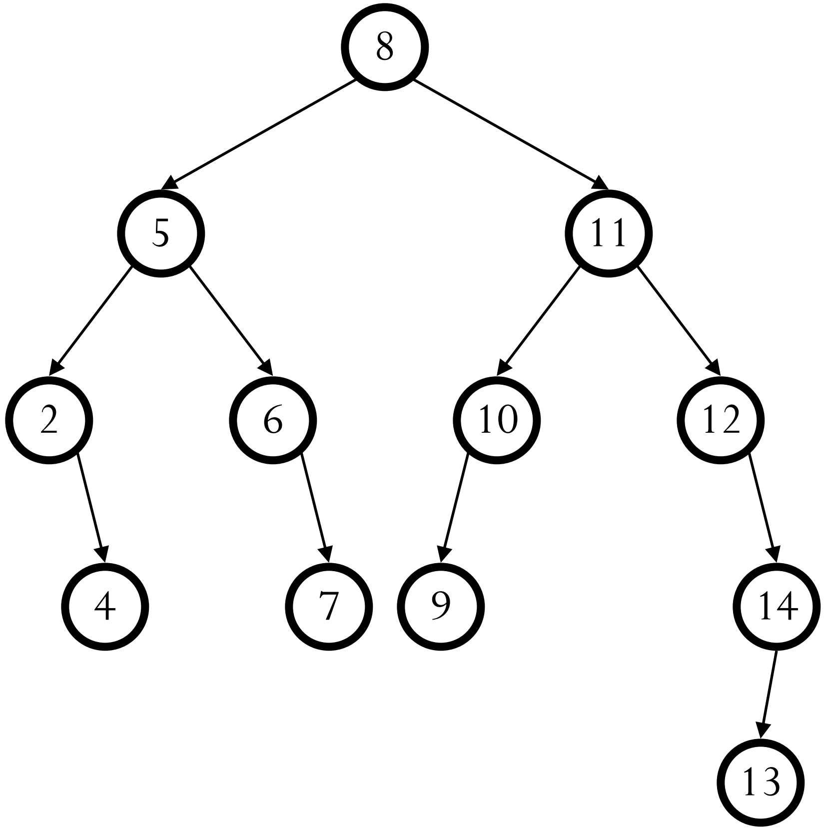
Binary Heap and Priority Queue

Dr. Animesh Chaturvedi
Assistant Professor: IIIT Dharwad
Young Researcher: Heidelberg Laureate Forum
Postdoc: King's College London & The Alan Turing Institute
PhD: IIT Indore MTech: IIITDM Jabalpur

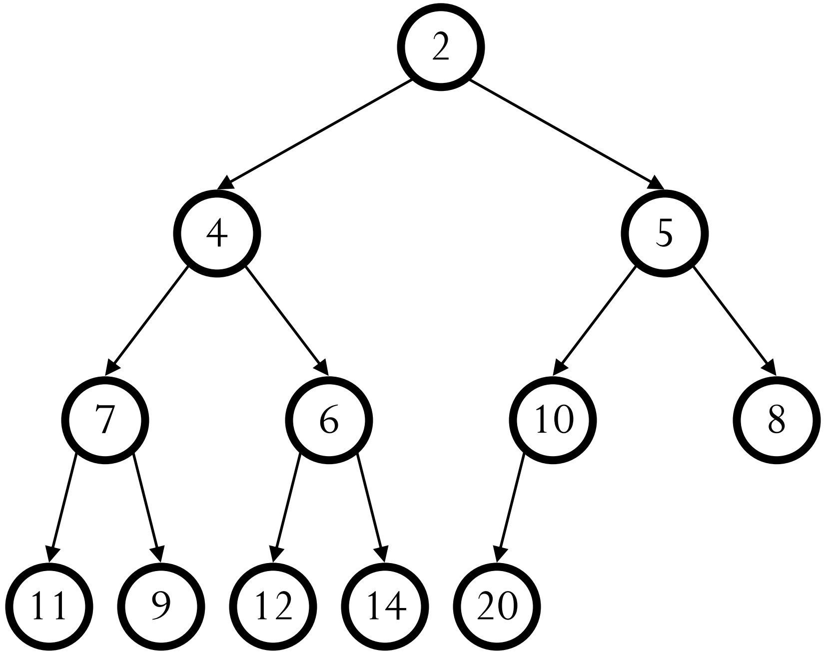
Binary Tree Data Structures

- **Unsorted list:**
  - ***insert:***
  - ***deleteMin:***
- Sorted list:
  - *insert:*
  - *deleteMin:*



Binary Heap - Data Structure

- Heap-order property
  - parent's key is less than children's keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always O(log n);
  - next open location always known



Binary Heap - Data Structure

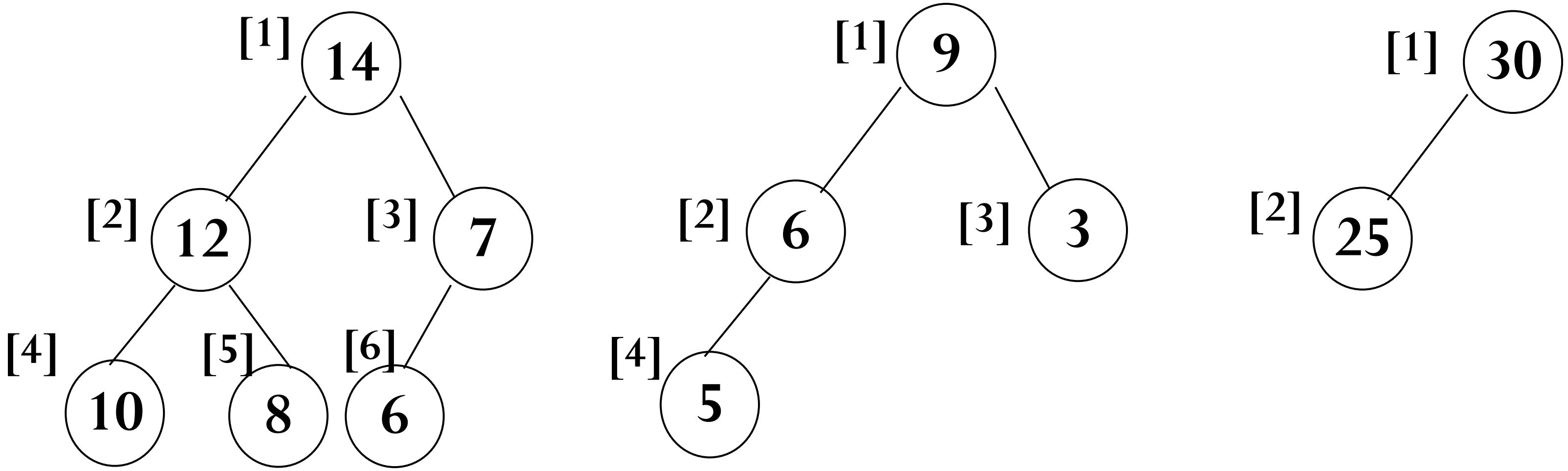
- A ***heap*** orders its node, but in a way different from a binary search tree
- A complete tree is a *heap* if
  - The value in the root is the smallest of the tree
  - Every subtree is also a heap
- Equivalently, a complete tree is a heap if
  - Node value < child value, for each child of the node
- **Note:** This use of the word "heap" is entirely different from the heap that is the allocation area in Java

Binary Heap

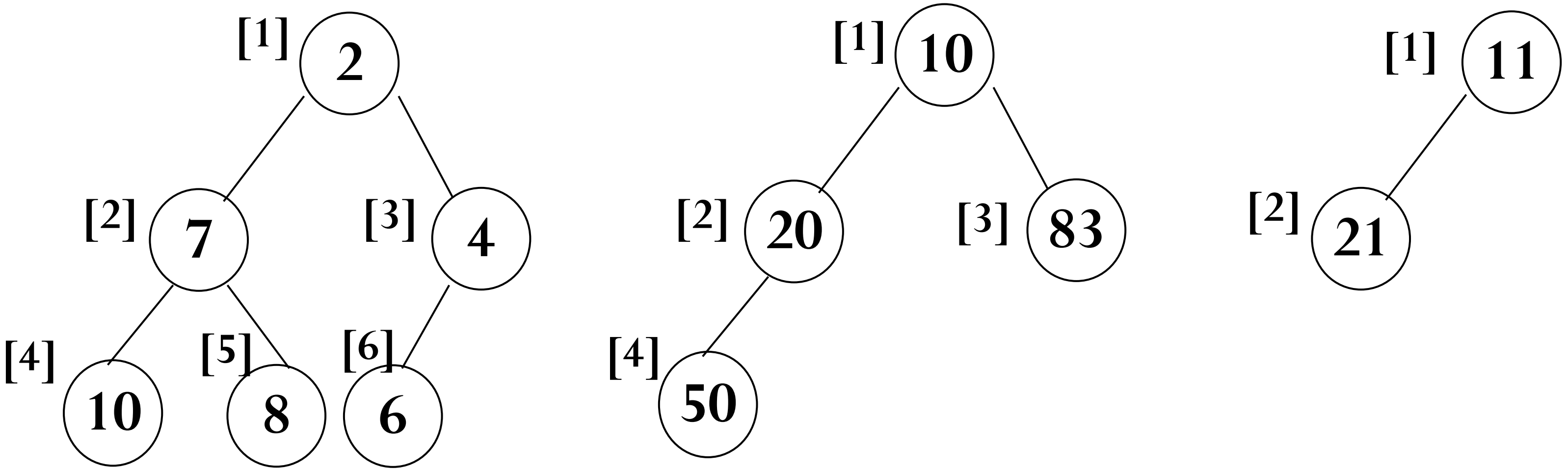
- *Max tree*: the key value in each node is no smaller than the key values in its children.
  - *Max heap* is a complete binary tree that is also a max tree.
- *Min tree*: the key value in each node is no larger than the key values in its children.
  - *Min heap* is a complete binary tree that is also a min tree.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap;
  - deletion of the largest element from the heap

Binary Heap

- The root of max heap contains the largest element.

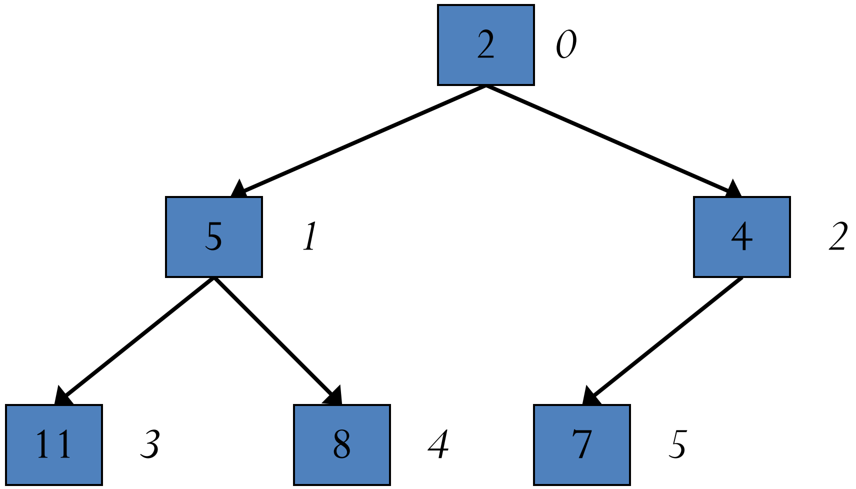


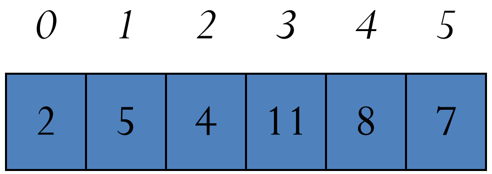
- The root of min heap contains the smallest element.



Implementing a Heap

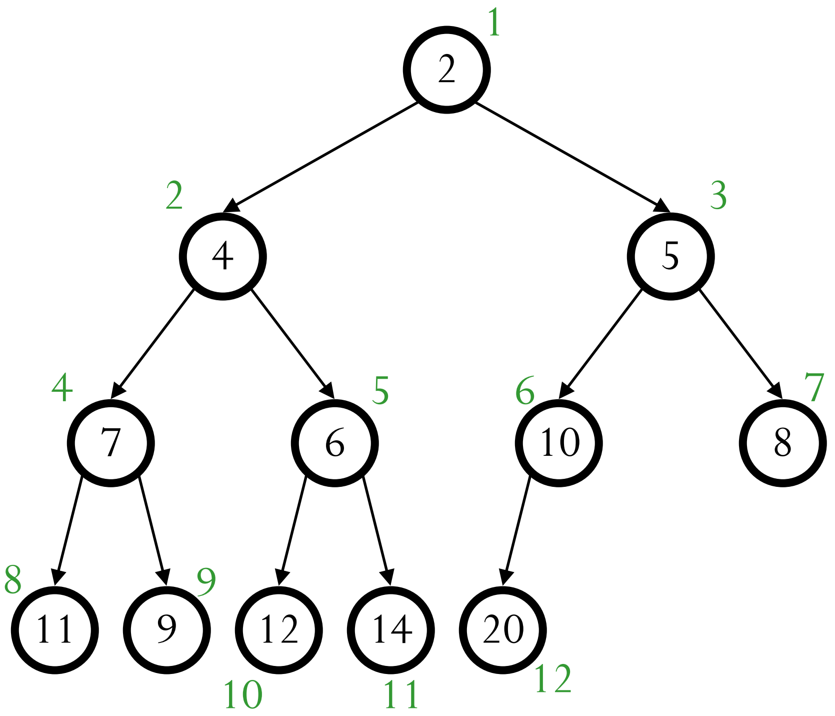
- Recall: a heap is a ***complete binary tree***
  - (plus the heap ordering property)
- A complete binary tree fits nicely in an ***array:***
  - The root is at index 0
  - Children of node at index i are at indices 2i+1, 2i+2





Storage (Min Heap)

- Calculations:
  - child:
  - parent:
  - root:
  - next free:



0	1	2	3	4	5	6	7	8	9	10	11	12	
12	2	4	5	7	6	10	8	11	9	12	14	20	

Removing an Item from a Heap

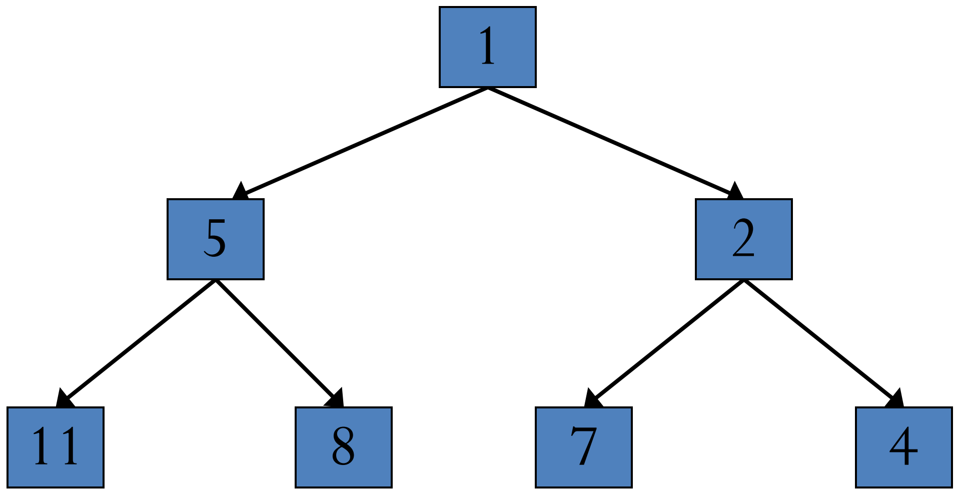
- Removing an item is always from the *top:*
  - Remove the *root* (minimum element):
    - Leaves a “hole”:
  - Fill the “hole” with the last item (lower right-hand) L
    - *Preserve completeness*
  - Swap L with smallest child, as necessary
    - *Restore “heap-ness”*

Example Removing From a Heap

Remove: returns 1

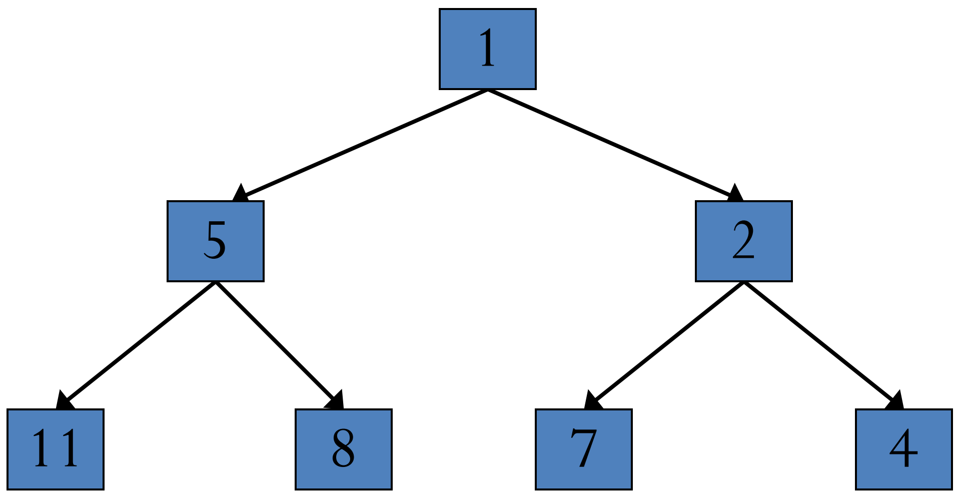
Move 4 to root

Swap down



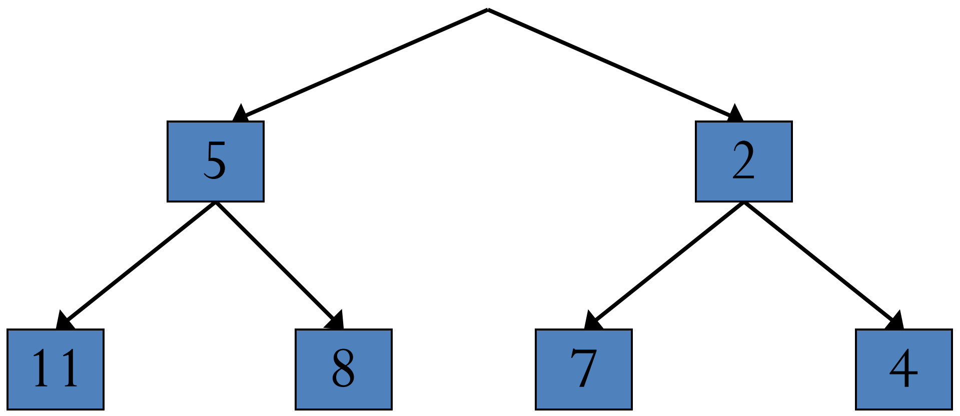
Example Removing From a Heap

Remove: returns 1



Example Removing From a Heap

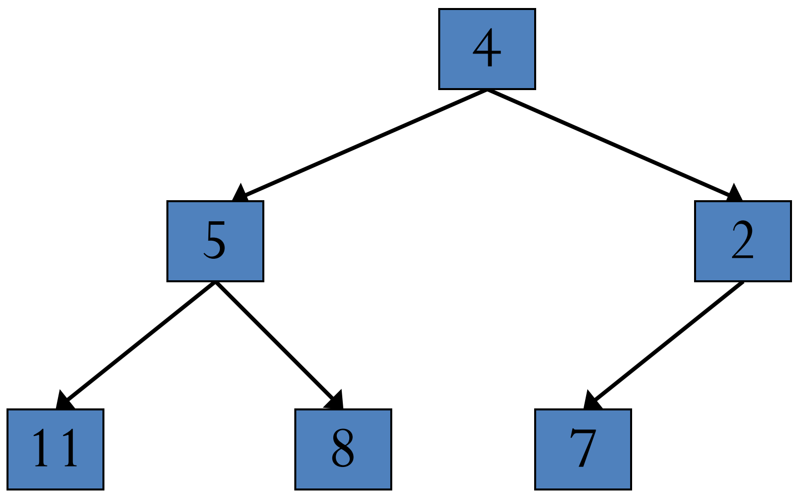
Remove: returns 1



Example Removing From a Heap

Remove: returns 1

Move 4 to root

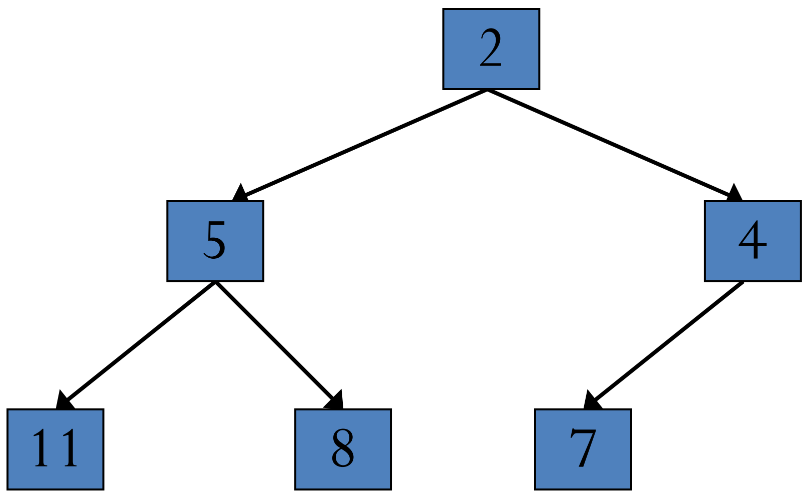


Example Removing From a Heap

Remove: returns 1

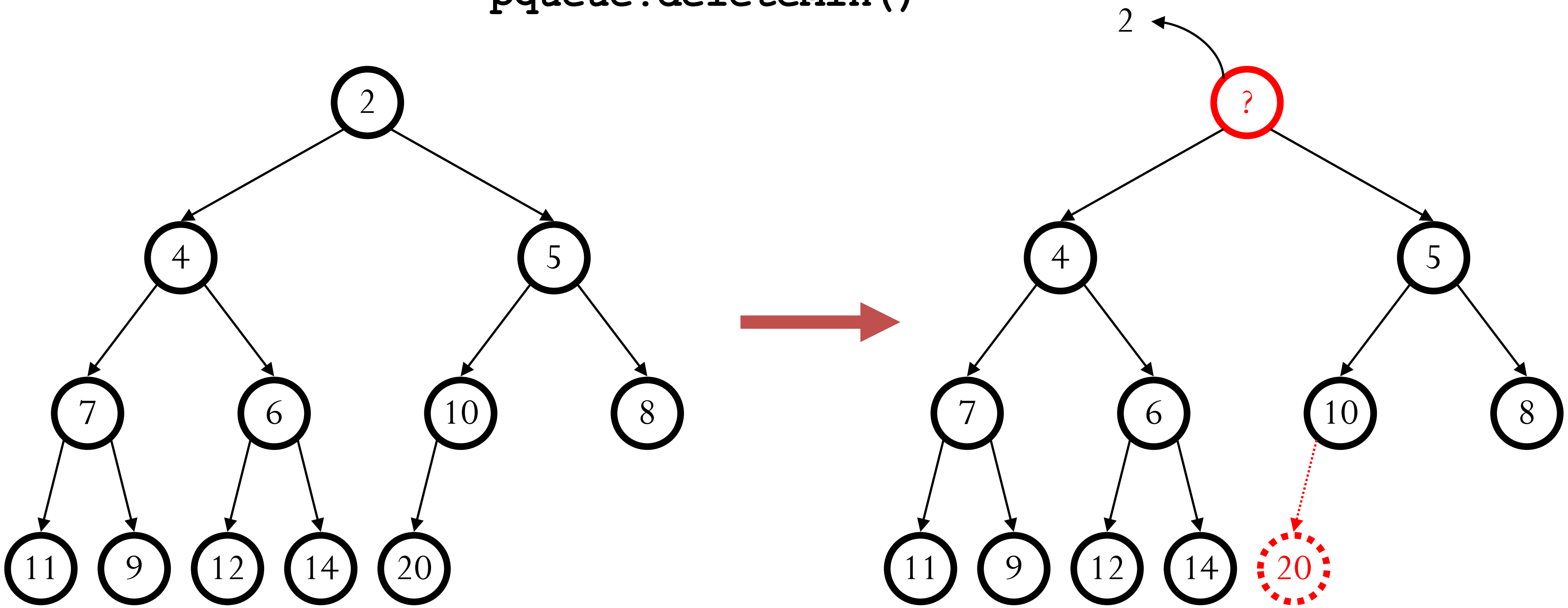
Move 4 to root

Swap down

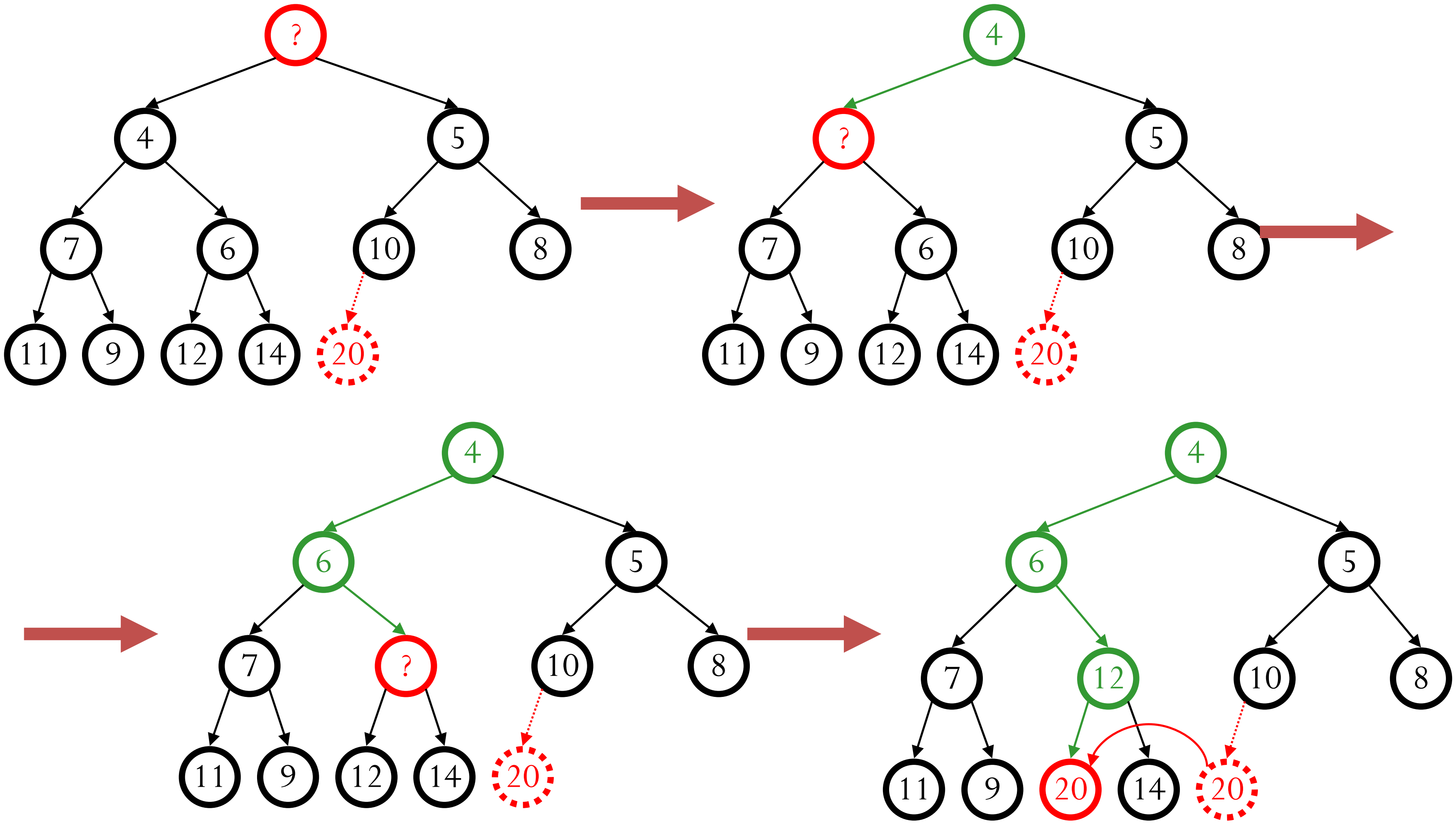


DeleteMin

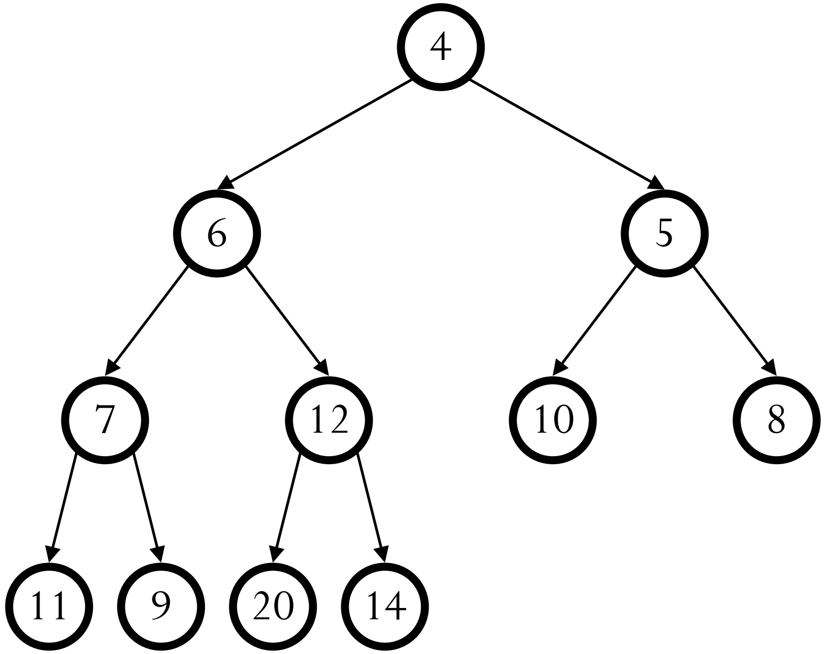
`pqueue.deleteMin()`



Percolate Down



Finally…

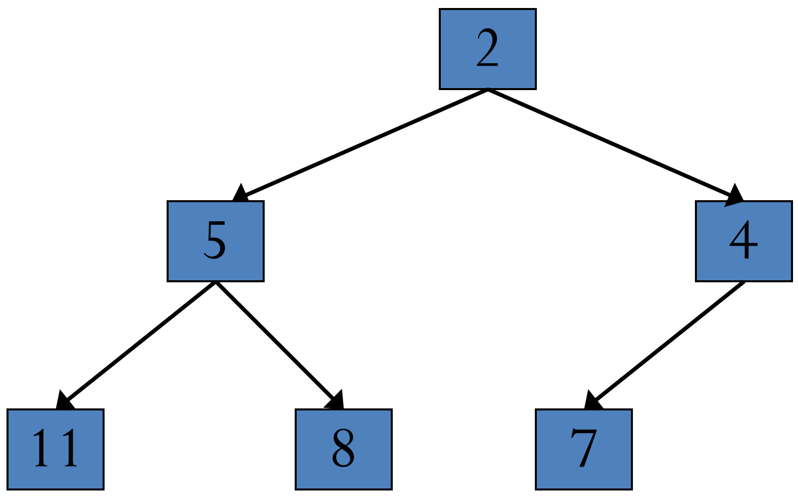


Inserting an Item into a Heap

1. Insert the item in the next position across the bottom of the complete tree: *preserve completeness*
2. *Restore "heap-ness":*
    1. **while** new item not root and < parent
    2. swap new item with parent

Example Inserting into a Heap

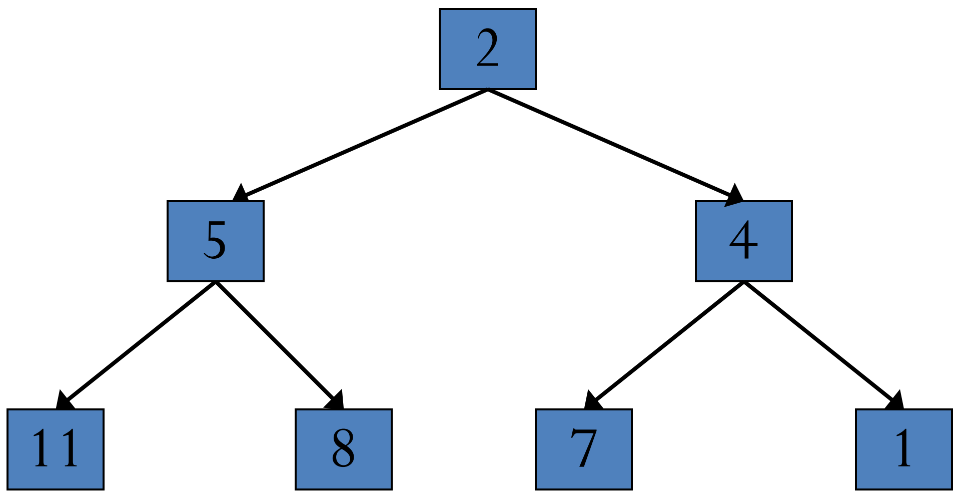
Insert 1



Example Inserting into a Heap

Insert 1

Add as leaf

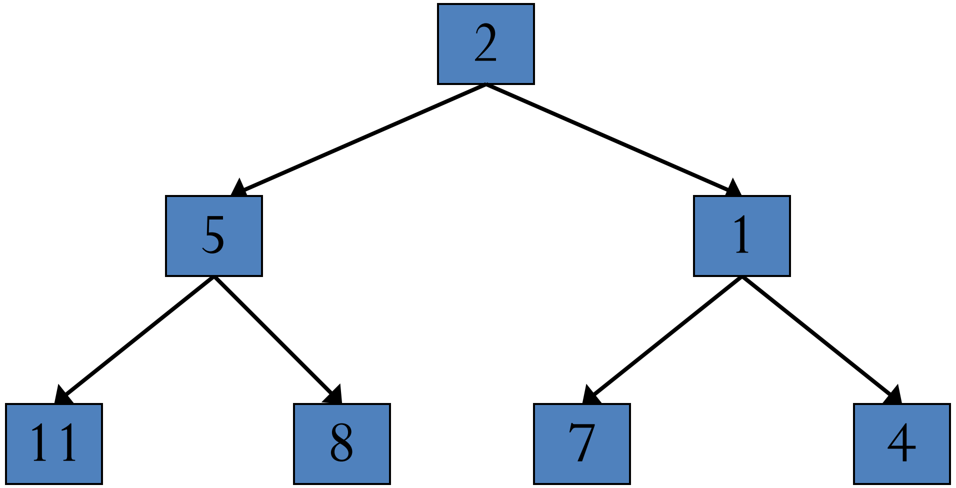


Example Inserting into a Heap

Insert 1

Add as leaf

Swap up



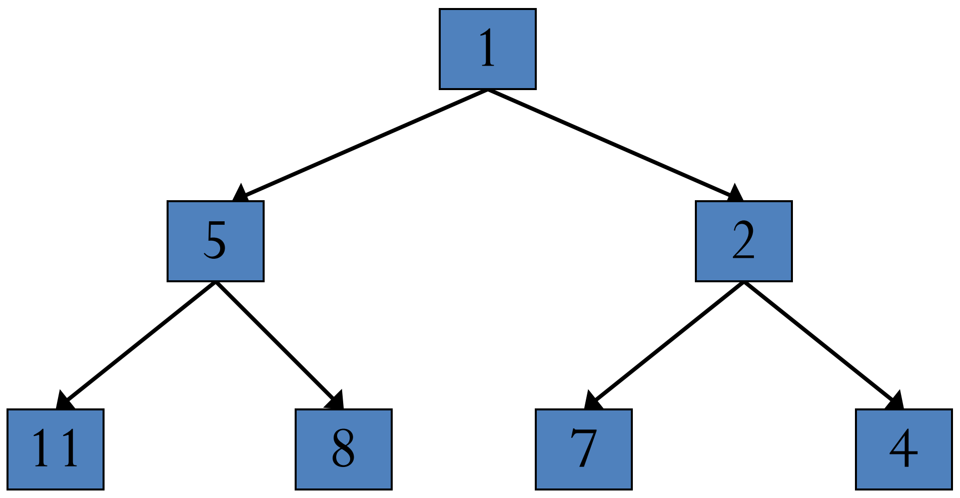
Example Inserting into a Heap

Insert 1

Add as leaf

Swap up

Swap up



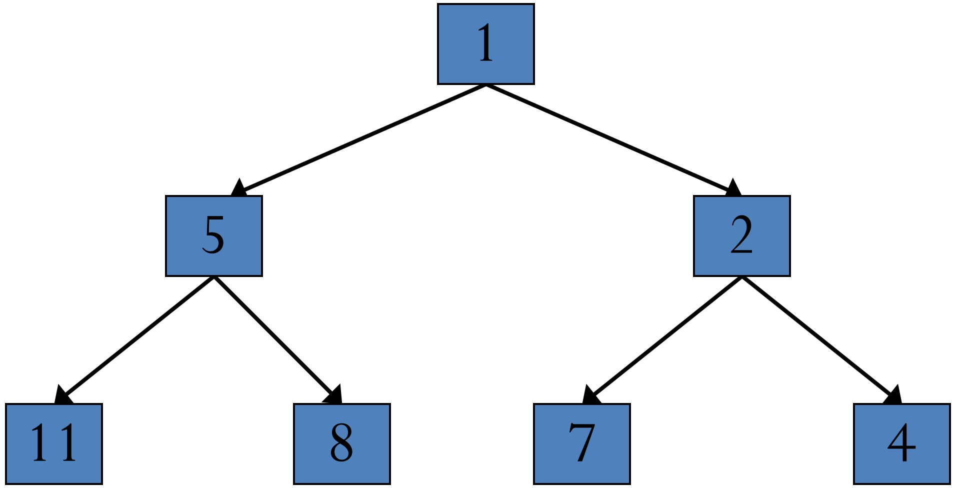
Example Inserting into a Heap

Insert 1

Add as leaf

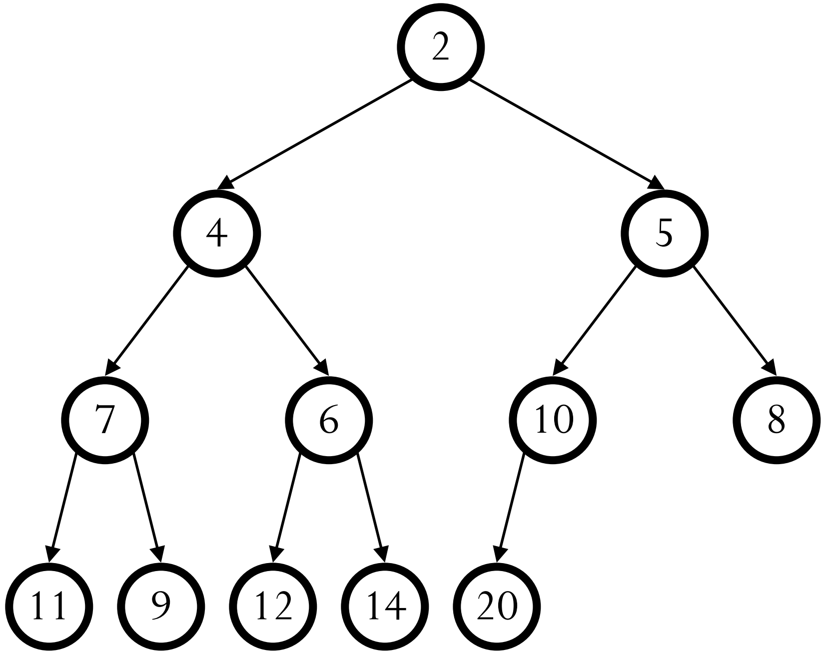
Swap up

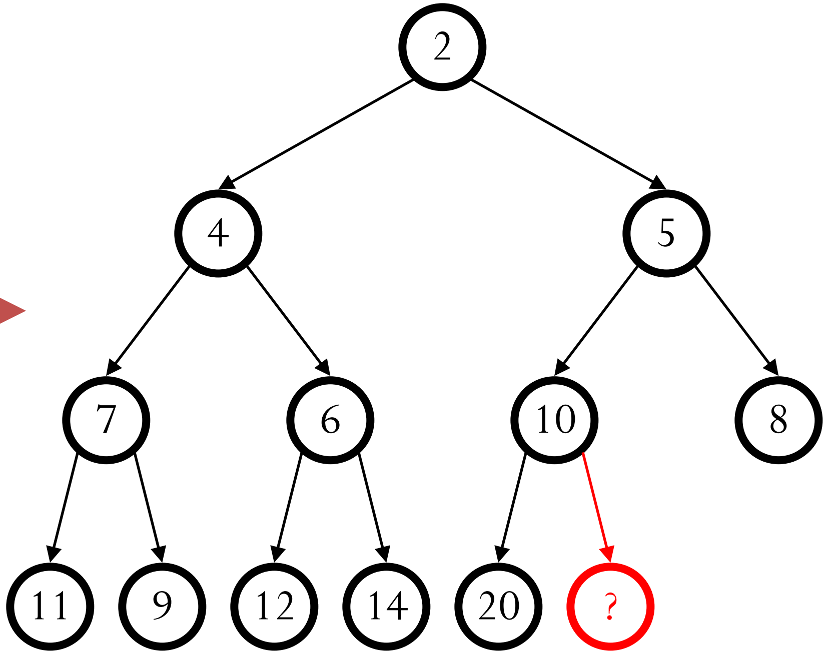
Swap up



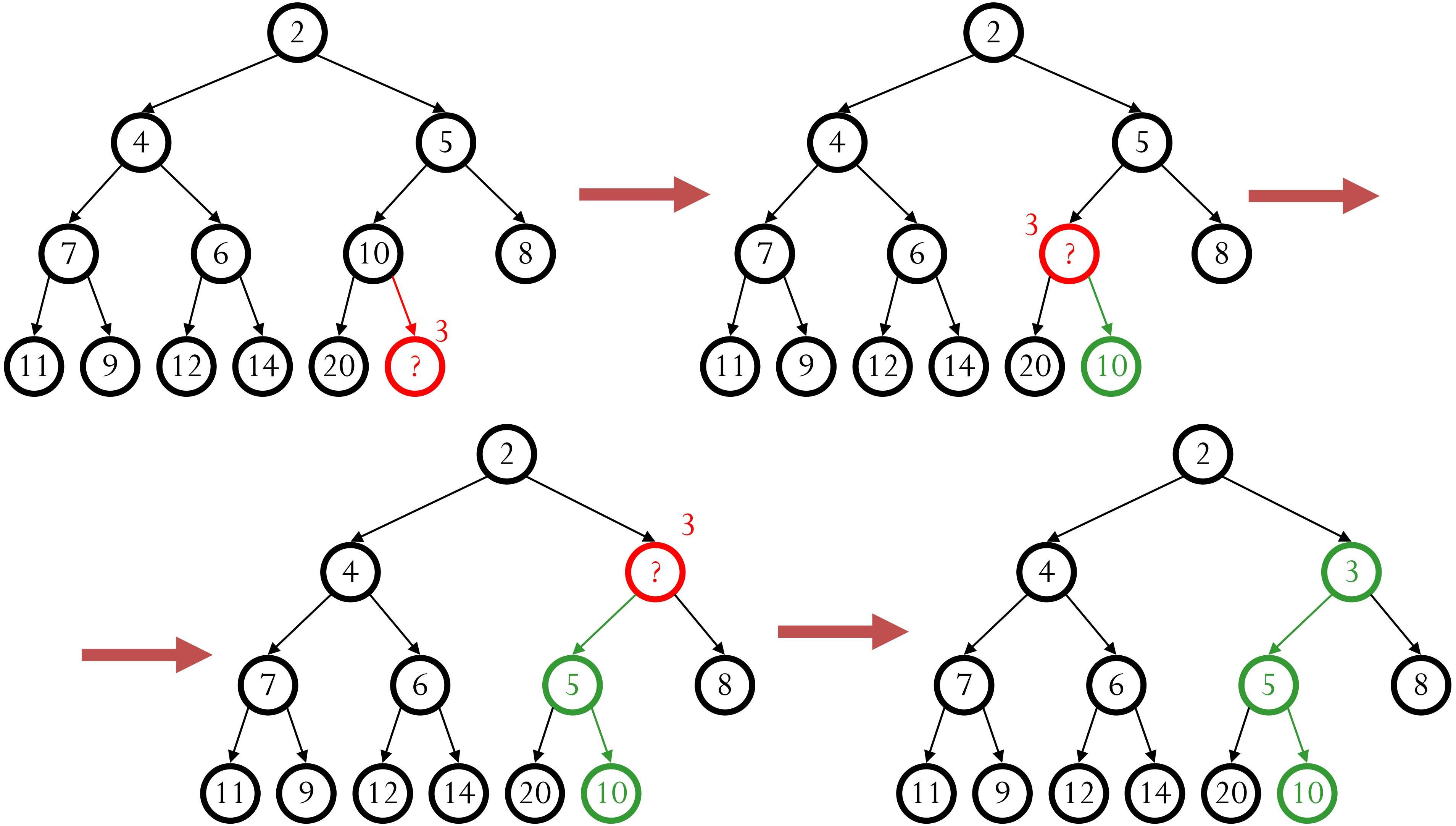
Insert

`pqueue.insert(3)`





Percolate Up

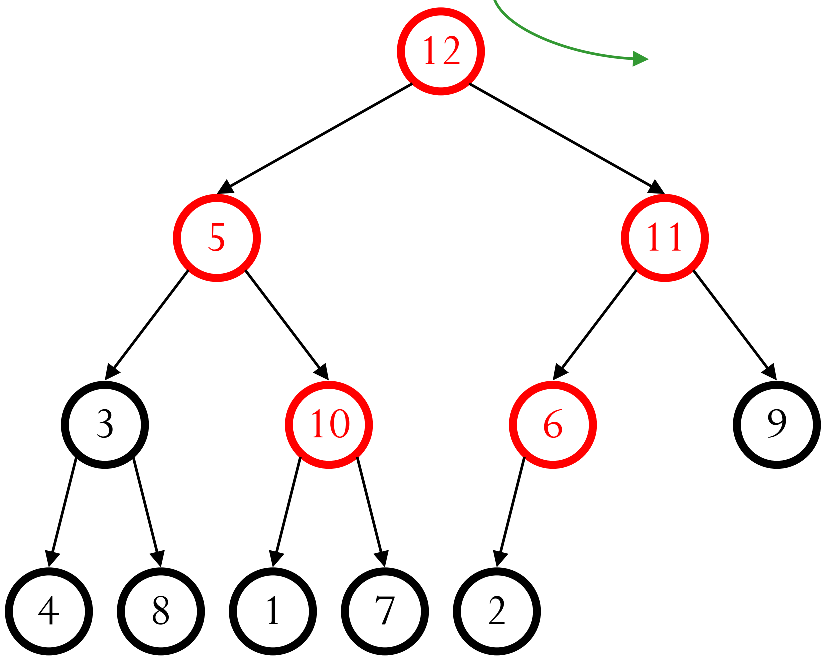


Build Heap

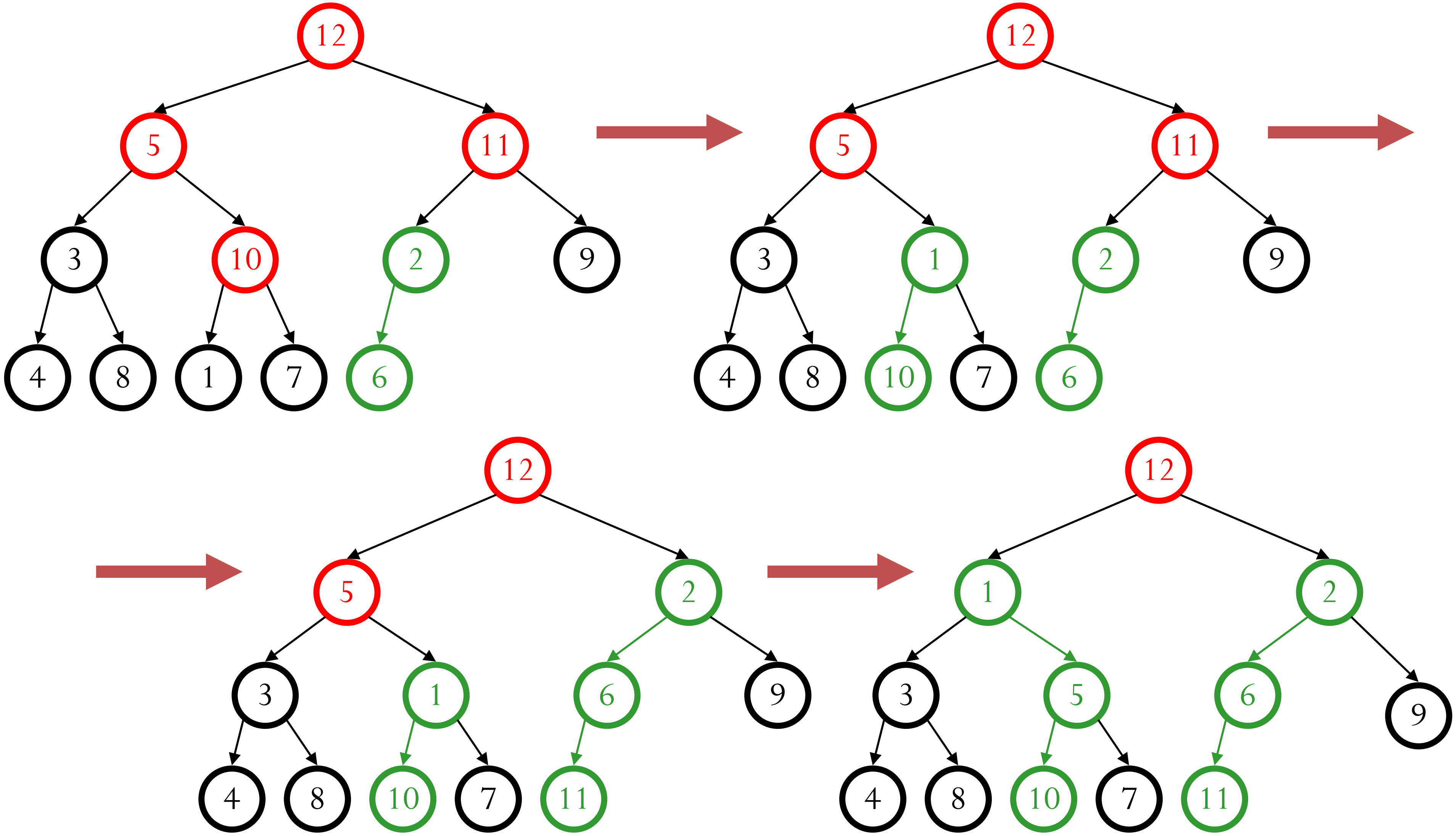
Floyd's Method.

12	5	11	3	10	6	9	4	8	1	7	2

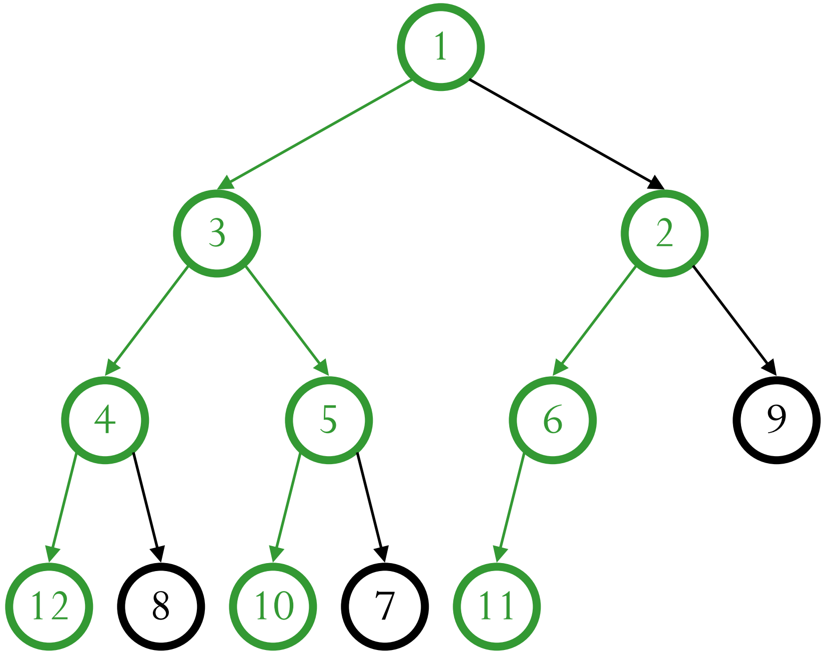
pretend it's a heap and fix the heap-order property!



Build Heap


12
5
11
3
10
2
9
4
8
1
7
6
12
5
11
3
1
2
9
4
8
10
7
6
12
5
2
3
1
6
9
4
8
10
7
11
12
1
2
3
5
6
9
4
8
10
7
11

Build Heap

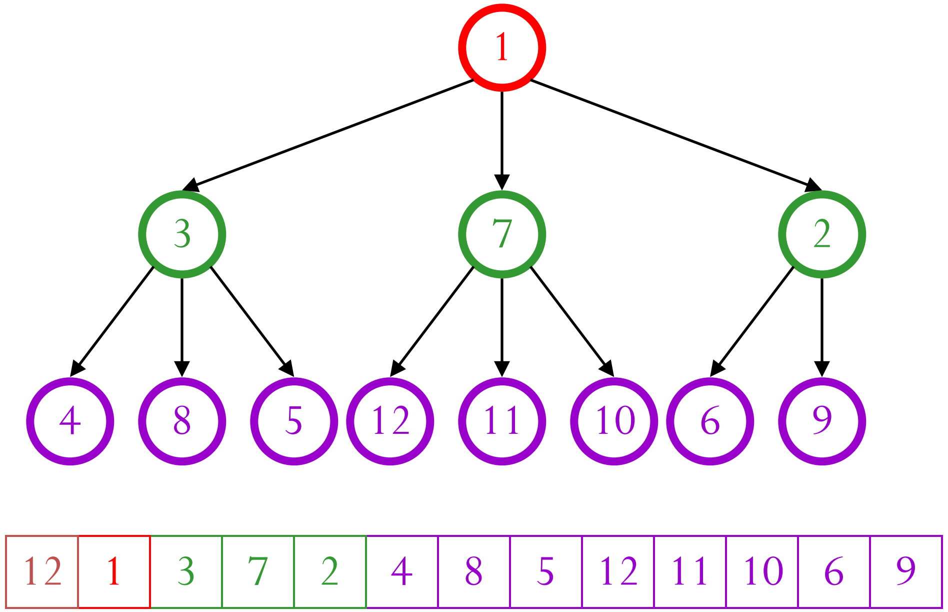


Thinking about Heaps

- Observations
  - finding a child/parent index is a multiply/divide by two
  - operations jump widely through the heap
  - each operation looks at only two new nodes
  - inserts are at least as common as deleteMins
- Realities
  - division and multiplication by powers of two are **fast**
  - looking at one new piece of data sucks in a cache line
  - with **huge** data sets, disk accesses dominate

Solution: d-Heaps

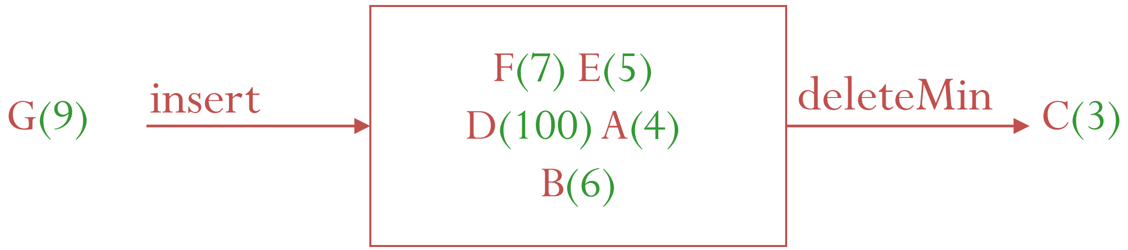
- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - optimize performance based on # of inserts/removes
  - choose a power of two for efficiency
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block



Priority Queue

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - is_empty



- Priority Queue property: for two elements in the queue, *x* and *y*, if *x* has a lower priority value than *y*, *x* will be deleted before *y*

Changing Priorities

- In many applications the priority of an object in a priority queue may change over time
  - if a job has been sitting in the printer queue for a long time increase its priority
  - unix "renice"

- Must have some (separate) way of find the position in the queue of the object to change (*e.g.* a hash table)

Other Priority Queue Operations

- buildHeap
  - given a set of items, build a heap
- decreaseKey
  - given the position of an object in the queue, reduce its priority value
- increaseKey
  - given the position of an object in the queue, increase its priority value
- remove
  - given the position of an object in the queue, remove it

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Simulate events
- Select symbols for compression
- Sort numbers
- Anything *greedy*

References

- Donald E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.
- Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §3.3.1, p.75.
- Donald E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.
- Cormen, Leiserson, and Rivest, Introduction to Algorithms, McGraw Hill, 1990, §11.1, p.200.
- Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §3.3.1, p.75.
- Koffman and Wolfgang, "Objects, Abstraction, Data Structures and Design using C++", John Wiley & Sons, Inc., Ch. 6.
- Wikipedia, http://en.wikipedia.org/wiki/Double-ended_queue
- CSE326: Data Structure, Department of Computer Science and Engineering, University of Washington https://courses.cs.washington.edu/courses/cse326
- Mike Scott, CS 307 Fundamentals of Computer Science, https://www.cs.utexas.edu/~scottm/cs307/
- Debasis Samanta, Computer Science & Engineering, Indian Institute of Technology Kharagpur, Spring-2017, Programming and Data Structures. https://cse.iitkgp.ac.in/~dsamanta/courses/pds/index.html

תודה רבה
Hebrew

Ευχαριστώ
Greek

Спасибо
Russian

Danke
German

धन्यवादः
Sanskrit

Merci
French

ধন্যবাদ
Bangla

நன்றி
Tamil

شكراً
Arabic

ಧನ್ಯವಾದಗಳು
Kannada

Thank You
English

നന്ദി
Malayalam

多謝
Traditional Chinese

Grazie
Italian

ధన్యవాదాలు
Telugu

આભાર
Gujarati

ਧੰਨਵਾਦ
Punjabi

धन्यवाद
Hindi & Marathi

多谢
Simplified Chinese

Gracias
Spanish

https://sites.google.com/site/animeshchaturvedi07

ありがとうございました
Japanese

Obrigado
Portuguese

ขอบคุณ
Thai

감사합니다
Korean