Hashing

1. Consider a simple hash function as "key mod 10" and sequence of [69, 48, 139, 648, 109, 75], implement and draw the result hash table handling the collision by (a) Quadratic Probing ($f(i) = i^2$) (b) double hashing (with the second function "7 – (key mod 7)")
   a. Quadratic Probing:
   hash(x) = x % 10 $f(i) = i^2$ $h_i(x) = (hash(x) + f(i)) \% ts$

   69 % 10 = 9
   48 % 10 = 8
   139 % 10 = 9
   ↘ $h_1(139) = (139 \% 10 + 1^2) \% 10$ = 0
   648 % 10 = 8
   ↘ $h_1(648) = (648 \% 10 + 1^2) \% 10$ = 9
   ↘ $h_2(648) = (648 \% 10 + 2^2) \% 10$ = 2
   109 % 10 = 9
   ↘ $h_1(109) = (109 \% 10 + 1^2) \% 10$ = 0
   ↘ $h_2(109) = (109 \% 10 + 2^2) \% 10$ = 3
   75 mod 10 = 5

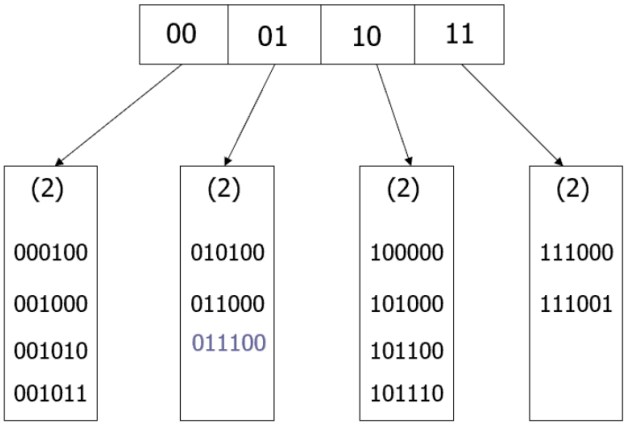
0	1	2	3	4	5	6	7	8	9
139		648	109		75			48	69

   b. Double Hashing: $hash_2(x) = 7 – (x \% 7)$
   hash(x) = x % 10 $f(i) = i * hash_2(x)$ $h_i(x) = (hash(x) + i*hash_2(x)) \% 10$

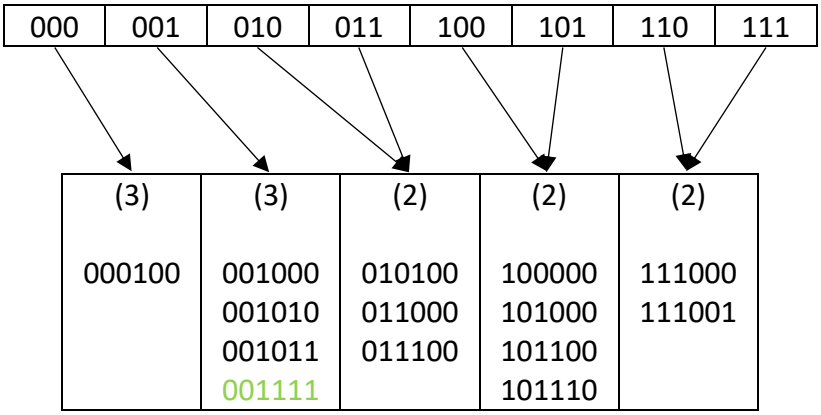
   69 % 10 = 9
   48 % 10 = 8
   139 % 10 = 9
   ↘ $h_1(139) = (139 + 1*(7 – (139 \% 7)) \% 10$ = 0
   648 % 10 = 8
   ↘ $h_1(648) = (648 + 1*(7 – (648 \% 7)) \% 10$ = 1
   109 % 10 = 9
   ↘ $h_1(109) = (109 + 1*(7 – (109 \% 7)) \% 10$ = 2
   75 % 10 = 5

0	1	2	3	4	5	6	7	8	9
139	648	109			75			48	69

2. Consider the extendible hash table:



Draw the table after inserting 001111. (This is a hand calculation question)



3. Given an array of integers, find two numbers such that they add up to a specific target number in O(n) Time, return the indices and return [-1 -1] if not found.
    1. Conceptual Description
        a. User inputs an array (list) of numbers and a target. For each number in the array, see if its compliment exists in the hash table, then return both of their indices; the number and its compliment would sum to the target. Otherwise, append the number and its index to the hash table, using the number as the key. If no number has a compliment, then return [-1, -1].
    2. Code
        a.

```
def sumTwo(array, target):
    hashTable = {}
    for index, num in enumerate(array):
        if (target - num) in hashTable:
            return [hashTable[target-num], index]
        hashTable[num] = index
    return [-1, -1]

numArray1 = [2, 7, 11, 15]
numArray2 = [15, 2, 7, 11]
target = 9

print("Return:", sumTwo(numArray1, target))
print("Return:", sumTwo(numArray2, target))
```

        b.

        Return: [0, 1]

        Return: [1, 2]
    3. Complexity Analysis
        a. O(*n*); let *n* be the number of elements in the input array. Each access to the hash table requires O(1) time.

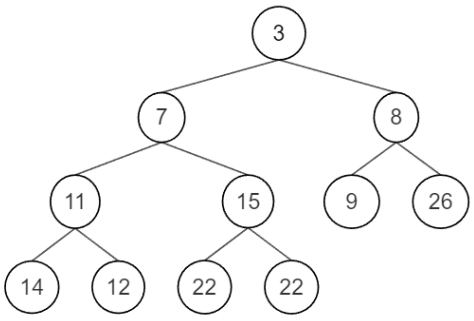
Heaps

4. Given numbers 8, 12, 9, 7, 22, 3, 26, 14, 11, 15, 22, (1) write a function to build a min heap, (2) draw your result min heap.
    1. Conceptual Description
        a. The minHeapSort() function takes a user's array and creates a minheap. Following the steps outlined in class, the function first selects the number at the end of the array, then compares the value of the number to its parent, and finally swaps the position of the number and its parent if the new node has a lower value than its parent. The steps repeat until the heap properties are upheld in the array.
    2. Code
        a.

```
def minHeapify(numArray, arrLength, indexNum):
    indexMin = indexNum
    indexChildLeft = 2 * indexNum + 1
    indexChildRight = 2 * indexNum + 2

    if indexChildLeft < arrLength and numArray[indexMin] >
numArray[indexChildLeft]:
        indexMin = indexChildLeft

    if indexChildRight < arrLength and numArray[indexMin] >
numArray[indexChildRight]:
        indexMin = indexChildRight

    if indexMin != indexNum:
        numArray[indexNum], numArray[indexMin] =
numArray[indexMin], numArray[indexNum]
        minHeapify(numArray, arrLength, indexMin)

numArray = [8, 12, 9, 7, 22, 3, 26, 14, 11, 15, 22]

print("List:", numArray)
for numIndex in range(len(numArray), -1, -1):
    minHeapify(numArray, len(numArray), numIndex)
print("Heap:", numArray)
```

        b.

        List: [8, 12, 9, 7, 22, 3, 26, 14, 11, 15, 22]
        Heap: [3, 7, 8, 11, 15, 9, 26, 14, 12, 22, 22]
    3. Min Heap Drawing



5. Given an integer array, find the top k largest numbers in it. Write the description, code, and time complexity analysis of the algorithm.
    1. Conceptual Description
        a. The largestNumbers() function takes a user's array and *n* number of integers, and returns a list of the *n* largest integers in descending order. To do so, the program first creates a maxheap using the user's array. Following the steps outlined in class, the function first selects the number at the end of the array, then compares the value of the number to its parent, and finally swaps the position of the number and its parent if the new node has a lower value than its parent. The steps repeat until the heap properties are upheld in the array. The function then swaps the first and last values in the array. Then, the last number is popped and appended to another list of the largest numbers. This entire process is repeated *n* times. Finally, the list of largest numbers is returned.
    2. Code
        a.

```
def maxHeapify(numArray, arrLength, indexNum):
    indexMin = indexNum
    indexChildLeft = 2 * indexNum + 1
    indexChildRight = 2 * indexNum + 2

    if indexChildLeft < arrLength and numArray[indexMin] <
numArray[indexChildLeft]:
        indexMin = indexChildLeft

    if indexChildRight < arrLength and numArray[indexMin] <
numArray[indexChildRight]:
        indexMin = indexChildRight

    if indexMin != indexNum:
        numArray[indexNum], numArray[indexMin] =
numArray[indexMin], numArray[indexNum]
        maxHeapify(numArray, arrLength, indexMin)

def largestNumbers(numArray, amtNums):
    arrLargest = []

    for i in range(amtNums):
        for numIndex in range(len(numArray), -1, -1):
            maxHeapify(numArray, len(numArray), numIndex)

        numArray[0], numArray[-1] = numArray[-1], numArray[0]
        arrLargest.append(numArray.pop())

    return arrLargest

numArray = [3, 10, 1000, -99, 4, 100]
numTargets = 6

print(largestNumbers(numArray, numTargets))
```

5. Given an integer array, find the top k largest numbers in it. Write the description, code, and time complexity analysis of the algorithm.
    2. Code (continued)
        b.

            [1000, 100, 10]
    3. Complexity Analysis
        a. O(klogn); the time complexity of maxHeapify is O(logn), where *n* is the length of the array. With largestNumbers() function, maxHeapify() is called *k* times, where *k* is the number of largest integers requested. Overall, the time complexity is O(klogn).