**Priority queues**

1. **A complete binary tree ADT based on a singly linked list**

class Node():  
 # 123456  
 def \_\_init\_\_(self, key=None, next=None):  
 # 123456789  
 self.key = key  
 self.next = next  
  
  
 def show(self, i):  
 # 123456  
 t = i, self.key  
 print("subscript : %-3d number value : %d" % t)  
  
  
# define a complete binary tree class  
class Tree():  
 # 123456789  
 def \_\_init\_\_(self):  
 # 123456789  
 self.head = Node()  
  
 # 123456789  
 def push(self, node): # add a node  
 # 123456，use the tail insert method  
 p = self.head  
 # search the last node  
 while p.next:  
 # 123456  
 p = p.next  
 # put the new node behind the last node  
 p.next = node  
  
 # 12345678  
 def pop(self): # delete the last node  
 # 123456789  
 if not self.head.next:  
 # 123456  
 print("This binary tree is already empty.")  
 return  
  
 pre = self.head;  
 p = self.head.next  
 while p.next: # search the last node  
 # 123456  
 pre = p  
 p = p.next  
 pre.next = None  
 print("The last node was successfully deleted.")  
  
 # 12345678  
 def popLast(self): # delete the last node  
 # 123456789  
 if not self.head.next:  
 # 123456  
 return  
  
 pre = self.head;  
 p = self.head.next  
 while p.next: # search the last node  
 # 123456  
 pre = p  
 p = p.next  
 # After the loop ，p is the last node  
 # pre is the node which in front of p  
 # so to delete p，we only need to empty pre's pointer field  
 pre.next = None  
  
 # 123456789  
 def size(self): # get size  
 # 123456  
 num = 0  
 p = self.head.next  
 while p:  
 # 123456789  
 num += 1  
 p = p.next  
 return num  
  
 # 123456789  
 def get(self, i):  
 # 123456789  
 n = self.size()  
 if i >= n:  
 # 123456789  
 return None  
 p = self.head  
 k = 0  
 while k <= i:  
 # 123456  
 k += 1  
 p = p.next  
 return p  
  
 # show all nodes  
 # 123456  
 def show(self):  
 # 123456  
 k = 0  
 p = self.head.next  
 while p:  
 # 123456  
 p.show(k)  
 k += 1  
 p = p.next  
  
  
 # get the left child  
 def getLeft(self, me):  
 # 123456  
 n = self.size()  
 left = me \* 2 + 1  
 if left >= n:  
 # 123456  
 return -1  
 return left  
  
 # get the right child  
 # 123456789  
 def getRight(self, me):  
 # 123456  
 n = self.size()  
 right = me \* 2 + 2  
 if right >= n:  
 # 123456  
 return -1  
 return right  
  
 # get parents  
 # 123456789  
 def getPar(self, me):  
 # 123456  
 if me == 0:  
 # 123456789  
 return -1  
 if me % 2 == 0:  
 me -= 2  
 else:  
 me -= 1  
 return me // 2

1. **A minimum priority queue based on the complete binary tree above**

mytree = Tree()  
  
  
# add a node  
def push():  
 # 123456789  
 while True:  
 # 123456789  
 num = input("Please enter the integer you want to increase (Enter a negative number to exit increment): ")  
 num = int(num)  
 if num < 0:  
 # 123456789  
 return  
 node = Node(num)  
 mytree.push(node)  
 print("A node is successfully added \n")  
  
  
  
def search():  
 # 123456  
 num = input("Please enter the index you want to find : ")  
 num = int(num)  
 mytree.search(num)  
  
  
 # a complete binary tree  
def BTMenu():  
 # 123456  
 print("1 Add several nodes")  
 print("2 Delete the last node")  
 print("3 Display a complete binary tree")  
 print("4 Look for a certain index")  
 print("0 Exit the binary tree module")  
 ss = "01234"  
 srr = list(ss)  
 while True:  
 # 123456789  
 flag = input("Please enter a selection of 0 to 4: ")  
 if flag in srr:  
 # 123456  
 return int(flag)  
  
  
def BTModel():  
 # 123456789  
 global mytree  
 while True:  
 # 123456  
 flag = BTMenu()  
 if flag == 1:  
 # 123456  
 push()  
  
 if flag == 2:  
 # 123456  
 mytree.pop()  
  
 if flag == 3:  
 # 123456  
 mytree.show()  
 if flag == 4:  
 # 123456  
 search()  
  
 if flag == 0:  
 # 123456  
 return  
  
 print("\n\n")  
  
  
  
class Heap(Tree):  
 def \_\_init\_\_(self):  
 super(Heap, self).\_\_init\_\_()  
  
 def insert(self, num):  
 # 123456789  
 node = Node(num)  
 self.push(node)  
 n = self.size()  
 self.up(n - 1)  
  
  
 def swap(self, a, b):  
 na = self.get(a)  
 nb = self.get(b)  
 na.key, nb.key = nb.key, na.key  
  
  
 def up(self, me):  
 par = self.getPar(me)  
 while par != -1:  
 # 123456  
 if self.get(par).key > self.get(me).key:  
 # 123456789  
 self.swap(par, me)  
 me = par  
 par = self.getPar(me)  
  
  
 def getMin(self, me):  
 # 123456789  
 left = self.getLeft(me)  
 if left < 0:  
 # 123456  
 return -1  
  
 right = self.getRight(me)  
 if right < 0:  
 return left  
 leftnum = self.get(left).key  
 rightnum = self.get(right).key  
 if leftnum < rightnum:  
 # 123456  
 return left  
 else:  
 return right  
  
  
 def down(self, me):  
 child = self.getMin(me)  
 while child != -1:  
  
 cnum = self.get(child).key  
 menum = self.get(me).key  
 if cnum >= menum:  
 return  
 self.swap(child, me)  
 me = child  
 child = self.getMin(me)  
  
 # 123456789  
 # delMin()  
 def delMin(self):  
 # 123456  
 if self.size() == 0: # 已经为空可  
 # 123456  
 print("The minimum priority queue is already empty.")  
 return  
  
 n = self.size()  
 num = self.head.next.key # 记录删除的最小值  
 self.swap(0, n - 1) # 把第一个和最后一个元素交换  
 self.pop() # 删除最后一个  
 self.down(0) # 执行下沉操作  
 print("The minimum value was successfully deleted: ", num)  
  
  
 def delLeast(self):  
 # 123456  
 if self.size() == 0:  
 return  
  
 n = self.size()  
 num = self.head.next.key  
 self.swap(0, n - 1)  
 self.popLast()  
 self.down(0)  
  
  
  
 # priority queues  
myqueue = Heap()  
  
  
  
 # insert()  
def insert():  
 # 123456789  
 while True:  
 # 123456789  
 num = input("Please enter the integer you want to increase (Enter a negative number to exit increment): ")  
 num = int(num)  
 if num < 0:  
 # 12345678  
 return  
  
 myqueue.insert(num)  
 print("A node is successfully added\n")  
  
  
  
def HeapMenu():  
 # 12345678  
 print("1 Add several nodes")  
 print("2 Delete the last node")  
 print("3 Display priority queues")  
 print("0 Exit program run")  
 ss = "0123"  
 srr = list(ss)  
 while True:  
 # 123456789  
 flag = input("Please enter a selection of 0 to 3: ")  
 if flag in srr:  
 # 123456  
 return int(flag)  
  
  
  
  
# minimum heap model  
def HeapModel():  
 # 123456789  
 global myqueue  
 myqueue = Heap()  
 while True:  
 # 123456789  
 flag = HeapMenu()  
 if flag == 1:  
 # 123456  
 insert()  
  
 if flag == 2:  
 # 123456789  
 myqueue.delMin()  
  
 if flag == 3:  
 # 123456789  
 myqueue.show()  
 if flag == 0:  
 # 123456789  
 return  
  
 print("\n\n")  
  
  
myinsert = []  
  
mydel = []  
  
num = 1000  
ttmin=0  
ttmax=0  
  
  
  
  
def InsertOne(i):  
  
 num = random.randint(0, 100000)  
 s = time.time()  
  
 myqueue.insert(num)  
 e = time.time()  
 myinsert.append(e - s)  
  
  
# test the insert()  
def testInsert():  
 global myinsert  
 myinsert = []  
 for i in range(num):  
 InsertOne(i)  
 n = len(myinsert)  
 a = 1000  
 for i in range(n):  
 myinsert[i] = int(myinsert[i] \* a \* a)  
 print("insert ok")  
  
  
# test the delete()  
def testDel():  
 # 123456789  
  
 global mydel  
 mydel = []  
 for i in range(num):  
 s = time.time()  
 myqueue.delLeast()  
 e = time.time()  
 mydel.append(e - s)  
  
 mydel.reverse()  
 n = len(mydel)  
 a = 1000  
 for i in range(n):  
 mydel[i] = int(mydel[i] \* a \* a)  
 print("del ok")

insert()

def insert():  
while True:  
 num = input("Please enter the integer you want to increase (Enter a negative number to exit increment): ")  
 num = int(num)  
 if num < 0 :  
 return

myqueue.insert(num)  
print("A node is successfully added\n")

To insert the data into the minimum heap, we should insert the data to the tail node of the linked list, and then search from the bottom to the top to find the fit subscript.

delMin()

def delMin(self):  
 # 123456  
 if self.size() == 0:   
 print("The minimum priority queue is already empty.")  
 return  
  
 n = self.size()  
 num = self.head.next.key   
 self.swap(0, n - 1)   
 self.pop()   
 self.down(0)   
 print("The minimum value was successfully deleted: ", num)

During the operation we find that the minimum element is always the first element of the linked list. So deleting the minimum value of the priority queue is also to delete the first element of the linked list. Firstly, we should swap the first element with the last element and then delete the last node. Then, perform a sinking operation. The minimum element has been deleted so far.

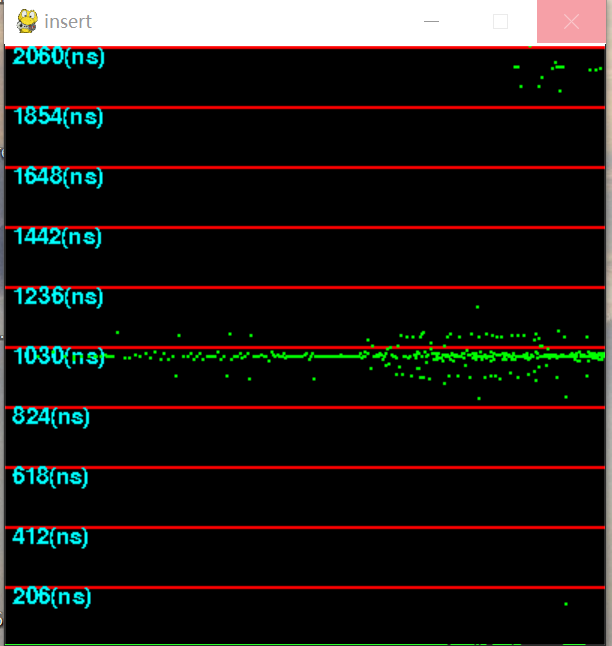
**3.Analyze the time complexity of the methods above in a minimum queue**

n is the data size, which is the number of data in the priority queue when the operation is performed. Time complexity generally refers to the average time complexity. The height of the completed binary tree which has n nodes is less than log2n. The time complexity of heaping is directly proportional with the height of the tree, that is O(logn).

Heaping is the main logical of inserting data into the heap and deleting the elements at the top of the heap so that their time complexity are all O(logn).

**4.perfomance benchmark**

insert（）



delMin()

