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

![Graph of settlements](image)

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? Why this city?

add road from @ to B since B is closed

b) What city would you connect next to expand your partial road network? Why?

it closest to the partial road system.

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

- connected
- no cycles - "tree"
- minimum spanning tree

MST

d) Does your algorithm come up with the overall best (globally optimal) result? Yes
Think of leaves as little mini-heaps of one item so min. has to be at root

PerceDown non-leaves starting at "last" non-leave and walking to root at index 1. Each non-leaf is perceDown into heap(s) below it to make bigger heap.
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 that's closest to partial MST**

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

```python
def prim(G,start):
    pq = PriorityQueue()  # dequeues
    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()  # get value()
        for nextVert in currentVert.getConnections():
            newCost = currentVert.getCost() + nextVert.getDistance()
            if newCost < nextVert.getCost():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(nextVert,newCost)
```

c) What is wrong with the code? (Fix the above code.)
- Using heap operation names with PriorityQueue up to be BinHeap
- Using tuples in the heap will not work well since two tuples are always compared on the first items (here priorities), OK for `perClear` and `perDown`, but not OK for `contains` and `decreaseKey`
- New Cost calculated incorrectly.

3. To avoid “massive” changes to the BinHeap class, it can store PriorityQueue objects:

```python
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
```

```python
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 hash(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?
- __lt__ and __gt__ need to compare key/priority in perClear, perDown, but __eq__ needs to compare vertexes when finding a vertex
from graph import Graph
from vertex import Vertex
from linked_queue import LinkedQueue
from binheap import BinHeap
from binheap import PriorityQueueEntry
import sys

def dijkstra(aGraph, start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in aGraph])
    while not pq.isEmpt():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() +
            currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance(newDist)
                nextVert.setPred(currentVert)
                pq.decreaseKey(PriorityQueueEntry(nextVert, newDist))

def prim(G, start):
    pq = BinHeap()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([PriorityQueueEntry(v.getDistance(), v) for v in G])
    while not pq.isEmpt():
        currentVert = pq.delMin().getValue()
        if currentVert.getPred() != None:
            print("Prim: edge", currentVert.getPred().getId(), "to",
            currentVert.getId())
        for nextVert in currentVert.getConnections():
            newCost = currentVert.getWeight(nextVert)
            if PriorityQueueEntry(0, nextVert) in pq and
            newCost < nextVert.getDistance():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(PriorityQueueEntry(newCost, nextVert))
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
def delMin(self):
    retval = self.heapList[1]
    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)
c) When used for Prim's algorithm what type of objects are the \texttt{vals} compared by \_\_eq\_? \texttt{Vertex} objects

d) What changes to the Graph and Vertex classes need to be made? \textit{None}

c) Complete the \_\_contains\_ and \texttt{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 -= 1
    def percDown(self, i):
        while (i * 2 + 1) <= 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 + 1
        else:
            if self.heapList[i*2] > self.heapList[i*2+1]:
                return i * 2 + 1
            else:
                return i * 2 + 2
    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.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:])

\texttt{def \_\_contains\_(self, value):}
\texttt{for index in range(1, self.currentSize + 1)}
\texttt{if value \texttt{==} self.heapList[index]}:
\texttt{return True}
\texttt{return False}
\texttt{def decreaseKey(self, decreasedValue):}
\texttt{"""Precondition: decreasedValue in heap already"""}
\texttt{if decreasedValue \texttt{in} self.heapList[0]}:
\texttt{return False}
\"""O(\texttt{n})\"""