Heapsort

Submitted by :

Hardik Parikh(hjp0608)
Soujanya Soni (sxs3298)

Overview of Presentation

- Heap Definition.
- Adding a Node.
- Removing a Node.
- Array Implementation.
- Analysis

What is Heap?

A Heap is a Binary Tree *H* that stores a collection of keys at its internal nodes and that satisfies two additional properties:

- Relational Property
- Structural Property

Heap Properties

•Heap-Order Property (Relational):

In a heap H, for every node *n* (except the root), the key stored in *n* is smaller than or equal to the key stored in *n* 's parent. (a Min-Heap) so A (i) <= A (parent [i])

•Complete Binary Tree (Structural):

A Binary Tree *T* is complete if each level but the last is full, and, in the last level, all of the internal nodes are to the left of the external nodes.

Two Basic Procedure on Heap

1. Heapify : Maintaining property of Heap
2. Build Heap:Construction of the Heap from the list of number.

Building a Heap

Root

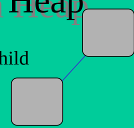
1. All leaf nodes occur at adjacent levels.

When a complete binary tree is built, its first node must be the root.

Building a Heap

2. All levels of the tree, except for the bottom level are completely filled. All nodes on the bottom level occur as far left as possible.

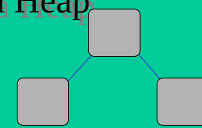
Left child of the root



The second node is always the left child of the root.

Building a Heap

3. The key of the root node is greater than or equal to the key of each child. Each subtree is also a **heap**.

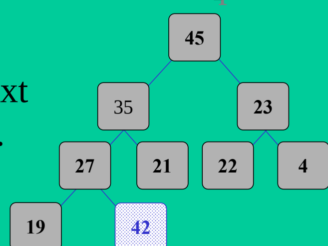


Right child of the root

The third node is always the right child of the root.

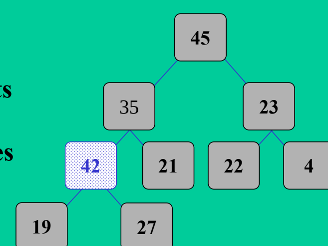
Adding a Node to a Heap

- Put the new node in the next available spot.



Adding a Node to a Heap

- **Push the new node upward, swapping with its parent until the new node reaches an acceptable location.**

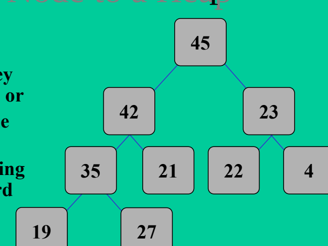


Adding a Node to a Heap

- **Push the new node upward, swapping with its parent until the new node reaches an acceptable location.**

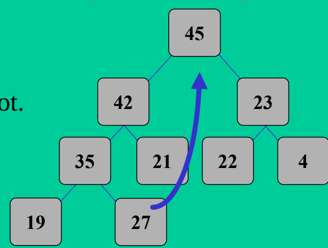
Adding a Node to a Heap

- **The parent has a key that is >= new node, or**
- **The node reaches the root.**
- **The process of pushing the new node upward is called [reheapification upward](#).**



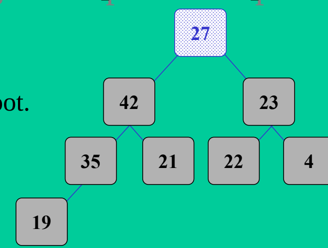
Removing the Top of a Heap

- Move the last node onto the root.



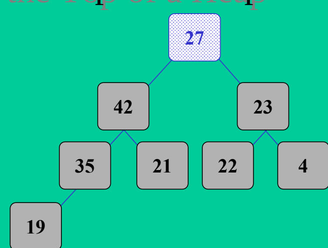
Removing the Top of a Heap

- Move the last node onto the root.



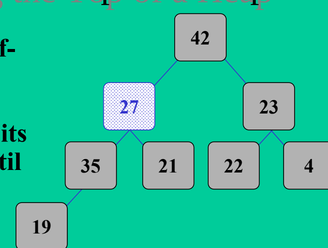
Removing the Top of a Heap

- **Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.**



Removing the Top of a Heap

- **Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.**

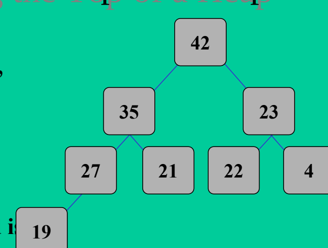


Removing the Top of a Heap

- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

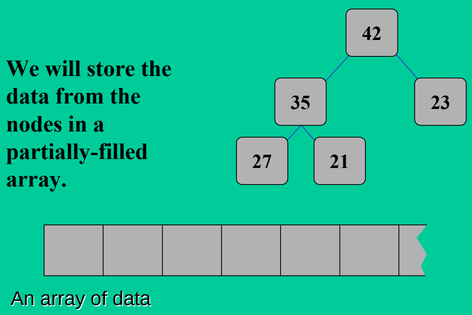
Removing the Top of a Heap

- **The children all have keys <= the out-of-place node, or**
- **The node reaches the leaf.**
- **The process of pushing the new node downward i called [reheapification downward].**



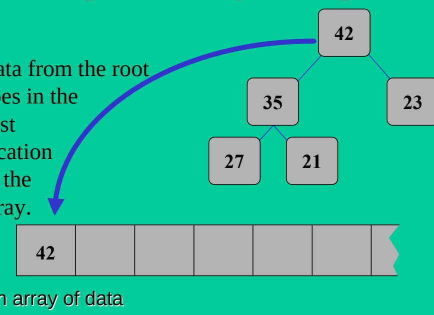
Implementing a Heap from An Array

- **We will store the data from the nodes in a partially-filled array.**



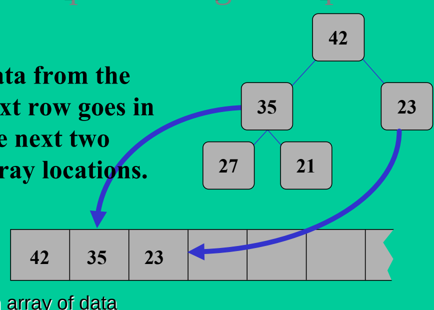
Implementing a Heap

- Data from the root goes in the first location of the array.



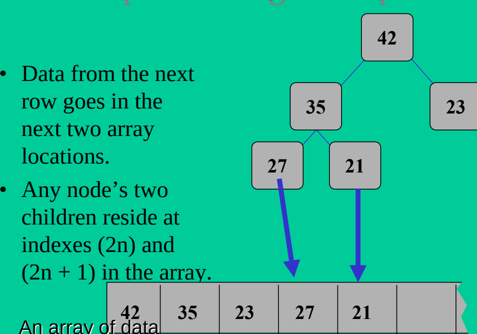
Implementing a Heap

- **Data from the next row goes in the next two array locations.**



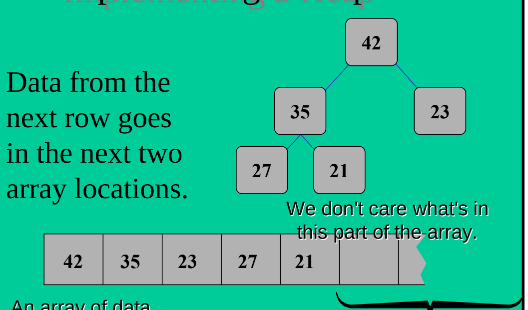
Implementing a Heap

- Data from the next row goes in the next two array locations.
- Any node's two children reside at indexes (2n) and (2n + 1) in the array.



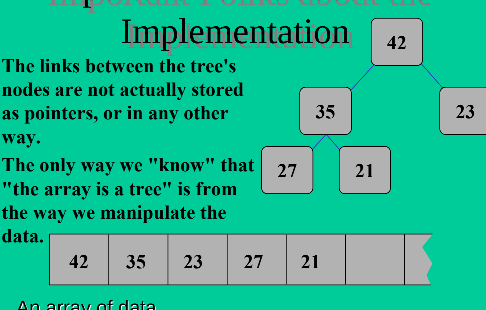
Implementing a Heap

- Data from the next row goes in the next two array locations.

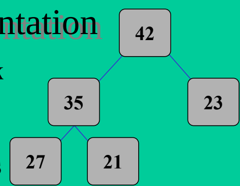


Important Points about the Implementation

- **The links between the tree's nodes are not actually stored as pointers, or in any other way.**
- **The only way we "know" that "the array is a tree" is from the way we manipulate the data.**



Important Points about the Implementation



- **If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.**

42	35	23	27	21		
[1]	[2]	[3]	[4]	[5]		

Heap Sort

Heap sort is the technique of sorting the list of number using "Heap".

Special about Heap sort

- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is O(n log n). The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.

Pseudocode for the heapsort algorithm

//Heapsort for the array called data with n elements
1. Convert the array of n elements into a heap.
2. unsorted = n; *// The number of elements in the unsorted side*
3. *while* (unsorted > 1)
{ *// Reduce the unsorted side by one*
unsorted - -;
Swap data[0] with data [unsorted].
The unsorted side of the array is now a heap with the root out of place.
Do a reheapification downward to turn the unsorted side back into a heap.
}

Analysis of Heapify-Method

If we put a value at root that is less than every value in the left and right subtree, then 'Heapify' will be called recursively until leaf is reached. To make recursive calls traverse the longest path to a leaf, choose value that make 'Heapify' always recurse on the left child. It follows the left branch when left child is greater than or equal to the right child, so putting 0 at the root and 1 at all other nodes, for example, will accomplished this task. With such values 'Heapify' will called *h* times, where *h* is the heap height so its running time will be θ (*h*) (since each call does (1) work), which is (lgn). Since we have a case in which Heapify's running time (lg n), its worst-case running time is Ω(lgn).

Analysis of Build Heap-Method

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.
- We can build a heap from an unordered array in linear time.

Analysis of Heapsort

- if we can build a data structure from our list in time X and
- if finding and removing the smallest object takes time Y then the total time will be O(X + n Y).
- In our case X will be 0(n) and Y will be O (log n) so,

total time will be O (n + n log n) = O (n log n)

Analysis of Heapsort(Cont'd)

Heapsort Algorithm consists of a few steps:

- Build Heap – runs in linear time O (n)
  - • produces a Heap from an unordered input array
  - • A bottom up process starting at node and working back to top
- Heapify – runs in O (log n) time
  - • maintains the heap property/manipulates Heaps
  - • Heapify is a top down process
- Heapsort – then runs in O (n log n) time
  - • sorts an array, in place
- Extract Min and Insert (a new key)
  - • each runs in O (log n) time
  - • allows the heap data structure to be used as a priority queue.

References

- http://www.cs.uml.edu/~tom/404/notes/6_Heapsort.pdf
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSort.htm
- http://nova.umuc.edu/~jarc/idsv/lesson3.html
- "Introduction to Algorithms" by Corman, Leiserson, Rivest ,Stein.

Questions

?