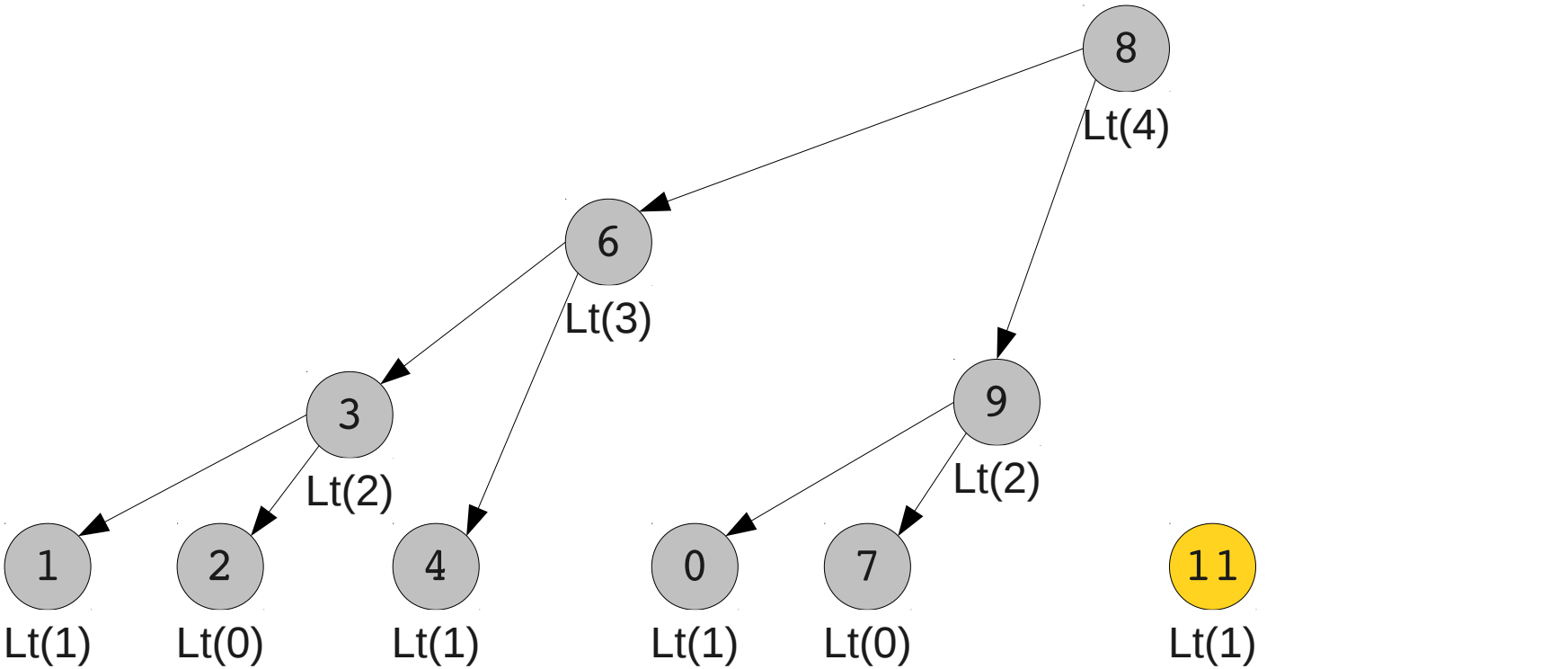
Size List: 4

Reducing Memory Usage



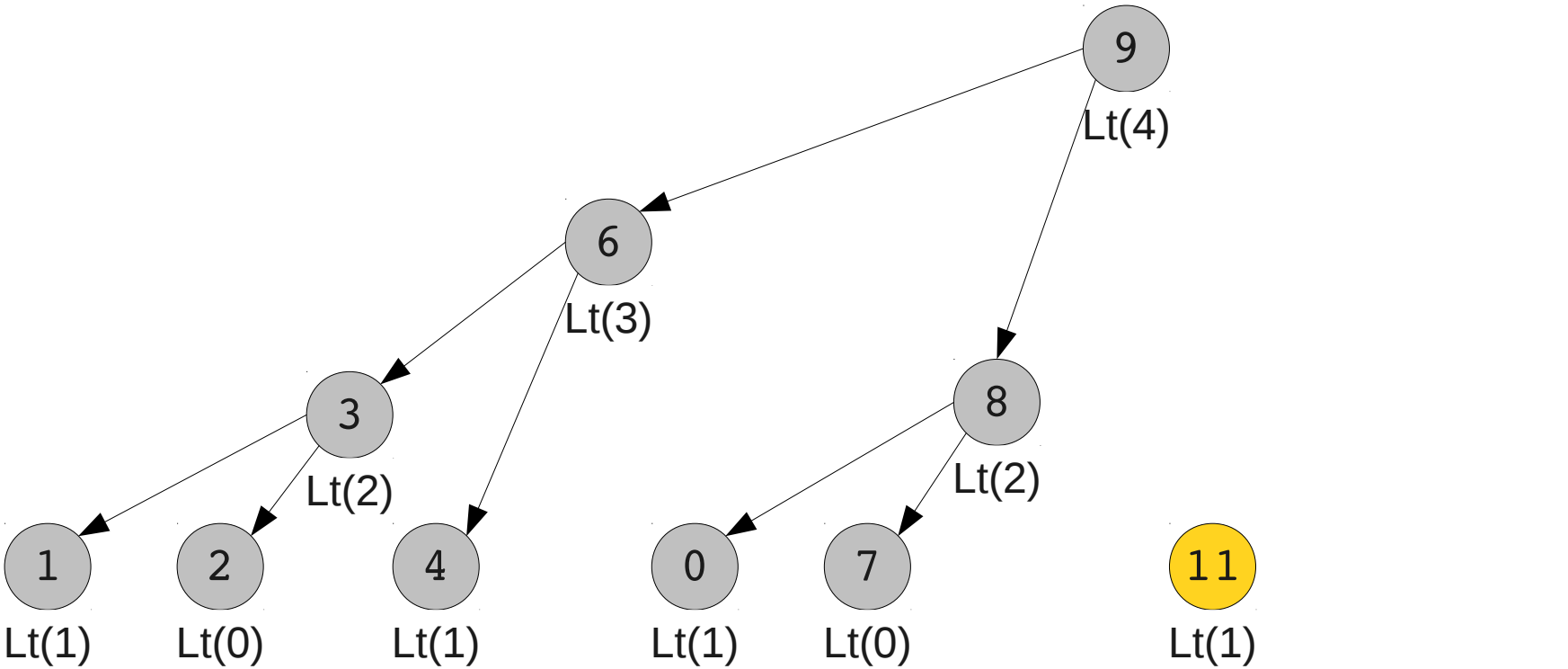
Size List: 4 1

Reducing Memory Usage



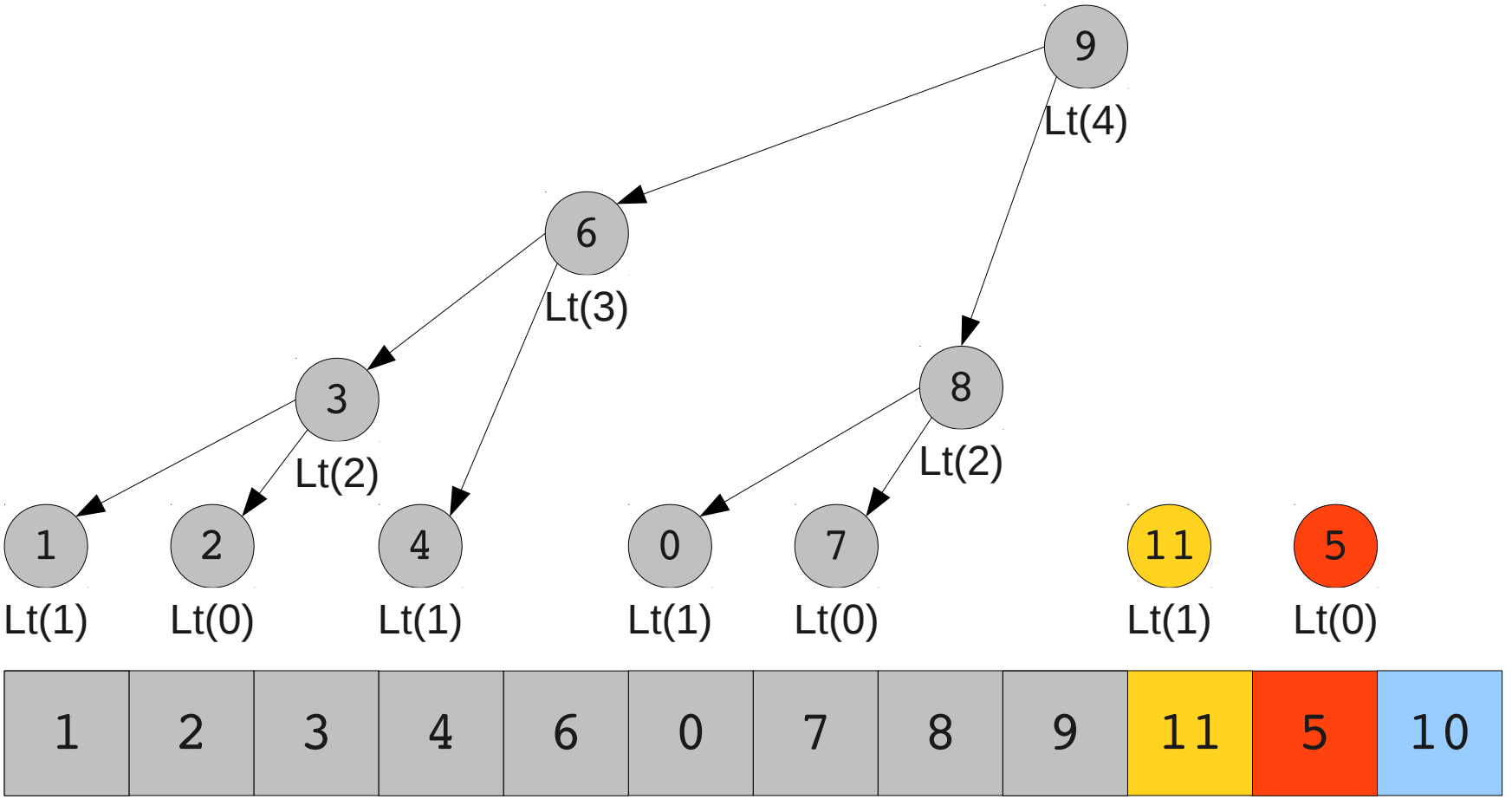
Size List: 4 1

Reducing Memory Usage



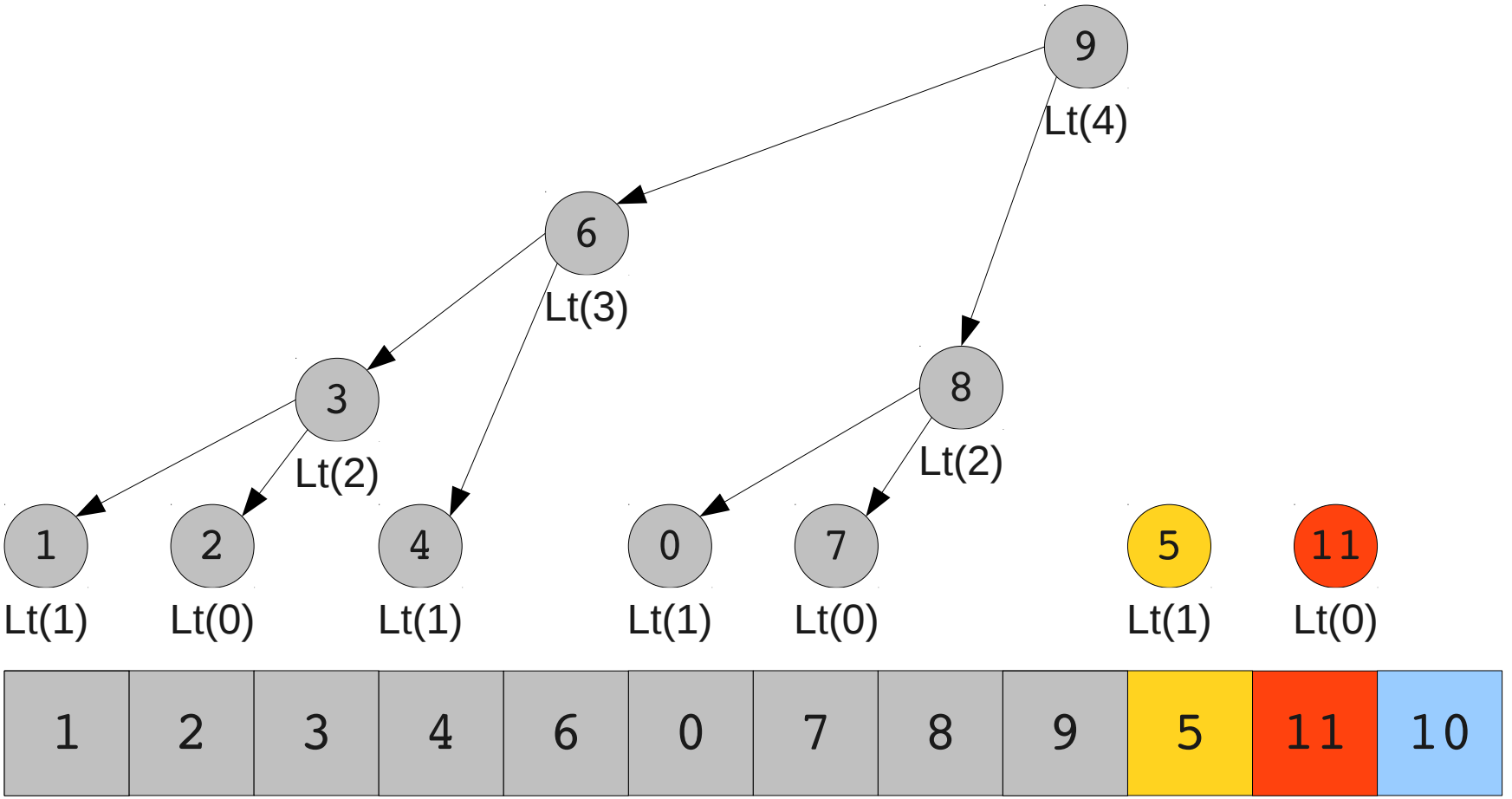
Size List: 4 1

Reducing Memory Usage



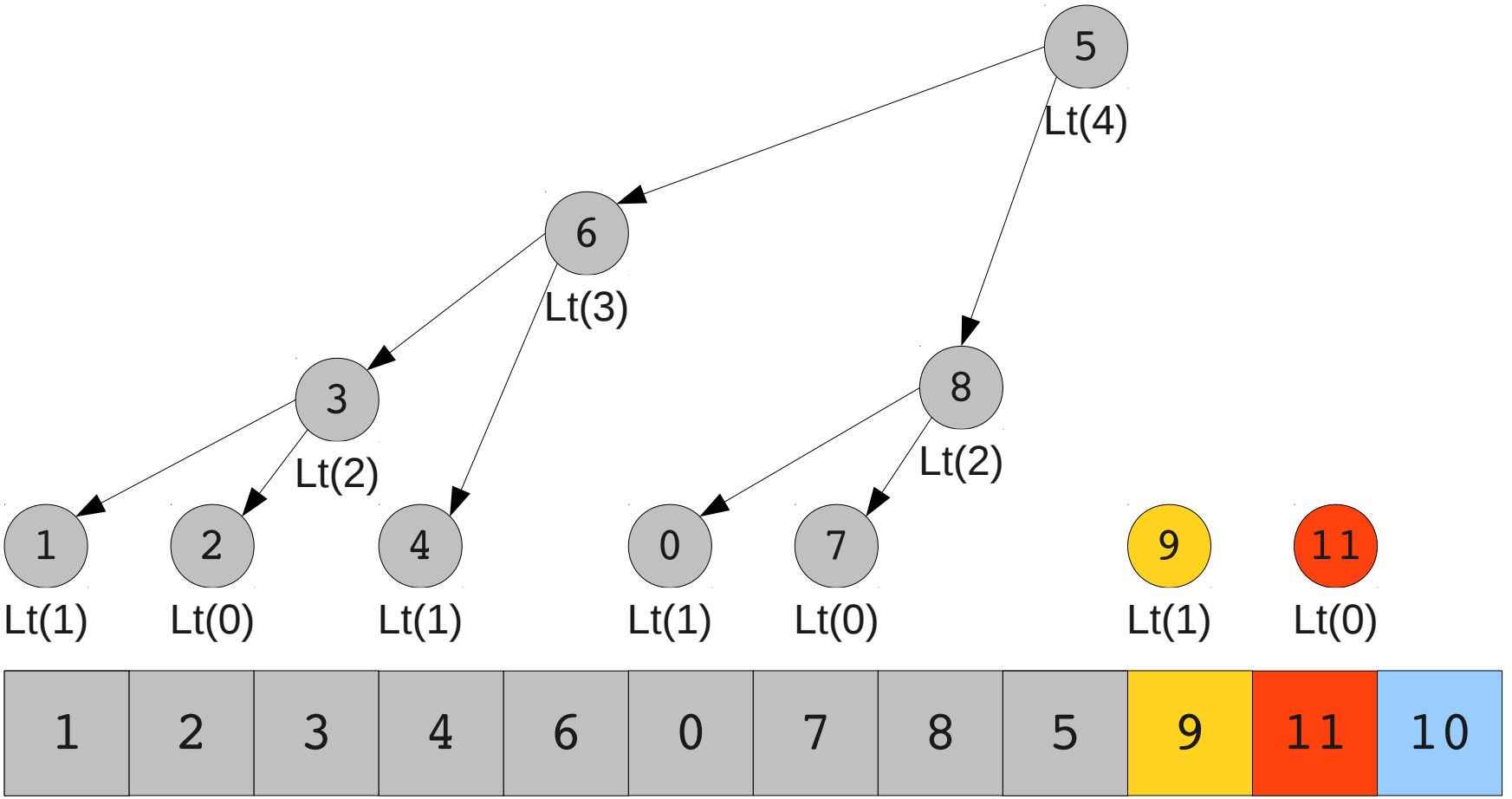
Size List: 4 1 0

Reducing Memory Usage



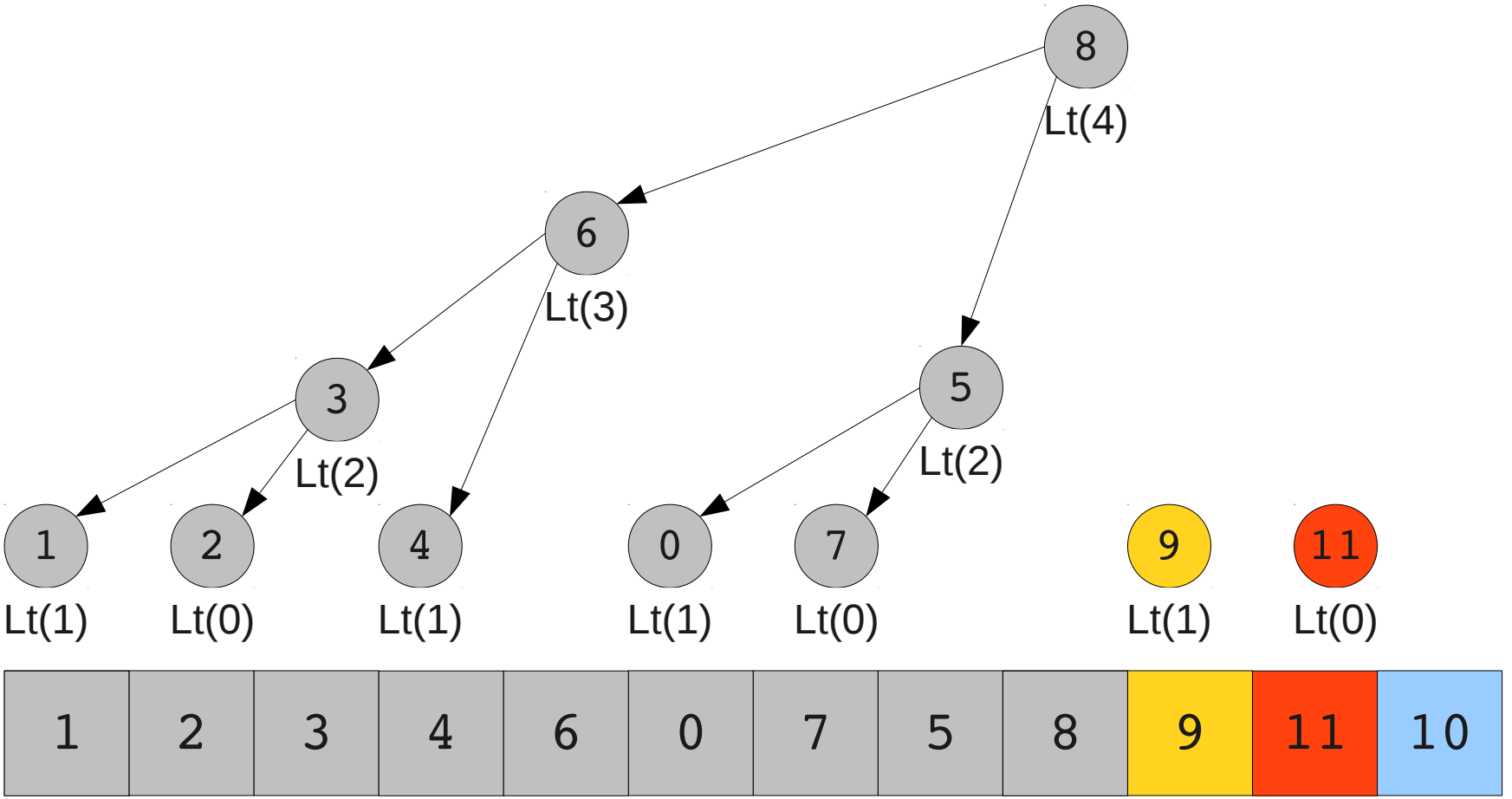
Size List: 4 1 0

Reducing Memory Usage



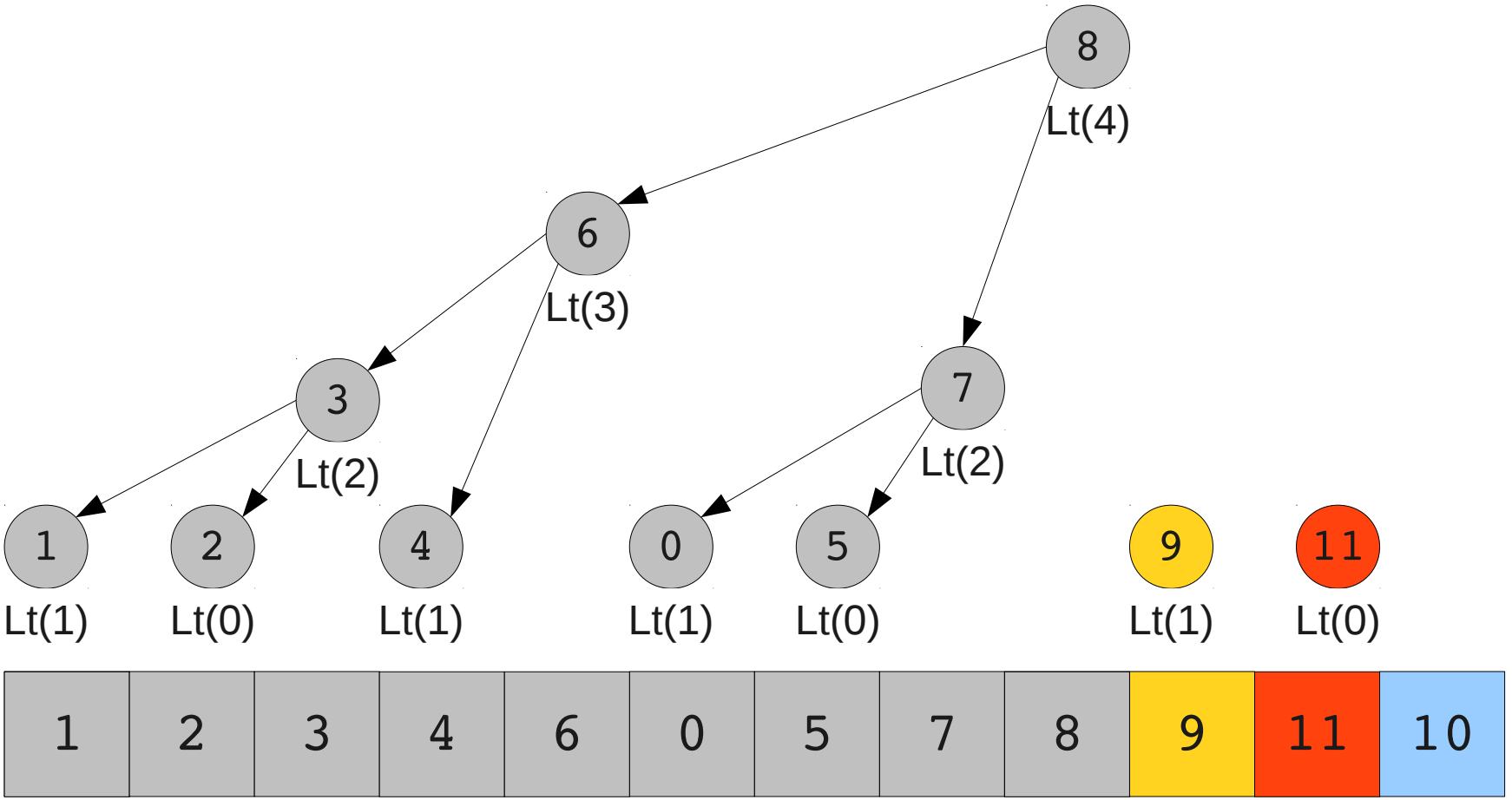
Size List: 4 1 0

Reducing Memory Usage



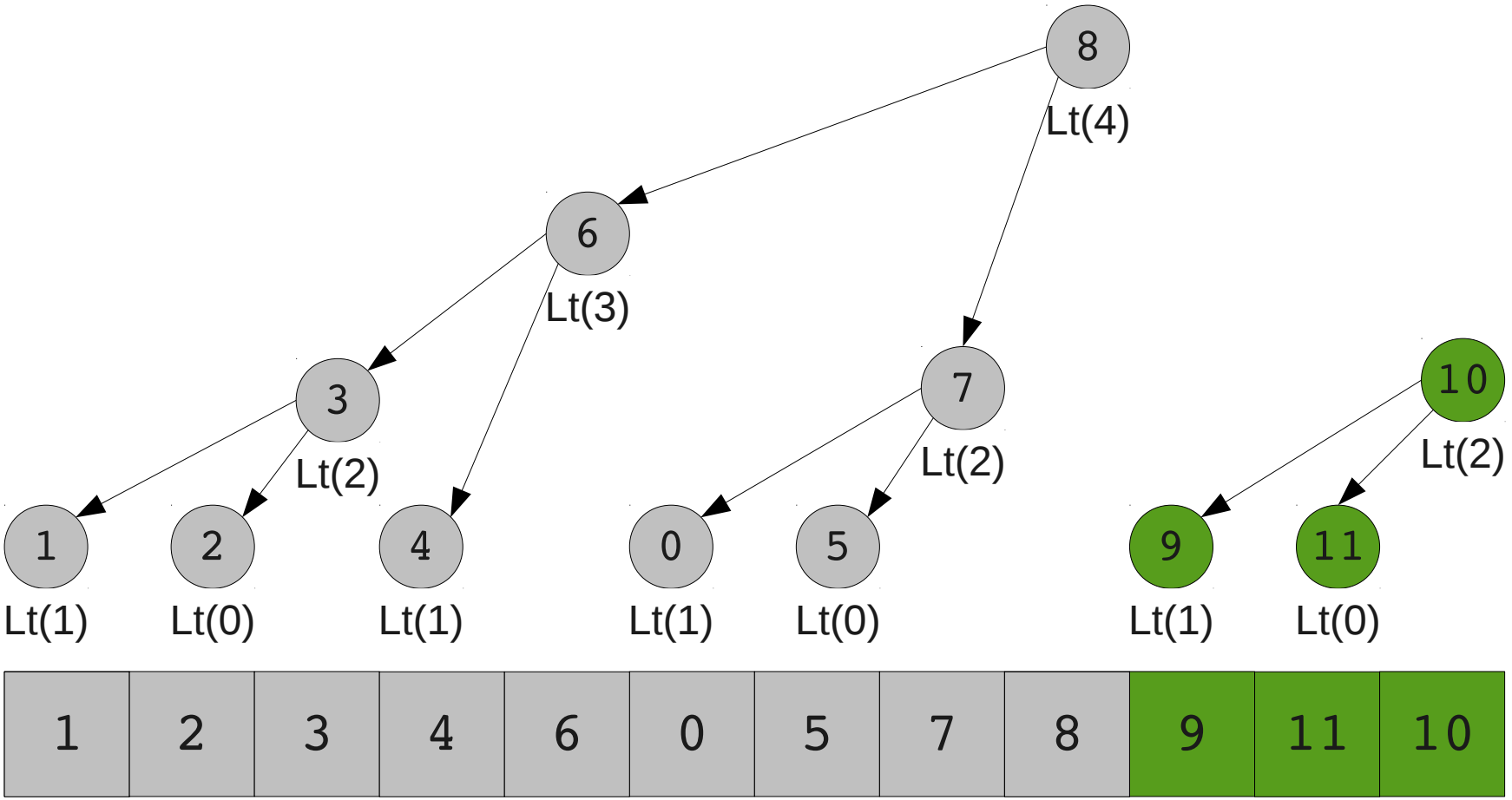
Size List: 4 1 0

Reducing Memory Usage



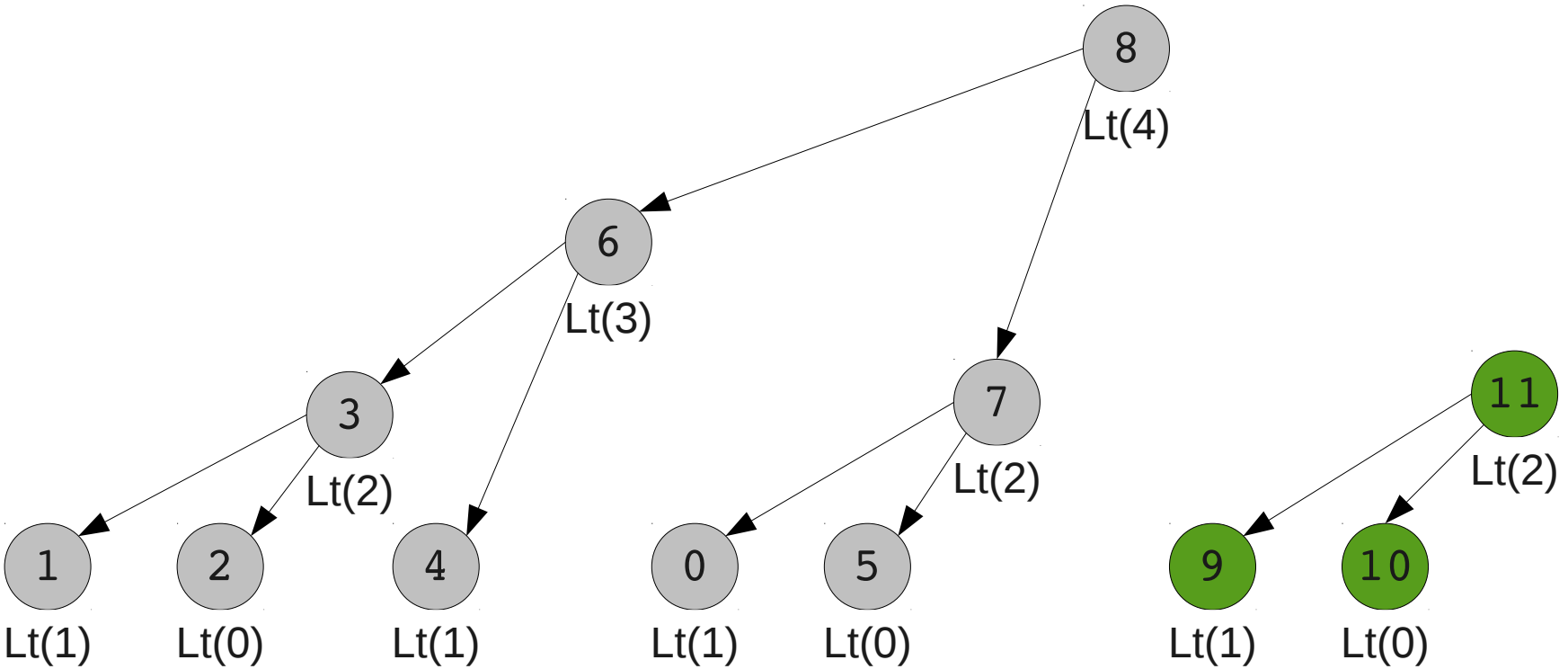
Size List: 4 1 0

Reducing Memory Usage



Size List: 4 2

Reducing Memory Usage



1	2	3	4	6	0	5	7	8	9	10	11

Size List:

4	2

Mostly Implicit Heap

- Keep track of used trees with auxiliary array.
- Store heap structure implicitly in array.
- Extra storage necessary for size list: **O(lg n)**.

Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Binary Heap	Naïve Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Implicit Binary Heap	Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
Leonardo Heap	Very Naïve Smoothsort	$O(n)$	$O(n \lg n)$	$O(n)$
Mostly Implicit Leonardo Heap	Naïve Smoothsort	$O(n)$	$O(n \lg n)$	$O(\lg n)$

From O(lg n) to O(1): A Sketch

- Possible to get O(1) memory usage; not for the faint of heart.
- Key idea: **Replace size list with bitvector.**
- Why is this difficult?
  - Harder to recover tree sizes.
  - Need amortized analysis to prove correctness.
  - Less adaptive.

Why O(1)?

- Each machine word has size **Ω(lg n)**.
  - Otherwise, cannot store pointers.
- Need **O(1)** of these words to hold bitvector.
- Fun term: **transdichotomous machine model**.
- You've already been doing this!

Code for Insertion Sort

```
void InsertionSort(vector<int>& elems) {
    for (int k = 1; k < elems.size(); ++k) {
        for (int j = k – 1; j >= 0; --j) {
            if (elems[j] < elems[k])
                swap(elems[j], elems[k]);
        }
    }
}
```

Code for Insertion Sort

```
void InsertionSort(vector<int>& elems) {
    for (int k = 1; k < elems.size(); ++k) {
        for (int j = k – 1; j >= 0; --j) {
            if (elems[j] < elems[k])
                swap(elems[j], elems[k]);
        }
    }
}
```

Code for Insertion Sort

```
void InsertionSort(vector<int>& elems) {
    for (int k = 1; k < elems.size(); ++k) {
        for (int j = k – 1; j >= 0; --j) {
            if (elems[j] < elems[k])
                swap(elems[j], elems[k]);
        }
    }
}
```

Priority Queue	Sorting Algorithm	Best Runtime	Worst Runtime	Memory
Unsorted Array	Naïve Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Implicit Unsorted Array	Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Implicit Sorted Array	Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Binary Heap	Naïve Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Implicit Binary Heap	Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
Leonardo Heap	Very Naïve Smoothsort	$O(n)$	$O(n \lg n)$	$O(n)$
Mostly Implicit Leonardo Heap	Naïve Smoothsort	$O(n)$	$O(n \lg n)$	$O(\lg n)$
Implicit Leonardo Heap	**Smoothsort**	$O(n)$	$O(n \lg n)$	$O(1)$

Summary

Good **priority queues** make
good **sorting algorithms**.

Implicit representations are better than **explicit representations**.

At the cost of **sanity and sleep**, you can make smoothsort run in O(1) space.

For more on smoothsort, including
an O(1) space implementation, see

http://www.keithschwarz.com/smoothsort/

More to Explore

- **Cartesian Tree Sort**
  - Another adaptive heapsort variant.
  - Runtime takes provably optimal advantage of presortedness, but uses Θ(n) memory.
- **Poplar Sort**
  - Faster version of smoothsort with size lists.
  - Not adaptive (best-case Ω(n lg n)), but much faster than heapsort on sorted data.
- **Thorup's Construction**
  - How to build a fast priority queue from a fast sorting algorithm (the inverse of this construction)
  - I haven't made it through the paper yet; let me know if you do!

Questions?