Heaps, Heap Operations, and Heapsort

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

1. Complete binary trees and heaps
2. Algorithm heapsort
3. Inserting a new heap node
4. Deleting the maximum key from a heap
5. Analysis of heapsort: linearithmic time in all cases
6. Implementation of heapsort

Algorithm Heapsort

Proposed by J. W. J. (Bill) Williams in 1964, heapsort improves over selection sort due to a special binary-tree data structure.

- This special type of a complete binary tree is called a **heap**.
- The worst-case $\Theta(n \log n)$ complexity (like mergesort).

Basic steps of heapsort:

1. Convert an array into a maximum (or alternatively – a minimum) heap in linear time $\Theta(n)$.
2. Sort the heap in $\Theta(n \log n)$ time by deleting n times the maximum item from the maximum heap (or the minimum item from the minimum heap, respectively).
   - Each deletion of the maximum (or minimum) item takes the logarithmic time, $\Theta(\log n)$.

Algorithm Heapsort: First Publication

ALGORITHM 232
HEAPSORT

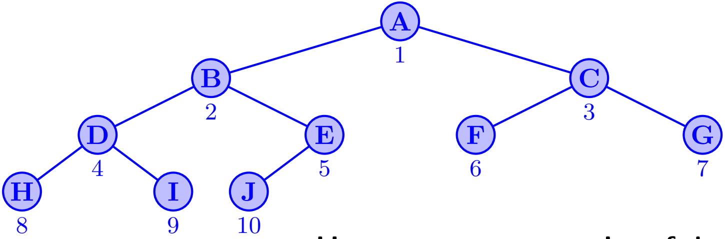
J. W. J. Williams (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, England

comment The following procedures are related to *TREESORT* [R. W. Floyd, Alg. 113, *Comm. ACM 5* (Aug. 1962), 434, and A. F. Kaupe, Jr., Alg. 143 and 144, *Comm. ACM 5* (Dec. 1962), 604] but avoid the use of pointers and so preserve storage space. All the procedures operate on single word items, stored as elements 1 to n of the array A. The elements are normally so arranged that $A[i] \leqq A[j]$ for $2 \leqq j \leqq n$, $i = j \div 2$. Such an arrangement will be called a heap. $A[1]$ is always the least element of the heap.

Volume 7 / Number 6 / June, 1964 Communications of the ACM 347

Complete Binary Tree: Definition 2.19 (Textbook)

- A binary tree, filled completely at all levels except, possibly, the bottom level, filled from left to right with no missing nodes.
- Each leaf is of depth h (the tree height) or $h - 1$.

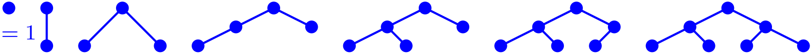


Linear array representation of the heap:

0	1	2	3	4	5	6	7	8	9	←Indices
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	
1	2	3	4	5	6	7	8	9	10	←Positions

Complete Binary Tree: Basic Properties

$\left[2^h \leq n \leq 2^{h+1} - 1\right]$ nodes in a complete binary tree of height h:



Storing a complete binary tree in a linear array

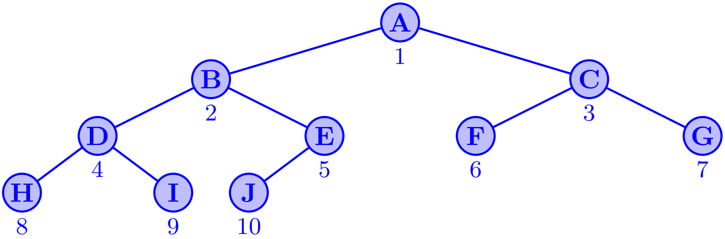
Node positions – by the level-order traversal (the root position is 1).

- Positions $p = i + 1$ of the array elements $a[i]$ with indices i.

If the node is in the position p then:

- the parent node is in the position $\left\lfloor \frac{p}{2} \right\rfloor$;
- the left child is in the position $2p$, and
- the right child is in the position $2p + 1$.

Positions of Nodes: Example 2.21 (Textbook)



A	B	C	D	E	F	G	H	I	J	
1	2	3	4	5	6	7	8	9	10	←Positions

- The node in position $p = 1$ is the root (no parent node).
- The nodes in positions $p = 6, 7, 8, 9, 10$ are the leaves (no children).
- A left child of the root, $p = 1$, is in position $2p = 2$.
- A right child of the root, $p = 1$, is in position $2p + 1 = 3$.
- For the node in position $p = 4$, the parent in position $\left\lfloor \frac{4}{2} \right\rfloor = 2$, a left child in position $2p = 8$, and a right child in position $2p + 1 = 9$.
- For the node in position $p = 5$, the parent in position $\left\lfloor \frac{5}{2} \right\rfloor = 2$, and the only left child in position $2p = 10$.

Binary Heap: Definition 2.22 (Textbook)

A (maximum) **heap** is a complete binary tree having a numerical key associated with each node, such that the key of each parent node is greater than or equal to the keys of its child nodes.

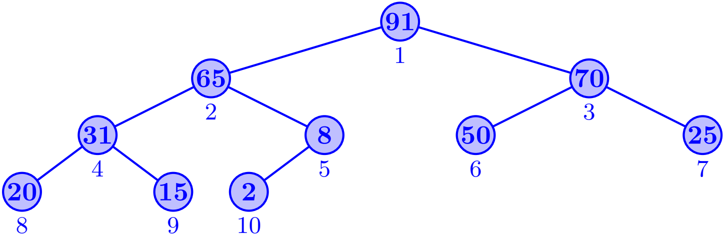
The heap order provides easy access to the maximum key associated with the root.

- Alternatively, a minimum heap has the key of each parent node, which is less than or equal to the keys of its child nodes.
- Then the minimum key is associated with the root.

Lemma 2.24 (Textbook): The height of a complete binary tree with n nodes is at most $\lfloor \lg n \rfloor$.

Proof: A complete binary tree of height h contains n nodes:
$2^h \le n \le 2^{h+1} - 1$; so that $h \le \lg n < h + 1$.

Maximum Heap and Its Array Representation



91	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**	
1	2	3	4	5	6	7	8	9	10	←Positions

Algorithm heapsort

1. Given an input list, build a heap by successively inserting the elements.
2. Delete the maximum repeatedly, arranging the elements in the output list in reverse order of deletion, until the heap is empty.

This is a variant of selection sort using a different data structure.

Inserting a Node into a Heap

Lemma 2.25 (Textbook): Inserting a new, $(n+1)$-st, node into a heap of n elements takes logarithmic time, $\mathrm{O}(\log n)$.

Proof:

1. Create a new, $(n+1)$-st, leaf position.
2. Place the new node with its associated key in this leaf.
3. If the inserted key preserves the heap order, the insertion is complete.
4. Otherwise, **bubble up**, or *percolate up* the new key towards the root by repeatedly swapping it with its parent until the heap order is restored.
5. There are at most h swaps, where h is the heap height, so that the running time is $\mathrm{O}(\log n)$.

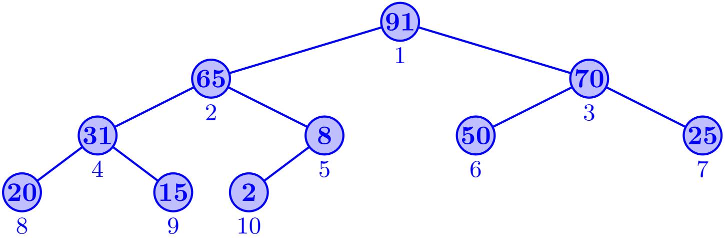
Inserting a Node: Example 2.26 (Textbook)

To insert an 11^{th} element, 75, into the heap of size $n = 10$ in Slide 9 takes three steps:

1. Create position $n + 1 = 11$ to initially place the new key, 75.
2. Swap the new key with its parent key, 8, in position $5 = \left\lfloor \frac{11}{2} \right\rfloor$ to restore the heap order.
3. Repeat the same swap for the parent key, 65, in position $2 = \left\lfloor \frac{5}{2} \right\rfloor$.
4. Terminate the process as the heap order is now restored.

Position Index	1 0	2 1	3 2	4 3	5 4	6 5	7 6	8 7	9 8	10 9	11 10
Initial array	91	65	70	31	8	50	25	20	15	2	
Array at step 1	91	65	70	31	8	50	25	20	15	2	**75**
Array at step 2	91	65	70	31	**75**	50	25	20	15	2	*8*
Array at steps 3–4	91	**75**	70	31	*65*	50	25	20	15	2	8

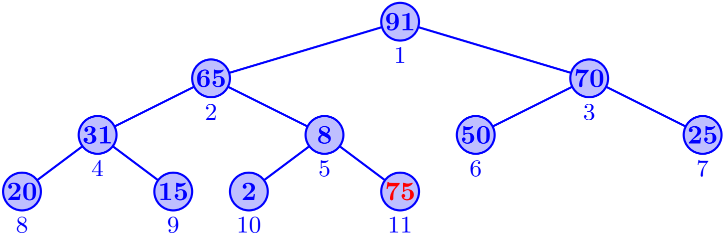
Inserting a Node: Example 2.26



91	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**	
1	2	3	4	5	6	7	8	9	10	$\leftarrow p$

Initial heap

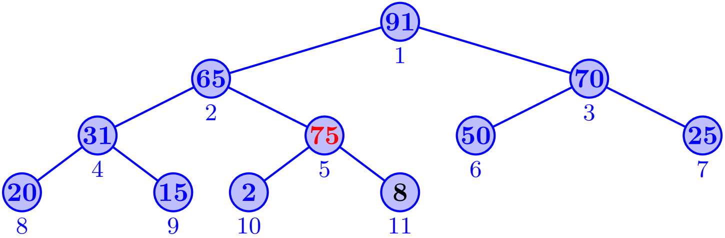
Inserting a Node: Example 2.26



91	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**	**75**	
1	2	3	4	5	6	7	8	9	10	11	$\leftarrow p$

Step 1: Initial position of the new key

Inserting a Node: Example 2.26

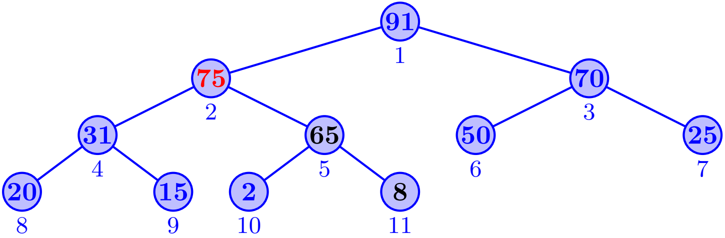


91	**65**	**70**	**31**	**75**	**50**	**25**	**20**	**15**	**2**	**8**
1	2	3	4	5	6	7	8	9	10	11

$\leftarrow p$

Step 2: Percolate up the new key

Inserting a Node: Example 2.26



91	**75**	**70**	**31**	**65**	**50**	**25**	**20**	**15**	**2**	**8**	
1	2	3	4	5	6	7	8	9	10	11	$\leftarrow p$

Step 3/4: Percolate up the new key and terminate

Deleting the Maximum Key from a Heap

Lemma 2.27 (Textbook): Deleting the maximum key from a heap of n elements takes logarithmic time, $\mathrm{O}(\log n)$, in the worst case.

Proof: The deletion reduces the heap size by one; therefore,

1. Eliminate the last leaf node and replace the deleted key in the root by the key associated with this leaf.
2. Then **percolate** the root key **down** the tree:
   - Compare the new root key to each child.
   - If at least one child is greater than the parent, swap the new root key with the larger child.
3. Repeat the percolation process until restoring the heap order.
4. There are at most h moves, where h is the heap height, so that the running time is $\mathrm{O}(\log n)$.

Due to percolating down the previous leaf key, the process usually terminates at or near the leaves.

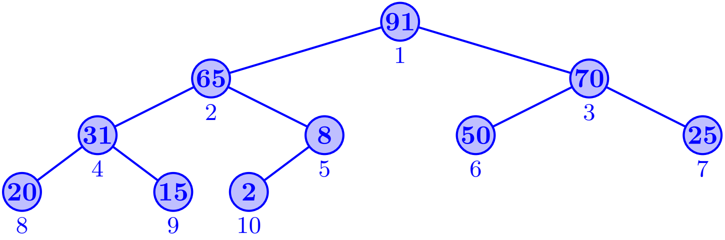
Deleting the Maximum Key: Example 2.28 (Textbook)

Deleting the maximum key, 91, from the heap in Slide 9, takes 3 steps:

1. Place key 2 from the eliminated position 10 at the root.
2. Percolate the new root key down by comparing to its children 65 and 70 in positions $2 = 2 \cdot 1$ and $3 = 2 \cdot 1 + 1$, respectively, and swapping with the larger child, 70, to restore the order.
3. Repeat the same swap for the children 50 and 25 in positions $6 = 2 \cdot 3$ and $7 = 2 \cdot 3 + 1$.
4. Terminate the process, because the heap order is now correct.

Position Index	1 0	2 1	3 2	4 3	5 4	6 5	7 6	8 7	9 8	10 9
Initial array	91	65	70	31	8	50	25	20	15	2
Array at step 1	**2**	65	70	31	8	50	25	20	15	
Array at step 2	*70*	65	**2**	31	8	50	25	20	15	
Array at steps 3–4	*70*	65	*50*	31	8	**2**	25	20	15	

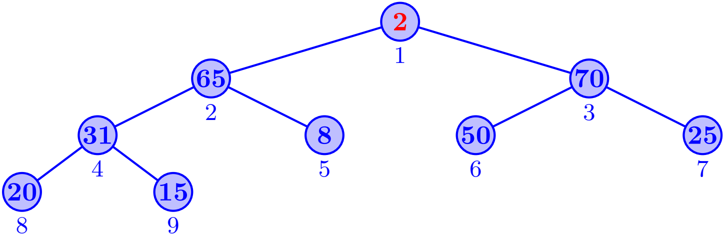
Deleting the Maximum Key: Example 2.28



91	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**	
1	2	3	4	5	6	7	8	9	10	$\leftarrow p$

Initial heap

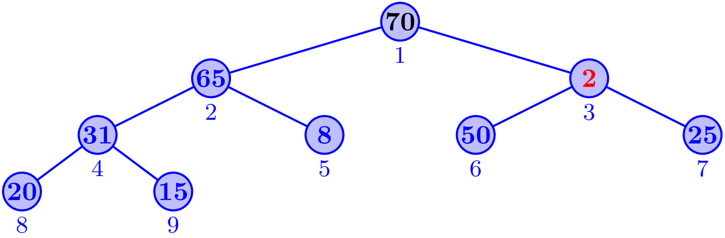
Deleting the Maximum Key: Example 2.28



2	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	
1	2	3	4	5	6	7	8	9	$\leftarrow p$

Step 1: Eliminate the last leaf and place its key to the root

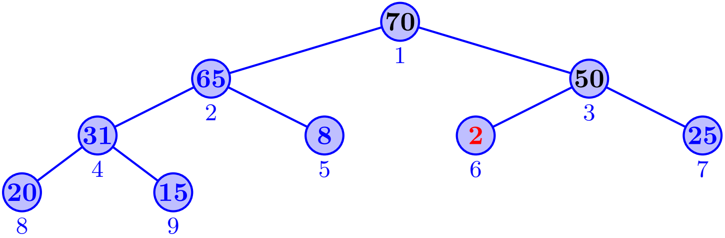
Deleting the Maximum Key: Example 2.28



70	**65**	**2**	**31**	**8**	**50**	**25**	**20**	**15**	
1	2	3	4	5	6	7	8	9	$\leftarrow p$

Step 2: Percolate the placed key down

Deleting the Maximum Key: Example 2.28



70	65	50	31	8	2	25	20	15	
1	2	3	4	5	6	7	8	9	$\leftarrow p$

Step 3: Percolate the placed key down and terminate

Time Complexity of Heapsort

Lemma 2.29 (Textbook): Heapsort runs in time in $\Theta(n \log n)$ in the best, worst, and average case.

Proof.

- The heap can be constructed in time $\mathrm{O}(n \log n)$.
    - Actually, even in time $\mathrm{O}(n)$, but this does not affect the result.
- Then heapsort repeats n times the deletion of the maximum key and restoration of the heap property (each resoration is logarithmic in the best, worst, and average case).

Therefore, the total time is[1]:

$$\log(n) + \log(n-1) + \ldots + \log(1) = \log(n!) \in \Theta(n \log n)$$

[1] The Stirling's approximation: $n! \approx n^n e^{-n}\sqrt{2\pi n}$.

Building a Heap in Linear Time, $\Theta(n)$

Heap as a recursive structure: left subheap $\leftarrow$ root $\rightarrow$ right subheap

Lemma 2.31: A heap can be built from a list of size n in $\Theta(n)$ time.

Proof:

- To form the heap, each of the two subtrees attached to the root are transformed into heaps of height at most $h-1$.
  - The left subtree is always of height $h-1$, whereas the right subtree could be of height $h-2$.
- In the worst case the root percolates down the tree for at most h steps that takes time $\mathrm{O}(h)$.
- Thus the worst-case time $T(h)$ to build a heap of height at most h is given by the recurrence $T(h) = 2T(h-1) + ch$.
- Thus $T(h) \in \mathrm{O}(2^h)$, or $T(h) \in \mathrm{O}(n)$ because $h = \lfloor \lg n \rfloor$, i.e. $2^h \leq n$.
- The lower bound is $\Theta(n)$ since every input element must be inspected. $\square$

Non-recursive percolate-down procedure: recursion is eliminated by applying this procedure in reverse level order.

Algorithm Heapsort: Pseudocode

algorithm `heapSort`

Input: array $a[0..n-1]$

begin

$\boxed{\text{Building a heap from array } a[0..n-1] \text{ in reverse level order}}$

for $i \leftarrow \lfloor \frac{n}{2} \rfloor - 1$ **while** $i \geq 0$ **step** $i \leftarrow i - 1$ **do**

`percolateDown`(a, i, n) $\boxed{\text{restore the heap in subarray } a[i..n-1]}$

end for

$\boxed{\text{Successive ordering of the heapified array } a[0..n-1]}$

for $i \leftarrow n - 1$ **while** $i \geq 1$ **step** $i \leftarrow i - 1$ **do**

`swap`$(a[0], a[i])$ $\boxed{\text{delete the maximum key to place it in order}}$

`percolateDown`$(a, 0, i)$ $\boxed{\text{restore the heap in subarray } a[0..i-1]}$

end for

end