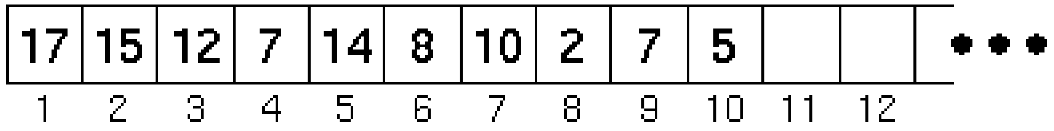
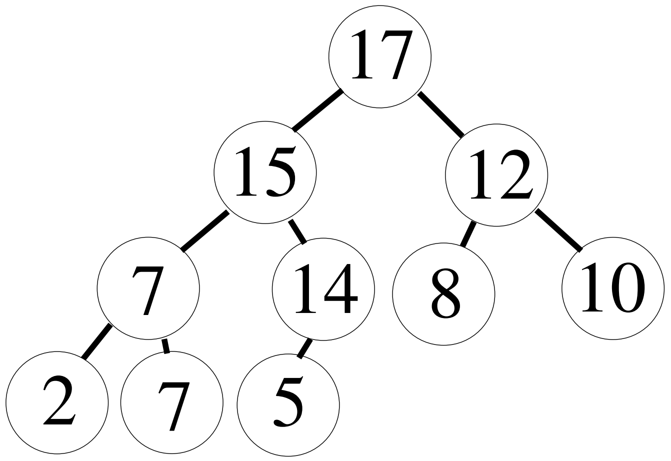
Heap

An array $A[1 \ldots n]$ is a *heap*, if $A[i] \geq A[2i]$ and $A[i] \geq A[2i+1]$ always when $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$ (and $2i+1 \leq n$).



The structure is easier to understand if we define the heap as a completely balanced binary tree, where

- the root is stored in the array at index 1
- the children of the node at index i are stored at $2i$ and $2i+1$ (if they exist)
- the parent of the node at index i is stored at $\lfloor \frac{i}{2} \rfloor$



Thus, the value of each node is larger or equal to the values of its children

Each level in the heap tree is full, except maybe the last one, where only some rightmost leaves may be missing

In order to make it easier to see the heap as a tree, let's define subroutines that find the parent and the children.

- they can be implemented very efficiently by shifting bits
- the running time of each is always $\Theta(1)$

```
PARENT($i$)
    **return** $\lfloor i/2 \rfloor$

LEFT($i$)
    **return** $2i$

RIGHT($i$)
    **return** $2i + 1$
```

$\Rightarrow$ Now the heap property can be given with:

$A[\text{PARENT}(i)] \geq A[i]$ *always when* $2 \leq i \leq A.heapsize$

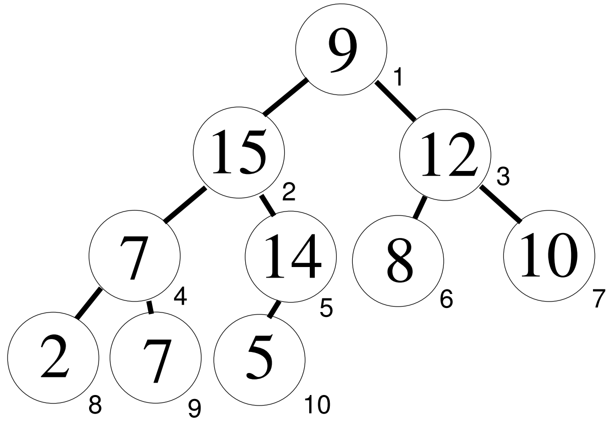
- $A.heapsize$ gives the size of the heap (we'll later see that it's not necessarily the size of the array)

Due to the heap property, the largest element of the heap is always its root, i.e. at the first index in the array.

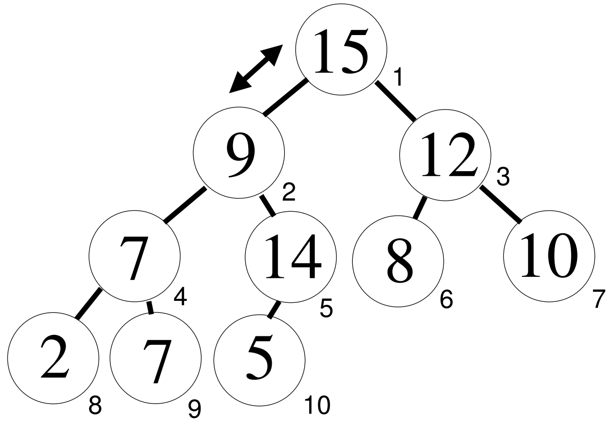
If the height of the heap is h, the amount of its nodes is between $2^h \ldots 2^{h+1} - 1$.
$\Rightarrow$ If there are n nodes in the heap its height is $\Theta(\lg n)$.

Adding an element from the top of the heap:

- let's assume that $A[1 \ldots n]$ is otherwise a heap, except that the heap property does not hold for the root of the heap tree
  - in other words $A[1] < A[2]$ or $A[1] < A[3]$

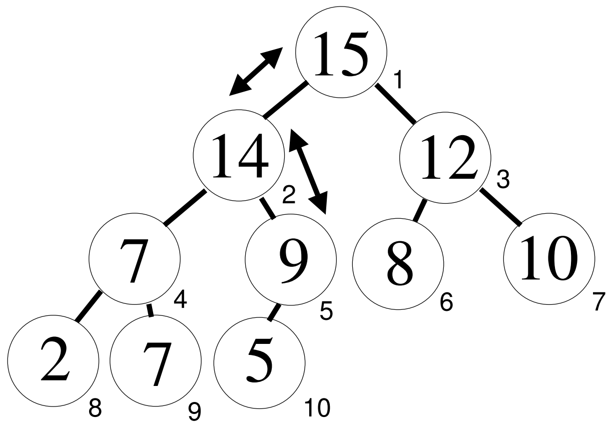


- the problem can be moved downwards in the tree by selecting the largest of the root's children and swapping it with the root
  - in order to maintain the heap property the largest of the children needs to be chosen - it is going to become the parent of the other child



- the same can be done to the subtree, whose root thus turned problematic and again to its subtree etc. until the problem disappears
  - the problem is solved for sure once the execution reaches a leaf

    $\Rightarrow$ the tree becomes a heap



The same as pseudocode

```
HEAPIFY($A, i$)                                  ($i$ is the index where the element might be too small)
1  repeat                                          (repeat until the heap is fixed)
2      $old\_i$ := $i$                             (store the value of $i$)
3      $l$ := LEFT($i$)
4      $r$ := RIGHT($i$)
5      if $l \leq A.heapsize$ and $A[l] > A[i]$ then    (the left child is larger than $i$)
6          $i$ := $l$
7      if $r \leq A.heapsize$ and $A[r] > A[i]$ then    (right child is even larger)
8          $i$ := $r$
9      if $i \neq old\_i$ then                     (if a larger child was found...)
10         exchange $A[old\_i] \leftrightarrow A[i]$    (...move the problem downwards)
11 until $i = old\_i$                              (if the heap is already fixed, exit)
```

- The execution is constant time if the condition on line 11 is true the first time it is met: $\Omega(1)$.
- In the worst case the new element needs to be moved all the way down to the leaf.
  $\Rightarrow$ The running time is $O(h)$ = $O(\lg n)$.

Building a heap

- the following algorithm converts an array into a heap:

```
BUILD-HEAP($A$)
1   $A.heapsize$ := $A.length$                  (the heap is built out of the entire array)
2   **for** $i$ := $n\lfloor A.length/2 \rfloor$ **downto** 1 **do**   (scan through the lower half of the array )
3       HEAPIFY($A, i$)                          (call Heapify)
```

- The array is scanned from the end towards the beginning and HEAPIFY is called for each node.
  - before calling HEAPIFY the heap property always holds for the subtree rooted at i except that the element in i may be too small
  - subtrees of size one don't need to be fixed as the heap property trivially holds
  - after HEAPIFY(A, i) the subtree rooted at i is a heap

  $\Rightarrow$ after HEAPIFY($A, 1$) the entire array is a heap

- BUILD-HEAP executes the **for**-loop $\lfloor \frac{n}{2} \rfloor$ times and HEAPIFY is $\Omega(1)$ and $O(\lg n)$ so
  - the best case running time is $\lfloor \frac{n}{2} \rfloor \cdot \Omega(1) + \Theta(n) = \Omega(n)$
  - the program never uses more than $\lfloor \frac{n}{2} \rfloor \cdot O(\lg n) + \Theta(n) = O(n \lg n)$
- The worst-case running time we get this way is however too pessimistic:
  - HEAPIFY is $O(h)$, where h is the height of the heap tree
  - as i changes the height of the tree changes

level	h	number of HEAPIFY calls
lowest	0	0
2nd	1	$\lfloor \frac{n}{4} \rfloor$
3rd	2	$\lfloor \frac{n}{8} \rfloor$
...	...	...
topmost	$\lfloor \lg n \rfloor$	1

  - thus the worst case runnign time is $\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \cdots = \frac{n}{2} \cdot \Sigma_{i=1}^{\infty} \frac{i}{2^i} = \frac{n}{2} \cdot 2 = n \Rightarrow O(n)$

$\Rightarrow$ the running time of BUILD-HEAP is always $\Theta(n)$

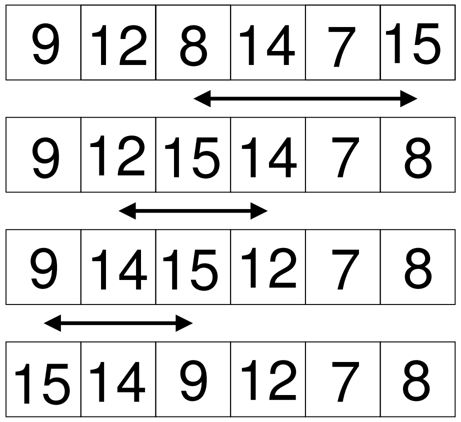
Sorting with a heap

The following algorithm can be used to sort the contents of the array efficiently:

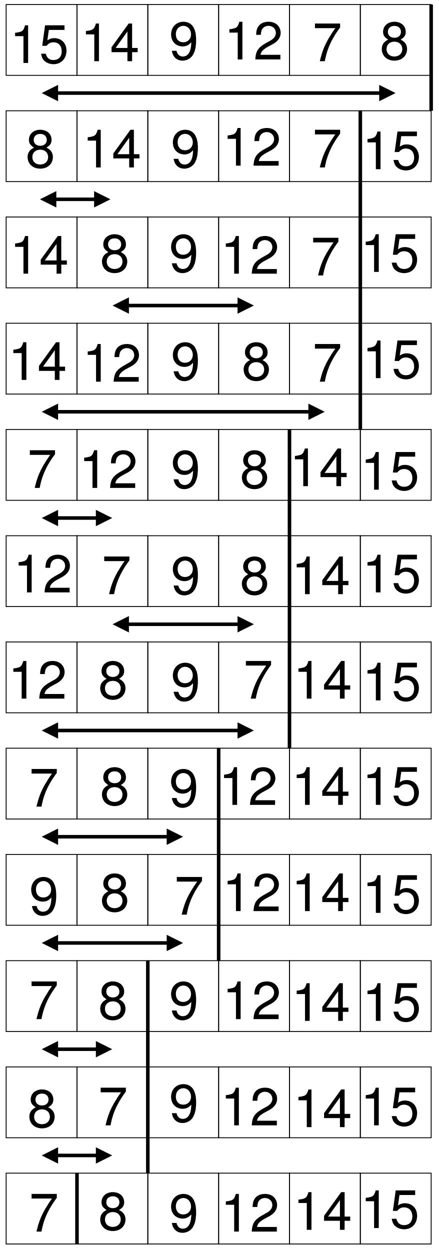
```
HEAPSORT($A$)
1  BUILD-HEAP($A$)                         (convert the array into a heap)
2  for $i$ := $A.length$ downto $2$ do     (scan the array from the last to the first element)
3      exchange $A[1] \leftrightarrow A[i]$  (move the heap´s largest element to the end)
4      $A.heapsize$ := $A.heapsize - 1$     (move the largest element outside the heap)
5      HEAPIFY($A, 1$)                      (fix the heap, which is otherwise fine...)
                                           (... except the first element may be too small)
```

Let´s draw a picture of the situation:

- first the array is converted into a heap
- it´s easy to see from the example, that the operation is not too laborous
  - the heap property is obviously weaker than the order



- the picture shows how the sorted range at the end of the array gets larger until the entire array is sorted
- the heap property is fixed each time the sorted range gets larger
- the fixing process seems complex in such a small example
  - the fixing process doesn't take a lot of steps even with large arrays, only a logarithmic amount



The running time of HEAPSORT consists of the following:

- BUILD-HEAP on line 1 is executed once: $\Theta(n)$
- the contents of the **for**-loop is executed n - 1 times
  - operations on lines 3 and 4 are constant time
  - HEAPIFY uses $\Omega(1)$ and $O(\lg n)$

  $\Rightarrow$ in total $\Omega(n)$ and $O(n \lg n)$
- the lower bound is exact
  - if all the elements have the same value the heap doesn't need to be fixed at all and HEAPIFY is always constant time
- the upper bound is also exact
  - proving this is more difficult and we find the upcoming result from the efficiency of sorting by counting sufficient

Note! The efficiency calculations abowe assume that the data structure used to store the heap provides a constant time indexing.

- Heap is worth using only when this is true

Advantages and disadvantages of HEAPSORT

Advantages:

- sorts the array in place
- never uses more than $\Theta(n \lg n)$ time

Disadvantages:

- the constant coefficient in the running time is quite large
- instability
  - elements with the same value don't maintain their order

8.3 Priority queue

A *priority queue* is a data structure for maintaining a set S of elements, each associated with a *key* value. The following operations can be performed:

- INSERT(S, x) inserts the element x into the set S
- MAXIMUM(S) returns the element with the largest key
  - if there are several elements with the same key, the operation can choose any one of them
- EXTRACT-MAX(S) removes and returns the element with the largest key
- alternatively the operations MINIMUM(S) and EXTRACT-MIN(S) can be implemented
  - there can be **only the maximum** or **only the minimun** operations implemented in the same queue

Priority queues can be used widely

- prioritizing tasks in an operating system
  - new tasks are added with the command INSERT
  - as the previous task is completed or interrupted the next one is chosen with EXTRACT-MAX
- action based simulation
  - the queue stores incoming (not yet simulated) actions
  - the key is the time the action occurs
  - an action can cause new actions
    $\Rightarrow$ they are added to the queue with INSERT
  - EXTRACT-MIN gives the next simulated action
- finding the shortest route on a map
  - cars driving at constant speed but choosing different routes are simulated until the first one reaches the destination
  - a priority queue is needed in practise in an algorithm for finding shortest paths, covered later

In practise, a priority queue could be implemented with an unsorted or sorted array, but that would be inefficient

- the operations MAXIMUM and EXTRACT-MAX are slow in an unsorted array
- INSERT is slow in a sorted array

A heap can be used to implement a priority queue efficiently instead.

- The elements of the set S are stored in the heap A.
- MAXIMUM(S) is really simple and works in $\Theta(1)$ running-time

```
HEAP-MAXIMUM($A$)
1  **if** $A.heapsize < 1$ **then**        *(there is no maximum in an empty heap)*
2      **error** "heap underflow"
3  **return** $A[1]$                       *(otherwise return the first element in the array)*
```

- EXTRACT-MAX(S) can be implemented by fixing the heap after the extraction with HEAPIFY.
- HEAPIFY dominates the running-time of the algorithm: $O(\lg n)$.

```
HEAP-EXTRACT-MAX($A$)
1  **if** $A.heapsize < 1$ **then**           *(no maximum in an empty heap)*
2      **error** "heap underflow"
3  $max$ := $A[1]$                           *(the largest element is at the first index)*
4  $A[1]$ := $A[A.heapsize]$                 *(make the last element the root)*
5  $A.heapsize$ := $A.heapsize - 1$          *(decrement the size of the heap)*
6  HEAPIFY($A, 1$)                           *(fix the heap)*
7  **return** $max$
```

- INSERT(S, x) adds a new element into the heap by making it a leaf and then by lifting it to its correct height based on its size
  - is works like HEAPIFY, but from bottom up
  - in the worst-case, the leaf needs to be lifted all the way up to the root: running-time $O(\lg n)$

```
HEAP-INSERT($A, key$)
1  $A.heapsize$ := $A.heapsize + 1$                   (increment the size of the heap)
2  $i$ := $A.heapsize$                                (start from the end of the array)
3  while $i > 1$ and $A[$PARENT$(i)] < key$ do        (continue until the root or ...)
                                                      (... or a parent with a larger value is reached)
4      $A[i]$ := $A[$PARENT$(i)]$                     (move the parent downwards)
5      $i$ := PARENT$(i)$                             (move upwards)
6  $A[i]$ := $key$                                    (place the key into its correct location)
```

$\Rightarrow$ Each operation in the priority queue can be made $O(\lg n)$ by using a heap.

A priority queue can be thought of as an abstract data type which stores the data (the set S) and provides the operations INSERT, MAXIMUM,EXTRACT-MAX.

- the user sees the names and the purpose of the operations but not the implementation
- the implementation is encapsulated into a package (Ada), a class (C++) or an independent file (C)

$\Rightarrow$ It's easy to maintain and change the implementation when needed without needing to change the code using the queue.

9 Different Types of Sorting Algorithms

This chapter discusses sorting algorithms that use other approaches than comparisons to determine the order of the elements.

The maximum efficiency of comparison sorts, i.e. sorting based on the comparisons of the elements, is aldo examined.

Finally, the chapter covers the factors for choosing an algorithm.

9.1 Other Sorting Algorithms

All sorting algorithms covered so far have been based on comparisons.

- They determine the correct order of elements based on comparing their values to eachother.

It is possible to use other information besides comparisons to sort data.

Sorting by counting

Let's assume that the value range of the keys is small, atmost the same range than the amount of the elements.

- For simplicity we assume that the keys of the elements are from the set $\{1, 2, \ldots, k\}$, and $k = O(n)$.
- The amount of elements with each given key is calculated.
- Based on the result the elements are placed directly into their correct positions.

```
COUNTING-SORT$(A, B, k)$
1  for $i$ := $1$ to $k$ do
2      $C[i]$ := $0$                               (initialize a temp array $C$ with zero)
3  for $j$ := $1$ to $A.length$ do
4      $C[A[j].key]$ := $C[A[j].key] + 1$          (calculate the amount of elements with key $= i$)
5  for $i$ := $2$ to $k$ do
6      $C[i]$ := $C[i] + C[i-1]$                   (calculate how many keys $\leq i$)
7  for $j$ := $A.length$ downto $1$ do             (scan the array from end to beginning)
8      $B[C[A[j].key]]$ := $A[j]$                  (place the element into the output array)
9      $C[A[j].key]$ := $C[A[j].key] - 1$          (the next correct location is a step to the left)
```

The algorithm places the elements to their correct location in reverse order to quarantee stability.

Running-time:

- The first and the third **for**-loop take $\Theta(k)$ time.
- The second and the last **for**-loop take $\Theta(n)$ time.

$\Rightarrow$ The running time is $\Theta(n + k)$.

- If $k = O(n)$, the running-time is $\Theta(n)$.
- All basic operations are simple and there are only a few of them in each loop so the constant coefficient is also small.

COUNTING-SORT is not worth using if $k \gg n$.

- The memory consumption of the algorithm is $\Theta(k)$.
- Usually $k \gg n$.
  - for example: all possible social security numbers $\gg$ the social security numbers of TUT personnel

Sometimes there is a need to be able to sort based on seeral keys or a key with several parts.

- the list of exam results: sort first based on the department and then those into an alphabetical order
- dates: first based on the year, then the month, and the day
- a deck of cards: first based on the suit and then those according to the numbers

The different criteria are taken into account as follows

- The most significant criterion according to which the values of the elements differ determines the result of the comparison.
- If the elements are equal with each criteria they are considered equal.

The problem can be solved with a comparison sort (e.g. by using a suitable comparison operator in QUICKSORT or MERGESORT). Example: comparing dates:

```
DATE-COMPARE($x, y$)
1  **if** $x.year < y.year$ **then return** “smaller”
2  **if** $x.year > y.year$ **then return** “greater”
3  **if** $x.month < y.month$ **then return** “smaller”
4  **if** $x.month > y.month$ **then return** “greater”
5  **if** $x.day < y.day$ **then return** “smaller”
6  **if** $x.day > y.day$ **then return** “greater”
7  **return** “equal”
```

Sometimes it makes sense to handle the input one criterion at a time.

- For example it’s easiest to sort a deck of cards into four piles based on the suits and then each suit separately.

The range of values in the significant criteria is often small when compared to the amount of elements and thus COUNTING-SORT can be used.

There are two different algorithms available for sorting with multiple keys.

- LSD-RADIX-SORT
  - the array is sorted first according to the least significant digit, then the second least significant etc.
  - the sorting algorithm needs to be stable - otherwise the array would be sorted only according to the most significant criterion
  - COUNTING-SORT is a suitable algorithm
  - comparison algorithms are not worth using since they would sort the array with approximately the same amount of effort directly at one go

```
LSD-RADIX-SORT($A, d$)
1  for $i$ := $1$ to $d$ do        (run through the criteria, least significant first)
2      ▷ sort $A$ with a stable sort according to criterion $i$
```

- MSD-RADIX-SORT
  - the array is first sorted according to the most significant digit and then the subarrays with equal keys according to the next significant digit etc.
  - does not require the sorting algorithm to be stable
  - usable when sorting character strings of different lengths
  - checks only as many of the sorting criterions as is needed to determine the order
  - more complex to implement than LSD-RADIX-SORT
    $\Rightarrow$ the algorithm is not given here

The efficiency of RADIX-SORT when using COUNTING-SORT:

- sorting according to one criterion: $\Theta(n + k)$
- amount of different criteria is d
  $\Rightarrow$ total efficiency $\Theta(dn + dk)$
- k is usually constant
  $\Rightarrow$ total efficiency $\Theta(dn)$, or $\Theta(n)$, if d is also constant

RADIX-SORT appears to be a $O(n)$ sorting algorithm with certain assumptions.

Is is better than the comparison sorts in general?

When analyzing the efficiency of sorting algorithms it makes sense to assume that all (or most) of the elements have different values.

- For example INSERTION-SORT is $O(n)$, if all elements are equal.
- If the elements are all different and the size of value range of one criterion is constant k, $k^d \geq n \Rightarrow d \geq log_k\, n$ = $\Theta(\lg n)$
  $\Rightarrow$ RADIX-SORT is $\Theta(dn) = \Theta(n \lg n)$, if we assume that the element values are mostly different from each other.

RADIX-SORT is asymptotically as slow as other good sorting algorithms.

- By assuming a constant d, RADIX-SORT is $\Theta(n)$, but then with large values of n most elements are equal to eachother.

Advantages and disadvantages of RADIX-SORT

Advantages:

- RADIX-SORT is able to compete in efficiency with QUICKSORT for example
  - if the keys are 32-bit numbers and the array is sorted according to 8 bits at a time
    $\Rightarrow k = 2^8$ and $d = 4$
    $\Rightarrow$ COUNTING-SORT is called four times
- RADIX-SORT is well suited for sorting according to keys with multiple parts when the parts of the key have a small value range.
  - e.g. sorting a text file according to the characters on the given columns (cmp. Unix or MS/DOS sort)

Disadvantages:

- COUNTING-SORT requires another array B of n elements where it builds the result and a temp array of k elements.
  $\Rightarrow$ It requires $\Theta(n)$ extra memory which is significantly larger than for example with QUICKSORT and HEAPSORT.

Bucket sort

Let's assume that the keys are within a known range of values and the key values are evenly distributed.

- Each key is just as probable.
- For the sake of an example we'll assume that the key values are between zero and one.
- Let's use n *buckets* $B[0] \dots\ B[n-1]$.

```
BUCKET-SORT($A$)
1  $n$ := $A.length$
2  for $i$ := $1$ to $n$ do                    (go through the elements)
3      INSERT($B[\lfloor n \cdot A[i] \rfloor], A[i]$)   (throw the element into the correct bucket)
4  $k$ := $1$                                  (start filling the array from index 1)
5  for $i$ := $0$ to $n-1$ do                  (go through the buckets)
6      while $B[i]$ not empty do               (empty non-empty buckets...)
7          $A[k]$ := EXTRACT-MIN($B[i]$)       (... by moving the elements, smallest first...)
8          $k$ := $k+1$                        (... into the correct location in the result array)
```

Implementation of the buckets:

- Operations INSERT and EXTRACT-MIN are needed.
  $\Rightarrow$ The bucket is actually a priority queue.
- The size of the buckets varies a lot.
  - usually the amount of elements in the bucket is ≈ 1
  - however it is possible that every element end up in the same bucket

  $\Rightarrow$ an implementation that uses a heap would require $\Theta(n)$ for each bucket,s $\Theta(n^2)$ in total
- On the other hand, the implementation does not need to be very efficient for large buckets since they are rare. $\Rightarrow$ In practise the buckets should be implemented as lists.
  - INSERT links the incoming element to its correct location in the list, Θ(list length) time is used
  - EXTRACT-MIN removes and returns the first element in the list, $\Theta(1)$ time is used

the average efficiency of BUCKET-SORT:

- We assumed the keys are evenly distributed.
  $\Rightarrow$ On average one element falls into each bucket and very rarely a significantly larger amount of elements fall into the same bucket.
- The first **for**-loop runs through all of the elements, $\Theta(n)$.
- The second **for**-loop runs through the buckets, $\Theta(n)$.
- The **while**-loop runs through all of the elements in all of its iterations in total once, $\Theta(n)$.
- INSERT is on average constant time, since there is on average one element in the bucket.
- EXTRACT-MIN is constant time.

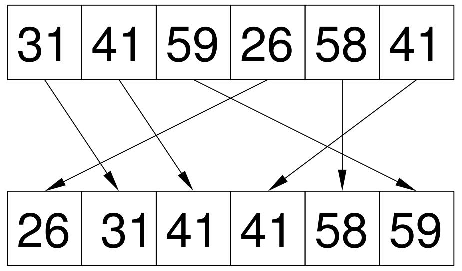
$\Rightarrow$ The total running-time is $\Theta(n)$ on average.

In the slowest case all elements fall into the same bucket in an ascending order.

- INSERT takes a linear amount of time

$\Rightarrow$ The total running-time is $\Theta(n^2)$ in the worst-case.

9.2 How fast can we sort?



Sorting an array actually creates the permutation of its elements where the original array is completely sorted.

- If the elements are all different, the permutation is unique. ⇒ Sorting searches for that permutation from the set of all possible permutations.

For example the functionality of INSERTION-SORT, MERGE-SORT, HEAPSORT and QUICKSORT is based on comparisons between the elements.

- Information about the correct permutation is collected only by comparing the elements together.

What would be the smallest amount of comparisons that is enough to find the correct permutation for sure?

- An array of n elements of different values has $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ i.e. $n!$ permutations.
- The amount of comparisons needed must find the only correct alternative.
- Each comparison $A[i] \leq A[j]$ (or $A[i]$ < $A[j]$) divides the permutations into two groups: those where the order of $A[i]$ and $A[j]$ must be switched and those where the order is correct so...
  - one comparison in enough to pick the right alternative from atmost two
  - two comparisons in enough to pick the right one from atmost four
  - $\ldots$
  - k comparisons in enough to pick the right alternative from atmost 2^k
    $\Rightarrow$ choosing the right one from x alternatives requires at least $\lceil \lg x \rceil$ comparisons

- If the size of the array is n, there are $n!$ permutations
  $\Rightarrow$ At least $\lceil \lg n! \rceil$ comparisons is required
  $\Rightarrow$ a comparison sort algorithm needs to use $\Omega(\lceil \lg n! \rceil)$ time.

How large is $\lceil \lg n! \rceil$?

- $\lceil \lg n! \rceil \geq \lg n! = \Sigma_{k=1}^{n} \lg k \geq \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n} \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{1}{2} n \lg n - \frac{1}{2} n = \Omega(n \lg n) - \Omega(n) = \Omega(n \lg n)$
- on the other hand $\lceil \lg n! \rceil < n \lg n + 1 = O(n \lg n)$
  $\Rightarrow \lceil \lg n! \rceil = \Theta(n \lg n)$

Every comparison sort algorithm needs to use $\Omega(n \lg n)$ time in the slowest case.

- On the other hand HEAPSORT and MERGE-SORT are $O(n \lg n)$ in the slowest case.

  $\Rightarrow$ *In the slowest case sorting based on comparisons between elements is possible in* $\Theta(n \lg n)$ *time, but no faster.*
- HEAPSORT and MERGE-SORT have an optimal asymptotic running-time in the slowest case.
- Sorting is truly asymptotically more time consuming than finding the median value, which can be done in the slowest possible case in $O(n)$.