1.class Node:

def \_\_init\_\_(self, key=None, next=None):

self.key = key

self.next = next

self.heapList=[0]

self.currentSize=0

def percUp(self, i):

while i // 2 >0:

if self.heapList[i] < self.heapList[i // 2]:

s=self.heapList[i // 2]

self.heapList[i // 2] = self.heapList[i]

self.heapList[i] = s

i = i // 2

def insert(self, k):

self.heapList.append(k)

self.currentSize = self.currentSize + 1

self.percUp(self.currentSize)

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 percDown(self, i):

while (i \* 2) <= self.currentSize:

a = self.minChild(i)

if self.heapList[i] > self.heapList[a]:

t = self.heapList[i]

self.heapList[i] = self.heapList[a]

self.heapList[a] = t

i = a

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 \_\_init\_\_(self, key, n=1, left=None, right=None):

self.key = key

self.left = left

self.right = right

self.n = n

def height(self):

pass

def parent(self, i):

if(i== 0):

return("i-0 doesn't have parent.")

else:

return (i - 1) / 2

def leftChild(self, i):

return i \* 2 + 1

def rightChild(self, i):

return i \* 2 + 2

def parent(self,i):

if(i== 0):

return("i-0 doesn't have parent.")

else:

return (i - 1) / 2

def leftChild(self,i):

return i \* 2 + 1

def rightChild(self,i):

return i \* 2 + 2

def create\_tree(self, arr, n, Node, i)

if (2 \* i + 1 < n):

Node.leftchild = new Node(arr[2 \* i + 1]);

left = leftchild<-node

create\_tree(arr, n , leftchild , 2 \* i + 1);

if (2 \* i + 2 < n):

Node. rightchild = new Node(arr[2 \* i + 2]);

right = rightchild<-node

create\_tree(arr, n, rightchild, 2 \*i+ 2);

2. class Node:

def \_\_init\_\_(self, key=None, next=None):

self.key = key

self.next = next

self.heapList=[0]

self.currentSize=0

def percUp(self, i):

while i // 2 >0:

if self.heapList[i] < self.heapList[i // 2]:

s=self.heapList[i // 2]

self.heapList[i // 2] = self.heapList[i]

self.heapList[i] = s

i = i // 2

def insert(self, k):

self.heapList.append(k)

self.currentSize = self.currentSize + 1

self.percUp(self.currentSize)

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 percDown(self, i):

while (i \* 2) <= self.currentSize:

a = self.minChild(i)

if self.heapList[i] > self.heapList[a]:

t = self.heapList[i]

self.heapList[i] = self.heapList[a]

self.heapList[a] = t

i = a

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 buildHeap(self, m):

i = len(m) // 2

self.currentSize = len(m)

self.heapList = [0] +m[:]

while (i>0):

self.percDown(i)

i = i - 1

3. The best case is that the number of newly inserted elements is the largest, no element needs to be moved, and the time complexity is O (1). The worst case is that the newly inserted element is the smallest, all the elements have to be moved, and the total number of moves is n+1, which is O(n). So the minimum priority queue time complexity is O (n).

4. thread\_pool\_high\_prio\_mode = statements

thread\_pool\_high\_prio\_tickets

thread\_pool\_idle\_timeout =30

thread\_pool\_max\_threads =50000

thread\_pool\_oversubscribe =2

thread\_pool\_size

thread\_pool\_stall\_limit = 100