Unit 13 - Heap

CS 201 - Data Structures II
Spring 2023
Habib University

Syeda Saleha Raza

Motivation

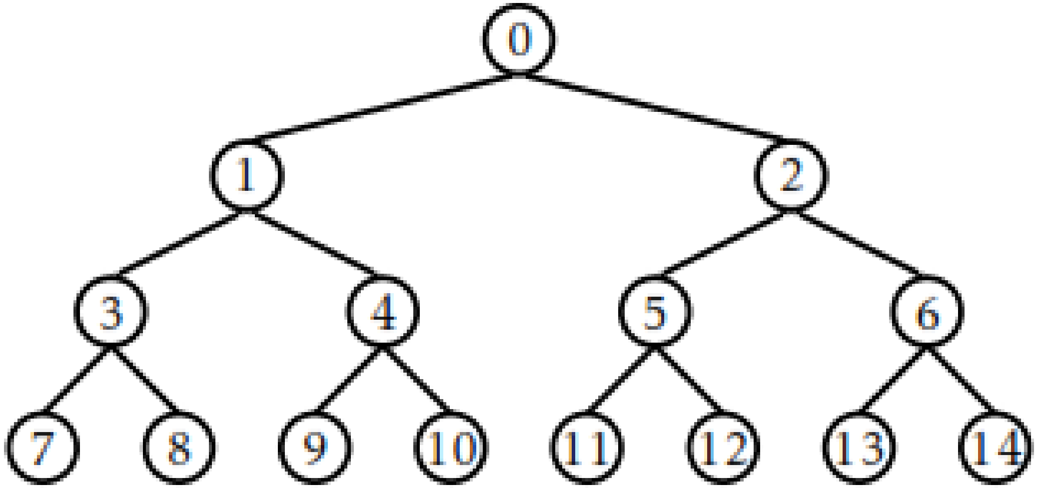
- Efficient implementation of priority queue

Heap

- Heap is a binary tree (**NOT BST**)
- Heap:
  - **Completeness** Property: Heap has restricted structure. It must be a complete binary tree .
  - **Ordering** Property: Relates parent value with that of its children
- MaxHeap property: Value of parent must be greater than **both** its children
- MinHeap property: Value of parent must be less than **both** its children
- Heap with n elements has height $O(\lg n)$

http://saravanan-thirumuruganathan.github.io/cse5311Fall2014/slides/7_Heap_UnionFind/7_Heap_UnionFind.pdf

Heap

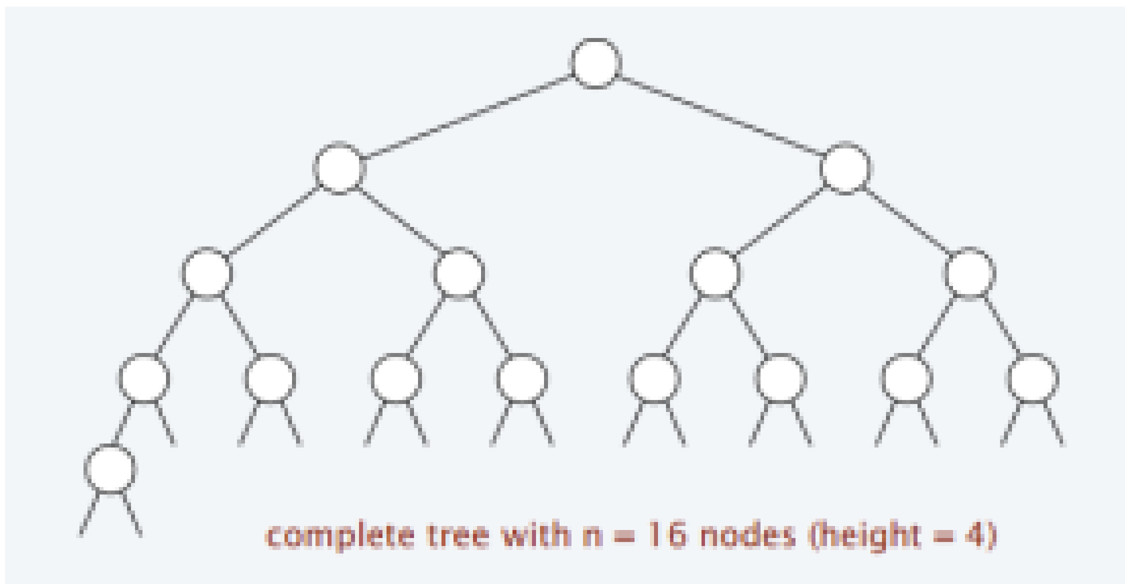


Heap Property

***Heap-Order Property*:** In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.

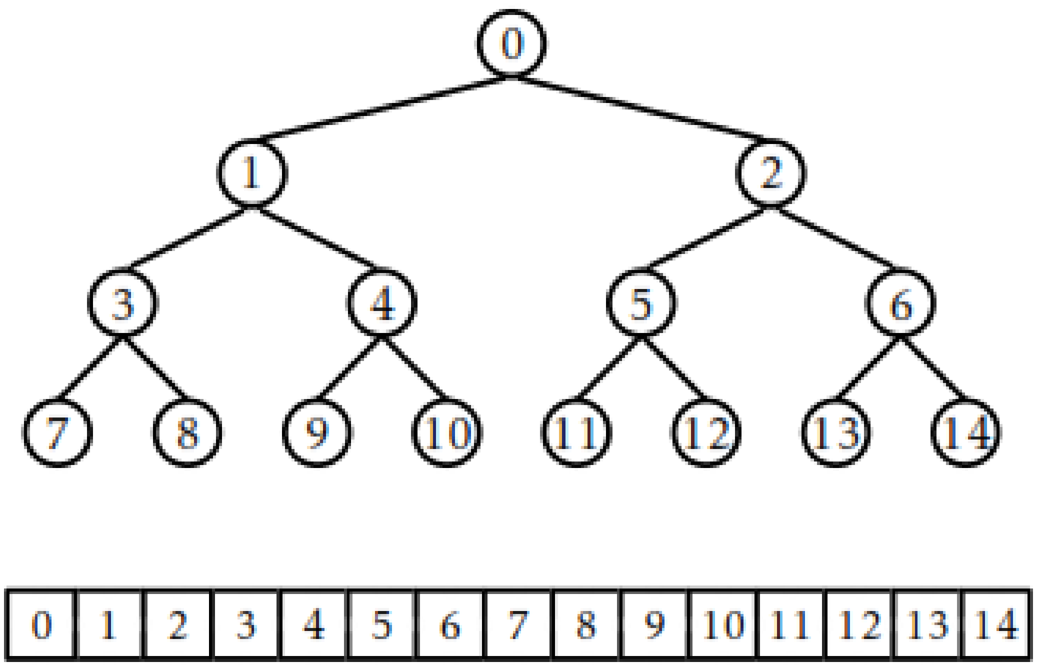
Complete Binary Tree property

- Perfectly balanced, except for bottom level
- Elements were inserted top-to-bottom and left-to-right



http://saravanan-thirumuruganathan.github.io/cse5311Fall2014/slides/7_Heap_UnionFind/7_Heap_UnionFind.pdf

Binary Heap



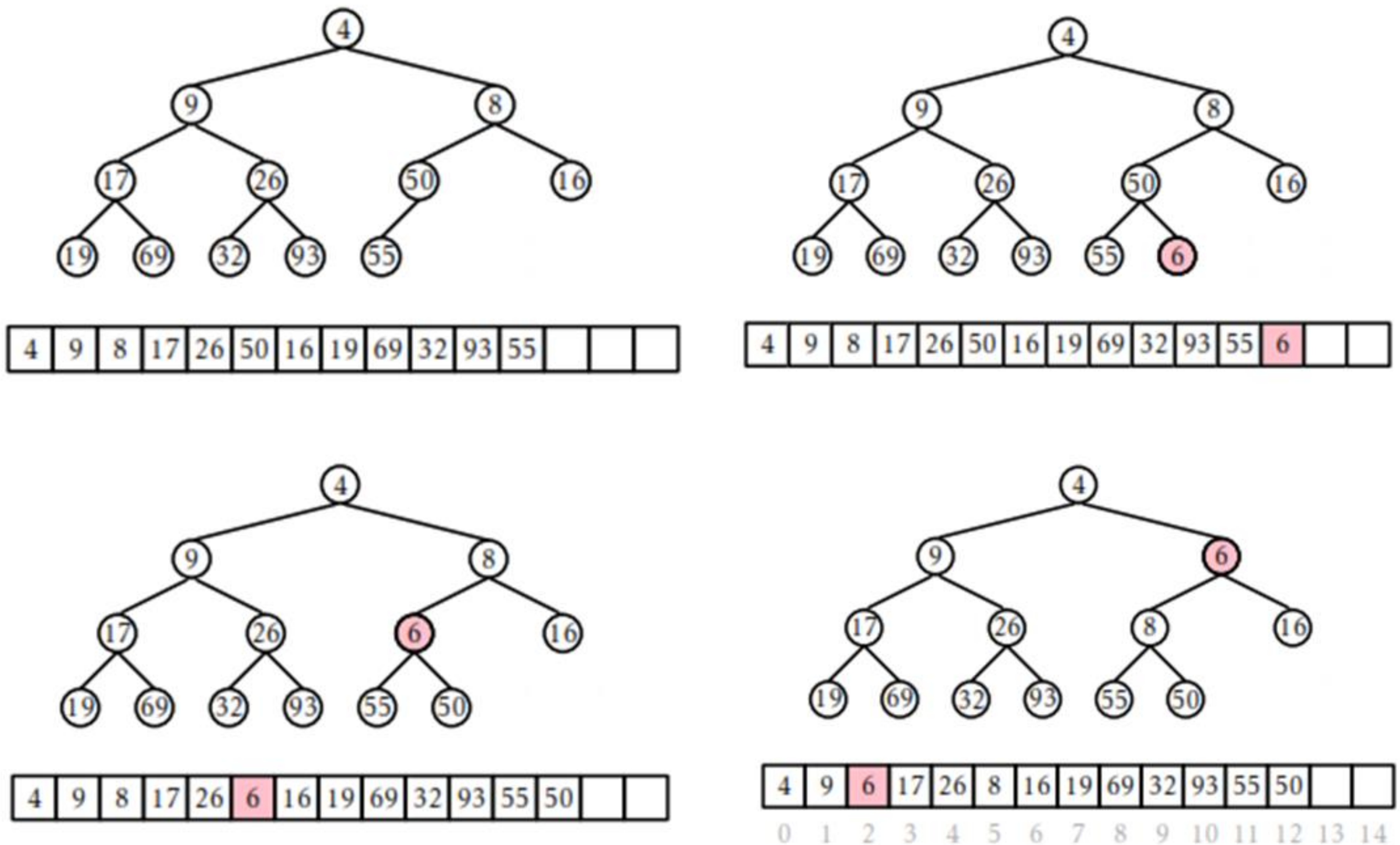
Accessing left/right child and parent

left(i)
  return $2 \cdot i + 1$

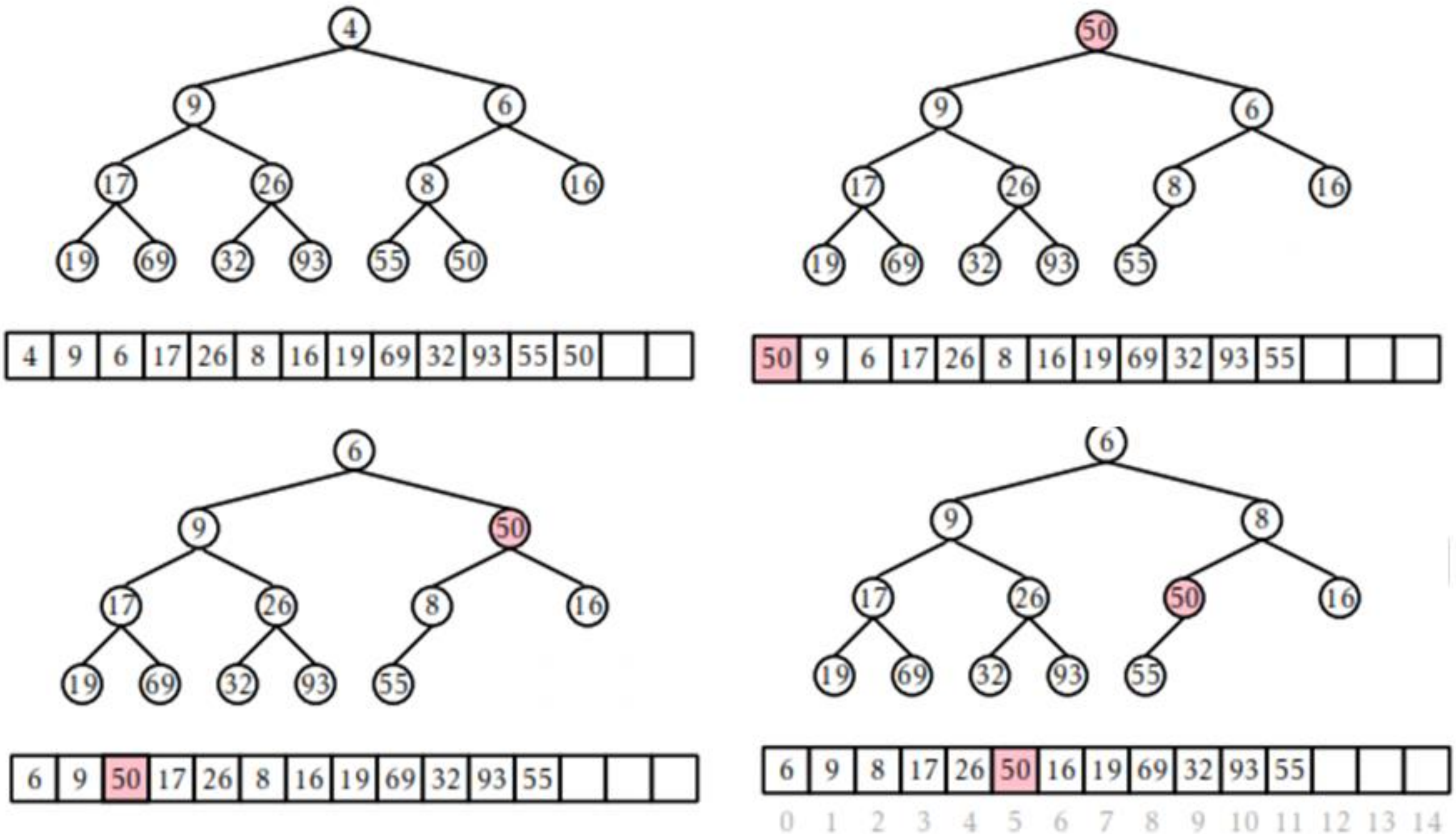
right(i)
  return $2 \cdot (i + 1)$

parent(i)
  return $(i - 1) \operatorname{div} 2$

Insertion – Bubbling up



Deletion – Trickling down



Example

- Build a heap using the following keys:
  - 9,6,14,3,7,11

Exercise

- Build a heap using the following keys:
  - 9,3,8,2,1,10,21,7

Complexity of min, remove_min and insertion

Complexity

- Insertion
- Min
- Remove_min

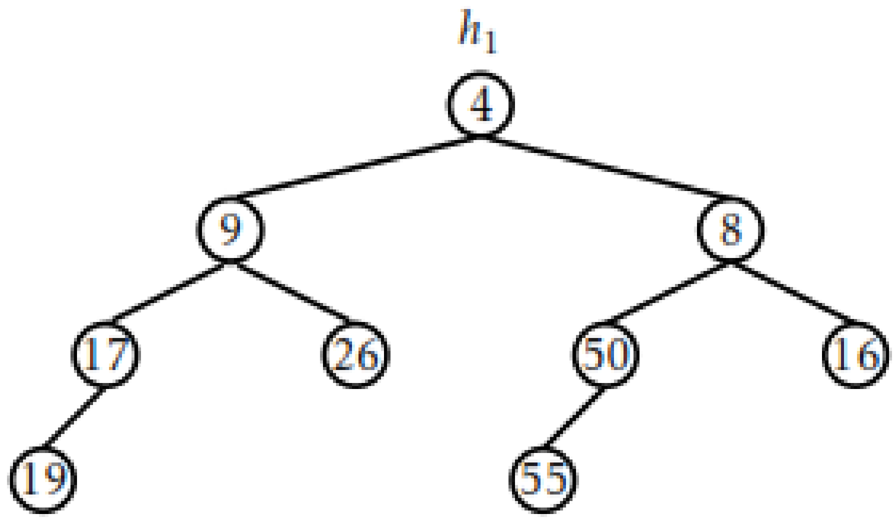
Complexity

Theorem 10.1. *A BinaryHeap implements the (priority) Queue interface. Ignoring the cost of calls to* resize(), *a BinaryHeap supports the operations* add(x) *and* remove() *in* $O(\log n)$ *time per operation.*

Meldable Heap

- A randomized meldable heap (also Meldable Heap or Randomized Meldable Priority Queue) is defined as a priority queue based data structure in which the underlying structure is also a heap-ordered binary tree. However, there are no hard and fast rules on the shape of the underlying binary tree.

Meldable Heap



Merge

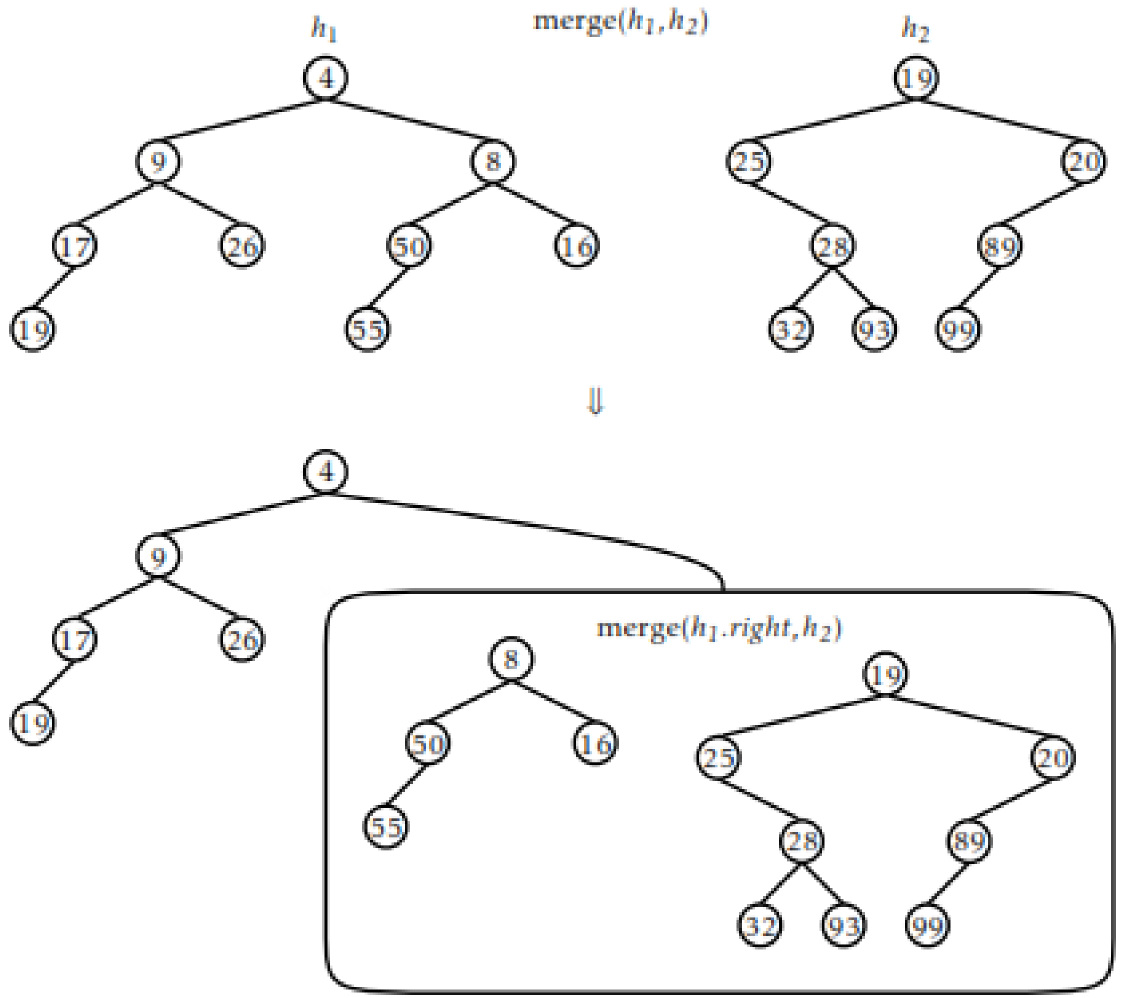


Figure 10.4: Merging h_1 and h_2 is done by merging h_2 with one of h_1.*left* or h_1.*right*.

Merge

merge(h_1, h_2)

    if $h_1 = nil$ **then return** h_2

    if $h_2 = nil$ **then return** h_1

    if $h_2.x < h_1.x$ **then** $(h_1, h2) \leftarrow (h_2, h_1)$

    if random_bit() **then**

        $h_1.left \leftarrow$ merge($h_1.left, h_2$)

        $h_1.left.parent \leftarrow h1$

    else

        $h_1.right \leftarrow$ merge($h_1.right, h_2$)

        $h_1.right.parent \leftarrow h1$

    return h_1

Add(..) operation using merge

add(x)
  $u \leftarrow$ new_node(x)
  $r \leftarrow$ merge(u, r)
  $r.parent \leftarrow nil$
  $n \leftarrow n + 1$
  return *true*

Remove() operation using merge

remove()

$x \leftarrow r.x$

$r \leftarrow \mathrm{merge}(r.\mathit{left}, r.\mathit{right})$

if $r \neq \mathit{nil}$ **then** $r.\mathit{parent} \leftarrow \mathit{nil}$

$n \leftarrow n - 1$

return x

Resources

- Data Structures and Algorithms in Python, by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013. (1st. ed.). Wiley Publishing
- Open Data Structures (pseudocode edition), by Pat Morin. Available online at http://opendatastructures.org

Thanks