

Topic: Heaps and priority queues

- The priority-queue problem
- The heap solution
- Binary trees and complete binary trees
- Running time of heap operations
- Array implementation of heap
- Operations: *Heap-insert*, *Heapify*, *Heap-extract*, *Heapsort*

The priority-queue problem

Example:

- We wish to store 8 print jobs while they wait for printer time
- We wish to assign each job a priority, say, (2, 1, 4, 1, 3, 2, 3, 5)
- We will want them to be sent to the printer in order of priority
- This is a job for a *priority queue*

Priority queue solution

- A priority queue is a collection with restricted access
- Each item in priority queue has a key value denoting its relative *priority*
- Operations: *Insert*, *Extract*
- *Extract* retrieves and deletes the item of highest priority
- Minimum queue treats low-valued key as high-priority; maximum queue prioritizes high-valued key

Sorting with a priority queue

PQ-Sort (A)

Initialize (PQ)

For $i \leftarrow 1$ to size(A)

 Insert (A_i , PQ)

For $i \leftarrow 1$ to size(A)

$A_i \leftarrow$ Extract (PQ)

- Precondition: A is any sequential or random-accessible collection
- Postcondition: A is sorted ascending
- Complexity: $n \times T_{Insert}(n) + n \times T_{Extract}(n)$

A simple, inefficient priority queue

- Store jobs in array in chronological order, with their priorities

Job	A	B	C	D	E	F	G	H
Priority	2	1	4	1	3	2	3	5

- Repeatedly retrieve, run, and delete first job found with the top priority (*Extract*)
- Complexity of *Extract*: $O(\quad)$
of *Insert*: $O(\quad)$

Trading *Insert*'s efficiency for *Extract*'s

- Store jobs in order of *priority*:

Job	B	D	A	F	E	G	C	H
Priority	1	1	2	2	3	3	4	5

(Chronological order here is ABCD...)

- Top-priority job is last in array
- Complexity of *Extract*: $O(\quad)$
of *Insert*: $O(\quad)$

Discussion problems

- What are complexities of priority queues based on *linked lists*:

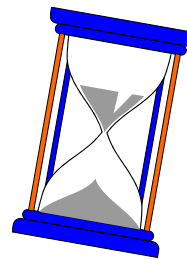
	<i>Insert</i>	<i>Extract</i>
- w/unordered list	$O(_)$	$O(_)$
- w/ascending list	$O(_)$	$O(_)$
- Could a doubly linked list improve a priority queue's efficiency?
- Can the *sum* of the complexities of *Insert* and *Extract* be improved?
- How does priority queue differ from queue data structure?

Priority-queue design problem

- Can a priority queue be built that attains running times
 - Better than $O(n)$ for *Insert* ...

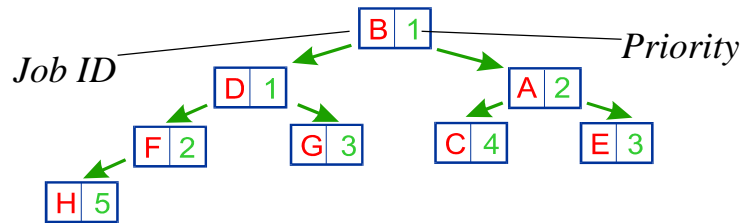
and

 - Better than $O(n)$ for *Extract*
- Can it be done using an array?



A more efficient priority queue

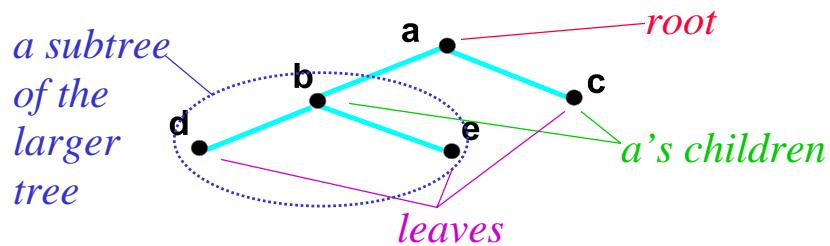
- In a minimum heap, the lowest-valued item is always at the front of queue (top of heap)



- A maximum heap reverses order rule
- Claim: this feature assures fast insertion and fast retrieval of the top-priority item

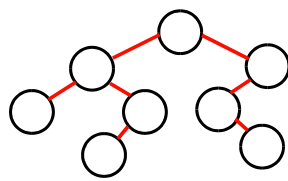
A tree is a branching structure

- In a tree, a node may be linked to more than one other node
- In a *binary tree*, each node may have up to two *child nodes*:
 - a left
 - a right

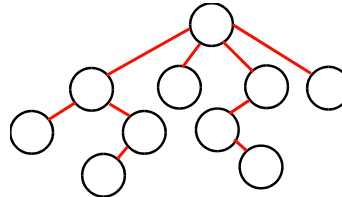


Parent and child nodes in a binary tree

- each node is adjacent to, and *parent* of, 0, 1 or 2 nodes that are its *children*
- each node except the *root* is adjacent to exactly one parent node



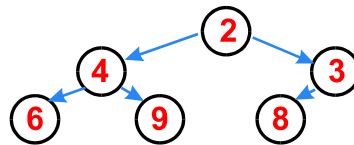
A binary tree



Not a binary tree

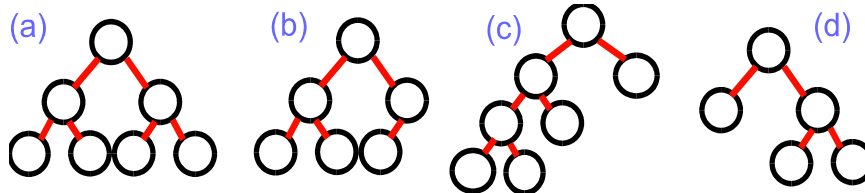
A heap is a kind of tree

- **The heap property** (for minimum heaps):
The priority value stored in each node is less than or equal to the value stored in either child
- Heap property applies to *all* nodes of a heap
- The heap property ensures that highest priority node is in root at all times



A heap is a kind of *complete binary tree*

- A complete binary tree has all levels full except possibly the bottom one, which is filled from the left
- Which are complete binary trees?



Data Structures

David Keil 8/05 13

Array implementation of complete binary tree

- Root is $A[1]$
- $A[i]$'s left child is $A[2i]$;
- its right child is $A[2i + 1]$
- Parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$
- If n is size, height is $\lceil \log_2 n \rceil$

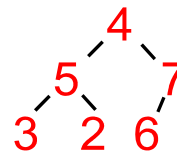
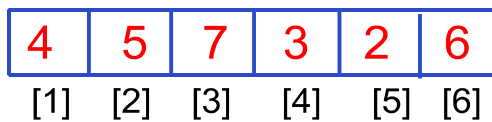
Java

$A[0]$

$A[2*i + 1]$

$A[2*i + 2]$

$A[(i-1)/2]$

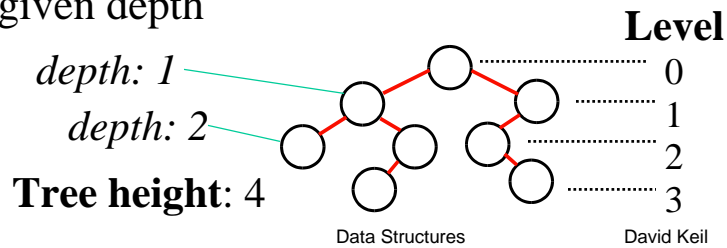


Data Structures

David Keil 8/05 14

Node depth and tree height

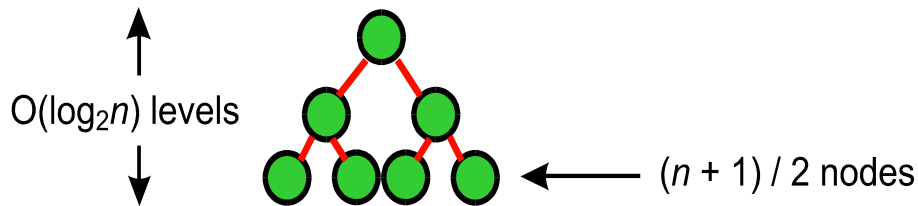
- **depth** or **level** of a node in a binary tree: length (number of edges) of the (unique) path from the root to the node
- **height** of tree: the number of nodes in the longest path from root to a node
- A **level** in a tree is the set of nodes of a given depth



Data Structures

David Keil 8/05 15

Height of a complete binary tree with n nodes is $O(\log_2 n)$



- ...because size n approximately doubles with each level added
- Or, $2^{height-1} \leq n \leq 2^{height} - 1$

Data Structures

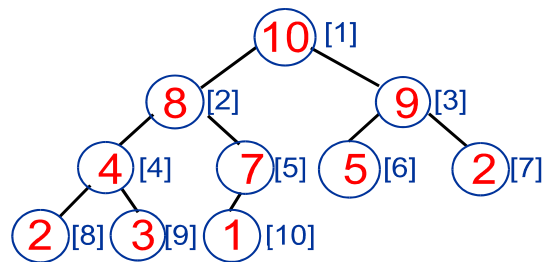
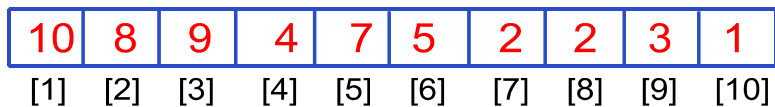
David Keil 8/05 16

Array implementation of heap

- Uses array implementation of complete binary tree
- Pointers are *not* used
- Left-right relationships not significant
- Running time for access to
 - root, from root? _____
 - a node's parent, given node's subscript? _____
 - farthest node from root? _____
 - root, from a leaf? _____
 - leaf, from root: _____

A maximum heap

- Revised heap property: for all i up to array size, $A[i] \leq A[i / 2]$



The *Heapify* operation

- $Heapify(A,i)$ applies to one node, with subscript i , of a heap stored in array A
- It assumes that subtrees with roots $Left-child(i)$ and $Right-child(i)$ already have the heap property
- Node $A[i]$ may meet or violate heap property
- $Heapify(A,i)$ causes the subtree with root subscript i to have the heap property

Intuition for *Heapify*

- The algorithm drags the value at a selected node down to its proper level where heap property applies to it
- It does this by recursively exchanging the value in the root node with the higher-priority child of its two children
- *Complexity*: Depth of complete binary tree with n nodes, i.e., $O(\text{---})$

Heapify(A,i) for minimum heap

$L \leftarrow \text{Left}(i)$
 $R \leftarrow \text{Right}(i)$
 if $L \leq \text{Heap-size}(A)$ and $A[L] < A[i]$
 $\text{smallest} \leftarrow L$
 else
 $\text{smallest} \leftarrow i$
 if $R \leq \text{Heap-size}(A)$ and $A[R] < A[\text{smallest}]$
 $\text{smallest} \leftarrow R$
 if $\text{smallest} \neq i$
 exchange $A[i]$ with $A[\text{smallest}]$
 Heapify(A, smallest)

Precondition: Left[i], Right[i] are roots of heaps

“Down-heap”

To convert an array to a heap

Overview: Heapify at every non-leaf node, starting at the bottom

Build-heap(A)

$\text{Heap-size}[A] \leftarrow \text{length}[A]$
 for $i \leftarrow \lfloor \text{length}[A] / 2 \rfloor$ down to 1
 Heapify(A,i)

Use version for min or max heap

Because leaves don't need to be heapified

Running time: $O(\quad)$

Extract value from heap

- Returns minimum value from min-heap
- Deletes that value from heap

Extract-min (A)

If $\text{Heap-size}(A) < 1$

Display message "Empty heap"

$min \leftarrow A[1]$

$A[1] \leftarrow A[\text{Heap-size}(A)]$

$\text{Heap-size}(A) \leftarrow \text{Heap-size}(A) - 1$

$\text{Heapify}(A, 1)$

Return min

[HERE GOES A SERIES
OF SLIDES FROM
TOMASSIA]

Intuition for *Heap-insert*

- Algorithm starts at a new leaf node
- Loop ascends a path toward root
- New node's value is in effect swapped upward until it reaches proper level

Insertion into a min-heap

Heap-insert (A, key)

$Heap\text{-size}(A) \leftarrow Heap\text{-size}(A) + 1$

$i \leftarrow Heap\text{-size}(A)$

while $i > 1$ and $A[\text{Parent}(i)] > \text{key}$

$A[i] \leftarrow A[\text{Parent}(i)]$

$i \leftarrow \text{Parent}(i)$

$A[i] \leftarrow \text{key}$

Up-heap

Running time: O()

[HERE FOLLOWS 1-1/2
PAGES WITH HEAP-
INSERTION ETC FROM
TOMASSIA]

To sort an array using a max-heap

Heapsort(A)

Build-heap(A)

for $i \leftarrow \text{length}(A)$ downto 2

 exchange $A[1]$ with $A[i]$

$\text{Heap-size}(A) \leftarrow \text{Heap-size}(A) - 1$

$\text{Heapify}(A, 1)$

- *In plain language...*
Repeatedly extract the highest value from heap and put it at front of a growing sorted array just after the heap
- *Complexity: $O(\quad)$*

Summary of heap implementation of priority queue

- Uses array to manage memory
- Insertion, *Extract-min* are logarithmic-time
- Enables $n \lg n$ sorting time
- Further study: mergeable heaps, e.g., binomial heap