Q1.

a) False. If the array is equally partitioned every time the partition is called, quicksort performs almost the same number of operations as mergesort; however, unlike mergesort, quicksort doesn't need to copy elements in the temporary array.
No partial marking. Give marks only if the justification is correct.

b) False. Because there is no order, we need to access all elements in the worst case to search for an element. Therefore, the worst-case time complexity is O(n).
No partial marking. Give marks only if the justification is correct.

Q2. The outermost loop executes n times, the middle loop executes log(n) time, and the innermost loop executes n times; therefore, the time complexity is **$O(n^2 * log(n))$.**
No partial marking. Give marks only if the justification is correct.

Q3. A different valid answer is also possible**. Give up to 1 mark if the solution is partially correct.**

```
int foo(int val) {
  while (1) {
    if (val <= 0) {
      val = val + 1;
      continue;
    }
    if ((val % 2) != 0) {
      val = (val * 2) + 1;
      continue;
    }
    if (val > 100000) {
      val = val + 1;
      continue;
    }
    break;
  }
}
```

Q5. The algorithm for reversing a list is shown below. Reversing both singly or doubly linked lists is acceptable. **Give up to one mark if the solution is partially correct.**

Reversal of a singly linked list..

```
struct node {
  int val;    // it could be any other type as well
  struct node *next;
};
Algorithm reverse_list(head)
  // head is the head of linked list of type struct node
  // Output: return the head of the reversed list
  prev = NULL
  cur = head

  while cur != NULL
    next = cur->next
    cur->next = prev
    prev = cur
    cur = next
  return prev
```

Reversal of a doubly linked list is shown on the next page.

Reversal of doubly linked lists is also fine. In that case, the algorithm works as follows.

```
struct node {
  int val;    // it could be any other type as well
  struct node *next;
  struct node *prev;
};
Algorithm reverse_list(head)
// head is the head of linked list of type struct node
// return value is the head of the reversed list
  cur = head
  prev = NULL

  while cur != NULL
    next = cur->next
    prev = cur->prev
    cur->next = prev
    cur->prev = next
    prev = cur
    cur = next
  return prev
```

Q4. The algorithm maintains the head and tail of 64 queues, one queue for each priority, in an array. The queues are implemented using a linked list. **Give at most one mark if the queue was implemented using an array because, in that case, the insertion and deletion will not be O(1) when the array is full.** The insertion takes place at the rear end of the linked list, and the deletion is done from the front. If this is not the case, then the time complexity will not be O(1). **Give max one mark if the insertion and deletion are not O(1) operations. Deduct up to one mark for minor mistakes.**

```
struct node {
  int val;
  struct node *next;
};
struct queue {
  struct node *head;
  struct node *tail;
};
struct queue *arr[65] = {NULL}
// arr is a global array of queues

Algorithm insert(val, priority)
// val is the input value
// priority is the priority
// Output: insert (val, priority) in PQueue
  n = malloc(sizeof(struct node))
  n->val = val
  n->next = NULL
  if arr[priority].tail == NULL
    arr[priority].tail = n
    arr[priority].head = n
  else
    arr[priority].tail->next = n
    arr[priority].tail = n

Algorithm delete()
// Output: delete an element with the highest priority
// If multiple elements have the same priority then
// delete the element that came first
// return the request_id of the deleted element
  i = 64
  while i > 0
    if arr[priority].head != NULL
      n = arr[priority].head
      arr[priority].head = n->next
      if arr[priority].head == NULL
        arr[priority].tail = NULL
      return n->request_id
    i = i - 1
  perror("pqueue empty")
```

Q7. We can delete the minimum element in a single pass or two passes, as shown on the next page. In the single pass, we recursively pop the elements from the stack and keep track of the minimum until we reach the end of the list. While returning from the recursive call, we push the values back if they are not minimum values. In the two-pass algorithm, in the first pass, we find the minimum value. In the second pass, we delete the minimum value. **Give a maximum mark of up to two if the solution is partially correct. Give zero marks if more than O(1) additional space is used.**

One Pass Algorithm

```
Algorithm find_and_delete_min(S, min)  // One-pass algorithm
// S is the stack
// min is the minimum value on the stack so far
  if stack_empty(S)
    return min
  v = pop(S)
  if v < min
    min = v
  min = find_and_delete_min(S, min)
  if min != v
    push(S, v)
  return min

Algorithm delete_min(S)
// S is the input stack
// output: delete and return the minimum value
  min = pop(S)
  push(S, min)
  return find_and_delete_min(S, min)
```

Two pass algorithm

```
Algorithm find_min(S, min) // two-pass algorithm
// S is the stack
// min is the minimum value on the stack so far
  if stack_empty(S)
    return min
  v = pop(S)
  if v < min
    min = v
  min = find_min(S, min)
  push(S, v)
  return min
```

```
Algorithm delete_minimum(S, min)
// S is the stack and min is the minimum value on the stack
// output: remove min from the stack
  v = pop(S)
  if (min != v)
    delete_minimum(S, min)
    push(S, v)

Algorithm delete_min(S)
// S is the input stack
// output: delete and return the minimum value
  min = pop(S)
  push(S, min)
  min = find_min(S, min)
  delete_minimum(S, min)
  return min
```

Q6. **The average time-complexity of the randomized quicksort algorithm is O(n * log(n)), whereas the worst case time-complexity is O(n^2). Give one mark for the correct algorithm and the correct time complexities. Give four marks for the correct derivation. There could be other valid answers too. Forward those answers to me in case of any doubts.**

```
Algorithm partition(arr, lo, hi)
// arr is the input array
// lo is lowest accessible index in the array
// hi is the highest accessible in the array
// output: randomly pick a value v from the arr[lo:hi],
// find the final position of v, say i, in the sorted array
// store v at index i and return i

  r = get_random(lo, hi)
  // get_random generates a random value between lo and hi
  pivot = arr[r]
  arr[r] = arr[0]
  arr[0] = pivot
  left = lo + 1;
  right = hi;

  while left <= right

    while left <= right and arr[left] < pivot
      left = left + 1
    while right >= left and arr[right] > pivot
      right = right - 1

    if left <= right
      exchange(arr, left, right)
      // exchange swap the value at indices left and right
      left = left + 1
      right = right - 1

  exchange(arr, lo, right)
  return right
```

Worst case complexity: T(1) = c1, T(n) = T(n-1) + c * n

= T(n-2) + c (n + (n-1)) // after first expansion

= T(n-3) + c (n + (n-1) + (n-2)) // after second expansion

= T(n-4) + c (n + (n-1) + (n-2) + (n-3) // after third expansion

= T(n – k – 1) + c (n + (n-1) + (n-2) + … + (n-k)) // after kth expansion

Substituting n-k-1 = 1

T(n) = T(1) + c(2 + 3 + .. + n) = c1 + c * $\left(\frac{n(n+1)}{2} - 1\right)$ = O(n^2) **// one mark for correct derivation**

Average time complexity: Time complexity, when the portioning algorithm returns i

$T(n) = T(i) + T(n-1-i) + c_1 n + c_2$

If each index has the same probability of being selected as a target position for the pivot, then the average complexity is

$$\mathrm{T(n)} = \frac{1}{n}\left(\sum_{i=0}^{n-1}(T(i) + T(n-1-i))\right) + \mathrm{c_1 n} + \mathrm{c_2}$$

$$\mathrm{T(n)} = \frac{1}{n}\left(\sum_{i=0}^{n-1}(T(i) + T(n-1-i))\right) + (\mathrm{c_1 n} + \mathrm{c_2})$$

$\mathrm{T(n)} = \frac{2}{n}\left(\sum_{i=0}^{n-1}(T(i))\right) + (\mathrm{c_1 n} + \mathrm{c_2})$ (Full history recurrence relation)

$$\mathrm{nT(n)} = 2\left(\sum_{i=0}^{n-1}(T(i))\right) + \mathrm{c_1 n^2} + \mathrm{c_2 n}$$

$$(\mathrm{n}-1)\mathrm{T(n}-1) = 2\left(\sum_{i=0}^{n-2}(T(i))\right) + \mathrm{c_1}(\mathrm{n}-1)^2 + \mathrm{c_2}(\mathrm{n}-1)$$

$\mathrm{nT(n)} - (\mathrm{n}-1)T(n-1) = 2T(n-1) + c_1(2n-1) + c_2$

$T(n) = \frac{n+1}{n}T(n-1) + 2c_1 + \frac{c_2 - c_1}{n} \leq \frac{n+1}{n}T(n-1) + c$

$$T(n) \leq \frac{n+1}{n}T(n-1) + c$$

$$= \frac{n+1}{n}\left(\frac{n}{n-1}T(n-2) + c\right) + c = \frac{n+1}{n-1}T(n-2) + c\left(1 + \frac{n+1}{n}\right)$$

$$= \frac{n+1}{n-1}\left(\frac{n-1}{n-2}T(n-3) + c\right) + c\left(1 + \frac{n+1}{n}\right) = \frac{n+1}{n-2}T(n-3) + c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1}\right)$$

$= \ \ldots$

$$= \frac{n+1}{n-k+1}T(n-k) + c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \ \ldots + \frac{n+1}{n-k+2}\right)$$

Substituting k = n – 1

$T(n) \leq \frac{n+1}{2}T(1) + c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \ \ldots + \frac{n+1}{3}\right)$

$$= \frac{n+1}{2}c_3 + c(n+1)\left(1 + \frac{1}{2} + \frac{1}{3} + .. + \frac{1}{n+1} - \frac{3}{2}\right)$$

Because, $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ $is\ O(\log n)$

$= \frac{n+1}{2}c_3 + c(n+1)\ (c_4(\log(n+1)) - \frac{3}{2}) \ = O(n \log n)$

// Correct derivation of average case: 3 marks
// Partially correct derivation of average case: 1 mark

Q8.

Pre-order: 8 6 3 0 5 17 11 13 **// 1.5 marks, no partial marking**

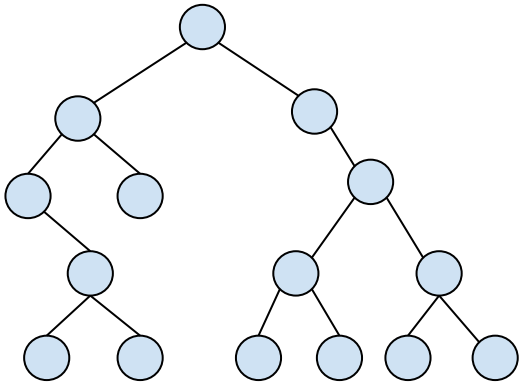
Post-order: 0 5 3 6 13 11 17 8 **// 1.5 marks, no partial marking**

Q9.

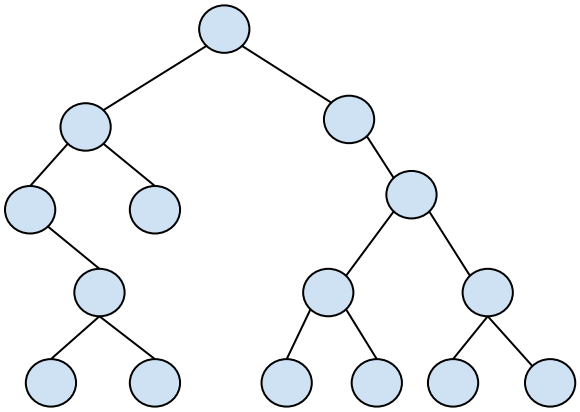
Deleting-7 **// 1.5 marks no partial marking**

Approach-1

Delete maximum value in the left of 7

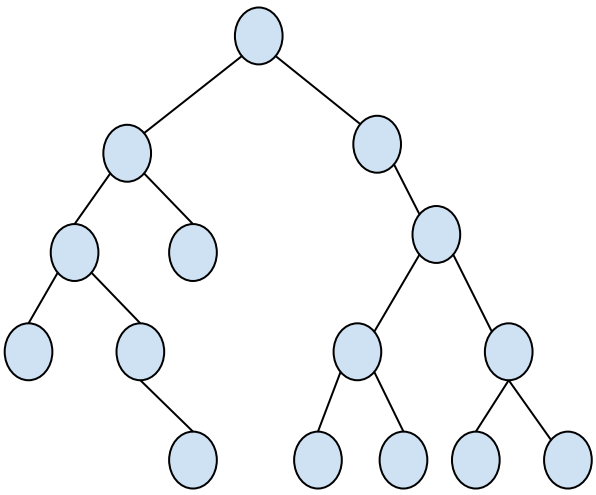


Copy the value of the deleted node to 7

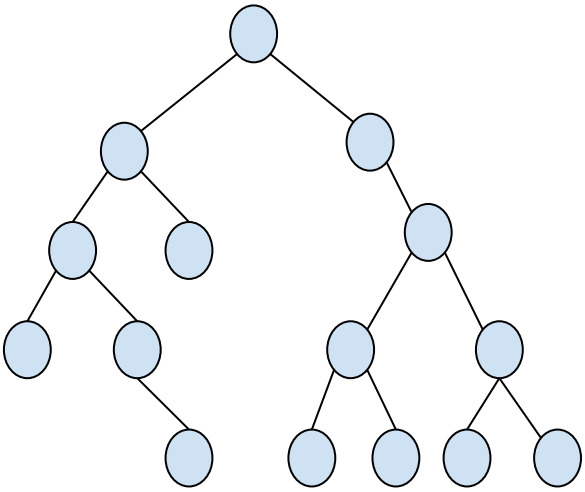


Approach-2

Delete minimum node in the right of 7



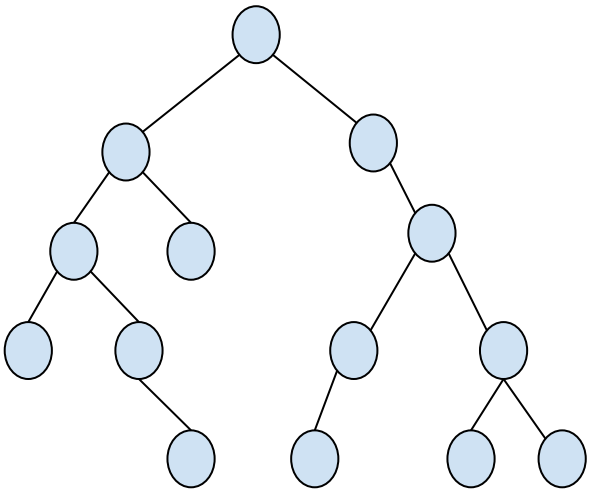
Copy the value of the deleted node to 7



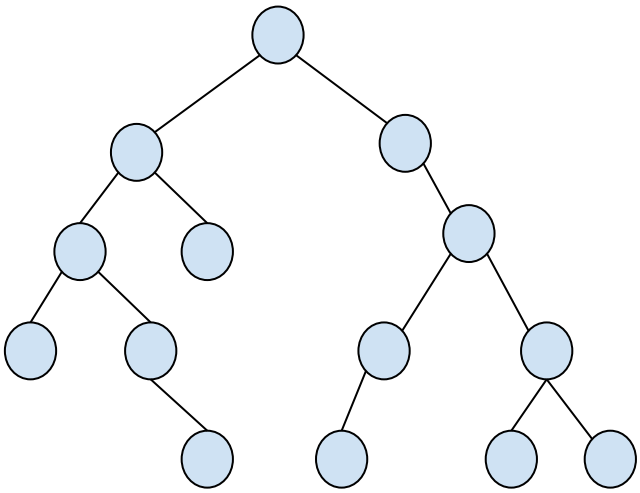
Deleting 33. **// 1.5 marks no partial marking**

Approach-1

Delete maximum value in the left of 33

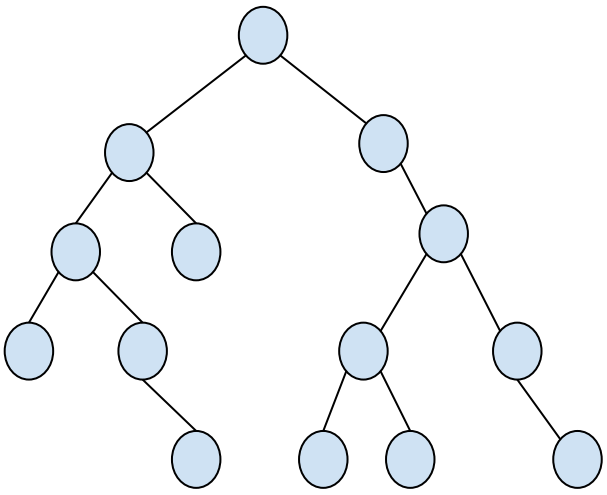


Copy the value of the deleted node to 33

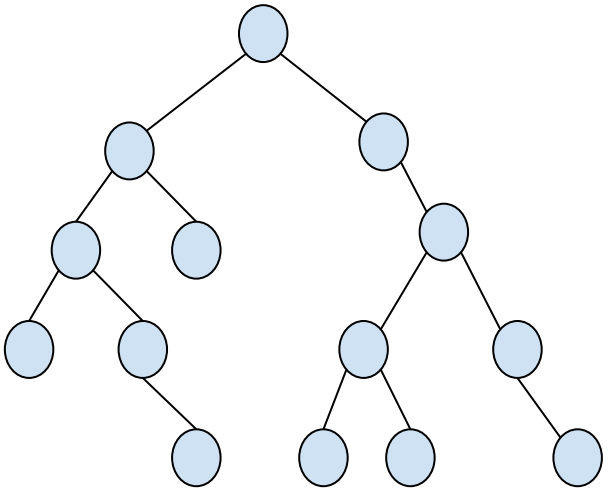


Approach-2:

Delete minimum node in the right of 33



Copy the value of the deleted node to 33



Q10. Give up to one mark for a partially correct algorithm.

```
Algorithm partition(arr, lo, hi)
// arr is the input array
// lo is lowest accessible index in the array
// hi is the highest accessible in the array
// output: randomly pick a value v from the arr[lo:hi],
// find the final position of v, say i, in the sorted array
// store v at index i and return i

  pivot = arr[0]
  left = lo + 1;
  right = hi;

  while left <= right

    while left <= right and arr[left] < pivot
      left = left + 1
    while right >= left and arr[right] > pivot
      right = right - 1

    if left <= right
      exchange(arr, left, right)
      // exchange swap the value at indices left and right
      left = left + 1
      right = right - 1

  exchange(arr, lo, right)
  return right

Algorithm quicksort(arr, lo, hi)
// input arr is the input array
// lo is lowest accessible index in the array
// hi is the highest accessible in the array
// Output: sorted arr[lo:hi]

  if lo >= hi
    return;
  p = partition(arr, lo, hi)
  quicksort(arr, lo, p - 1)
  quicksort(arr, p + 1, hi)
```

after first partition	2 3 1 4 10 16 8 9 11	**// 1 mark**
after second partition	1 2 3 4 10 16 8 9 11	**// 1 mark**
after third partition	1 2 3 4 8 9 10 16 11	**// 1 mark**
after fourth partition	1 2 3 4 8 9 10 16 11	**// 0.5 mark**
after fifth partition	1 2 3 4 8 9 10 11 16	**// 0.5 mark**

Q11. One mark for a partially correct algorithm.

```
Algorithm merge(arr, lo, mid, hi)
// arr is the input array. arr[lo:mid] and arr[mid+1:hi] are already sorted
// output: sorted arr[lo:hi]
  n_elem_1 = mid - lo + 1;
  n_elem_2 = hi - mid;

  tmp1[n_elem_1 + 1] // allocating array
  tmp2[n_elem_2 + 1]

  for i in 0 to n_elem_1 - 1
    tmp1[i] = arr[lo + i]

  for j in 0 to n_elem_2 - 1
    tmp2[j] = arr[mid + 1 + j]

  tmp1[i] = tmp2[j] = INT_MAX   // INT_MAX is maximum possible integer value

  i = j = 0
  for k in lo to hi
    if tmp1[i] <= tmp2[j]
      arr[k] = tmp1[i]
      i = i + 1
    else
      arr[k] = tmp2[j]
      j = j + 1

Algorithm mergesort(arr, lo, hi)
// input arr is the input array
// lo is lowest accessible index in the array
// hi is the highest accessible in the array
// Output: sorted arr[lo:hi]
  if (lo < hi)
    mid = floor((lo + hi) / 2) // floor ignores value after the decimal point
    mergesort(arr, lo, mid)
    mergesort(arr, mid+1, hi)
    merge(arr, lo, mid, hi)
```

After first merge	4 9 8 10 2 16 1 3 11	**// 0.4 marks**
After second merge	4 8 9 10 2 16 1 3 11	**// 0.4 marks**
After third merge	4 8 9 2 10 16 1 3 11	**// 0.4 marks**
After fourth merge	2 4 8 9 10 16 1 3 11	**// 0.4 marks**
After fifth merge	2 4 8 9 10 1 16 3 11	**// 0.4 marks**
After sixth merge	2 4 8 9 10 1 16 3 11	**// 0.4 marks**
After seventh merge	2 4 8 9 10 1 3 11 16	**// 0.3 marks**
After eighth merge	1 2 3 4 8 9 10 11 16	**// 0.3 marks**

Q12.

$T(n)$

$= 5T(n-8) + 100$

$= 5(5T(n-16) + 100) + 100$ ***Give total one mark if this step is correct***

$= 5^2T(n-8*2) + 100(1+5)$ *After first expansion*

$= 5^2(5T(n-24) + 100) + 100(1+5)$ ***Give total two mark if this step is correct***

$= 5^3T(n-8*3) + 100(1+5+5^2)$ *After second expansion*

$= 5^3(T(n-32) + 100) + 100(1+5+5^2)$

$= 5^4T(n-8*4) + 100(1+5+5^2+5^3)$ *After third expansion*

$= 5^{k+1}T\big(n-8*(k+1)\big) + 100\big(1+5+5^2+\cdots+5^k\big)$ *After kth expansion*

Give total three mark if the previous step is correct

Substituting $n-8k-8=0$, *when* $n \geq 8$

$$k = \frac{n-8}{8}$$

$$T(n) = 5^{\frac{n}{8}} + 100(5^{\frac{n}{8}} - 1)$$

Q13. To find the median, we need to first compute the total number of elements followed by deleting n/2 minimum elements. Notice that the additional space is not allowed, so the minimum values can't be copied in some additional array. After deleting the elements, the next minimum is the median. **Give zero marks if more than O(1) space is used. Give up to two marks if the algorithm is partially correct.**

```
struct node {
  int val; struct node *next;
};
Algorithm count(head)
// head is the head of a linked list
// Output: return the number of elements in the linked list
  ret = 0
  while head != NULL
    ret = ret + 1
    head = head->next
  return ret

Algorithm find_min(head)
// head is the head of a linked list
// Output: return the address of the minimum element in the linked list
  min = head
  while head != NULL
     if head->val < min->val
        min = head
     head = head->next
  return min

Algorithm delete_min(head, min)
// head is the head of a linked list
// min is the address of the node we want to delete
// Output: returns the new head after deleting min from the list
  if min == head
    return head->next
  temp = head
  while temp->next != min
    temp = temp->next
  temp->next = min->next
  return head

Algorithm find_median(head)
  // head is the head of a linked list
  // Output: returns the median of the values stored in the list
  n = count(head)
  n = floor(n / 2) // floor ignores the value after the decimal point
  for i in 0 to n
     min = find_min(head)
     head = delete_min(head)
  return find_min(head)
```

Q14. We can find the minimum in two ways. There could be other ways as well. In the first algorithm, we recursively find the minimum of leaf subtree x and right subtree y. Then we find the minimum among root, x, and y and return it to the parent. In the second algorithm, we use a pointer variable to store the minimum value. We can use a general tree traversing algorithm and update the minimum value stored in the pointer variable.
Give a maximum of one mark if the algorithm is for a binary search tree. Give up to two marks if the algorithm is partially correct. Give zero if more than O(1) additional space is used.

Algorithm-1

```
struct node {
  int val;
  struct node *left;
  struct node *right;
};
Algorithm find_min(root)  // first method
// root is the root of a binary tree of type struct node
// Output: return the address of the node that contains the minimum value
   if root == NULL
     return NULL
   min1 = find_min(root->left);
   min2 = find_min(root->right);
   ret = root;
   if min1 and min1->val < ret->val
     ret = min1
   if min2 and min2->val < ret->val
     ret = min2
   return ret
```

Algorithm-2

```
struct node {
  int val;
  struct node *left;
  struct node *right;
};
Algorithm find_min(root, min)  // second method
// root is the root of a binary tree of type struct node
// min is a pointer to an integer
// initially *min contains the value in the root
// Output: return the minimum value in *min
  if (root == NULL)
    return;
  if root->val < *min
    *min = root->val
  find_min(root->left, min)
  find_min(root->right, min)
```