WS 2007/2008

# Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

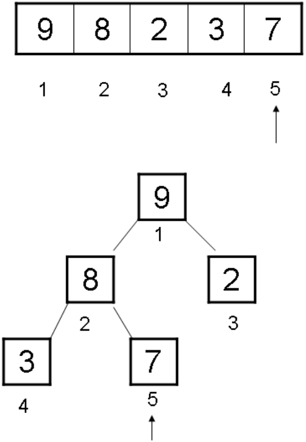
http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

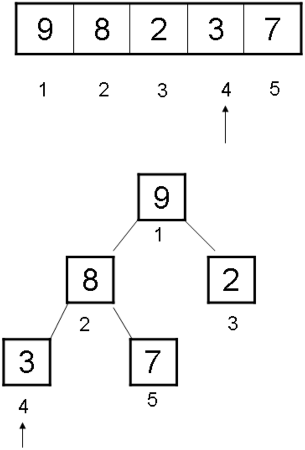
## Heapsort Algorithm:

void HeapSort(key $A[]$, unsigned $n$){
  for $k := n$ downto 1 do // create_heap
    reheap($A, n, k$)
  od
  for $k := n$ downto 1 do // $n\times$ delete_min
    swap $A[1]$ and $A[k]$
    reheap($A, k, 1$)
  od
  for $k := 1$ to $\lfloor n/2 \rfloor$ do // reverse sorted array
    swap $A[k]$ and $A[n-k+1]$
  od
}

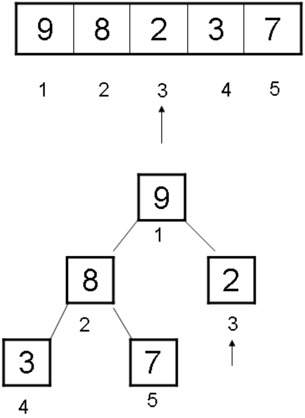
# Animation of Heapsort: Create Heap



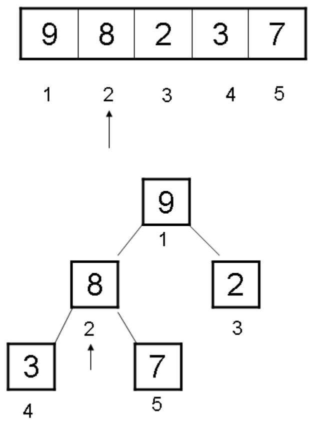
# Animation of Heapsort: Create Heap



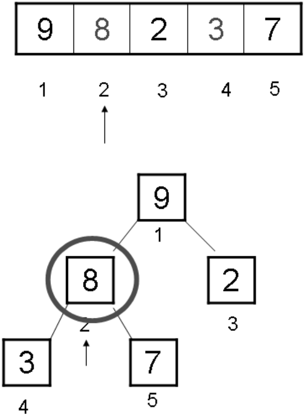
# Animation of Heapsort: Create Heap



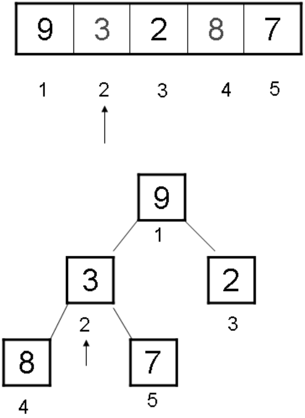
# Animation of Heapsort: Create Heap



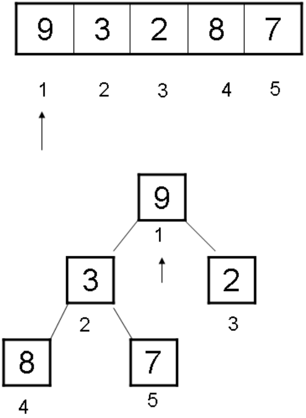
# Animation of Heapsort: Create Heap



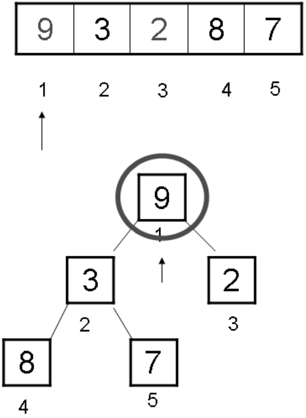
# Animation of Heapsort: Create Heap



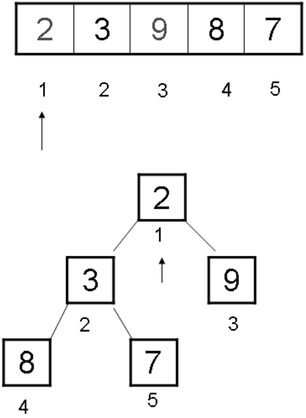
# Animation of Heapsort: Create Heap



# Animation of Heapsort: Create Heap



# Animation of Heapsort: Create Heap



## Heapsort Algorithm:

void HeapSort(key $A[]$, unsigned $n$){
  for $k := n$ downto 1 do // create_heap
    reheap$(A, n, k)$
  od
  for $k := n$ downto 1 do // $n\times$ delete_min
    swap $A[1]$ and $A[k]$
    reheap$(A, k, 1)$
  od
  for $k := 1$ to $\lfloor n/2 \rfloor$ do // reverse sorted array
    swap $A[k]$ and $A[n - k + 1]$
  od
}

For create_heap() we define

$V_{\text{create}}(n) :=$ # comparisons for create_heap()

Here it holds that

$$
\begin{aligned}
V_{\text{create}}(n) &\leq \sum_{i=1}^{n} V_{\text{reheap}}(n,i) \\
&\leq 2\sum_{i=1}^{n}(\lfloor \log n \rfloor - \lfloor \log i \rfloor) \\
&\leq 2\sum_{i=1}^{n}(\log n - \log i + 1) \\
&= 2n\log n + 2n - 2\sum_{i=2}^{n}\log i \\
&\leq^* 2n\log n + 2n - 2n\log n - 2/\ln 2(n-1) \\
&\leq 5n
\end{aligned}
$$

This shows that our upper bound on the complexity of create_heap was too pessimistic. $*$ It holds that

$$2\sum_{i=2}^{n}\log i \geq 2n\log n - 2/\ln 2(n-1)$$

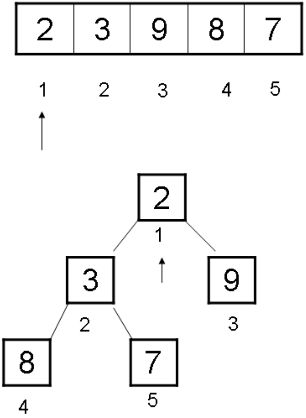
For create_heap() we define
$V_{\text{create}}(n) :=$ # comparisons for create_heap()

Here it holds that

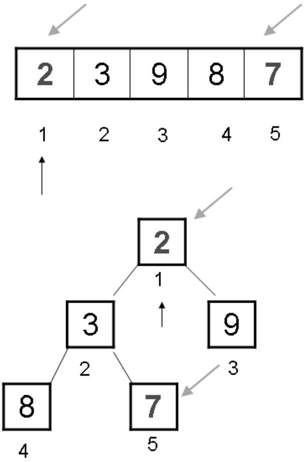
$$\begin{array}{rcl} V_{\text{create}}(n) & \leq & \sum_{i=1}^{n} V_{\text{reheap}}(n,i) \\ & \leq & 2\sum_{i=1}^{n}(\lfloor \log n \rfloor - \lfloor \log i \rfloor) \\ & \leq & 2\sum_{i=1}^{n}(\log n - \log i + 1) \\ & = & 2n\log n + 2n - 2\sum_{i=2}^{n}\log i \\ & \leq^{*} & 2n\log n + 2n - 2n\log n - 2/\ln 2(n-1) \\ & \leq & 5n \end{array}$$

This shows that our upper bound on the complexity of create_heap was too pessimistic. $*$ It holds that
$2\sum_{i=2}^{n}\log i \geq 2n\log n - 2/\ln 2(n-1)$

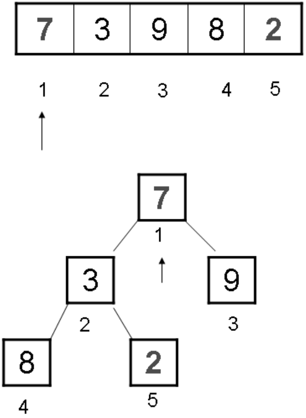
# Animation of Heapsort: Sorting



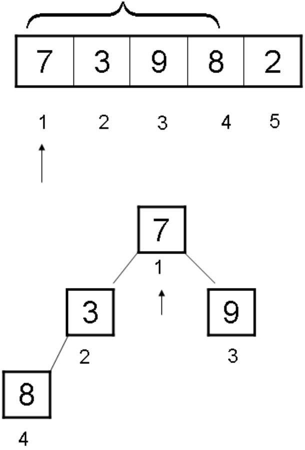
# Animation of Heapsort: Sorting



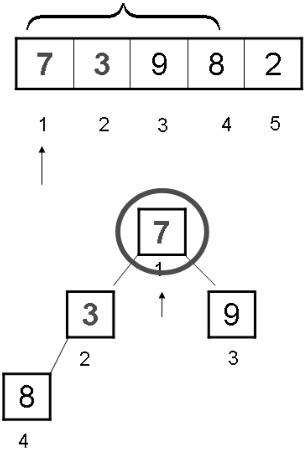
# Animation of Heapsort: Sorting



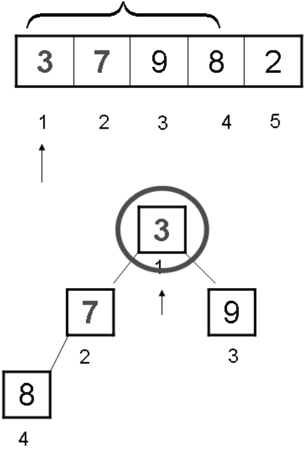
# Animation of Heapsort: Sorting



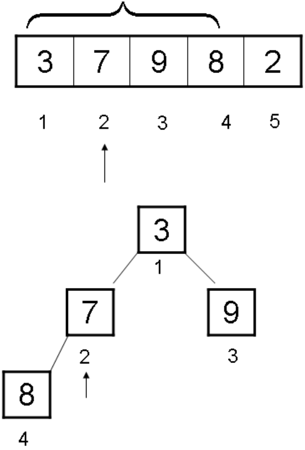
# Animation of Heapsort: Sorting



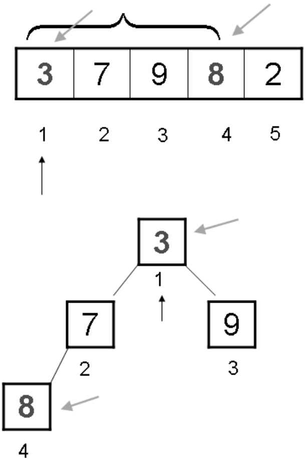
# Animation of Heapsort: Sorting



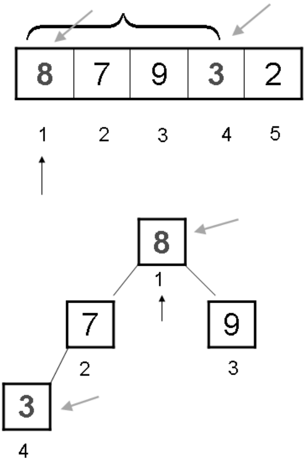
# Animation of Heapsort: Sorting



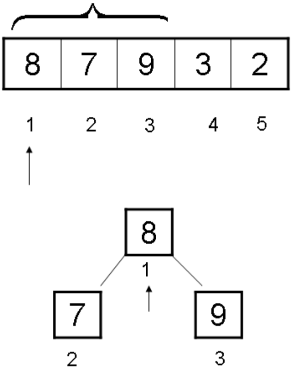
# Animation of Heapsort: Sorting



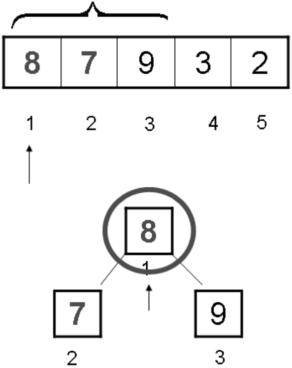
# Animation of Heapsort: Sorting



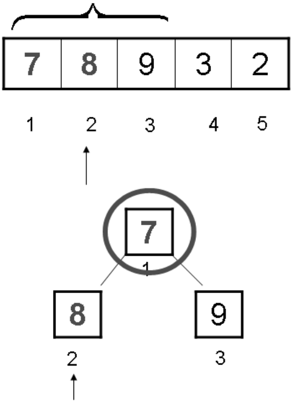
# Animation of Heapsort: Sorting



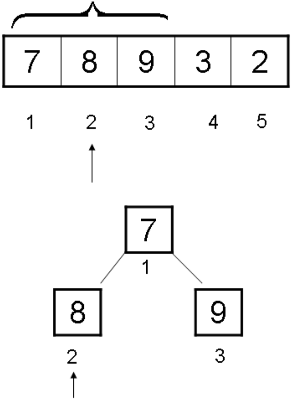
# Animation of Heapsort: Sorting



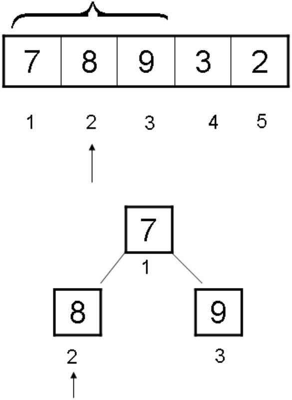
# Animation of Heapsort: Sorting



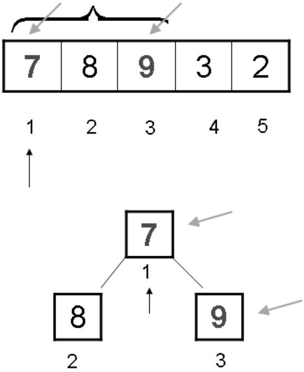
# Animation of Heapsort: Sorting



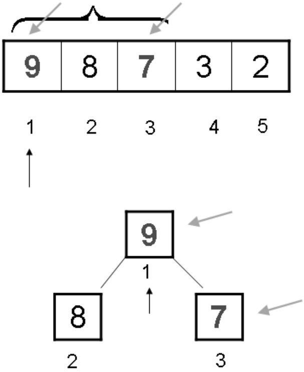
# Animation of Heapsort: Sorting



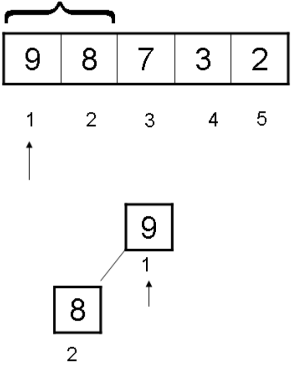
# Animation of Heapsort: Sorting



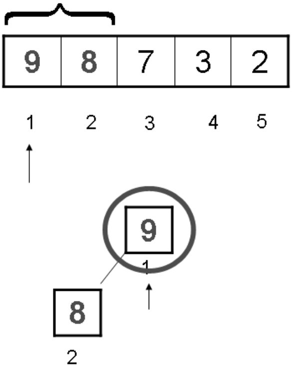
# Animation of Heapsort: Sorting



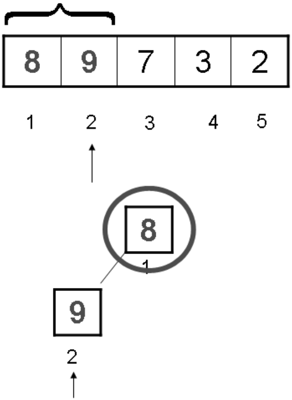
# Animation of Heapsort: Sorting



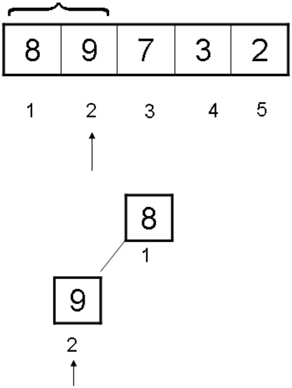
# Animation of Heapsort: Sorting



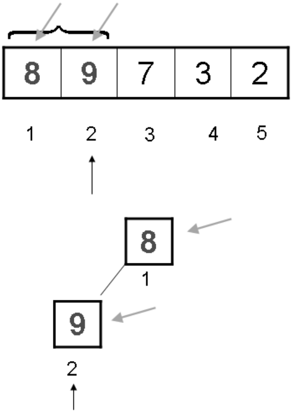
# Animation of Heapsort: Sorting



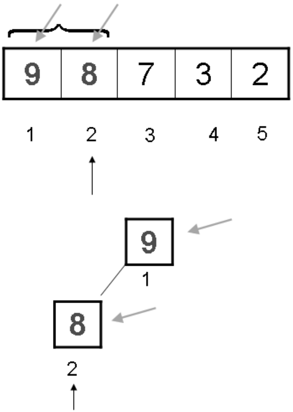
# Animation of Heapsort: Sorting



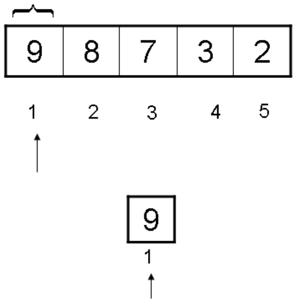
# Animation of Heapsort: Sorting



# Animation of Heapsort: Sorting



# Animation of Heapsort: Sorting



## Heapsort Algorithm:

void HeapSort(key $A[]$, unsigned $n$){
  for $k := n$ downto 1 do // create_heap
    reheap$(A, n, k)$
  od
  for $k := n$ downto 1 do // $n\times$ delete_min
    swap $A[1]$ and $A[k]$
    reheap$(A, k, 1)$
  od
  for $k := 1$ to $\lfloor n/2 \rfloor$ do // reverse sorted array
    swap $A[k]$ and $A[n-k+1]$
  od
}

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:

| | | |
|---|---|---|
| $< p$ | $p$ | $> p$ |

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys$< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.

| | | |
|---|---|---|
| $< p$ | $p$ | $> p$ |

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys$< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys $< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.

| | | |
|---|---|---|
| $< p$ | $p$ | $> p$ |

# 1. Quick Sort

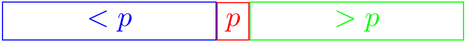
- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys $< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys$< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.

# 1. Quick Sort

- To inductively derive the Quicksort algorithm, we merely need a different way of strengthening the induction hypothesis, as compared to the other sorting algorithms.
- Suppose for the time being that all keys are unique.
- Base case ($n \leq 1$): trivial
- Inductive step:
  - Rearrange the array containing the keys as follows: A pivot element $p$ is selected among the keys. All keys$< p$ in the array are moved to the left, all keys $> p$ are moved to the right.
  - Sort the two subarrays recursively
  - Recombination is trivial – the sorted subarrays only need to be joined to yield the sorted array.



- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.

- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.

- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.
    - We use two pointer $i$ and $j$ moving from $\ell$ to the right, and from $r - 1$ to the left, respectively.
    - When $A[i] > p$ *and* $A[j] < p$ then we swap $A[i]$ and $A[j]$.
    - When $i \geq j$ we move the pivot to the right position.

- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.
  - We use two pointer $i$ and $j$ moving from $\ell$ to the right, and from $r - 1$ to the left, respectively.
  - When $A[i] > p$ *and* $A[j] < p$ then we swap $A[i]$ and $A[j]$.
  - When $i \geq j$ we move the pivot to the right position.

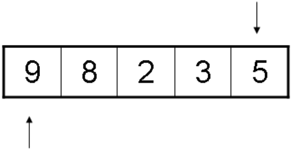
- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.
  - We use two pointer $i$ and $j$ moving from $\ell$ to the right, and from $r-1$ to the left, respectively.
  - When $A[i] > p$ *and* $A[j] < p$ then we swap $A[i]$ and $A[j]$.
  - When $i \geq j$ we move the pivot to the right position.

- The main degree of freedom in the Quicksort algorithm is the choice of the pivot. The median would be ideal in the sense that it would lead to a perfect divide-and-conquer algorithm.
- To save the cost of finding the median we choose the rightmost key in the array as our pivot.
- The rearrangement can be carried out as follows: Suppose we are to rearrange array $A[]$ between the indices $\ell$ and $r$, with respect to pivot $p = A[r]$.
  - We use two pointer $i$ and $j$ moving from $\ell$ to the right, and from $r - 1$ to the left, respectively.
  - When $A[i] > p$ *and* $A[j] < p$ then we swap $A[i]$ and $A[j]$.
  - When $i \geq j$ we move the pivot to the right position.

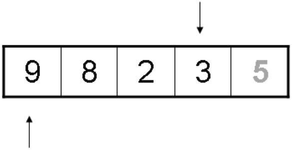
## Rearrangement algorithm:

unsigned partition(key $A[]$, unsigned $\ell, r$){
  unsigned $i := \ell$
  unsigned $j := r - 1$
  unsigned $piv := r$
  while $(i < j)$ do
    while $(A[i] < A[piv] \wedge i < j)$ do $i + +$ od
    while $(A[j] > A[piv] \wedge i < j)$ do $j - -$ od
    if $(i < j)$ then
      swap $A[i]$ and $A[j]$
    else
      swap $A[i]$ and $A[piv]$
    fi
  od
  return i
}

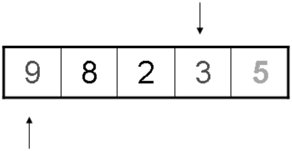
# Animation of Quicksrot



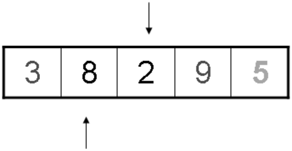
# Animation of Quicksrot



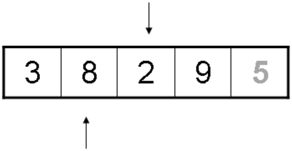
# Animation of Quicksrot



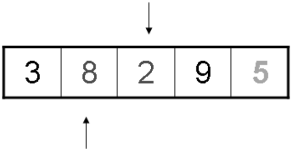
# Animation of Quicksrot



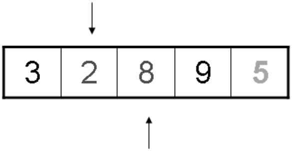
# Animation of Quicksrot



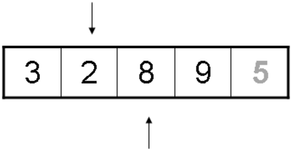
# Animation of Quicksrot



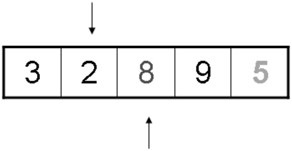
# Animation of Quicksrot



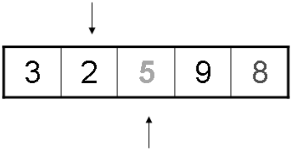
# Animation of Quicksrot



# Animation of Quicksrot



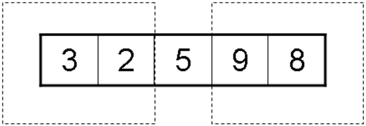
# Animation of Quicksrot



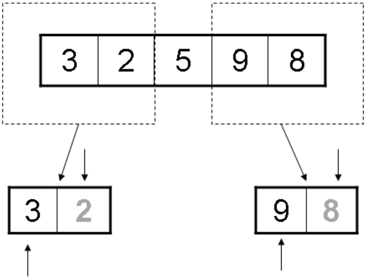
# Animation of Quicksrot

| | | | | |
|---|---|---|---|---|
| 3 | 2 | 5 | 9 | 8 |

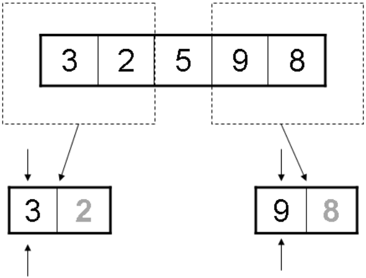
# Animation of Quicksrot



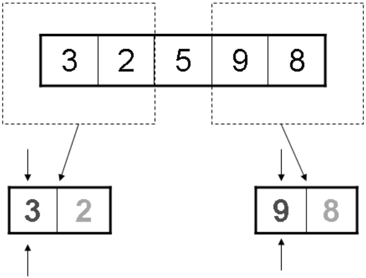
# Animation of Quicksrot



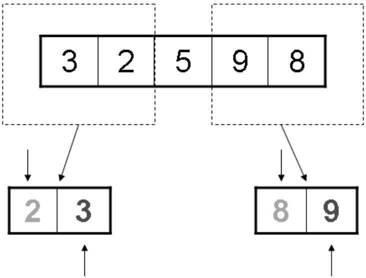
# Animation of Quicksrot



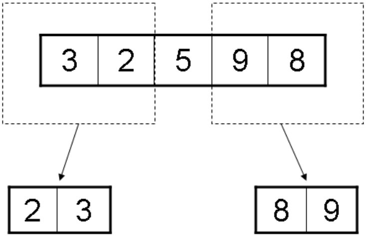
# Animation of Quicksrot



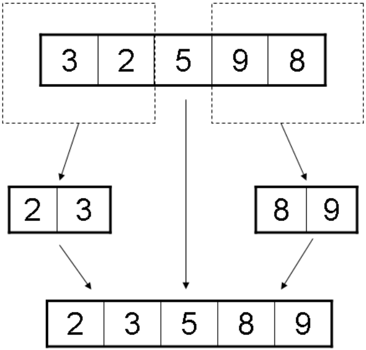
# Animation of Quicksrot



# Animation of Quicksrot



# Animation of Quicksrot



Sorting algorithm:

```
void QuickSort(key A[], unsigned ℓ, r){
  if (ℓ > r) then return
  else piv :=partition(A, ℓ, r)
    QuickSort(A, ℓ, piv − 1)
    QuickSort(A, piv + 1, r)
  fi
}
```

# Complexity:

Definition 1

Let $M$ be a totally ordered set, and let $x \in M$. Then the rank Rank($x$) of element $x$ is equal to $k$ if and only if $|\{x' \in M : x' < x\}| = k - 1$, i.e. if $x$ is the $k$-th smallest element of $M$ with respect to the ordering.

Element $x$ is the median of set $M$ if Rank($x$)= $\lceil \frac{|M|+1}{2} \rceil$.

- The number of comparisons in QuickSort depends on the rank of the pivot.
- All comparisons take place in the partition() function.
- Each call partition($A[], \ell, r$) costs $r - \ell$ comparisons
- Thus, the number $c(n)$ of comparisons in QuickSort can be informally described as

$$"c(n) = (n-1) + c(k-1) + c(n-k)",$$

where $k$ is the rank of the pivot in the *original* call of quicksort. For $c(k-1)$ and $c(n-k)$, different ranks apply.

# Complexity:

Definition 1

Let $M$ be a totally ordered set, and let $x \in M$. Then the rank Rank($x$) of element $x$ is equal to $k$ if and only if $|\{x' \in M : x' < x\}| = k - 1$, i.e. if $x$ is the $k$-th smallest element of $M$ with respect to the ordering.

Element $x$ is the median of set $M$ if Rank($x$)= $\lceil \frac{|M|+1}{2} \rceil$.

- The number of comparisons in QuickSort depends on the rank of the pivot.
- All comparisons take place in the partition() function.
- Each call partition($A[], \ell, r$) costs $r - \ell$ comparisons
- Thus, the number $c(n)$ of comparisons in QuickSort can be informally described as

$$"c(n) = (n-1) + c(k-1) + c(n-k)",$$

where $k$ is the rank of the pivot in the *original* call of quicksort. For $c(k-1)$ and $c(n-k)$, different ranks apply.

# Complexity:

Definition 1

Let $M$ be a totally ordered set, and let $x \in M$. Then the rank Rank($x$) of element $x$ is equal to $k$ if and only if $|\{x' \in M : x' < x\}| = k - 1$, i.e. if $x$ is the $k$-th smallest element of $M$ with respect to the ordering.
Element $x$ is the median of set $M$ if Rank($x$)$= \lceil \frac{|M|+1}{2} \rceil$.

- The number of comparisons in QuickSort depends on the rank of the pivot.
- All comparisons take place in the partition() function.
- Each call partition($A[], \ell, r$) costs $r - \ell$ comparisons
- Thus, the number $c(n)$ of comparisons in QuickSort can be informally described as

$$"c(n) = (n-1) + c(k-1) + c(n-k)",$$

where $k$ is the rank of the pivot in the *original* call of quicksort. For $c(k-1)$ and $c(n-k)$, different ranks apply.

# Complexity:

Definition 1

Let $M$ be a totally ordered set, and let $x \in M$. Then the rank Rank$(x)$ of element $x$ is equal to $k$ if and only if $|\{x' \in M : x' < x\}| = k - 1$, i.e. if $x$ is the $k$-th smallest element of $M$ with respect to the ordering.
Element $x$ is the median of set $M$ if Rank$(x)$= $\lceil \frac{|M|+1}{2} \rceil$.

- The number of comparisons in QuickSort depends on the rank of the pivot.
- All comparisons take place in the partition() function.
- Each call partition$(A[], \ell, r)$ costs $r - \ell$ comparisons
- Thus, the number $c(n)$ of comparisons in QuickSort can be informally described as

$$"c(n) = (n-1) + c(k-1) + c(n-k)",$$

where $k$ is the rank of the pivot in the *original* call of quicksort. For $c(k-1)$ and $c(n-k)$, different ranks apply.

How to find the worst case scenario for the above recurrence relation?

- Consider the tree of recursive calls within QuickSort. Each call QuickSort($A, \ell, r$) gives rise to two new calls QuickSort($A, \ell, piv - 1$) and QuickSort($A, piv + 1, r$).
- The *total* number of comparisons executed in the partition() steps of these two calls (i.e. excluding their recursive sub-calls) is $r - \ell$ and thus independent of the rank of the pivot.
- Hence, the total number of comparisons is only dependent on the depth of the call tree.
- The depth of the call tree is maximized if, in each sub-call, the pivot is either the minimum or the maximum of the elements to be sorted.
- If this is the case all the time, we get the following recurrence relation:

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c(n-1) + 0 & \text{if } n \geq 2 \end{cases}$$

How to find the worst case scenario for the above recurrence relation?

- Consider the tree of recursive calls within QuickSort. Each call QuickSort($A, \ell, r$) gives rise to two new calls QuickSort($A, \ell, piv - 1$) and QuickSort($A, piv + 1, r$).
- The *total* number of comparisons executed in the partition() steps of these two calls (i.e. excluding their recursive sub-calls) is $r - \ell$ and thus independent of the rank of the pivot.
- Hence, the total number of comparisons is only dependent on the depth of the call tree.
- The depth of the call tree is maximized if, in each sub-call, the pivot is either the minimum or the maximum of the elements to be sorted.
- If this is the case all the time, we get the following recurrence relation:

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c(n-1) + 0 & \text{if } n \geq 2 \end{cases}$$

How to find the worst case scenario for the above recurrence relation?

- Consider the tree of recursive calls within QuickSort. Each call QuickSort$(A, \ell, r)$ gives rise to two new calls QuickSort$(A, \ell, piv - 1)$ and QuickSort$(A, piv + 1, r)$.
- The *total* number of comparisons executed in the partition() steps of these two calls (i.e. excluding their recursive sub-calls) is $r - \ell$ and thus independent of the rank of the pivot.
- Hence, the total number of comparisons is only dependent on the depth of the call tree.
- The depth of the call tree is maximized if, in each sub-call, the pivot is either the minimum or the maximum of the elements to be sorted.
- If this is the case all the time, we get the following recurrence relation:

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c(n-1) + 0 & \text{if } n \geq 2 \end{cases}$$

How to find the worst case scenario for the above recurrence relation?

- Consider the tree of recursive calls within QuickSort. Each call QuickSort$(A, \ell, r)$ gives rise to two new calls QuickSort$(A, \ell, piv - 1)$ and QuickSort$(A, piv + 1, r)$.
- The *total* number of comparisons executed in the partition() steps of these two calls (i.e. excluding their recursive sub-calls) is $r - \ell$ and thus independent of the rank of the pivot.
- Hence, the total number of comparisons is only dependent on the depth of the call tree.
- The depth of the call tree is maximized if, in each sub-call, the pivot is either the minimum or the maximum of the elements to be sorted.
- If this is the case all the time, we get the following recurrence relation:

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c(n-1) + 0 & \text{if } n \geq 2 \end{cases}$$

How to find the worst case scenario for the above recurrence relation?

- Consider the tree of recursive calls within QuickSort. Each call QuickSort$(A, \ell, r)$ gives rise to two new calls QuickSort$(A, \ell, piv - 1)$ and QuickSort$(A, piv + 1, r)$.
- The *total* number of comparisons executed in the partition() steps of these two calls (i.e. excluding their recursive sub-calls) is $r - \ell$ and thus independent of the rank of the pivot.
- Hence, the total number of comparisons is only dependent on the depth of the call tree.
- The depth of the call tree is maximized if, in each sub-call, the pivot is either the minimum or the maximum of the elements to be sorted.
- If this is the case all the time, we get the following recurrence relation:

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c(n-1) + 0 & \text{if } n \geq 2 \end{cases}$$

- Expanding this recurrence relation, we see that

$$c(n) = \sum_{i=1}^{n-1} i - 1 = \Theta(n^2)$$

- In the worst case QuickSort is a quadratic-time procedure.
- In the best case in which the pivot always is the median and QuickSort is an ideal divide-and-conquer algorithm, we have

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c\left(\lfloor \frac{n-1}{2} \rfloor\right) + c\left(\lceil \frac{n-1}{2} \rceil\right) & \text{if } n \geq 2 \end{cases}$$

- This recurrence should remind us of MergeSort. Its solution is

$$c(n) = \Theta(n \log n).$$

- QuickSort is one of the most popular algorithms for average-case analysis. Even though we won't do this here, we will mention that, if the pivot ranks are random, independent and uniformly distributed, the expected number of comparisons coincides with the best case.

- Expanding this recurrence relation, we see that

$$c(n) = \sum_{i=1}^{n-1} i - 1 = \Theta(n^2)$$

- In the worst case QuickSort is a quadratic-time procedure.
- In the best case in which the pivot always is the median and QuickSort is an ideal divide-and-conquer algorithm, we have

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c\left(\lfloor \frac{n-1}{2} \rfloor\right) + c\left(\lceil \frac{n-1}{2} \rceil\right) & \text{if } n \geq 2 \end{cases}$$

- This recurrence should remind us of MergeSort. Its solution is

$$c(n) = \Theta(n \log n).$$

- QuickSort is one of the most popular algorithms for average-case analysis. Even though we won't do this here, we will mention that, if the pivot ranks are random, independent and uniformly distributed, the expected number of comparisons coincides with the best case.

- Expanding this recurrence relation, we see that

$$c(n) = \sum_{i=1}^{n-1} i - 1 = \Theta(n^2)$$

- In the worst case QuickSort is a quadratic-time procedure.
- In the best case in which the pivot always is the median and QuickSort is an ideal divide-and-conquer algorithm, we have

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c\left(\lfloor \frac{n-1}{2} \rfloor\right) + c\left(\lceil \frac{n-1}{2} \rceil\right) & \text{if } n \geq 2 \end{cases}$$

- This recurrence should remind us of MergeSort. Its solution is

$$c(n) = \Theta(n \log n).$$

- QuickSort is one of the most popular algorithms for average-case analysis. Even though we won't do this here, we will mention that, if the pivot ranks are random, independent and uniformly distributed, the expected number of comparisons coincides with the best case.

- Expanding this recurrence relation, we see that

$$c(n) = \sum_{i=1}^{n-1} i - 1 = \Theta(n^2)$$

- In the worst case QuickSort is a quadratic-time procedure.
- In the best case in which the pivot always is the median and QuickSort is an ideal divide-and-conquer algorithm, we have

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c\left(\lfloor \frac{n-1}{2} \rfloor\right) + c\left(\lceil \frac{n-1}{2} \rceil\right) & \text{if } n \geq 2 \end{cases}$$

- This recurrence should remind us of MergeSort. Its solution is

$$c(n) = \Theta(n \log n).$$

- QuickSort is one of the most popular algorithms for average-case analysis. Even though we won't do this here, we will mention that, if the pivot ranks are random, independent and uniformly distributed, the expected number of comparisons coincides with the best case.

- Expanding this recurrence relation, we see that

$$c(n) = \sum_{i=1}^{n-1} i - 1 = \Theta(n^2)$$

- In the worst case QuickSort is a quadratic-time procedure.
- In the best case in which the pivot always is the median and QuickSort is an ideal divide-and-conquer algorithm, we have

$$c(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ (n-1) + c\left(\lfloor \frac{n-1}{2} \rfloor\right) + c\left(\lceil \frac{n-1}{2} \rceil\right) & \text{if } n \geq 2 \end{cases}$$

- This recurrence should remind us of MergeSort. Its solution is

$$c(n) = \Theta(n \log n).$$

- QuickSort is one of the most popular algorithms for average-case analysis. Even though we won't do this here, we will mention that, if the pivot ranks are random, independent and uniformly distributed, the expected number of comparisons coincides with the best case.

## 2. Conclusion

- SelectionSort ($O(n^2)$)
- InsertionSort ($O(n^2)$ using linear search, $O(n\log(n))$ using binary search)
- MergeSort ($O(n\log(n))$ using binary search)
- QuickSrot ($O(n\log(n))$ using binary search)
- Are there more efficient algorithms for sorting ???

## 2. Conclusion

- SelectionSort ($O(n^2)$)
- InsertionSort ($O(n^2)$ using linear search, $O(n \log(n))$ using binary search)
- MergeSort ($O(n \log(n))$ using binary search)
- QuickSrot ($O(n \log(n))$ using binary search)
- Are there more efficient algorithms for sorting ???

## 2. Conclusion

- SelectionSort ($O(n^2)$)
- InsertionSort ($O(n^2)$ using linear search, $O(n\log(n))$ using binary search)
- MergeSort ($O(n\log(n))$ using binary search)
- QuickSrot ($O(n\log(n))$ using binary search)
- Are there more efficient algorithms for sorting ???

## 2. Conclusion

- SelectionSort ($O(n^2)$)
- InsertionSort ($O(n^2)$ using linear search, $O(n\log(n))$ using binary search)
- MergeSort ($O(n\log(n))$ using binary search)
- QuickSrot ($O(n\log(n))$ using binary search)
- Are there more efficient algorithms for sorting ???

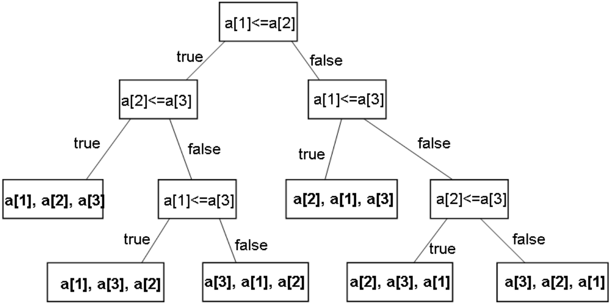
## 2. Conclusion

- SelectionSort ($O(n^2)$)
- InsertionSort ($O(n^2)$ using linear search, $O(n\log(n))$ using binary search)
- MergeSort ($O(n\log(n))$ using binary search)
- QuickSrot ($O(n\log(n))$ using binary search)
- Are there more efficient algorithms for sorting ???

## 3. Lower Bounds for Decision Trees

### Definition 2

A decision tree is a binary tree in which each internal node is annotated by a comparison of two elements. The leaves of the decision tree are annotated by the respective permutations that will put an input sequence into sorted order.



**Theorem 4**

*Any decision tree that sorts $n$ elements has height $\Omega(n \log(n))$.*

**Proof.**

- To sort $n$ elements a decision tree needs $n!$ leaves.
- For the height of the decision tree holds: $h \geq \log(n!)$.
- Since $n! \geq (\frac{n}{2})^{n/2}$, we obtain:

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}(\log(n) - 1) \geq \frac{n}{4}\log(n)$$

  for $n \geq 4$.
- Thus, we need at least $\frac{n}{4}\log(n)$ comparsions, in other words:

$$h = \Omega(n \log(n))$$

□

**Theorem 5**

*MergeSort and HeapSort are asymptotically optimal comparison*

**Theorem 4**

*Any decision tree that sorts $n$ elements has height $\Omega(n \log(n))$.*

**Proof.**

- To sort $n$ elements a decision tree needs $n!$ leaves.
- For the height of the decision tree holds: $h \geq \log(n!)$.
- Since $n! \geq (\frac{n}{2})^{n/2}$, we obtain:

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}(\log(n) - 1) \geq \frac{n}{4}\log(n)$$

  for $n \geq 4$.
- Thus, we need at least $\frac{n}{4}\log(n)$ comparsions, in other words:

$$h = \Omega(n \log(n))$$

□

**Theorem 5**

*MergeSort and HeapSort are asymptotically optimal comparison*

**Theorem 4**

*Any decision tree that sorts $n$ elements has height $\Omega(n\log(n))$.*

**Proof.**

- To sort $n$ elements a decision tree needs $n!$ leaves.
- For the height of the decision tree holds: $h \geq \log(n!)$.
- Since $n! \geq (\frac{n}{2})^{n/2}$, we obtain:

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}(\log(n) - 1) \geq \frac{n}{4}\log(n)$$

  for $n \geq 4$.
- Thus, we need at least $\frac{n}{4}\log(n)$ comparsions, in other words:

$$h = \Omega(n\log(n))$$

□

**Theorem 5**

*MergeSort and HeapSort are asymptotically optimal comparison*

## Theorem 4

*Any decision tree that sorts $n$ elements has height $\Omega(n\log(n))$.*

## Proof.

- To sort $n$ elements a decision tree needs $n!$ leaves.
- For the height of the decision tree holds: $h \geq \log(n!)$.
- Since $n! \geq (\frac{n}{2})^{n/2}$, we obtain:

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}(\log(n)-1) \geq \frac{n}{4}\log(n)$$

  for $n \geq 4$.
- Thus, we need at least $\frac{n}{4}\log(n)$ comparsions, in other words:

$$h = \Omega(n\log(n))$$

□

## Theorem 5

*MergeSort and HeapSort are asymptotically optimal comparison*

# Chapter III Data Structures

- Remember from the chapter on sorting that we are dealing with a set of data elements. Each data element is uniquely defined by a key value of type 'key'. Additionally, it has a data contents of type 'data'
- The key values are contained in a (typically large) universe $U$.
- For the sake of representational simplicity we pretend that $U = \{1, 2, \ldots, N\}$.
- Let $n$ be the maximum number of keys in a data set, and let $m$ be the current size of the set. Hence:

$$m \leq n \leq N$$

# Chapter III Data Structures

- Remember from the chapter on sorting that we are dealing with a set of data elements. Each data element is uniquely defined by a key value of type 'key'. Additionally, it has a data contents of type 'data'
- The key values are contained in a (typically large) universe $U$.
- For the sake of representational simplicity we pretend that $U = \{1, 2, \ldots, N\}$.
- Let $n$ be the maximum number of keys in a data set, and let $m$ be the current size of the set. Hence:

$$m \leq n \leq N$$

# Chapter III Data Structures

- Remember from the chapter on sorting that we are dealing with a set of data elements. Each data element is uniquely defined by a key value of type 'key'. Additionally, it has a data contents of type 'data'
- The key values are contained in a (typically large) universe $U$.
- For the sake of representational simplicity we pretend that $U = \{1, 2, \ldots, N\}$.
- Let $n$ be the maximum number of keys in a data set, and let $m$ be the current size of the set. Hence:

$$m \leq n \leq N$$

# Chapter III Data Structures

- Remember from the chapter on sorting that we are dealing with a set of data elements. Each data element is uniquely defined by a key value of type 'key'. Additionally, it has a data contents of type 'data'
- The key values are contained in a (typically large) universe $U$.
- For the sake of representational simplicity we pretend that $U = \{1, 2, \ldots, N\}$.
- Let $n$ be the maximum number of keys in a data set, and let $m$ be the current size of the set. Hence:

$$m \leq n \leq N$$

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

- Linear storage in an array, in random order (advantage: random access in constant time)
- Linear storage in an array, in sorted order (additional advantage: search in logarithmic time)
- Linear storage in a linked list (advantage: deletion of individual elements in constant time)
- Heap (disadvantage: search takes linear time)

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

- Linear storage in an array, in random order (advantage: random access in constant time)
- Linear storage in an array, in sorted order (additional advantage: search in logarithmic time)
- Linear storage in a linked list (advantage: deletion of individual elements in constant time)
- Heap (disadvantage: search takes linear time)

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

- Linear storage in an array, in random order (advantage: random access in constant time)
- Linear storage in an array, in sorted order (additional advantage: search in logarithmic time)
- Linear storage in a linked list (advantage: deletion of individual elements in constant time)
- Heap (disadvantage: search takes linear time)

## Definition 6

A dictionary is a data structure for storing a data set that is equipped with the following operations:

1. data is_delement(key $k$)
2. insert(data $x$, key $k$)
3. delete(key $k$)

We have already seen several basic methods of storing data sets of the mentioned type:

- Linear storage in an array, in random order (advantage: random access in constant time)
- Linear storage in an array, in sorted order (additional advantage: search in logarithmic time)
- Linear storage in a linked list (advantage: deletion of individual elements in constant time)
- Heap (disadvantage: search takes linear time)

## 1. Binary Search Trees

In this section we will, once again, use trees to store data elements. Heaps are not too well suited as dictionary structures because searching arbitrary key values cannot be done efficiently. Imagine searching for the maximum key value. Starting at the root, we do not know which branch to follow. So in the worst case the entire tree has to be traversed.

This problem shall now be addressed. Suppose again that all key values are unique.

**Definition 7**

A binary tree whose vertices are annotated with key values satisfies the search tree condition iff, for every vertex $v$, the key stored in $v$ is greater than all keys stored in $v$'s left subtree and less than all keys stored in $v$'s right subtree.

## 1. Binary Search Trees

In this section we will, once again, use trees to store data elements. Heaps are not too well suited as dictionary structures because searching arbitrary key values cannot be done efficiently. Imagine searching for the maximum key value. Starting at the root, we do not know which branch to follow. So in the worst case the entire tree has to be traversed.
This problem shall now be addressed. Suppose again that all key values are unique.

### Definition 7

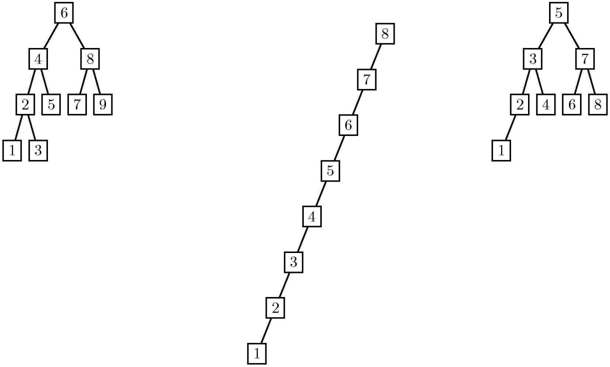
A binary tree whose vertices are annotated with key values satisfies the search tree condition iff, for every vertex $v$, the key stored in $v$ is greater than all keys stored in $v$'s left subtree and less than all keys stored in $v$'s right subtree.

## Definition 8

A binary search tree is an undirected, rooted binary tree whose vertices are annotated with key values in such a way that the search tree condition is satisfied.

The tree is stored based on records: Each vertex $v$ is associated to a record with the fields $v.key$ (key value), $v.data$ (data contents), $v.left$ (left child) and $v.right$ (right child).

**Example:**



## 1.1 Operations on Binary Search Trees

The main operations defined for binary search trees are

- is_element() (aka "find")
- insert()
- delete()

### 1.1.1 is_element

A key $k$ is given. If $k$ is contained in the tree then its associated data contents is output and the global variable $location$ is set to point to the vertex annotated with $k$. Otherwise "NULL" is emitted and $location$ is set to the node at which the search has ended. The procedure is essentially a binary search, thanks to the search tree condition.

## 1.1 Operations on Binary Search Trees

The main operations defined for binary search trees are

- is_element() (aka "find")
- insert()
- delete()

### 1.1.1 is_element

A key $k$ is given. If $k$ is contained in the tree then its associated data contents is output and the global variable $location$ is set to point to the vertex annotated with $k$. Otherwise "NULL" is emitted and $location$ is set to the node at which the search has ended. The procedure is essentially a binary search, thanks to the search tree condition.

### 1.1 Operations on Binary Search Trees

The main operations defined for binary search trees are

- is_element() (aka "find")
- insert()
- delete()

#### 1.1.1 is_element

A key $k$ is given. If $k$ is contained in the tree then its associated data contents is output and the global variable $location$ is set to point to the vertex annotated with $k$. Otherwise "NULL" is emitted and $location$ is set to the node at which the search has ended. The procedure is essentially a binary search, thanks to the search tree condition.

## 1.1 Operations on Binary Search Trees

The main operations defined for binary search trees are

- is_element() (aka "find")
- insert()
- delete()

### 1.1.1 is_element

A key $k$ is given. If $k$ is contained in the tree then its associated data contents is output and the global variable $location$ is set to point to the vertex annotated with $k$. Otherwise "NULL" is emitted and $location$ is set to the node at which the search has ended. The procedure is essentially a binary search, thanks to the search tree condition.

## 1.1 Operations on Binary Search Trees

The main operations defined for binary search trees are

- is_element() (aka "find")
- insert()
- delete()

### 1.1.1 is_element

A key $k$ is given. If $k$ is contained in the tree then its associated data contents is output and the global variable $location$ is set to point to the vertex annotated with $k$. Otherwise "NULL" is emitted and $location$ is set to the node at which the search has ended. The procedure is essentially a binary search, thanks to the search tree condition.

## Algorithm:

```
data is_element(key k){
  v:=root of the tree
  while (v is not a leaf) do
    if (v.key = k) then location:=v; return v.data
    elsif (v.key > k) then
      if (v.left ≠ NULL) then v := v.left
      else location :=NULL; return NULL;
    else
      if (v.right ≠ NULL) then v := v.right
      else location :=NULL; return NULL;
    fi
  od
  location := v
  if (v.key = k) then return v.data
  else return NULL
  fi
}
```

### 1.1.2 insert

First, is_element() is executed. Suppose, the key value $k$ to be inserted is not yet contained in the tree. Then the search ends a vertex (leaf or vertex with only one child) which is stored in $location$. This is the position where a new leaf containing $k$ should be added.

**Algorithm**

void insert(key $k$, data $d$){
   if (is_element($k$)=NULL) then
      $v$:=new vertex
      $v.key := k$; $v.data := d$
      if ($location.key > k$) then $location.left := v$
      else $location.right := v$
      fi
   fi
}

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:
   - i We search for vertex $w$ with the minimal key in $v$'s right subtree.
   - ii We swap $v$ and $w$.
   - iii We delete $v$ in its new position. There it has at most one child (see cases 1. and 2. above).

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:
   i We search for vertex $w$ with the minimal key in $v$'s right subtree.
   ii We swap $v$ and $w$.
   iii We delete $v$ in its new position. There it has at most one child (see cases 1. and 2. above).

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:
   - i We search for vertex $w$ with the minimal key in $v$'s right subtree.
   - ii We swap $v$ and $w$.
   - iii We delete $v$ in its new position. There it has at most one child (see cases 1. and 2. above).

### 1.1.3 delete

First, is_element is executed. Suppose it leads to a vertex $v := location$ containing $k$. There are three possible cases for the result:

1. $v$ is a leaf. In this case it can simply be removed.
2. $v$ has exactly one child. Then we can replace $v$ by its child and are done.
3. $v$ has two children. Removing $v$ will disconnect the tree. We perform the following steps to take care of this case:
   i We search for vertex $w$ with the minimal key in $v$'s right subtree.
   ii We swap $v$ and $w$.
   iii We delete $v$ in its new position. There it has at most one child (see cases 1. and 2. above).

## Algorithm

void delete(key $k$){
  if (is_element($k$)=NULL) then return
  $v := location$
  if ($v$ is a leaf) then remove $v$
  elsif ($v.left$=NULL od $v.right$=NULL) then
    replace $v$ by its child
  else
    $w := v.right$
    while ($w.left \neq$NULL) do $w := w.left$ od
    swap $v$ and $w$
    delete $v$ at its new position
  fi
}

### 1.1.4 Complexities

In all three algorithms the complexity is governed by the number of steps taken by the top-down traversal of the tree. In the worst case, a longest path from the root to a leaf has to be traversed. This means, the time complexity is $O(h)$, where $h$ is the height of the tree.

If the tree is degenerate (i.e. consists of one long path), $h = \Theta(n)$.

In the good case of a balanced search tree, we have $h = O(\log n)$.

In the following sections we will see how to force search trees into a balanced shape.

Some trivia:

### 1.1.4 Complexities

In all three algorithms the complexity is governed by the number of steps taken by the top-down traversal of the tree. In the worst case, a longest path from the root to a leaf has to be traversed. This means, the time complexity is $O(h)$, where $h$ is the height of the tree.
If the tree is degenerate (i.e. consists of one long path), $h = \Theta(n)$.
In the good case of a balanced search tree, we have $h = O(\log n)$.
In the following sections we will see how to force search trees into a balanced shape.

Some trivia:

### 1.1.4 Complexities

In all three algorithms the complexity is governed by the number of steps taken by the top-down traversal of the tree. In the worst case, a longest path from the root to a leaf has to be traversed. This means, the time complexity is $O(h)$, where $h$ is the height of the tree.

If the tree is degenerate (i.e. consists of one long path), $h = \Theta(n)$.

In the good case of a balanced search tree, we have $h = O(\log n)$.

In the following sections we will see how to force search trees into a balanced shape.

Some trivia:

- If the search tree was generated by $n$ insert-operations in random order (with all possible permutations equally likely), the expected height is $\Theta(\log n)$.
- If all possible tree shapes are equally likely, the expected height is $\Theta(\sqrt{n})$.

### 1.1.4 Complexities

In all three algorithms the complexity is governed by the number of steps taken by the top-down traversal of the tree. In the worst case, a longest path from the root to a leaf has to be traversed. This means, the time complexity is $O(h)$, where $h$ is the height of the tree.
If the tree is degenerate (i.e. consists of one long path), $h = \Theta(n)$.
In the good case of a balanced search tree, we have $h = O(\log n)$.
In the following sections we will see how to force search trees into a balanced shape.
Some trivia:

- If the search tree was generated by $n$ insert-operations in random order (with all possible permutations equally likely), the expected height is $\Theta(\log n)$.
- If all possible tree shapes are equally likely, the expected height is $\Theta(\sqrt{n})$.