Linear time sorting, Heaps and HeapSort

Presentation by Antonio-Gabriel Sturzu

Agenda

- Counting sort
- Radix sort
- Heaps
- Heap Sort
- Applications

Counting sort

- Assumes that each input of the array is in the interval [0,k] and that k=O(n)
- Counts for every number how many elements are less than or equal to it
- Using this information it puts every number in its correct place
- It is a stable sort

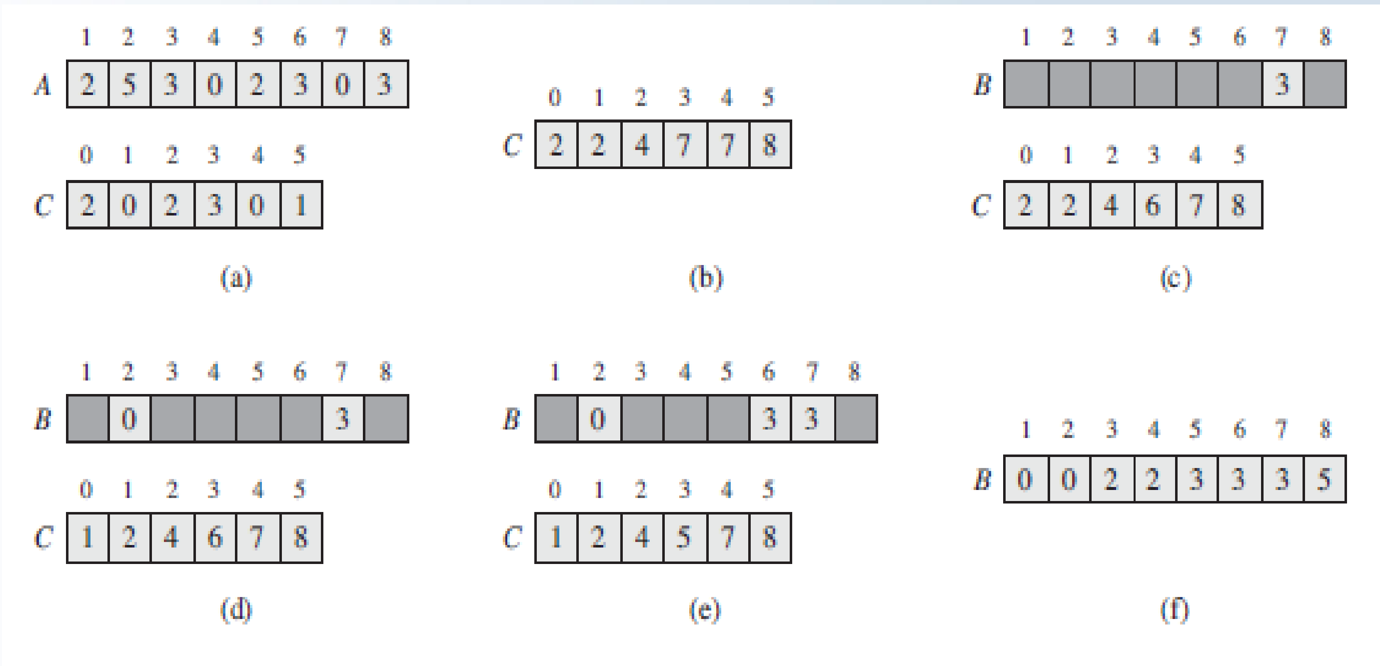
Counting sort

- Pseudocode:

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Counting sort

- Example:



Radix sort

- Used for sorting information keyed on multiple fields
- For example sorting some dates that are keyed on year, month and day
- But also it can be used to sort 64 bit integers in linear time
- Actually if we had infinite memory we could sort any vector of numbers in linear time no matter how big they are !

Radix sort

- Example for sorting 3 digit numbers:

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

Radix sort

- The pseudocode is extremely straightforward
- It is assumed that digit 1 is the least significant one and so one

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

Radix sort

- If the chosen stable sort algorithm can sort in $\Theta(n+k)$ time where n is the number of elements and k is the maximum value for a digit then the total running time for radix sort will be $\Theta(d(n+k))$ where d is the maximum number of digits that a number in the input vector can have
- If the numbers in the vector don't have the same number of digits this is not a problem because we can padd them in order to achieve d digits

Radix sort

- We can prove that radix sort is correct by using induction on the current digit on which we must sort
- Induction: After k steps the numbers will be correctly sorted based on the first k digits
- Base case: After first step the numbers are correctly sorted after the first digit (the least significant digit)
- This is true because we are using a correct stable sort algorithm

Radix sort

- Now we assume P(k) true and try to prove P(k+1)
- At the k+1'th step the sorting algorithm sorts the numbers on the k+1'th digit
- For different digits it is obvious that after k+1 steps the numbers are in correct order because the stable sorting algorithm will put the numbers with the lower value digits first
- It is not that obvious what happens if the digits are equal

Radix sort

- Now we use the fact that the sorting algorithm is stable and the induction step
- After k steps we know that the elements are correctly sorted
- So, for the elements with equal k+1'th digits if they preserve their relative order they will remain sorted
- Now, because the stable sort algorithm preserves the relative order for equal elements the proof is complete !

Radix sort

- On the general case:
  - If we have n b bit numbers and any positive integer $r \le b$ then the running time of radix sort will be $\Theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\Theta(n+k)$ where the digits span in the $[0,k-1]$ interval
- The question that remains is when it is effective to use radix sort ?

Radix sort

- If b=O(logn) and we choose r approximately logn then the running time of radix sort will be Θ(n)
- The problem is that radix sort uses much more memory than quicksort for example which sorts the vector inplace if we don't take into account the stack usage
- Also the constant factor hidden by the Θ notation is bigger in practice than for quicksort
- Mainly it depends how well you implement the radix sort and if you want to optimize memory usage

Radix sort

- If you want to optimize for memory usage you might want to use an inplace sorting algorithm
- The choice also depends heavely on the type of input data that you have
- Problem:
  - Give an efficient algorithm for sorting a vector of 64 bit integers in linear time

Heaps

- The binary heap represents an almost complete binary tree
- It comes in two flavours:
  - Max heap
  - Min heap
- In this presentation we will discuss only maxheaps
- But the same things presented for maxheaps also apply to minheaps as well

Heaps

- A heap is usually stored as an array
- It uses the following three operations:
  – left(i)=2*i
  – right(i)=2*i+1
  – parent(i)=i/2
- A maxheap has the following property:
  – For every i V[i]<=V[parent(i)]
- From this property we can conclude that the maximum element in a heap will be stored in the root

Heaps

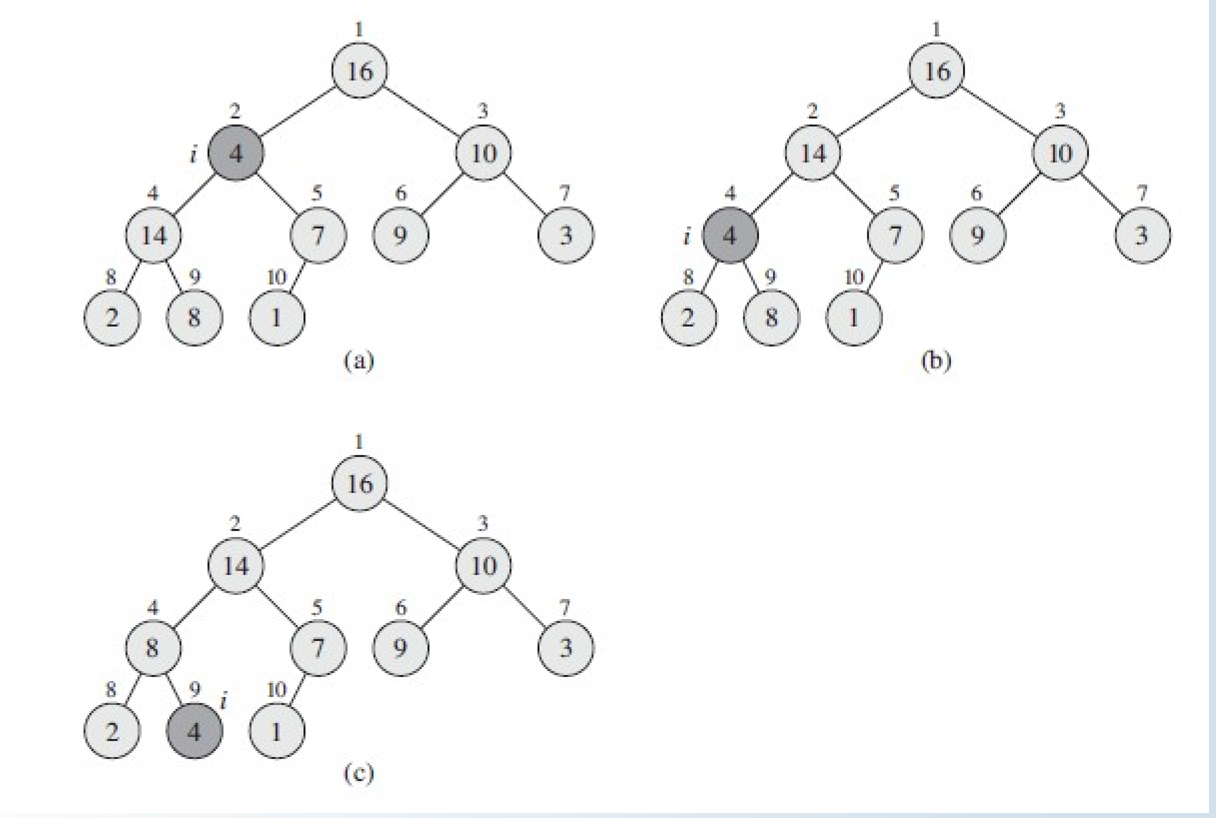
- Our first goal is how to build a heap out of an arbitrary array
- In order to do this we define a procedure that maintains the heap property called maxheapify
- The procedure receives as inputs the array and an index i
- It also assumes that the maxheap property is kept for the binary trees rooted at left(i) and right(i)

Heaps

- The problem might be that the tree rooted at node i does not obey the maxheap property
- This can happen if A[i] is smaller than either of its children

Heaps

- Example for maxheapify procedure:



Heaps

- The pseudocode is straightforward:

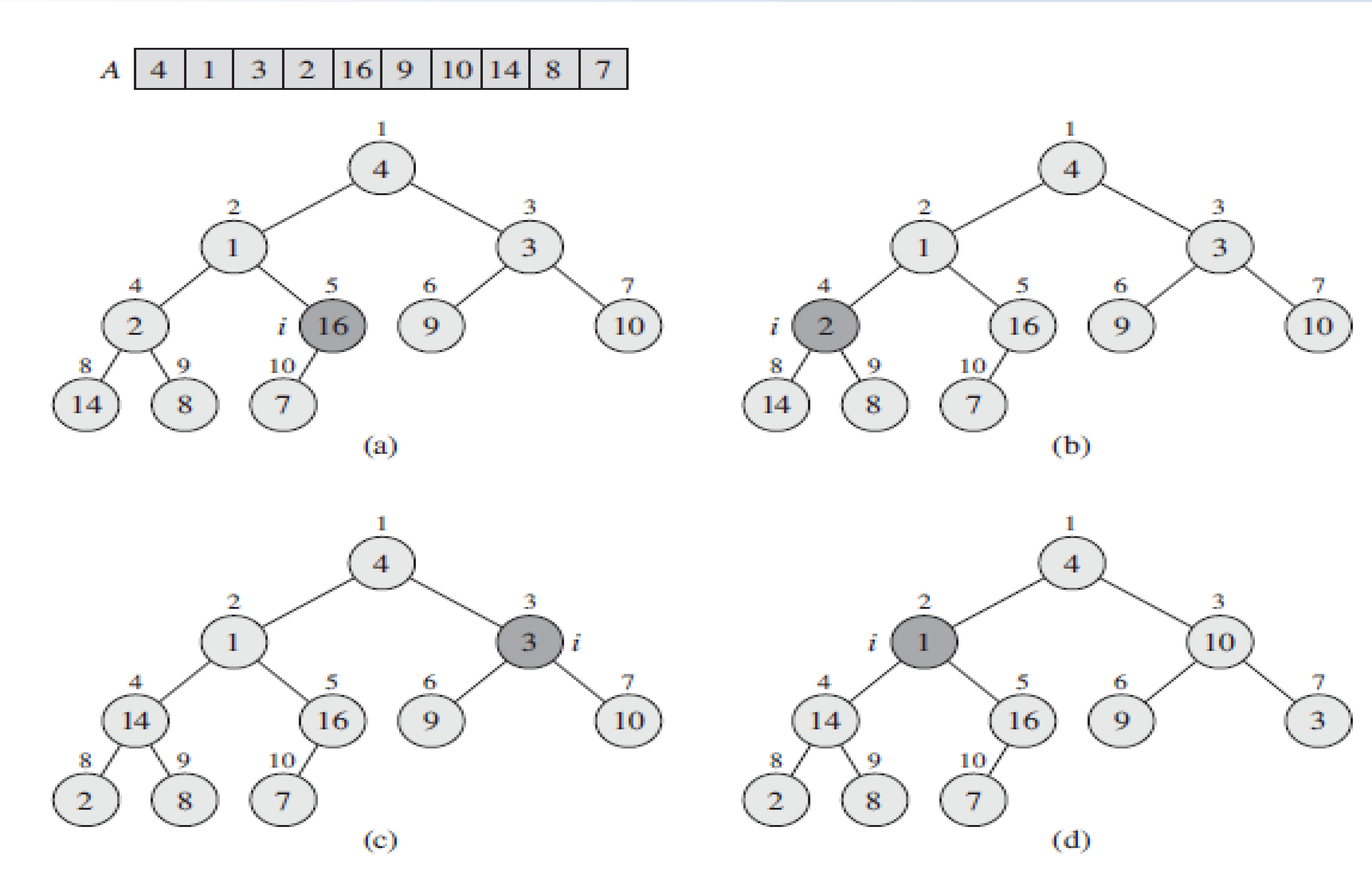
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Heaps

- The time complexity of maxheapify is O(h) where h is the height of the tree rooted at i
- If the heap has n elements then h is O(logn)
- The procedure is quite efficient except for the recursive call
- How could we eliminate the recursion ?

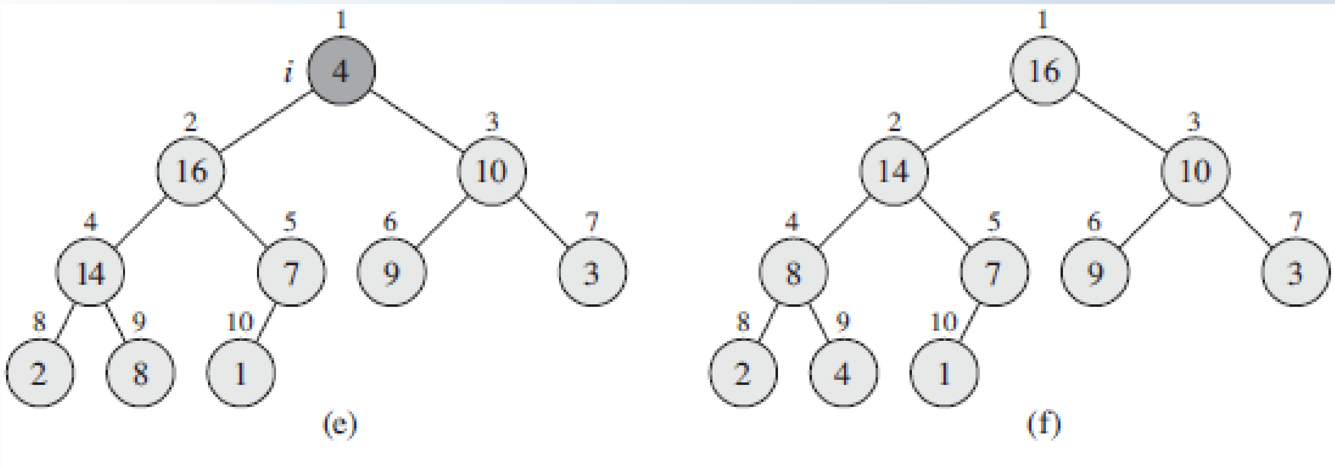
Heaps

- Building the heap:



Heaps

- Building the heap:



Heaps

- Pseudocode is quite simple:

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Heaps

- Time complexity for building a maxheap
- A first obvious upper bound would be O(nlogn) because we have n nodes, we call maxheapify on half of them and the maxheapify operation takes O(logn)
- A tighter upper bound exists
- For a height h we have at most ceil(n/2^{h+1}) nodes

Heaps

- The execution time will therefore be

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Heaps

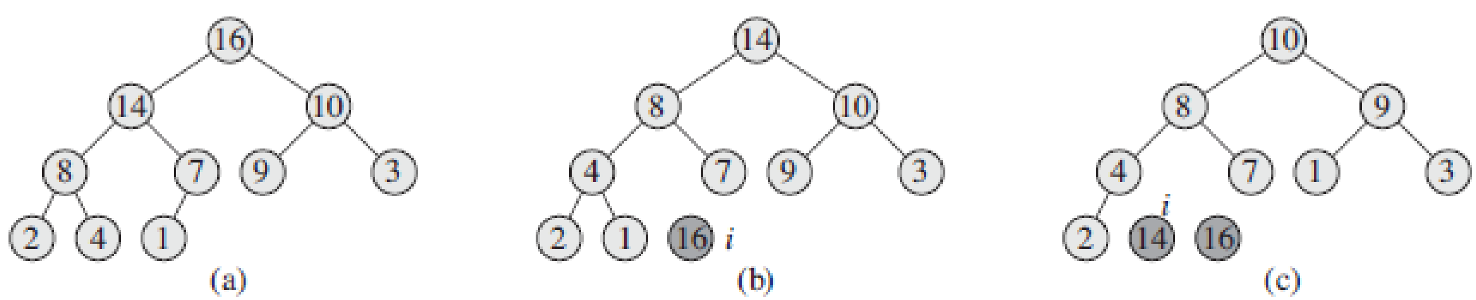
- It turns out that this is actually O(n) !
- Now finally the heapsort algorithm
- Pseudocode:

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

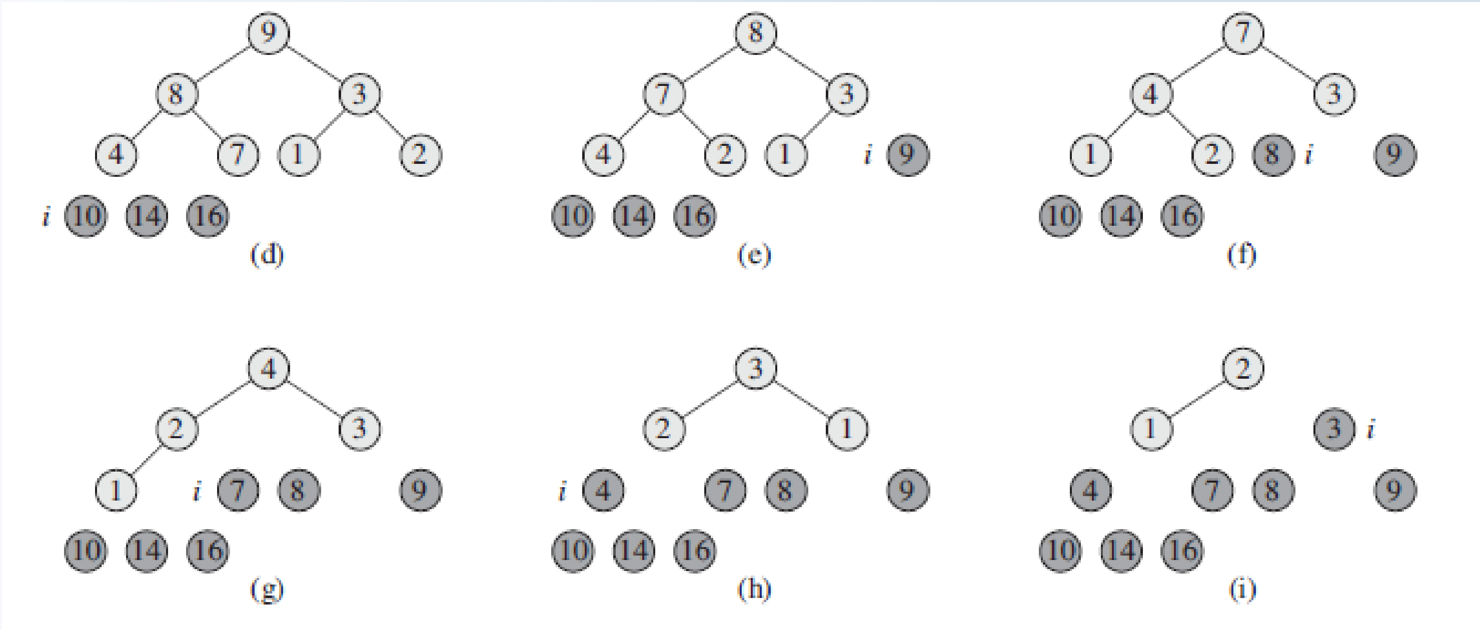
Heapsort

- Example:

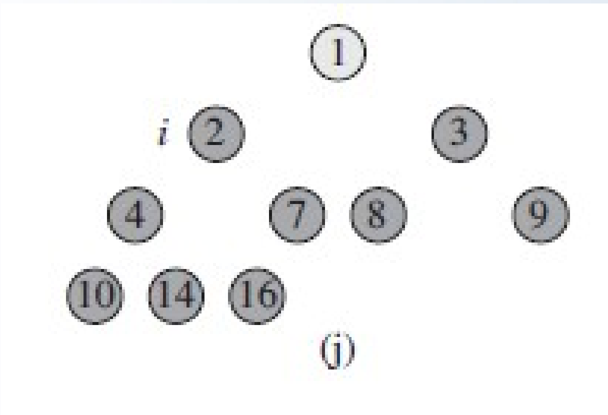
A										
	1	2	3	4	7	8	9	10	14	16



Heapsort



Heapsort



Heaps

- The time complexity is O(nlogn)
- The heapsort algorithm can be proved using an invariant
- Besides heapsort the main application of heaps is priority queues
- Also, heaps can be used to implement other tricky data structures where you need to retrieve the maximum and minimum in logarithmic time

Heaps

- Problems:
  - How to efficiently merge k sorted arrays where the sum of the lenghts of all the arrays is n ?
  - How can a search engine keep in real time a list of the top k most frequent queries in that day ?
  - Say we receive in real time a flow of n numbers that we don't know in advance. How can we efficiently maintain and retrieve the median values as we receive new numbers ?