COSC 2436: Final Exam Review

Hashing #1

Insert the following values into a hash table using linear probing. Assume the hash table is of size 10.

{54, 75, 24, 45, 18, 10}

Hashing #1

Insert the following values into a hash table using linear probing. Assume the hash table is of size 10.

{54, 75, 24, 45, 18, 10}

0	1	2	3	4	5	6	7	8	9
10				54	75	24	45	18	

Hashing #2

The code below shows double hashing. What is wrong with the code?

```
void doubleHashing(int table[], int x, int tableSize){
  for(int i = 0; i < tableSize; i++){
    int index = (hash1(x, tableSize) + (i * hash2(x, 7))) % tableSize;
    if(table[index] == -1){
      table[index] = x;
    }
  }
}
```

Hashing #2

The code below shows double hashing. What is wrong with the code?

There should be a *break* statement after line 77. If there is no break statement, *x* will keep getting added to the table.

```
73 void doubleHashing(int table[], int x, int tableSize){
74   for(int i = 0; i < tableSize; i++){
75     int index = (hash1(x, tableSize) + (i * hash2(x, 7))) % tableSize;
76     if(table[index] == -1){
77       table[index] = x;
78     }
79   }
80 }
```

add *break;*

Hashing #3

Match the following hash function with its correct description.

- **Direct Hashing ____**
- **Linear Probing ____**
- **Double Hashing ____**

a) data is overwritten during collision

b) clustering can occur

c) uses two hash functions

Hashing #3

Match the following hash function with its correct description.

- **Direct Hashing __a__**
- **Linear Probing __b__**
- **Double Hashing __c__**

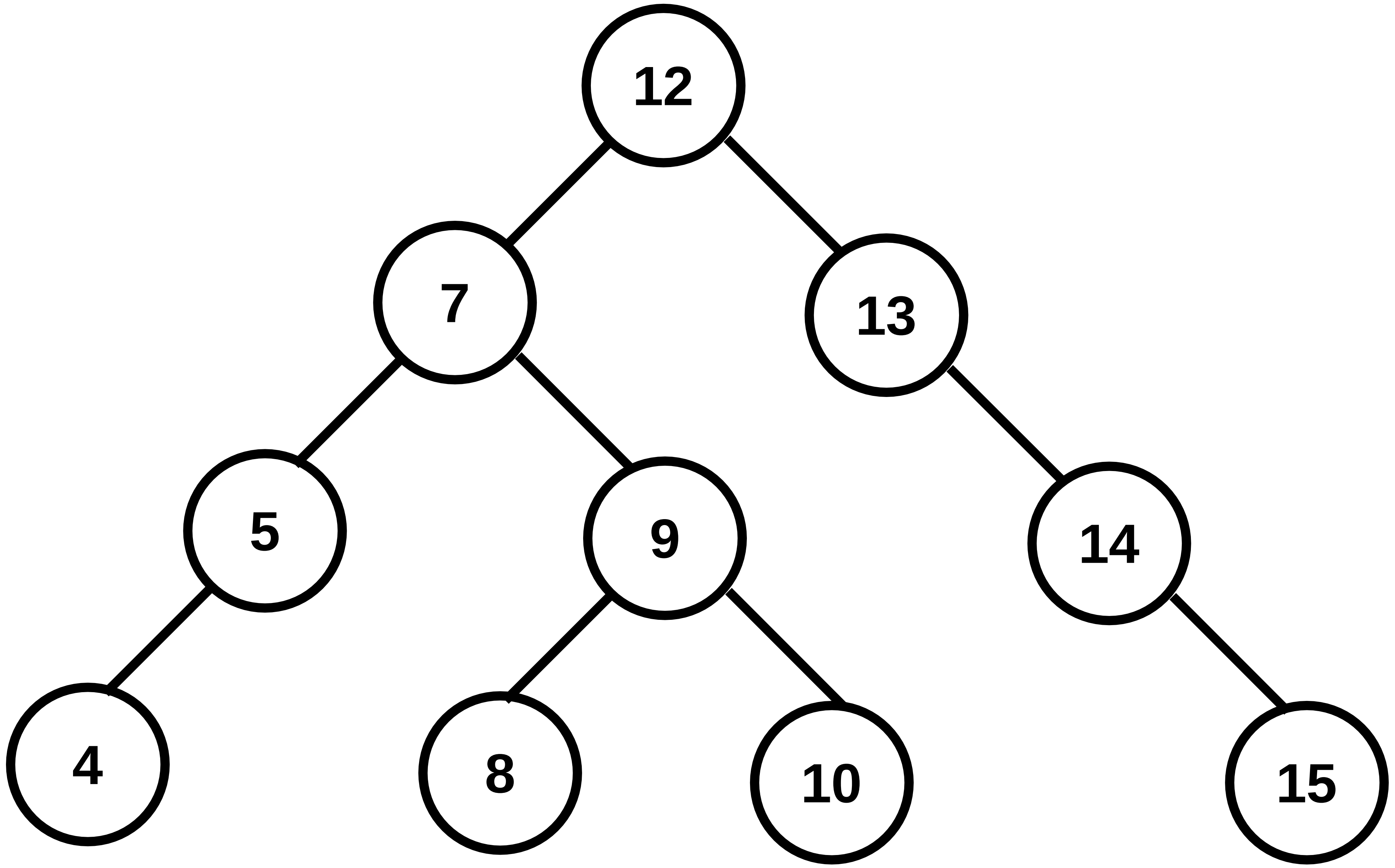
a) data is overwritten during collision

b) clustering can occur

c) uses two hash functions

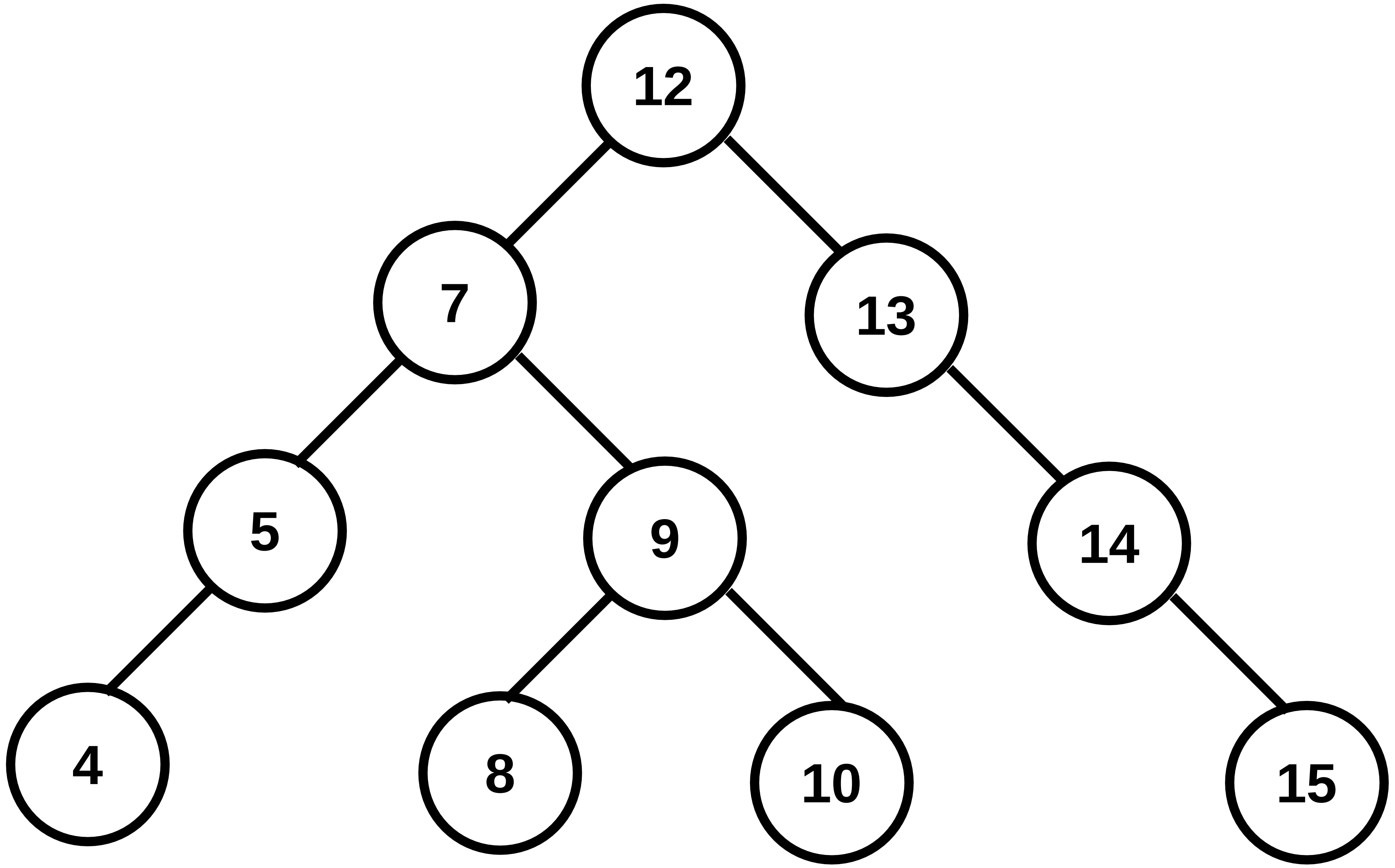
BST #1

Perform preorder traversal on the following BST.



BST #1

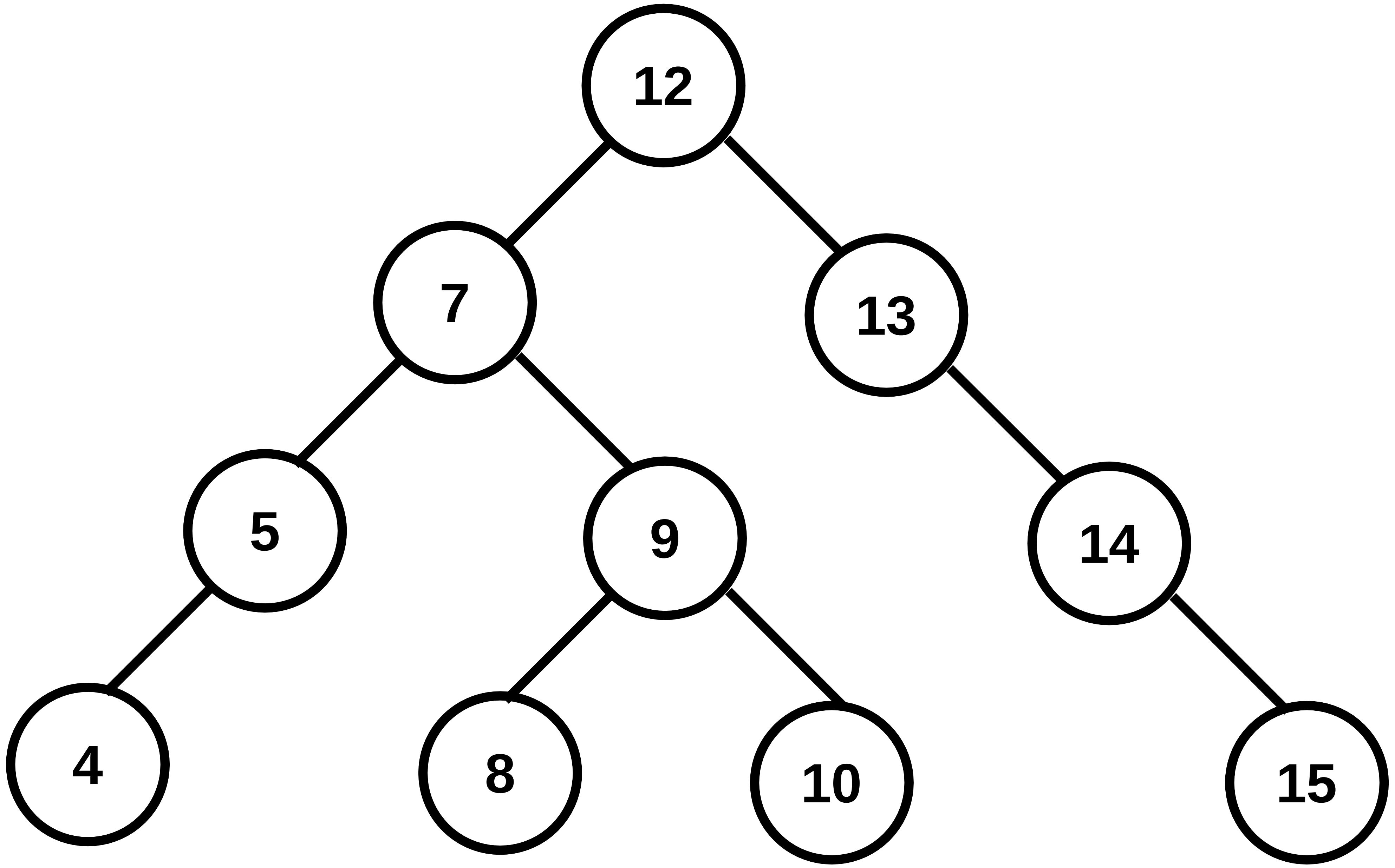
Perform preorder traversal on the following BST.



Preorder: 12 7 5 4 9 8 10 13 14 15

BST #2

Write the function which produced the output below.



Output: 4 5 8 10 9 7 15 14 13 12

BST #2

Write the function which produced the output below.

```
void postorder(node *n){
  if(n == nullptr)
    return;
  postorder(n->left);
  postorder(n->right);
  cout << n->value << " ";
}
```

BST #3

Write the function getSum( ) which returns the sum of all the values in a BST.

```
struct node{
   int value;
   node *right;
   node *left;
};

int getSum(node *n){



}
```

BST #3

Write the function getSum( ) which returns the sum of all the values in a BST.

```
int getSum(node *n){
  if(n == nullptr)
    return 0;
  return (n->value + getSum(n->right) + getSum(n->left));
}
```

BST #4

Write the function leafCount( ) which returns the number of leafs in a BST.

```
struct node{
    int value;
    node *left;
    node *right;
};

int leafCount(node *root){




}
```

BST #4

Write the function leafCount( ) which returns the number of leafs in a BST.

```
int leafCount(node *n){
    if(n == nullptr)
        return 0;
    else if(n->left == nullptr && n->right == nullptr)
        return 1;
    else
        return leafCount(n->left) + leafCount(n->right);
}
```

BST #5

Which of the following is <u>NOT</u> a property of a BST:

A) Best case time complexity is O(log(n))
B) Left-Child < Root < Right-Child
C) Only contains unique values
D) Can have more than 2 children
E) None of the above

BST #5

Which of the following is <u>NOT</u> a property of a BST:

A) Best case time complexity is O(log(n))
B) Left-Child < Root < Right-Child
C) Only contains unique values
D) Can have more than 2 children
E) None of the above

D) Can have more than 2 children

AVL Tree #1

Perform the following AVL Tree commands.

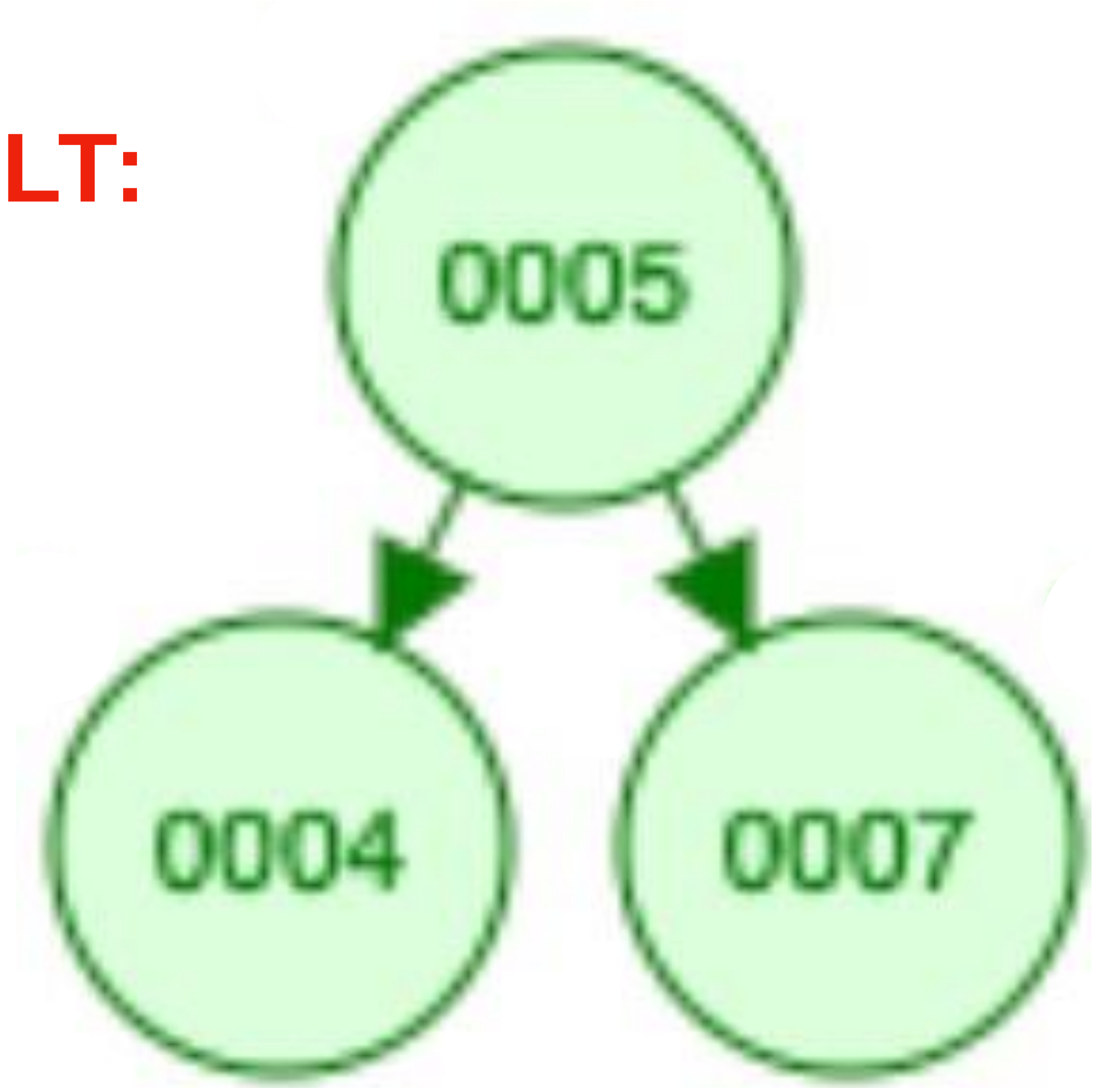
- Insert(5)
- Insert(6)
- Insert(7)
- Insert(2)
- Insert(3)
- Insert(4)
- Delete(6)
- Delete(3)
- Delete(2)

AVL Tree #1

Perform the following AVL Tree commands.

- Insert(5)
- Insert(6)
- Insert(7)
- Insert(2)
- Insert(3)
- Insert(4)
- Delete(6)
- Delete(3)
- Delete(2)

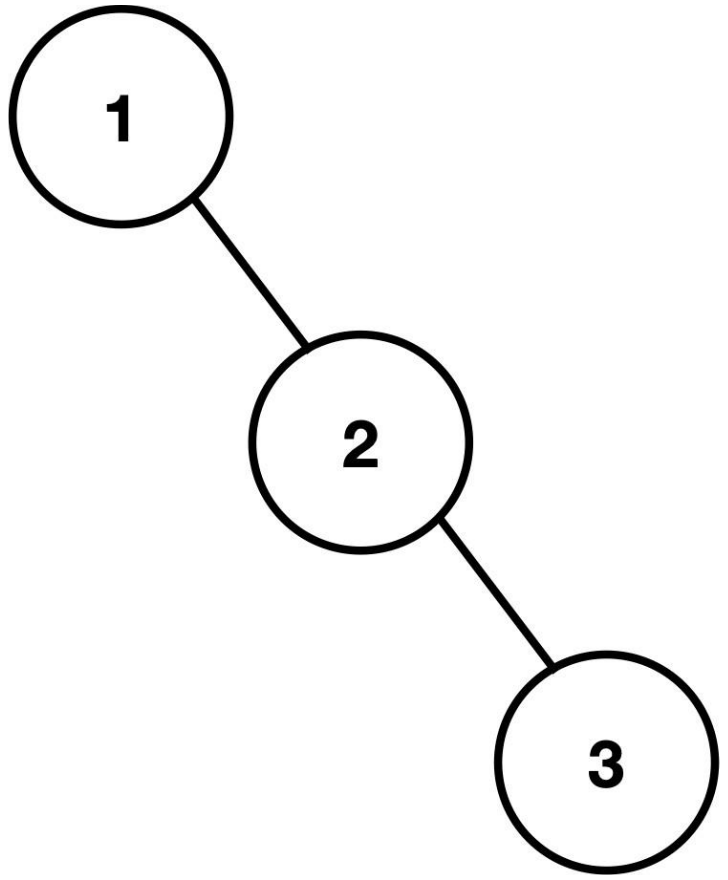
RESULT:

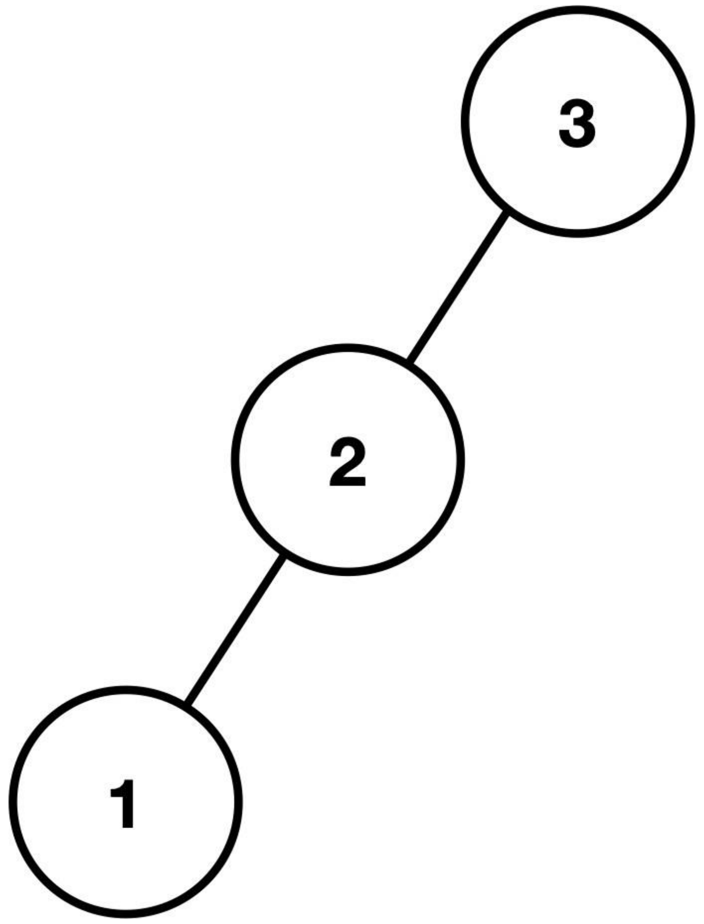


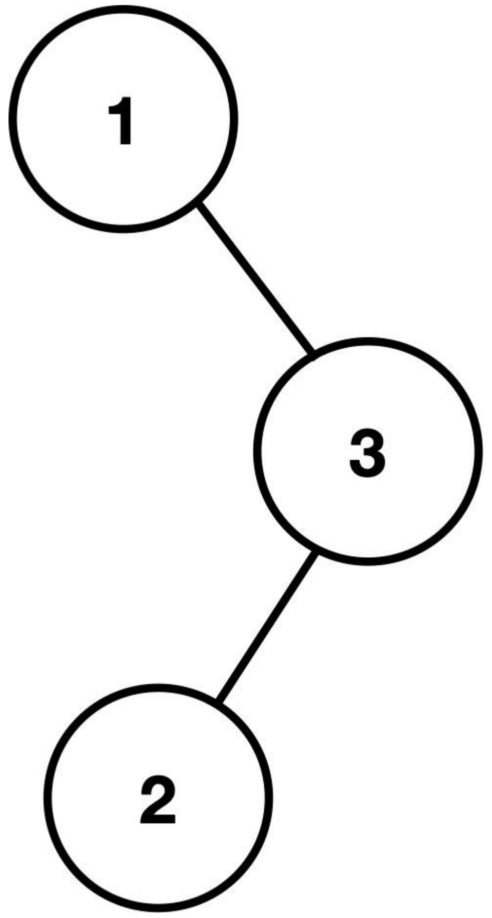
AVL Tree #2

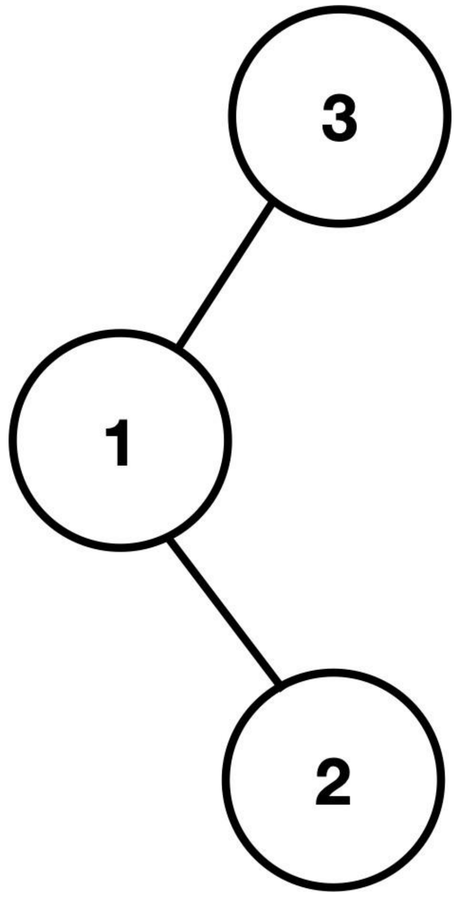
Match the following picture with the rotation that should be performed:

- **Single Right Rotation ____**
- **Single Left Rotation ____**
- **Right Left Rotation ____**
- **Left Right Rotation ____**

a)


b)


c)


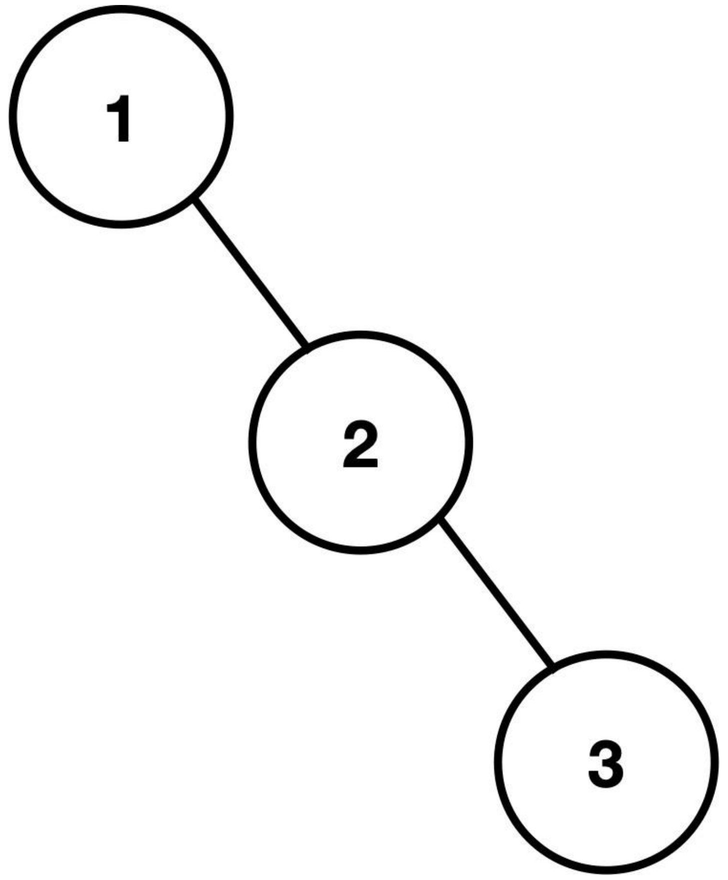
d)


AVL Tree #2

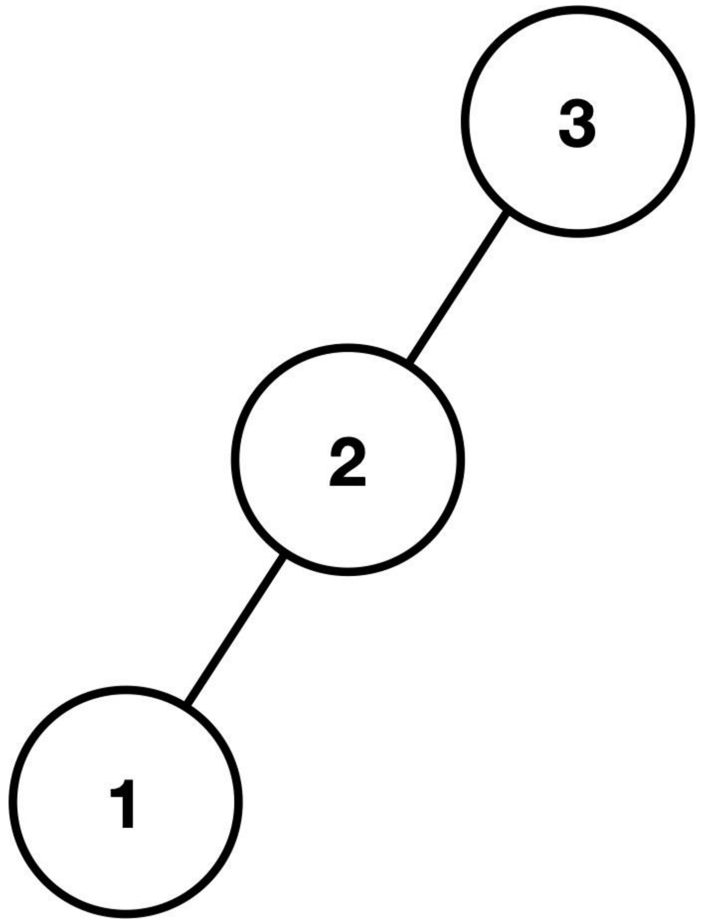
Match the following picture with the rotation that should be performed:

- **Single Right Rotation __b__**
- **Single Left Rotation __a__**
- **Left Right Rotation __c__**
- **Right Left Rotation __d__**

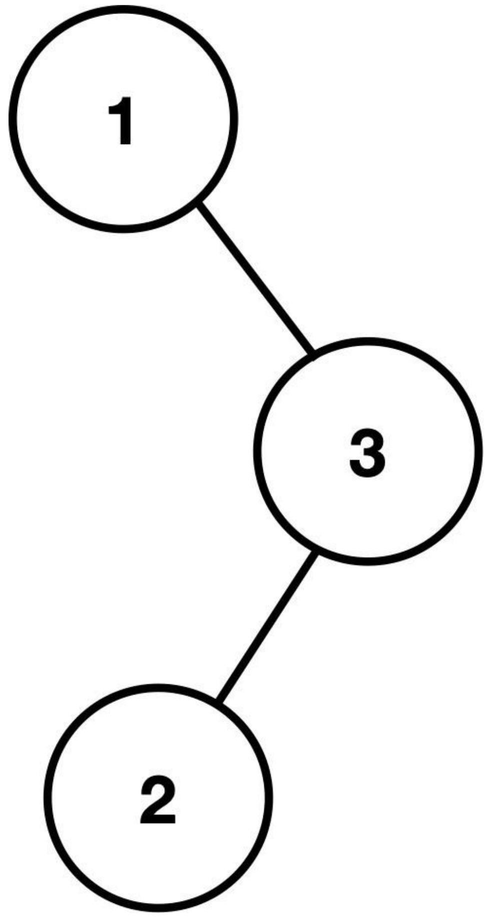
a)



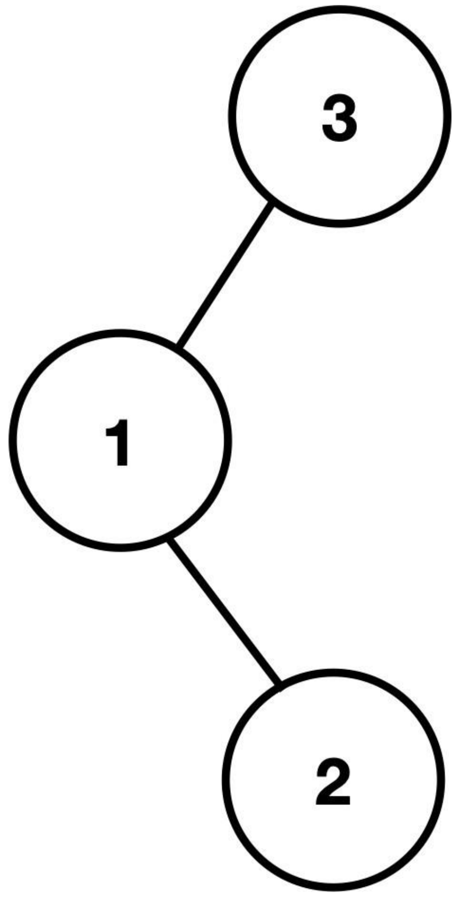
b)



c)



d)



AVL Tree #3

Indicate whether the following statements are true or false

- The acceptable balance factor values for an AVL Tree are: -1, 0, and 1.
- The worst case time complexity for an AVL Tree is O(n).
- The maximum height of an AVL Tree with 7 nodes is 2.
- An AVL Tree is a BST with a self balancing property.

AVL Tree #3

Indicate whether the following statements are true or false

- **The acceptable balance factor values for an AVL Tree are: -1, 0, and 1.**

  true

- **The worst case time complexity for an AVL Tree is O(n).**

  false

- **The maximum height of an AVL Tree with 7 nodes is 2.**

  false

- **An AVL Tree is a BST with a self balancing property.**

  true

Stack & Queue #1

Implement an enqueue function using two stacks. No other data structure is allowed.

```
class Queue{
   private:
      stack<int> s1;
      stack<int> s2;
   public:
      void push(int x);
};

void Queue::enqueue(int x){



}
```

Stack & Queue #1

```
void Queue::enqueue(int x){
   while(!s1.empty()){
      s2.push(s1.top());
      s1.pop();
   }
   s1.push(x);
   while(!s2.empty()){
      s1.push(s2.top());
      s2.pop();
   }
}
```

Stack & Queue #2

Write a function to delete all of the occurrences of a certain value from a queue. You are only allowed one additional variable.

```
void deleteAll(queue<int> &q, int x){







}
```

Stack & Queues #2

```cpp
void deleteAll(queue<int> &q, int x){
  int size = q.size();
  while(size > 0){
    if(q.front() != x)
      q.push(q.front());
    q.pop();
    size--;
  }
}
```

Stack & Queue #3

Write a function that returns the minimum value of a stack and has a time complexity of O(1).

```
class Stack{
   private:
      stack<int> s;
   public:
      void push(int x);
      int getMin();
};

void Stack::push(int x){

}

int Stack::getMin(){

}
```

Stack & Queue #3

```cpp
int Stack::getMin(){
  if(!s.empty())
    return s.top();
  else
    return -1000;
}
```

```cpp
void Stack::push(int x){
  if(s.empty())
    s.push(x);
  else{
    if(x <= s.top())
      s.push(x);
    else{
      stack<int> tempStack;
      while(!s.empty() && x > s.top()){
        tempStack.push(s.top());
        s.pop();
      }
      s.push(x);
      while(!tempStack.empty()){
        s.push(tempStack.top());
        tempStack.pop();
      }
    }
  }
}
```

Stack & Queue #4

Implement a function to find the n-th smallest element in an array using priority queue. You must implement both this function and enqueue function for priority queue.

```
void enqueue(int value){}

int nthSmallest(int arr[], int size, int n){}
```

Stack & Queue #4

```
int nthSmallest(int arr[], int size, int n) {
  priority_queue q;
  for (int i=0; i<size; i++) {
    q.enqueue(arr[i]);
  }
  for (int i=1; i<n; i++) {
    q.dequeue();
  }
  return q.front();
}
```

```
void enqueue(int value) {
    node* temp = new node();
    temp->data = value;
    temp->next = NULL;

    if (isEmpty()) {
        front=temp;
        rear=temp;
    }
    else {
        node* cu = front;
        node* prev = NULL;
        if (temp->data<cu->data) {
            temp->next = front;
            front = temp;
        }
        else {
            while (cu!=NULL && temp->data>=cu->data) {
                prev = cu;
                cu = cu->next;
            }
            prev->next = temp;
            temp->next = cu;
            if (temp->next==NULL)
                rear = temp;

        }
    }
}
```

Sorting #1

Perform heap sort on the array below. Sort the numbers in ascending order. Show all steps.

{4, 1, 2, 9, 5, 8, 3}

Sorting #1

Perform heap sort on the array below. Sort the numbers in ascending order.

{4, 1, 2, 9, 5, 8, 3} <- heapify
{9, 5, 8, 1, 4, 2, 3} <- swap
{3, 5, 8, 1, 4, 2, 9} <- heapify
{8, 5, 3, 1, 4, 2, 9} <- swap
{2, 5, 3, 1, 4, 8, 9} <- heapify
{5, 4, 3, 1, 2, 8, 9} <- swap
{2, 4, 3, 1, 5, 8, 9} <- heapify
{4, 2, 3, 1, 5, 8, 9} <- swap
{1, 2, 3, 4, 5, 8, 9} <- heapify
{3, 2, 1, 4, 5, 8, 9} <- swap
{1, 2, 3, 4, 5, 8, 9} <- heapify
{2, 1, 3, 4, 5, 8, 9} <- swap
{1, 2, 3, 4, 5, 8, 9}

Sorting #2

Implement a sorting function that have the time complexity as follow

Best: O(nlog(n))

Average: O(nlog(n))

Worse: O(n^2)

Sorting #2

Quick Sort:

Write the code for quick sort below as well, look at Exam 2 Basics-Code for the code.

Sorting #3

Describe the worse case scenario for quick sort.

Sorting #3

Worse case O(n^2) When the pivot is at the beginning, or the end of the array and the array is already sorted

Sorting #4

Implement a function that check if an array is a min heap. Return true if the array represent a min heap, return false otherwise.

Sorting #4

```
bool isMinHeap(int arr[], int size) {
    for (int i = 0; i < (size - 2) / 2; i++)
        if (arr[i] > arr[2 * i + 1] || arr[i] > arr[2 * i +
2])
            return false;
    return true;
}
```