HEAP SORT (9.4.2)

EECS 2011

Heap Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods *insert* and *deleteMin* take $O(\log n)$ time
  - methods *size*, *isEmpty*, and *findMin* take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Sorting Using a Heap

- Input: array A with *n* elements to be sorted
- Temporary array T of size at least *n*+1, which will work as a heap.
- 2 steps:

1. Create a heap T using the elements of A
   - Insert each element A[i] into T using T.insert(A[i])
2. Call T.deleteMin() *n* times to move the elements from T to A, one by one.

Sorting Code

```
for (i = 0; i++; i < n)
    T.insert(A[i]);
for (i = 0; i++; i < n)
    A[i] = T.deleteMin();
```

5

Analysis of Heap Sort

- Stirling's approximation: $n! \approx n^n e^{-n} \sqrt{2\pi n}$
- Insertions
  log1 + log 2 + … + log n = log(n!) = O(nlogn)
- Deletions
  log1 + log 2 + … + log n = log(n!) = O(nlogn)
- Total = O(nlogn)

6

IN-PLACE HEAP SORT

7

In-place Heap Sort

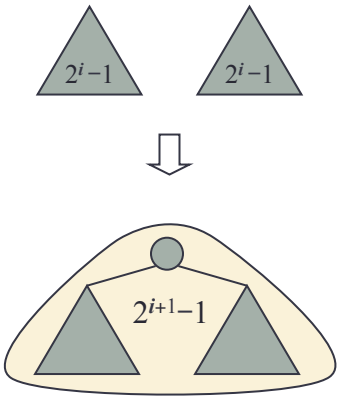
- The heap sort algorithm we just discussed requires a temporary array T (a min heap).
- In-place heap sort uses only one array, the original array storing the inputs.
- 2 steps:

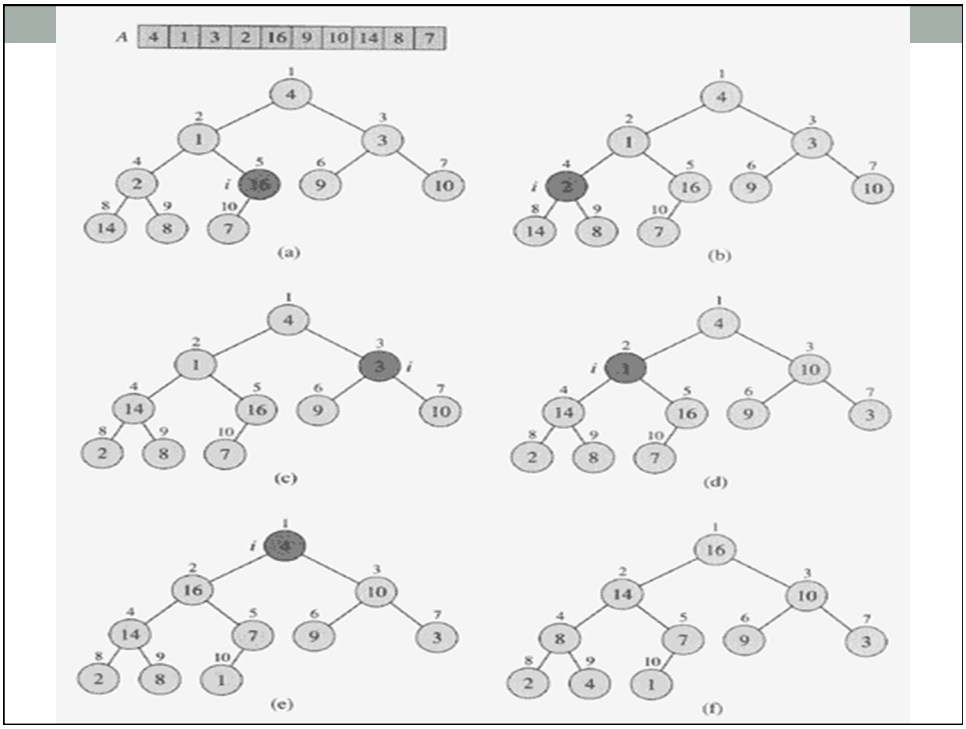
1. Transform the original array to a ***max*** heap using *buildHeap* procedure ("heapify")
2. Call *delete**Max***( ) *n* times to get the array sorted.

8

Step 1: *buildHeap*

- Input: a non-heap binary tree stored in an array
- Output: a heap stored in the same array
- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys





10

buildHeap Example

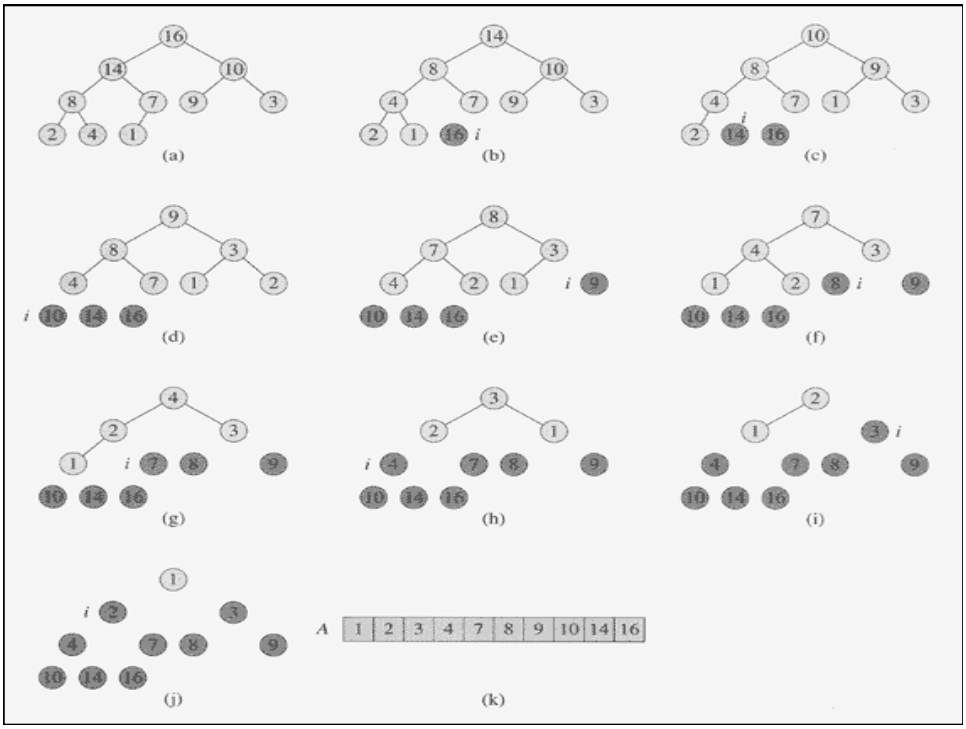
- See demo with max heaps at
www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
www.cs.usfca.edu/~galles/visualization/HeapSort.html

- "Heapify" from height 1 to *h* (bottom-up construction)

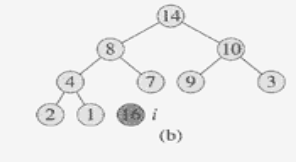
Step 2: Call *deleteMax*

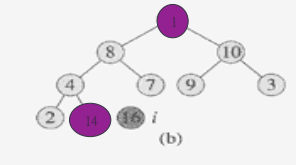
- The first step is to build a max heap using *buildHeap.*
- Call *deleteMax* to remove the max item (the root).
  - The heap size is reduced by one.
  - The last entry of the heap is now empty.
  - Store the item just removed into that location (*copyMax*).
- Repeat *deleteMax* and *copyMax* until the heap is empty (*n* – 1 times).
- Examples: next slides
- Demo/animation

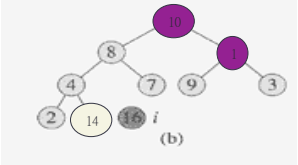
  www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
  www2.hawaii.edu/~copley/665/HSApplet.html

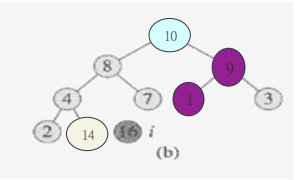


Breakdown of Step (b)



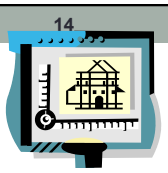






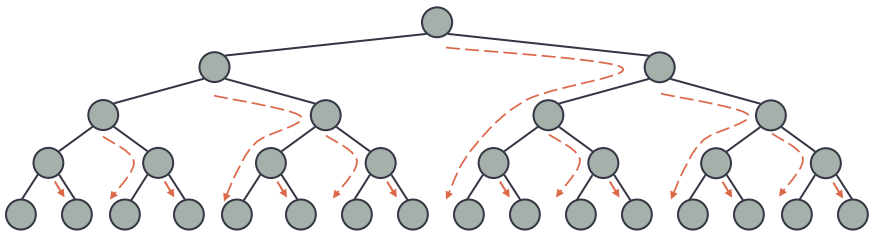

```
temp = A[1];    // 14
A[1] = A[i-1]; // A[1] = 1
// Perform down-heap percolation
// So 10 now is the new root.
// 1 is a leaf.
A[i-1] = temp; // 14
i = i-1;
```


Analysis of *buildHeap*



- Bottom-up heap construction runs in $O(n)$ time.
- Bottom-up heap construction is faster than n successive insertions (slide 3), which take $O(\ \ )$.

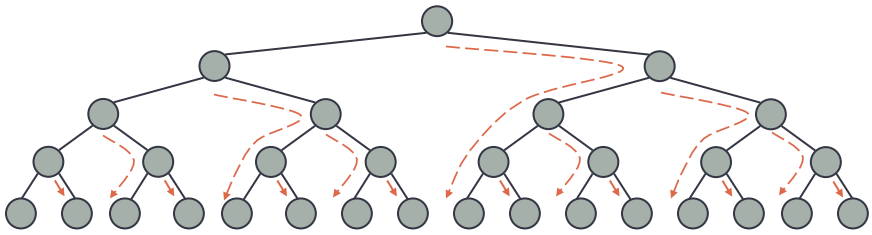
  $\Rightarrow$ speeds up the first phase of heap-sort.



15

Analysis of *buildHeap* (2)

- Theorem: For the complete binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of all the nodes is $2^{h+1} - 1 - (h + 1)$
- *buildHeap* thus runs in $O(n)$ time



16

Analysis of In-place Heap Sort

- Build a max heap using *buildHeap* $\Rightarrow O(\ )$
- Repeat
  - *deleteMax* $\Rightarrow O(\ )$
  - *copyMax* $\Rightarrow O(\ )$

  until the heap is empty $\Rightarrow n$ iterations

Total running time = $O(n \log n)$

Review of Heap Sort

Using a temp heap T

```
for (i = 0; i++; i < n)
   T.insert(A[i]);
for (i = 0; i++; i < n)
   A[i] = T.deleteMin();
```

Note: ***min*** heap

In-place sorting

```
run buildHeap on A;
repeat
   deleteMax;
   copyMax;
until the heap is empty;
```

Note: ***max*** heap

18

Next lecture …

- Hash Tables (section 10.2)