Transform and Conquer

Transform and Conquer

Idea: Some problems are easier/simpler to solve after they are first transformed into another form.

Can be broken into the following techniques:

- **Simplify –** transform to a simpler or more convenient instance of the same problem
- **Convert** – transform to a different representation of the same problem
- **Reduce** – transform to an instance of a different problem for which an algorithm is already available

Transform and Conquer: Simplify

Simplification – Pre-sorting

- **Pre-sorting:** Many problems involving arrays are easier when the array is sorted
- General approach of pre-sorting based algorithms:

1. **Transform:** Sort the array
2. **Conquer:** Solve the transformed problem instance, taking advantage of the array being sorted

Instance Simplification – Pre-sorting

Examples:

- Searching
- Checking if all elements are distinct (element uniqueness)
- Computing the median (selection problem)
- Many geometric algorithms (e.g., Quick hull)
- Activity selection (Greedy)

Search in a Sorted List

- **Problem:** Search for a key k in a array A[0 … n – 1].
- **Pre-sorting-based algorithm:**
  1. Sort the array using an efficient sorting algorithm.
  2. Apply binary search.
- **Efficiency:** $O(n \log n) + O(\log n) = O(n \log n)$
- Not better than **sequential search**, but if the array is **static** and search is performed many times, then it may be worth the extra effort of pre-sorting (**amortised**).

Is Pre-sorting Worth the Effort?

Pre-sorting search:

1. Merge sort uses $n \log n$ comparisons on average
2. Binary search uses $\log n$ comparisons on average

Linear search in an unsorted array uses $n/2$ comparisons on average

Example – an array of size 10^6 would require 20×10^6 steps to sort. If we looked for values in the array a 100,000 times it would take 100,000*20 = 2,000,000 steps – a total of 2.2×10^6 steps. If we did not sort it and used linear search each time, it would take $10^5 \times 0.5 \times 10^6 = 5 \times 10^{10}$ steps

Uniqueness Checking

Problem: check whether all elements of an array A[0..N-1] are unique

- Without sorting
  - `for i from 0 to N – 2`
    - `for j from i + 1 to N – 1`
      - `if (A[i] == A[j]) return false`
  - `return true`

Uniqueness Checking

Problem: check whether all elements of an array A[0..N-1] are unique

- With sorting
  - `for i from 0 to N – 2`
    - `if (A[i] == A[i+1]) return false`
  - `return true`

Uniqueness Checking

Problem: check whether all elements of an array A[0..N-1] are unique

- Complexity
    - Without sorting
        - O(N^2)
    - With sorting
        - O(NlgN) + O(N) = O(NlgN)

Computing the Median

Problem: given an array A[0..N-1], return the median element M (i.e., M >= half of the elements in A and M <= half of the elements in A)

- Without sorting
    - Using quick select O(N), but O(N^2) in the worst case
- With sorting
    - O(NlgN) + O(1)

Activity Selection

Problem: given an array of activities A[0..N-1]. Each activity A[i] has a start_time and finish_time. What is the maximum number of activities a single person can do? (a person cannot do two or more activities at the same time)

- Brute force approach
  - Generate all subsets of activities
  - A subset is valid if it contains **no** two activities A[i] and A[j] that overlap each other
  - Complexity: O(2^N)

Activity Selection

Greedy approach

```
sort the tasks based on their finished_time
include the first task in the result
for i = 1 to N - 1
  if A[i] does not overlap the last added task
    add A[i] to the result
    mark A[i] as the last added task
return result
```

Activity Selection

```
Input: A = [(4, 5), (2, 6), (1, 3), (6, 7)]
Sort: A = [(1, 3), (4, 5), (2, 6), (6, 7)]
Pick first task:
  Result = [(1, 3)]
Second task does not overlap, add it:
  Result = [(1, 3), (4, 5)]
Third task does overlap => skip it
Fourth task does not overlap, add it:
 Result = [(1, 3), (4, 5), (6, 7)]
```

- Complexity:
  - Sorting O(N*lg(N))
  - Going through the sorted array: O(N)
  - Final: O(N*lg(N))

Transform and Conquer: Convert

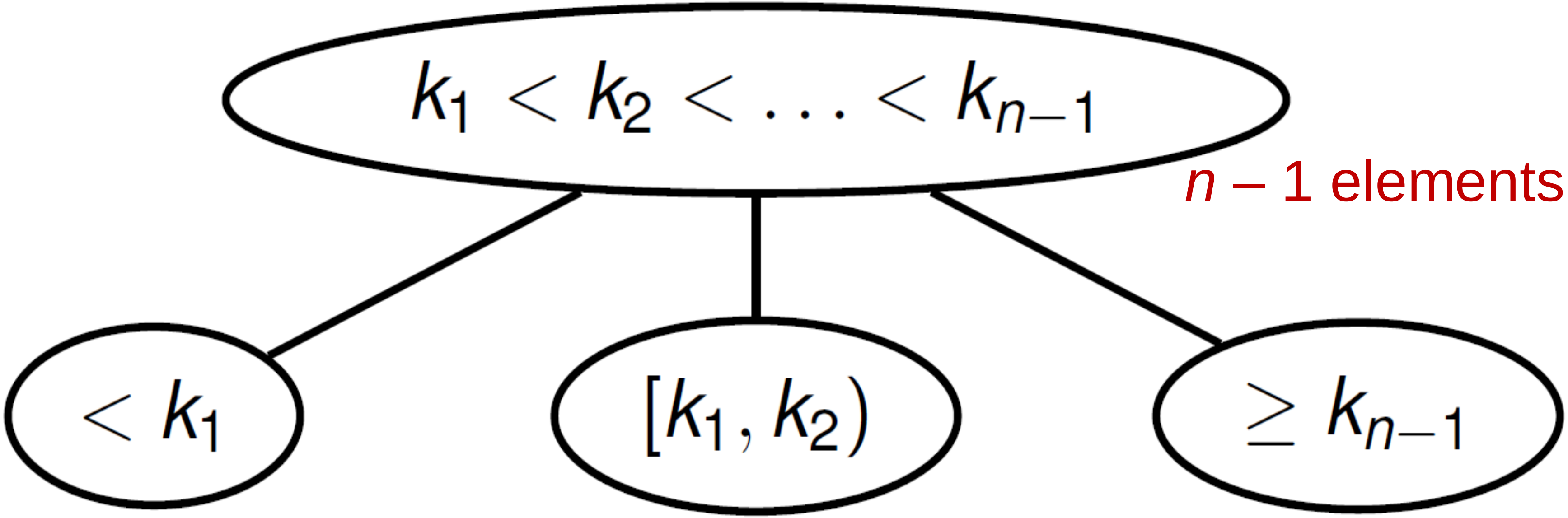
Balanced Search Trees

- Why balanced search trees are desirable?
  - Frequent insertion and deletion operations make trees go off-balance.
  - Recall: The worst-case performance using simple binary trees is dependent on the height of the tree
- As a result, a great deal of research effort has been invested in keeping binary search trees Balanced and of minimum height
- Multiway search trees: 2-3 Trees
- Binary heaps

Multiway Search Trees

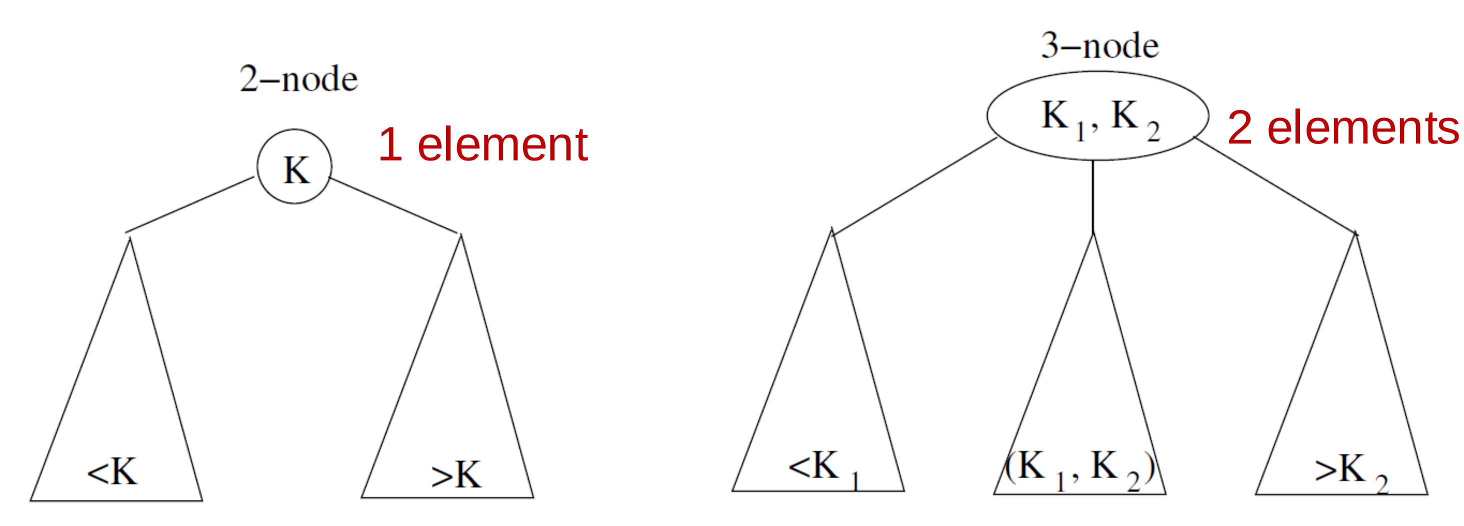
- A **multiway search tree** is a search tree which allows more than one element per node.
- A node of a search tree is called **n-node** if it contains *n* – 1 ordered elements, dividing the entire element range into n intervals.

Multiway Search Trees



2-3 Trees

- A **2-3 tree** is a search tree which mixes 2-nodes and 3-nodes to keep the tree height-balanced (i.e., all leaves are on the same level).

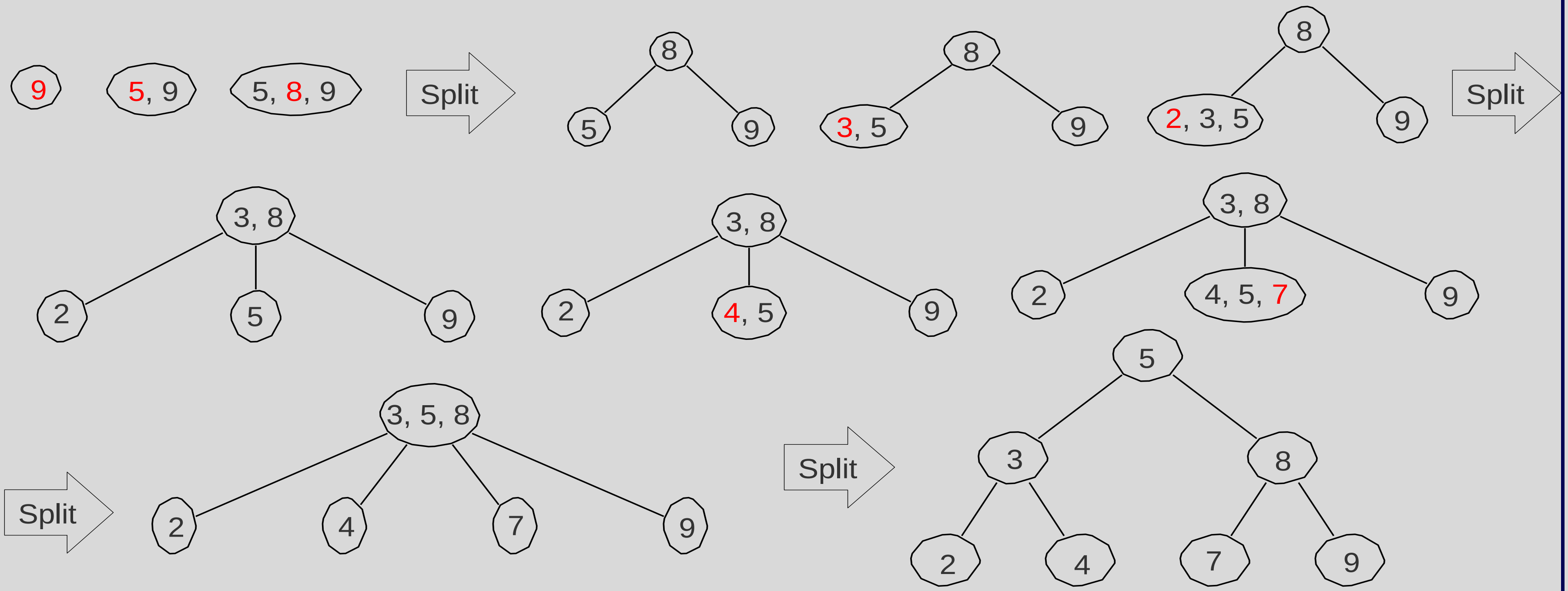


2-3 Trees – Construction

- A 2-3 tree is constructed by successive insertions of given elements, with a new element always inserted into a leaf of the tree, following **2-3 parent-child rules**
- If the leaf becomes a **4-node** (has 3 elements), it is **split into three nodes**, with the middle element **promoted** to the parent node.

2-3 Trees – Construction

- Construct a 2-3 tree for the sequence: 9, 5, 8, 3, 2, 4, 7

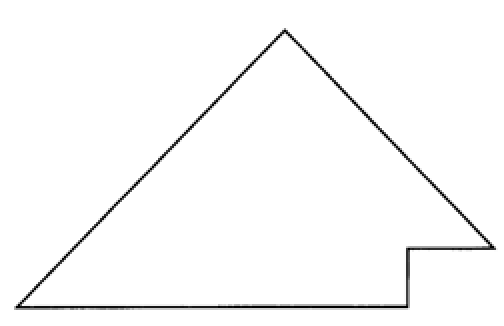


2-3 Trees – Analysis

- $\log_3(n+1) - 1 \quad \leq \quad h \quad \leq \quad \log_2(n+1) - 1$
- **Search**, **Insert** and **Delete** are all O(log*n*)
- **Rebalancing** on average is cheaper and may occur less frequently than AVL tree
- Another way to create a balanced search tree

Complete Binary Tree

A binary tree that is either full or full through the next-to-last level

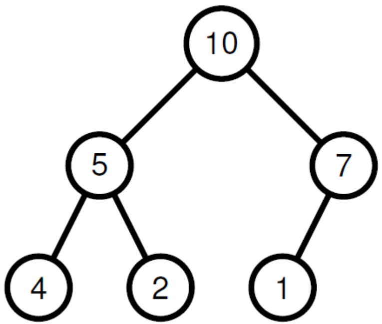


The last level is full from left to right - i.e., leaves are as far to the left as possible

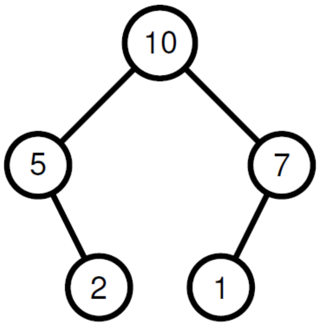
Heaps

- A heap is a binary tree that satisfies these special SHAPE and ORDER properties:
  - Its shape must be a complete binary tree
  - For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children (max heap)
  - Has heaps as subtrees
- A heap is similar to a BST but differs in two ways
  - A BST is sorted, a heap is sorted in a much “weaker” sense
  - A BST comes in many different shapes, a heap is always a complete binary tree

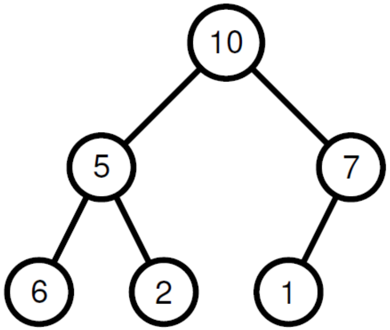
Heaps



(a) a heap



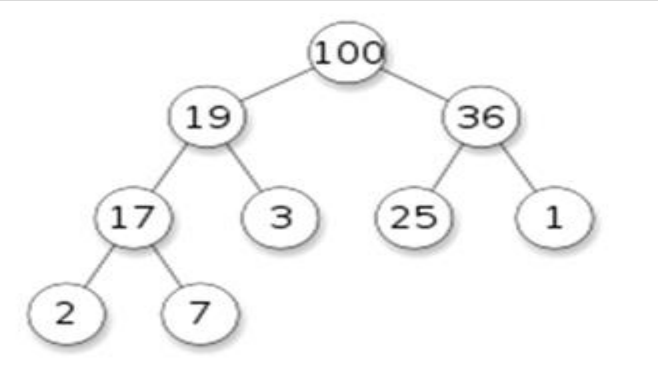
(b) **not** a heap

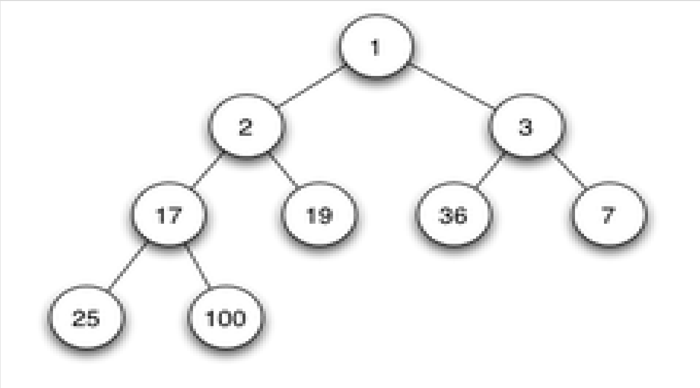


(c) **not** a heap

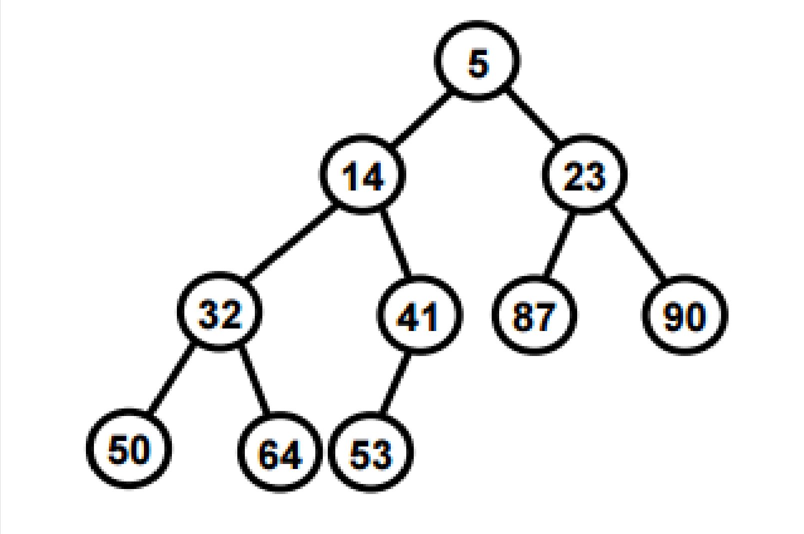
Max and Min Heap

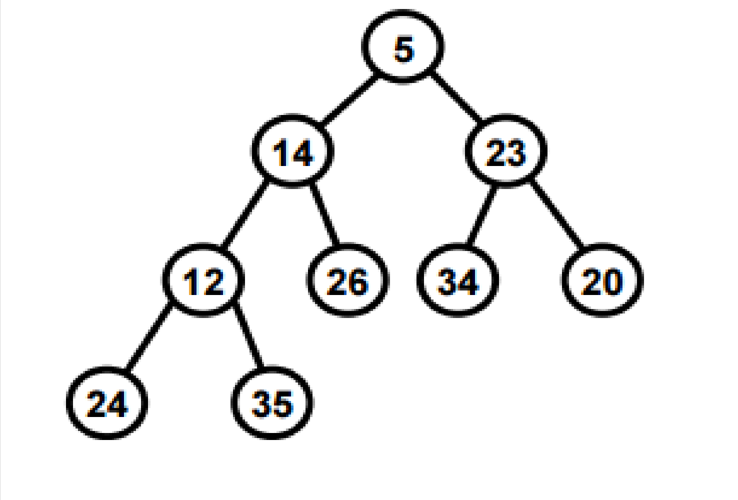
heap-order property → maintaining this property (i.e., ***heapify***) takes ***O(*****log *n*)**	
Max heap: in every subtree, root holds the largest value → access maximum in constant time	Min heap: in every subtree, root holds the smallest value → access minimum in constant time





Which of these are Min Heaps





Heaps – Applications

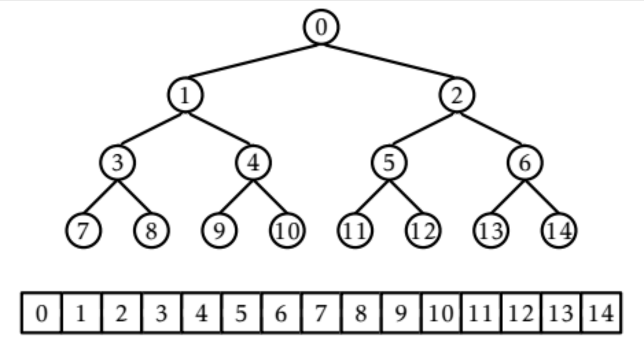
Efficient data structure for several important applications, including:

- Implement priority queues
- Finding max/min in an array of elements
- Fast implementations of graph algorithms like Dijkstra's and Prim’s algorithms
- Implement heapsort

Heap Representation

- Embed a complete binary tree into an array

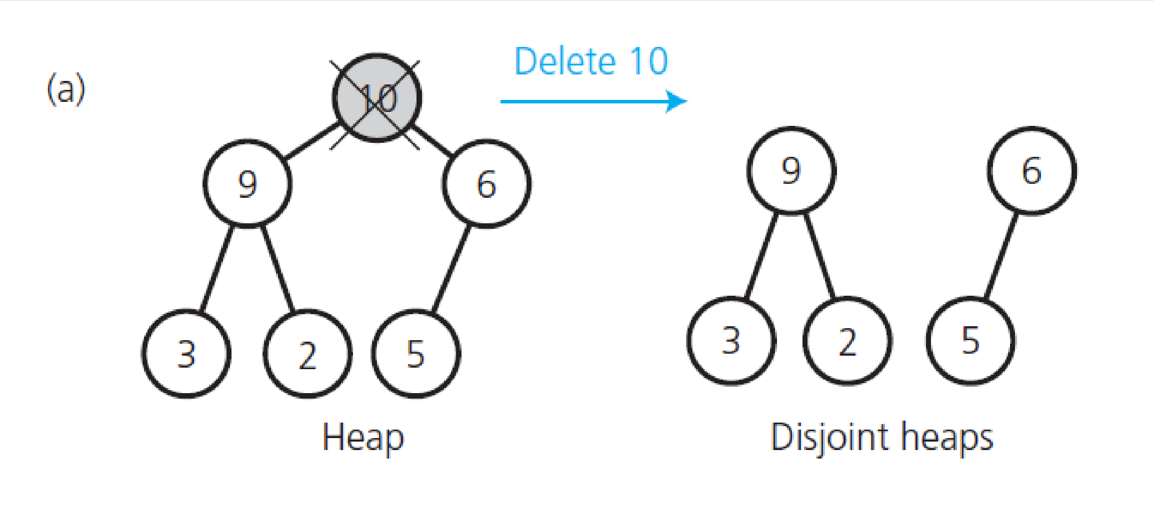
→ lays out the nodes in ***breadth-first* order** (top down, layer by layer, left to right)



  - Left child of the node at index i is at index `left(i) = 2i + 1`
  - Right child of the node at index i is at index `right(i) = 2i + 2`
  - Parent of the node at index i is at index `parent(i) = (i–1)/2`

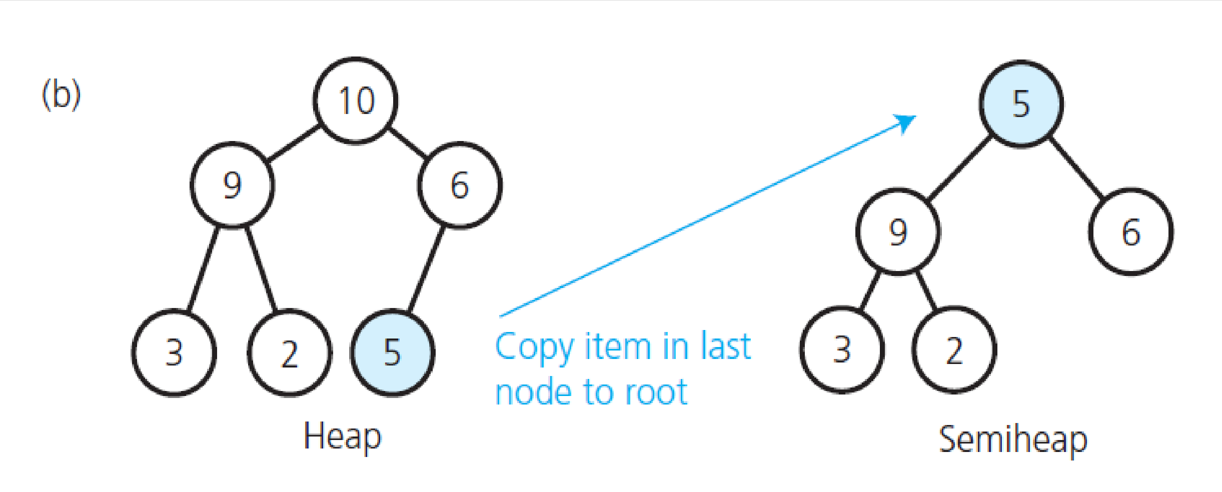
Removing the Heap's Root

- Removing the heap's root creates 2 disjoint heaps
- How can you combine these two heaps and make a new heap



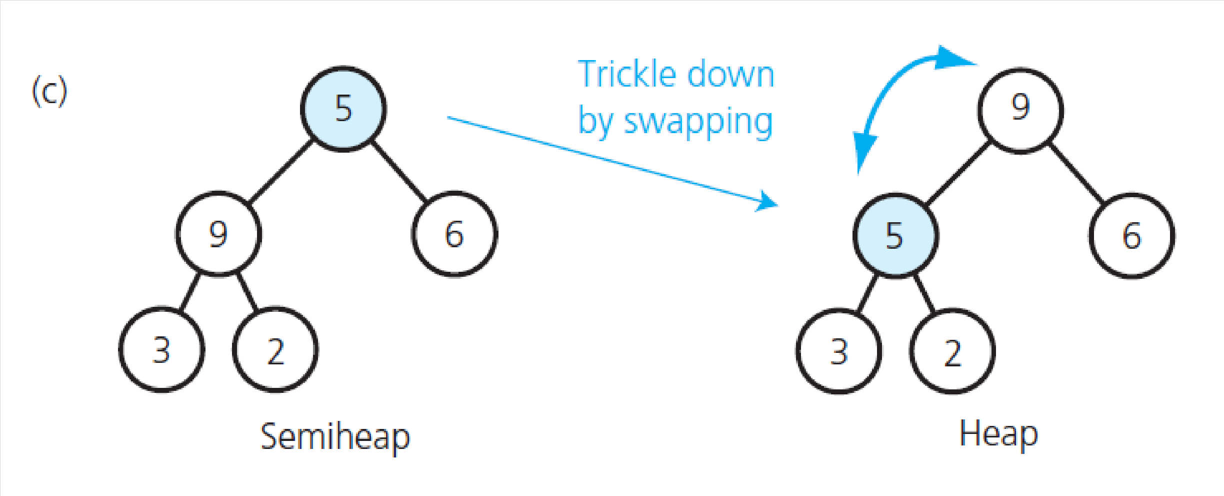
Creating a Semi-heap

Deleting the root will create two heaps. Taking the last item and copying it to the root creates a semi-heap – i.e. a heap that only the root is out of place



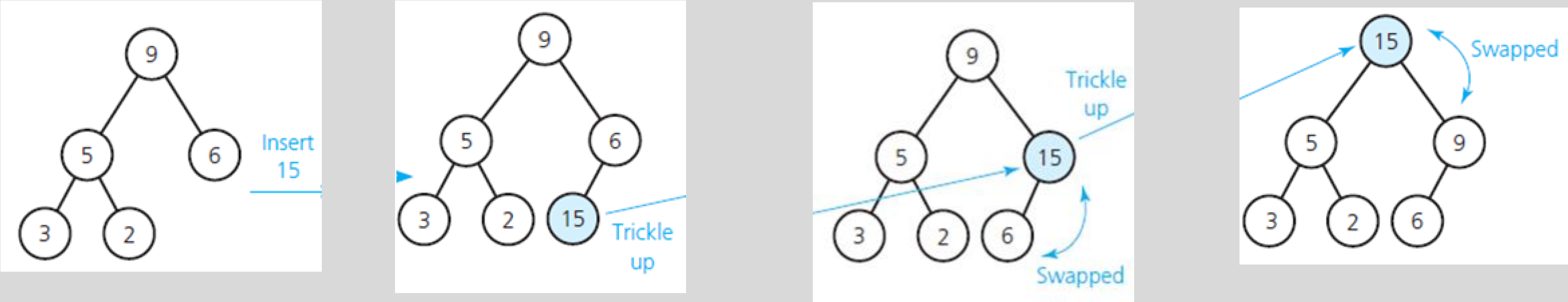
Rebuilding the Heap

To rebuild the semi-heap, swap the root with its largest child and repeat until the node is bigger than both its children



Inserting into a Heap

- Insertion into a heap is the opposite of remove. The new item is inserted into the bottom and then trickles up
- Recall that the parent of items[i] is stored in items[(i-1)/2]

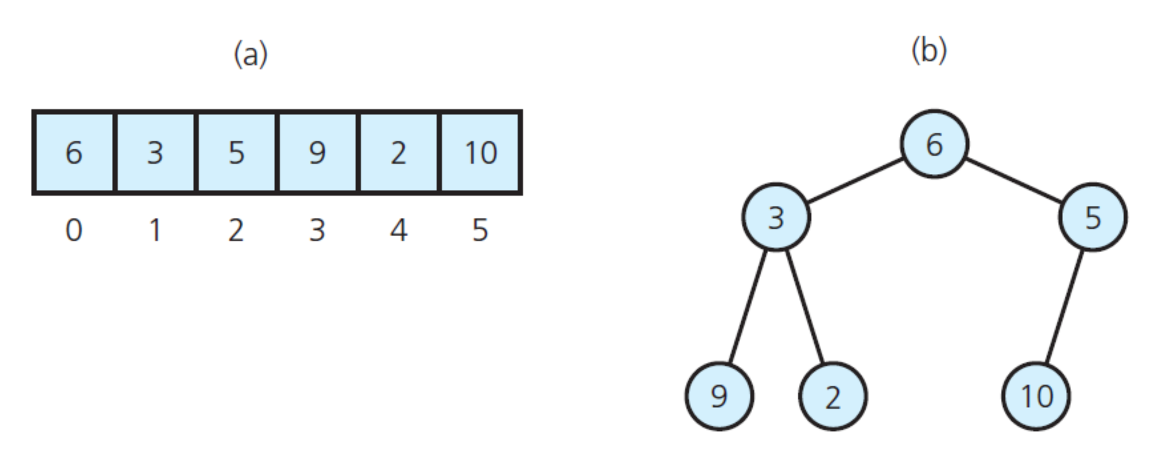


An Array and its Binary Tree

How can we convert the Binary Tree into a Max Heap?

(a) The initial contents of an array;

(b) The array's corresponding complete binary tree

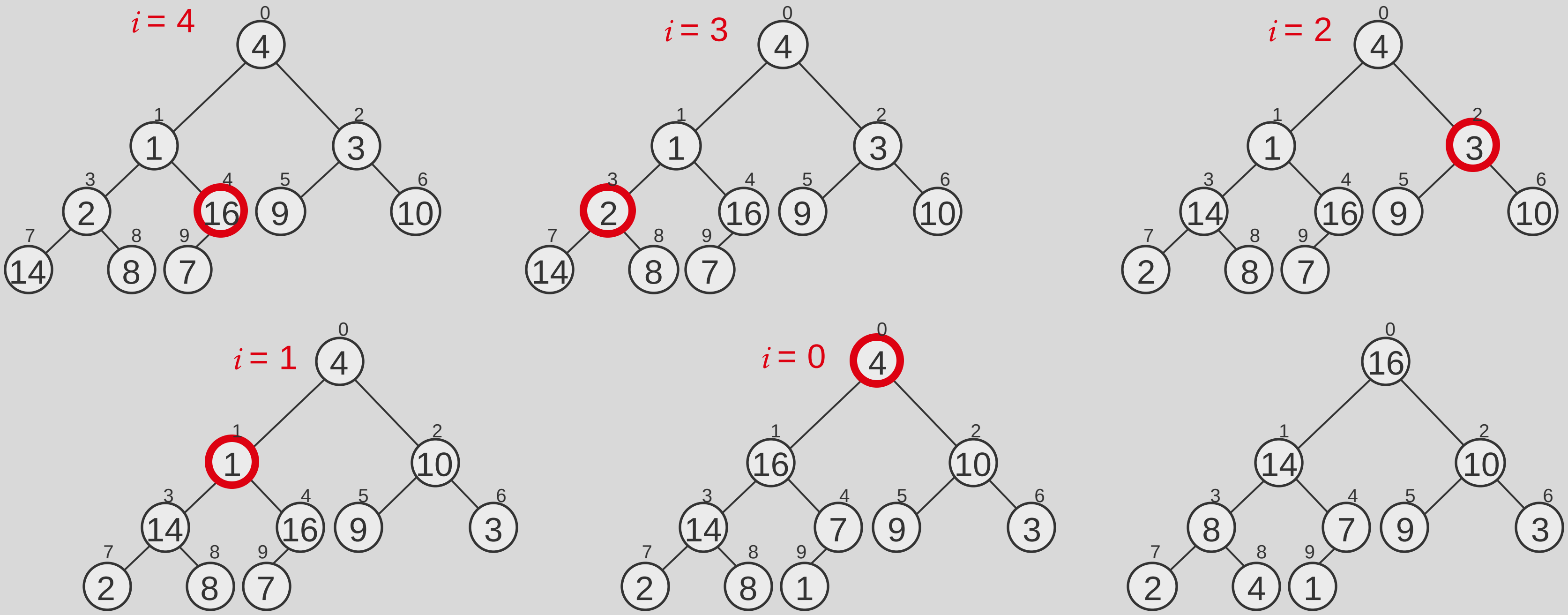


Array to Max Heap Conversion

- Done by going through the array that represents the heap from the end to the beginning
- Looking for an element that is smaller than its child, and then swapping with the biggest child recursively

Array to Max Heap Conversion

4	1	3	2	16	9	10	14	8	7



Heap Construction - Analysis

- The height of the heap is h = log$n$
- Each call to **repair** takes O(log$n$) time.
- There are n/2 ∈ O(n) such calls.
- Therefore, O(n log$n$) is an upper bound on the running time of **building the heap**.
- Note: a tighter bound for heap construction is O(n). Ref: https://en.wikipedia.org/wiki/Binary_heap#Building_a_heap

Heap Sort

- Task: sort an array A in ascending order
- Algorithm
-
```
build a heap from A

while (heap.size > 1)
  heap[0] <-> heap[size - 1]
  heap.size—
  heapify
```

Heap Sort – Analysis

Worst-Case Analysis:

- **Stage 1**: Build heap for a given list of n keys (where the number of nodes at level $i = 2^i$).

$$C_w(n) \in O(n)$$

- **Stage 2**: Repeat **dequeue** *n* times

$$\mathrm{C_w(n)} = \sum_{\mathrm{i=1}}^{\mathrm{n}} level(i) \in \mathrm{O(n \log n)}$$

Heap Sort – Analysis

- Heap sort is **not stable**.
- The total worst-case efficiency is
  - $O(n \log n) + O(n) \in O(n \log n)$.

Heap Sort – Questions

Heap sort vs. Merge sort vs. Quick sort

- **Average case:** Heap sort generally faster than Merge sort but slower than Quick sort.
- **Worst case:** Heap sort is comparable with Merge sort, but faster than Quick sort.
- **Stability:** Merge sort is only stable sorting algorithm out of the three.

Transform and Conquer: Reduce

Least Common Multiple

- **Problem:**
  - The **Least Common Multiple** of two positive integers m and n, denoted **LCM**(m, n) is defined as the smallest integer that is divisible by both m and n.
- **Example:** LCM(24, 60) = 120; LCM(5, 11) = 55

Least Common Multiple

- **Simple approach:**
  - Find the common patterns between the two numbers
  - To be more precise, compute the **common prime factors** of m and n. The LCM is the product of all the common prime factors times each **non-common factor** of n and m.

Least Common Multiple

- Compute the **common prime factors** of *m* and *n*. The LCM is the product of all the common prime factors times each **non-common factor** of *n* and *m*.
- **Example:** Find LCM(24, 60)
  - 24 = 2 x 2 x 2 x 3
  - 60 = 2 x 2 x 3 x 5
  - LCM(24, 60) = (2 x 2 x 3) x 2 x 5 = 120

Least Common Multiple

- Finding primes by brute force is inefficient. The problem can be solved by reduction using Euclid's algorithm.
- Recall that the Greatest Common Divisor (GCD(*m*, *n*)) is the product of all the common prime factors of *m* and *n*.

$$\mathrm{LCM}(m, n) = \frac{m \times n}{\mathrm{GCD}(m, n)}$$

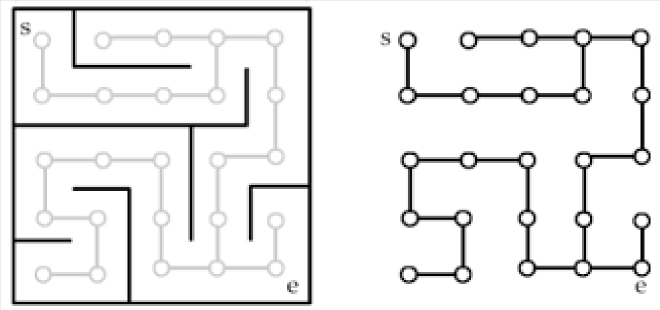
- GCD can be computed efficiently by using Euclid's method.

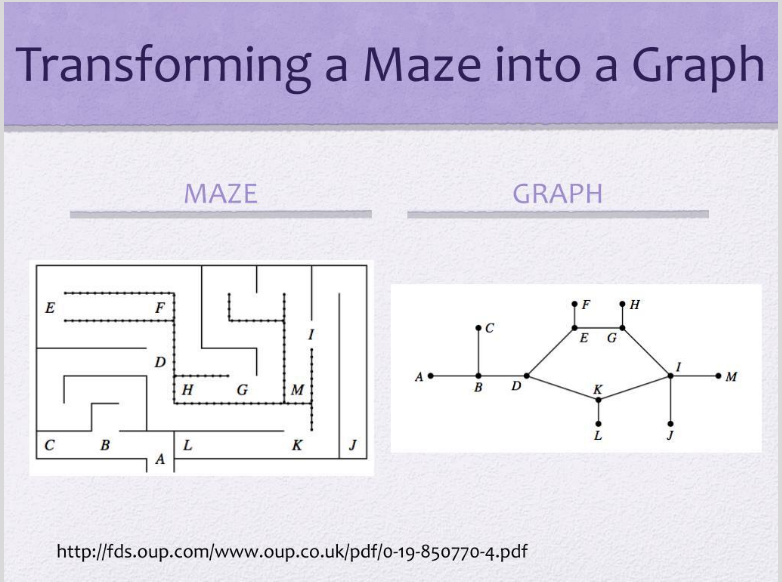
Greatest Common Divisor

- Euclid’s algorithm to find the GCD of two numbers
- `gcd(a, b)`
- `    if b == 0 return a`
- `    return gcd(b, a % b)`
- Example:
  - gcd(27, 12) = gcd(12, 3) = gcd(3, 0) = 3
  - gcd(1001, 300) = gcd(300, 101) = gcd(101, 98) = gcd(98, 3) = gcd(3, 2) = gcd(2, 1) = gcd(1, 0) = 1

Solving a Maze

- Convert a maze to a graph and then solve using DFS, BFS, Dijkstra, A* etc.





Two Water Jugs

- How to get exactly 2 liters of water from a 3 liters and a 4 liters jugs
- These jugs don’t have markings

49

Two Water Jugs - Modeling

- Model the amount of water in 2 jugs as <X, Y>
- Initially we have <0, 0>
- We want to achieve either <2, y> or <x, 2> (x,y >= 0; x<=3; y<=4)
- We can move from a state <x1, y1> to several other states <x2, y2>, <x3, y3>, <x4, y4>, etc. by taking some actions

50

Two Water Jugs - Actions

- Operation you can perform to change the state of the two jugs
  - Empty a jug: <x, y> becomes <x, 0> or <0, y>
  - Fill a jug: <x, y> becomes <x, 4> or <3, y>
  - Pour water from one jug to the other until the first one is empty or the second one is full. Assume we pour water from the first jug to the second jug
    - <x, y> becomes <0, x + y> OR <x + y - 4, 4>
- Use BFS on the state space to find the path from <0, 0> to one of the goal state <2, y> or <x, 2>

51

Two Water Jugs - BFS

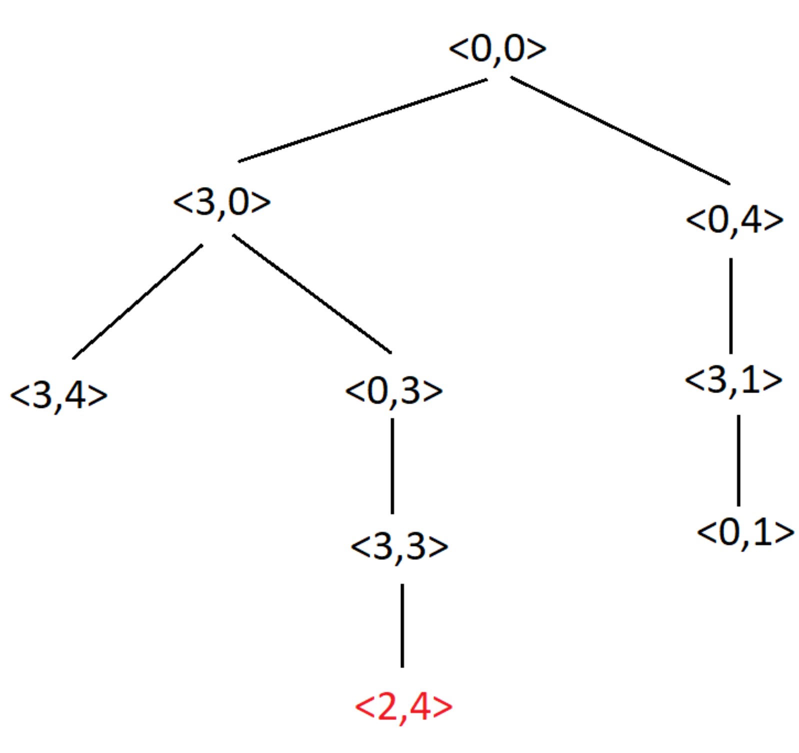
```
• enqueue the initial state

  while queue != empty
    currState = dequeue

    if currState == goal state
      announce & quit

    generate valid & not
    duplicated states from
    currState and enqueue

  end while
```



Wrapping things up

Learning objectives

- Understand the ***Transform & Conquer*** approach
  - **Simplify –** transform to a simpler or more convenient instance of the same problem
    - Pre-sorting
  - **Convert** – transform to a different representation of the same problem
    - Balanced search tree
    - Binary heap
  - **Reduce** – transform to an instance of a different problem for which an algorithm is already available
    - Calculate the LCM between two numbers
    - Use graph search