Programming in Perl

Week Ten

Algorithms and data structuring

Why Are Algorithms Important?

- Algorithms are language independent representations of the steps needed to accomplish a task
- If you have a knowledge of how to accomplish a task before you start working on a program, you can structure your data to make it easier and/or faster to process the data
- We will look at some sorting functions to see how this is accomplished

Searching

- Sorting is one way to structure your data to make it easier to process
- Sorted data is faster to search
- Example:
  - Suppose you we have an array of student ID's and we need to know if the ID 42 has been used
  - We want to add the ID if it is not in the array
  - (No fair using a hash, let's pretend that Perl doesn't have hashes to see how we can do it!)

Searching

- Using `grep`:

```
# assume @student_id is already defined
my $id = 42;
unless(grep {$_== $id} @student_id) {
    push @student_id, $id;
}
```

- Problem: It looks at every ID, even if it found it early in the search
- This algorithm always takes `N` tests for an array of length `N`

Searching

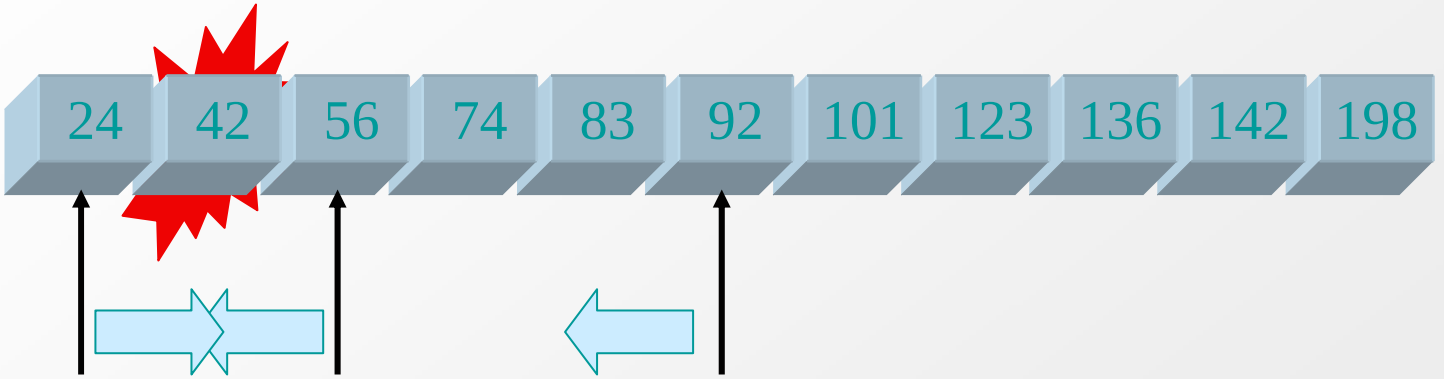
- Refine it a little!

```
my $id = 42;
my $found = 0;
foreach $item (@student_id) {
    if ($item == $id) {
        $found =1;
        last;
    }
}
push @student_id unless $found;
```

- On the average, this routine takes `N/2` test to find if the ID is in the array
- Both of the routines are linear searches

Searching

- If the data were sorted into ascending order, we can speed up the tests



This is called a “Binary Sort”

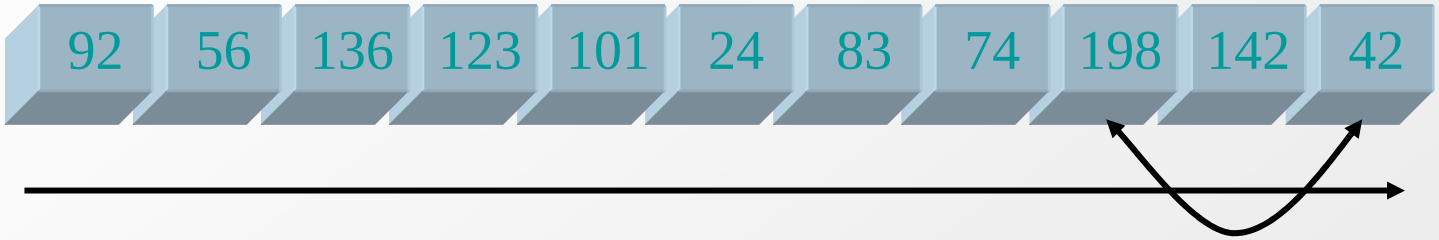
It only works on sorted data!

Sorting

- There are sorting routines from simple to complex
  - What you use depends on the task: each routine has a set of properties that you must match to the task
  - Some are fast, but you must resort every time you add data
  - Others maintain the sort as you insert data, but are slow
  - Still others are a compromise between the extremes
- The type of data you have can also effect the algorithm you pick
  - Some work better on lots of data but poorly on small
  - Some don’t preserve the order of identical keys: The “Stability”
  - Some may not work well with nearly sorted data: The “Sensitivity”

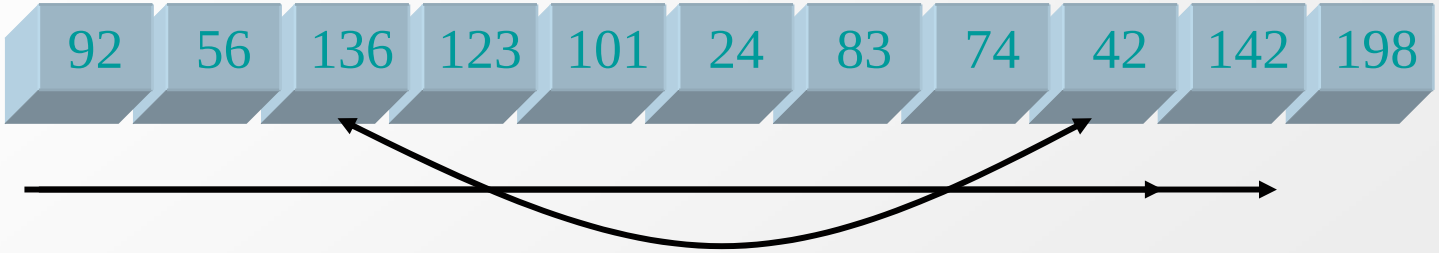
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



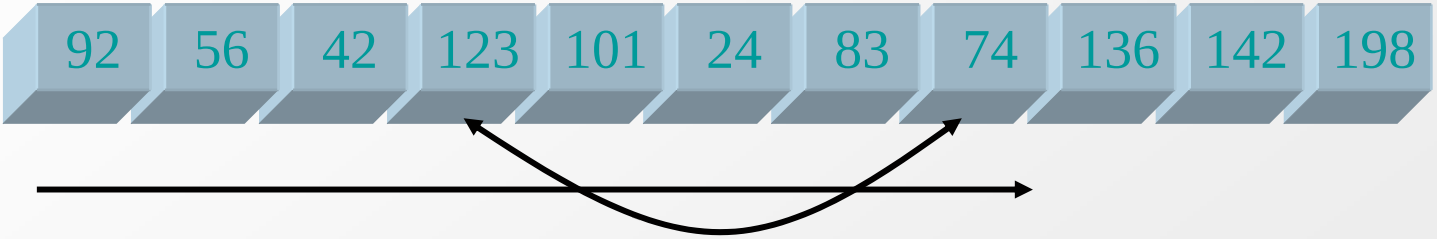
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



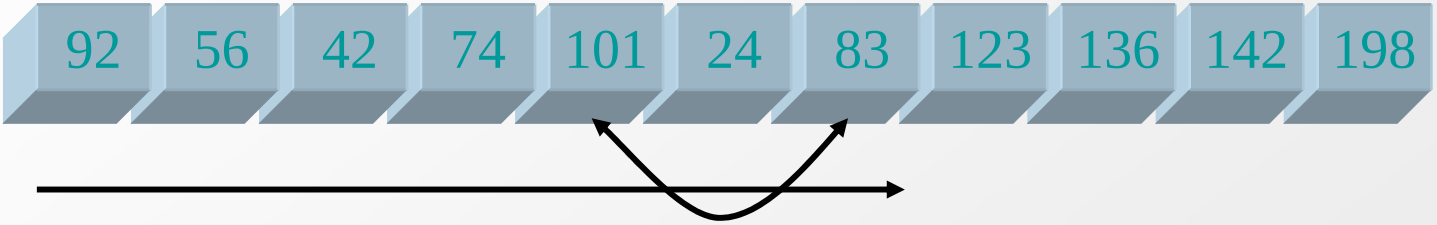
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



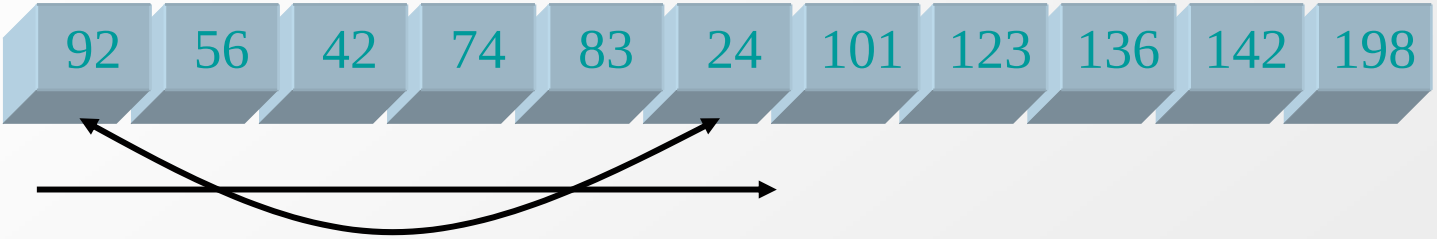
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



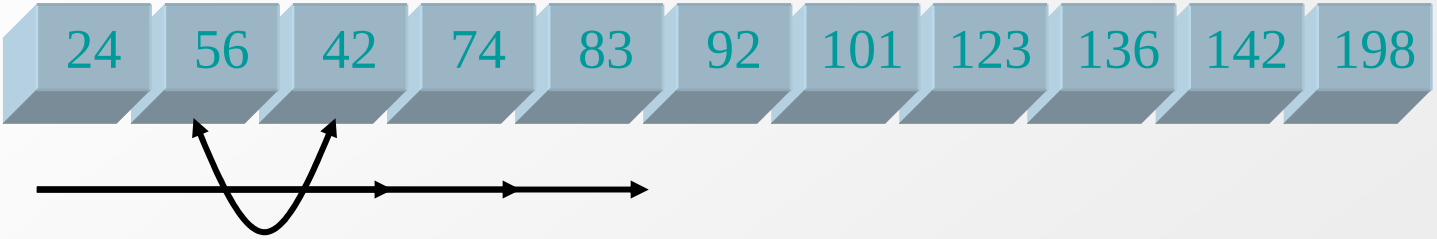
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



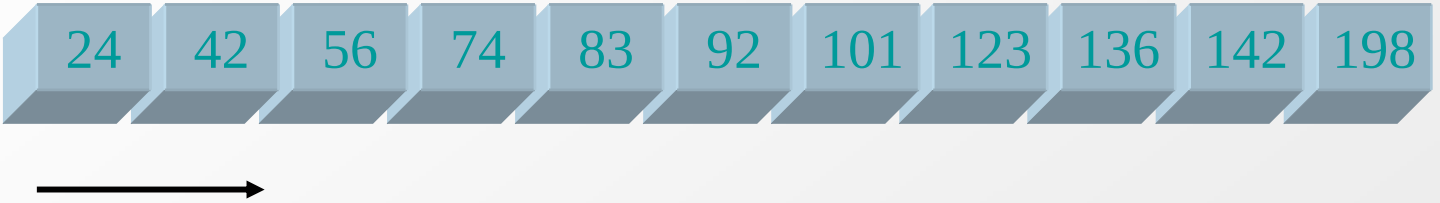
Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



Selection Sort

- One simple sort is the “selection sort”
  - Find the largest (or smallest) value and put it into the proper place. Lather. Rinse. Repeat.



Selection sort is slow! (N^2)

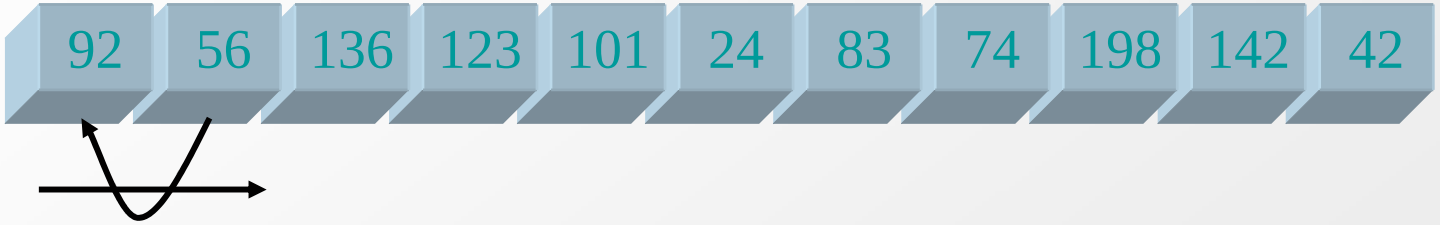
But is stable and insensitive

Selection Sort

```
sub selection_sort {
    return unless @_;
    my $in    = \@_;    # take reference to the input array
    my $upper = $#_;    # length of input array
    for (my $i = $upper; $i; $i--) {
        my $max = $i;
        for (my $y = 0; $j < $I; $j++) {
            $max = $j if $in->[$j] > $in->[$max];
        }
        ($in->[$i], $in->[$max]) = ($in->[$max], $in->[$i]);
    }
}
```

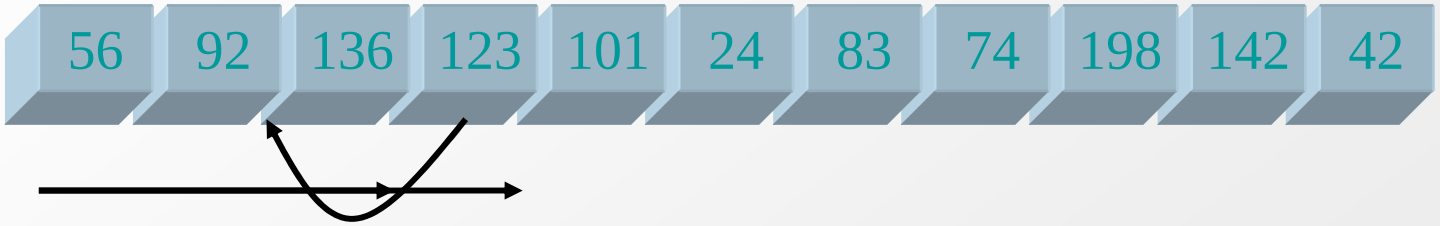
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



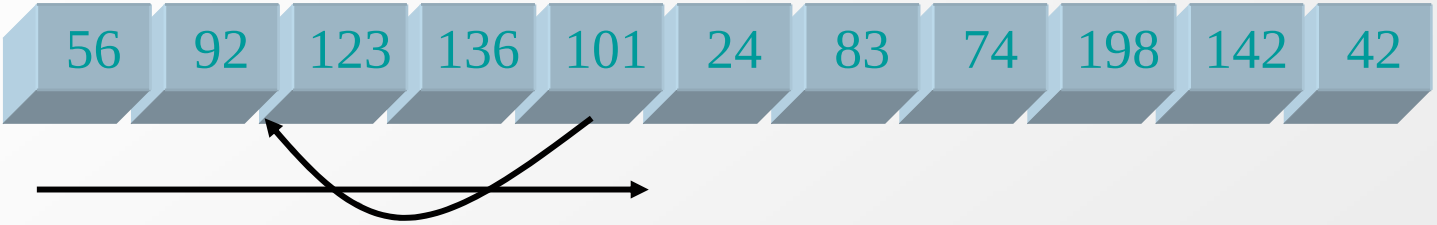
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



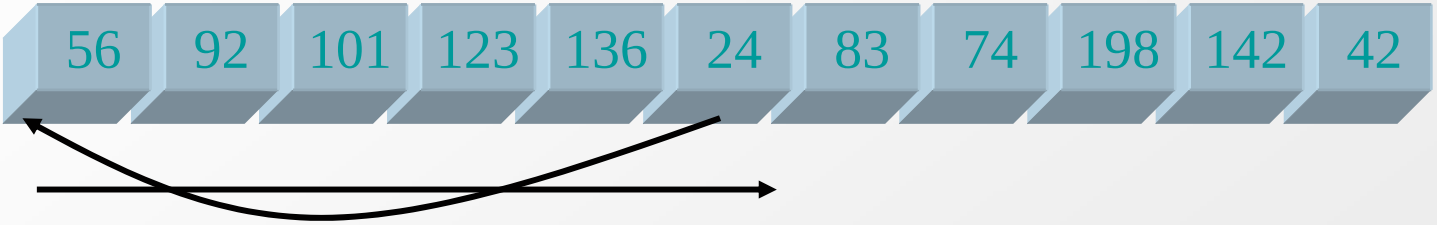
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



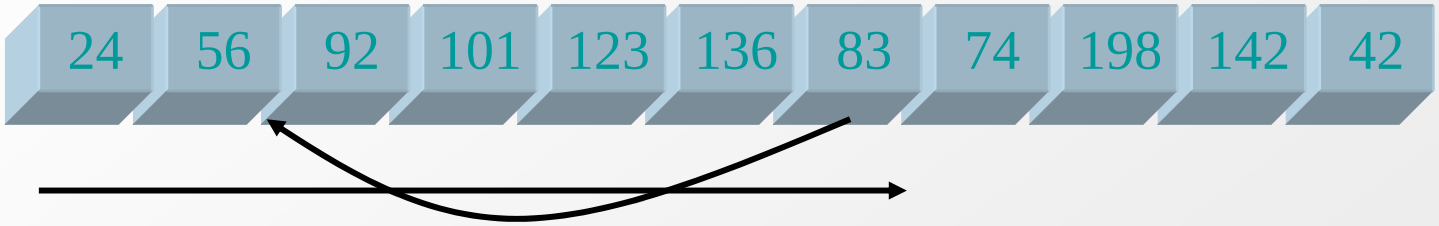
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



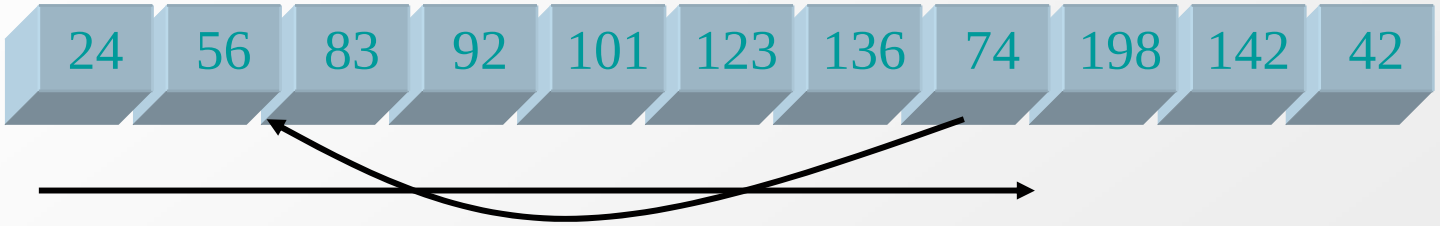
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



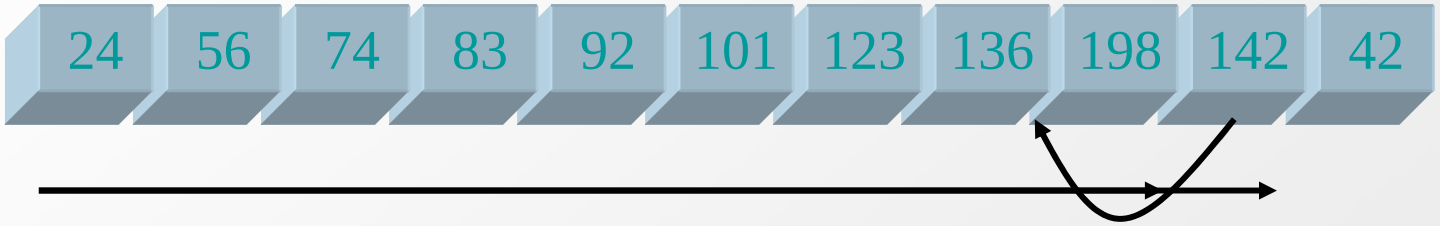
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



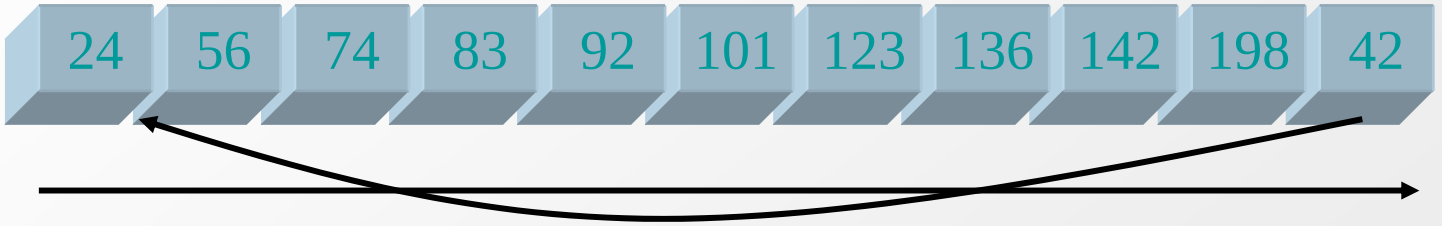
Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place



Insertion Sort

- The insertion sort scans through the array, inserting the smallest value in the proper place

The insertion sort also works in N^2 worst case but works rather well if the data is almost sorted

Insertion Sort

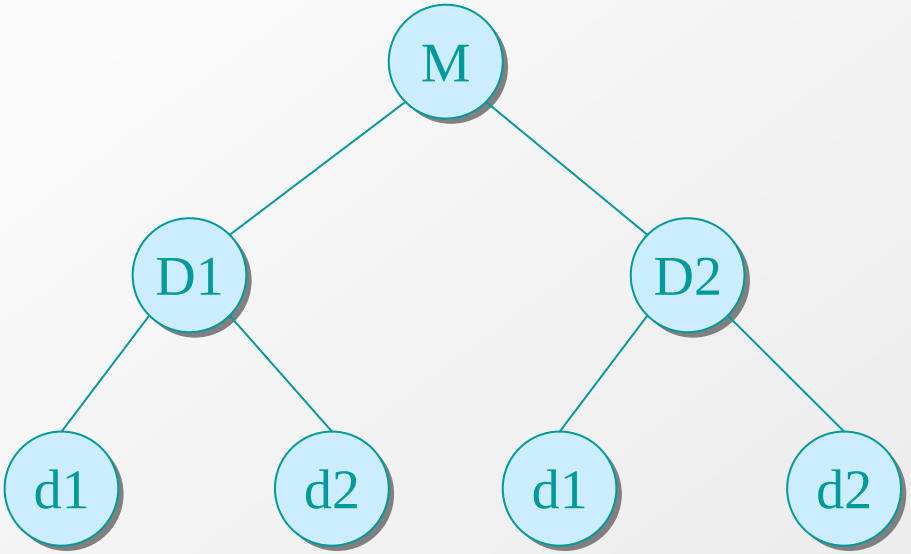
```
sub insertion_sort {
    return unless @_;
    my $in = \@_;
    for (my $j = 1; $j < @_; $j++) {
        my $val = $in->[$j];
        my $i = ($j - 1);
        while ($i >= 0 and $in->[$i] > $val) {
            $in->[$i+1] = $in->[$i];
            $i = $i -1;
        }
        $in->[$i+1] = $val;
    }
}
```

Heap Sort

- The previous sorts both had the problem of taking N^2 time to sort
  - The each added element in the array, the time to sort goes up exponentially
  - Each also required you to make extensive changes to the array, moving data around in the array, sometimes several times
- By changing the way the data is stored, i.e. in an array, we can speed up the sorting

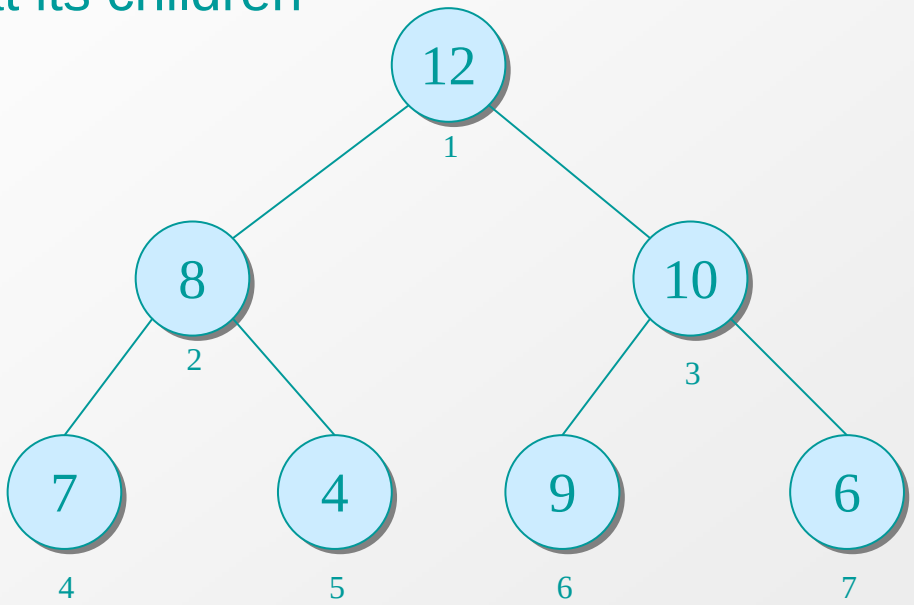
Heap Sort

- A heap is a binary tree structure



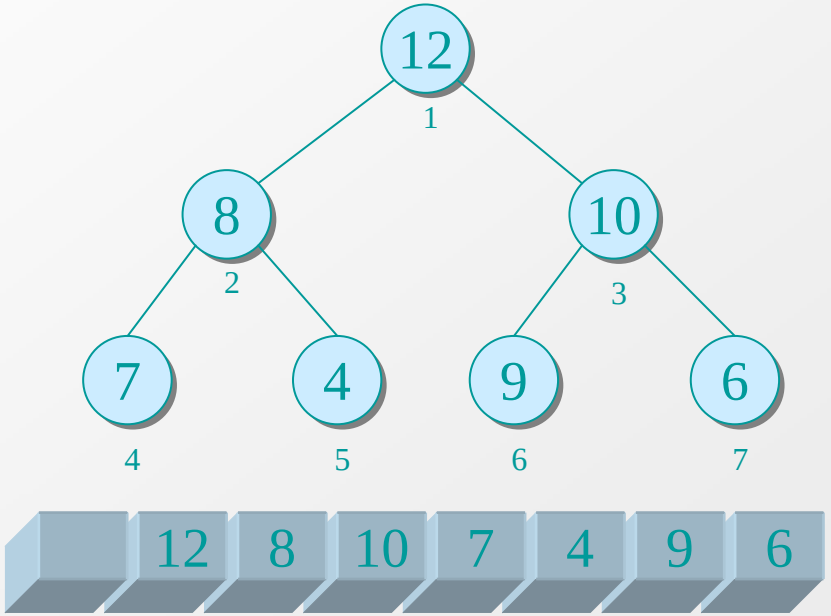
Heap Sort

- A heap has the property that any parent node must be greater that its children



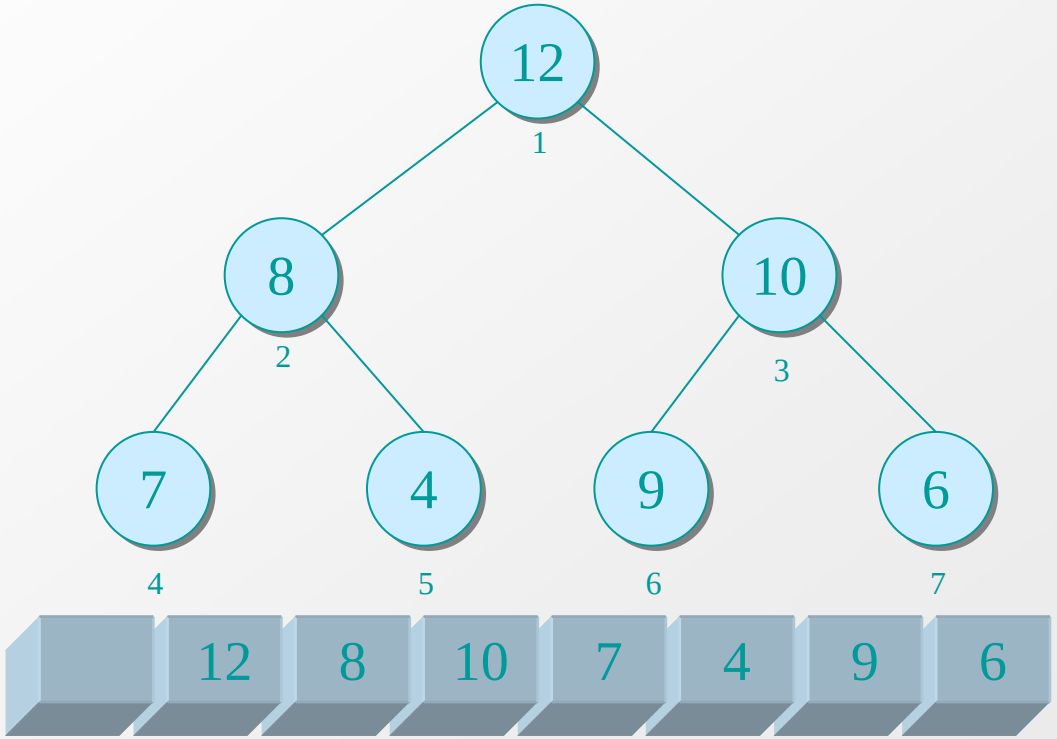
Heap Sort

- We can move through this array by using the properties
  - The Parent of `node(i) = node( int( i / 2 ))`
  - Left Child of `node(i) = node( i * 2 )`
  - Right Child of `node(i) = node( i * 2 + 1)`



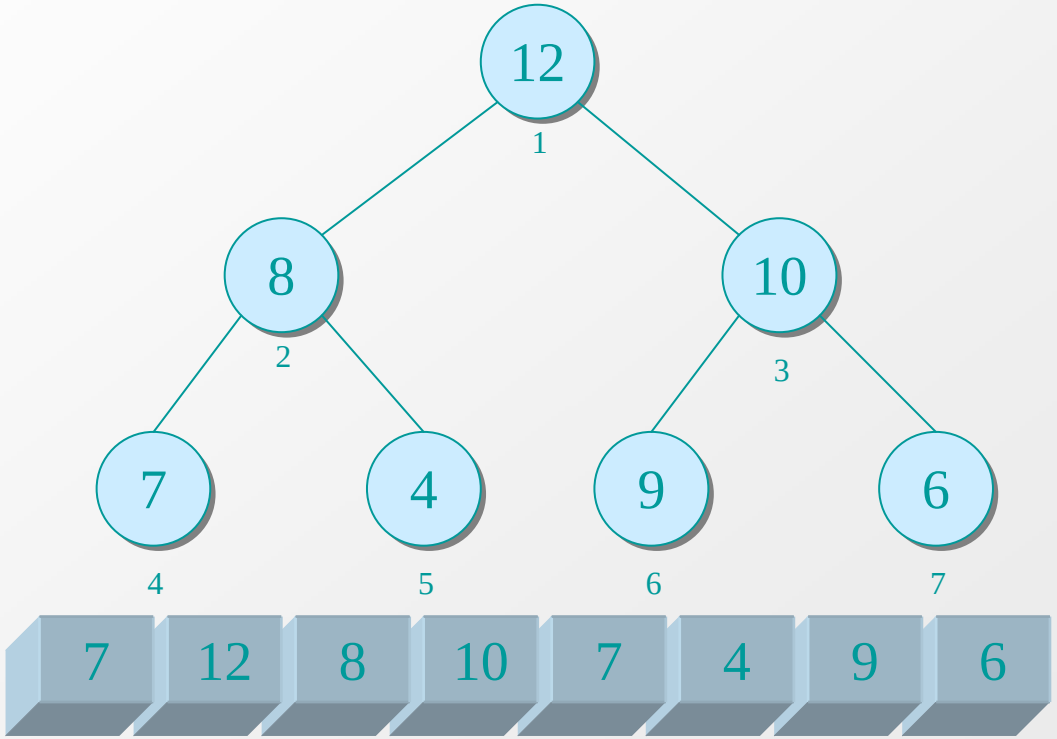
Heap Sort

- To sort a heap
  - Put the heap size into array[0]



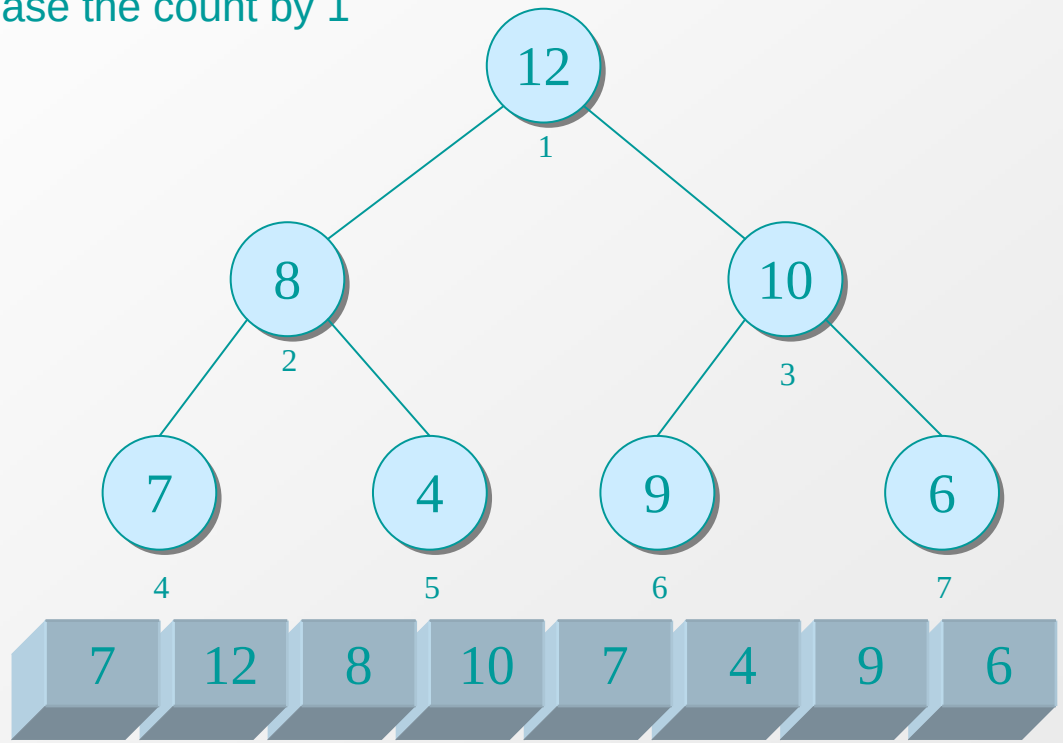
Heap Sort

- To sort a heap
  - Put the heap size into array[0]



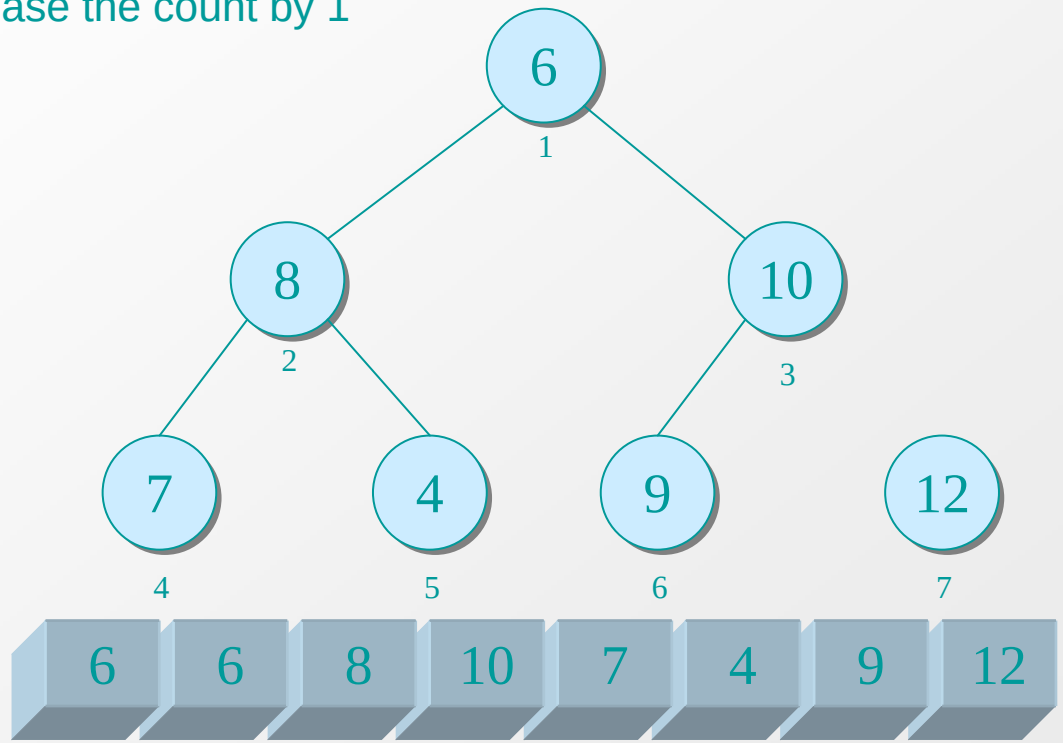
Heap Sort

- To sort a heap
  - Swap the largest number with the last element of the heap
  - Decrease the count by 1



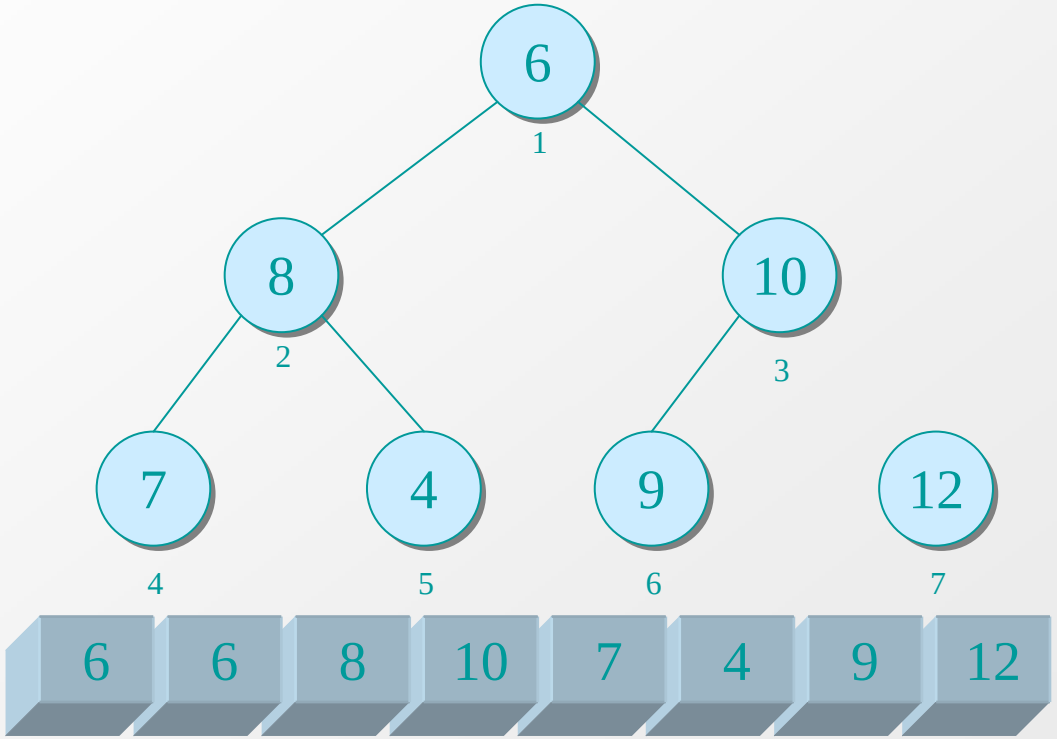
Heap Sort

- To sort a heap
  - Swap the largest number with the last element of the heap
  - Decrease the count by 1



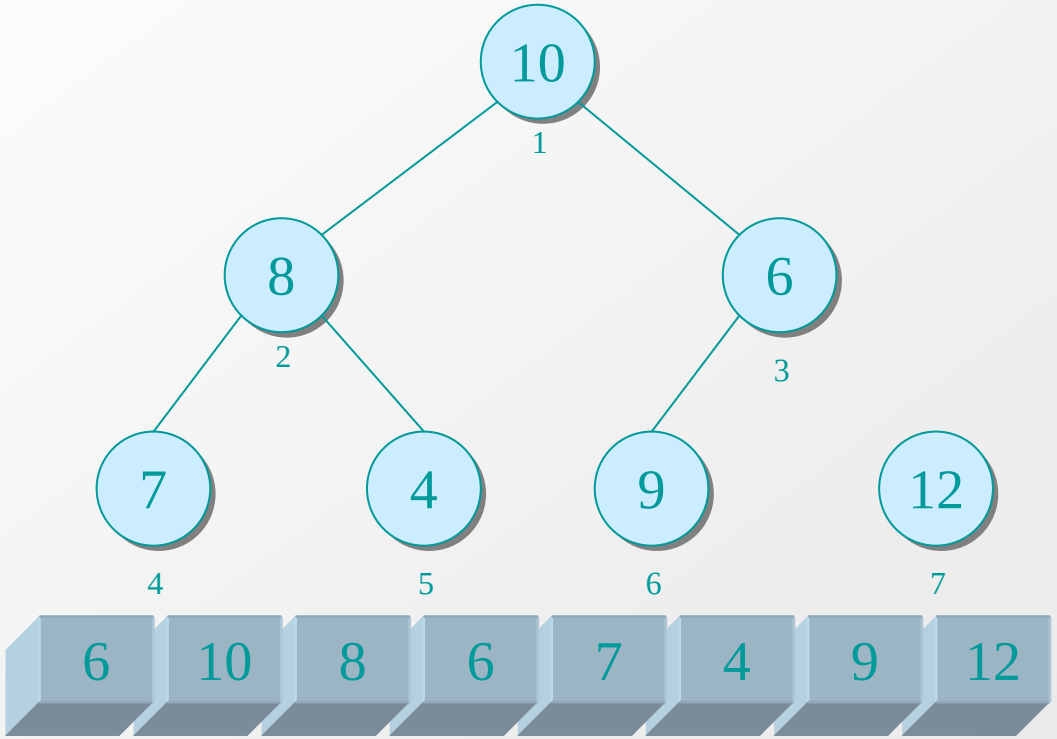
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



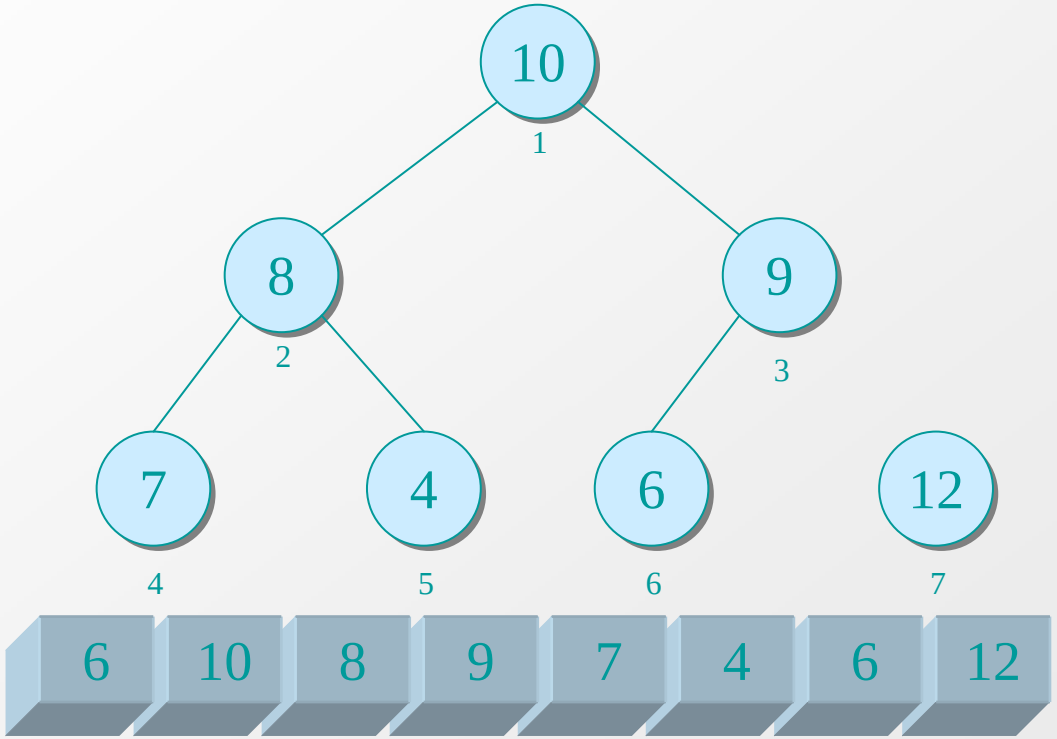
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



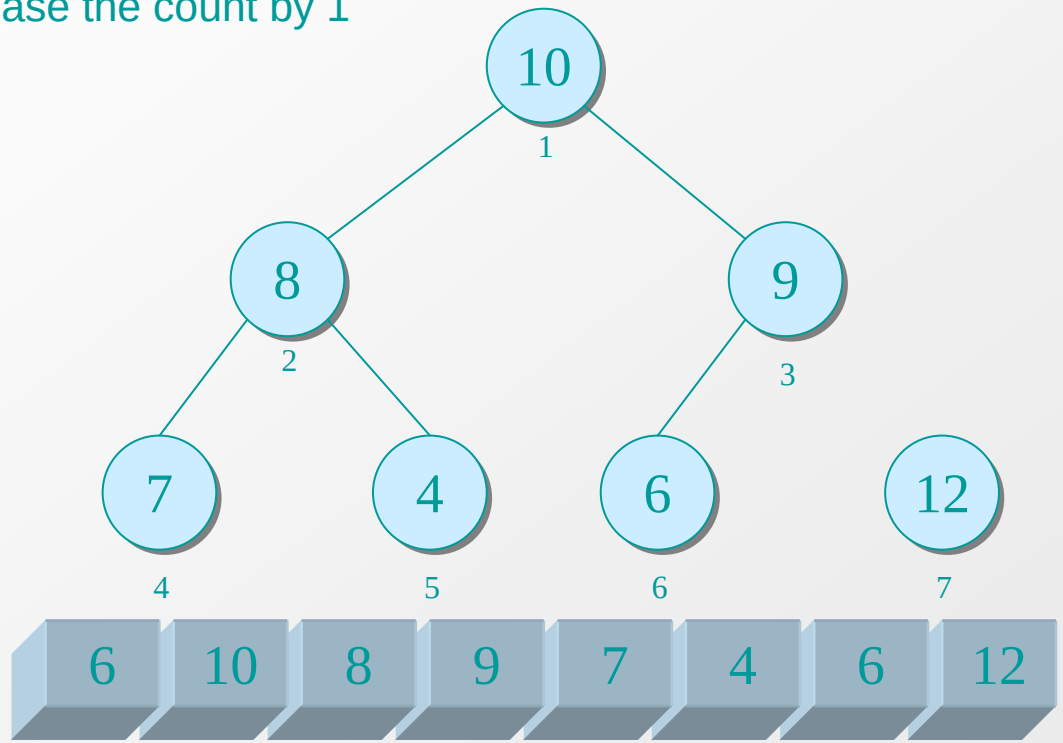
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



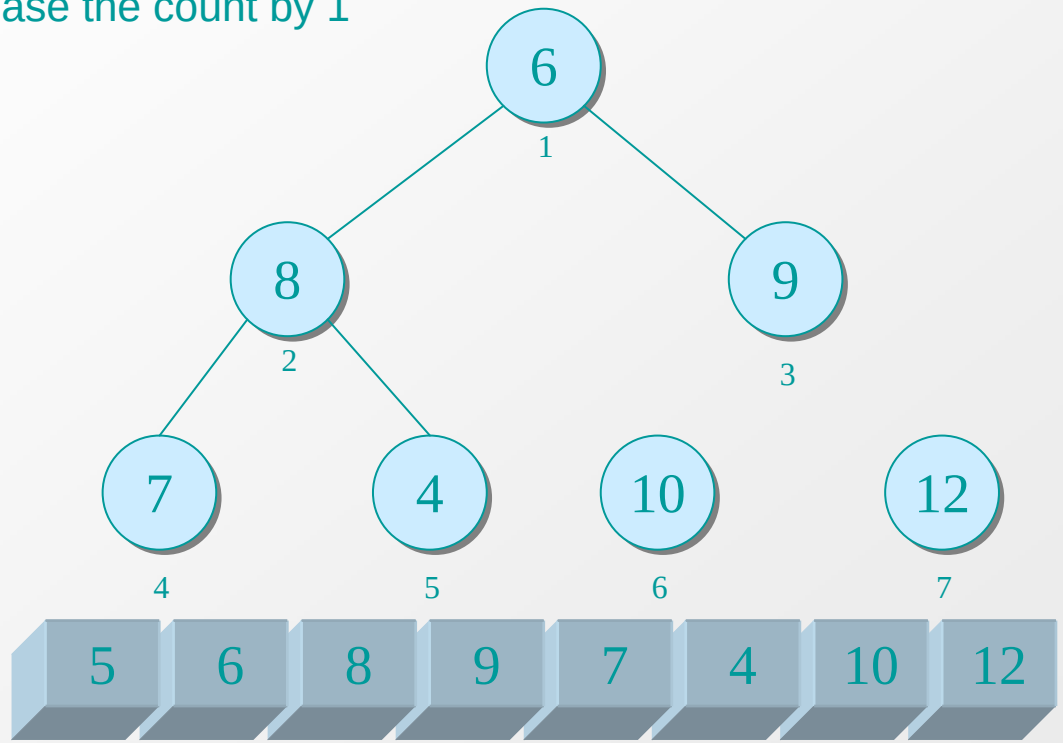
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



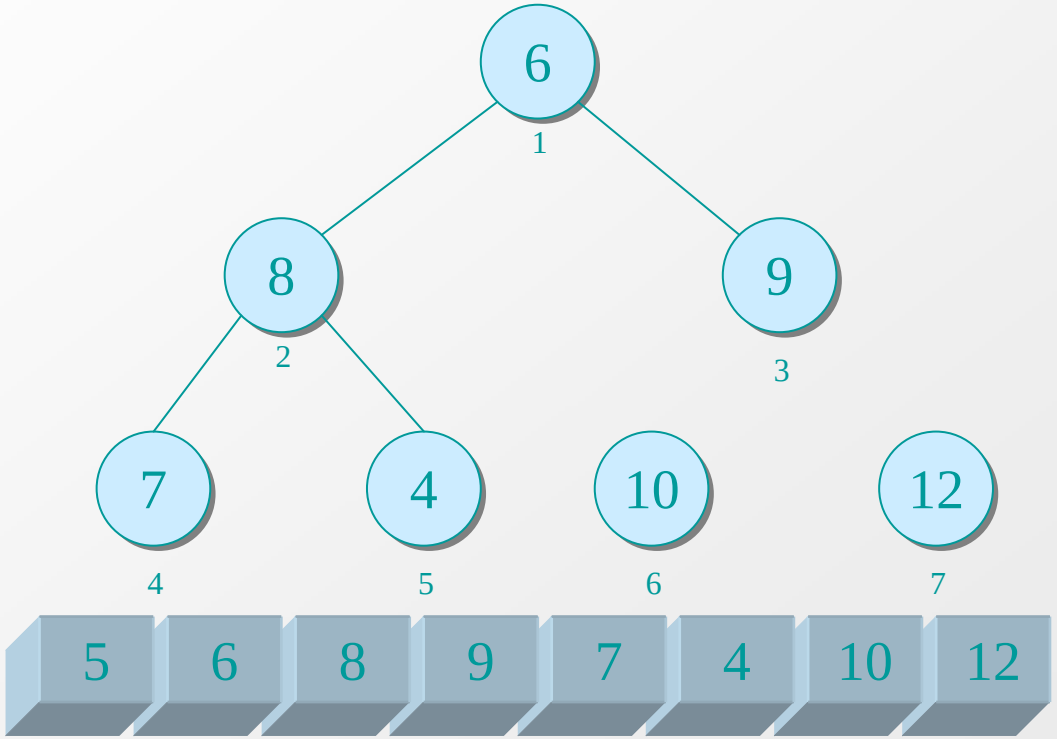
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



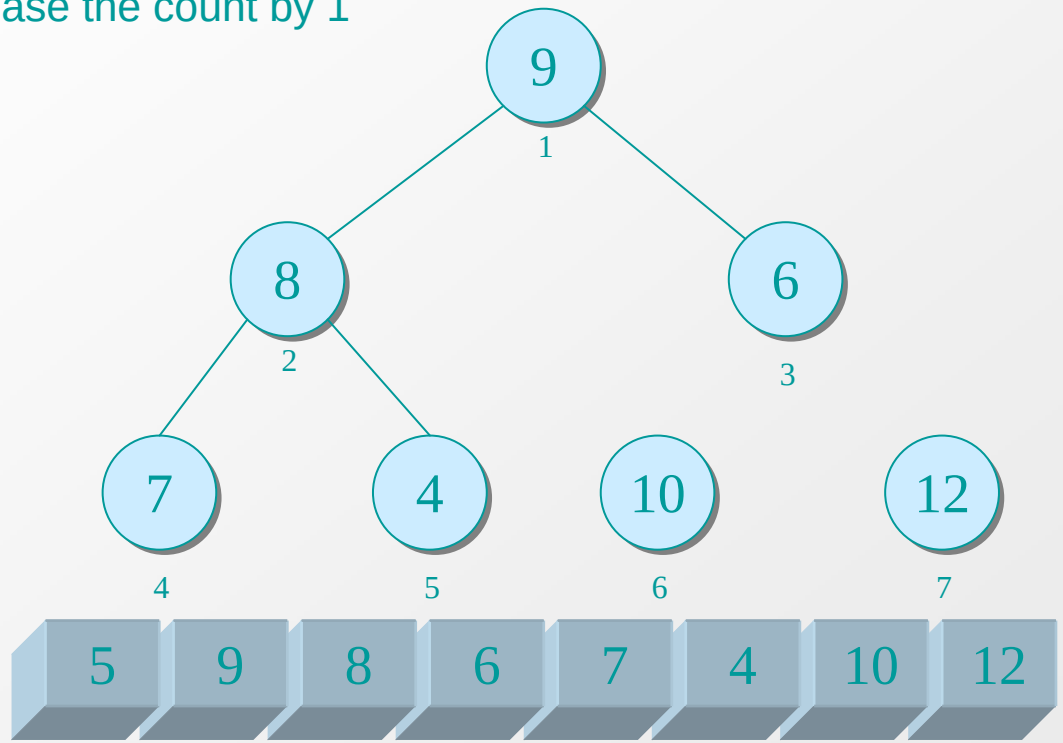
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



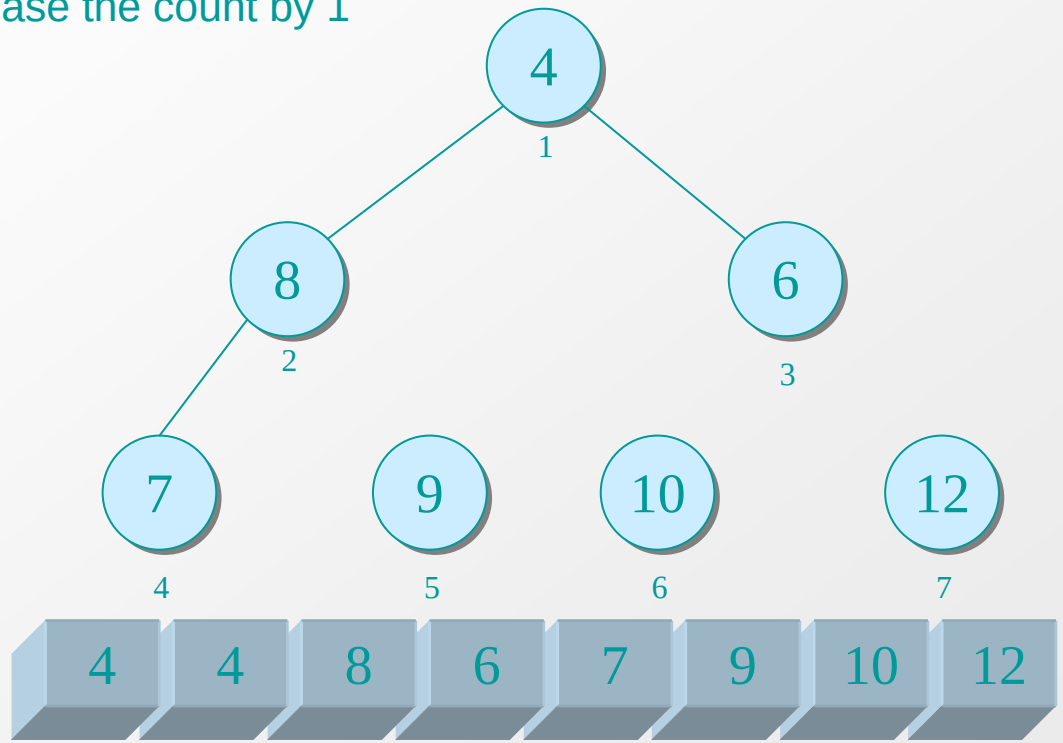
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



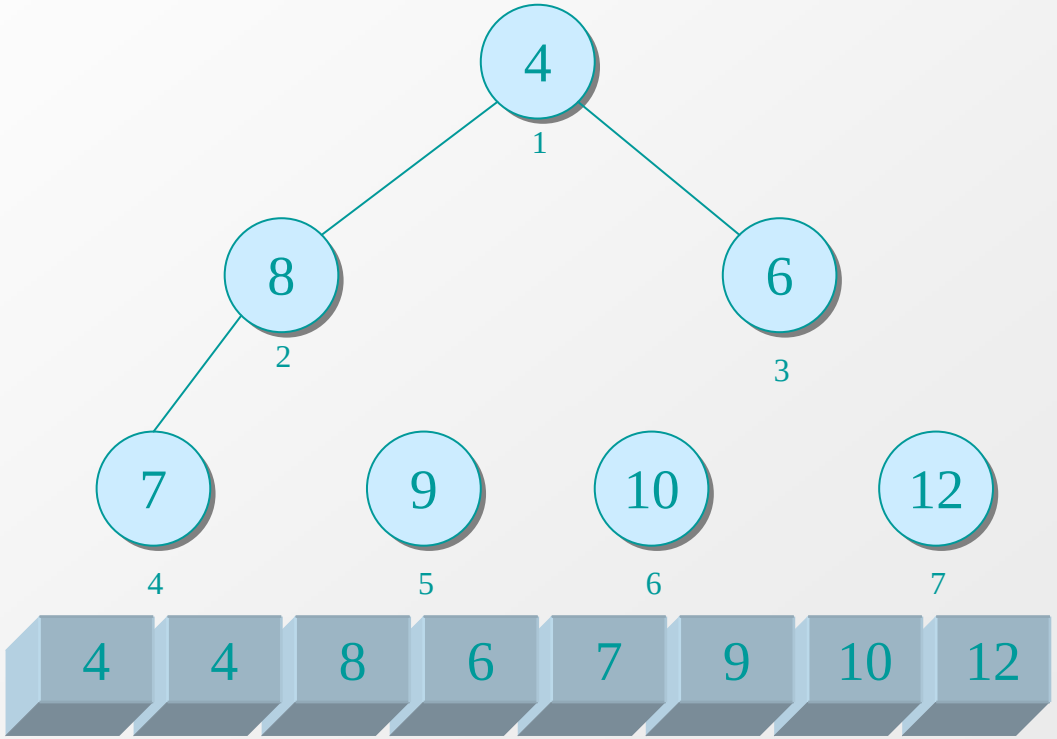
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



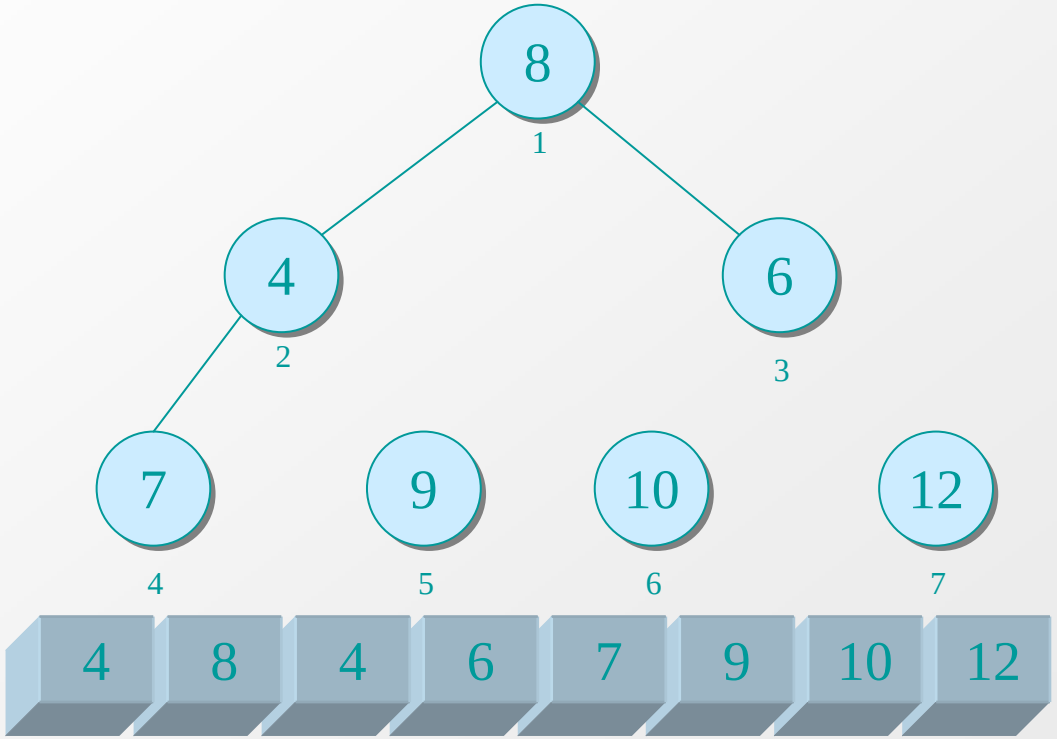
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



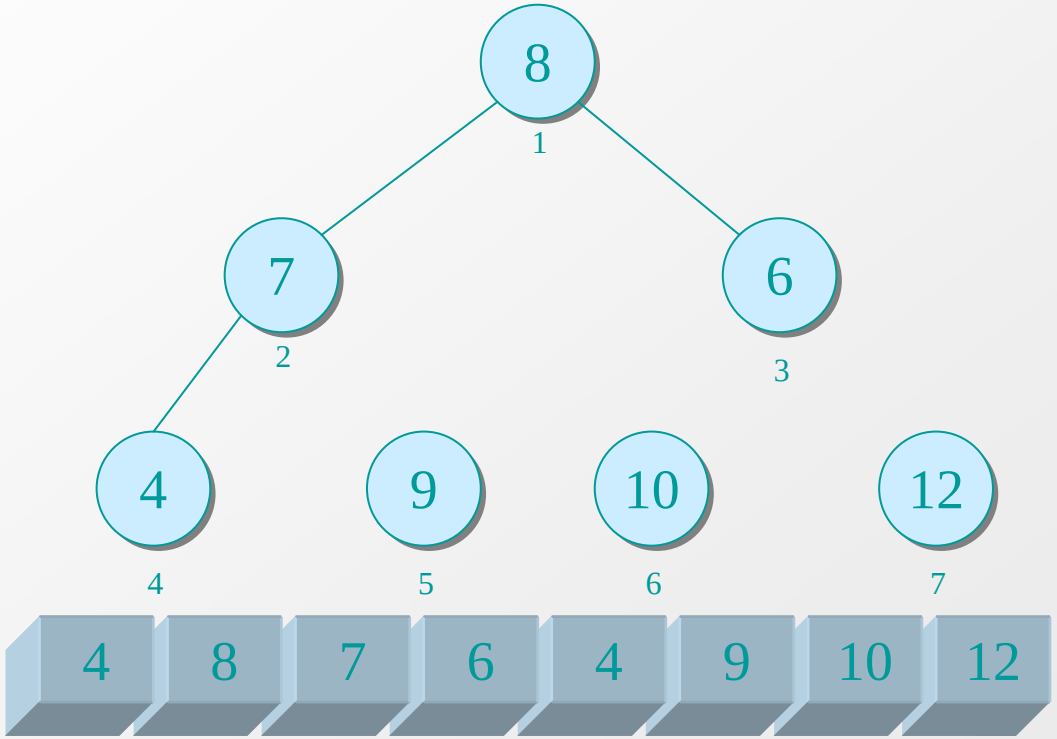
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



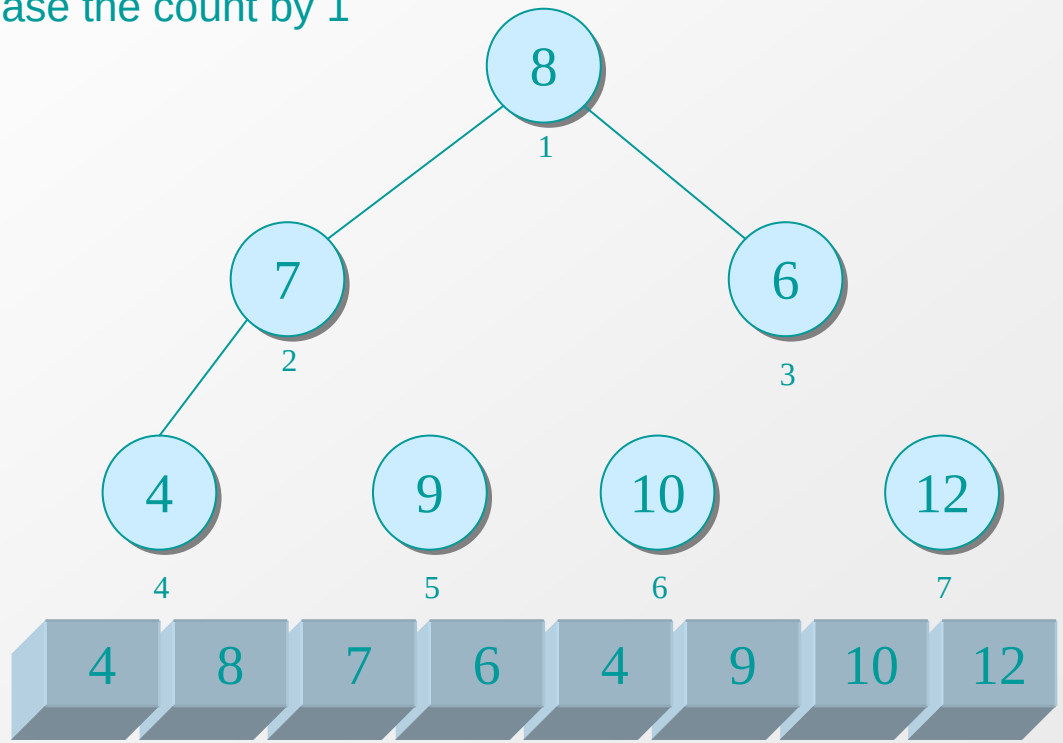
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



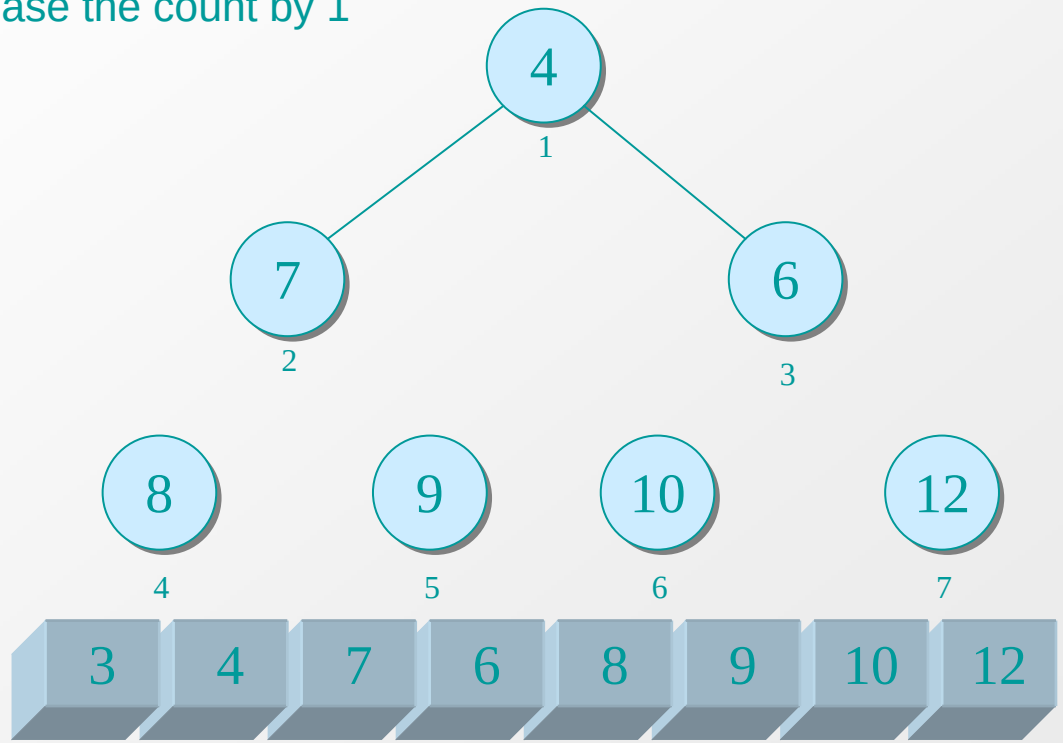
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



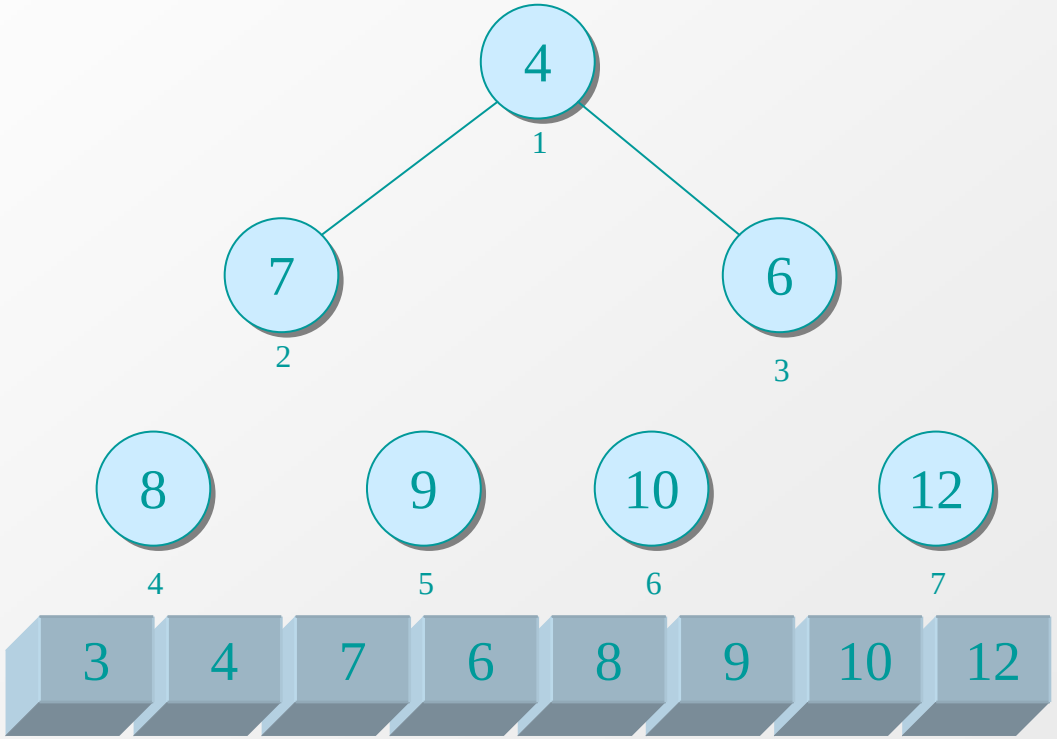
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



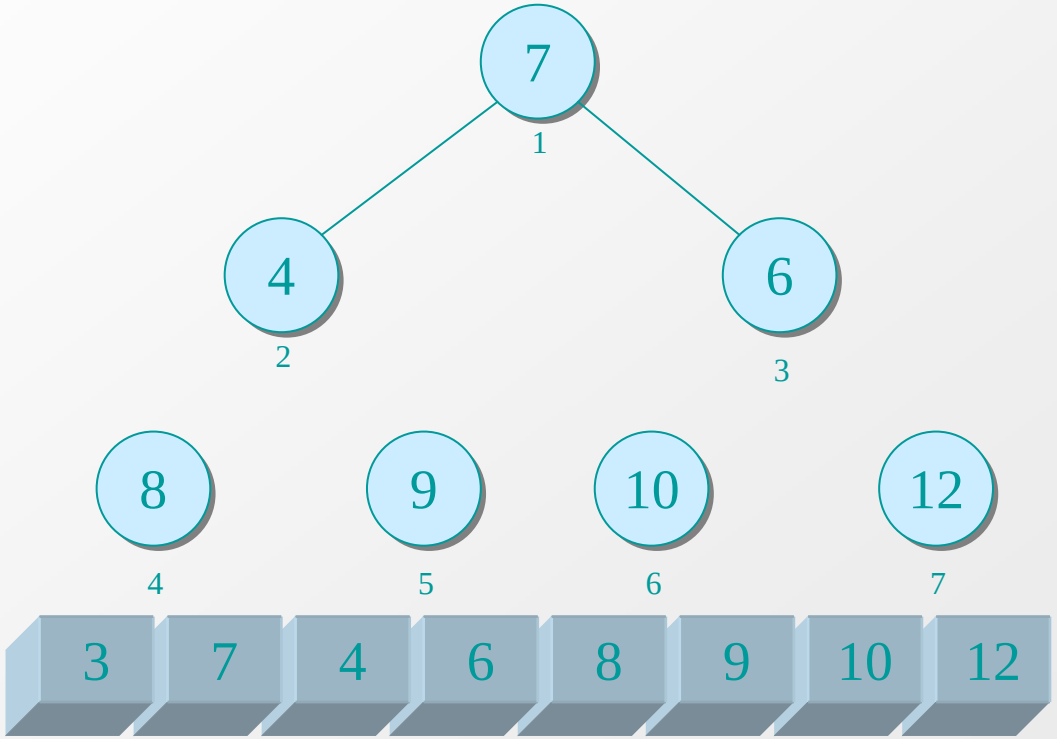
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



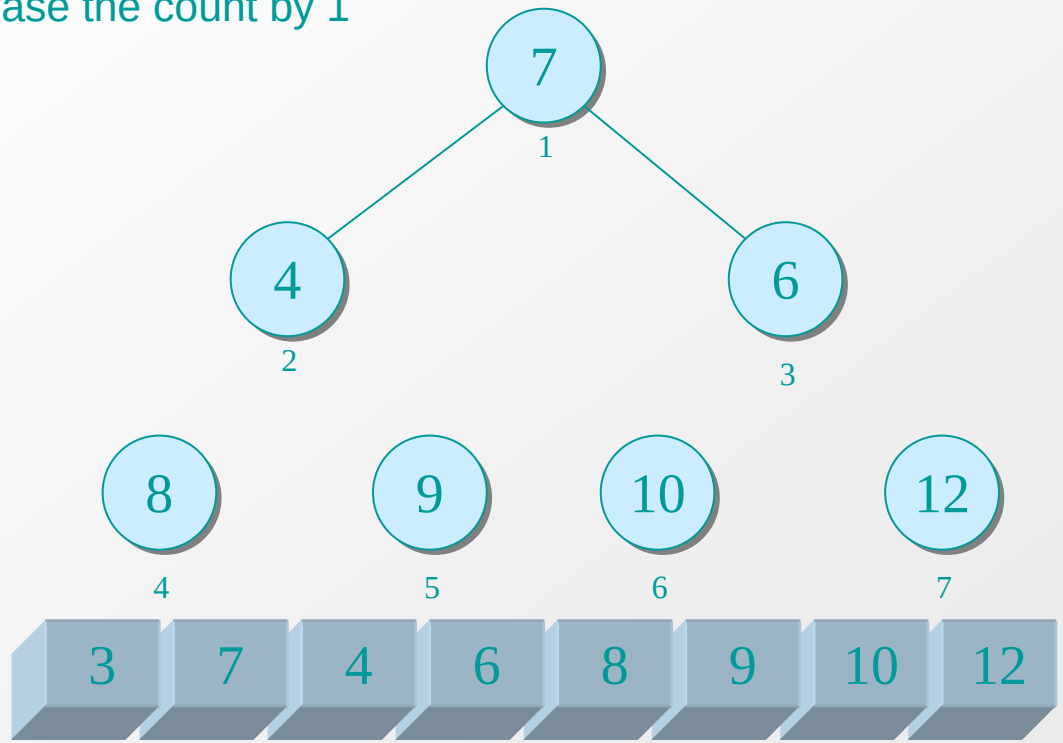
Heap Sort

- To sort a heap
  - The tree is no longer a heap. Push the value down until it is



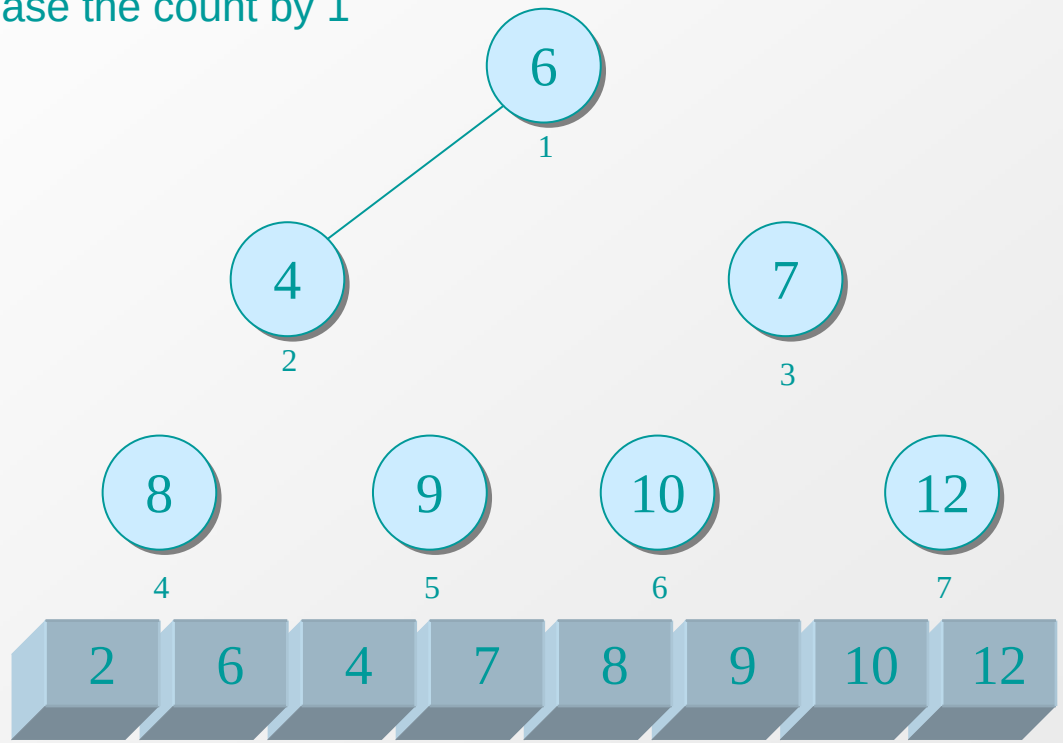
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



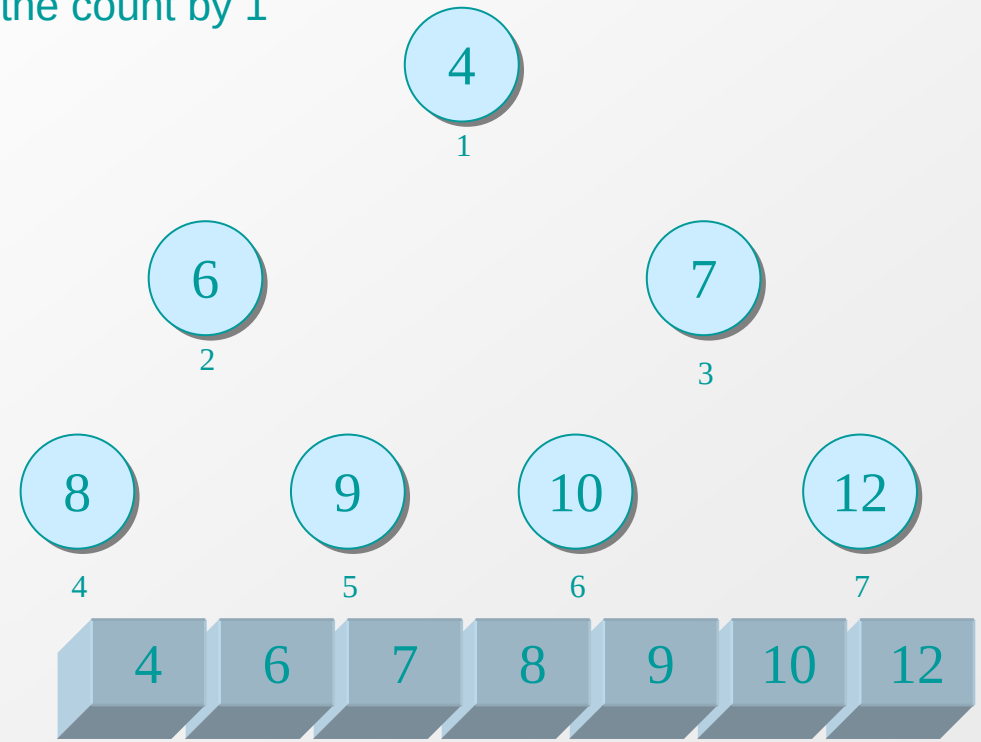
Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1



Heap Sort

- To sort a heap
  - Take the largest value and swap it for the last element in the heap
  - Decrease the count by 1

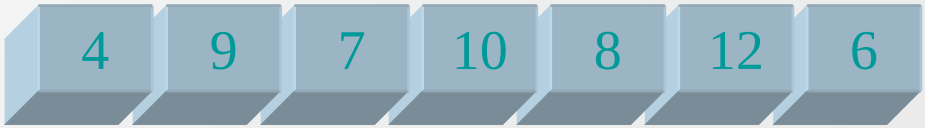


Heap Sort

```
sub pushdown {
    my ($heap, $i) = @_;
    my $size = $heap->[0];
    while($i <= $size/2) {
        my $child = $i * 2;
        if ($child < $size and $heap->[$child]) {
            $child++;
        }
        if ($heap->[$i] >= $heap->[$child]) { last }
        ($heap->[$i], $heap->[$child])
            = ($heap->[$child], $heap->[$i]);
        $i = $child;
    }
}
```

Building the Heap

- To build a heap
  - Fist, `unshift` the arrays size into it's first element

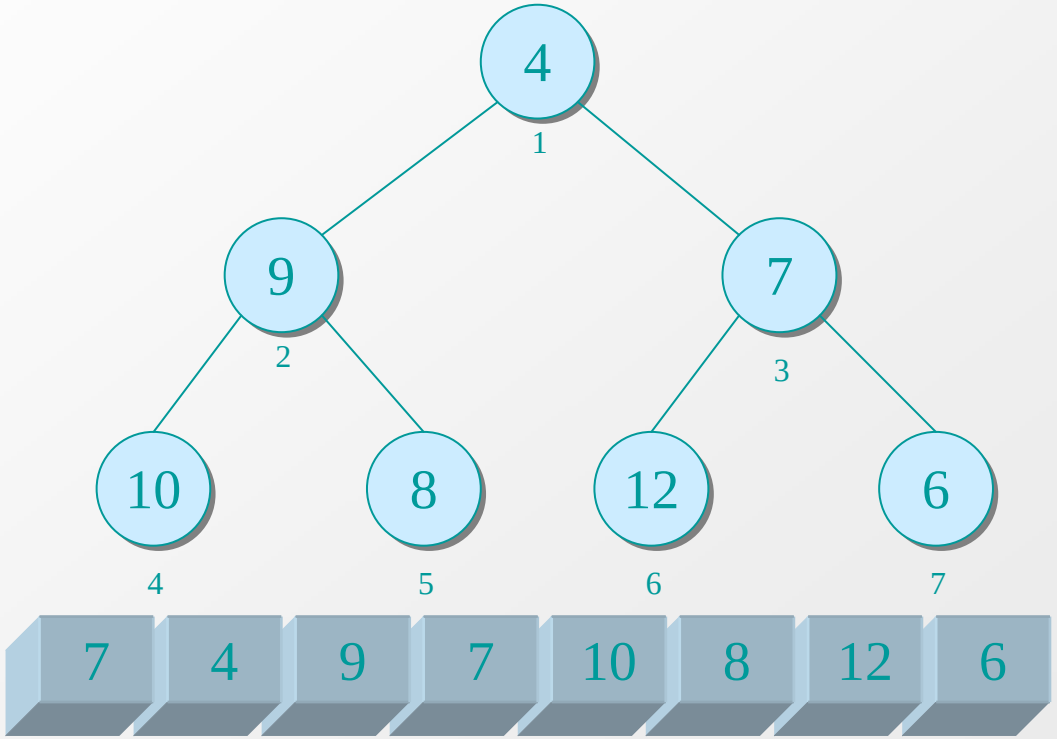


Building the Heap

- To build a heap
  - Fist, `unshift` the arrays size into it's first element

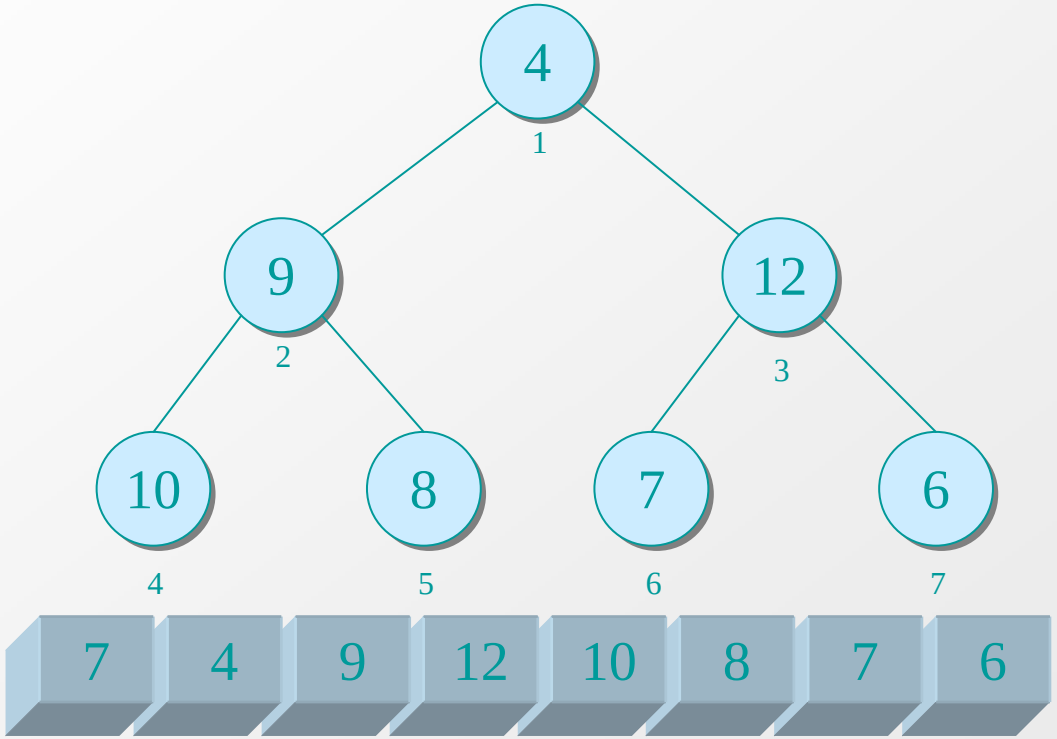
Building the Heap

- To build a heap
  - Call `pushdown()` on the last parent node



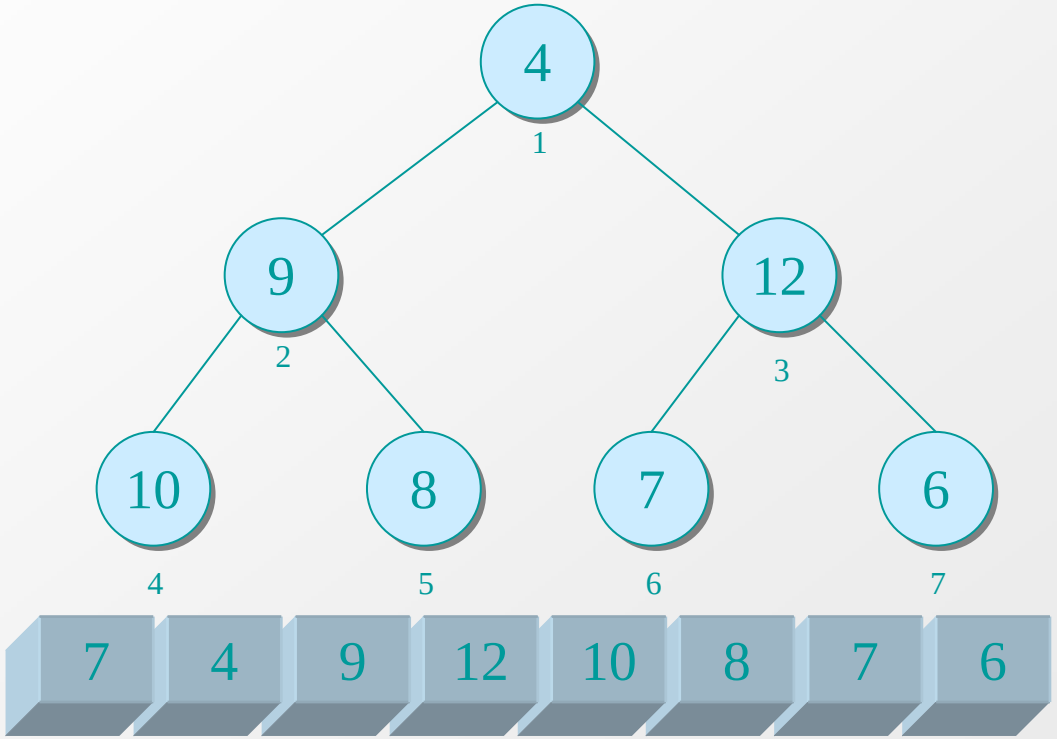
Building the Heap

- To build a heap
  - Call `pushdown()` on the last parent node



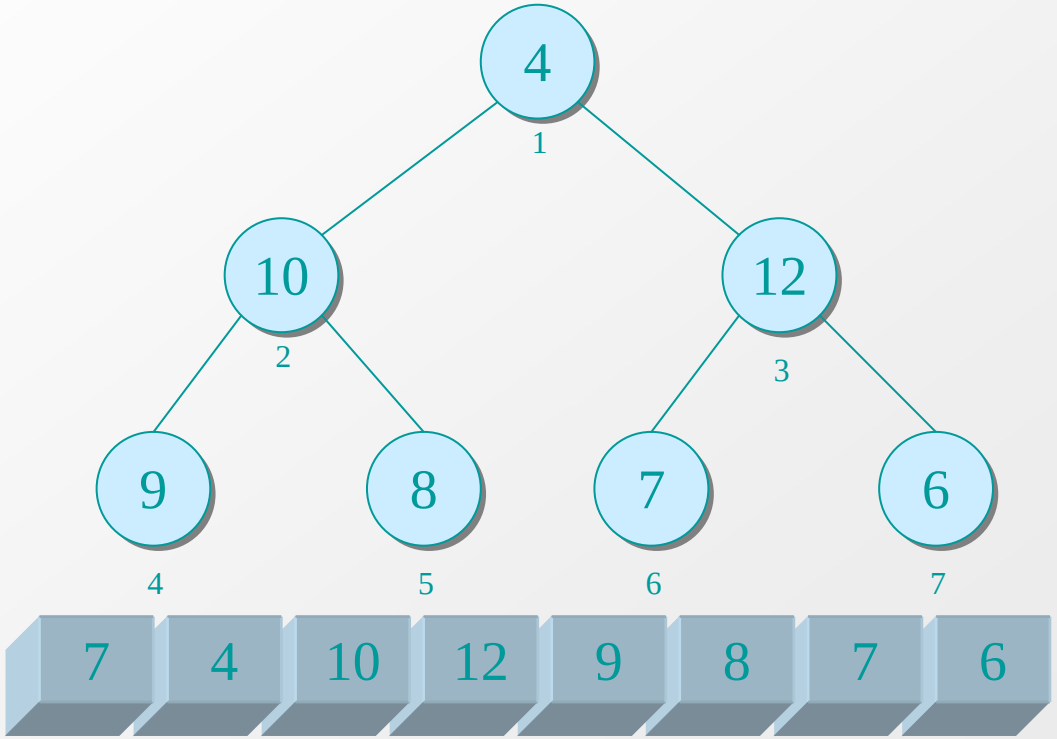
Building the Heap

- To build a heap
  - Call `pushdown()` on the next to the last parent node



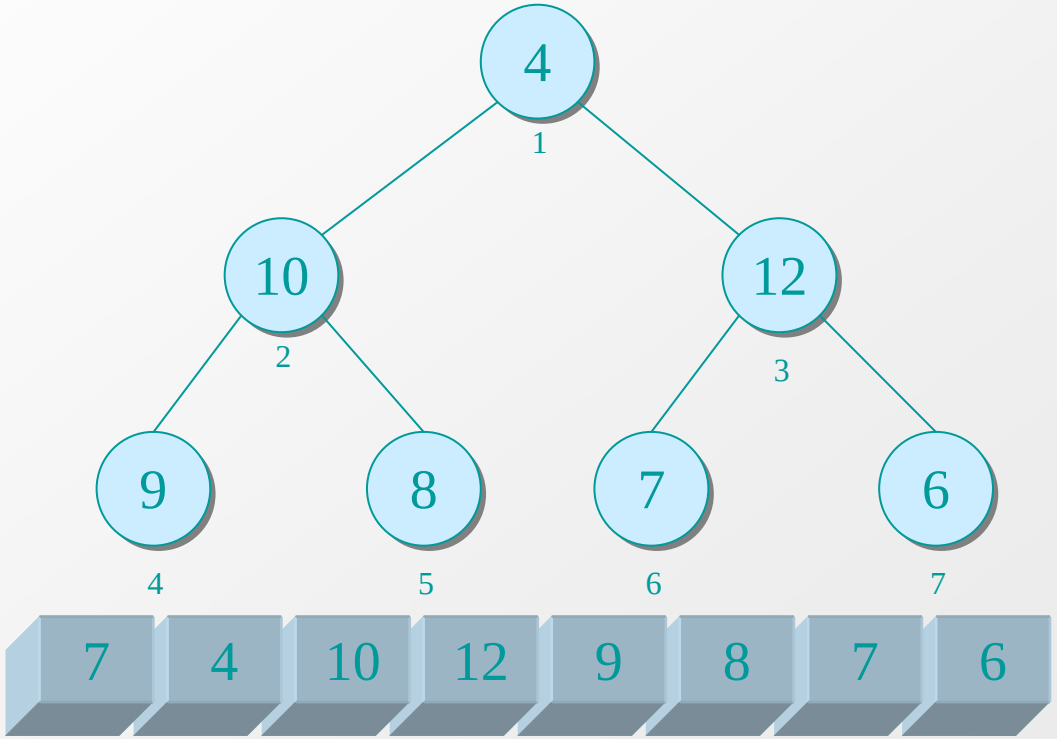
Building the Heap

- To build a heap
  - Call `pushdown()` on the next to the last parent node



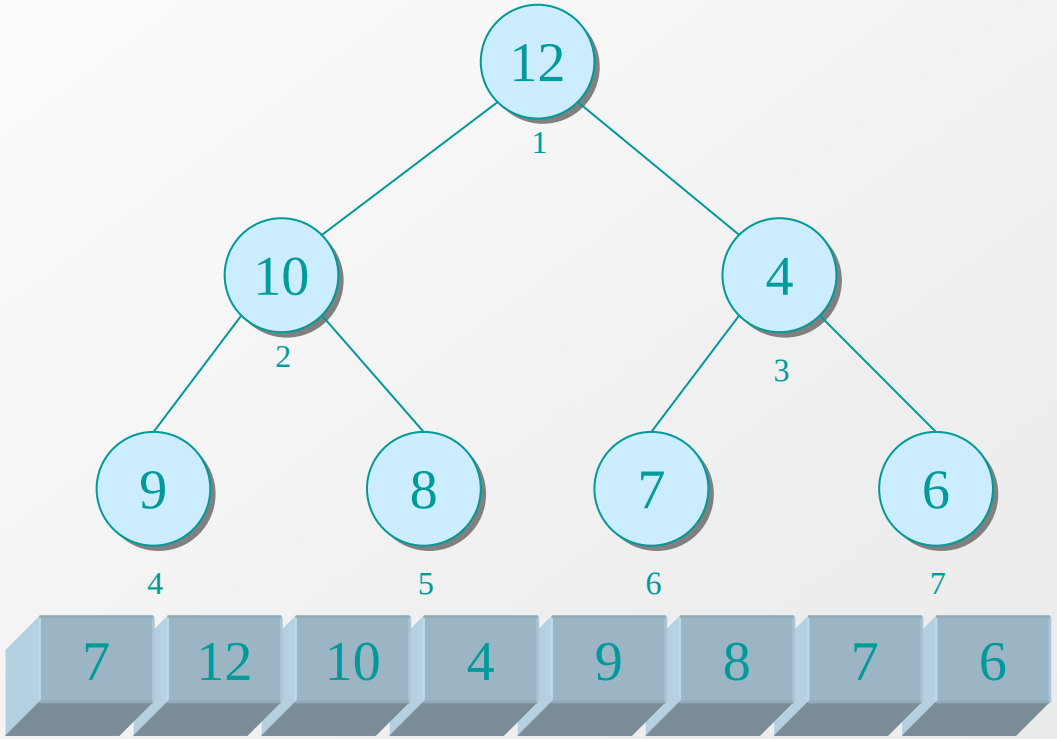
Building the Heap

- To build a heap
  - Call `pushdown()` on the head parent node



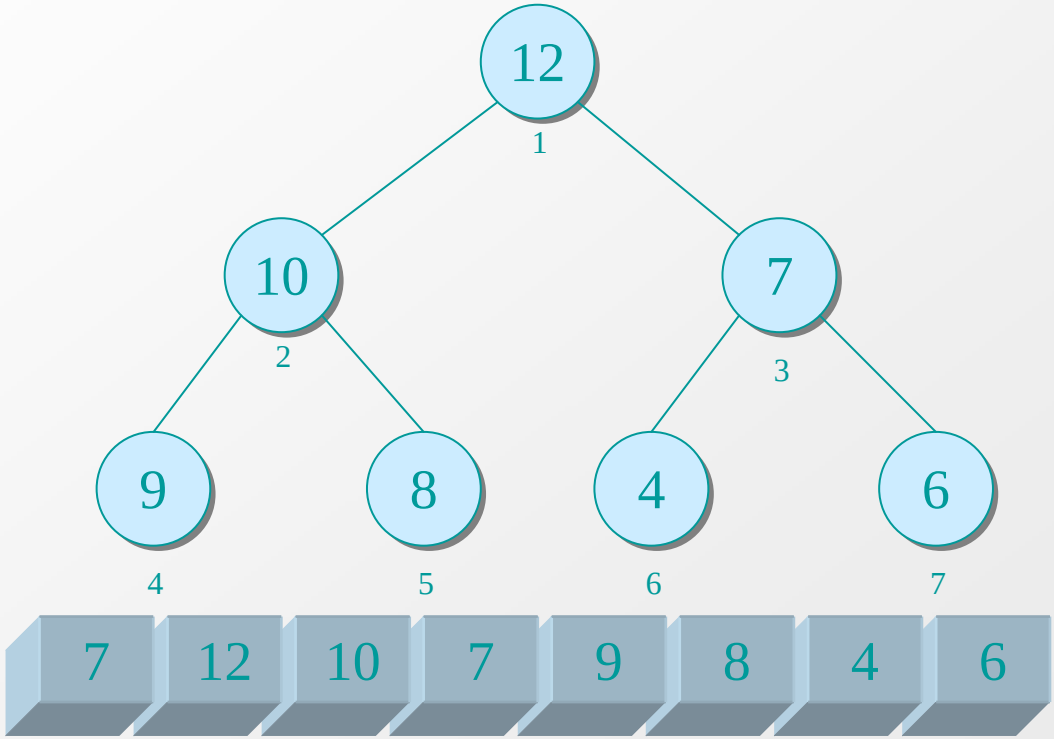
Building the Heap

- To build a heap
  - Call `pushdown()` on the head parent node



Building the Heap

- To build a heap
  - Call `pushdown()` on the head parent node



Heap Sort

```
sub heap_sort {
    my $heap = \@_;
    unshift @$heap, scalar @_;
    for (my $i = int($heap->[0] / 2); $i >= 1; $i--) {
        pushdown($heap, $i);
    }
    for (my $i = $heap->[0]; $i >= 2; $i--) {
        ($heap->[1], $heap->[$i]) = ($heap->[$i], $heap->[1]);
        $heap->[0]--;
        pushdown($heap, 1);
    }
    shift $heap;
}
```

Heap Sort

- This appears to be as much time, and a lot more work than, say, an insertion short
- The heap sort works in $N * \log_2 N$
- $\log_2 N$ is roughly the height of the heap, so $N * h$
- The height grow slowly: for $N = 1000$, h is 10
- So for larger arrays a heap sort is faster!

Other Algorithms

- The binary heap is an advanced data structure, there are others
  - Linked Lists, Circular Linked Lists, Doubly-Linked Lists
  - General heaps
  - Check out the `Heaps` module on CPAN
- Other Algorithms
  - Sets, Matrices
  - Graphs
  - Strings
  - Statistics, Numerical Analysis
  - Number Theory, Cryptography

Homework Ten

- Rewrite the `pushdown()` routine to use recursion rather than a while loop to push an element down the heap. Will this help or hurt the running time of the heap sort or make little difference?