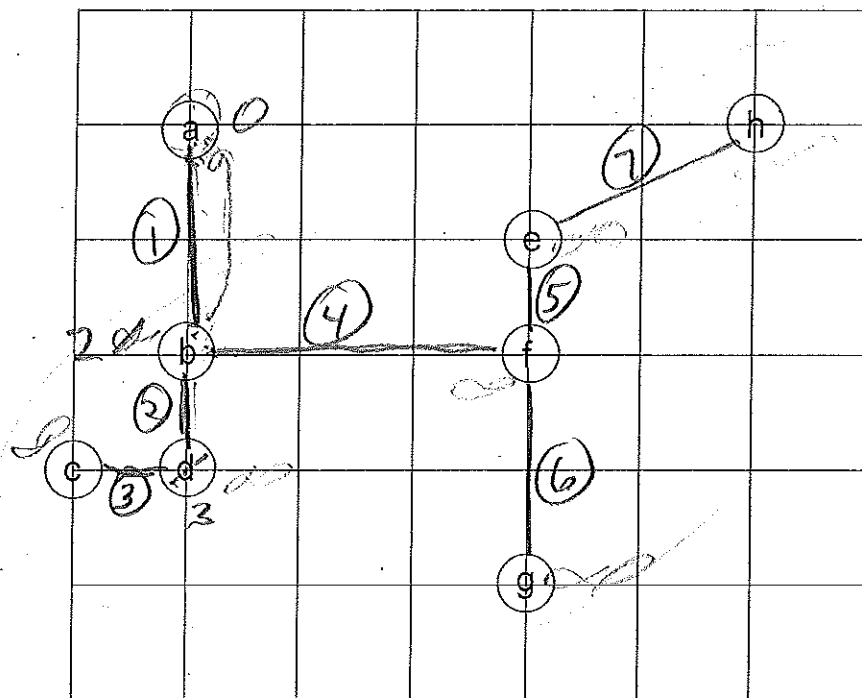


1. Suppose you had a map of settlements on the planet X  
(Assume edges could connect all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

- a) Assuming we start at city "a" which city would you connect first? <sup>b</sup> Why this city?

because "b" is closest city to "a"

- b) What city would you connect next to expand your partial road network? <sup>d</sup> because it is closest to partial road system.

- c) What would be some characteristics of the resulting "graph" after all the cities are connected?

no cycles - "tree"

min. "spanning tree" (MST)

- d) Does your algorithm come up with the overall best (globally optimal) result?

Yes - Prim's alg. gives optimal MST

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree?

*add next closest vertex not in the subgraph to any vertex in the subgraph*

b) Consider the textbook's Prim's Algorithm code (Listing 7.12 p. 346) which is incorrect.

```
def prim(G, start):
    pq = PriorityQueue() BinHeap()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin().getValue()
        for nextVert in currentVert.getConnections():
            newCost = currentVert.getWeight(nextVert) \
                + currentVert.getDistance()
            if nextVert in pq and newCost < nextVert.getDistance():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
            pq.decreaseKey(nextVert, newCost)
```

*PriorityQueueEntry(0, nextVert)*

*PriorityQueueEntry(v.getDistance(), v)*

*PriorityQueueEntry(newCost, nextVert)*

c) What is wrong with the code? (Fix the above code.)

By storing tuples we always compare by the first item when comparing two tuples. This works fine for > and < operations of percolateUp and percolateDown of BinHeap, but not for == operation needed by --contains--.

3. To avoid "massive" changes to the binHeap class, it can store PriorityQueueEntry objects. *to compare by second item in tuple*

```
class PriorityQueueEntry:
    def __init__(self, x, y):
        self.key = x
        self.val = y
    def getKey(self):
        return self.key
    def getValue(self):
        return self.val
    def setValue(self, newValue):
        self.val = newValue
```

```
def __lt__(self, other):
    return self.key < other.key
def __gt__(self, other):
    return self.key > other.key
def __eq__(self, other):
    return self.val == other.val
def __hash__(self):
    return self.key
```

a) Update the above Prim's algorithm code to use PriorityQueueEntry objects.

b) Why do the \_\_lt\_\_ and \_\_gt\_\_ methods compare key attributes, but \_\_eq\_\_ compare val attributes?

*used by percolateUp and percolateDown to compare priorities*

*used by --contains-- to compare vertices.*

- c) When used for Prim's algorithm what type of objects are the vals compared by `__eq__`? *vertices*
- d) What changes to the Graph and Vertex classes need to be made? *none*

- e) Complete the `__contains__` and `decreaseKey` methods.

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2] < self.heapList[i*2+1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def isEmpty(self):
        return self.currentSize == 0

    def size(self):
        return self.currentSize

    def __str__(self):
        return str(self.heapList[1:])
```

```
def __contains__(self, value):
```

*for index in range(1, self.currentSize + 1):*  
*if value == self.heapList[index]:*  
*self.foundIndex = index*  
*return True*  
*return False*

```
def decreaseKey(self, decreasedValue):
    """Precondition: decreasedValue
    in heap already"""
```

*if decreasedValue in self:*  
*self.heapList[self.foundIndex]*  
*= decreasedValue*  
*self.percUp(self.foundIndex)*  
*else:*  
*raise - - -*

*(see attached for BinHeap complete code + Prim's complete code)*

binheap.py

```
# Bradley N. Miller, David L. Ranum
# Introduction to Data Structures and Algorithms in Python
# Copyright 2005
# Modified by Mark Fienup 2016 to include a __contains__ and decreaseKey
method
import unittest

# this heap takes key value pairs, we will assume that the keys are integers
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
        foundIndex = 0

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i//2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
```

binheap.py

```
self.percUp(self.currentSize)
```

```
def delMin(self):  
    retval = self.heapList[1]  
    self.heapList[1] = self.heapList[self.currentSize]  
    self.currentSize = self.currentSize - 1  
    self.heapList.pop()  
    self.percDown(1)  
    return retval
```

```
def isEmpty(self):  
    return self.currentSize == 0
```

```
def size(self):  
    return self.currentSize
```

```
def __str__(self):  
    return str(self.heapList[1:])
```

```
def __contains__(self, value):  
    for index in range(len(self.heapList)-1,0,-1):  
        if self.heapList[index] == value:  
            self.foundIndex = index  
            return True  
    return False
```

```
def decreaseKey(self, decreasedValue):  
    """Precondition: decreasedValue in heap already"""  
    if not decreasedValue in self:  
        raise(ValueError, "decreaseKey value must be in heap")  
  
    self.heapList[self.foundIndex] = decreasedValue  
    self.percUp(self.foundIndex)
```

```
class PriorityQueueEntry:  
    def __init__(self, x, y):  
        self.key = x  
        self.val = y
```

```
def getKey(self):  
    return self.key
```

```
def getValue(self):  
    return self.val
```

```
def setValue(self, newValue):  
    self.val = newValue
```

```
def __lt__(self, other):
    return self.key < other.key

def __gt__(self, other):
    return self.key > other.key

def __eq__(self, other):
    return self.val == other.val

def __hash__(self):
    return self.key
```

```
class TestBinHeap(unittest.TestCase):
```

```
    def setUp(self):
        self.theHeap = BinHeap()
        self.theHeap.insert(PriorityQueueEntry(5, 'a'))

        self.theHeap.insert(PriorityQueueEntry(9, 'd'))
        self.theHeap.insert(PriorityQueueEntry(1, 'x'))
        self.theHeap.insert(PriorityQueueEntry(2, 'y'))
        self.theHeap.insert(PriorityQueueEntry(3, 'z'))

    def testInsert(self):
        assert self.theHeap.currentSize == 5

    def testDelmin(self):
        assert self.theHeap.delMin().getValue() == 'x'
        assert self.theHeap.delMin().getValue() == 'y'
        assert self.theHeap.delMin().getValue() == 'z'
        assert self.theHeap.delMin().getValue() == 'a'

    def testMixed(self):
        myHeap = BinHeap()
        myHeap.insert(9)
        myHeap.insert(1)
        myHeap.insert(5)
        assert myHeap.delMin() == 1
        myHeap.insert(2)
        myHeap.insert(7)
        assert myHeap.delMin() == 2
        assert myHeap.delMin() == 5

    def testDupes(self):
        myHeap = BinHeap()
        myHeap.insert(9)
        myHeap.insert(1)
        myHeap.insert(8)
        myHeap.insert(1)
        assert myHeap.currentSize == 4
        assert myHeap.delMin() == 1
        assert myHeap.delMin() == 1
```

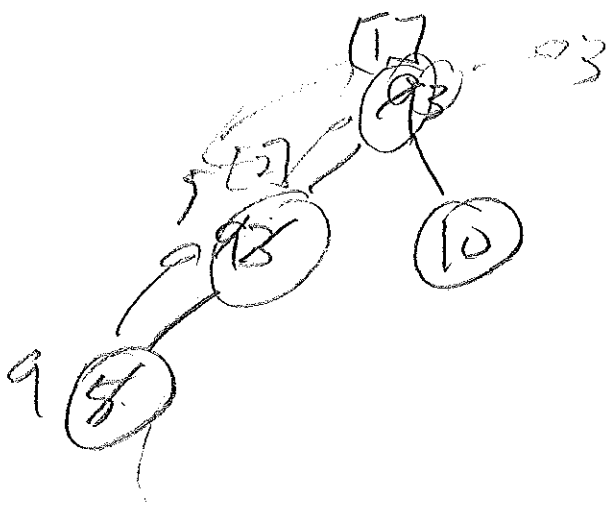
binheap.py

```
assert myHeap.delMin() == 8
```

```
def testBuildHeap(self):  
    myHeap = BinHeap()  
    myHeap.buildHeap([9,5,6,2,3])  
    f = myHeap.delMin()  
    #print("f = ", f)  
    assert f == 2  
    assert myHeap.delMin() == 3  
    assert myHeap.delMin() == 5  
    assert myHeap.delMin() == 6  
    assert myHeap.delMin() == 9
```

```
if __name__ == '__main__':  
    ## d = {}  
    ## d[PriorityQueueEntry(1,'z')] = 10  
    ## unittest.main()  
    suite = unittest.makeSuite(TestBinHeap)  
    unittest.TextTestRunner().run(suite)
```

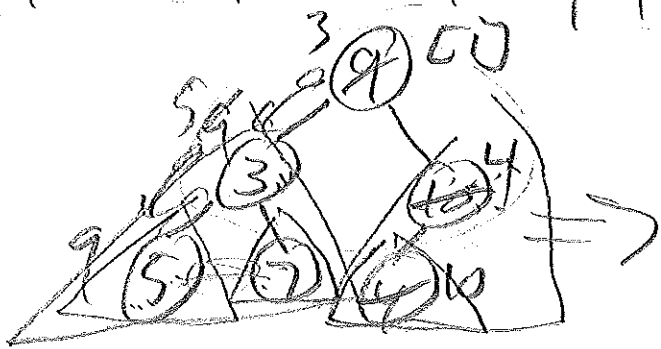
0 1 2 3  
 (9 3 10 5 7 4]



0 1 2  
 9 3

~~Array~~  
 Build Heap

0 1 2 3 4 5 6  
 9 3 10 5 7 4



9