Chapter 10 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. What is the worst-case time complexity of pushing an element into a priority queue of size n that is implemented using an unsorted and a sorted vector as the underlying container, respectively?
   A) Unsorted: $\Theta(1)$, Sorted: $\Theta(1)$
   B) Unsorted: $\Theta(1)$, Sorted: $\Theta(n)$
   C) Unsorted: $\Theta(n)$, Sorted: $\Theta(1)$
   D) Unsorted: $\Theta(n)$, Sorted: $\Theta(n)$
   E) Unsorted: $\Theta(\log(n))$, Sorted: $\Theta(\log(n))$

2. What is the worst-case time complexity of removing the highest priority element from a priority queue of size n that is implemented using an unsorted and a sorted vector as the underlying container, respectively?
   A) Unsorted: $\Theta(1)$, Sorted: $\Theta(1)$
   B) Unsorted: $\Theta(1)$, Sorted: $\Theta(n)$
   C) Unsorted: $\Theta(n)$, Sorted: $\Theta(1)$
   D) Unsorted: $\Theta(n)$, Sorted: $\Theta(n)$
   E) Unsorted: $\Theta(\log(n))$, Sorted: $\Theta(\log(n))$

3. For which of the following real-life situations would a priority queue be **LEAST** useful?
   A) A student who has limited time to study for several exams
   B) An email survey that offers a gift card to the first 100 respondents
   C) An emergency call center that takes calls for an entire city
   D) A professor who wants to assign seats to students in alphabetical order
   E) A restaurant that offers quicker service to loyal rewards members

4. Which of the following statements is **TRUE**?
   A) The total time complexity of inserting an element into a priority queue of size n that is implemented with an unsorted sequence container and then taking it out is worst-case $\Theta(n^2)$
   B) The time complexities of insertion and removal are both worst-case $\Theta(\log(n))$ if a priority queue is implemented using a binary heap
   C) The time complexities of insertion and removal are both $\Theta(n)$ if a priority queue is implemented using a sorted sequence container
   D) Implementing a priority queue using an array of linked lists is always preferable to implementing one using a binary heap, as linked lists allow for $\Theta(1)$ insertion and removal
   E) More that one of the above

5. Your friend implemented a priority queue using an unknown container and sent this implementation to you. In order to identify the type of container that was used, you timed how long it took for this implementation to push and pop n elements, for different values of n. The results are shown in the table below:

Number of Elements *n*	**10,000**	**25,000**	**50,000**	**100,000**	**250,000**	**500,000**	**1,000,000**
Total Push Time for *n* Elements (s)	0.00048	0.00117	0.00244	0.00495	0.00974	0.01902	0.04170
Total Pop Time for *n* Elements (s)	0.21113	0.96067	3.71797	13.1249	62.7012	232.871	937.932

   Which of the following container types could your friend have used to implement this priority queue?
   A) Unsorted sequence container
   B) Sorted sequence container
   C) Binary heap
   D) Pairing heap
   E) More than one of the above

6. Consider the following snippet of code:

```cpp
int main () {
  std::priority_queue<int32_t> pq;
  pq.push(22);
  pq.push(34);
  pq.push(15);
  pq.push(41);
  pq.push(26);
  while (!pq.empty()) {
    std::cout << pq.top() << ' ';
    pq.pop();
  } // while
} // main()
```

   What does this code output?
   A) `15 22 26 34 41`
   B) `22 34 15 41 26`
   C) `41 34 26 22 15`
   D) `26 41 15 34 22`
   E) None of the above

7. Consider the following snippet of code:

```cpp
struct CustomComparator {
  bool operator() (const int32_t lhs, const int32_t rhs) const {
    // abs returns the absolute value of x
    return std::abs(lhs - 25) > std::abs(rhs - 25);
  } // operator()()
};

int main () {
  std::priority_queue<int32_t, std::vector<int32_t>, CustomComparator> pq;
  pq.push(22);
  pq.push(34);
  pq.push(15);
  pq.push(41);
  pq.push(26);
  while (!pq.empty()) {
    std::cout << pq.top() << ' ';
    pq.pop();
  } // while
} // main()
```

What does this code output?

A) `41 34 26 22 15`
B) `41 15 34 22 26`
C) `15 22 26 34 41`
D) `26 22 34 15 41`
E) None of the above

8. Which of the following STL data structures can be used as the underlying data container for a `std::priority_queue<>`?

I. `std::vector<>`
II. `std::list<>`
III. `std::deque<>`

A) I only
B) III only
C) I and II only
D) I and III only
E) I, II, and III

9. Consider the following two functions, `foo()` and `bar()`, which each take in a vector of integers and inserts its values into a priority queue. The `foo()` method iterates over the vector and pushes in the values one by one, while the `bar()` method uses a range constructor to construct the resultant priority queue.

```cpp
std::priority_queue<int32_t> foo(const std::vector<int32_t>& input) {
  std::priority_queue<int32_t> pq;
  for (const int32_t val : input) {
    pq.push(val);
  } // for val

  return pq;
} // foo()

std::priority_queue<int32_t> bar(const std::vector<int32_t>& input) {
  std::priority_queue<int32_t> pq(input.begin(), input.end());
  return pq;
} // bar()
```

If the input vector has a size of n, what are the time complexities of `foo()` and `bar()`?

A) `foo()`: $\Theta(n)$, `bar()`: $\Theta(n)$
B) `foo()`: $\Theta(n)$, `bar()`: $\Theta(n\log(n))$
C) `foo()`: $\Theta(n\log(n))$, `bar()`: $\Theta(n)$
D) `foo()`: $\Theta(n\log(n))$, `bar()`: $\Theta(n\log(n))$
E) None of the above

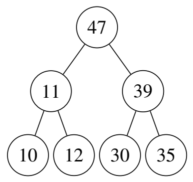
10. Which of the following statements is **FALSE** regarding the pairing heap?

A) A node in a pairing heap may have more than two children
B) A pairing heap can be implemented using parent or previous pointers
C) A queue can be effectively used to meld smaller pairing heaps into a larger pairing heap
D) The worst-case time complexity of inserting an element into a pairing heap of size n cannot be better than $\Theta(\log(n))$
E) None of the above

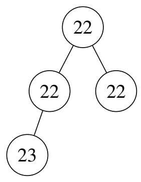
11. Which one of the following statements is **NOT** necessarily true?
    A) In a max-heap, the key at each node must be larger than the keys of the node's children
    B) No node in a max-heap has a key larger than the root's key
    C) A max-heap gives easy access to the largest element in the heap
    D) A heap must have the completeness property
    E) Heaps can be implemented using both binary trees and arrays

12. Which of the following are valid binary heaps?

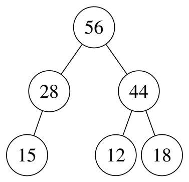
A)



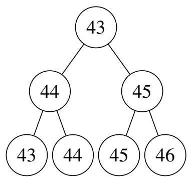
B)



C)



D)



E) More than one of the above are valid binary heaps

13. Which of the following represents a valid binary **min**-heap? The "top" of the heap is the element at the beginning of the array.
    A) `[2, 13, 8, 16, 13, 10, 40, 25, 17]`
    B) `[47, 9, 12, 9, 2, 10, 10, 4, 3, 1]`
    C) `[3, 5, 6, 7, 12, 15, 14, 9, 10, 11]`
    D) `[59, 58, 60, 57, 85, 49, 32, 21, 5]`
    E) None of the above

14. Which of the following represents a valid binary **max**-heap? The "top" of the heap is the element at the beginning of the array.
    A) `[14, 25, 27, 26, 28, 24, 30, 32, 34]`
    B) `[52, 11, 23, 9, 10, 12, 11, 10, 9, 8]`
    C) `[33, 24, 24, 7, 9, 24, 18, 7, 5, 9, 8]`
    D) `[76, 54, 33, 56, 32, 55, 33, 12, 14]`
    E) None of the above

15. What is the worst-case time complexity of fixing a max-heap of size n after an element in the heap is modified?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

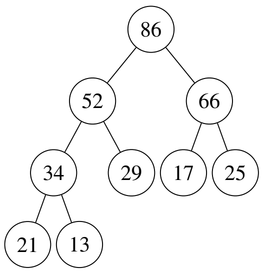
16. What is the worst-case auxiliary space required to run heapify on an array of n integers, if you use the most efficient implementation?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

17. Four approaches to heapify are described below:
    I. Proceeding from the bottom of the heap to the top, while repeatedly calling `fix_up()`
    II. Proceeding from the bottom of the heap to the top, while repeatedly calling `fix_down()`
    III. Proceeding from the top of the heap to the bottom, while repeatedly calling `fix_up()`
    IV. Proceeding from the top of the heap to the bottom, while repeatedly calling `fix_down()`

    Which of the above approaches successfully builds a valid heap?
    A) I and II only
    B) I and IV only
    C) II and III only
    D) III and IV only
    E) I, II, III, and IV

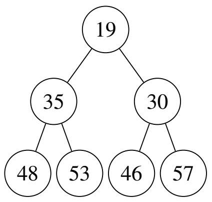
18. Why is it more efficient to build a heap using the `fix_down()` approach instead of the `fix_up()` approach?
    A) If you correctly heapify an array of size n using `fix_down()`, you would only need to call `fix_down()` on $\Theta(\log(n))$ elements instead of $\Theta(n)$ elements
    B) If you correctly heapify an array of size n using `fix_up()`, you would have to call `fix_up()` on $\Theta(n\log(n))$ elements instead of $\Theta(n)$ elements
    C) If you correctly heapify an array using `fix_down()`, you wouldn't have to call `fix_down()` on any internal nodes
    D) If you correctly heapify an array using `fix_down()`, you wouldn't have to call `fix_down()` on any leaf nodes
    E) More than one of the above

19. Consider the following max-heap:



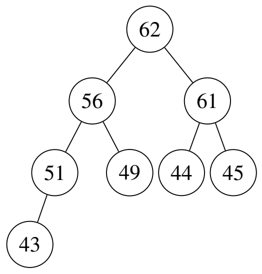
You insert the value of 93 into this max-heap. After the invariant is fixed, what is the final array representation of this max-heap?
A) `[93, 86, 66, 34, 52, 17, 25, 21, 13, 29]`
B) `[93, 86, 66, 52, 34, 17, 25, 21, 13, 29]`
C) `[93, 52, 86, 34, 29, 66, 25, 21, 13, 17]`
D) `[93, 86, 66, 52, 29, 17, 25, 34, 13, 21]`
E) None of the above

20. Consider the following min-heap:



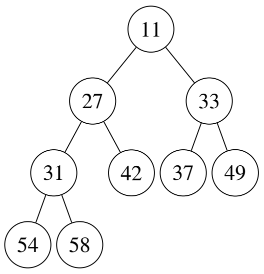
You insert the value of 26 into this min-heap. After the invariant is fixed, what is the final array representation of this min-heap?
A) `[19, 26, 30, 35, 48, 46, 57, 53]`
B) `[19, 26, 30, 35, 53, 46, 57, 48]`
C) `[19, 35, 30, 26, 53, 46, 57, 48]`
D) `[19, 35, 30, 48, 53, 46, 57, 26]`
E) None of the above

21. Consider the following max-heap:



You delete the value of 62 from this max-heap. After the invariant is fixed, what is the final array representation of this max-heap?
A) `[61, 56, 43, 51, 49, 44, 45]`
B) `[61, 56, 45, 51, 49, 44, 43]`
C) `[61, 56, 51, 43, 49, 44, 45]`
D) `[61, 56, 51, 49, 44, 45, 43]`
E) None of the above

22. Consider the following min-heap:



You delete the value of 11 from this min-heap. After the invariant is fixed, what is the final array representation of this min-heap?
A) `[33, 27, 37, 31, 42, 58, 49, 54]`
B) `[27, 42, 33, 31, 58, 37, 49, 54]`
C) `[58, 27, 33, 31, 42, 37, 49, 54]`
D) `[27, 31, 33, 54, 42, 37, 49, 58]`
E) None of the above

23. Consider the following unsorted array:

`[13, 24, 1, 58, 69, 10, 32, 27, 71]`

Perform a $\Theta(n)$ heapify operation to turn this data into a valid **max**-heap. What are the contents of the array after the heapify operation?
A) `[71, 13, 32, 24, 69, 10, 1, 27, 58]`
B) `[71, 69, 32, 27, 13, 10, 1, 24, 58]`
C) `[71, 69, 58, 32, 27, 13, 10, 24, 1]`
D) `[71, 69, 32, 58, 13, 10, 1, 27, 24]`
E) None of the above

24. Consider the following unsorted array:

`[63, 47, 50, 15, 58, 39, 26, 53, 12, 57]`

Perform a $\Theta(n)$ heapify operation to turn this data into a valid **min**-heap. What are the contents of the array after the heapify operation?
A) `[12, 15, 26, 47, 53, 39, 50, 57, 63, 58]`
B) `[12, 15, 26, 47, 58, 39, 50, 53, 57, 63]`
C) `[12, 15, 26, 47, 57, 39, 50, 53, 63, 58]`
D) `[12, 15, 26, 53, 47, 39, 50, 58, 63, 57]`
E) None of the above

25. Given an empty **min**-heap priority queue, you push the following values in this order:

34, 29, 22, 25, 19, 12, 15, 37

What would the contents of the underlying array look like if the top value were popped off the heap?
A) `[12, 22, 15, 34, 25, 29, 19]`
B) `[15, 22, 19, 34, 25, 29, 37]`
C) `[15, 22, 37, 34, 25, 29, 19]`
D) `[15, 19, 22, 34, 25, 29, 37]`
E) None of the above

26. Consider a **min**-heap represented by the following array:

`[63, 74, 67, 85, 91, 94, 72, 88]`

Perform the following operations using the algorithms for binary heaps discussed in this chapter. Ensure that the heap property is restored at the end of every individual operation.
1. Push the value of 60 into this min-heap.
2. Push the value of 79 into this min-heap.
3. Update element 85 to have a value of 58.
4. Update element 67 to have a value of 96.
5. Remove the min element from the heap.

What does the array representation look like after all five operations are completed?
A) `[60, 72, 63, 94, 96, 74, 79, 88, 91]`
B) `[60, 63, 72, 74, 79, 94, 96, 88, 91]`
C) `[60, 74, 79, 63, 91, 94, 72, 88, 96]`
D) `[60, 63, 72, 74, 79, 88, 91, 94, 96]`
E) None of the above

27. Given an empty **max**-heap priority queue, push in the following letters in this order (priority is determined using alphabetical order, where later letters have higher priority than earlier letters):

H, E, L, L, O, W, O, R, L, D

What would the contents of the underlying array look like after all the letters are inserted?
A) `[W, R, O, L, L, H, O, E, L, D]`
B) `[W, O, R, L, L, H, O, E, L, D]`
C) `[W, R, O, O, H, L, L, E, L, D]`
D) `[W, R, O, L, O, L, H, E, L, D]`
E) None of the above

28. You are given the following array, with the following contents:

`[H, E, L, L, O, W, O, R, L, D]`

Perform a *bottom-up* heapify operation to turn this data into a valid **max**-heap. What are the contents of the array after the heapify operation? Note: for this problem, when fixing down, swap with the left child in the case of a tie.
A) `[W, R, O, L, L, H, O, E, L, D]`
B) `[W, O, R, L, L, H, O, E, L, D]`
C) `[W, R, O, O, H, L, L, E, L, D]`
D) `[W, R, O, L, O, L, H, E, L, D]`
E) None of the above

29. You are given the following array, with the following contents:

    `[H, E, L, L, O, W, O, R, L, D]`

    Perform a *top-down* heapify operation to turn this data into a valid **max**-heap. What are the contents of the array after the heapify operation?
    A) `[W, R, O, L, L, H, O, E, L, D]`
    B) `[W, O, R, L, L, H, O, E, L, D]`
    C) `[W, R, O, O, H, L, L, E, L, D]`
    D) `[W, R, O, L, O, L, H, E, L, D]`
    E) None of the above

30. Given an empty **min**-heap priority queue, push in the following letters in this order (priority is determined using alphabetical order, where earlier letters have higher priority than later letters):

    D, O, U, B, L, E, D, E, L, E, T, E

    What would the contents of the underlying array look like after all the letters are inserted?
    A) `[B, D, D, E, E, E, E, U, L, L, O, T]`
    B) `[B, D, D, E, E, E, E, O, L, L, T, U]`
    C) `[B, D, D, E, E, E, U, O, L, L, T, E]`
    D) `[B, D, D, E, E, E, U, L, L, T, O, E]`
    E) None of the above

31. You are given the following array, with the following contents:

    `[D, O, U, B, L, E, D, E, L, E, T, E]`

    Perform a *bottom-up* heapify operation to turn this data into a valid **min**-heap. What are the contents of the array after the heapify operation?
    A) `[B, D, D, E, E, E, E, U, L, L, O, T]`
    B) `[B, D, D, E, E, E, E, O, L, L, T, U]`
    C) `[B, D, D, E, E, E, U, O, L, L, T, E]`
    D) `[B, D, D, E, E, E, U, L, L, T, O, E]`
    E) None of the above

32. You are given the following array, with the following contents:

    `[D, O, U, B, L, E, D, E, L, E, T, E]`

    Perform a *top-down* heapify operation to turn this data into a valid **min**-heap. What are the contents of the array after the heapify operation?
    A) `[B, D, D, E, E, E, E, U, L, L, O, T]`
    B) `[B, D, D, E, E, E, E, O, L, L, T, U]`
    C) `[B, D, D, E, E, E, U, O, L, L, T, E]`
    D) `[B, D, D, E, E, E, U, L, L, T, O, E]`
    E) None of the above

33. Suppose you wanted to implement a `pop_random()` method for a priority queue of size n that is implemented using an array-based binary heap. When this method is invoked, a random element is swapped with the last element in the heap's underlying array and then popped out from the back. What is the *minimum* number of times you must call `fix_up()` or `fix_down()` on the resultant array to guarantee that the binary heap property is preserved after this random element is swapped to the back and then popped out of the array?
    A) 1
    B) 2
    C) $\lceil \log(n) \rceil$
    D) $\lceil n/2 \rceil$
    E) n

34. What is the maximum possible difference in depth between any two leaf nodes of a binary heap containing n nodes with a height h?
    A) 0
    B) 1
    C) h
    D) $\lceil \log(n) \rceil$
    E) $\lceil n/h \rceil$

35. What is the worst-case time complexity of removing the highest priority element from a *heap-ordered* binary tree of size n that is **NOT** complete, provided that the tree must still remain heap-ordered after the removal?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n \log(n))$
    E) $\Theta(n^2)$

36. After removing the top element from a binary heap and then calling `fix_down()` on the new root R (which was the value that was swapped from the back of the heap to the top, before the top element was removed), you notice that R ended up at the last position of the underlying array. Which of the following claims is guaranteed to be **TRUE**?
    A) The priority of R is the highest of all remaining elements in the binary heap after the removal
    B) The priority of R is the lowest of all remaining elements in the binary heap after the removal
    C) The priority of R is at least as high as the priority of the element that was removed
    D) More than one of the above
    E) None of the above

37. Which of the following **MUST** be implemented for a type `T` for the following code to compile?

    ```
    std::priority_queue<T> pq;
    ```

    A) `operator<`
    B) `operator==`
    C) `operator++`
    D) More than one of the above
    E) None of the above

38. You are given an integer array `nums` and an integer k. Implement a function that returns the k^{th} largest element in the array. For example, given the array `[5,2,3,1,6,4]` and $k = 3$, you would return 4, since that is the 3rd largest element in the array. You may assume that k is a valid number between 1 and the length of the array.

    ```
    int32_t find_kth_largest(const std::vector<int32_t>& nums, int32_t k);
    ```

    You solution should run in $\Theta(n\log(k))$ time and take up worst-case $\Theta(k)$ auxiliary space, where n is the size of the input array.

39. You are stuck in a underground tunnel, and a giant pile of rubble of size S is blocking your escape route. Luckily, you have N different sticks of dynamite on hand, each with its own explosive power E. If you use the i^{th} stick of dynamite, the amount of rubble blocking your way decreases by E_i. Your goal is to get rid of all the rubble so that you can escape, but there is a catch: you are given an additional number k, such that you can only use **at most** k sticks of dynamite. Implement the following function, which returns the minimum number of sticks of dynamite necessary to remove all the rubble. Return `-1` if it is impossible to escape using at most k sticks of dynamite.

    ```
    int32_t min_tries_to_escape(const std::vector<int32_t>& dynamite, int32_t rubble_size, int32_t k);
    ```

    For example, if you are given `dynamite = [1,4,5,2,4]`, `rubble_size = 12`, and $k = 4$, you would return a value of 3, since at least three sticks of dynamite are needed to fully remove the rubble (the ones with explosive power 4, 5, and 4). However, if k were 2 (and everything else were the same), then you return `-1`, since it would be impossible to remove all the rubble with just two sticks of dynamite.

    Your solution should run in worst-case $\Theta(n\log(k))$ time and $\Theta(k)$ auxiliary space, where n is the length of the `dynamite` vector.

40. You are given an array of `points` where `points[i] = [x_i, y_i]` represents a point on the Cartesian plane, as well as an integer k. Implement a function that returns the k closest points to the origin (0, 0). Note that the distance between any two points (x_1, y_1) and (x_2, y_2) on a Cartesian plane is equal to $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. You may return the solution in any order, and the points in the solution are guaranteed to be unique. For example, given `[[101, 183],[-203,280],[281,-370]]` and $k = 2$, you would return `[[101,183],[-203,280]]` (in any order), since these are the two points that are closest to the origin.

    ```
    std::vector<std::vector<int32_t>> k_closest_to_origin(
      const std::vector<std::vector<int32_t>>& points, int32_t k);
    ```

    Your solution should run in worst-case $\Theta(n\log(k))$ time and $\Theta(k)$ auxiliary space, where n is the number of points in the input vector.

41. It is the middle of summer, and the professors are looking to hire a new batch of EECS 281 student instructors to prepare for yet another busy fall semester. You are given a $\Theta(1)$ function `get_qualification()`, which is a secret, proprietary algorithm that returns a score from 0 to 100 indicating how qualified an applicant is to become an EECS 281 staff member (where 0 is the least qualified and 100 is the most qualified). Implement the function `hire_staff()`, which takes in a vector of `Applicant` objects and the number of openings k, and returns a vector of the k applicants that should be hired (based on whoever has the highest qualification scores). If two applicants have the same qualification score, choose the one with the lower application ID first.

    ```
    struct Applicant {
      std::string uniqname;
      int32_t application_id;
    };

    // This method is already implemented for you - it returns a number
    // in the range [0, 100] that identifies how qualified an applicant is
    int32_t get_qualification(const Applicant& applicant);

    // Implement this function
    std::vector<Applicant> hire_staff(const std::vector<Applicant>& applicants, int32_t k);
    ```

    Your solution should run in worst-case $\Theta(n\log(k))$ time and $\Theta(k)$ auxiliary space, where n is the number of applicants in the input vector.

Chapter 10 Exercise Solutions

1. **The correct answer is (B).** If the underlying priority queue container is unsorted, then inserting the element is trivial (since you can insert it anywhere). However, in a sorted container, you will have to find the position to insert in (and shift all subsequent elements after the point of insertion), which would take linear time.

2. **The correct answer is (C).** If the underlying priority queue container is unsorted, then the item with the highest priority that you want to remove could be anywhere in the container, which would require a linear search before you can remove it. On the other hand, if the container is sorted, you know where the highest priority element is and can remove it in constant time.

3. **The correct answer is (B).** While all of these options involve a choice to be made depending on some sort of priority (such as the importance of an exam), the scenario in option (B) bases its priority on FIFO arrival order, which is more similar to the functionality of a standard queue rather than a priority queue.

4. **The correct answer is (B).** Option (A) is false because unsorted sequence containers provide $\Theta(1)$ insertion and $\Theta(n)$ removal. Option (C) is false because sorted sequence containers provide $\Theta(1)$ removal. Option (D) is false because an array of linked lists is only viable for priorities of small integers and may involve other inefficiencies (such as memory overhead). Only option (B) is true: if you use a binary heap to implement a priority queue, then the time complexities of insertion and removal are both $\Theta(\log(n))$.

5. **The correct answer is (A).** Notice that the time rquired to pop elements grows much more rapidly than the time required to push elements as the number of elements you have to push or pop increases. This would point to a container where inserting is easy and removing is difficult. Of the choices provided, an unsorted sequence container would make the most sense, as removing is a linear time operation due to the unsorted nature of the container (you have to linear search for the element with the greatest priority to pop). An unsorted sequence container provides $\Theta(1)$ insertion and $\Theta(n)$ removal, which would be result in a $\Theta(n)$ and $\Theta(n^2)$ relationship in total if performed n times. This fits with the data provided: the time for push grows linearly with n, while the time for pop grows quadratically with n. None of the other answer choices exhibit these same time complexities for push and pop.

6. **The correct answer is (C).** The `std::priority_queue<>` defaults to the `std::less (<)` comparator if no comparator is explicitly provided, so larger elements have the greatest priority. Thus, the elements are popped out in descending order.

7. **The correct answer is (D).** The comparator is in a `std::greater (>)` format, so elements with a smaller value for `abs(rhs - 25)` have greater priority and are popped out first. In other words, the elements are popped out in order of how far away they are from 25, with elements closer to 25 popped out before elements farther from 25.

8. **The correct answer is (D).** The underlying container for a `std::priority_queue<>` must be sequential and support constant time random access. This only applies for vectors and deques.

9. **The correct answer is (C).** Each individual call to `.push()` takes $\Theta(\log(n))$ time, so the time complexity of `foo()` is $n \times \Theta(\log(n)) = \Theta(n\log(n))$. However, if you use the range constructor, the input range can be heapified in $\Theta(n)$ time instead. This is why it is better to initialize a priority queue with a range of data using the range constructor (if possible), instead of pushing each element in one by one.

10. **The correct answer is (D).** Because priority queues can have more than two children, inserting an element can be done in constant time by just melding the lower priority root as a child of the higher priority one.

11. **The correct answer is (A).** Option (A) is not necessarily true because a key at each node may be identical to one of its children due to the possibility of ties. It does not always have to be larger.

12. **The correct answer is (B).** Option (A) is not a valid binary heap because 12 is larger than 11. Option (C) is not a valid binary heap because it is not complete (28 has one child but 44 has two). Option (D) is not a valid binary heap because 44's parent and child both include 43.

13. **The correct answer is (A).** A min-heap is valid if the children of each element (located at indices $2i$ and $2i+1$ using 1-indexing) are not smaller than it. Option (B) is not a valid binary min-heap because the children of 47, 9 and 12, are smaller than 47. Option (C) is not a valid binary min-heap because the child of 12 (11) is smaller than 12. Option (D) is not a valid binary min-heap because a child of 59 (58) is smaller than 59. Option (A) is valid: 13 and 8 are not smaller than 2, 16 and 13 are not smaller than 13, 10 and 40 are not smaller than 8, and 25 and 17 are not smaller than 16.

14. **The correct answer is (C).** A max-heap is valid if the children of each element (located at indices $2i$ and $2i+1$ using 1-indexing) are not larger than it. Option (A) is not a valid binary max-heap because the children of 14 (25 and 27) are not larger than 14. Option (B) is not a valid max-heap because a child of 9 (10) is larger than 9. Option (D) is not a valid binary max-heap because a child of 54 (56) is larger than 54. Option (C) is valid: 24 and 24 are not larger than 33, 7 and 9 are not larger than 24, 24 and 18 are not larger than 24, 7 and 5 are not larger than 7, and 9 and 8 are not larger than 9.

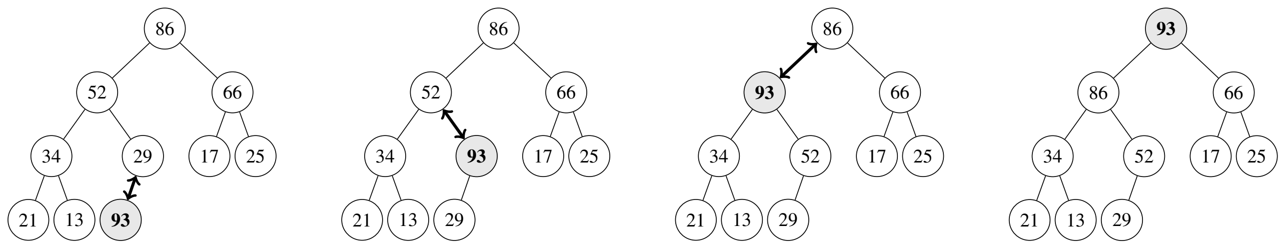
15. **The correct answer is (B).** After an element in a max-heap is modified, it only needs to move at most $\Theta(\log(n))$ levels to return to a valid location (since there are $\Theta(\log(n))$ levels in a heap with n elements).

16. **The correct answer is (A).** You can perform a heapify in place, using the given array of elements and either `fix_up()` or `fix_down()`.

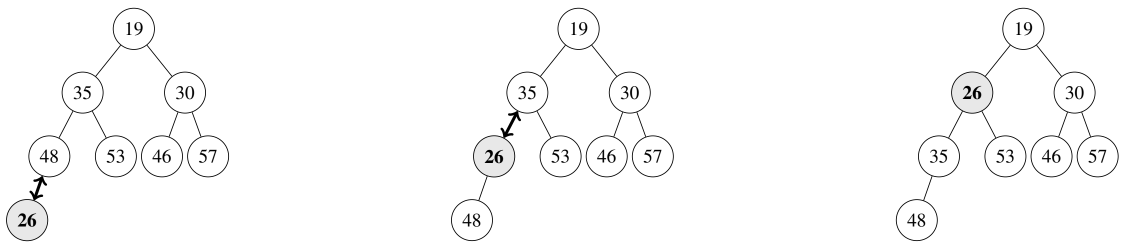
17. **The correct answer is (C).** To produce a valid heap, you must fix in the direction you are coming from. As an example, option I fails when the array is `[5,3,6,1,2,4,7]`. Option IV fails when the array is `[1,2,3,4,5,6,7]`.

18. **The correct answer is (D).** If you heapify an array by calling `fix_down()`, you wouldn't have to call `fix_down()` on any of the leaf nodes since they have no children (`fix_down()` tries to fix in the direction of the children, but leaf nodes don't have any).

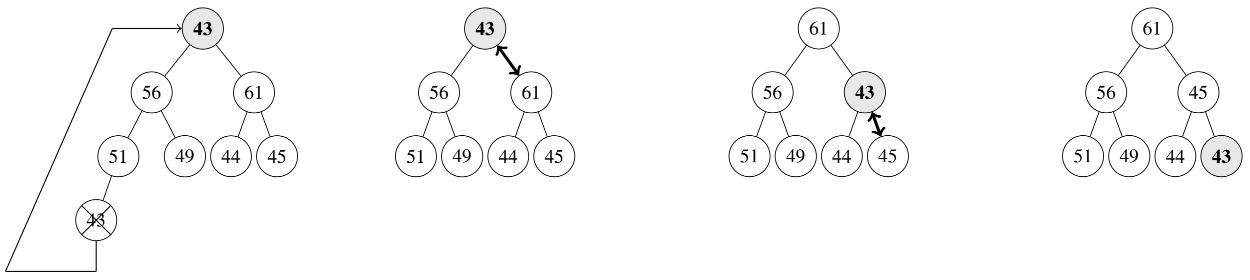
19. **The correct answer is (A).** Add 93 to the end of the heap and move it up until it is in the correct position.



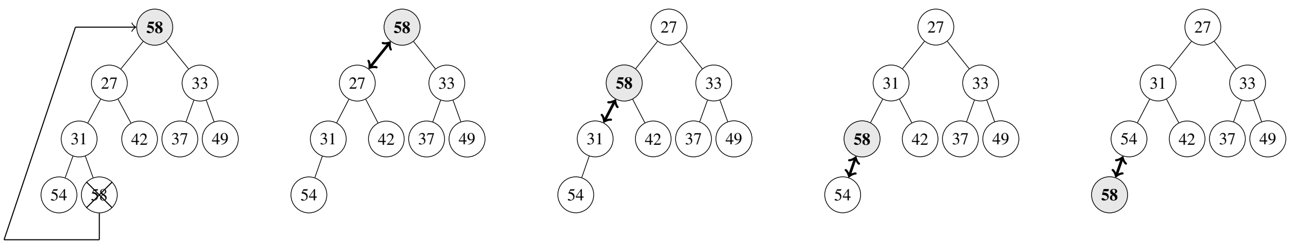
20. **The correct answer is (B).** Add 26 to the end of the heap and move it up until it is in the correct position.



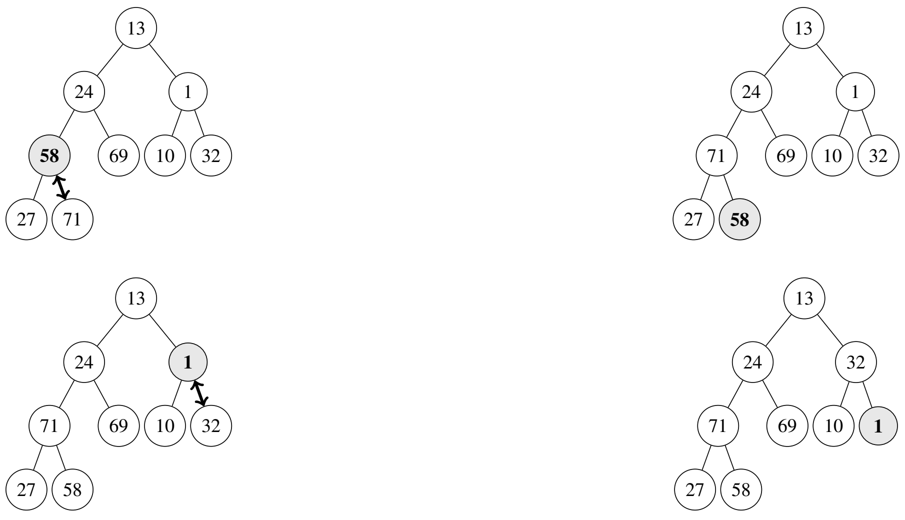
21. **The correct answer is (B).** Move 43 to the top of the heap after deleting 62, and then fix it down until it is in the correct position.

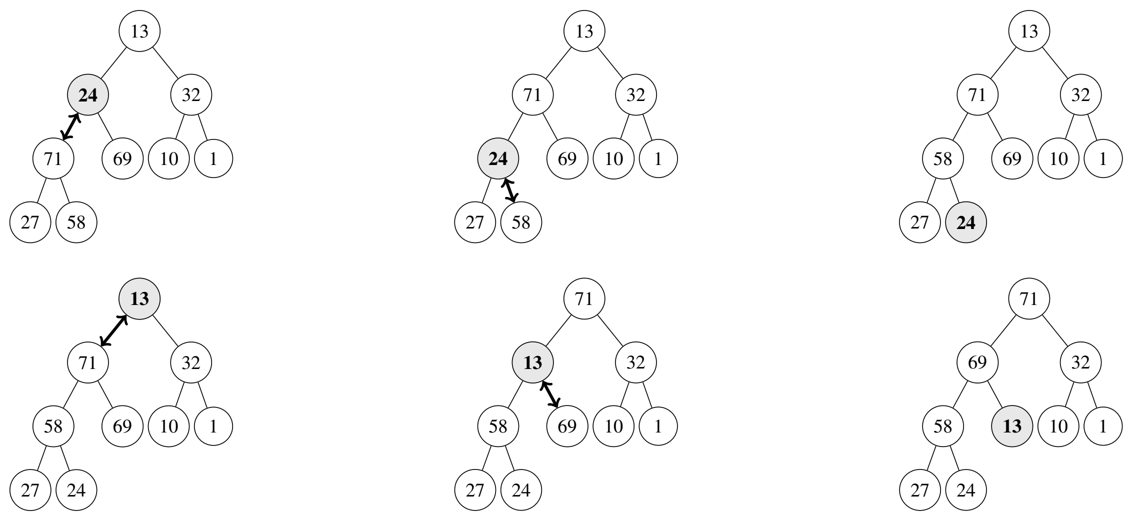


22. **The correct answer is (D).** Move 58 to the top of the heap after deleting 11, and then fix it down until it is in the correct position.

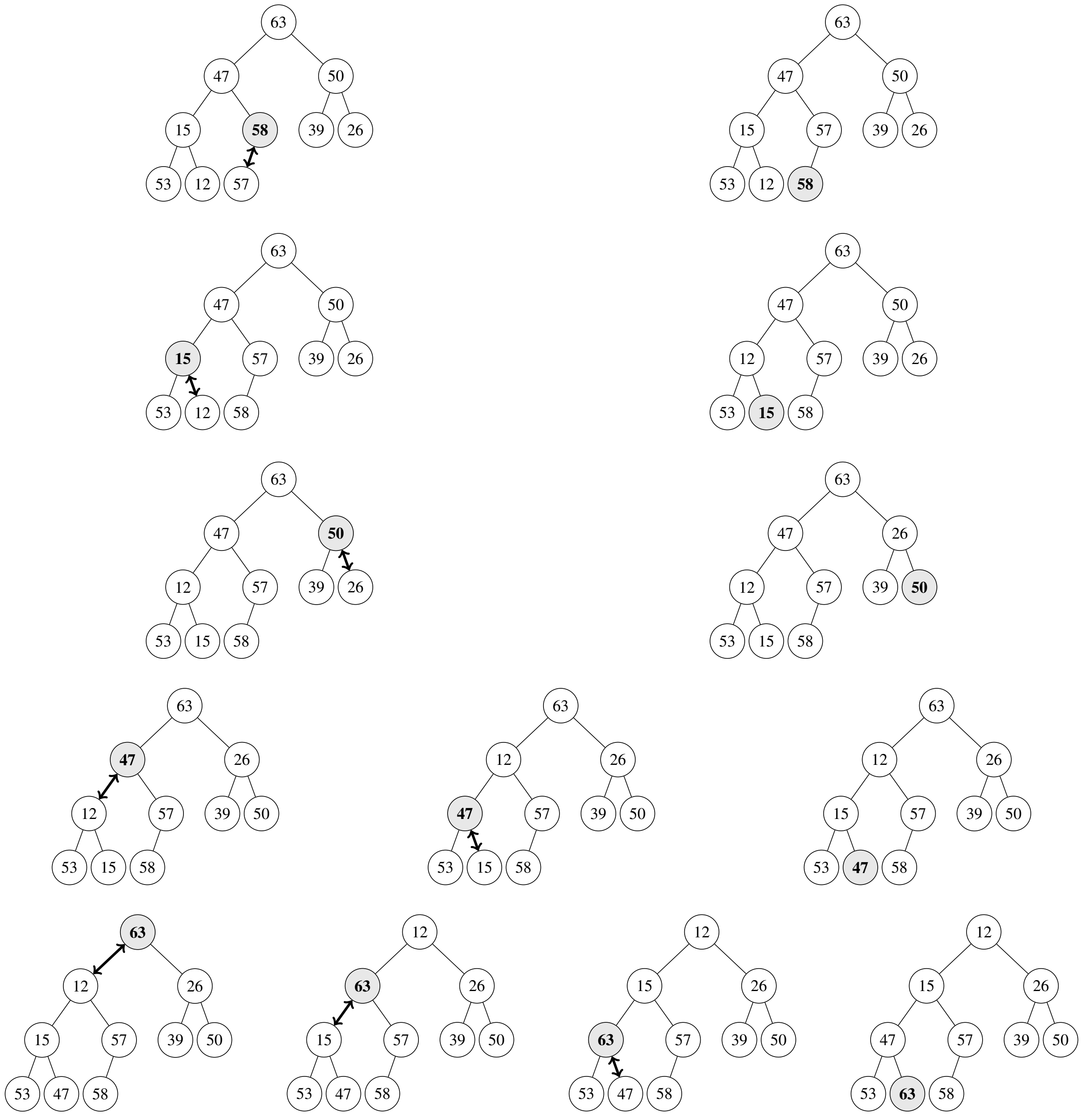


23. **The correct answer is (D).** For a $\Theta(n)$ heapify, start from the bottom internal nodes and move upwards, continuously calling `fix_down()`:

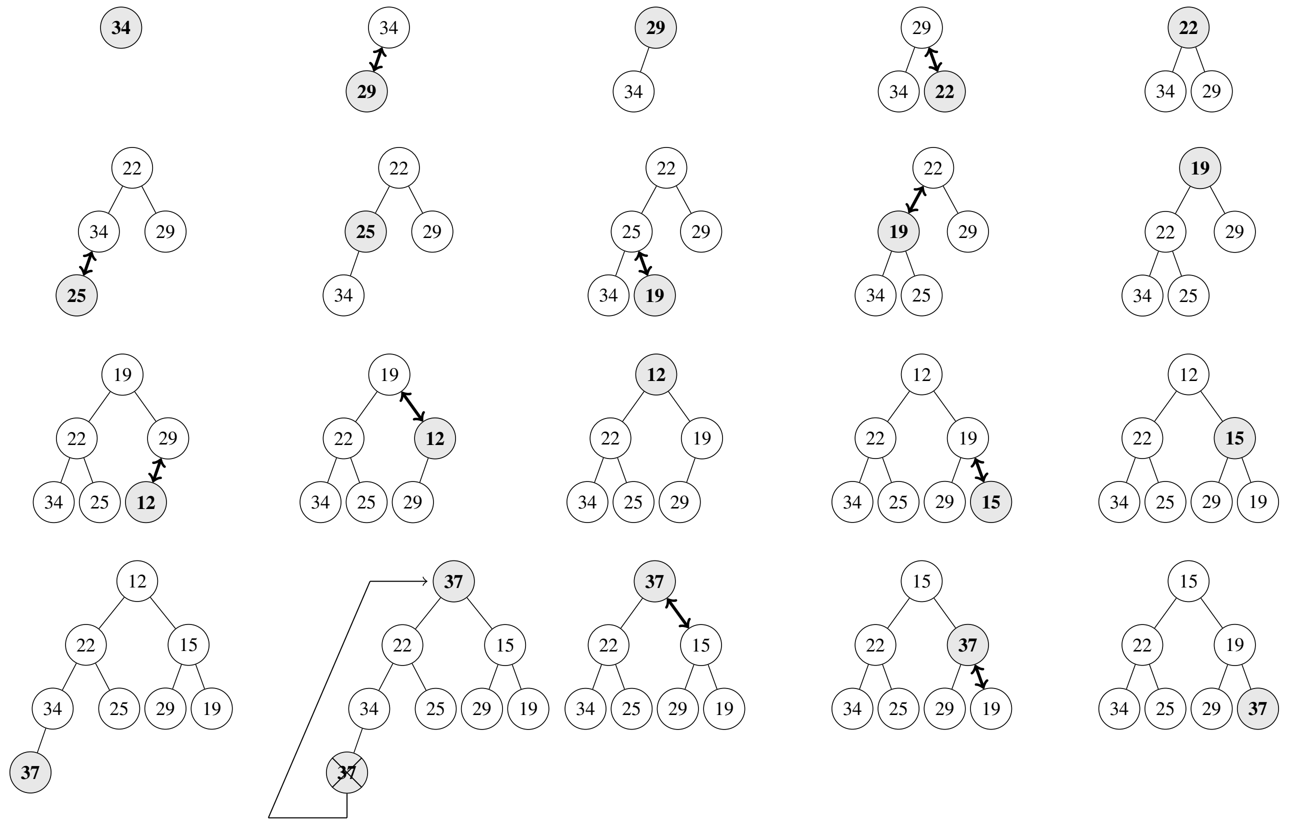




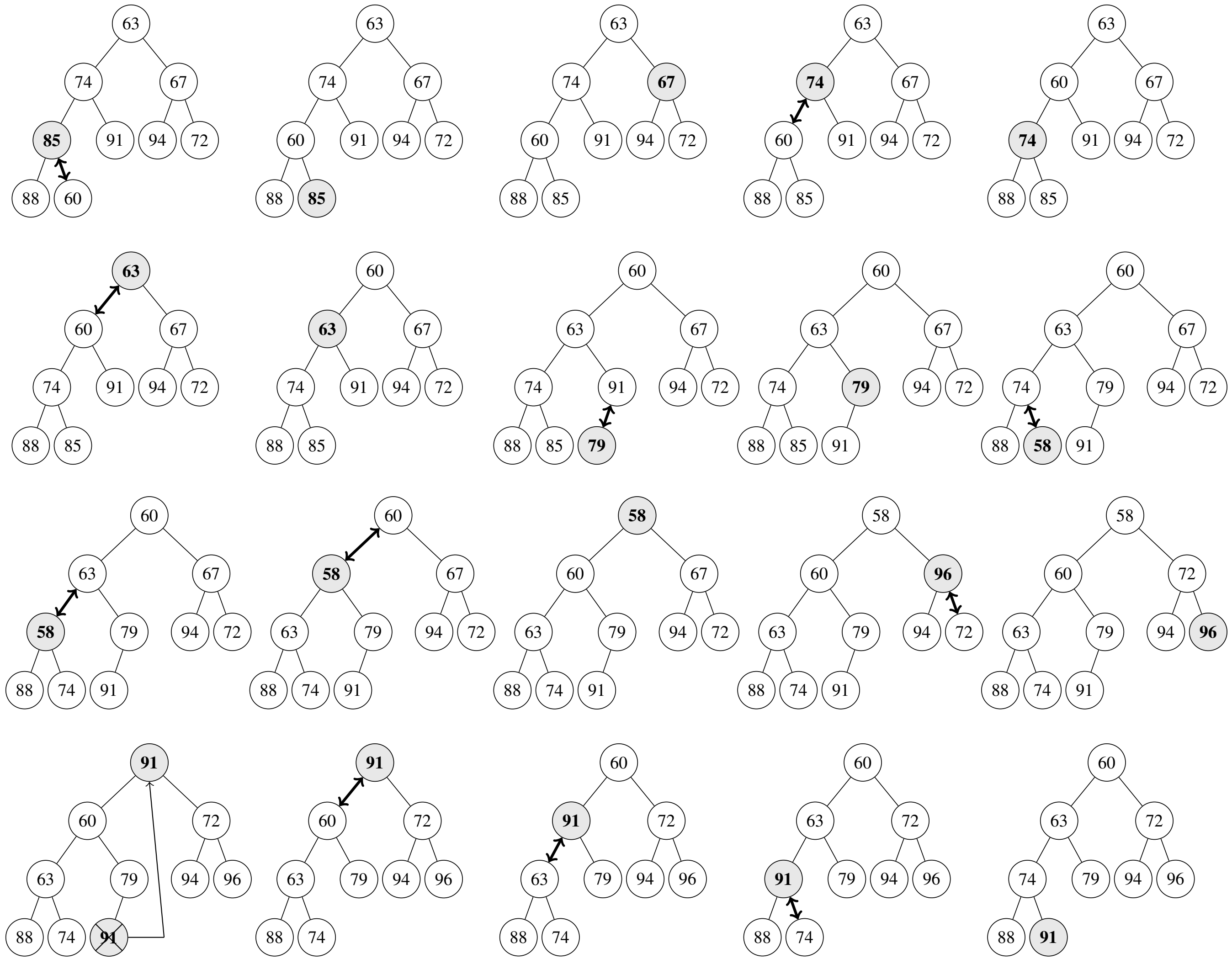
24. **The correct answer is (C).** For a $\Theta(n)$ heapify, start from the bottom internal nodes and move upwards, continuously calling `fix_down()`:



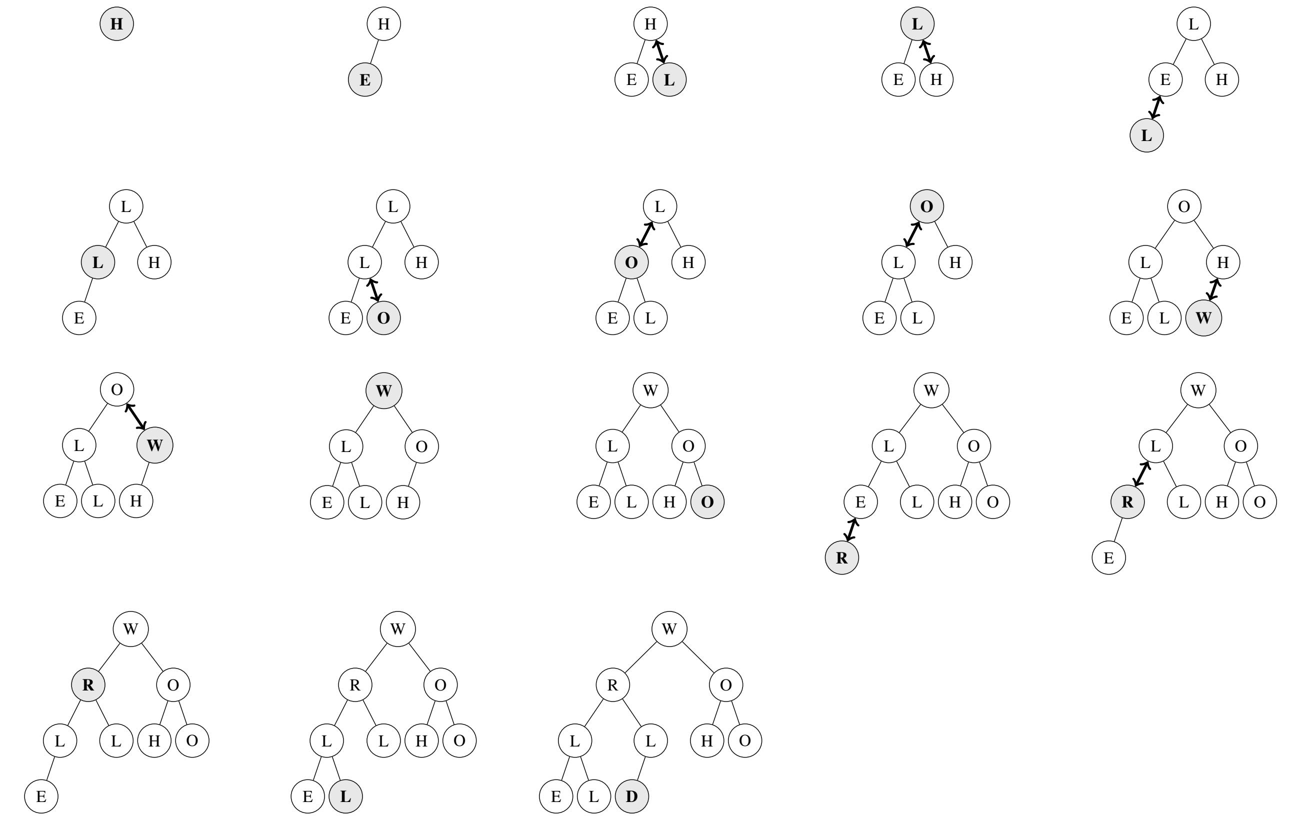
25. **The correct answer is (D).** The steps are shown below:



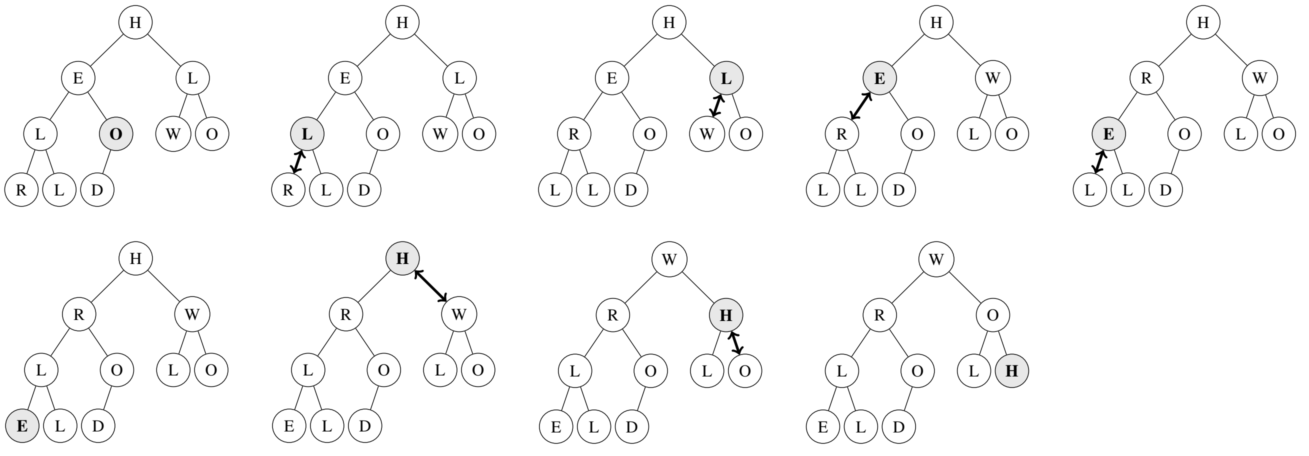
26. **The correct answer is (B).** The steps are shown below:



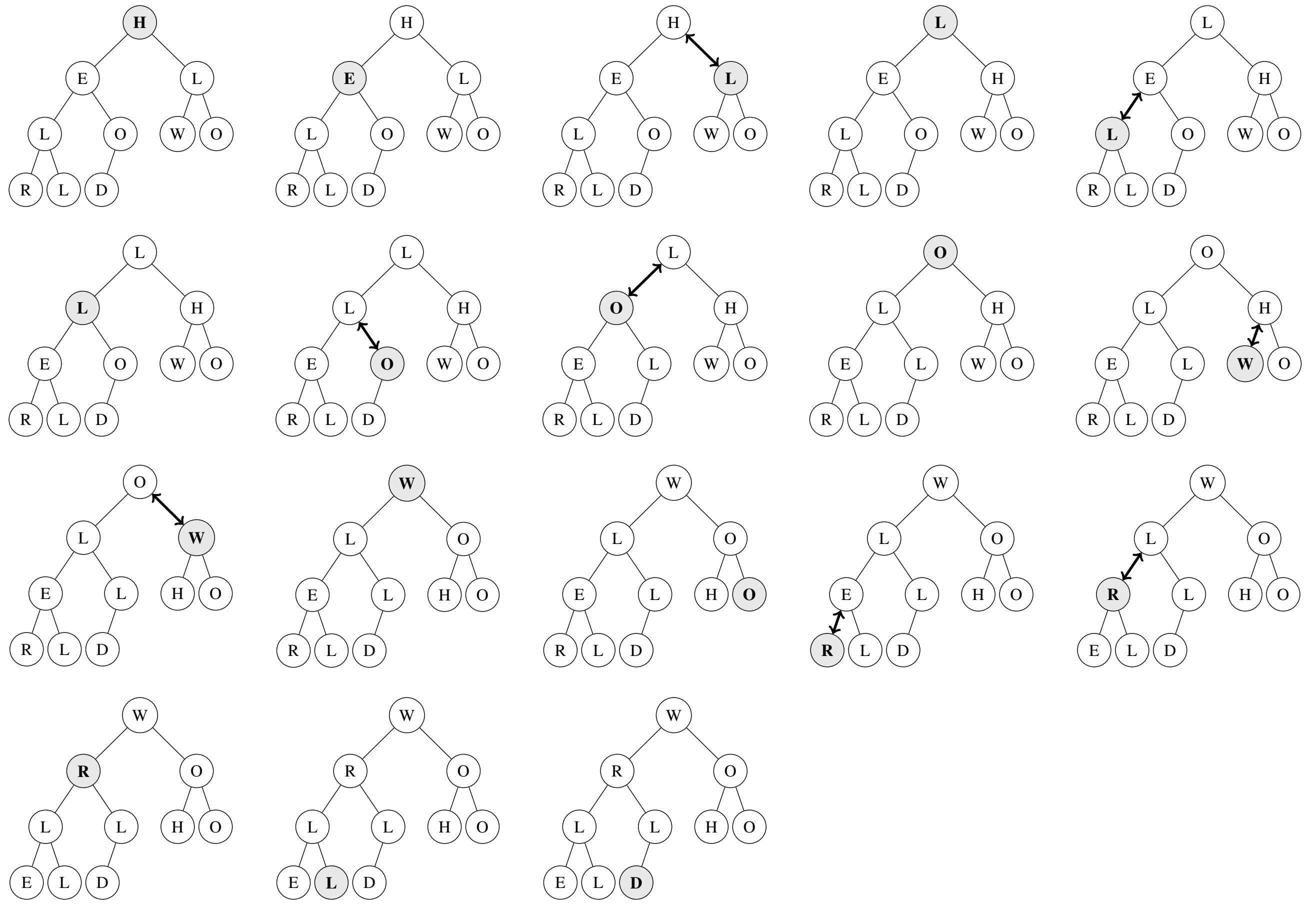
27. **The correct answer is (A).** The steps are shown below:



28. **The correct answer is (D).** Start from the bottom internal nodes and move upwards, continuously calling `fix_down()`:



29. **The correct answer is (A).** Start from the top and move downwards, continuously calling `fix_up()`:



30. **The correct answer is (B).** See question 27 for an example on how to approach this problem.

31. **The correct answer is (C).** See question 28 for an example on how to approach this problem.

32. **The correct answer is (B).** See question 29 for an example on how to approach this problem.

33. **The correct answer is (A).** After the swap and removal, there is only one value that could be out of place, so only one call to `fix_up()` or `fix_down()` is needed to fix this value.

34. **The correct answer is (B).** A binary heap must be complete, so there cannot be a gap of more than one level between two leaf nodes.

35. **The correct answer is (C).** If a heap-ordered binary tree is not guaranteed to be complete, you no longer have a $\Theta(\log(n))$ limit on the height of the tree. In the worst-case, the tree is in the form of a stick, which could cause you to iterate all n nodes while fixing an element.

36. **The correct answer is (E).** Just because R moved to the last position of the underlying array after swapping and fixing down, it does not mean that R has a lower priority than all other elements. Any of the leaf nodes in the binary heap could be the value with the lowest priority, and not just the value at the last position in the underlying array.

37. **The correct answer is (A).** Only `operator<` is needed for the priority queue initialization to compile.

38. You can solve this problem trivially by sorting the array, or by pushing all the elements into a max-priority queue and popping k elements out, but these solutions would end up taking $\Theta(n\log(n))$ time. How do we do better than $\Theta(n\log(n))$? The idea is to use a min-heap of size k instead, and only push in a value if it is larger than the value at the top of the min-priority queue. By doing so, the min-priority queue essentially stores the k largest values you have encountered so far, and you can easily check if a new element is among the k largest you have encountered by comparing its value with the smallest value at the top of the min-heap. This reduces the worst-case time complexity from $\Theta(n\log(n))$ to $\Theta(n\log(k))$, since the priority queue only has size k instead of n. One implementation of this solution is shown below:

```cpp
int32_t find_kth_largest(const std::vector<int32_t>& nums, int32_t k) {
  std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>>
    pq(nums.begin(), nums.begin() + k);

  for (int32_t i = k; i < nums.size(); ++i) {
    if (nums[i] > pq.top()) {
      pq.pop();
      pq.push(nums[i]);
    } // if
  } // for i

  return pq.top();
} // find_kth_largest()
```

39. To minimize the number of dynamite sticks we need to remove all the rubble, we will want to use the most powerful dynamite first. This can be done by placing the available dynamite into a priority queue. However, since we are only allowed to use at most k sticks of dynamite, we can use the same strategy as in the previous problem and instead use a min-priority queue to keep track of the k most powerful dynamite sticks we have encountered so far. An implementation of this solution is shown below:

```cpp
int32_t min_tries_to_escape(const std::vector<int32_t>& dynamite, int32_t rubble_size, int32_t k) {
  std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>>
    pq(dynamite.begin(), dynamite.begin() + k);

  for (int32_t i = k; i < dynamite.size(); ++i) {
    if (dynamite[i] > pq.top()) {
      pq.pop();
      pq.push(dynamite[i]);
    } // if
  } // for i

  int32_t num_sticks_used = 0;
  while (rubble_size > 0 && !pq.empty()) {
    rubble_size -= pq.top();
    pq.pop();
    ++num_sticks_used;
  } // while

  return rubble_size > 0 ? -1 : num_sticks_used;
} // find_kth_largest()
```

40. This is another variation of the "k closest" problem that was demonstrated in the previous two problems. The simplest solution is to calculate the distance between every point and the origin and sort all the points by this distance, to retrieve the k closest points. However, this runs in $\Theta(n\log(n))$ time if we are given n points. We can improve on this solution by keeping track of a max-heap of size k. For each point we encounter, we add it to the max-heap if its distance is lower than the distance of the point at the top of the heap (extracting the point originally at the top of the heap in the process) — this ensures that our max-heap will always store the k closest points encountered so far. Once we finish looking at all the input points, we can just return the contents of the max-heap as our solution. An implementation of this solution is shown below:

```cpp
struct PointWithDistance {
  int32_t x;
  int32_t y;
  double distance;

  PointWithDistance(int32_t x_in, int32_t y_in) : x{x_in}, y{y_in} {
    // potential optimization: sqrt is not actually necessary since you are comparing relative
    // distances, since a larger (x^2 + y^2) also implies a larger std::sqrt(x^2 + y^2)
    distance = std::sqrt(x * x + y * y);
  } // PointWithDistance()

  bool operator<(const PointWithDistance& other) const {
    return distance < other.distance;
  } // operator<()
};

std::vector<std::vector<int32_t>> k_closest_to_origin(
    const std::vector<std::vector<int32_t>>& points, int32_t k) {
  std::priority_queue<PointWithDistance> pq;
  for (const auto& point : points) {
    pq.emplace(point[0], point[1]);
    if (pq.size() > k) {
      pq.pop();
    } // if
  } // for point

  std::vector<std::vector<int32_t>> solution;
  solution.reserve(k);

  while (!pq.empty()) {
    const auto& next = pq.top();
    solution.push_back({next.x, next.y});
    pq.pop();
  } // while

  return solution;
} // k_closest_to_origin()
```

41. The idea behind this problem is the same as the previous problem. To avoid the $\Theta(n\log(n))$ time complexity of sorting all applicants by qualification score, we will instead of use a min-heap of size k to keep track of the k most qualified applicants encountered so far. An implementation of this solution is shown below (this example uses a custom comparator, but you can overload `operator<` as well, as in the previous problem):

```cpp
struct Applicant {
  std::string uniqname;
  int32_t application_id;
};

// This method is already implemented for you - it returns a number
// in the range [0, 100] that identifies how qualified an applicant is
int32_t get_qualification(const Applicant& applicant);

struct QualificationData {
  Applicant applicant;
  int32_t qualification;

  explicit QualificationData(const Applicant& applicant_in) : applicant{applicant_in} {
    qualification = get_qualification(applicant);
  } // QualificationData()
};

struct ApplicantCompare {
  bool operator() (const QualificationData& lhs, const QualificationData& rhs) {
    if (lhs.qualification == rhs.qualification) {
      return lhs.applicant.application_id < rhs.applicant.application_id;
    } // if

    return lhs.qualification > rhs.qualification;
  } // operator()()
};

std::vector<Applicant> hire_staff(const std::vector<Applicant>& applicants, int32_t k) {
  std::priority_queue<QualificationData, std::vector<QualificationData>, ApplicantCompare> pq;
  for (const auto& applicant : applicants) {
    pq.emplace(applicant);
    if (pq.size() > k) {
      pq.pop();
    } // if
  } // for applicant

  std::vector<Applicant> new_staff;
  new_staff.reserve(k);

  while (!pq.empty()) {
    const auto& next = pq.top();
    new_staff.push_back(next.applicant);
    pq.pop();
  } // while

  return new_staff;
} // hire_staff()
```