COSC 2436: Final Exam Review

Final Exam Topics

- Hashing
- Binary Search Tree (BST)
- AVL Tree
- Graphs
- Stacks & Queues
- Sorting

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

Write the function for quadratic probing.

```
void quadraticProbing(int table[], int tableSize, int x){




}
```

Hashing #2

Write the function for quadratic probing.

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

Hashing #3

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

```
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   }
```

Hashing #3

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 #4

Match the following hash function with its correct description.

- **Direct Hashing _____**
- **Linear Probing _____**
- **Quadratic Probing _____**
- **Separate Chaining _____**
- **Double Hashing _____**

a) table never fills up

b) data is overwritten during collision

c) clustering can occur

d) uses two hash functions

e) probes by i^2 if collision is found

Hashing #4

Match the following hash function with its correct description.

- **Direct Hashing __b__**
- **Linear Probing __c___**
- **Quadratic Probing __e___**
- **Separate Chaining __a___**
- **Double Hashing __d___**

a) table never fills up

b) data is overwritten during collision

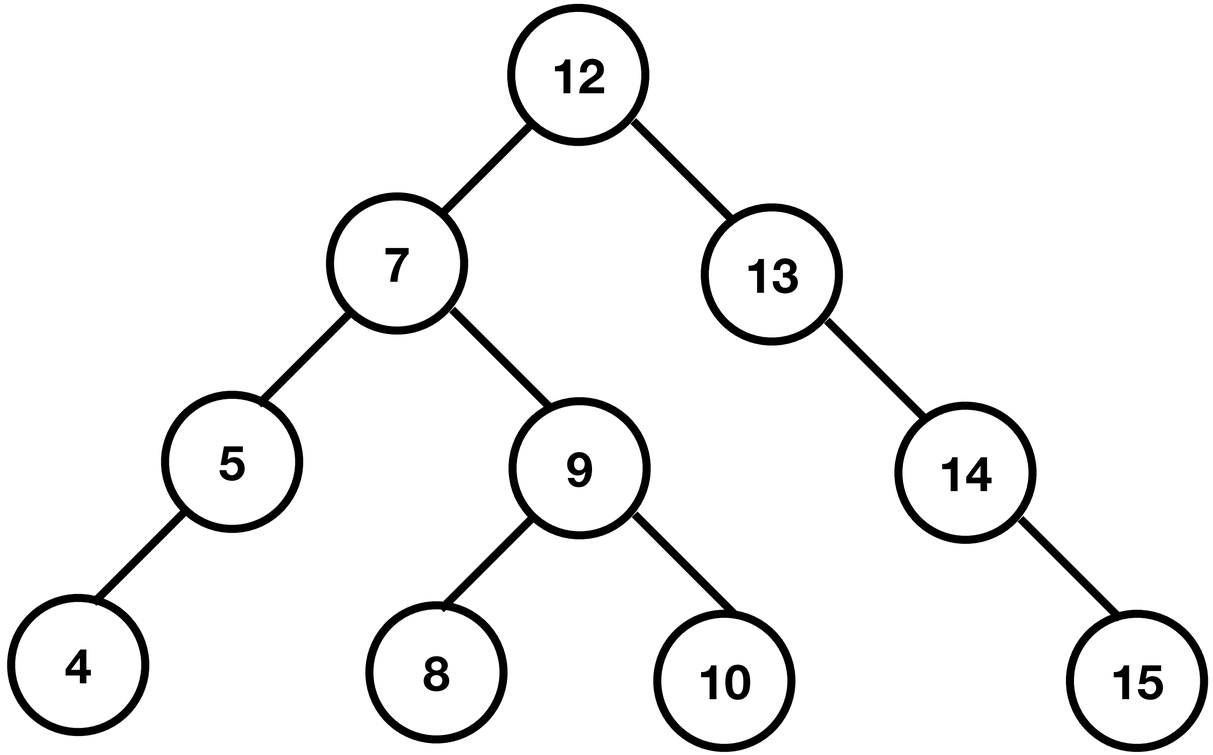
c) clustering can occur

d) uses two hash functions

e) probes by i^2 if collision is found

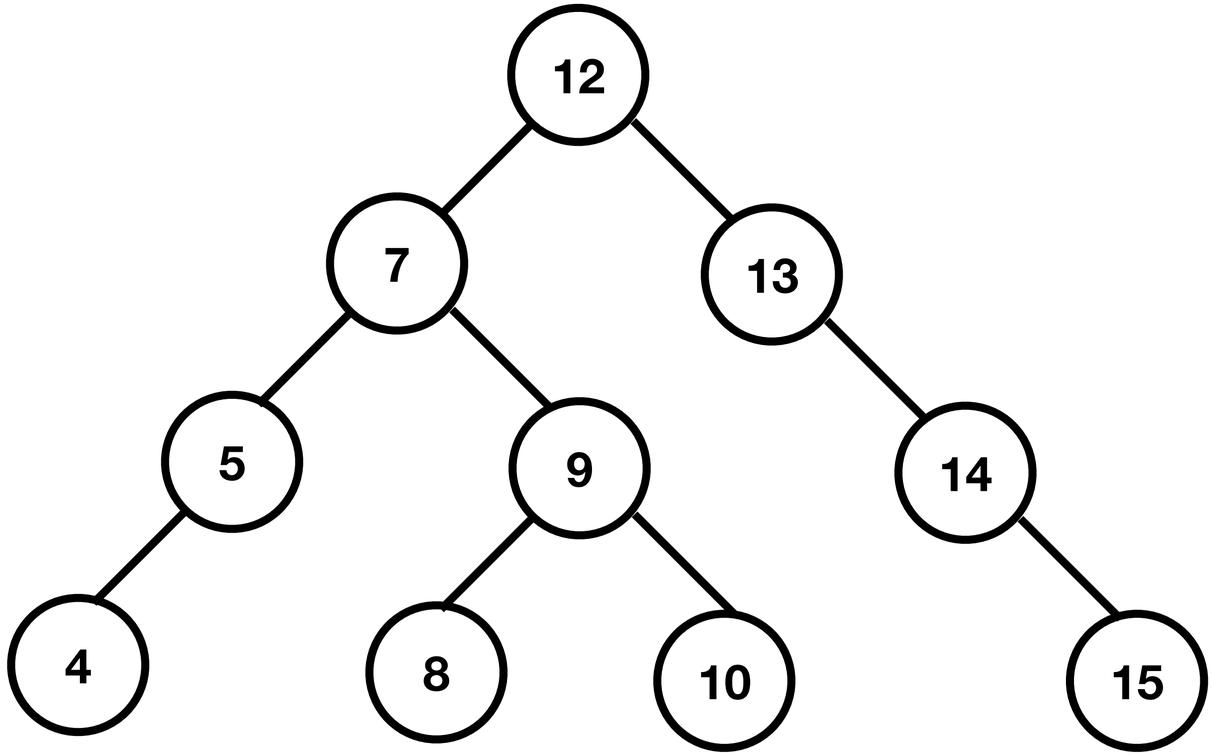
BST #1

Perform preorder traversal on the following BST.



BST #1

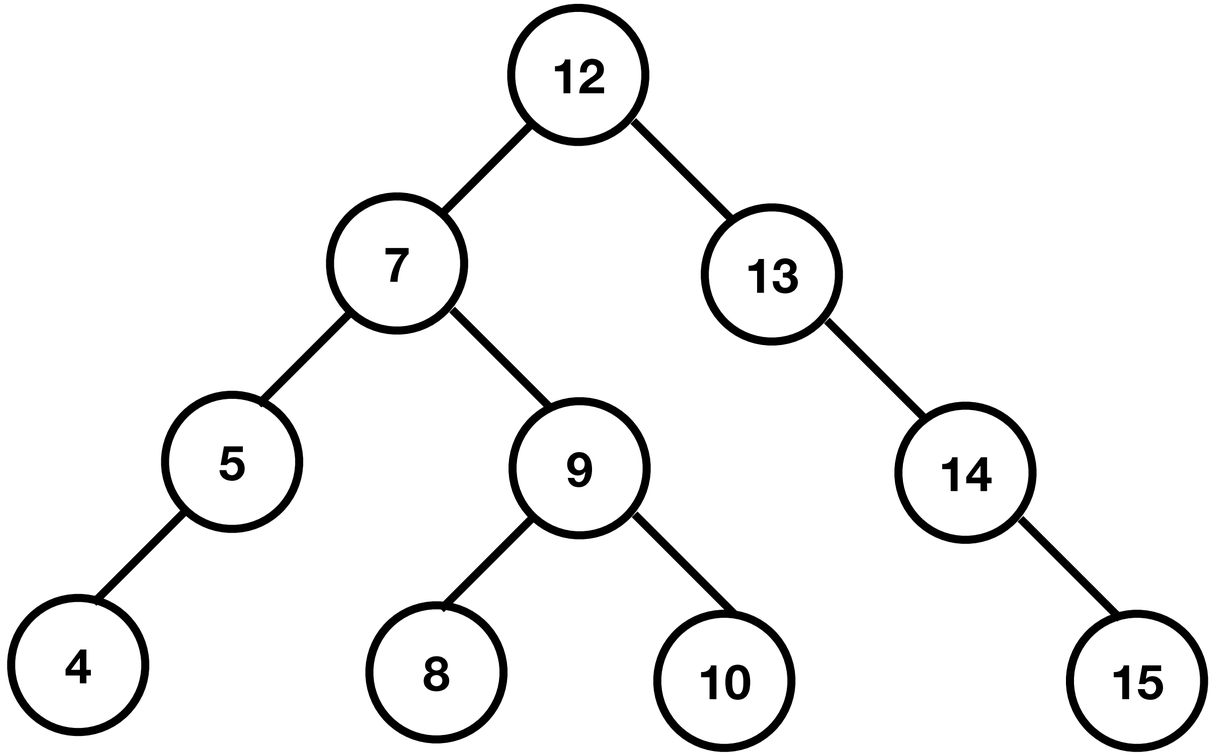
Perform preorder traversal on the following BST.



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 NOT 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 NOT 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.

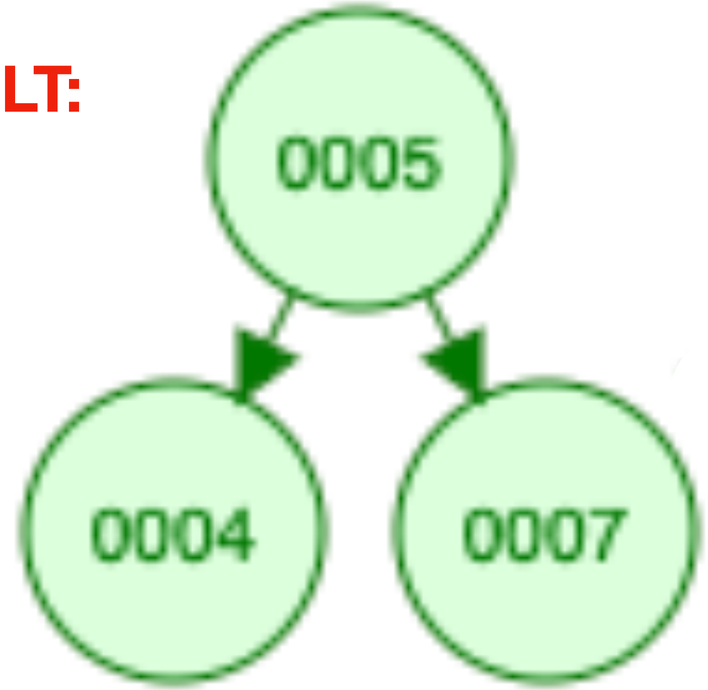
- **Insert(5)**
- **Insert(6)**
- **Insert(7)**
- **Insert(2)**
- **Insert(3)**
- **Insert(4)**
- **Delete(6)**
- **Delete(3)**
- **Delete(2)**

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:

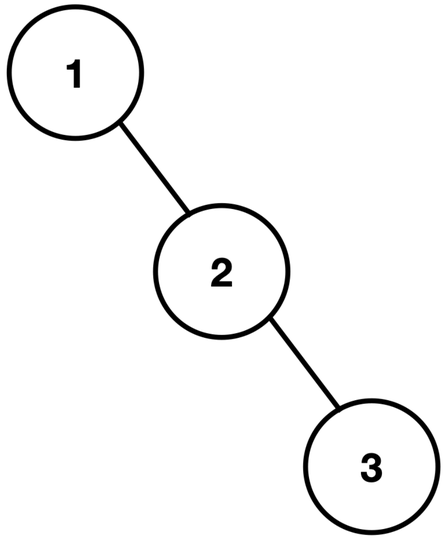


AVL Tree #2

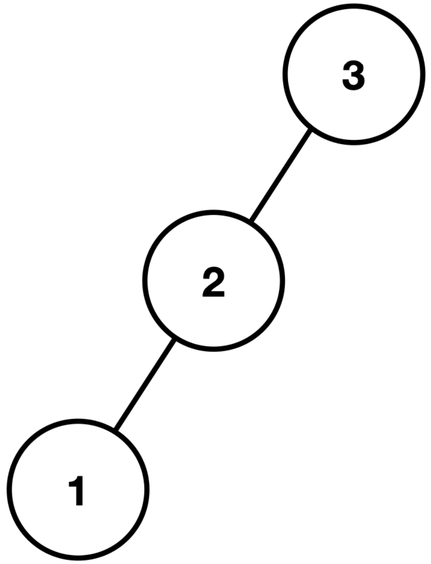
Match the following picture with the rotation that should be performed:

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

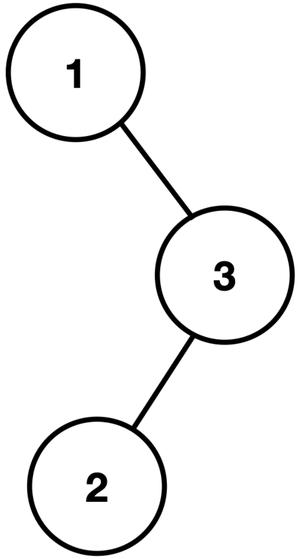
a)



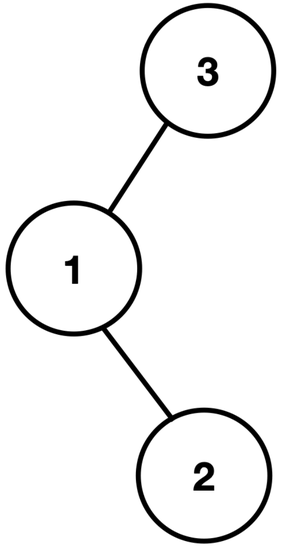
b)



c)



d)

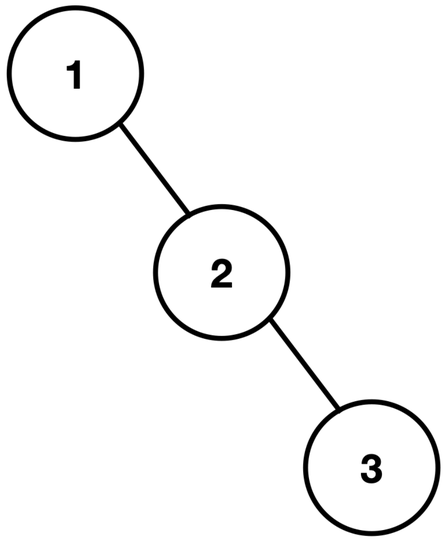


AVL Tree #2

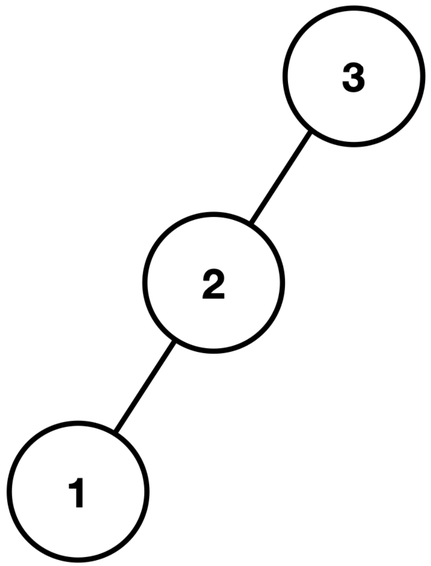
Match the following picture with the rotation that should be performed:

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

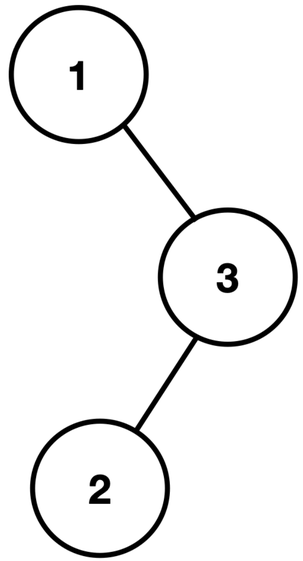
a)



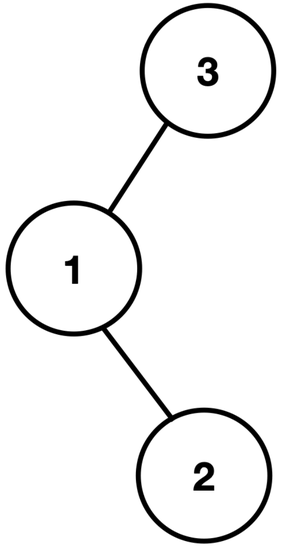
b)



c)



d)



AVL Tree #3

Write the function to perform a single left rotation on a given node.

```
node *singleLeftRotation(node *n){




}
```

AVL Tree #3

Write the function to perform a single left rotation on a given node.

```
11 ▼ node *singleLeftRotation(node *n){
12       node *newParent = n->right;
13       node *newRight = newParent->left;
14       newParent->left = n;
15       n->right = newRight;
16       return newParent;
17   }
```

AVL Tree #4

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 #4

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).**

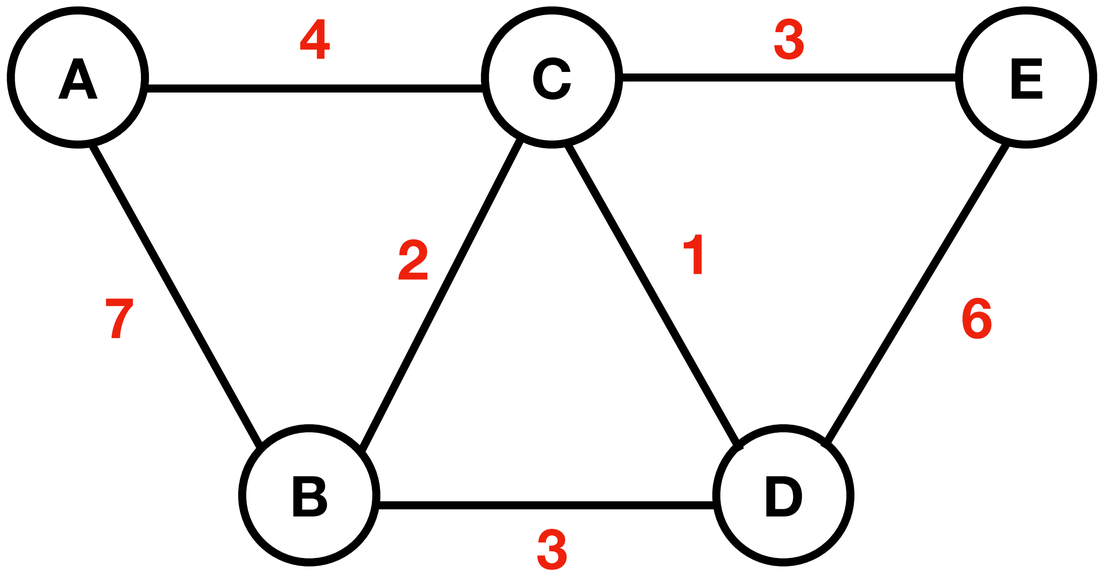
  false
- **The maximum height of an AVL Tree with 7 nodes is 2.**

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

  true

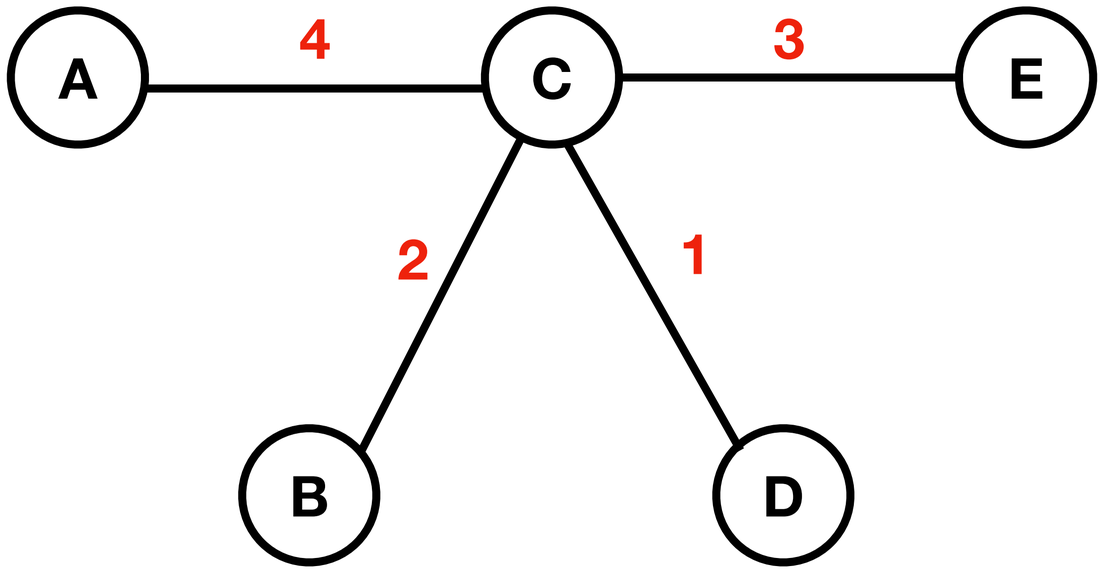
Graphs #1

Find the MST of the following graph using Prim's Algorithm. Start from vertex B.



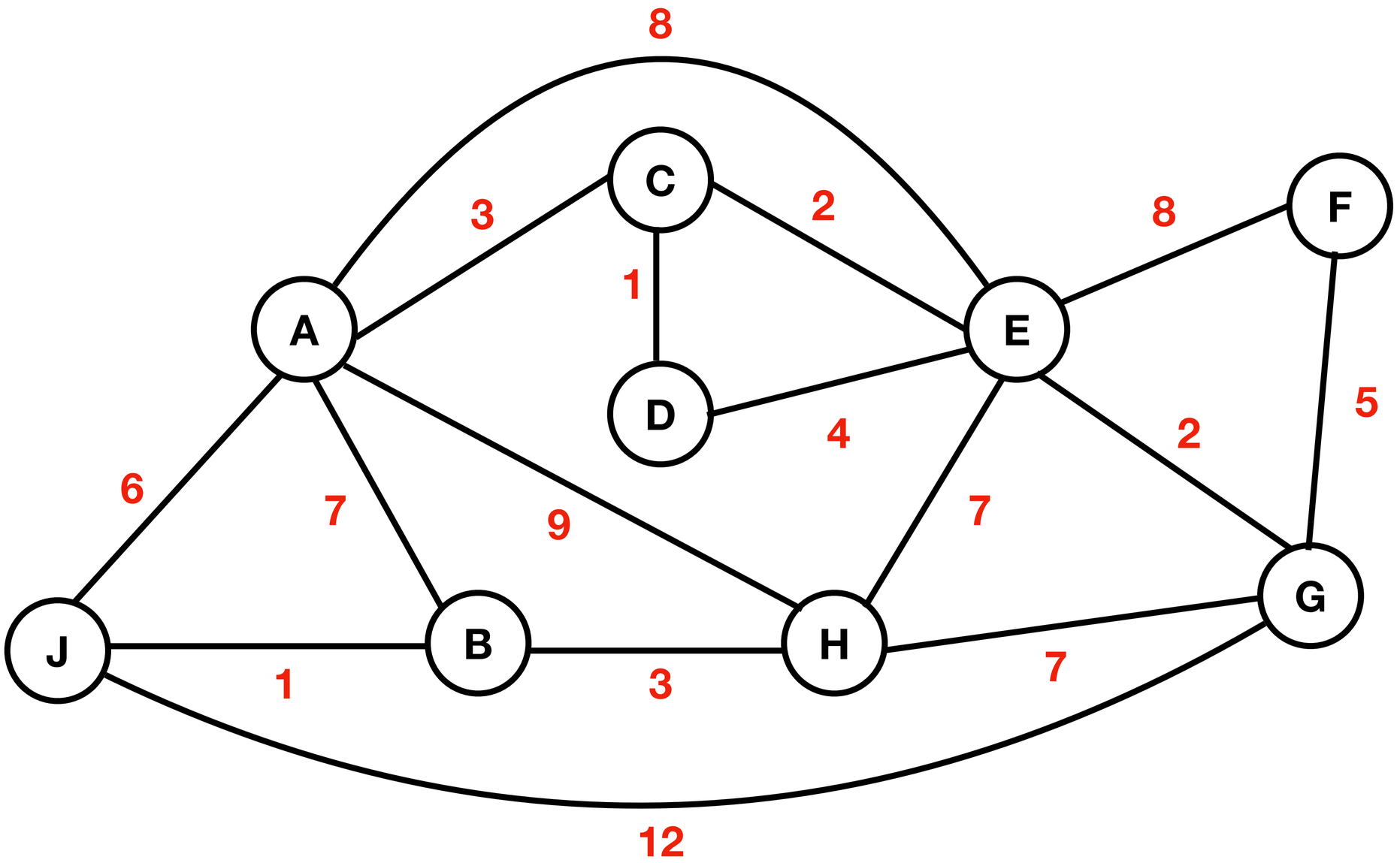
Graphs #1

Find the MST of the following graph using Prim's Algorithm. Start from vertex B.



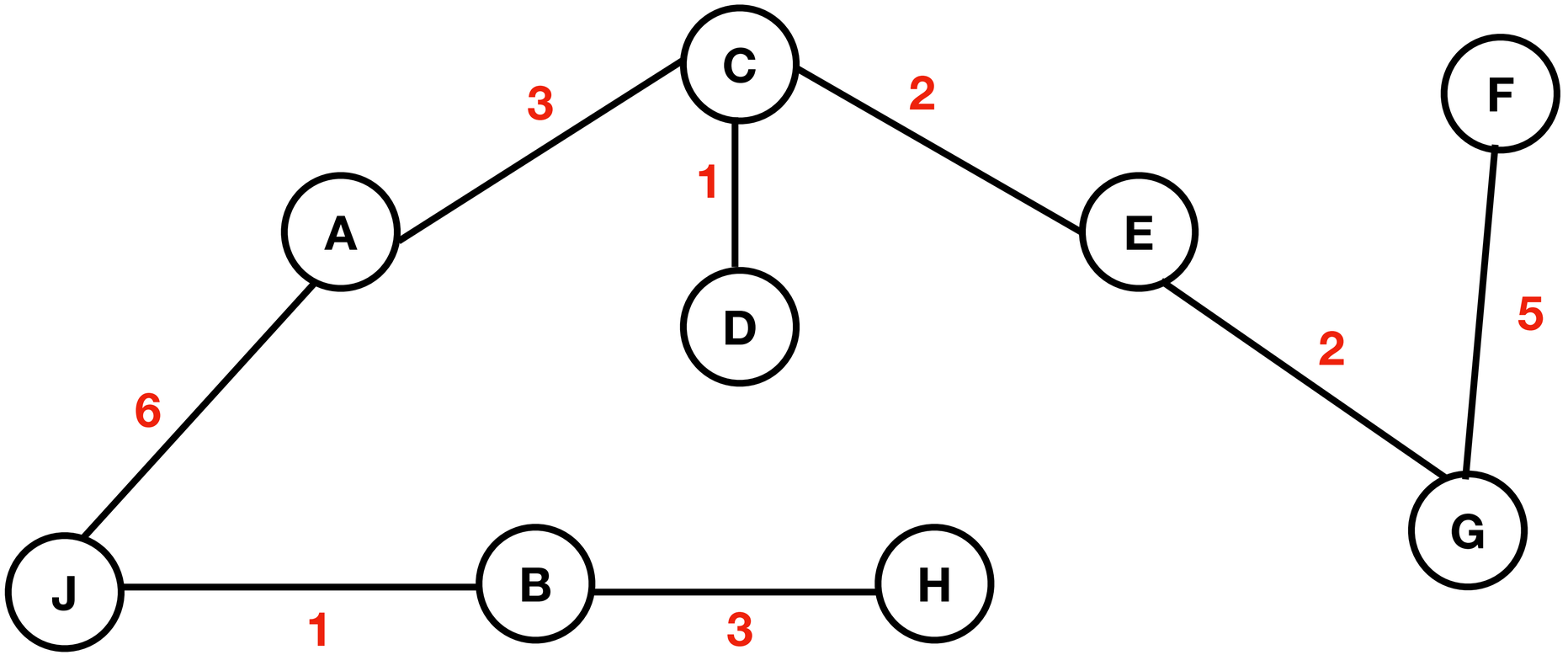
Graphs #2

Find the MST of the following graph using Kruskal's Algorithm.



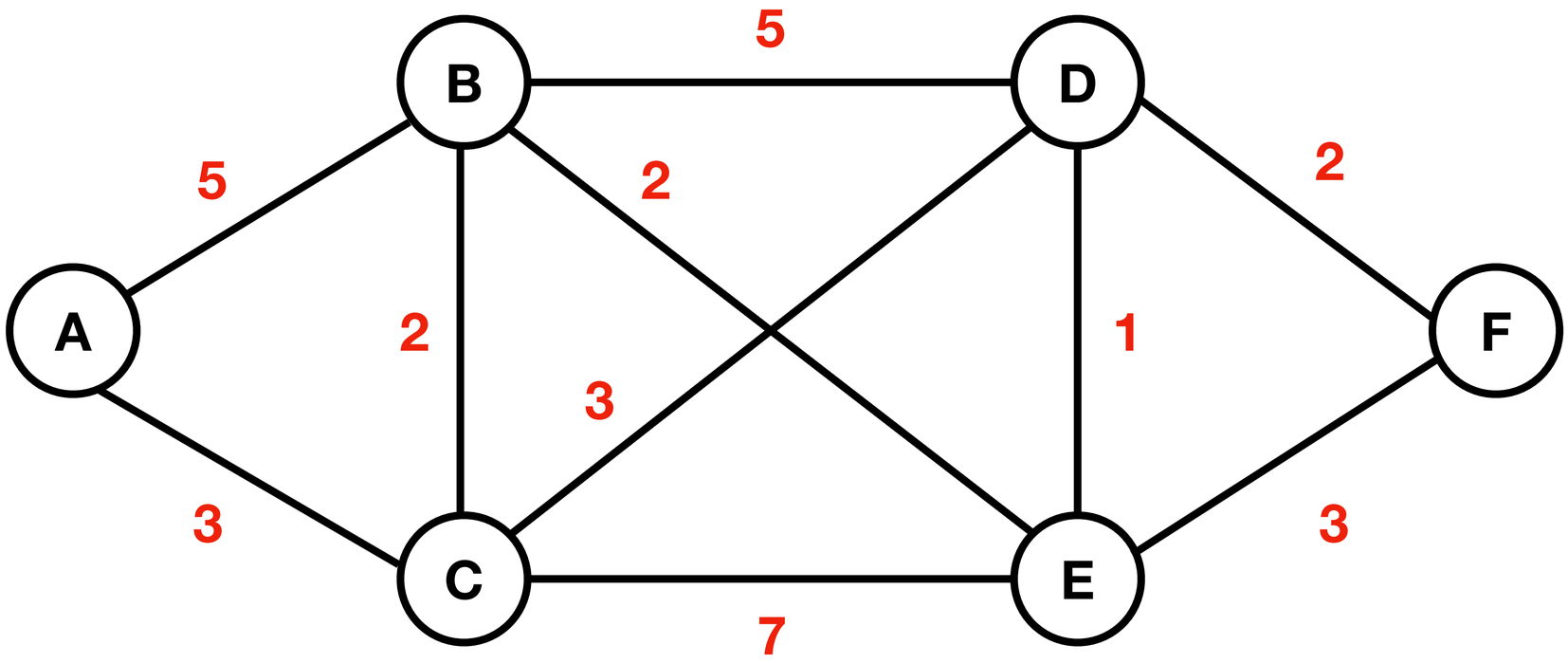
Graphs #2

Find the MST of the following graph using Kruskal's Algorithm.



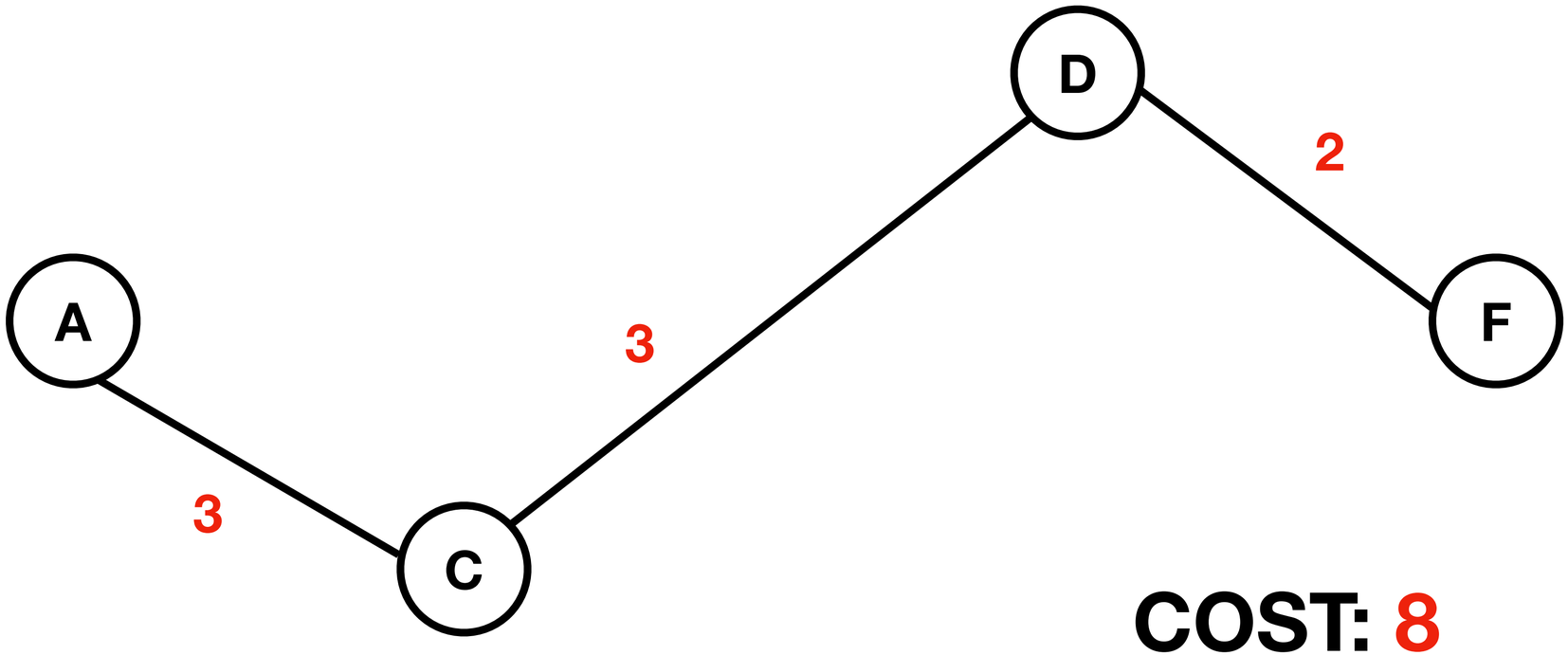
Graphs #3

Find the shortest path from vertex A to vertex F with minimum cost.



Graphs #3

Find the shortest path from vertex A to vertex F with minimum cost.



COST: 8

Graphs #4

Write DFS for an adjacency matrix.

```
void DFS(int **graph, int source, int n){









}
```

Graphs #4

```
void DFS(int **graph, int source, int n){
  stack<int> s;
  bool *visited = new bool[n];
  for(int i = 0; i < n; i++)
    visited[i] = false;
  s.push(source);
  while(!s.empty()){
    int v = s.top();
    s.pop();
    if(!visited[v]){
      cout << v << " ";
      visited[v] = true;
    }
    for(int i = 0; i < n; i++){
      if(graph[v][i] != 0 && !visited[i])
        s.push(i);
    }
  }
  cout << endl;
  delete [] visited;
}
```

Graphs #5

Is the following unweighted graph directed or undirected? Explain.

0	1	0	1	0	0	1
1	0	1	0	1	0	0
0	1	0	0	1	0	0
1	0	0	0	0	0	0
0	1	1	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	0	0

Graphs #5

Is the following unweighted graph directed or undirected? Explain.

Directed

0	1	0	1	0	0	1
1	0	1	0	1	0	0
0	1	0	0	1	0	0
1	0	0	0	0	0	0
0	1	1	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	0	0

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

```
 7 ▼ void deleteAll(queue<int> &q, int x){
 8       int size = q.size();
 9 ▼     while(size > 0){
10           if(q.front() != x)
11               q.push(q.front());
12           q.pop();
13           size--;
14       }
15   }
```

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

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

```
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();
      }
    }
  }
}
```

Sorting #1

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

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

Sorting #1

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

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

Sorting #2

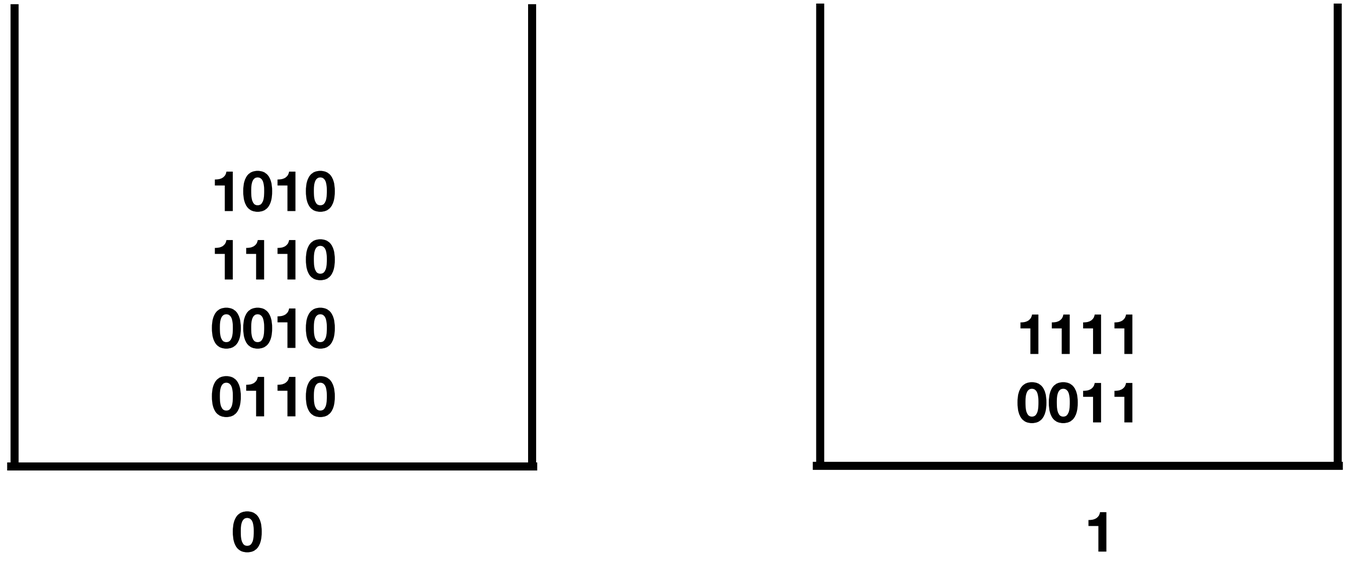
Perform bucket sort on the array of binary numbers below.

{0110, 0010, 0011, 1111, 1110, 1010}

Sorting #2

Perform bucket sort on the array of binary numbers below.

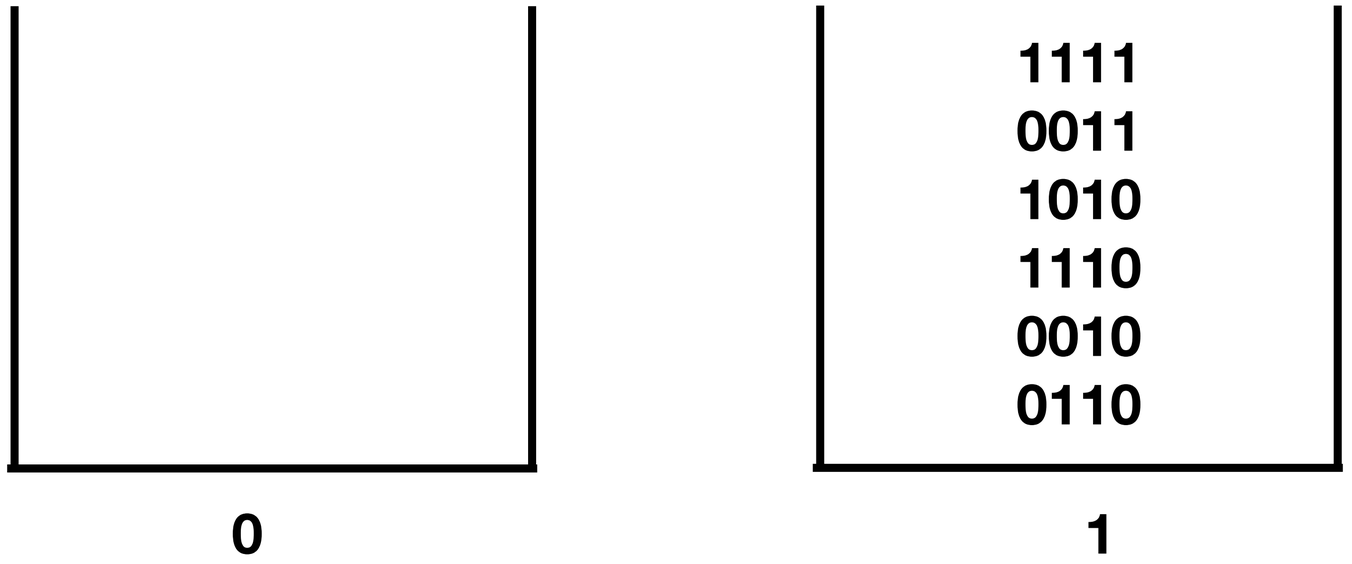
{0110, 0010, 0011, 1111, 1110, 1010}



0110, 0010, 1110, 1010, 0011, 1111

Sorting #2

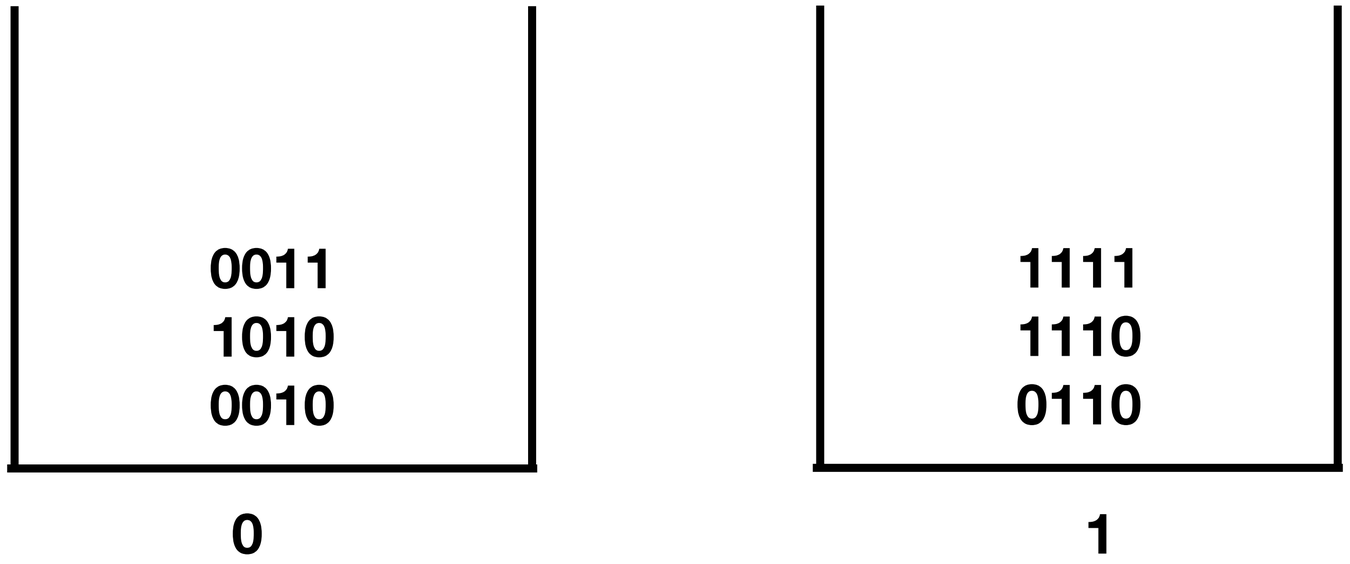
0110, 0010, 1110, 1010, 0011, 1111



0110, 0010, 1110, 1010, 0011, 1111

Sorting #2

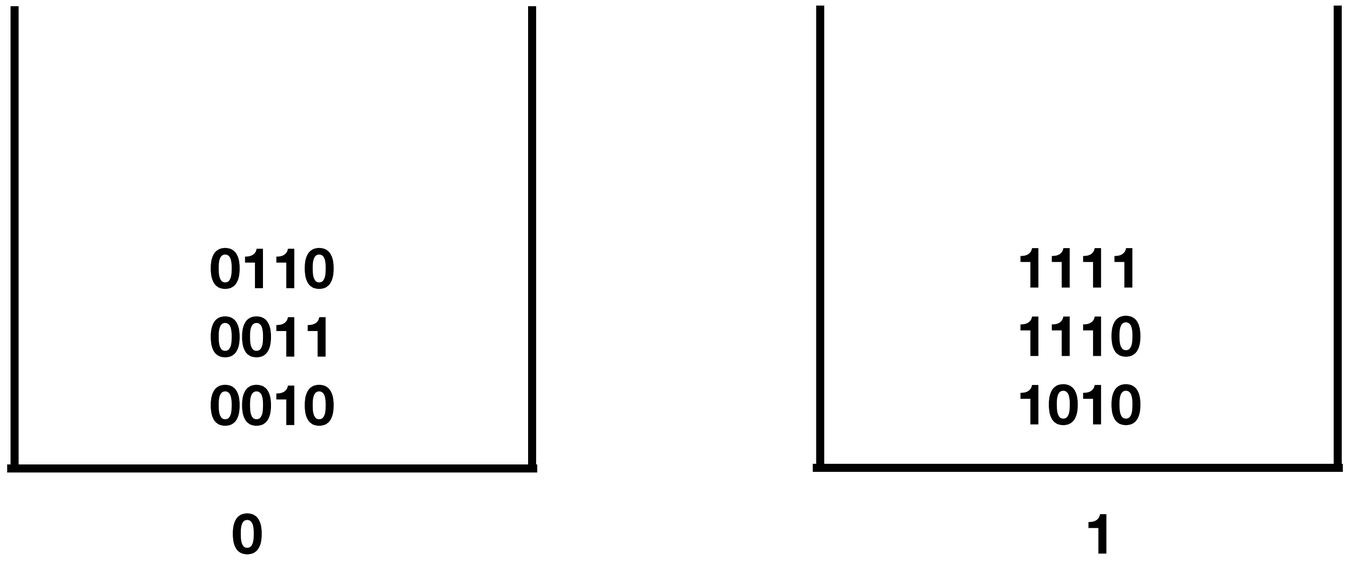
0110, 0010, 1110, 1010, 0011, 1111



0010, 1010, 0011, 0110, 1110, 1111

Sorting #2

0010, 1010, 0011, 0110, 1110, 1111



FINAL ANSWER: 0010, 0011, 0110, 1010, 1110, 1111

Hashing Review

- **Know how to trace and code the following hashing methods:**
  - **Quadratic Probing**
  - **Linear Probing**
  - **Direct Hashing**
  - **Double Hashing**
  - **Separate Chaining**

BST & AVL Tree Review

- **Know how to perform inOrder, reverseOrder, postOrder, and preOrder traversals**
- **Know how to do basic BST & AVL Tree functions (insert, search, height, balance, delete, etc.)**
- **Know how to trace insert and delete for AVL tree**
- **Know the rotations for AVL Tree**

Graph Review

- **Know how to trace Prims, Kruskals and Dijkstras**
- **Know how to code BFS and DFS**
- **Know the basics of adjacency matrix and adjacency list**

Stack & Queue Review

- Know the basic stack and queue functions (push(), pop(), top(), front(), size(), empty(), etc.)

Sorting Review

- Know how to trace the different sorting algorithms covered this semester

Final Remarks

- **You will have 3 hours to take the final exam**
- **Bring your own pencil(s) to take the final exam**
- **For more practice, review the documents found on BlackBoard**