# WRAP-UP

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

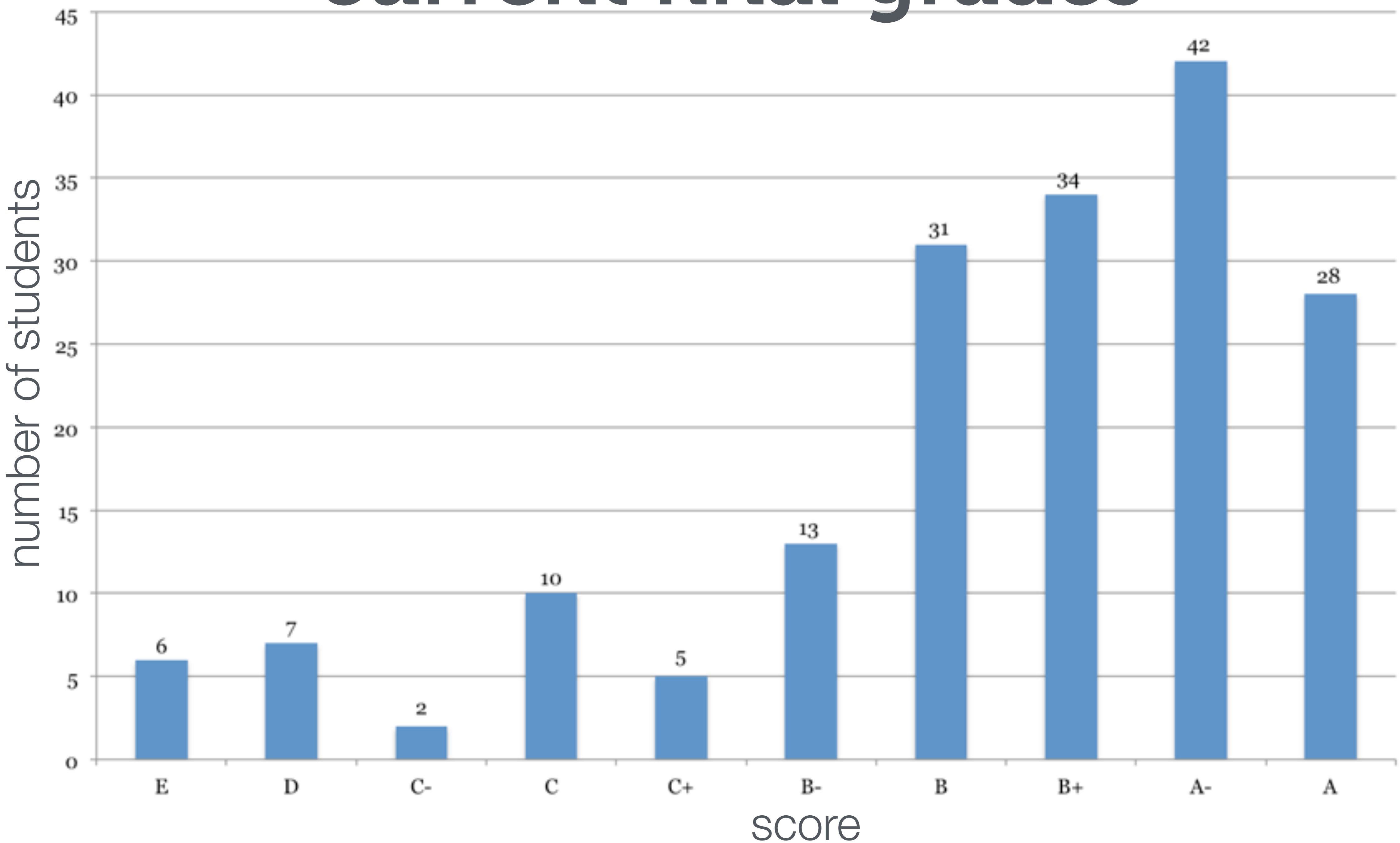
# administration…

-Ryan’s Review tonight!

-office hours today

-final exam on Thursday, 10:30am

# current final grades




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 89-87 | B+ | 79-77 | C+ | 69-67 | D+ | | |
| 100-93 | A | 86-83 | B | 76-73 | C | 66-63 | D | 59-0 | E |
| 92-90 | A- | 82-80 | B- | 72-70 | C- | 62-60 | D- | | |

# topics

- Java basics
  - variables, types
  - control flow
  - reference types
  - classes, methods

- OOP
  - inheritance
  - polymorphism
  - interfaces

- generics
  - wild cards
  - generic classes
  - generic static methods

- comparators

- primitive type wrappers

- algorithm analysis
    - growth rates
    - Big-O
    - determining the complexity of algorithms

- sorting
  - selection sort
  - insertion sort
  - Shell sort
  - mergesort
  - quicksort

-you must be able to demonstrate that you understand why these algorithms perform as they do

- recursion
    - writing recursive methods
    - reading recursive methods
    - base cases
    - when to use recursion vs iteration

-linked structures
  - -memory allocation
  - -array vs linked structures
  - -access arrays vs linked structures

-linked lists
  - -implementation
  - -performance

- stacks
    - implementation
        - *array and linked list versions*
    - performance

- queues
    - implementation
        - *array and linked list versions*
    - performance

- trees
    - traversals
        - *pre-, in-, and post-orders*
    - reasoning about:
        - *depth of nodes*
        - *number and type (leaf, inner, root)*
        - *balance*

- you should know the performance characteristics and usefulness of any data structure we have studied

- binary search trees
    - importance of a balanced BST
    - what cases balance/unbalance
    - insertion, deletion, search

- graphs
    - uses of graphs
    - DFS
    - BFS
    - Dijkstra's algorithm

- hash tables
  - linear probing
  - quadratic probing
  - separate chaining
  - clustering
  - load factor
  - performance (time and space)

- hash functions
  - what makes a good hash functions?
  - what rules must a hash function obey?

- complete binary trees
  - reasoning about height
  - how many nodes does each level contain?
  - representing complete trees as an array

- binary heaps (min, max, and min-max)
    - structure property (complete)
    - order property (min, max, and min-max)
    - performance of all operations
        - *deleteMin() and deleteMax()*
        - *add()*
        - *percolate up*
        - *percolate down*
        - *build-heap operation*

- binary
    - base two number system
    - bits
    - bytes

- hexadecimal
    - base sixteen number system
    - bytes in hex

- converting between bases 2, 10, and 16

- ASCII
    - you don’t need to memorize the table!

- Huffman compression
    - binary tries
    - constructing a Huffman tree
    - compression
    - decompression
    - tie-breaking
        - *and why it’s necessary!*
    - encoding tree reconstruction information

# questions

# finding the maximum item in an array

## ALGORITHM?

1) initialize `max` to the first element

2) scan through each item in the array
   - if the item is greater than `max`, update `max`

## WHAT IS THE BIG-O COMPLEXITY OF THIS ALGORITHM?

1) **c**
2) **log N**
3) **N**
4) **N log N**
5) **$N^2$**
6) **$N^3$**

# finding the smallest difference

## ALGORITHM?

```
diff = MAX_INTEGER;
for(int i=0; i<array.length-1; i++)
{
  num1 = array[i];
  for(int j=i+1; j<array.length; j++)
  {
    num2 = array[j];
    if (abs(num1-num2) < diff)
      diff = abs(num1-num2);
  }
}
return diff;
```

1) **c**
2) **log N**
3) **N**
4) **N log N**
5) **$N^2$**
6) **$N^3$**

WHAT IS THE BIG-O COMPLEXITY OF THIS ALGORITHM?

# analyze the running time

```
for(int i=0; i<n; i+=2)
  sum++;
```

```
for(int i=0; i<n; i++)
  for(int j=0; j<n*n; j++)
    sum++
```

```
for(int i=0; i<n; i*=2)
  sum++;
```

1) **c**
2) **log N**
3) **N**
4) **N log N**
5) **$N^2$**
6) **$N^3$**

# selection sort

# selection sort

1) find the minimum item in the unsorted part of the array
2) swap it with the first item in the unsorted part of the array
3) repeat steps 1 and 2 to sort the remainder of the array

WHAT DOES THIS LOOK LIKE?

## WHAT IS THE BEST-CASE COMPLEXITY OF SELECTION SORT?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **$N^2$**
F) **$N^3$**

# insertion sort

good for small **N**

# insertion sort

1) the first array item is the sorted portion of the array
2) take the second item and insert it in the sorted portion
3) repeat steps 1 and 2 to sort the remainder of the array

WHAT DOES THIS LOOK LIKE?

# worst case scenario…

-what are the number of inversions in the worst case?

-what ***IS*** the worst case?

| | | | | | | |
|---|---|---|---|---|---|---|
| 76 | 45 | 11 | 9 | 0 | -3 | -8 |

— INVERTED

HOW MANY INVERSIONS ARE THERE?

-when every **unique pair** is inverted…

-how many unique pairs are there?

-(hint: remember Gauss's trick!)

$$N * (N-1)/2 = (N^2 - N)/2$$

# insertion sort is O(N+I)

WHAT IS THE WORST-CASE COMPLEXITY OF INSERTION SORT?

A) c
B) log N
C) N
D) N log N
E) $N^2$
F) $N^3$

# insertion sort is O(N+I)

WHAT IS THE BEST-CASE COMPLEXITY OF INSERTION SORT?

A) c
B) log N
C) N
D) N log N
E) $N^2$
F) $N^3$

# selection vs insertion

| | selection | insertion |
|---|---|---|
| WORST: | $O(N^2)$ | $O(N^2)$ |
| AVERAGE: | $O(N^2)$ | $O(N^2)$ |
| BEST: | $O(N^2)$ | $O(N)$ |

WHICH ONE PERFORMS BETTER IN PRACTICE?

A) **selection**
B) **insertion**

# shellsort

the simplest subquadratic sorting algorithm

# shellsort

insertion sort, with a twist

1) set the **gap size** to **N/2**

2) consider the subarrays with elements at **gap size** from each other

3) do insertion sort on each of the subarrays

4) divide the **gap size** by 2

5) repeat steps 2 — 4 until the is **gap size** is <1

WHAT DOES THIS LOOK LIKE?

HOW DO WE DESCRIBE **INSERTION SORT** WITH RESPECT TO **SHELLSORT**?

# exercise 1

-how to compute **N!**

**N! = N * N-1 * N-2 * ... * 2 * 1**

-how would you compute this using a for-loop?

-how would you compute this using recursion?

-think about:

*-what is the base case?*

*-what is recursive?*

# exercise 1

-how to compute **N!**

**$N! = N * N-1 * N-2 * \ldots * 2 * 1$**

-how would you compute this using a for-loop?

-how would you compute this using recursion?

-think about:

*-what is the base case?*

*-what is recursive?*

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **$N^2$**
F) **$N^3$**

WHAT IS THE COMPLEXITY OF THE FOR-LOOP METHOD?

# exercise 1

-how to compute **N!**

**N! = N * N-1 * N-2 * ... * 2 * 1**

-how would you compute this using a for-loop?

-how would you compute this using recursion?

-think about:

*-what is the base case?*

*-what is recursive?*

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **$N^2$**
F) **$N^3$**

WHAT IS THE COMPLEXITY OF THE RECURSIVE METHOD?

# mergesort

divide and conquer

# mergesort

1) divide the array in half

2) ~~sort the left half~~

3) ~~sort the right half~~

2) take the left half, and go back to step 1 UNTIL???

3) take the right half, and go back to step 1 UNTIL???

4) merge the two halves together

WHAT DOES THIS LOOK LIKE?

```
void mergesort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int mid = (left + right) / 2;
  mergsort(arr, left, mid);
  mergsort(arr, mid+1, right);
  merge(arr, left, mid+1, right);
}
```

DIVIDE

CONQUER

# WHAT IS THE COMPLEXITY OF MERGESORT?

IS THIS THE WORST || AVERAGE || BEST-CASE?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **$N^2$**
F) **$N^3$**

# quicksort

another divide and conquer

# quicksort

1) select an item in the array to be the ***pivot***

2) ***partition*** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) sort the left half

4) sort the right half

3) take the left half, and go back to step 1 UNTIL???

4) take the right half, and go back to step 1 UNTIL???

WHAT DOES THIS LOOK LIKE?

# in-place partitioning

1) select an item in the array to be the ***pivot***
2) swap the pivot with the last item in the array *(just get it out of the way)*
3) step from left to right until we find an item > pivot
   *-this item needs to be on the **right** of the partition*
4) step from right to left until we find an item < pivot
   *-this item needs to be on the **left** of the partition*
5) swap items
6) continue until left and right stepping cross
7) swap pivot with left stepping item

WHAT DOES THIS LOOK LIKE?

# quicksort complexity

- performance of quick sort heavily depends on which array item is chosen as the pivot

- **best case:** pivot partions the array into two equally-sized subarrays *at each stage* — **O(N log N)**

- **worst case:** partition generates an empty subarray *at each stage* — **$O(N^2)$**

- **average case:** bound is **O(N log N)**
  - proof is quite involved, see the textbook if you are curious

# Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | O(n log(n)) | O(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(n) |
| Timsort | O(n) | O(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Shell Sort | O(n) | O((nlog(n))^2) | O((nlog(n))^2) | O(1) |
| Bucket Sort | O(n+k) | O(n+k) | O(n^2) | O(n) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n+k) |

# **linked list** vs **array**

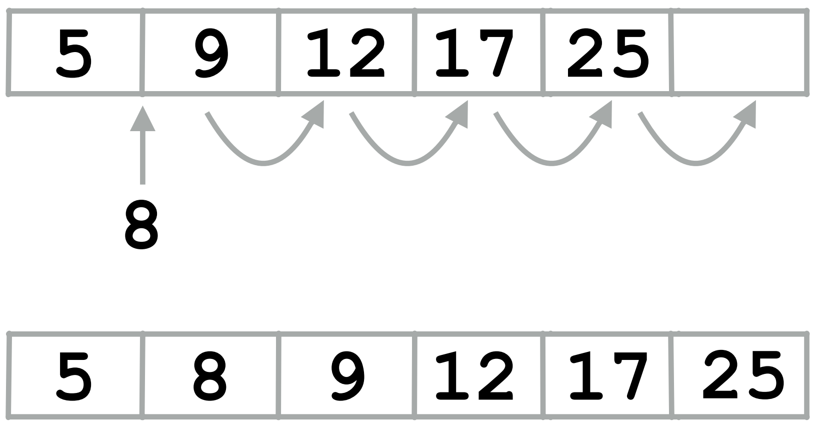
-cost of accessing a random item at location `i`?

-cost of `removeFirst()`?

-cost of `addFirst()`?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) $N^2$
F) $N^3$

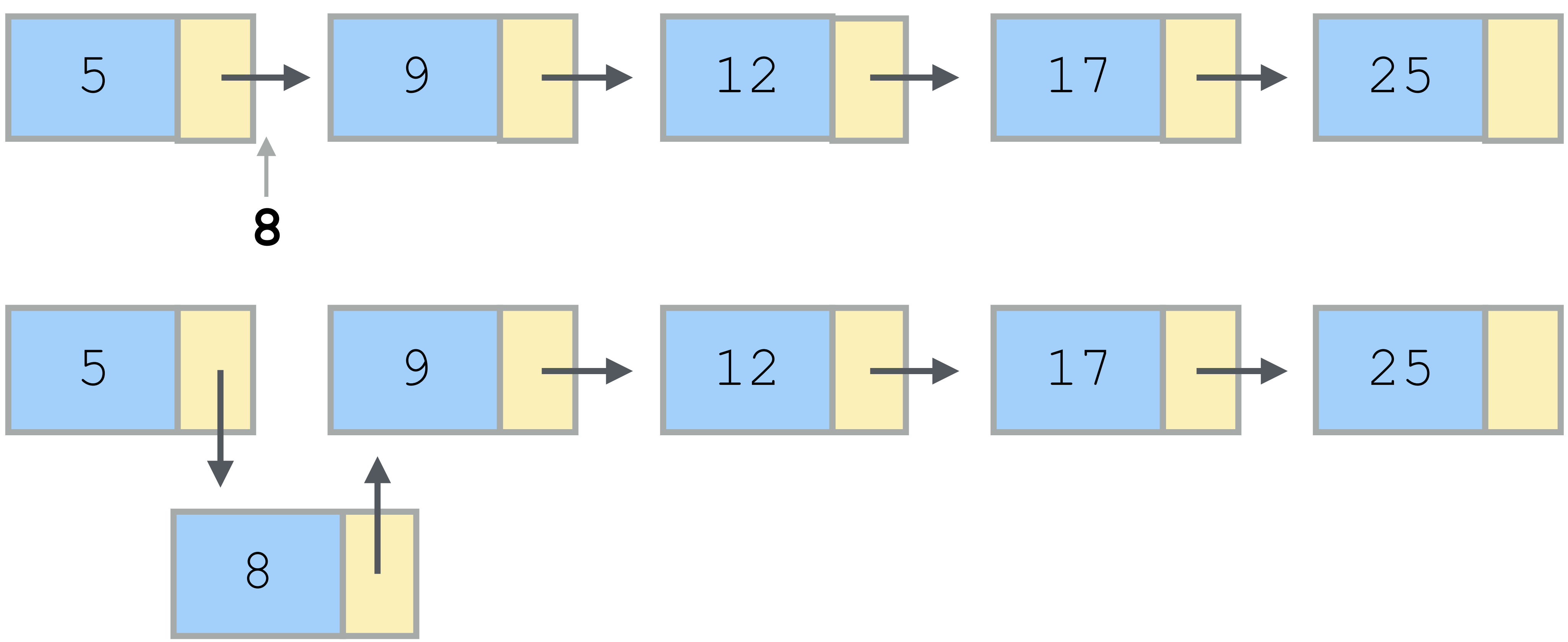
inserting into an array:




## WHAT IS THE COST OF INSERTION?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) $\mathbf{N^2}$
F) $\mathbf{N^3}$

inserting into a linked list:



The root is ___.

The height is ___.

The parent of **`v3`** is ___.

The depth of **`v3`** is ___.

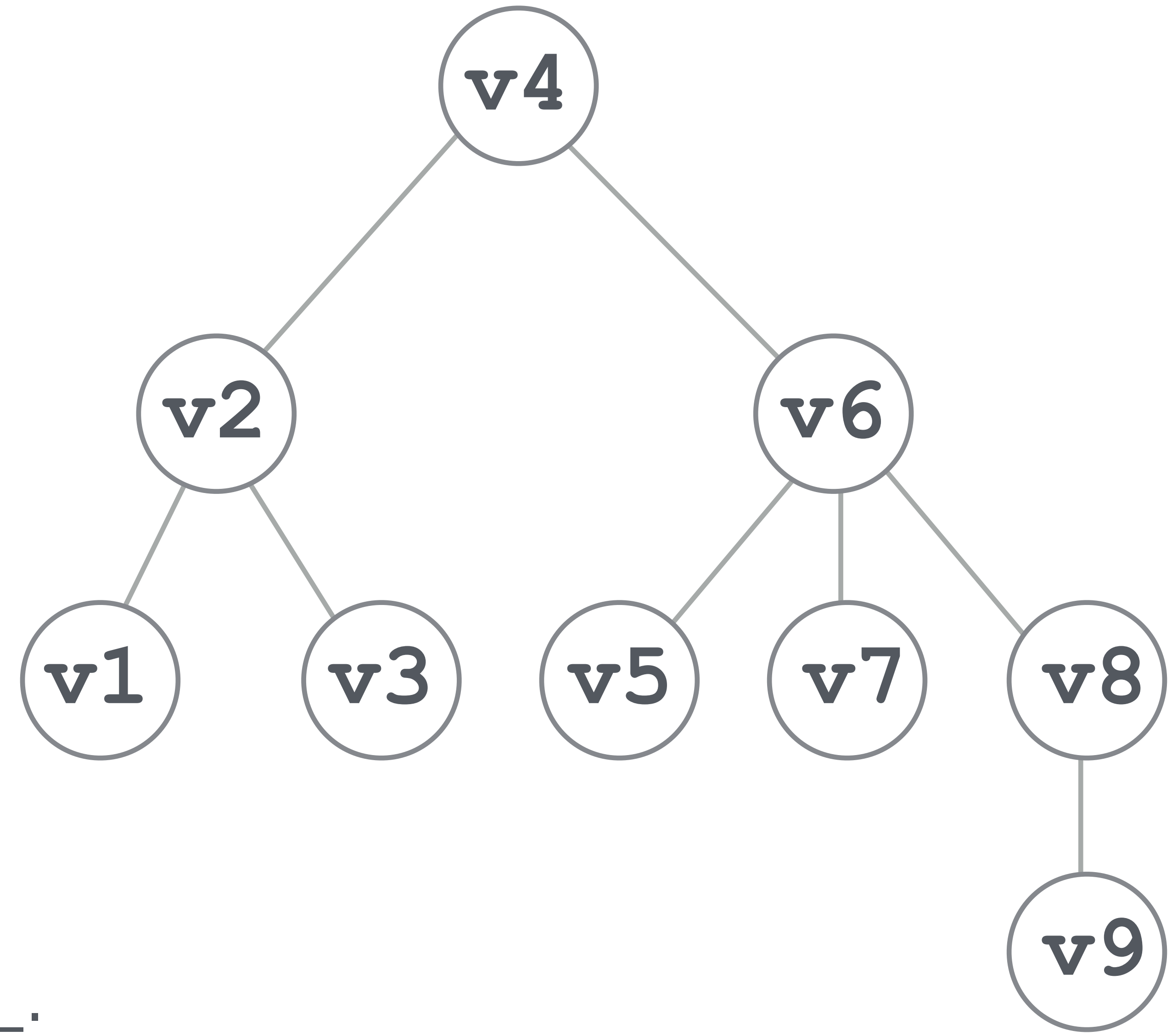
The children of **`v6`** are ___.

The ancestors of **`v1`** are ___.

The descendants of **`v6`** are ___.

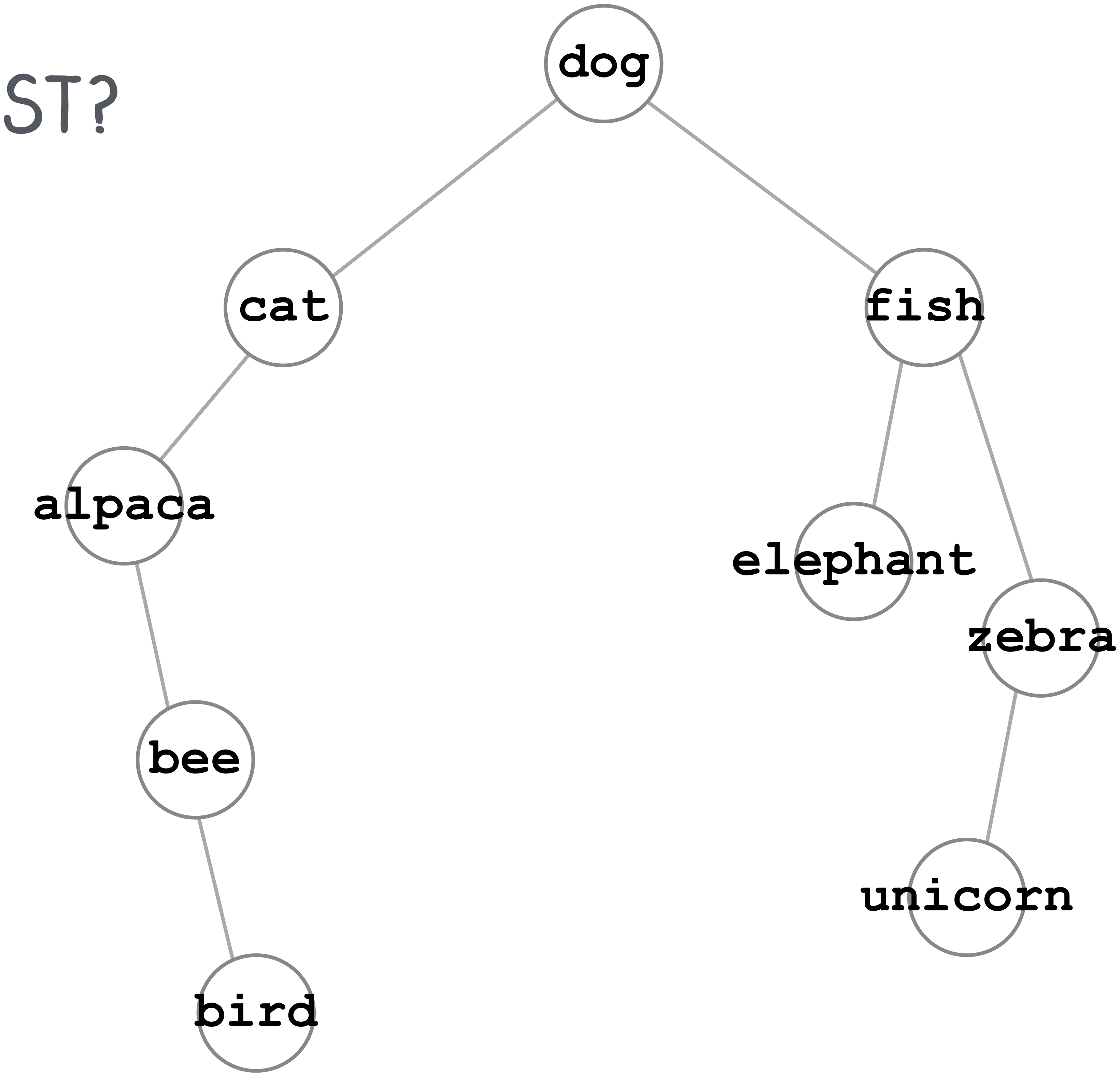
The leaves are ___.

Every node other than ___ is the root of a subtree.



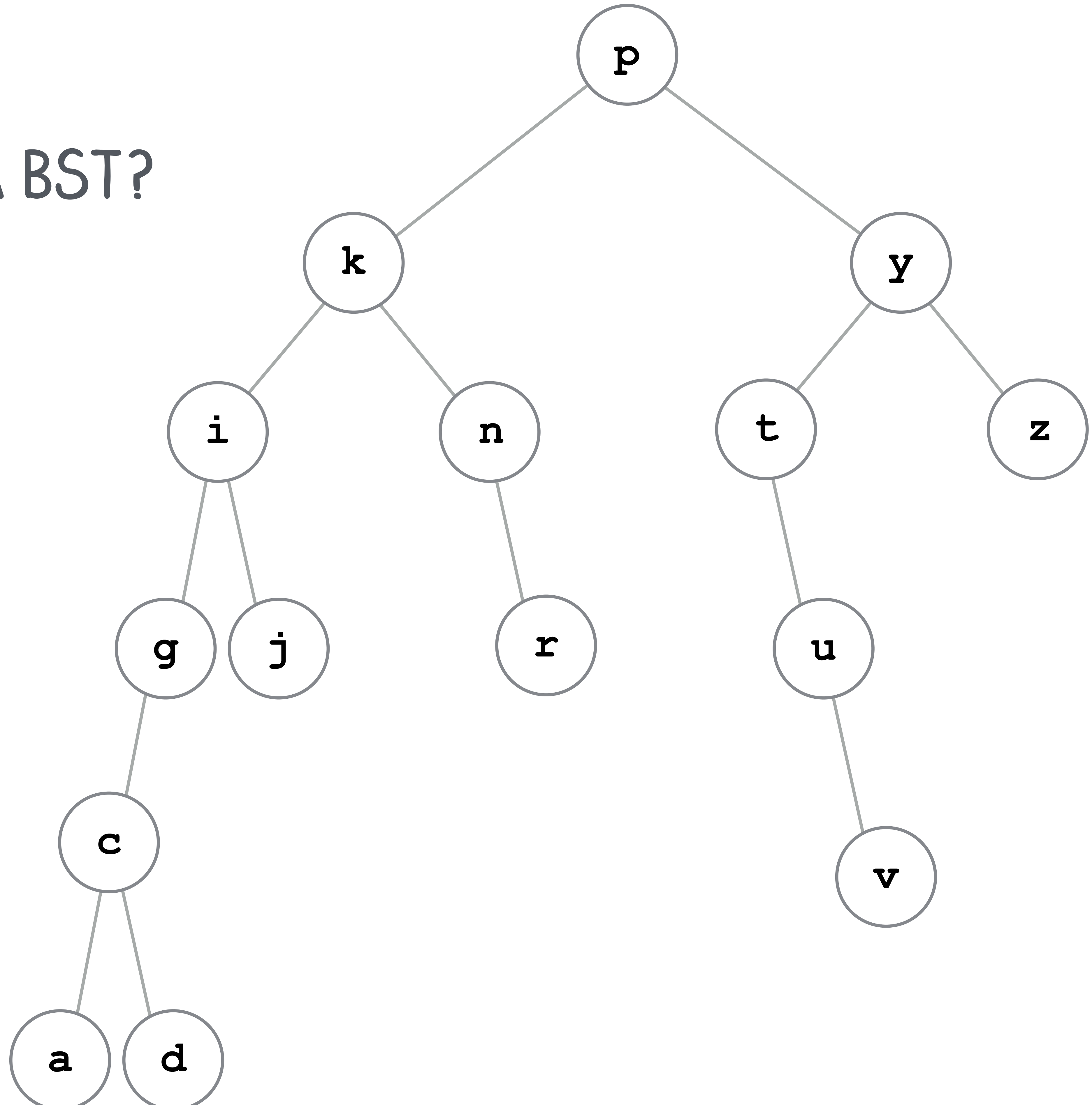
## IS THIS A BST?

A) **yes**
B) **no**

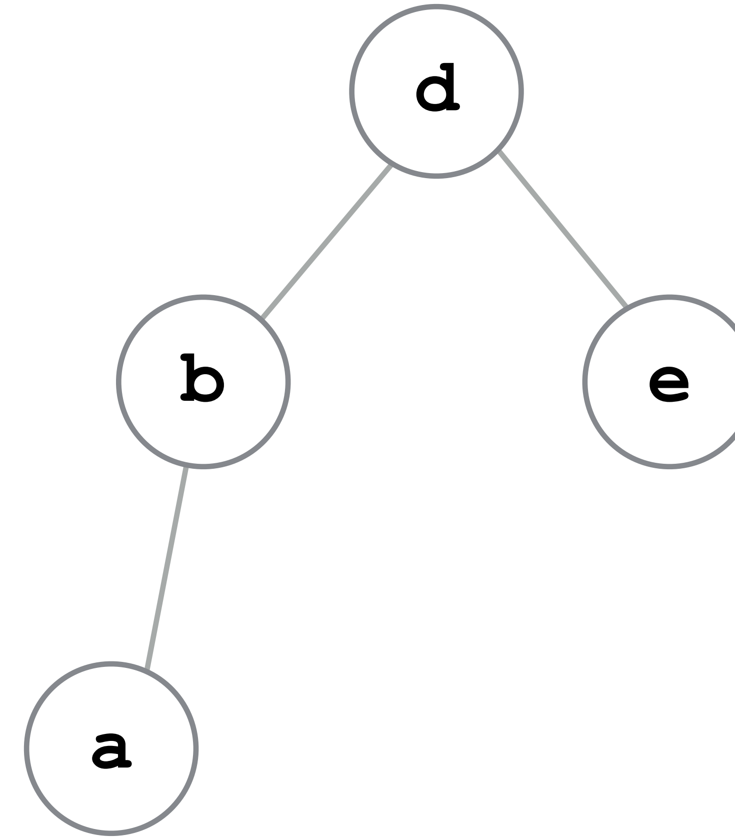


# IS THIS A BST?

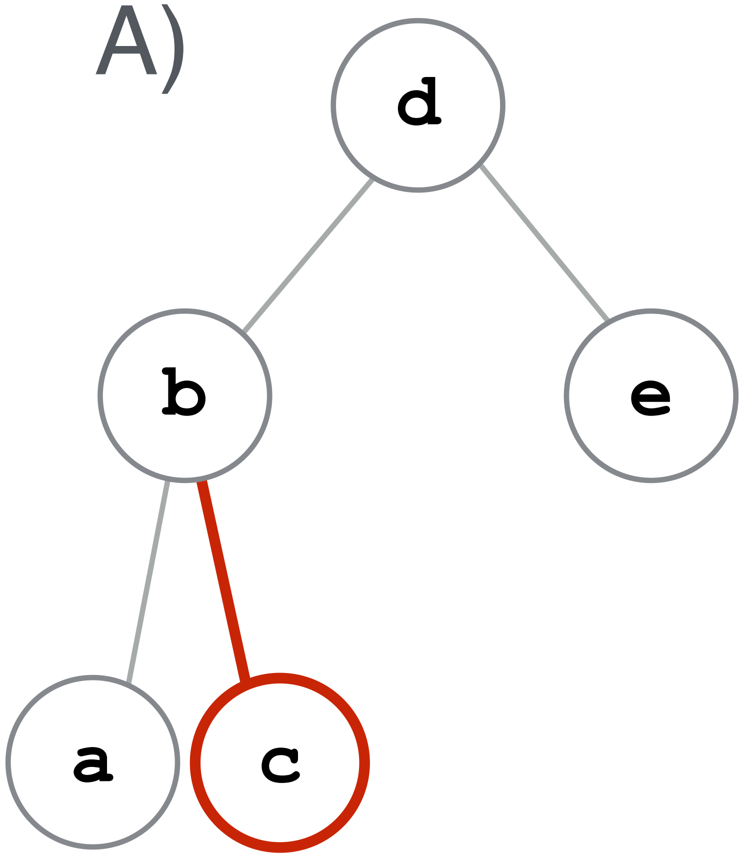
A) **yes**
B) **no**



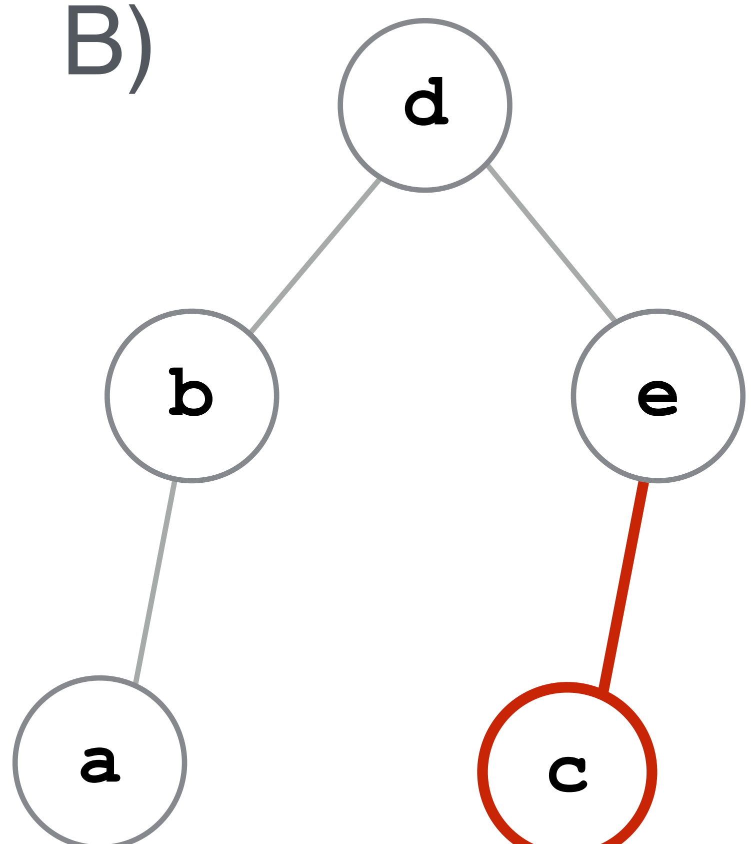
# WHICH OF THE FOLLOWING TREES IS THE RESULT OF ADDING `c` TO THIS BST?



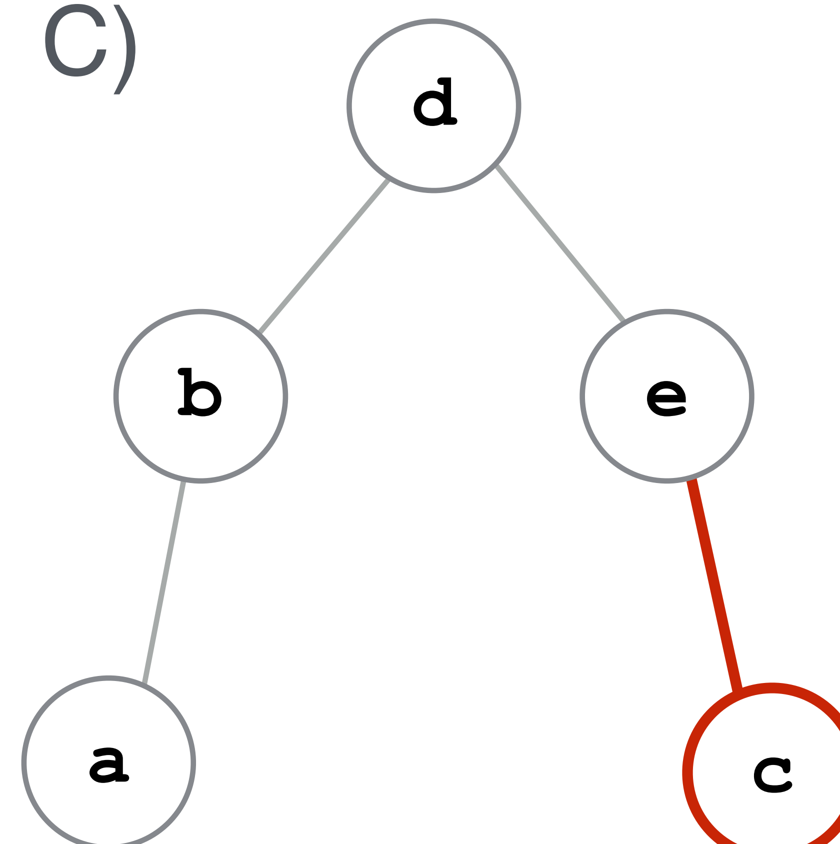

A)




B)

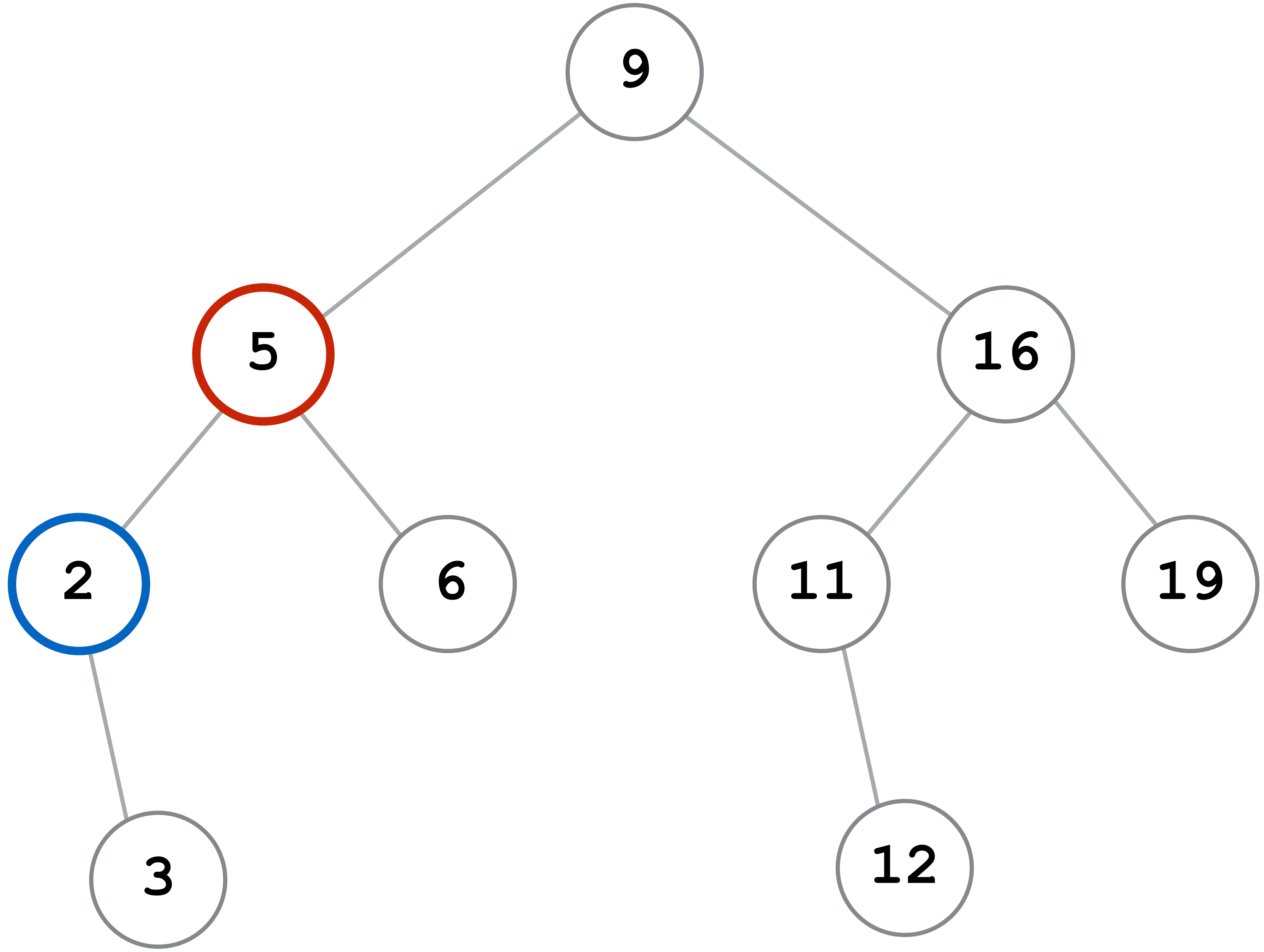



C)



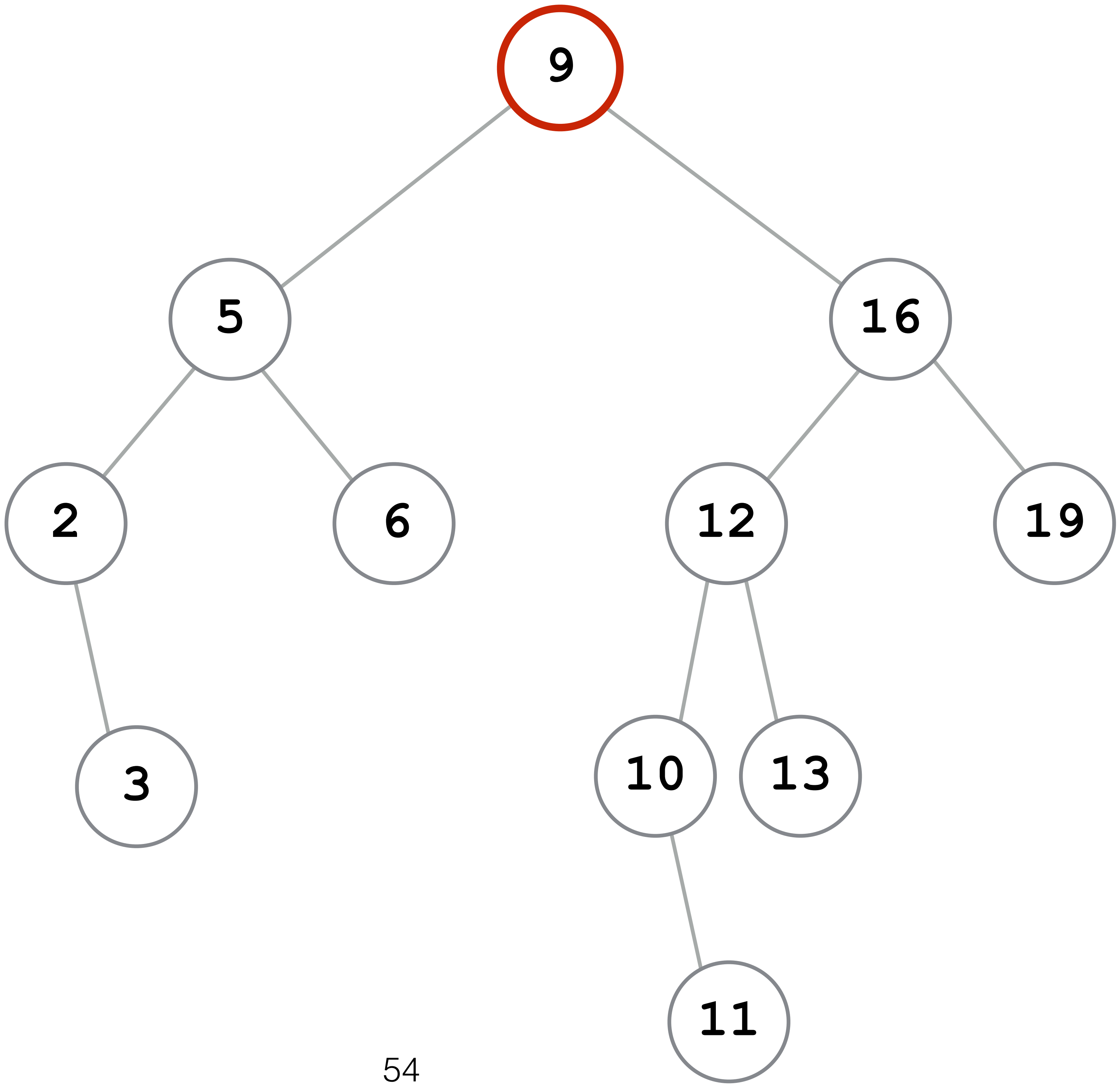
# WHAT WILL 5'S LEFT CHILD BE AFTER DELETING 2?

A) **3**
B) **6**
C) **null**



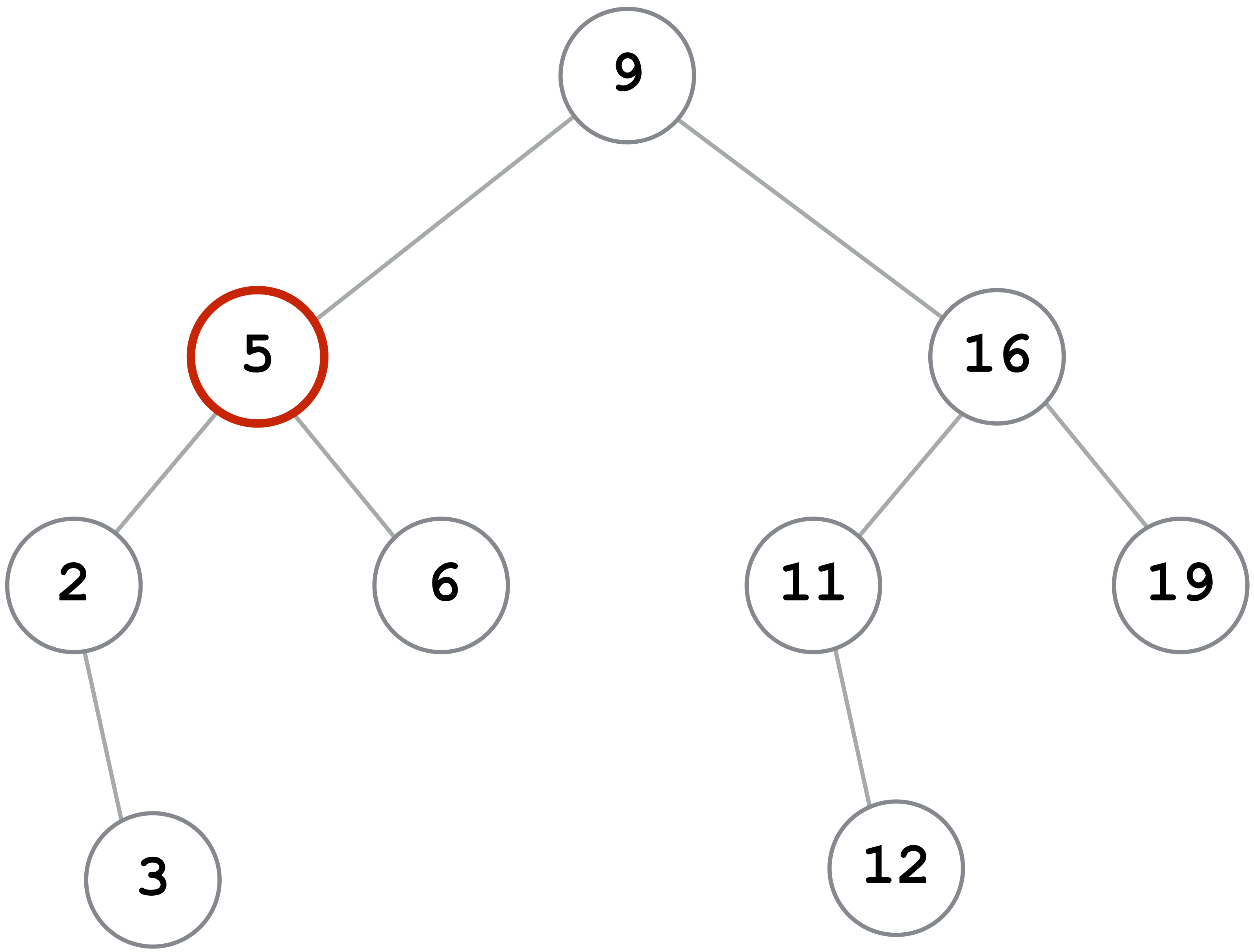
# WHAT NODE WILL REPLACE 9 AFTER DELETING 9?

A) **6**
B) **10**
C) **13**
D) **19**



# WHAT NODE WILL REPLACE 5 AFTER DELETING 5?

A) **2**
B) **3**
C) **6**
D) **12**

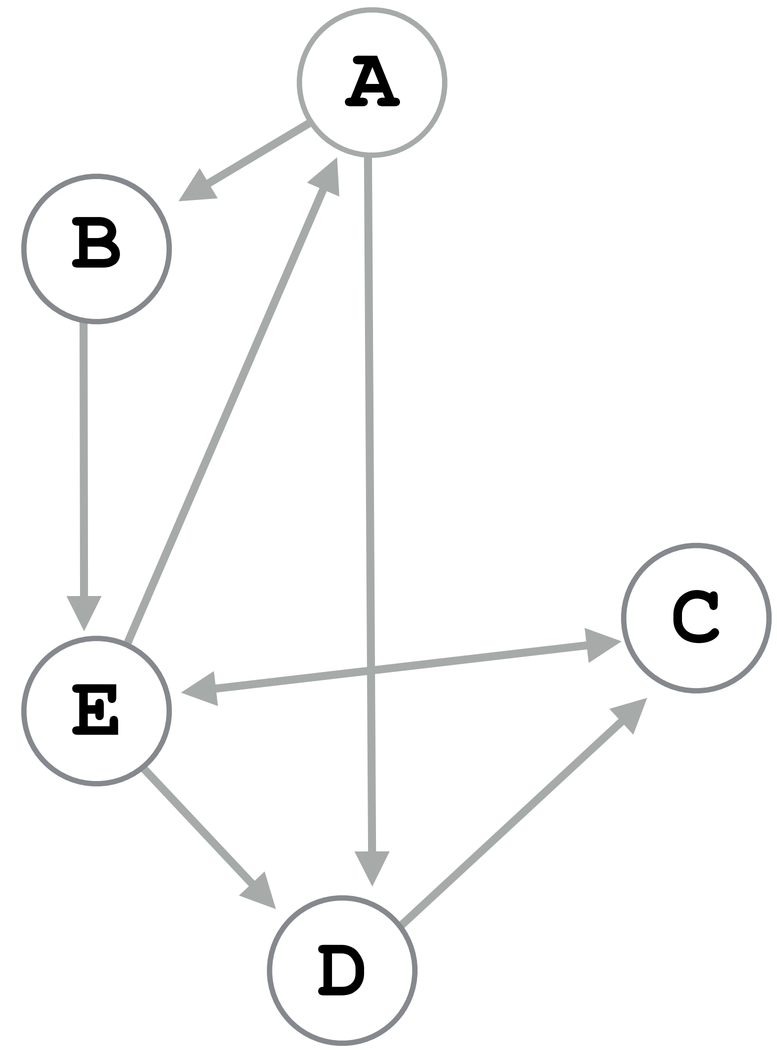


# Data Structure Operations

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | - | O(1) | O(1) | O(1) | - | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(n) | O(n) | O(n) | O(n) |
| B-Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |

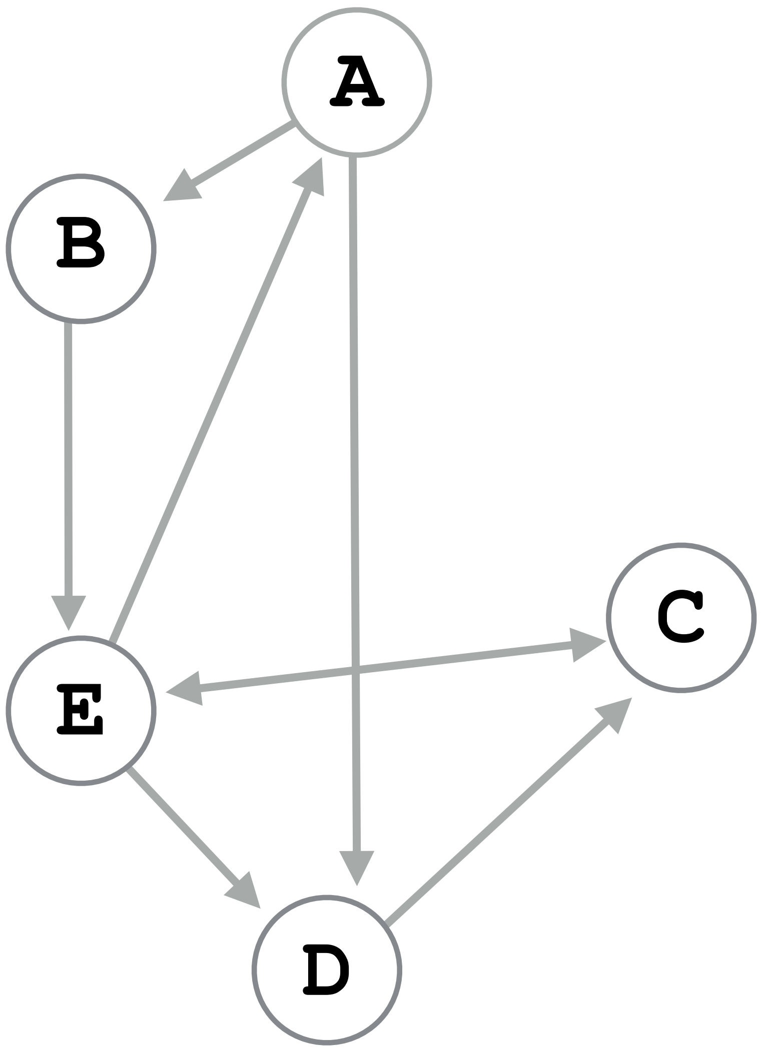
# WHAT PATH WILL BFS FIND FROM **B** TO **C**?

A) **B E C**
B) **B E A D C**
C) **B E D C**
D) **none**



# WHAT PATH WILL DFS FIND FROM A TO D?

A) **A B E D**
B) **A D**
C) **none**
D) **this is a trick question**



# WHAT IS TRUE OF DFS, SEARCHING FROM A START NODE TO A GOAL NODE?

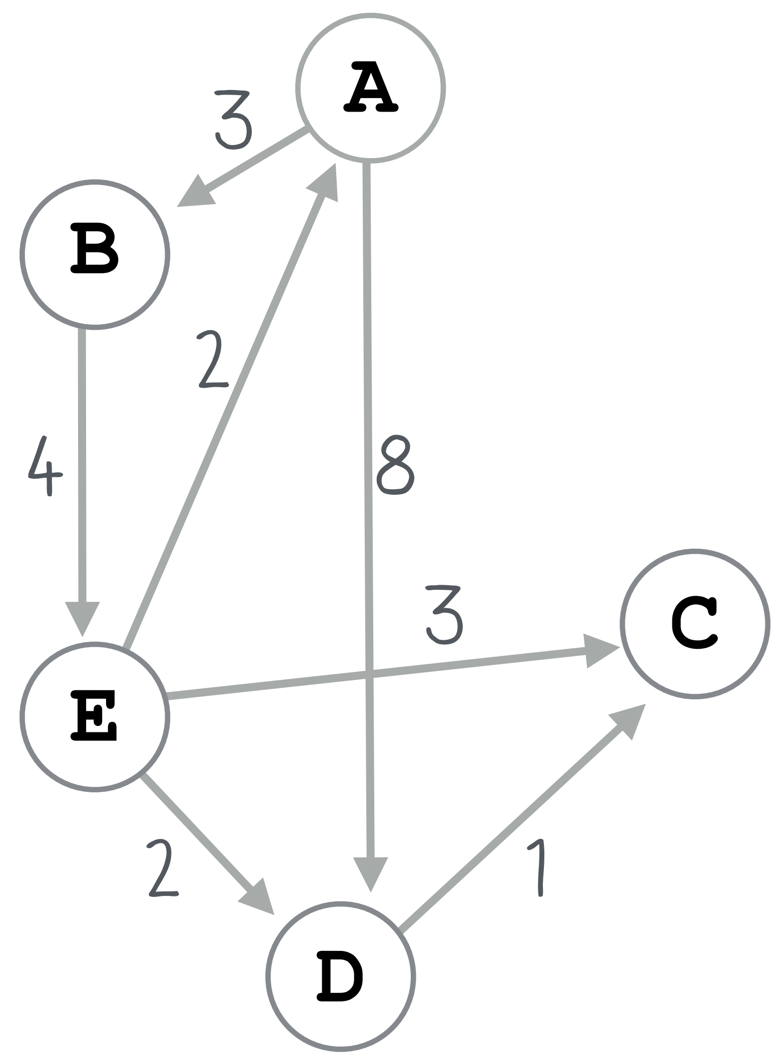
A) **if a path exists, it will find it**
B) **it is guaranteed to find the shortest path**
C) **it is guaranteed to not find the shortest path**
D) **it must be careful about cycles**
E) **a, b, and d**
F) **a, c, and d**
G) **a and d**

# WHAT IS TRUE OF BFS, SEARCHING FROM A START NODE TO A GOAL NODE?

A) **if a path exists, it will find it**
B) **it is guaranteed to find the shortest path**
C) **it is guaranteed to not find the shortest path**
D) **it must be careful about cycles**
E) **a, b, and d**
F) **a, c, and d**
G) **a and d**

# WHAT PATH WILL DIJKSTRA'S FIND FROM **A** TO **C**?

A) **A B E C**
B) **A D C**
C) **A B E D C**



# WHAT IS THE LOAD FACTOR λ FOR THE FOLLOWING HASH TABLE?

A) **4**
B) **6**
C) **0.4**
D) **0.5**
E) **0.6**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 104 | 34 | | 19 | 111 | 98 | | 52 | | |

# USING LINEAR PROBING, IN WHAT INDEX WILL ITEM 93 BE ADDED?

A) **1**
B) **5**
C) **6**
D) **7**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| array: | 49 | | 9 | 58 | 34 | | | | 18 | 89 |
| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

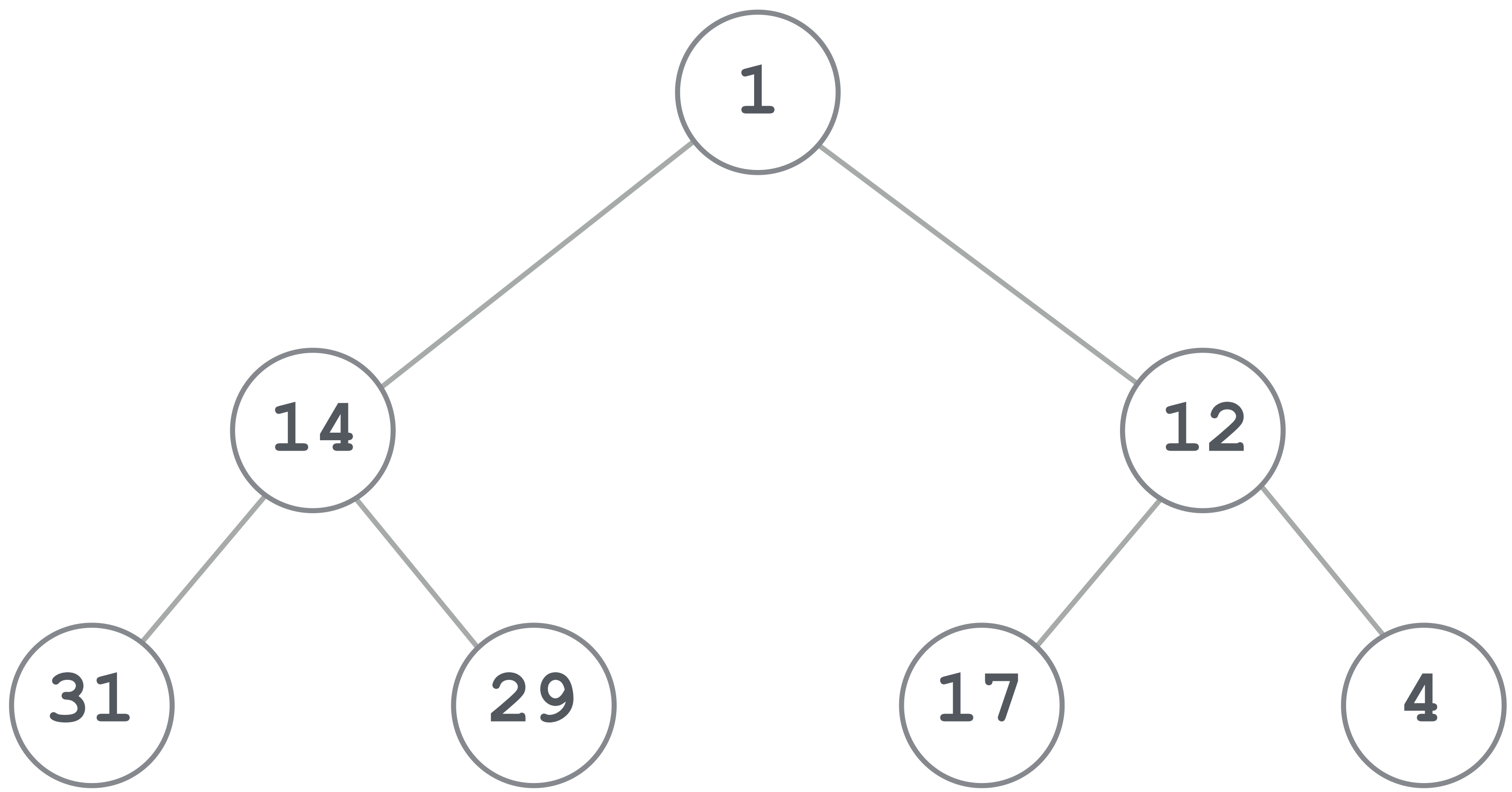
USING QUADRATIC PROBING, IN WHAT INDEX WILL ITEM 22 BE ADDED?

A) **1**
B) **5**
C) **6**
D) **7**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| array: | 49 | | 9 | 58 | 34 | | | | 18 | 89 |
| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

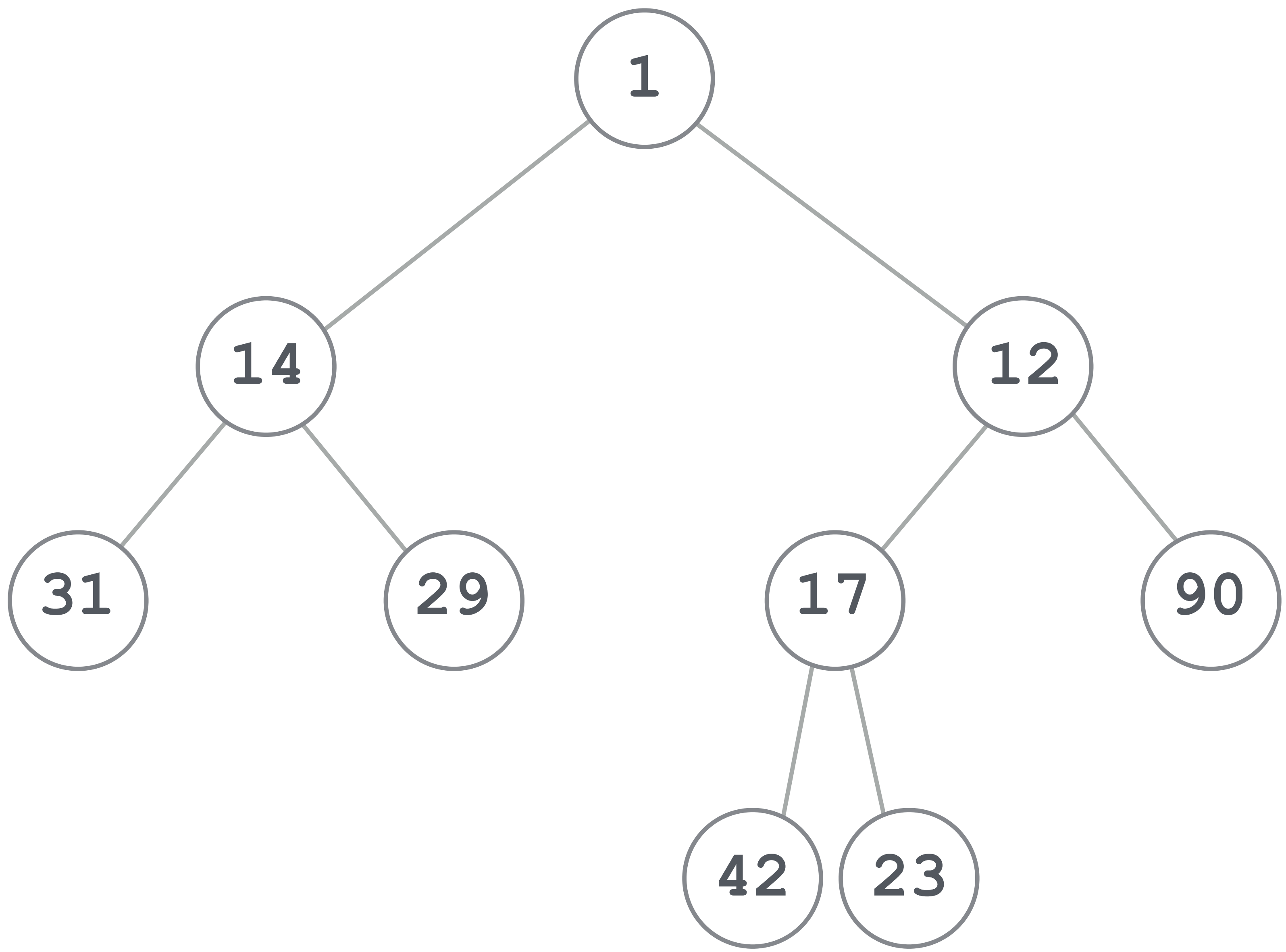
# IS THIS A MIN-HEAP?

A) **yes**
B) **no**



# IS THIS A MIN-HEAP?

A) **yes**
B) **no**



# WHAT IS THE BINARY REPRESENTATION OF THE NUMBER 39?

A) **1 0 1 0 0 1**
B) **1 0 0 1 1 1**
C) **0 1 0 1 1 1**
D) **1 1 0 0 0 1**

## HOW MANY DIFFERENT VALUES CAN 4 BITS HOLD?

A) **7**
B) **8**
C) **15**
D) **16**
E) **31**
F) **32**

## WHAT IS THE HEX VALUE OF THESE 8 BITS?
## 1010 0010

A) **B2**
B) **A2**
C) **12**
D) **10**

# WHAT STRING DO THESE BITS ENCODE?
# 0 1 1 0 0 0 0 1 0

A) **low**
B) **wow**
C) **wool**
D) **were**