Slide 1

Lecture 5: Heaps

Sequential Unfoldment of Natural Law

Amortized Analysis & Trees   1

1

Slide 2

Wholeness Statement

A heap is a binary tree that stores sortable elements at each internal node and maintains *heap-order* and is *complete* (balanced). Heap-order means that for every node v (except the root), $key(v) \geq key(parent(v))$. *Science of Consciousness*: Pure consciousness is the field of wholeness, perfectly orderly, balanced, and complete. Through regular TM practice we release stress and automatically develop the qualities of the unified field in our lives.

Amortized Analysis & Trees   2

2

Slide 3

The Heap Data Structure

Heaps   3

3

Slide 4

Min-Heap ADT

- A Heap stores a collection of sortable elements
- Main methods of the Heap ADT
  - insertElem(e) inserts and returns the new Position (node) inserted into the Heap that contains the element e
  - removeMin() removes and returns the smallest element in the Heap
- Additional methods
  - minElem() returns the smallest element, but does not remove it
  - size(), isEmpty()

Heaps   4

4

Slide 5

Comparator

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic Heap uses an auxiliary comparator
- The comparator is external to the elements being compared
- When the Heap needs to compare two elements, it uses its comparator

- The Comparator Method returns a number as its return type indicating the relative size of its input arguments

 - compare(x, y) returns a negative number if x<y, zero if x=y, and a positive number if x>y

Heaps   5

5

Slide 6

What is a heap

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal node of depth $h-1$

last node
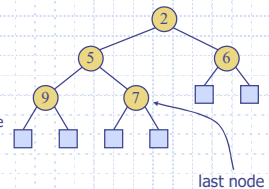
Heaps   6

6

Heap-Order Property

◆ For all internal nodes v (except the root):

$$key(v) \geq key(parent(v))$$

- That is, the key of every child node is greater than or equal to the key of its parent node

Heaps 7

7

Other Properties of a Heap

◆ A heap is a binary tree whose values are in ascending order on every path from root to leaf

◆ Values are stored in internal nodes only

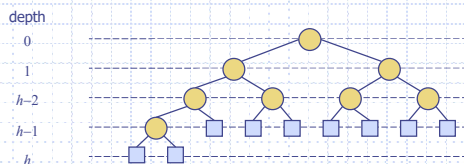◆ A heap is a binary tree whose root contains the minimum value and whose subtrees are heaps

Heaps 8

8

Heap

◆ All leaves of the tree are on two adjacent levels
◆ The binary tree is complete on every level except the deepest level.

Heaps 9

9

Adding Nodes to a Heap

◆ New nodes must be added left to right at the lowest level, i.e., the level containing internal and external nodes or containing all external nodes
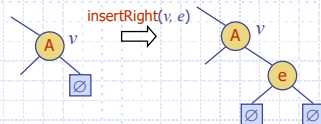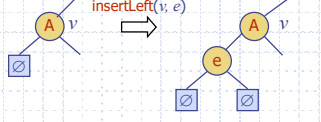
Heaps 10

10

Insert Methods

insertRight(v, e)
Right child must be external!
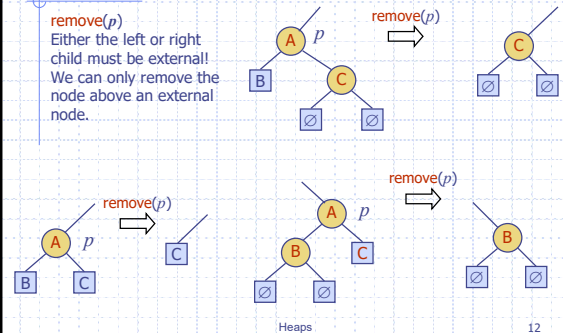

insertLeft(v, e)
Left child must be external.

Heaps 11

11

Remove Method

remove(p)
Either the left or right child must be external! We can only remove the node above an external node.

Heaps 12

12

Insertion into a Heap

- Important property of a min-heap
  - The smallest element is at the root
- The insertElem(k) insertion algorithm consists of three steps
  1. Find the insertion node z (the new last node)
  2. Store k at z and expand z into an internal node (either insertRight(
  3. Restore the heap-order property (Upheap)
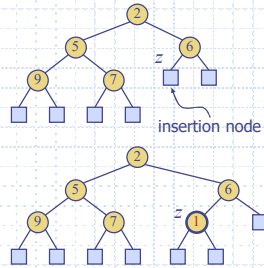
insertion node

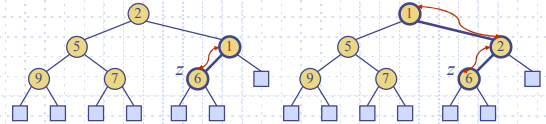Heaps   13

13

Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

Heaps   14

14

Finding the Insertion Node and Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - While the current node is a right child, go to the parent node
  - If the current node is a left child, go to the right child
  - While the current node is internal, go to the left child
- Similar algorithm for updating the last node after a removal
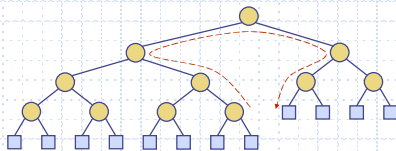
Heaps   15

15

Exercise

- Insert the following keys into a heap represented as a Tree:

  9, 6, 5, 14, 4, 12, 15, 3, 2

Heaps   16

16

Efficient Representation of A Heap

Use an Array or Sequence with efficient random access

Heaps   17

17

Efficient Data Structure for Implementing the Heap

- Heap implementation: use an array to store the binary tree.
- Node objects are referenced by index:
  - Index 0 is empty and not used.
  - Root node is at index 1
  - Left child is at 2*index
  - Right child is at 2*index+1
  - Position would contain the element and index into the array

B A D   C E

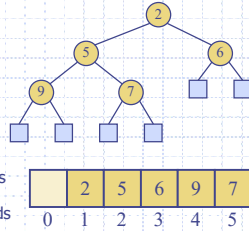Heaps   18

18

3
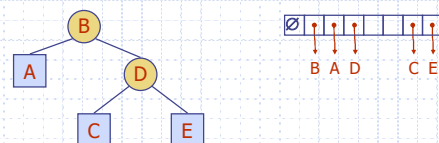
Sequence (or Array) based Heap Implementation

- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The leaves are not represented
- The cell at rank 0 is not used
- Operation insertItem corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- No empty cells in the array
- Yields in-place heap-sort

	2	5	6	9	7
0	1	2	3	4	5

Heaps   19

19

Exercise

- Insert the following keys into a heap represented as an array-based Tree:
  9, 6, 5, 14, 4, 12, 15, 3, 2

Heaps   20

20

Array-Based Implementation of Binary Tree

Operation	Time
size, isEmpty	
positions, elements	
swapElements(p, q), replaceElement(p, e)	
root(), parent(p), children(p)	
isInternal(p), isExternal(p), isRoot(p)	

Heaps   21

21

Array-Based Implementation of Binary Tree

Operation	Time
size, isEmpty	1
positions, elements	n
swapElements(p, q), replaceElement(p, e)	1
root(), parent(p), children(p)	1
isInternal(p), isExternal(p), isRoot(p)	1

Heaps   22

22

Pseudo Code for upHeap

Algorithm *upHeap(H, size)*
  Input Array H representing a heap and the size of H (size $\geq$ 1)
  Output H with the heap property restored

  $i \leftarrow size$
  $parent \leftarrow size/2$
  while $1 \leq parent \wedge \text{key}(H[parent]) > \text{key}(H[i])$ **do**
    swapElements(H[parent], H[i])
    $i \leftarrow parent$
    $parent \leftarrow$ Math.floor($i/2$)

Algorithm *swapElements(p, q)*
  $temp \leftarrow p.element$
  $p.element \leftarrow q.element$ {swap elements}
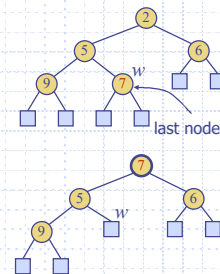 $q.element \leftarrow temp$

Heaps   23

23

Removal from a Heap

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
 - Restore the heap-order property

Heaps   24

24

4

Downheap

- ◆ After replacing the root key with the key *k* of the last node, the heap-order property may be violated
- ◆ Algorithm downheap restores the heap-order property by swapping key *k* along a downward path from the root
- ◆ Downheap terminates when key *k* reaches a leaf or a node whose children have keys greater than or equal to *k*
- ◆ Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time
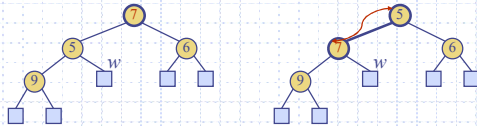
Heaps

25

25

Iterative Version of downHeap

Algorithm *downHeap(H, size)*
  Input Array H representing a heap and the size of H (size $\geq$ 1)
  Output H with the heap property restored
  property ← **false**
  i ← 1
  while ¬ *property* **do**
    smallest ← **indexOfMin**(H, i, size)
    if *smallest* $\neq$ **i** **then**
      swapElements(*H*[*smallest*], *H*[*i*])
      i ← *smallest*
    else
      property ← true

  return H

Heaps

26

26

Helper for downHeap Algorithm

Algorithm *indexOfMin(A, r, size)*
  Input array *A*, an index *r* (referencing an item of A), and *size* of the heap stored in *A*
  Output the rank of element in *A* containing the smallest value

  smallest ← r
  left ← 2*r
  right ← 2*r + 1
  if left $\leq$ size $\wedge$ key(*A*[left]) < key(*A*[smallest]) **then**
    smallest ← left
  if right $\leq$ size $\wedge$ key(*A*[right]) < key(*A*[smallest]) **then**
    smallest ← right
  return *smallest*

Heaps

27

27

Analysis of Heap Operations

- ◆ Upheap()
- ◆ Downheap()
- ◆ insertElem(element)
- ◆ removeMin()
  - removes element with minimum key if a min-heap (need a comparator for items)
  - removeMax() removes element with maximum key if a Max-Heap (parent is greater or equal to its children)

- ◆ The Heap data structure is the basis of the HeapSort and is the basis of the proper implementation of a PriorityQueue (covered tomorrow)

Heaps

28

28

Main Point

1. A heap is a binary tree that stores sortable elements at each internal node and maintains *heap-order* and is *complete*. Heap-order means that for every node *v* (except the root), $key(v) \geq key(parent(v))$. *Science of Consciousess*: Pure consciousness is the field of wholeness, perfectly orderly, and complete. Through regular TM practice we release stress and automatically develop the qualities of the unified field in our lives.
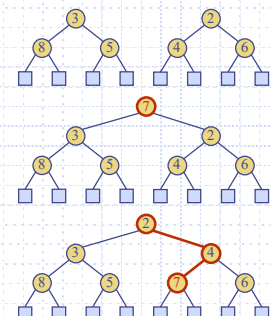
Heaps

29

29

Merging Two Heaps

- ◆ We are given two heaps and a key *k*
- ◆ We create a new heap with the root node storing *k* and with the two heaps as subtrees
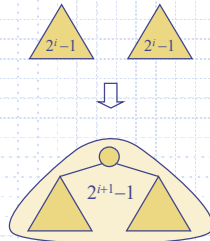- ◆ We call downHeap to restore the heap-order property

Priority Queues & Sorting

30

30

5

Bottom-up Heap Construction (§ 2.4.4)

- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases
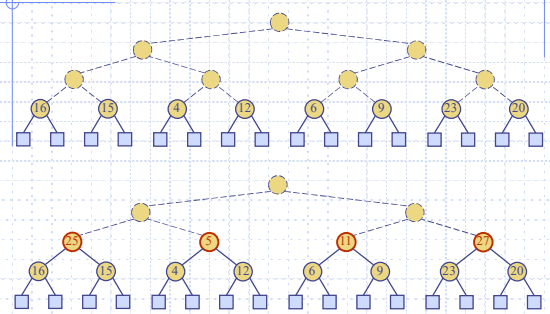- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

$2^i - 1$ $2^i - 1$

$2^{i+1} - 1$

Priority Queues & Sorting   31

31

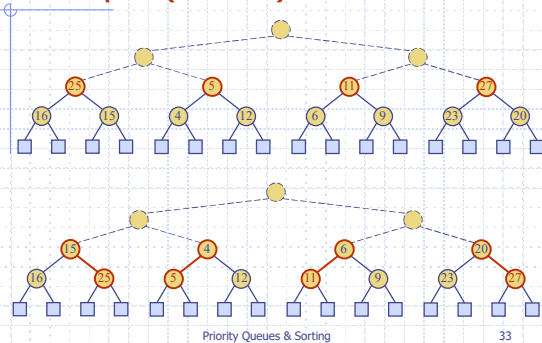## Example

Priority Queues & Sorting   32

32

Example (contd.)
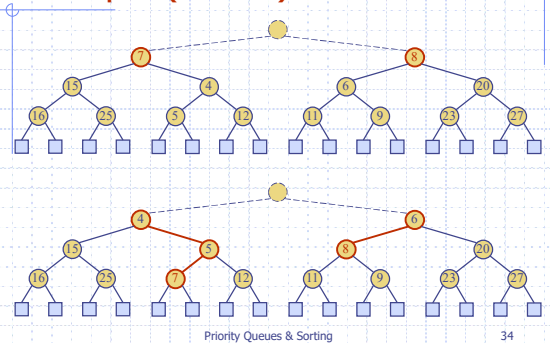
Priority Queues & Sorting   33
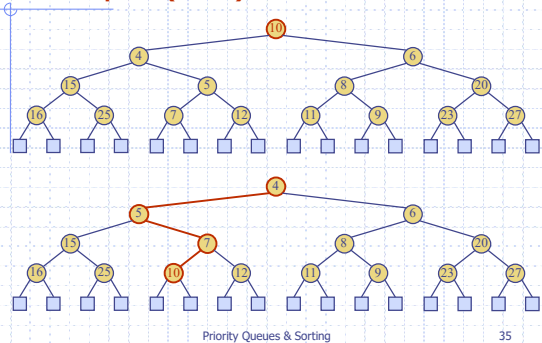
33

Example (contd.)

Priority Queues & Sorting   34

34

Example (end)

Priority Queues & Sorting   35

35

Analysis of Bottom-up Heap Construction

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort
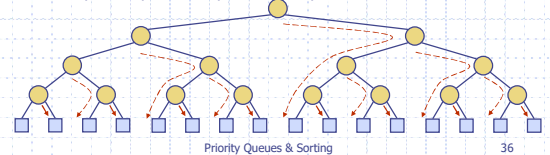
Priority Queues & Sorting   36

36

6

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The Tree ADT models a hierarchical structure between objects simplified to a parent-child relation. A Heap is a binary tree with two properties, each path from root to leaf is in sorted order and the tree is always balanced.

2. The Heap can be implemented in two ways and thus its operations will have varying efficiencies, e.g., the binary tree can be implemented as either a set of recursively defined nodes or as an array of elements.

Heaps   37

37

3. **Transcendental Consciousness** is pure intelligence, the abstract substance out of which the universe is made.

4. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously organize and govern all activities and processes in creation.

5. **Wholeness moving within itself** : In Unity Consciousness, awareness is awake to its own value, the full value of the intelligence of nature. One's consciousness supports the knowledge that outer is the expression of inner, creation is the play and display of the Self.

Heaps   38

38