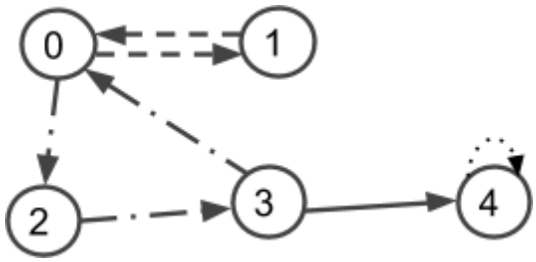
Homework 7

1. Please write a function findMaxInOutDegree(self) function to find the node with maximum outdegree and the node with max indegree (for adjacency matrix and for adjaceny list) for directed graph.



Output:

Max in Degree nodes: 0, 4

Max ou Degree nodes: 0

2. Please write a function findCommonAdjcentNodes(self, node 1, node2) function to find the nodes that are adjacent to node1 and node 2 for an undirected graph (for adjacency matrix and for adjaceny list).

3. Given a directed graph G with N nodes. You can use any graph representation (Adjacency matrix or Adjacency list) to solve the following problems.

a Write a function **printSinkNodes** that takes in a graph and finds outthe sink nodes of the given graph G.

Function Signature:

printSinkNodes(self);

4. Simulate the behavior of a hash table as described and implementedin lecture. Assume the following:

   i. The hash table array has an initial **capacity/size** of **5.** Capacity/size is the number of slots present in the array that is used to buildthe hashtable.
   ii. As an element is added to the hashtable, the **count** of the hashtable increases. count is defined as the total number of elements present in the hashtable at any given moment.
   iii. The hash table uses **separate chaining** to resolve collisions.
   iv. The **hash function** returns the absolute value of the element **mod** the

capacity of the hash table

v. **rehashing** occurs after addition of an element where the load factor is ≥ **0.6** and doubles the capacity of the hash table. Loadfactor is defined as the number of number of elements added sofar divided by the capacity of the whole array

Fill in the diagram in Table: 1 to show the ***final state*** of the hash table after the following operations are performed. Leave a box empty if an array element is unused. **Also write the count, capacity, and loadfactor of the final hash table.** Write the load factor in 0.x format, such as 0.5 or 0.75.

/*

The function *put(self, a)* takes in a hashTable **ht** and an integer **a**. It inserts the integer a into the hashTable **ht** after calculating hashing function. The function *delete(self, a)* takes in a hashTable ht and an integer **a**. It deletes the integer **a** from the hashTable ht.The function *find(self, a)* takes an integer a.It returns true/false based on whether the integer **a** is present in the hashTable **ht**.

*/

```
ht = new hashTable()

ht.put(18)
ht.add(75)
ht.add(25)
ht.delete(18)
ht.remove(75)
ht.add(18)
ht.add(15)
add(h, 159)
if (ht.find(25)):
  ht.add(24)
    remove(18)
else:

    ht.add(18)

if (ht.size > 7): add(h,
    24)

remove(h,75)
```

hashTable[0]	

Homework 7

hashTable[1]	
hashTable[2]	
hashTable[3]	
hashTable[4]	
hashTable[5]	
hashTable[6]	
hashTable[7]	
hashTable[8]	
hashTable[9]	
hashTable[10]	
size	
capacity	
loadfactor	

Table: 1

Homework 7

5. Simulate the behavior of a hash table as described and implementedin lecture. Assume the following:

- The hash table array has an initial capacity/size of **5.** Capacity/size is the number of slots present in the array that is used to build the hashtable.
- As an element is added to the hashtable the **count** of the hashtableincreases. count is defined as the total number of elements
  present in the hashtable at any given moment.
- The hash table uses **linear probing** to resolve collisions.
- The **hash function** returns the absolute value of the element modthe capacity of the hash table.
- **rehashing** occurs at the *end* of the add function when the load factor is ≥ **0.8** and doubles the capacity of the hash table. Loadfactor is defined as the number of number of elements added so far divided by the capacity of the whole array

Fill in the diagram in Table: 2 to show the ***final state*** of the hash
table after the following operations are performed. Leave a box empty if an array element is unused. **Also write the size, capacity, and loadfactor of the final hash table.** Write the load factor in 0.x format, such as 0.5 or 0.75.

/*

The function *put(self, a)* takes in a hashTable **ht** and an integer **a**. It inserts the integer a into the hashTable **ht** after calculating hashing function. The function *delete(self, a)* takes in a hashTable ht and an integer **a**. It deletes the integer **a** from the

hashTable ht.The function *find(self, a)* takes an integer a.It returns true/false based on whether the integer **a** is present in the hashTable **ht**.

*/

```
ht = hashTable()
ht.add(25);
ht.add(993);
ht.add(83);
add(598);
ht.delete(34);
```

Homework 7

```
ht.add(18);
if (ht.find(993)): {
        ht.add(66);
}
ht.add(75);
```

hashTable[0]	
hashTable[1]	
hashTable[2]	
hashTable[3]	
hashTable[4]	
hashTable[5]	
hashTable[6]	
hashTable[7]	
hashTable[8]	
hashTable[9]	
hashTable[10]	
size	
capacity	
loadfactor	

Table: 2

Homework 7

6. Suppose, you have a hash table of size 11. you have a hash function h(k) = (2k+5) mod 11. The following values are inserted in the order: 12, 44,13, 88, 23, 94, 11, 39, 20, 16 and 5. The size of the table is 11.
    - **a.** Write down the code for the above hash function that will return the index of the key in the hash table.
    - **b.** You are using linear probing to avoid collision, draw the hash table with index and calculate the maximum prob, total prob, average prob and load factor of the hash table.
    - **c.** Using the same hash function but this time the collisions are handled by double hashing with the secondary hash function h2(k) = 7 – (k mod 7). Draw the hash table with index and calculate the maximum prob, total prob, average prob and load factor of the hash table.