University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI 5 YEARS

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 7: next Friday @ 11:59 pm
  - Lab 6: Monday 10/31 @ 11:59 pm
  - Nothing due this week
- Midterm Exam
  - Wednesday 10/19 (MW Section) and Thursday 10/20 (TuTh Section)
    - in-person, closed-book
- Live QA Session on Piazza every Friday 4:30-5:30 pm
- Video for debugging using VS Code

Previous lecture

- LZW
    - implementation concerns
- Shannon's Entropy
- Comparing LZW vs Huffman
- Burrows-Wheeler Compression Algorithm
- ADT Priority Queue
    - array and BST implementations

This Lecture

- ADT Priority Queue (PQ)
  - Heap implementation
- Heap Sort
- Indexable PQ

Repetitive Highest Priority Problem

- Input:
  - a (large) dynamic set of data items
    - each item has a priority
    - e.g., highest priority is minimum item
    - e.g., highest priority is maximum item
  - a *stream* of zero or more of each of the following operations
    - Find a highest priority item in the set
    - Insert an item to the set
    - Remove a highest priority item from the set
- Examples
  - Selection sort
    - Repeatedly, remove a minimum item from the array and insert it in its correct position in the sorted array
  - Huffman trie construction
    - Each iteration: remove a minimum tree from the forest (**twice**) and insert a new tree

Let's create an ADT!

- The ADT Priority Queue (PQ)

- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()

Muddiest Points

- **Q: PQ runtimes for different data structures**

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)

Muddiest Points

- **Q:** **PQ runtimes for different data structures**

	findMin	removeMin	insert
Unsorted Array	O(n)	O(n)	O(1)
Sorted Array	O(1)	O(1)	O(n)
Red-Black BST	O(log n)	O(log n)	O(log n)

Is a BST overkill to implement ADT PQ?

- Balanced BST (e.g., RB-BST) provides *log n* runntime time for all operations
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
  - Can we take advantage of this to improve our runtime?
    - Yes!

The heap

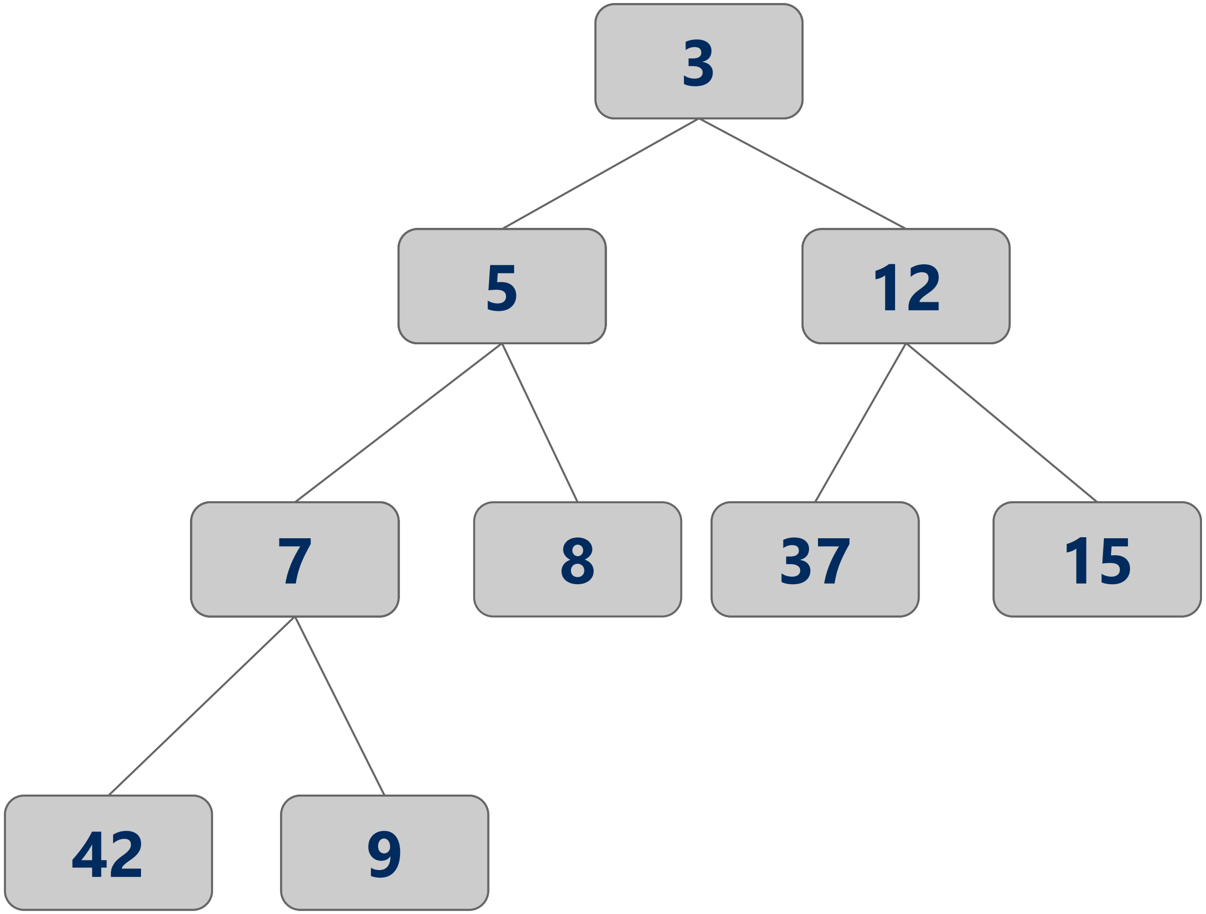
- A heap is **complete** binary tree such that for each node T in the tree:
    - T.item is of a higher priority than T.right_child.item
    - T.item is of a higher priority than T.left_child.item

- It does not matter how T.left_child.item relates to T.right_child.item
    - This is a relaxation of the approach needed by a BST

The *heap property*

Min Heap Example

- In a Min Heap, a highest priority item is a minimum item



Heap PQ runtimes

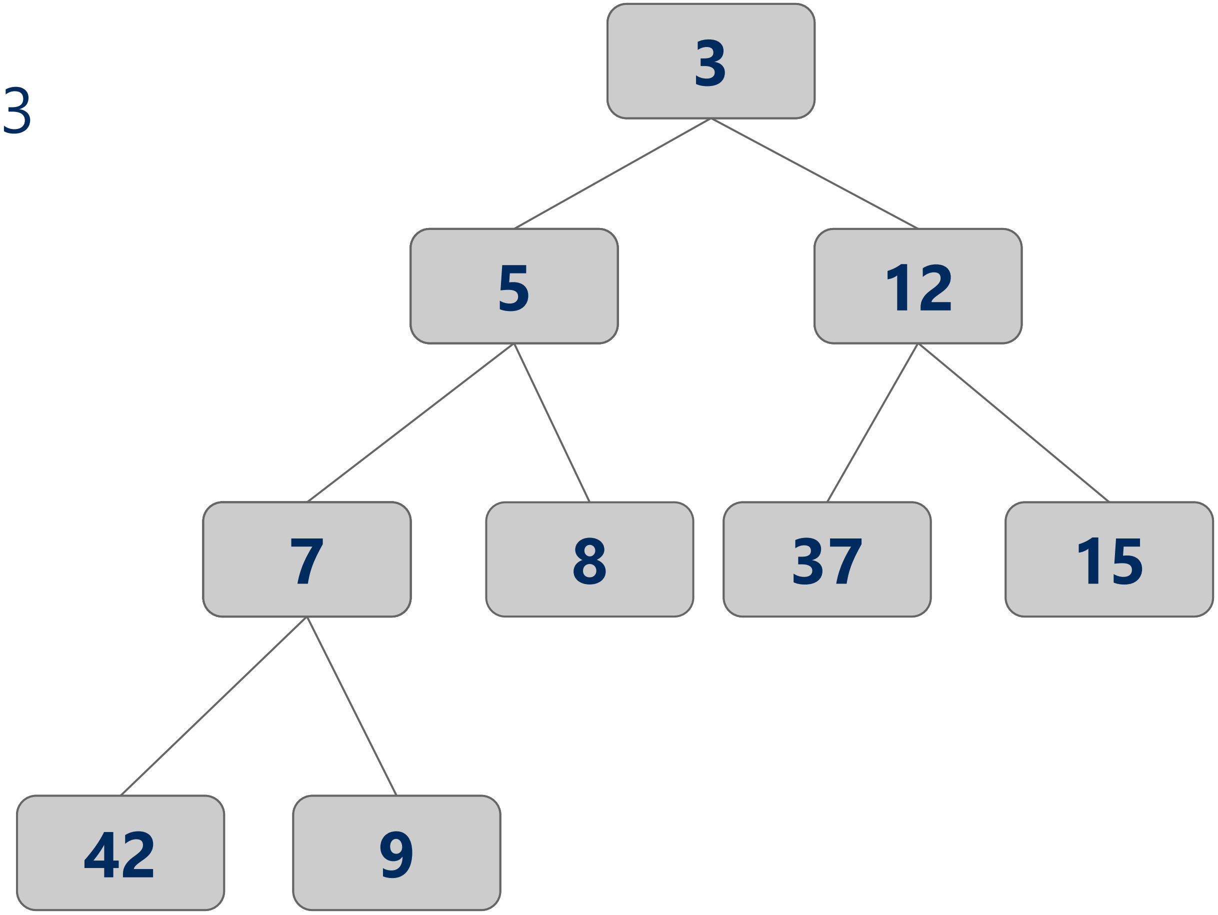
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

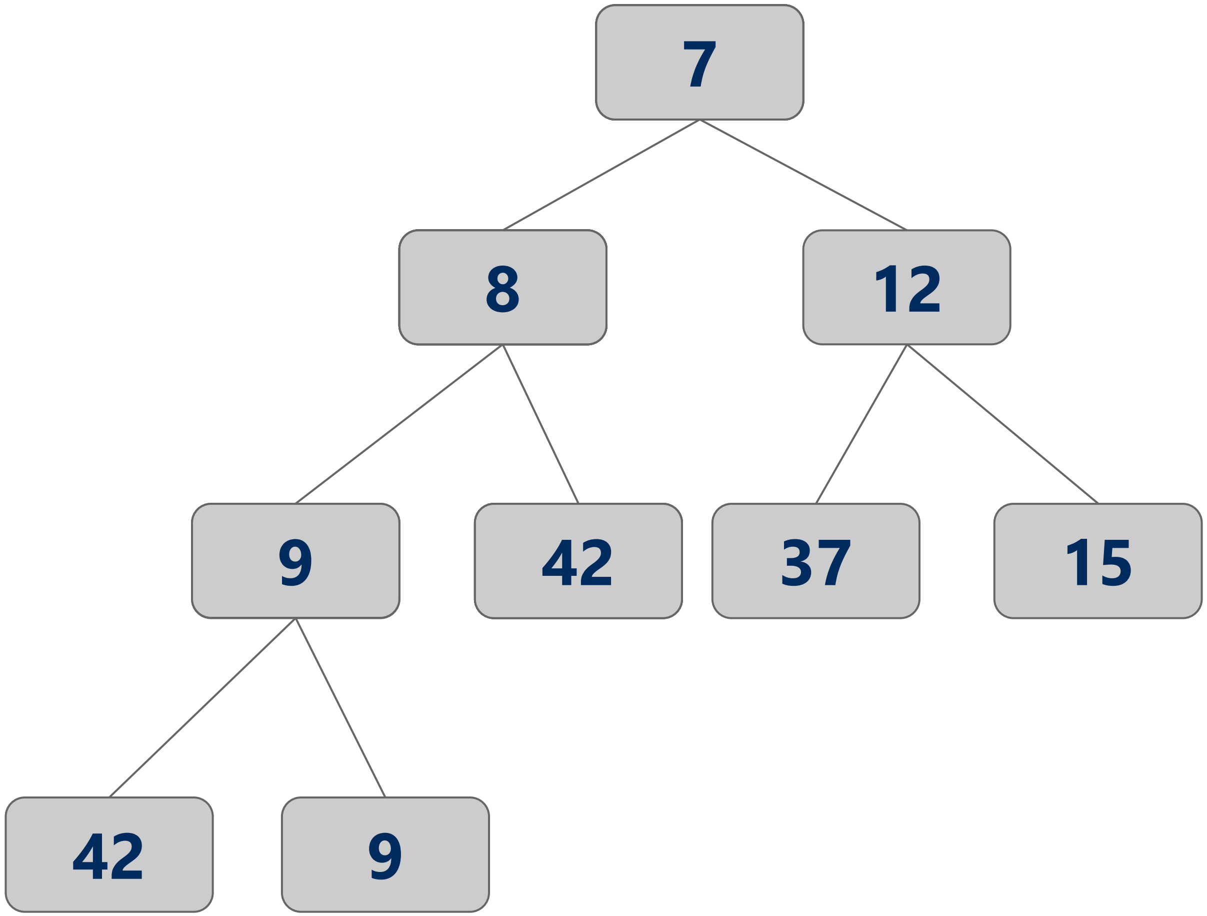


Heap remove

- Tricky to delete root…
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property…
      - So we push the root down the tree until it is supporting the heap property

Min heap removal

NO!



Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is lg n
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$

Heap implementation

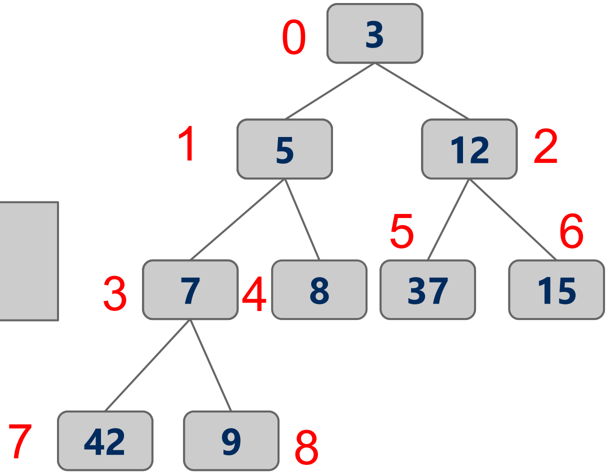
- Simply implement tree nodes like for BST
  - This requires overhead for dynamic node allocation
  - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
  - We can easily represent a complete binary tree using an array

Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = $\lfloor (i - 1) / 2 \rfloor$
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

For arrays indexed from 0

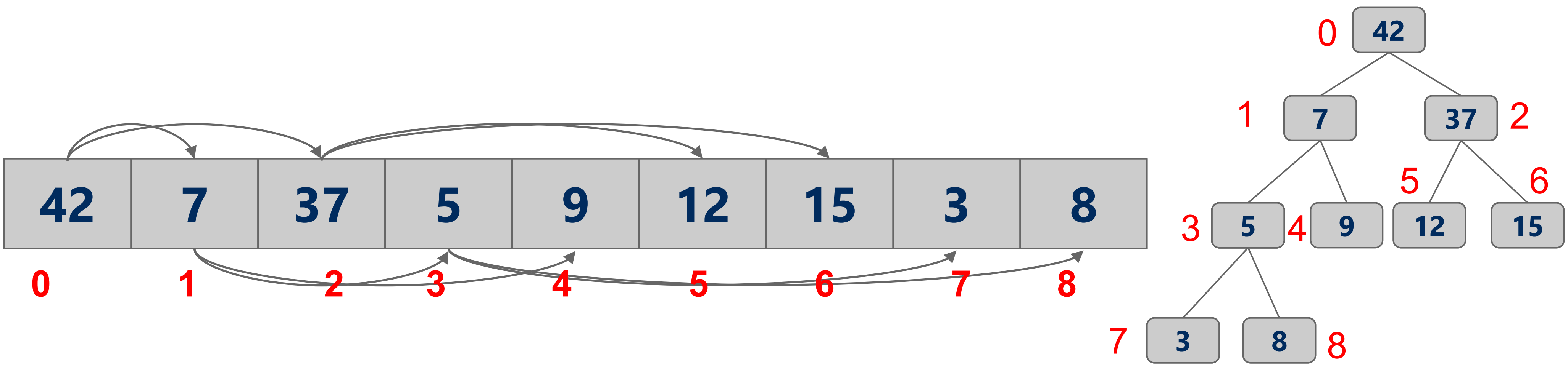
3	5	12	7	8	37	15	42	9
0	1	2	3	4	5	6	7	8



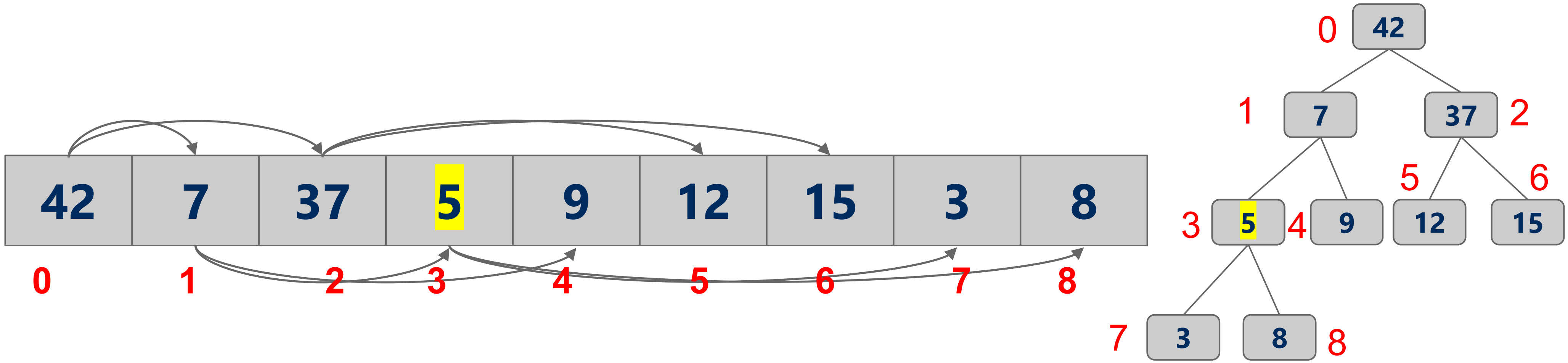
Can we turn any array into a heap?

- Yes!
- Any array can be thought of as a complete tree!
- We can change it into a heap using the following algorithm
- Scan through the array **right to** left starting from the rightmost non-leaf
    - the largest index *i* such that left_child(i) is a valid index (i.e., < n)
    - $2i+1 < n \rightarrow i < (n-1)/2$
    - push the node down the tree until it is supporting the heap property
- This is called the **Heapify** operation

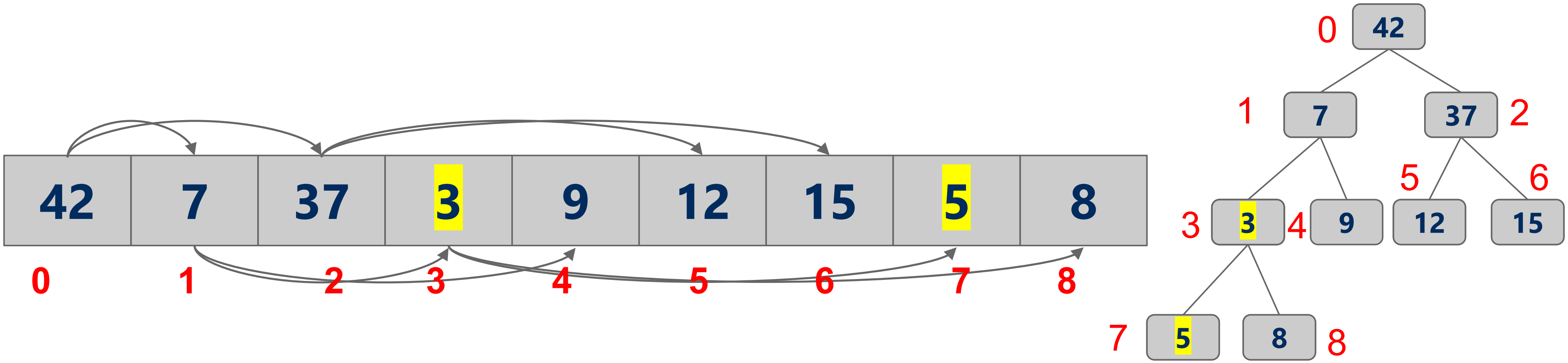
Heapify Example: Building a Min Heap



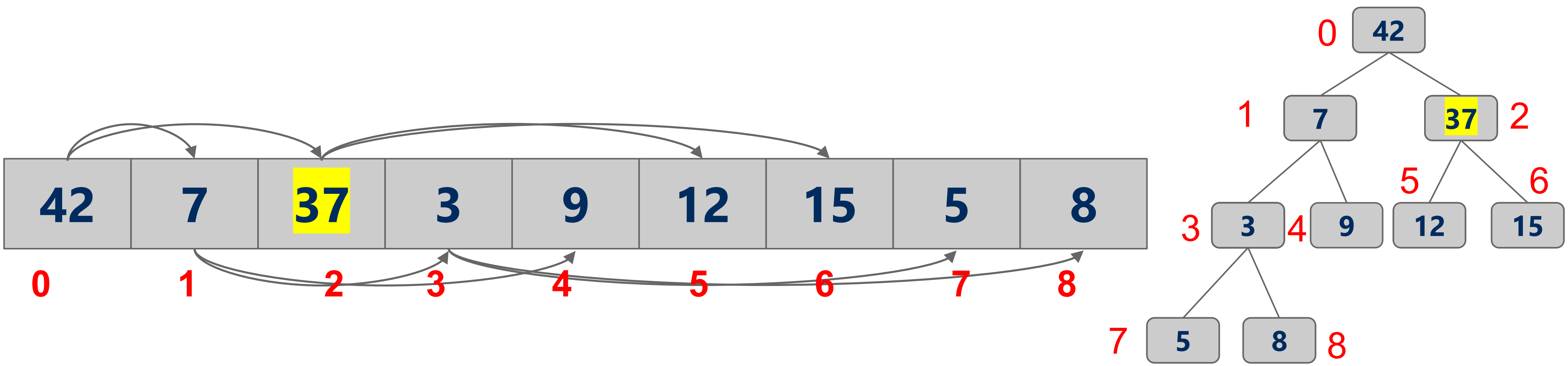
Heapify Example: Building a Min Heap



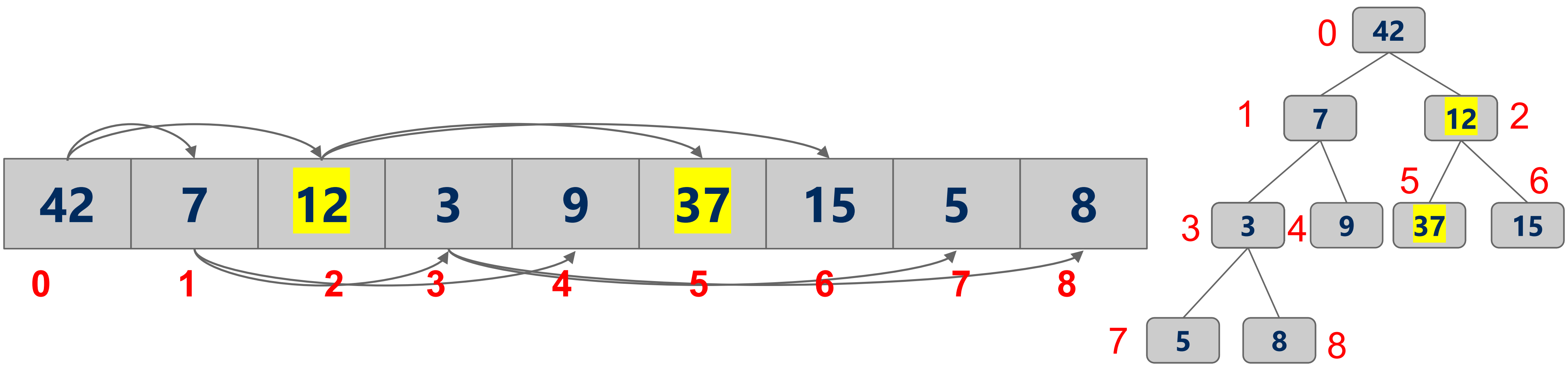
Heapify Example: Building a Min Heap



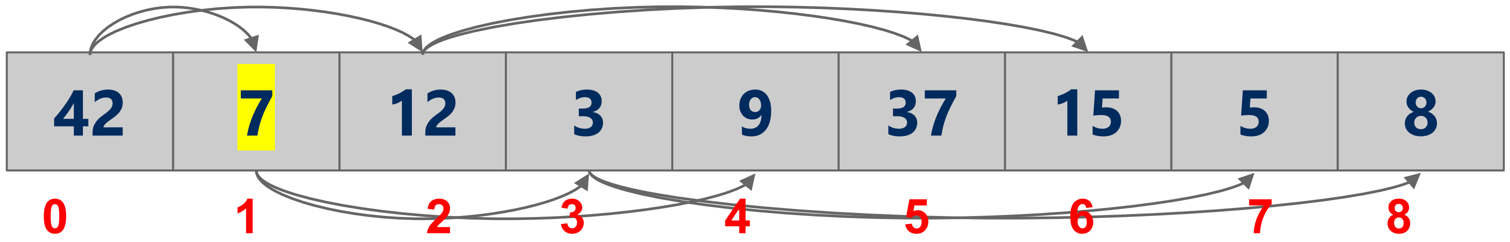
Heapify Example: Building a Min Heap

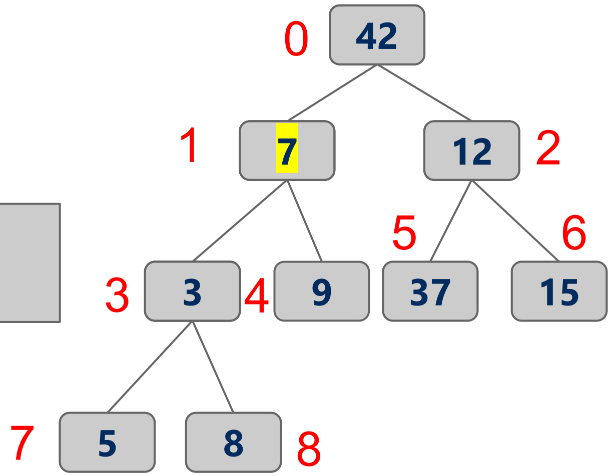


Heapify Example: Building a Min Heap

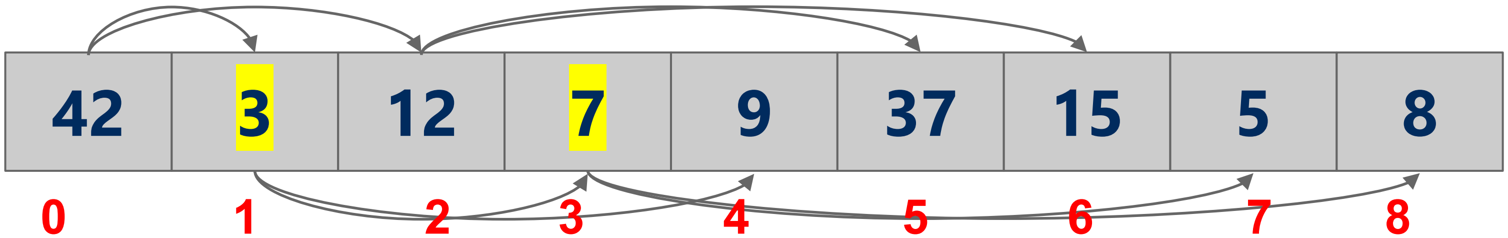


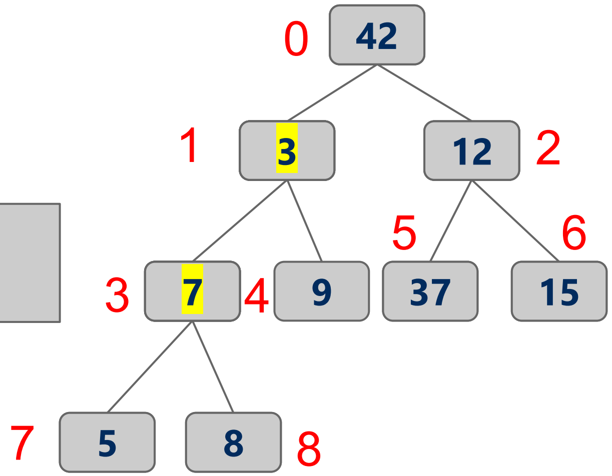
Heapify Example: Building a Min Heap



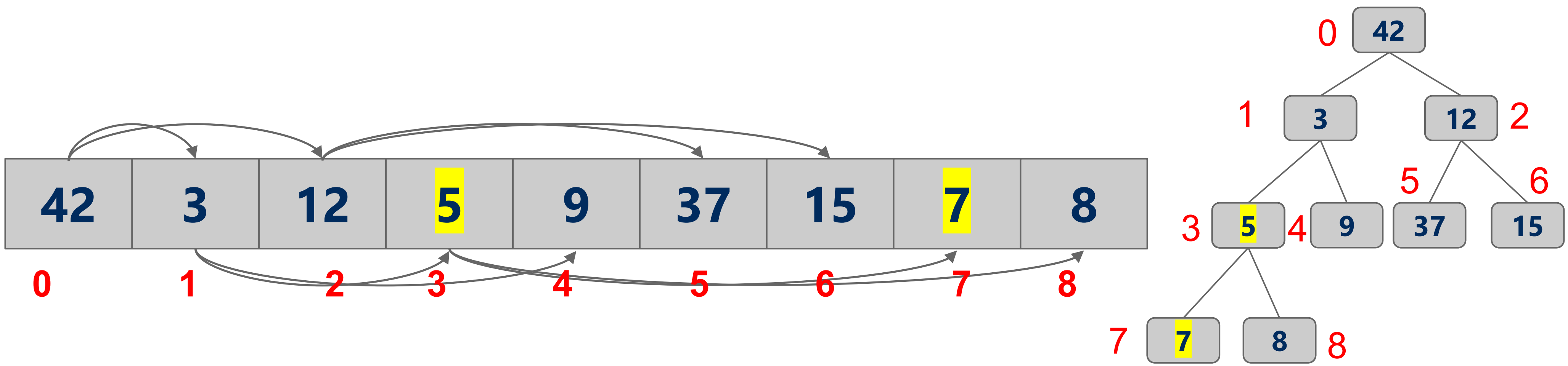


Heapify Example: Building a Min Heap

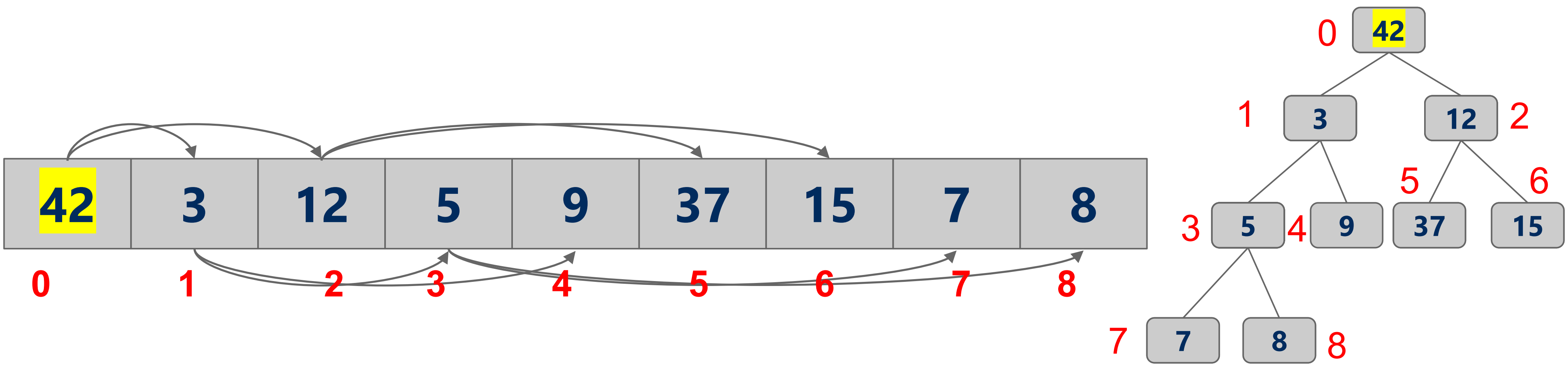




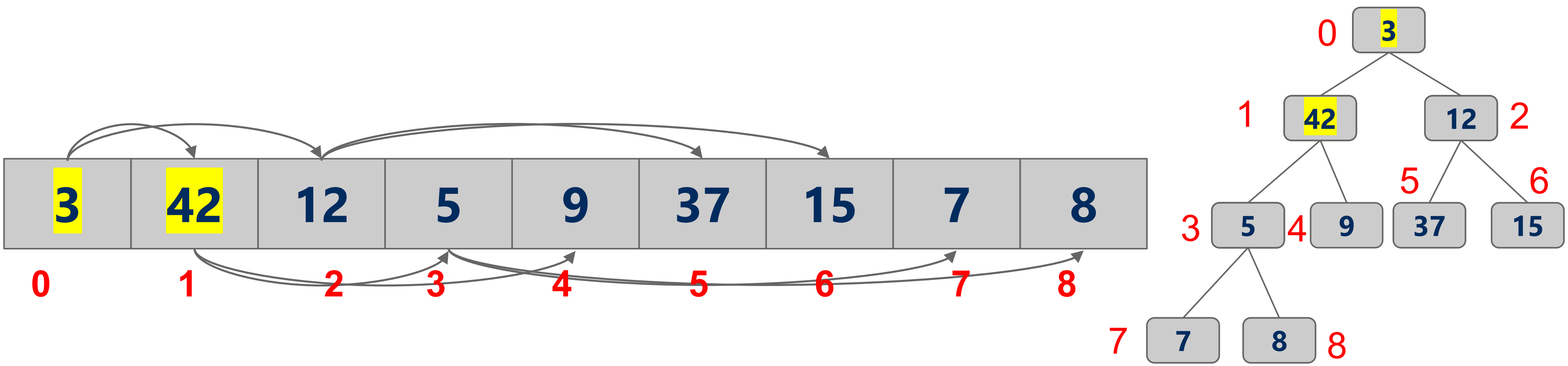
Heapify Example: Building a Min Heap



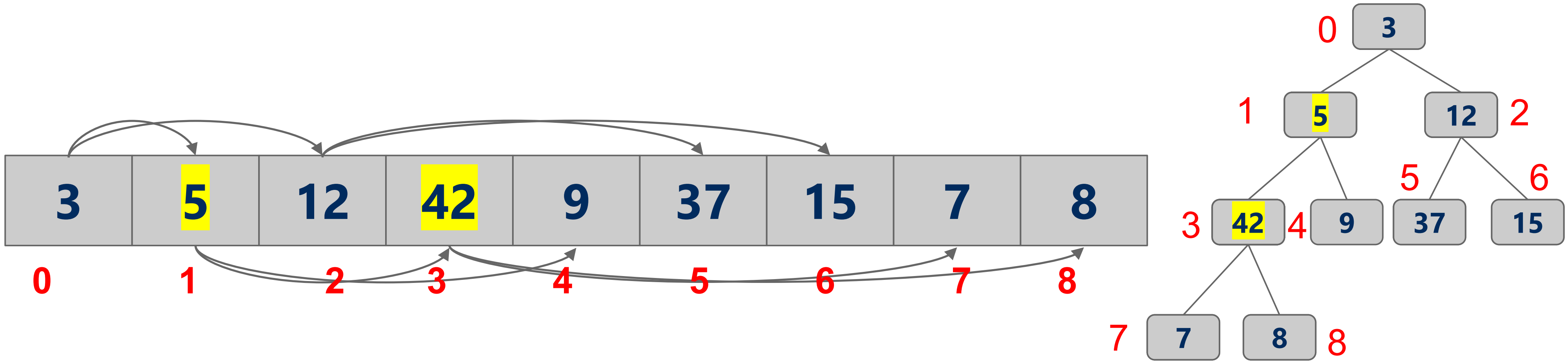
Heapify Example: Building a Min Heap



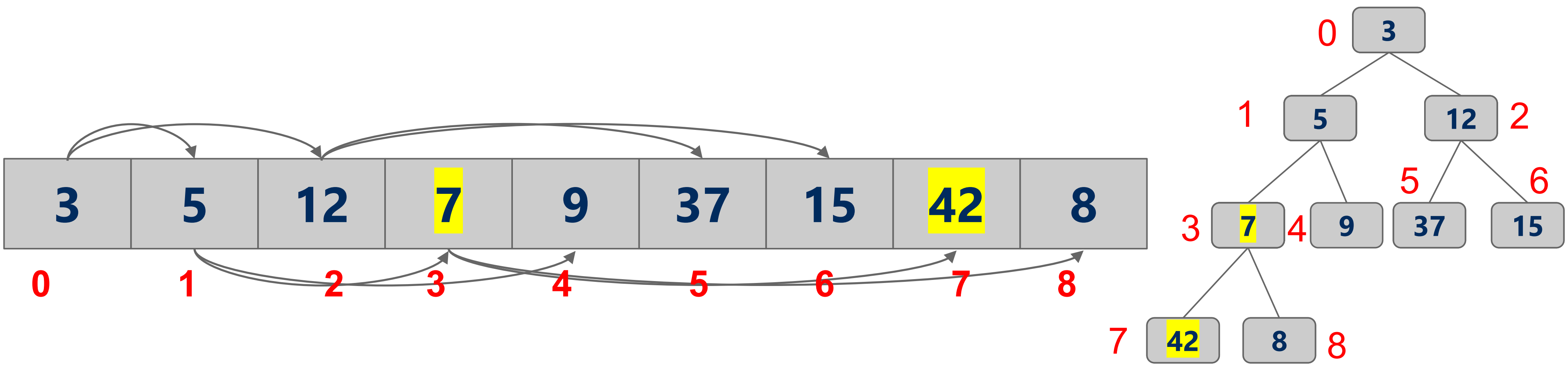
Heapify Example: Building a Min Heap



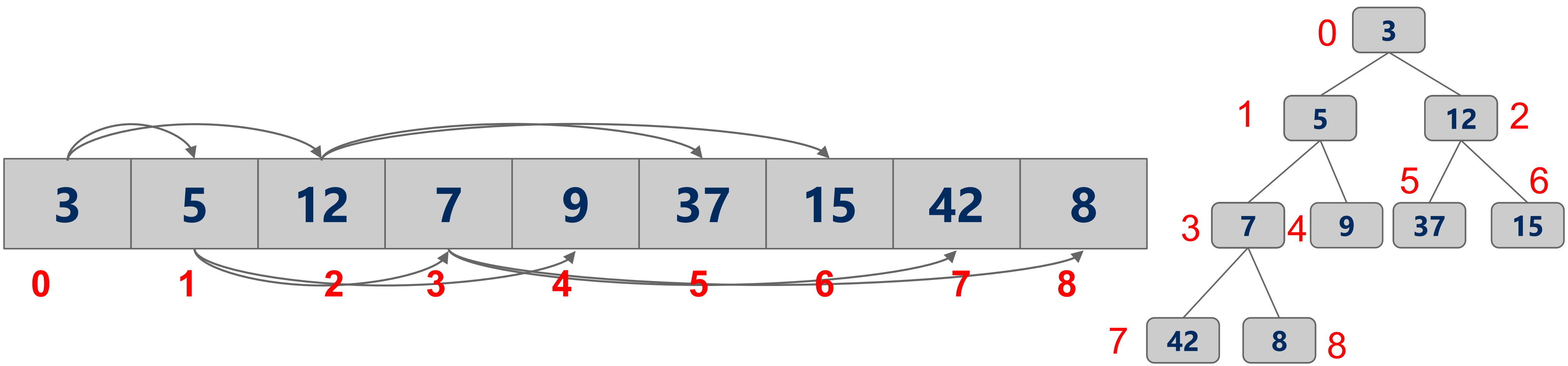
Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap

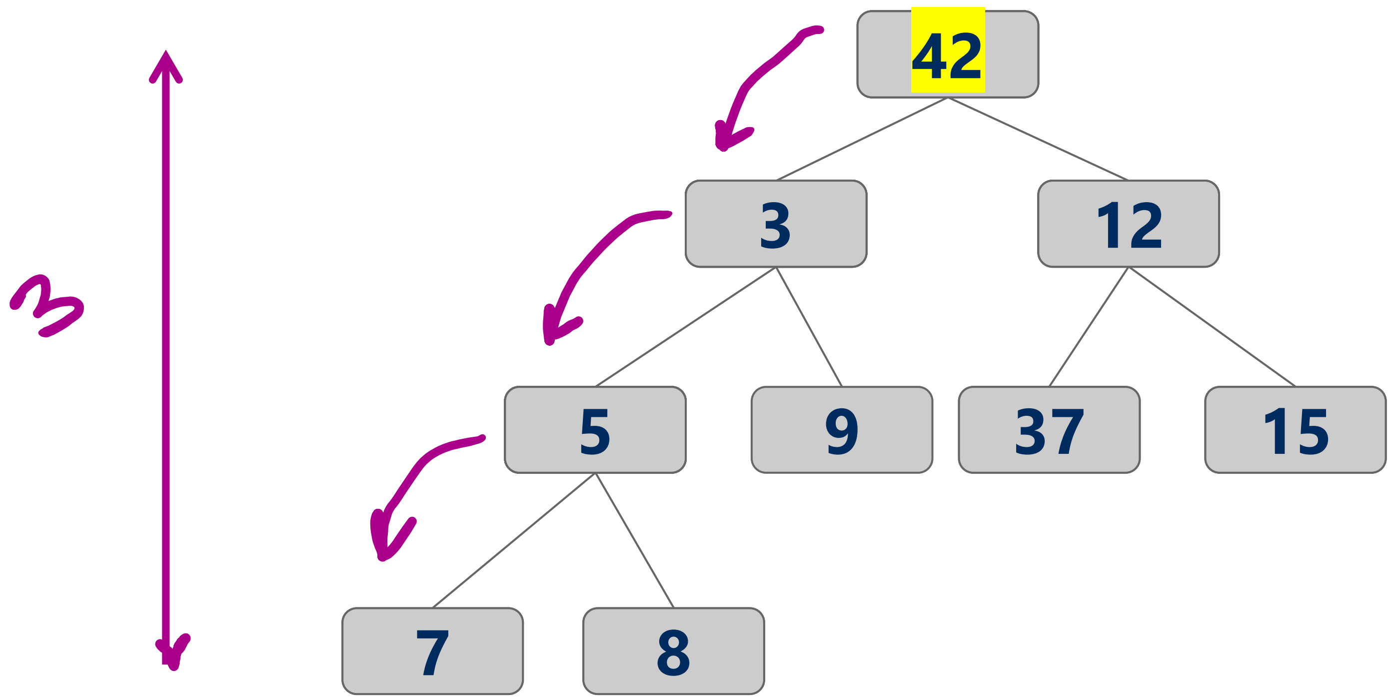


Heapify Running time

- Upper bound analysis:
    - We make about n/2 downheap operations
        - log n each
    - So, O(n log n)

Heapify Running time

- A tighter analysis
  - for each node that we start from, we make at most *height[node]* swaps



Heapify Running time: A tighter analysis

- $Runtime = \sum_{i=1}^{n} height[n]$
- $= \sum_{i=0}^{\log n} number\ of\ nodes\ with\ height\ i$
- Assume a full tree
  - A node with height *i* has 2^i nodes in its subtree including itself
  - Assume *k* nodes with height *i*:
  - they will have $k2^i$ nodes in their subtrees
  - $k2^i <= n \rightarrow k <= n/2^i$
- So, at most $n/2^i$ nodes exist with height *I*
- $\sum_{i=o}^{\log n} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \ \ldots$
- $= \ \theta(largest\ term) = \ \theta(n)$

Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

Storing Objects in PQ

- What if we want to **<u>update</u>** an Object in the heap?
  - What is the runtime to find an arbitrary item in a heap?
    - $\Theta(n)$
    - Hence, updating an item in the heap is $\Theta(n)$
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
   class CardPrice implements Comparable<CardPrice>{
         public String store;
         public double price;
         public CardPrice(String s, double p) { … }
         public int compareTo(CardPrice o) {
               if (price < o.price) { return -1; }
               else if (price > o.price) { return 1; }
               else { return 0; }
         }
}
```

Indirection example

- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

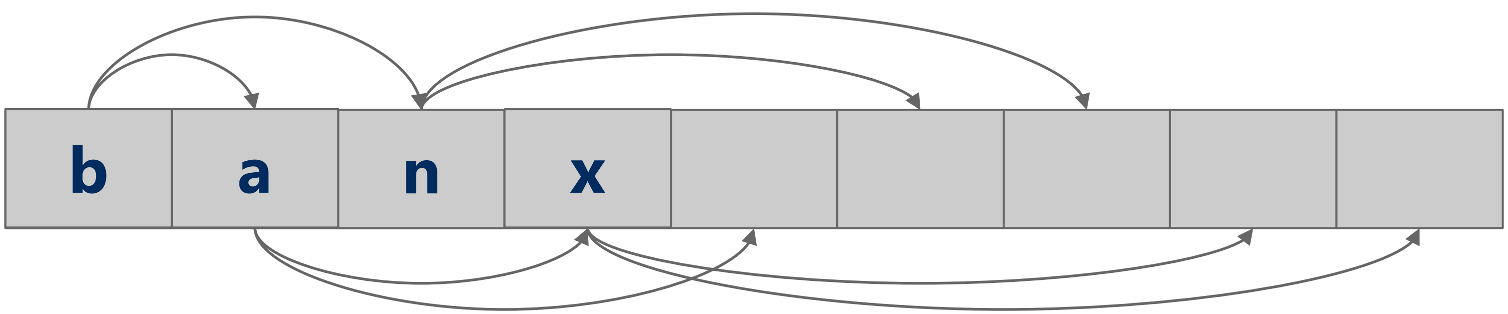
- Update price for NE: 340.00

- Update price for NCIX: 345.00

- Update price for BB: 200.00

Indirection

"NE":2
"AMZN":1
"NCIX":3
"BB":0



Indexable PQ Discussion

- How are our runtimes affected?
- space utilization?
- how should we implement the indirection?
- what are the tradeoffs?