# BINARY SEARCH TREES

Problem Solving with Computers-II

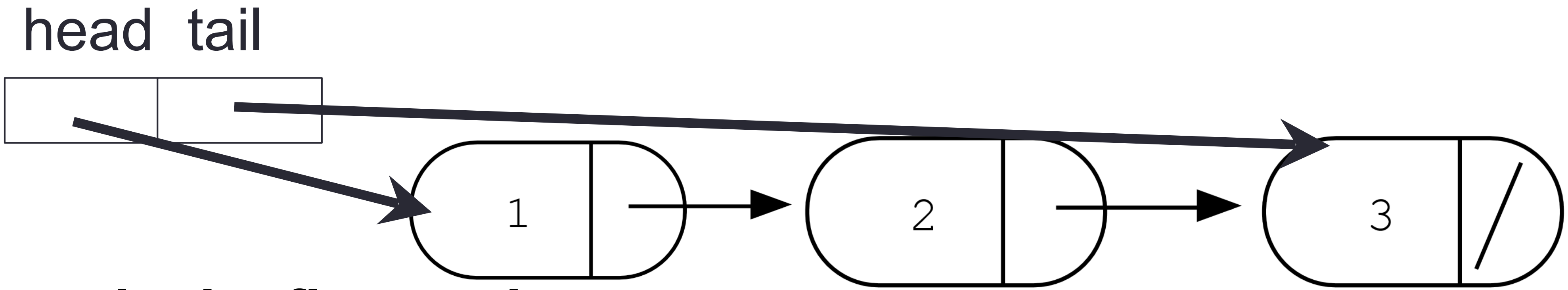
# Concept Question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
class Node {
    public:
        int info;
        Node *next;
};
```

Which of the following objects are deleted when the destructor of Linked-list is called?

**(A)**




**(B): only the first node**

**(C): A and B**

**(D): All the nodes of the linked list**

**(E): A and D**

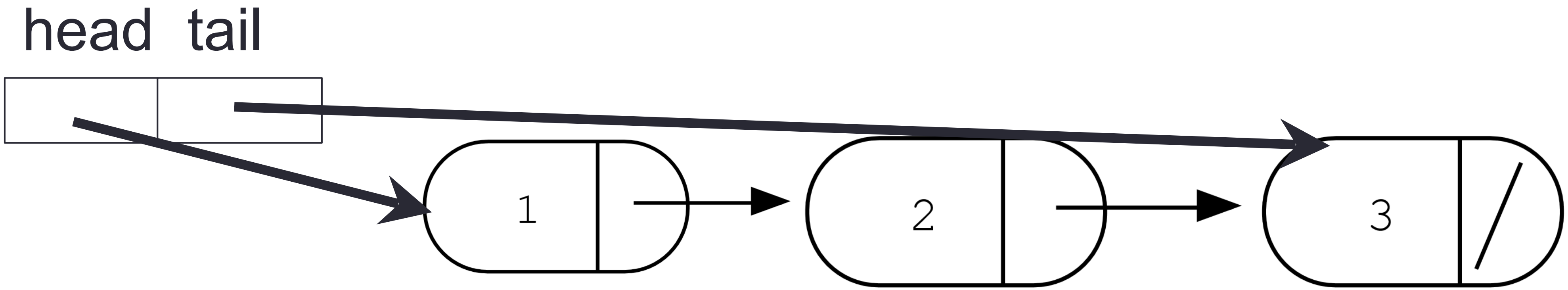
# Concept question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
Node::~Node(){
   delete next;
}
```

Which of the following objects are deleted when the destructor of Linked-list is called?

**(A)**




**(B): All the nodes in the linked-list**

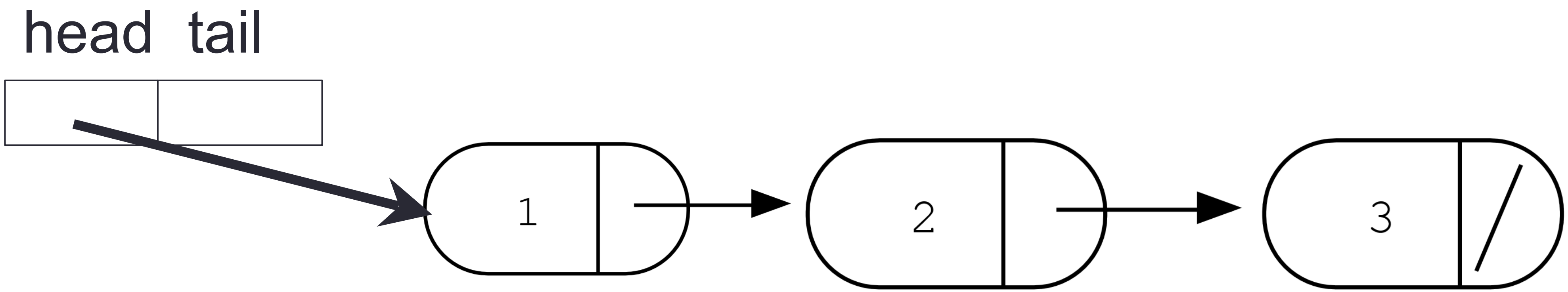
**(C): A and B**

**(D): Program crashes with a segmentation fault**

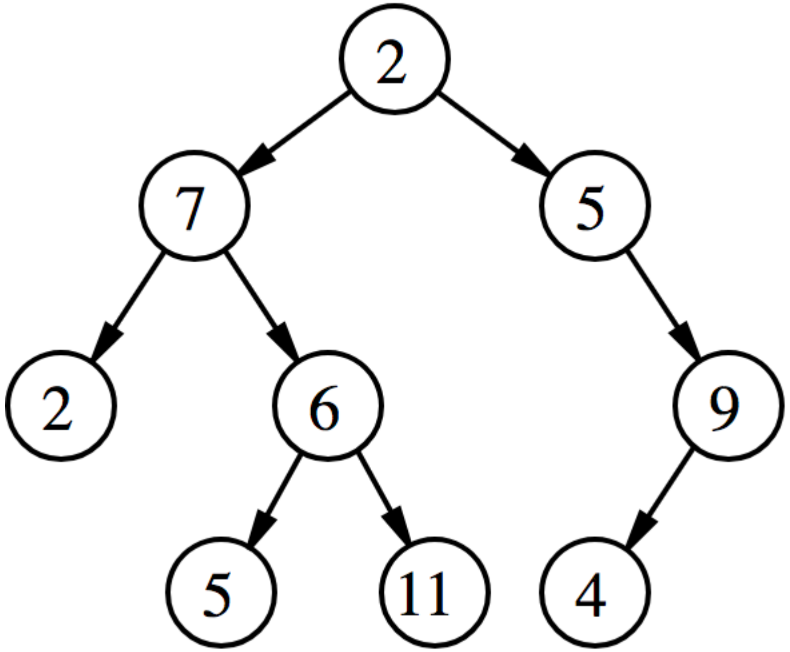
**(E): None of the above**

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
Node::~Node(){
   delete next;
}
```



# Trees

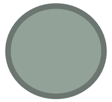



A tree has following general properties:

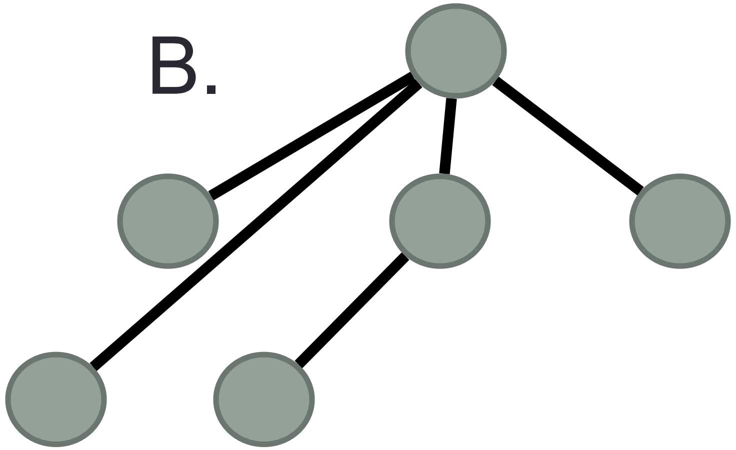
- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node;
  A direction is: *parent -> children*
- *Leaf node: Node that has no children*

# Which of the following is/are a tree?

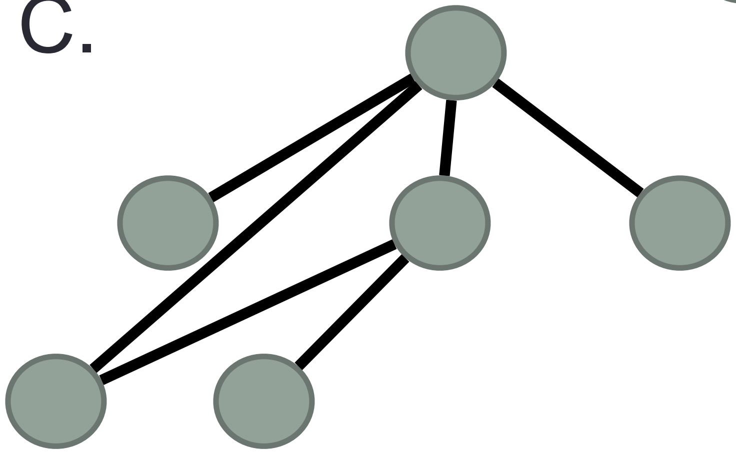
A.



B.



C.



D. A & B

E. All of A-C

# Binary Search Trees

- What are the operations supported?

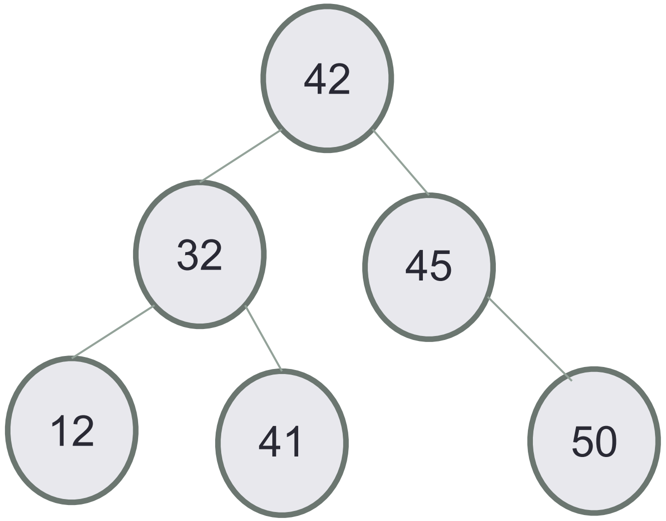
- What are the running times of these operations?

- How do you implement the BST i.e. operations supported by it?

# Operations supported by Sorted arrays and Binary Search Trees (BST)

| **Operations** | |
|---|---|
| Min | |
| Max | |
| Successor | |
| Predecessor | |
| Search | |
| Insert | |
| Delete | |
| Print elements in order | |

# Binary Search Tree – What is it?



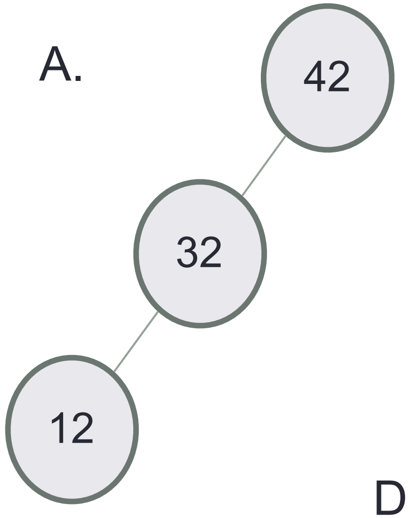

- Each node:
  - stores a key (k)
  - has a pointer to left child, right child and parent (optional)
  - Satisfies the Search Tree Property

For any node,
Keys in node's left subtree <= Node's key
Node's key < Keys in node's right subtree

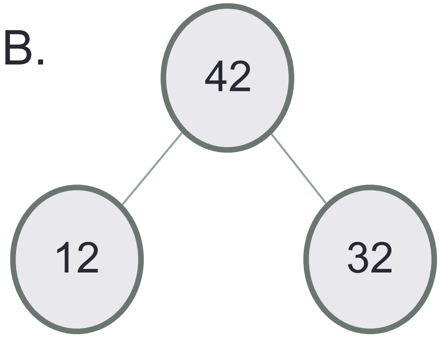
Do the keys have to be integers?

# Which of the following is/are a binary search tree?

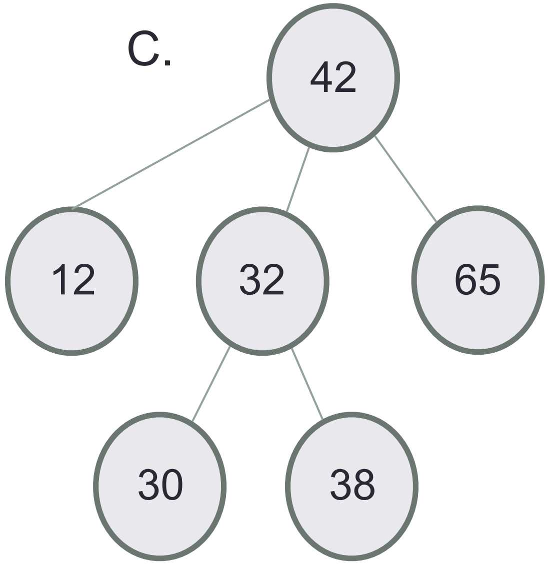
A.



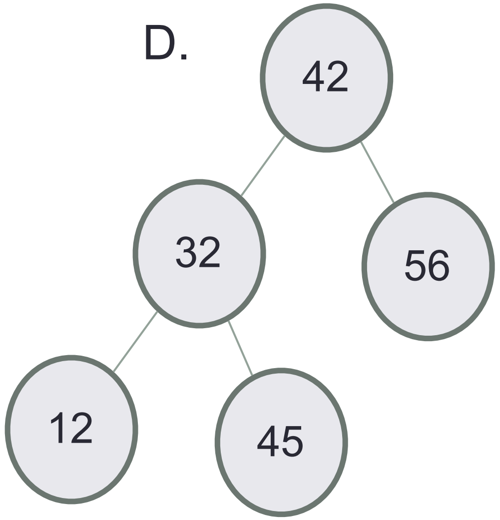

B.




C.

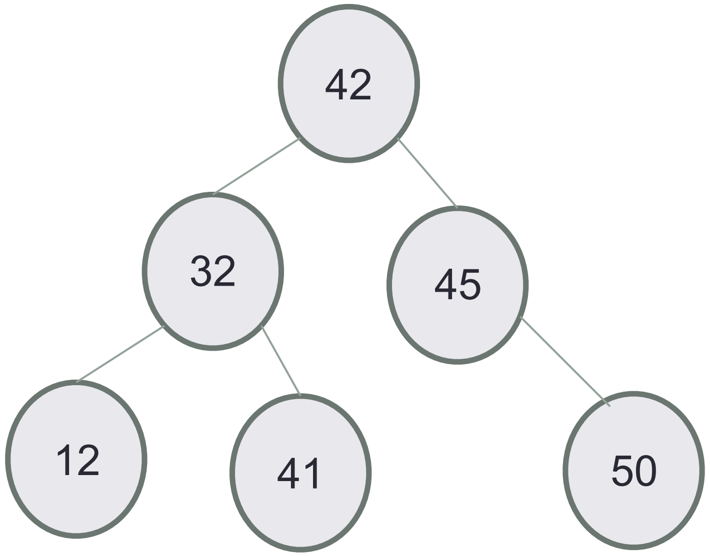



D.




E. More than one of these

# BSTs allow efficient search!




- Start at the root;
- Trace down a path by comparing **k** with the key of the current node x:
    - If the keys are equal: we have found the key
    - If **k** < key[x] search in the left subtree of x
    - If **k** > key[x] search in the right subtree of x

**Search for 41, then search for 53**

# A node in a BST

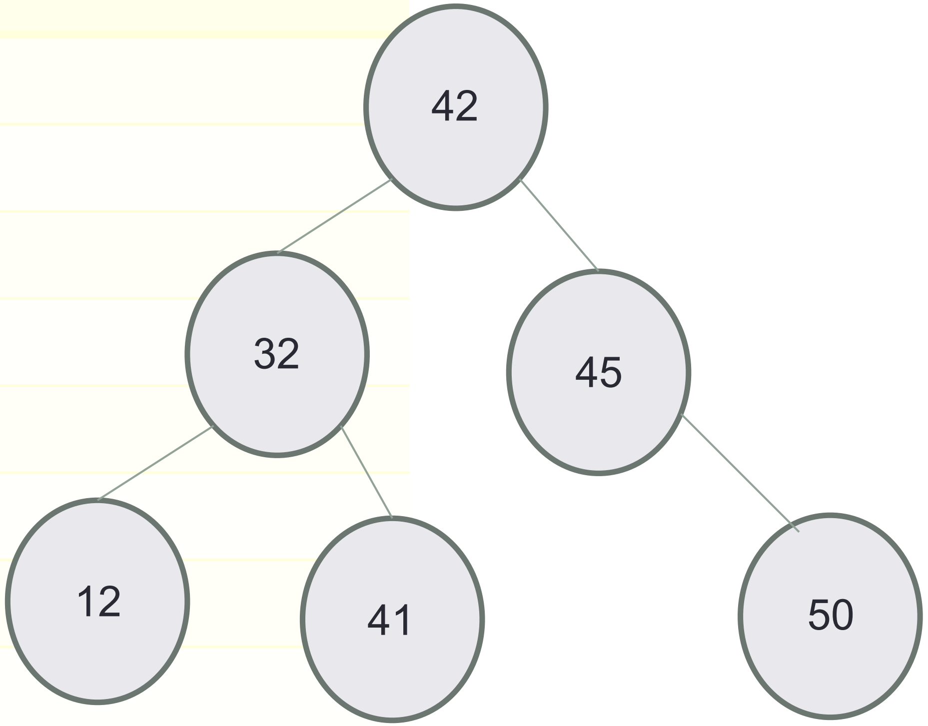
```
class BSTNode {

public:
   BSTNode* left;
   BSTNode* right;
   BSTNode* parent;
   int const data;

   BSTNode( const int & d ) : data(d) {
      left = right = parent = 0;
   }
};
```

# Define the BST ADT

| **Operations** |
|---|
| Search |
| Insert |
| Min |
| Max |
| Successor |
| Predecessor |
| Delete |
| Print elements in order |

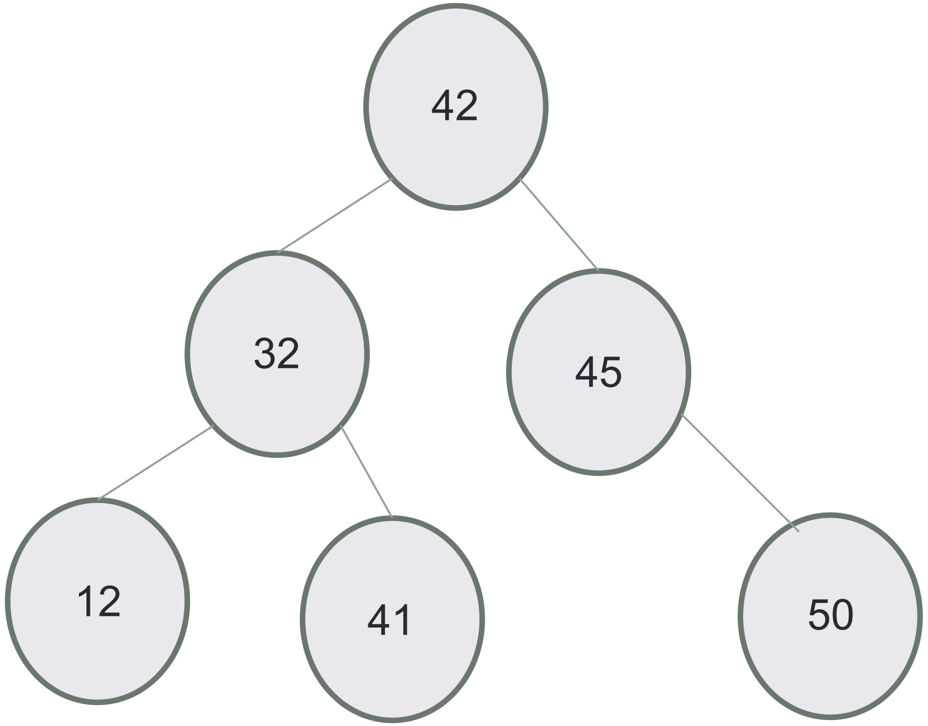


# Traversing down the tree

- Suppose n is a pointer to the root. What is the output of the following code:

```
n = n->left;
n = n->right;
cout<<n->data<<endl;
```

A. 42
B. 32
C. 12
D. 41
E. Segfault

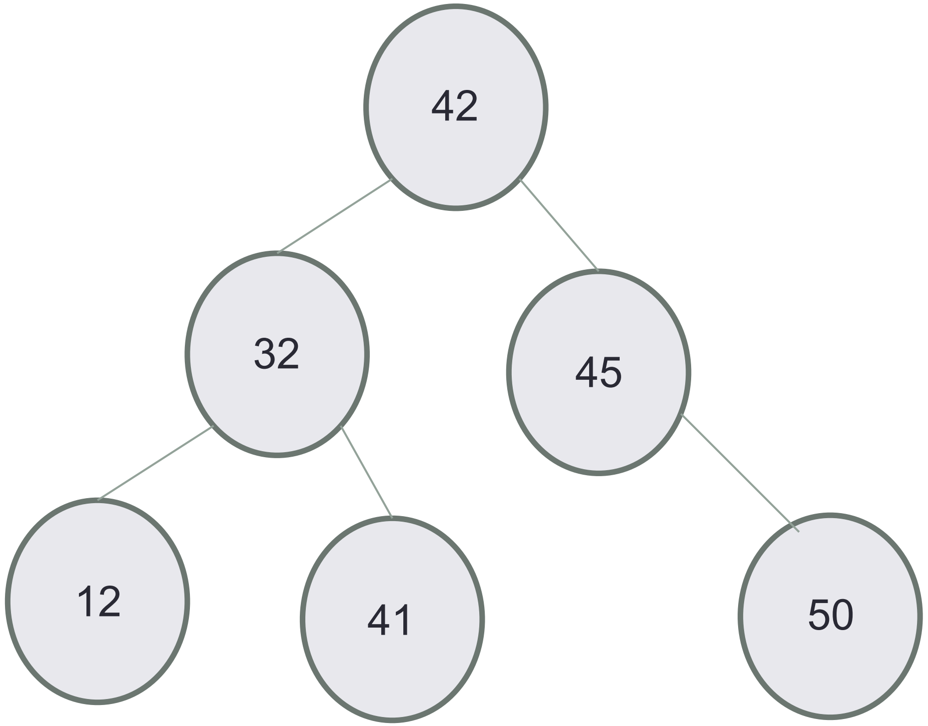


# Traversing up the tree

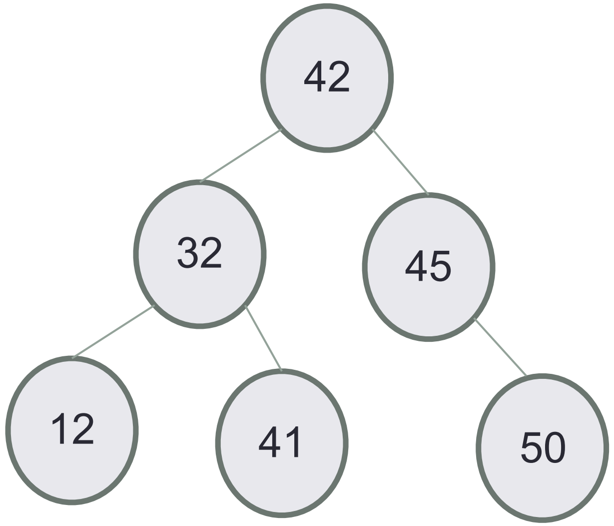
- Suppose n is a pointer to the node with value 50.
- What is the output of the following code:

```
n = n->parent;
n = n->parent;
n = n->left;
cout<<n->data<<endl;
```

A. 42
B. 32
C. 12
D. 45
E. Segfault



# Insert



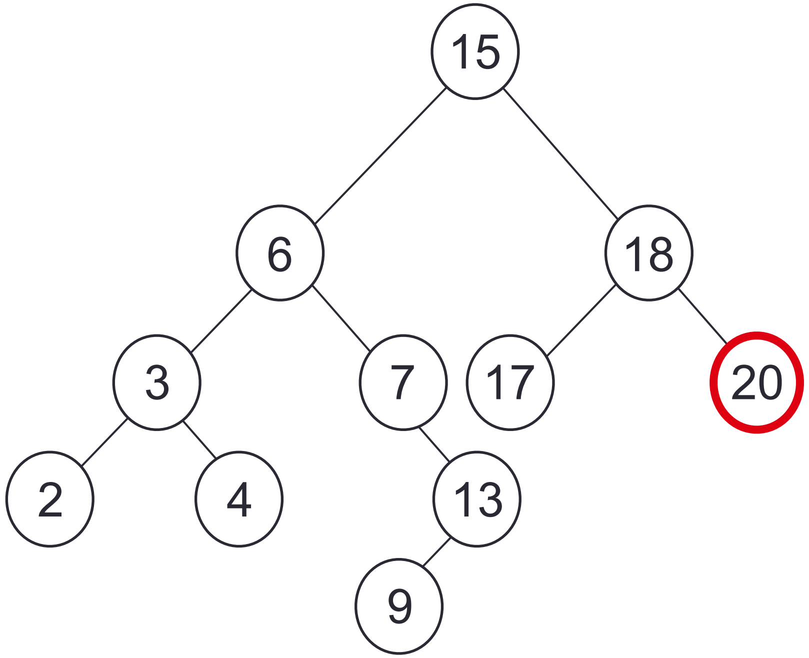

- Insert 40
- Search for the key
- Insert at the spot you expected to find it

# Max

**Goal**: find the maximum key value in a BST

Following right child pointers from the root, until a leaf node is encountered. The least node has the max value

**Alg:** `int BST::max()`




**Maximum = 20**

# Min

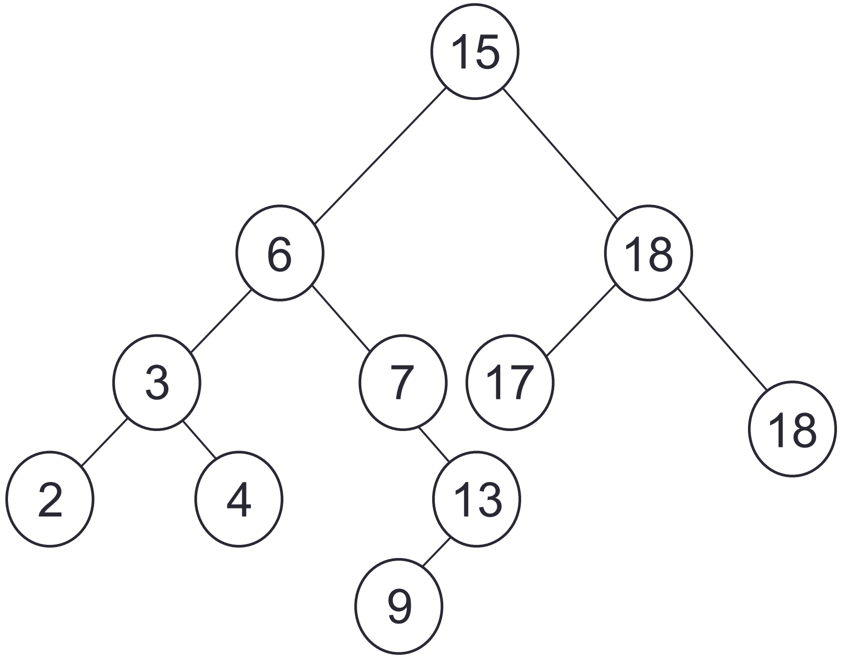
**Goal**: find the minimum key value in a BST

Start at the root.

Follow _________ child pointers from the root, until a leaf node is encountered

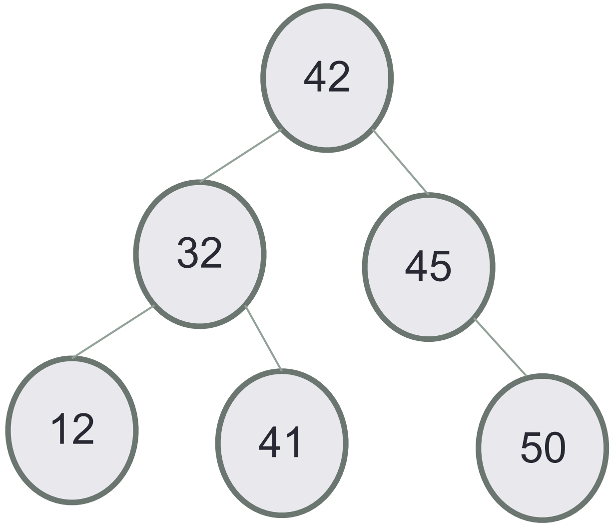
Leaf node has the min key value

**Alg:** `int BST::min()`




Min = ?

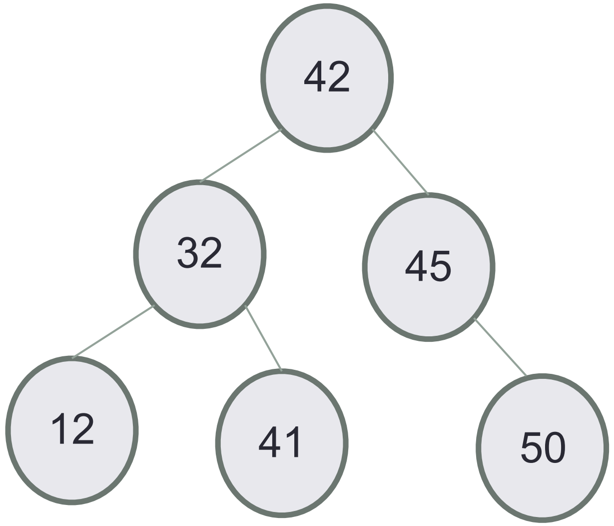
# In order traversal: print elements in sorted order




Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

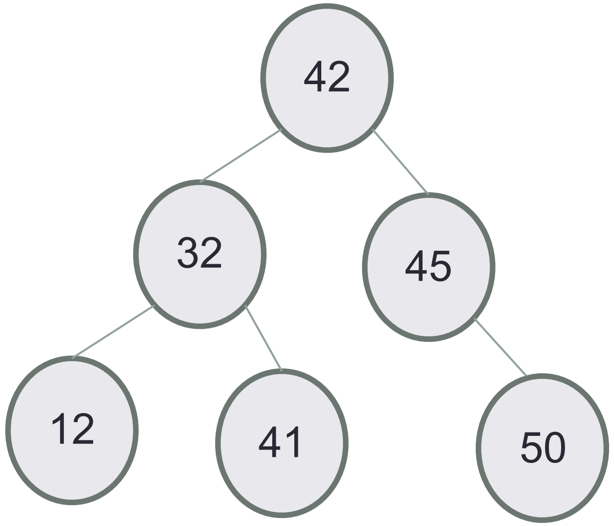
# Pre-order traversal: nice way to linearize your tree!




Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

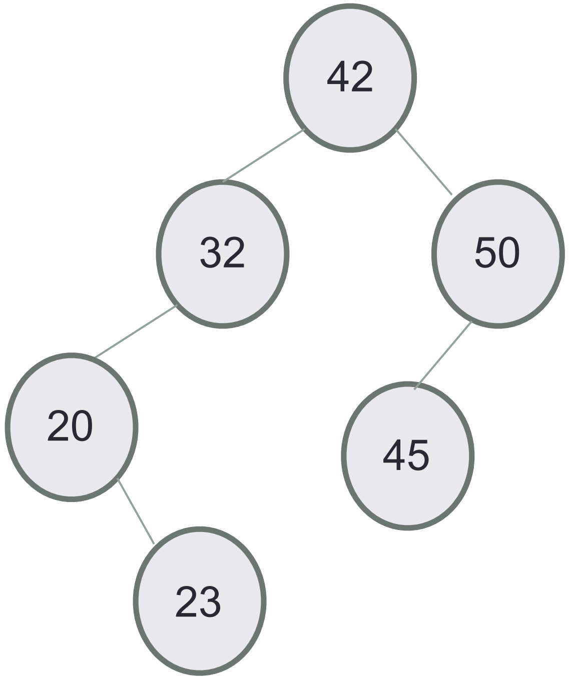
# Post-order traversal: use in recursive destructors!




Algorithm Postorder(tree)

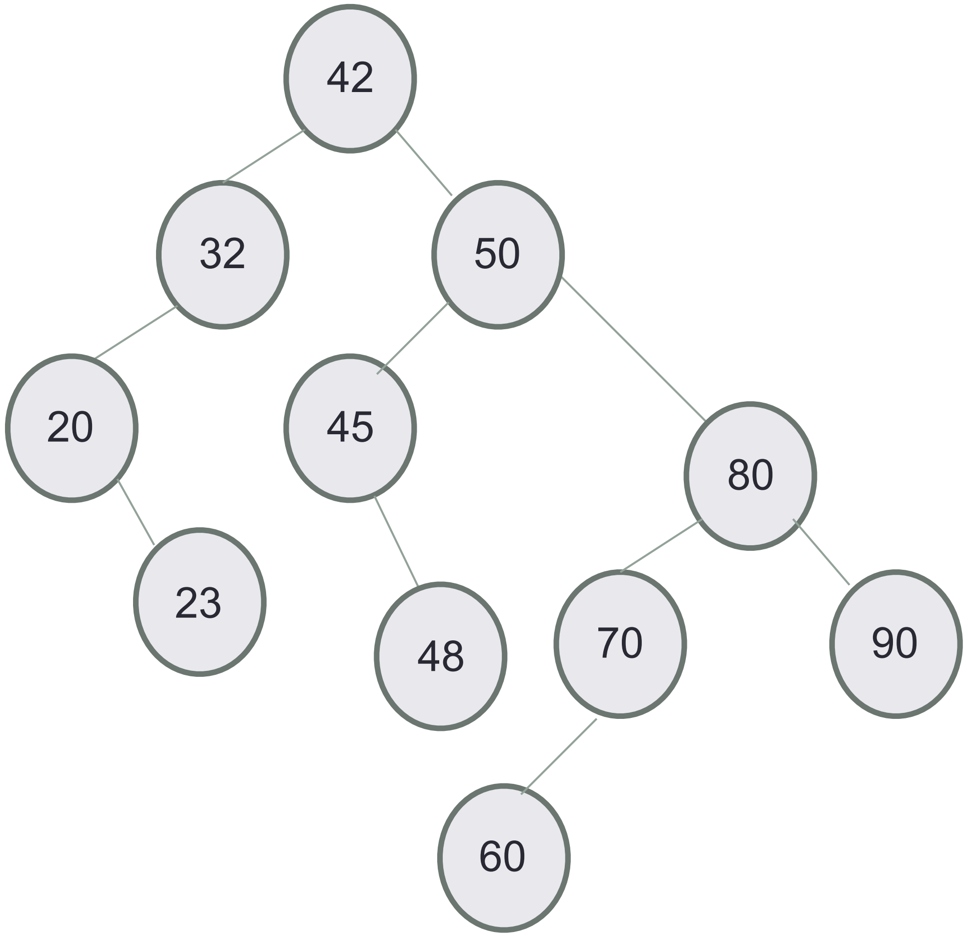
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

# Predecessor: Next smallest element



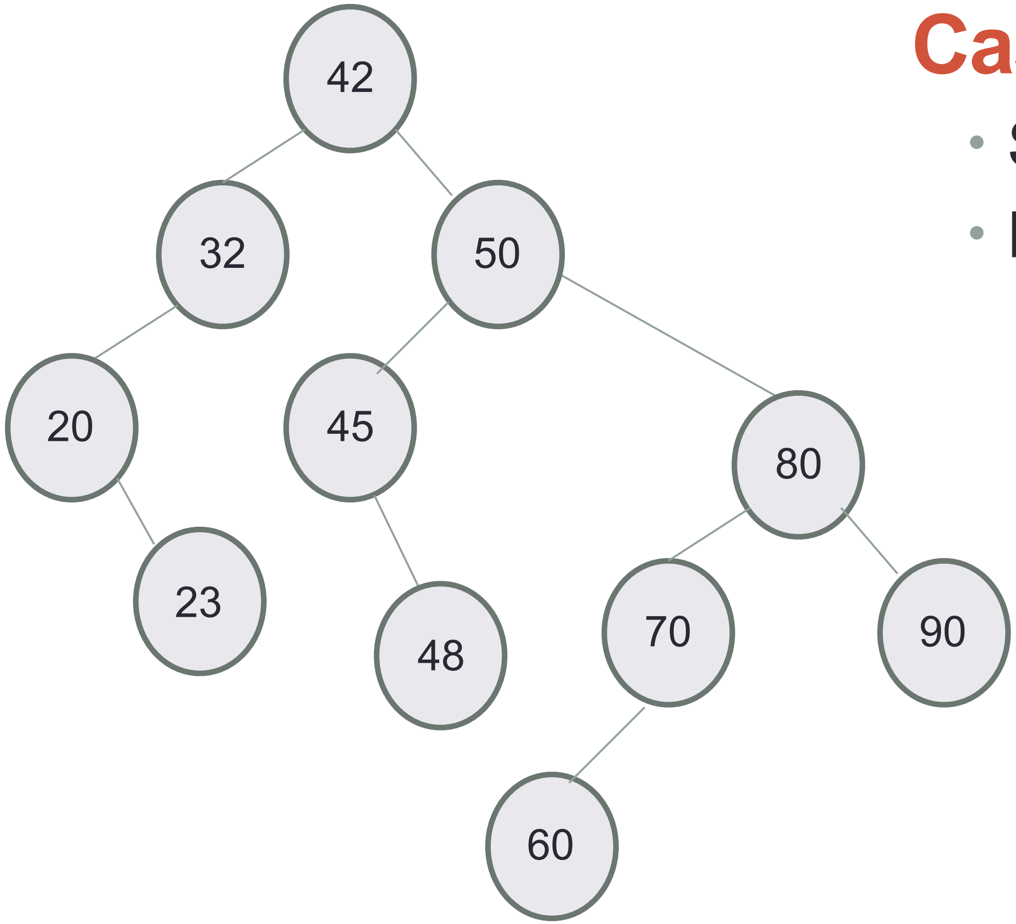

- What is the predecessor of 32?
- What is the predecessor of 45?

# Successor: Next largest element




- What is the successor of 45?
- What is the successor of 50?
- What is the successor of 60?

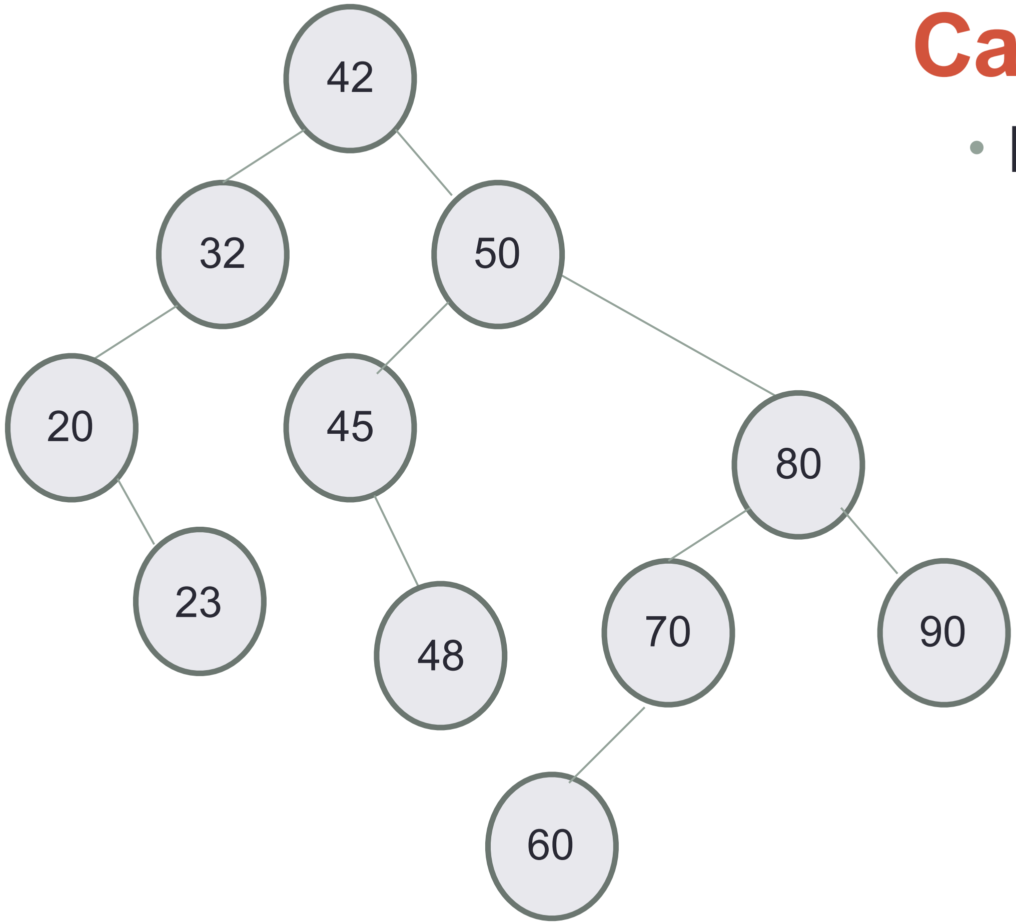
# Delete: Case 1




## Case 1: Node is a leaf node

- Set parent's (left/right) child pointer to null
- Delete the node

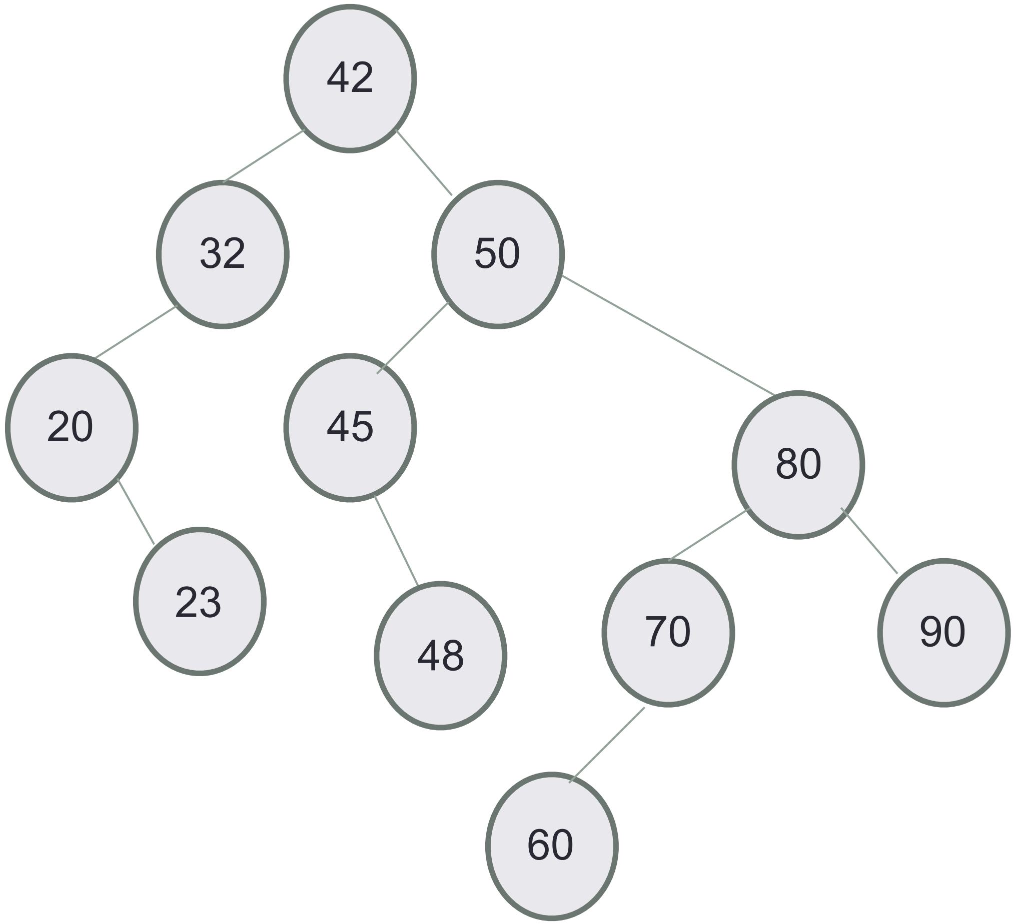
# Delete: Case 2




## Case 2 Node has only one child

- Replace the node by its only child

# Delete: Case 3




## Case 3 Node has two children

- Can we still replace the node by one of its children? Why or Why not?

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

- Ex. Binary search for 33.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 13 | 14 | 25 | 33 | 43 | 51 | 53 | 64 | 72 | 84 | 93 | 95 | 96 | 97 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| lo | | | | | | | | | | | | | | hi |