Sorting

Data Structures and Algorithms
Andrei Bulatov

Sorting Problem

The Sorting Problem

Instance:

A sequence of n numbers $\langle a_1, \ldots, a_n \rangle$

Objective:

A permutation (reordering) $\langle a'_1, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le \ldots \le a'_n$

The numbers are called keys

Insertion Sort

The most natural sorting algorithm

Input: `array A of length n`

Output: `sorted array A`

Method:

```
for  j=2 to n do
  set key:=A[j]
  set i:=j-1
  while i>0 and A[i]>key do
    set A[i+1]:= A[i],  i:=i-1
    set A[i+1]:=key
  endwhile
endfor
```

Insert A[j] into sorted sequence A[1…j-1]

Insertion Sort: Soundness

Insertion Sort consists of a single loop

We use technique called loop invariant that is a property P such that

(*Initialization*) it is true before the first iteration of the loop

(*Maintenance*) if it is true before an iteration of the loop, it remains true before the next iteration

(*Termination*) when the loop terminates, the property helps to establish the correctness of the algorithm

Invariant for Insertion Sort:

At the start of each iteration of the for loop, the subarray A[1…j – 1] consists of the elements originally in A[1…j – 1] but in sorted order

Insertion Sort: Soundness

(*Initialization*)

initially j = 2, so the subarray of interest contains only one element, A[1]. The invariant is true

(*Maintenance*)

Suppose the property is true before iteration j of the loop, i.e. the array A[1…j – 1] is sorted.

If $A[i-1] \leq A[j] < A[i]$, then all the elements A[i], …, A[j – 1] are moved to the next position, and A[j] is inserted in place of A[i], maintaining the order

(*Termination*)

Obvious. When the loop terminates, all the A[1…n] elements are properly ordered.

Insertion Sort: Running Time

Input: `array A of length n`

Output: `sorted array A`

Method:

`for j=2 to n do`	n times
`  set key:=A[j]`	1
`  set i:=j-1`	1
`  while i>0 and A[i]>key do`	j – 1 times
`    set A[i+1]:= A[i], i:=i-1`	1
`    set A[i+1]:=key`	1
`  endwhile`	
`endfor`	

$$\sum_{j=2}^{n}(2+3(j-1)) = 3\sum_{j=2}^{n} j-(n-1) = 3\frac{n(n-1)}{2}-n-2 \; = O(n^2)$$

MergeSort, Divide and Conquer

Recursive algorithms: Call themselves on subproblem

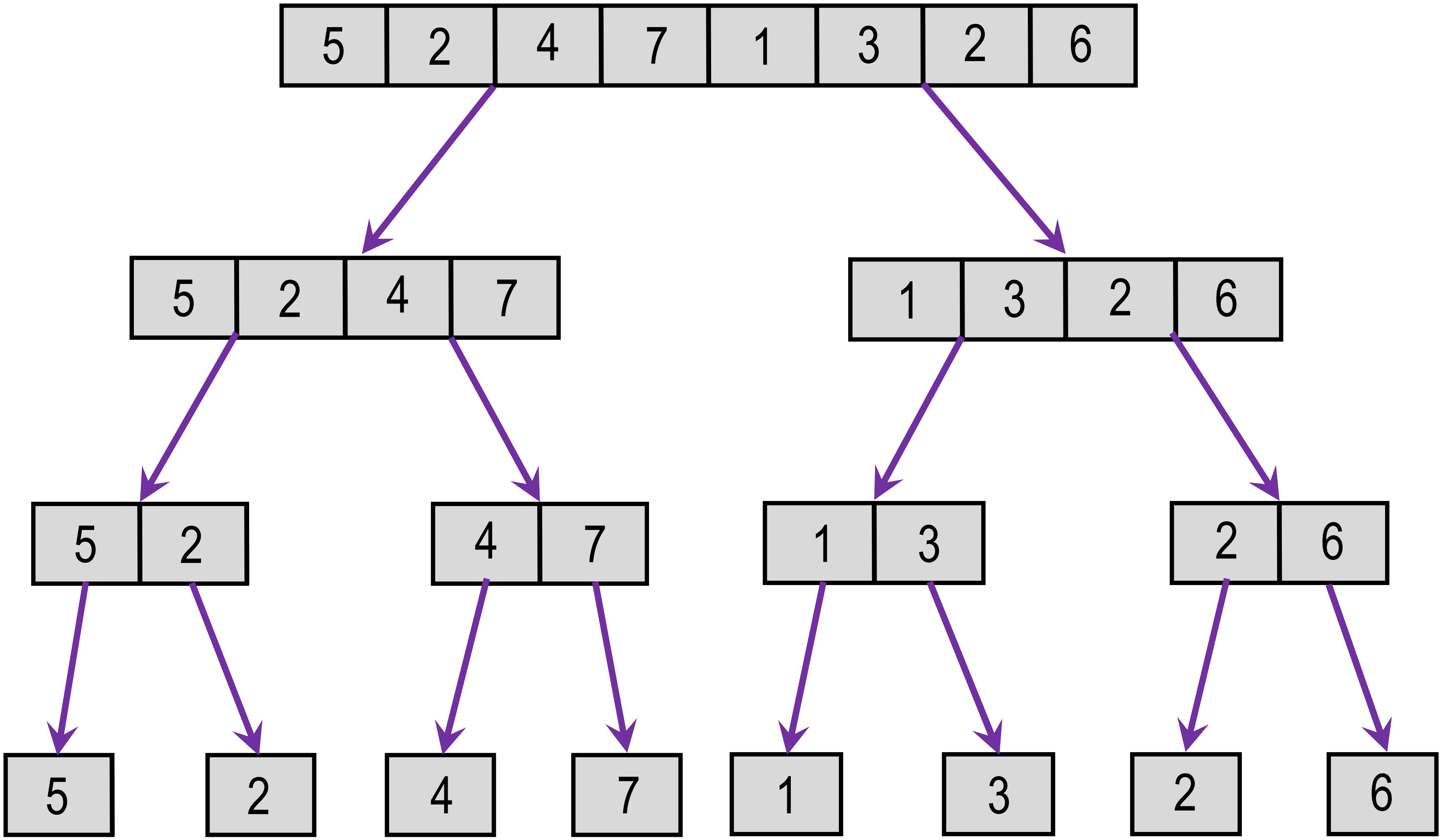
Divide and Conquer algorithms:

- Split a problem into subproblems (divide)
- Solve subproblems recursively (conquer)
- Combine solutions to subproblems (combine)

MergeSort

- Divide: Split a given sequence into halves
- Conquer: By calling itself sort the two halves
- Combine: Merge the two sorted arrays into one

MergeSort



MergeSort

MergeSort(A,p,r)

Input: `array A, positions p,r`

Output: `array A such that entries A[p],…,A[r] are sorted`

Method:

```
if  p<r then do
    set q:=⌊(p+r)/2⌋
    MergeSort(A,p,q)
    MergeSort(A,q+1,r)
    Merge(A,p,q,r)
endif
```

Merge

The Merge procedure is applied to array A and three positions p, q, r in this array

Assume

$p \le q < r$

A[p], …, A[q] and A[q+1], …, A[r] are ordered

Outputs ordered sequence in positions A[p], …, A[r]

This sequence is generated by comparing the two elements on the top of subarrays and moving the smaller one

Merge

```
Merge(A,p,q,r)
set u:=q-p+1,
set v:=r-q
set L[1...u]:=A[p...q]
set R[1...v]:=A[q+1...r]
set i:=1,  j:=1
for k=p to r do
    if l[i]≤R[j] then
       set A[k]:=L[i],  i:=i+1
    else
       set A[k]:=R[j],  j:=j+1
endfor
```

```
set L[u+1]:=∞
set R[v+1]:=∞
```

sentinel cards

Merge: Soundness

Invariant for Merge:

At the start of each iteration of the **for** loop, the subarray A[p…k-1] contains the k – p smallest elements of L[1…u+1] and R[1…v+1] in sorted order.

Moreover, L[i] and R[j] are the smallest elements of the corresponding arrays that have not been copied to A

(*Initialization*)

Initially k = p, so the subarray A[p…k – 1] is empty.

It contains the k – p = 0 smallest elements of L and R.

Since i = j = 1, both L[i] and R[j] are the smallest elements in the corresponding arrays.

The invariant is true

Merge: Soundness

(*Maintenance*)

Suppose the property is true before iteration k of the loop.

If $L[i] \leq R[j]$, then L[i} is the smallest element not yet copied into A.

Since $A[p \ldots k-1]$ contains the $k-p$ smallest elements, after the iteration $L[i]$ is copied into $A[k]$, and $A[p \ldots k]$ contains the $k-p+1$ smallest elements.

New top elements of L and R are clearly the smallest ones

After incrementing k the loop invariant is true again

If $L[i] < R[j]$ the argument is similar

(*Termination*)

Obvious. When the loop terminates, $A[p \ldots r]$ contains the $k-p$ smallest elements from L and R, that is all but the sentinels.

Example

5	2	4	7	1	3	2	6

MergeSort: Soundness

Theorem

MergeSort returns a sorted array

Proof

Follows from the soundness of Merge.

Finish the proof yourself

QED

MergeSort: Running Time

The running time of Merge when applied to two arrays of total size n is $\Theta(n)$

The running time, $T(n)$, of MergeSort is

$$T(n) = Cn + T(n/2) + T(n/2) + Dn$$

divide recursion Merge

If $n = 1$ then $T(1) = C$

Recursion tree

MergeSort: Soundness

Recursion tree

There are 2^i nodes on level i

Each node requires $\frac{Cn + Dn}{2^i}$ work

Total work on each level: (C+D)n

There are log n levels

Theorem

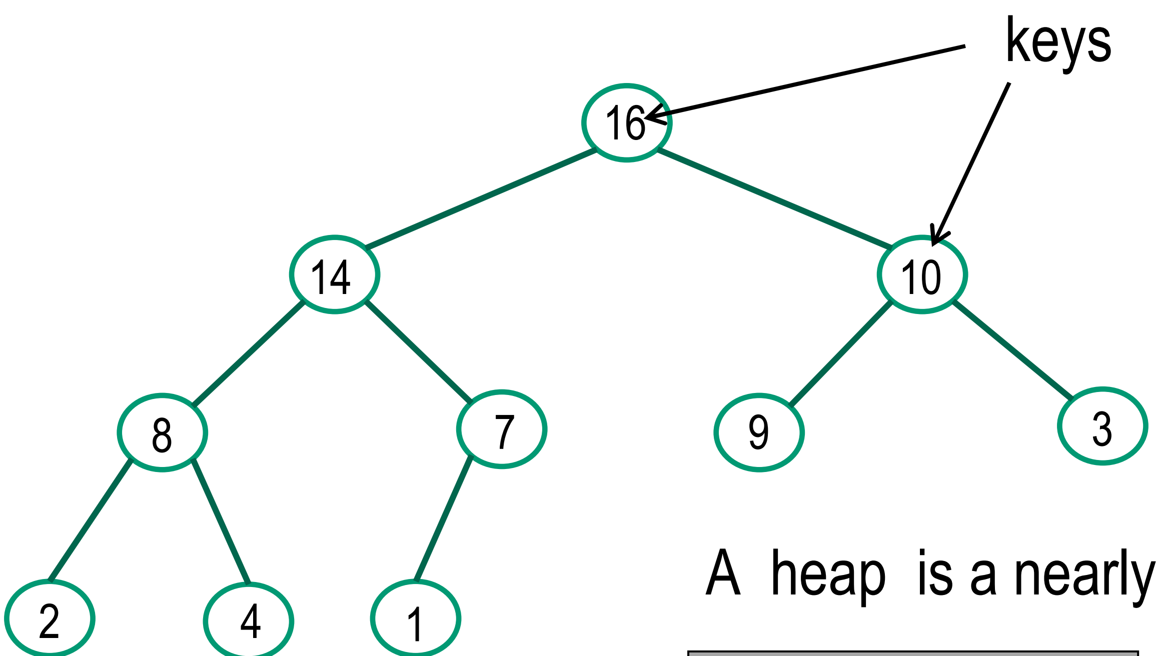
The running time of MergeSort is Θ(n log n)

Heaps

The main setback of the InsertionSort is that to insert it needs to scan a substantial part of the array.

Can it be sped up?

Yes! Using binary trees --- heaps



A heap is a nearly complete binary tree

Recall trees etc.

Heap Property

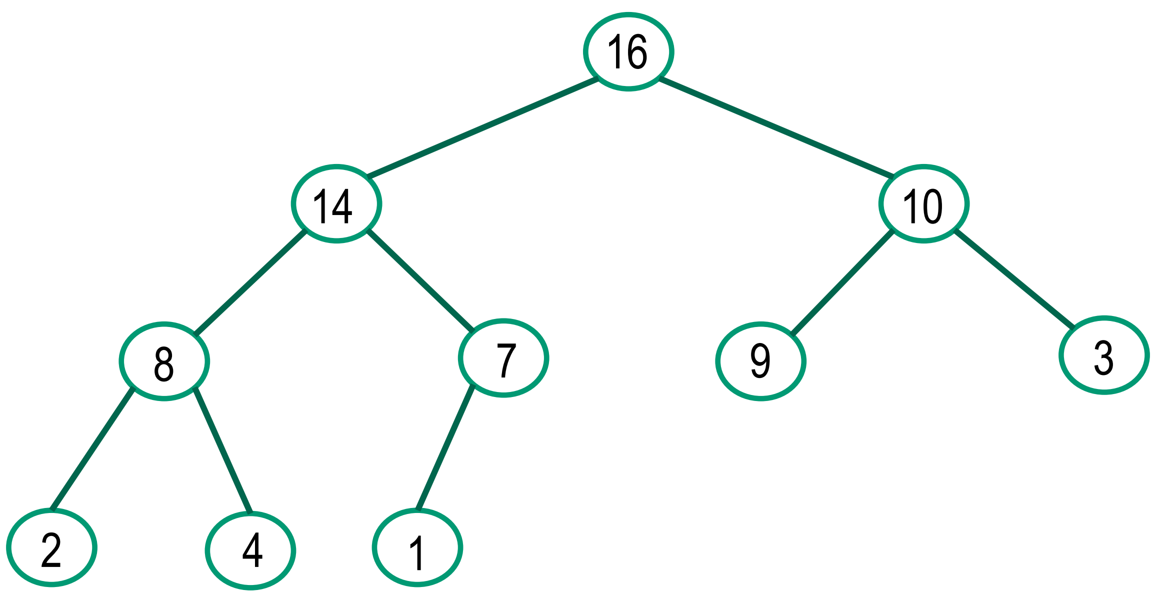
Let Parent(i) denote the parent of the vertex i

Max-Heap Property:

$$\text{Key}(\text{Parent}(i)) \geq \text{Key}(i) \quad \text{for all } i$$

Min-Heap Property:

$$\text{Key}(\text{Parent}(i)) \leq \text{Key}(i) \quad \text{for all } i$$



Heaps

Nearly complete binary tree means that the length of any path from the root to a leaf can vary by at most one

The height of a vertex i is the length of the longest simple downward path from i

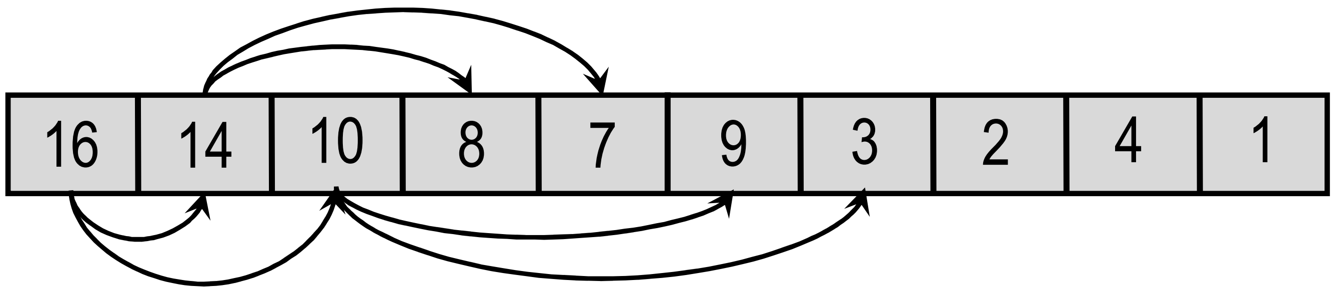
Therefore the height of the root is around log n

Heap Operations

	Goal running time
Creating a max-heap	O(n)
Accessing the minimal element (root)	O(1)
Inserting an element	O(log n)
Deleting an element	O(log n)

Implementing Heaps and Operations

Heap can be implemented by an array



Children:

leftChild(i) = 2i

rightChild(i) = 2i + 1

Parent: parent(i) = $\lfloor i / 2 \rfloor$

Length: length(H) = the number of elements in H

Insertion

Insert(H,key)

```
set n:=length(n),
set H[n+1]:=key
HeapifyUp(H,n+1)
```

```
HeapifyUp(H,i)
if i>1 then
   set j:=parent(i)=⌊i/2⌋
   if Key[H[i]]>Key[H[j]] then
      swap array entries H[i] and H[j]
      HeapifyUp(H,j)
   endif
endif
```

HeapifyUp: Soundness

Theorem

The procedure HeapifyUp(H,i) fixes the heap property in O(log i) time, assuming that the array H is almost a heap with the key of H[i] too large.

The running time of Insertion is O(log n)

Proof

Induction on i.

Base Case i = 1 is obvious

Induction Case: Swapping elements takes O(1) time

It remains to observe that after swapping H remains a heap or almost heap

Deletion

Delete(H,i)

```
set n:=length(n),
set H[i]:=H[n]
if Key[H[i]]>Key[H[parent(i)]] then
   HeapifyUp(H,i)
endif
if Key[H[i]]<Key[H[leftChild(i)]] or
  Key[H[i]]<Key[H[rightChild(i)]] then
   HeapifyDown(H,i)
endif
```

Deletion (cntd)

```
HeapifyDown(H,i)
set n:=length(H)
if 2i>n then Terminate with H unchanged
else if 2i<n then do
   set left:=2i, right:=2i+1
   let j be the index that minimizes Key[H[left]]
  and Key[H[right]]
else if 2i=n then   set j:=2i
endif
if Key[H[j]]>Key[H[i]] then
    swap array entries H[i] and H[j]
    HeapifyDown(H,j)
endif
```

HeapifyDown: Soundness

Theorem

The procedure HeapifyDown(H,i) fixes the heap property in O(log i) time, assuming that the array H is almost a heap with the key of H[i] too small.

The running time of Deletion is O(log n)

Proof DIY

Homework

Explain how to implement creating a heap, accessing and deleting the maximal element