Huffman Tree

In order to implement a Huffman tree for text encoding one needs to build a priority queue based on individual character frequency mapping.

Frequency mapping is accomplished using this method of the **Pqueue class**:

|  |
| --- |
| @staticmethod  def text\_freq\_dic(text):  frequency = {}  for character in text:  if not character in frequency:  frequency[character] = 0  frequency[character] += 1  return frequency |

This process is O(n) where n is the number of chracters contained in the text.

The priority queue is built by pushing the frequency map into the queue. Where they will be placed in the queue according to their frequencies. Low frequency characters are in the front. Higher frequency characters lie in the back:

|  |
| --- |
| @staticmethod  def PQ\_Text(text):  d= PQueue.text\_freq\_dic(text)  PQ=PQueue()  for key in d:  PQ.enqueue(key,d[key])  return PQ.items |

The priority queue is backed by a list (i.e. a contiguous array) and relies on append and pop from front (pop(0)) for enqueing and dequeing elements respectively. However one needs to resort the array based on priority which is O(nlog(n)) process.

|  |
| --- |
| def enqueue(self,data,pri):  self.items.append(PQNode(data,pri))  self.items=sorted(self.items, reverse=False ,key=lambda x: x.pr)  self.nitems +=1    def enqueue\_node(self,node):  self.items.append(node)  self.items=sorted(self.items, reverse=False ,key=lambda x: x.pr)  self.nitems +=1      def is\_empty(self):  return self.items==[]    def dequ(self):  if self.is\_empty():  raise EmptyQueueError |

|  |
| --- |
| This method is part of the class HuffmanTree class  def build\_HfTree\_from\_Txt(self,text):  if text=="":  print("Empty Text!")  return None  PQ=PQueue()  PQ.build\_from\_text(text)  PQ.build\_tree()  self.root=PQ.items[0]  this code references this piece of code from the priority queue class  @staticmethod  def PQ\_Text(text):  d= PQueue.text\_freq\_dic(text)  PQ=PQueue()  for key in d:  PQ.enqueue(key,d[key])  return PQ.items      def build\_from\_text(self,text):  self.items=self.PQ\_Text(text)  self.nitems=len(self.items) |

The Huffman Tree algorith relies on iteratively executing the following algorithm:

1. Pop element
2. Pop element
3. Build a small tree from the two popped element with data containing any keyword (here I used “\*” and a frequency equal to the sum of the frequencies of the two poppoed nodes.
4. Push that root of this small tree back into the priority queue at the appropriate location.
5. Repeat until one node is left in the priority queue.

|  |
| --- |
| def build\_tree(self):  while self.size()> 1:  n1= self.dequ()  n2= self.dequ()  b= HuffmanTree()  b.root=PQNode('\*\*', (n1.pr +n2.pr))  b.root.leftchild=n1  b.root.rightchild=n2  self.enqueue\_node(b.root) |

Once the queue has one element this element is basically the fulll tree encoding of the characters. This tree is a binary tree with low frequency characters at the top and high frequency lying at the bottom of the tree. The tree height is O(log(n)) and according searching for leafs is O(log(n)) process.

|  |
| --- |
| These methods are used to locate a node on the Huffman tree (O(log(n))  def search\_node(self,key):  node=self.\_search\_node(key,self.root)  return node    def \_search\_node(self,key,node):  if node!=None:  if node.data==key:  return node  else:  n=self.\_search\_node(key,node.leftchild)  if n is not None:  return n  else:  m=self.\_search\_node(key,node.rightchild)  if m is not None:  return m  return None    def encode(self,node\_s):  path=self.Path\_to\_node(node\_s)    dic={"left":"0" , "right":"1"}  st=""  for i in path[1:]:  st+=(dic[i[1]])  return st    These methods are needed for building a path of moves from root to leaf nodes (essentially either descend left or right )  def Path\_to\_node(self,node\_s):  path=[]  self.\_Path\_to\_node(self.root,node\_s,path,'root')  return path    def \_Path\_to\_node(self,root,node\_s,path,direction):  ## Basic case:  if root==None:  return 0    path.append((root.data, direction))    ## If node is found  if root.data==node\_s.data:  return 1    if (self.\_Path\_to\_node(root.leftchild,node\_s,path,'left')) or (self.\_Path\_to\_node(root.rightchild,node\_s,path,'right')):  return True    path.pop() |

Finally the encoding and decoding methods make use of the above classes

|  |
| --- |
| import sys  def huffman\_encoding(text):  b=HuffmanTree()  O(n log (n)) building the priority queue and huffman tree  b.build\_HfTree\_from\_Txt(text)  Mapping={}  encoding=""  for ch in text:  if ch in Mapping.keys():  encoding+=Mapping[ch]  This loop is repeated n times and the body is O(log (n)) for searching binary tree.  So overall cost is O(n log (n))  else:  node=b.search\_node(ch)  Mapping[ch]=b.encode(node)  encoding+=Mapping[ch]    return encoding , b  def huffman\_decoding(text\_code,tree):    decoded=""  node=tree.root  ## Traverse the tree according to the text\_code  for i in text\_code:  if i=="1":  node=node.rightchild  if node.leftchild is None and node.rightchild is None:  decoded+=node.data  node=tree.root  else:  node=node.leftchild  if (node.leftchild is None) and (node.rightchild is None):  decoded+=node.data  node=tree.root    return decoded |

As explained above the algorithm for encoding is O(n log (n)) and following similar reasoning the decoding if also O(n log (n))