Lecture 8

Instructor: Subrahmanyam Kalyanasundaram

9th September 2019

Plan

- Last class, we saw 2-3-4 trees (or (2,4)-trees)
- We saw that the INSERT and DELETE procedures

Plan

- Last class, we saw 2-3-4 trees (or (2,4)-trees)
- We saw that the Insert and Delete procedures

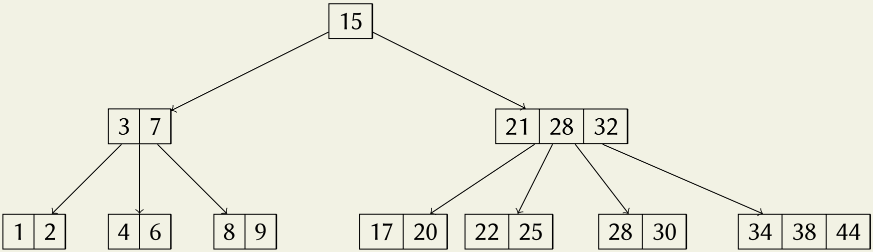
- Today, we see B-trees
- After that, we see binary heaps

Exam on Thursday, 12 Sep

2-3-4 Trees

- Search trees, but not binary search trees
- Each node can store 1, 2, or 3 keys
- If a node stores d keys, then it has $d + 1$ children
- All leaf nodes are NIL nodes
- All leaf nodes are at the same level

Example



No NIL nodes are shown above

2-3-4 Trees

- What can we say about the height of a 2-3-4 tree?
- $1/2 \log(n+1) \leq h \leq \log(n+1)$
- All operations, query and modify, are $O(\log n)$

2-3-4 Trees: Implementation

Each node contains:

- d, the number of keys in the node
- $x_1, x_2, \ldots, x_d$, the keys in increasing order
- The pointers to the $d + 1$ children
- A bit that indicates whether the node is an external node

(a, b)-tree

- 2-3-4 Trees are $(2, 4)$-trees
- In (a, b)-trees, each node has at least $a - 1$ and at most $b - 1$ keys
- So each node has at least a and at most b children
- The lower bound of $a - 1$ is **not** applicable to the root

(a, b)-tree

- 2-3-4 Trees are $(2, 4)$-trees

- In (a, b)-trees, each node has at least $a - 1$ and at most $b - 1$ keys
- So each node has at least a and at most b children
- The lower bound of $a - 1$ is **not** applicable to the root

- Exercise: Show that $2(a - 1) \leq b - 1$ has to be satisfied.

B-Tree

- B-Trees are (a, b)-trees with large values of a and b
- In a large database, the tree may be stored in the secondary memory
- Accessing a “page” takes time
- It helps if the entire page is a node

Insert

- Search for the key, insert at leaf, may need to recurse up

- The procedure for Insert in CLRS avoids recursing up
- This procedure splits every full node pre-emptively while searching
- If the leaf node needs to be split, then the parent is sure to have room to accommodate the median

Insert

- Search for the key, insert at leaf, may need to recurse up

- The procedure for Insert in CLRS avoids recursing up
- This procedure splits every full node pre-emptively while searching
- If the leaf node needs to be split, then the parent is sure to have room to accommodate the median

- Avoids the upwards recursion!

Delete

- This also can be executed in one-pass
- During the search for the node, pre-emptively merge the nodes that have min no. of elements, and can't borrow

Delete

- This also can be executed in one-pass
- During the search for the node, pre-emptively merge the nodes that have min no. of elements, and can't borrow
- Exercise: Read the procedure in CLRS

Heaps

Heap

A max-heap supports the following functions:

- Insert(val) – Inserts val into the heap.
- ExtractMax() – Returns and removes the maximum element from the heap.

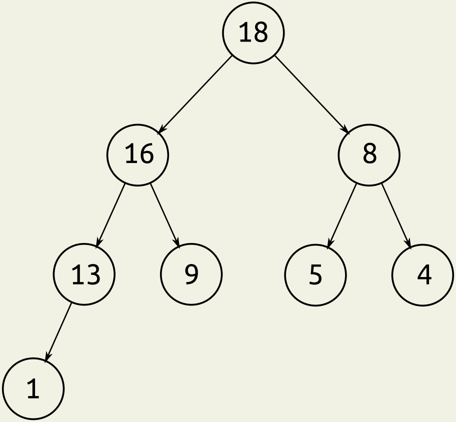
Binary max heap

A binary max-heap satisfies the following properties:

1. Structural Property: Is a complete binary tree except possibly for the lowest level, which is "left-filled".
2. Heap Property: The value of a node is greater than that of both its children.

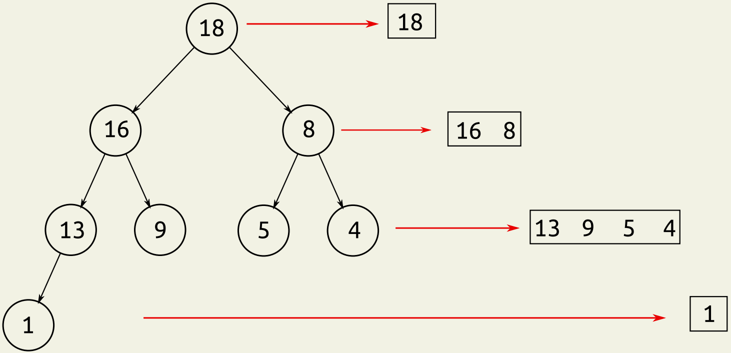
Data Structure

Heaps are usually implemented using arrays.



Data Structure

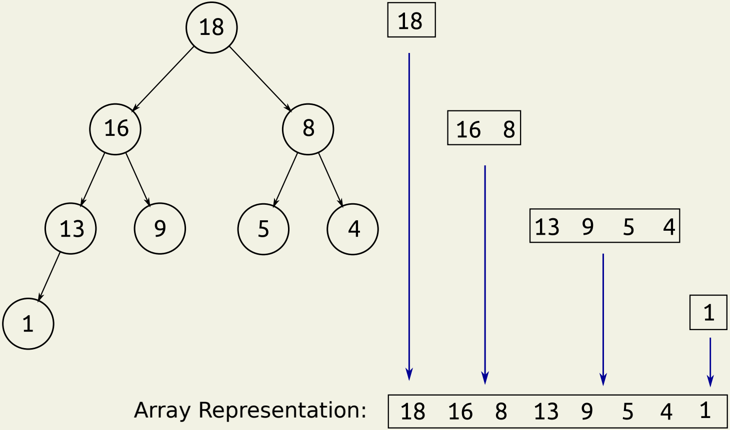
Read off from top to bottom, left to right.



Array Representation:

Data Structure

Read off from top to bottom, left to right.



Questions

About Heaps:

1. How many nodes does a height h heap have? (both bounds)
2. What is the maximum height of a heap with n nodes?

About the array implementation:

1. What is the array index of the children of the node at $A[i]$?
2. What is the array index of the right sibling of the node at $A[i]$?

Heaps using arrays

Typically, a heap is built starting with an arbitrary array:

- Procedure BuildHeap(Array A) – Takes an array and rearranges the elements to form a heap.

In Object Oriented languages, BuildHeap is essentially the *Constructor* of class Heap.
The procedure BuildHeap works by using a method called Heapify(*node*).

Heapify

The HEAPIFY(*node*) procedure:

- If *node* violates the heap property:
  1. Swap value of *node* with the largest of its two children.
  2. Call HEAPIFY on the child replaced.
- Else, do nothing and return.

Heapify

The HEAPIFY(*node*) procedure:

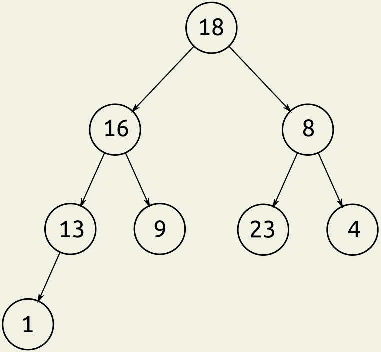
- If *node* violates the heap property:
  1. Swap value of *node* with the largest of its two children.
  2. Call HEAPIFY on the child replaced.
- Else, do nothing and return.

Note:

- The Heapify procedure assumes that both the subtrees under *node* are already heaps.
- It merely resolves the possible conflict between the value at *node* and its children and recurses.

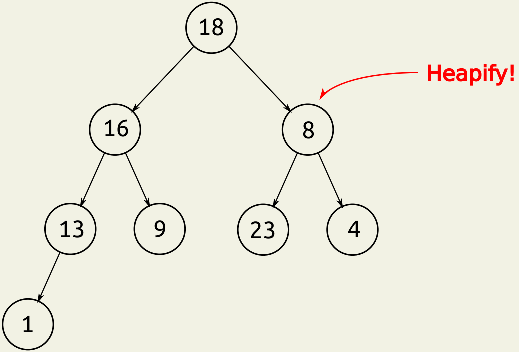
Heapify

Example:



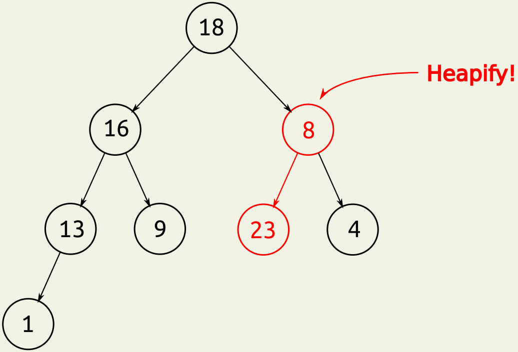
Heapify

Example:



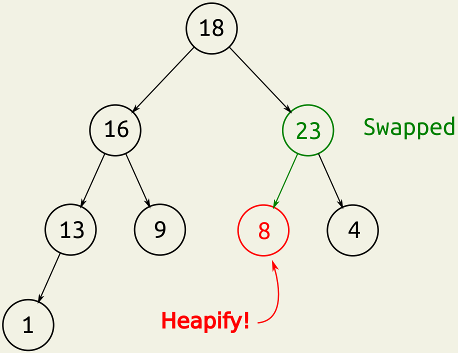
Heapify

Example:



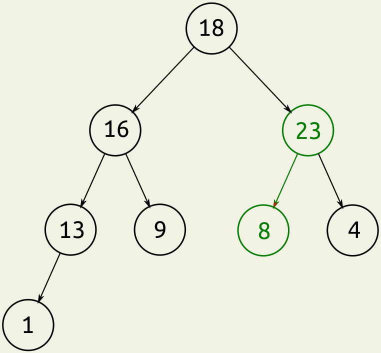
Heapify

Example:



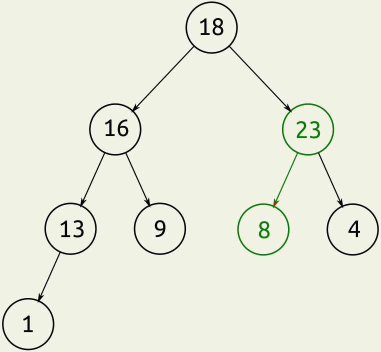
Heapify

Example:



Heapify

Note that Heapify only resolves conflicts downwards.



Building a Heap

Two ways:

- William's method: Take each element and use INSERT procedure.
- Floyd's method: Take all elements in an arbitrary array. Heapify repeatedly.

Building a Heap

The procedure BuildHeap(A) by Floyd is the following:

- For i from n to 1:
  - Heapify(i)

Building a Heap

The procedure BuildHeap(A) by Floyd is the following:

- For i from n to 1:
  - Heapify(i)

Note: Indices $n/2$ to n form leaves of the heap.
The leaves are already heaps (trivially).
Hence it suffices to run the above loop from $n/2$ to 1.

Analysis of Floyd's Method

- We need to do $n/2$ Heapify operations
- Each Heapify can take $O(\log n)$ time
- So total time is $O(n \log n)$

Analysis of Floyd's Method

- We need to do $n/2$ Heapify operations
- Each Heapify can take $O(\log n)$ time
- So total time is $O(n \log n)$

- But most of the Heapify operations are small
- We have $n/2$ nodes at height 1, $n/4$ nodes at height 2 and so on
- It can be shown that BuildHeap(A) takes only $O(n)$ time

Heap Sort

- Given an array A:
- Run BuildHeap(A)
- Repeatedly do ExtractMax()
- What is the total time?

Exercises

Write the following procedures:

- INSERT(*val*):
  - Insert new value as the last element in the array.
  - Repeatedly Heapify *upwards* from the new element.
- EXTRACTMAX(): Swap positions of root with last leaf. Heapfiy at new root.