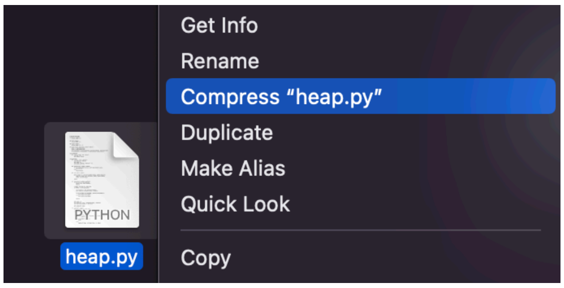
Advanced: PSET 2

Huffman Encoding

In this problem, you will be implementing a popular compression algorithm, Huffman Encoding, that makes use of priority queues!

What is compression?

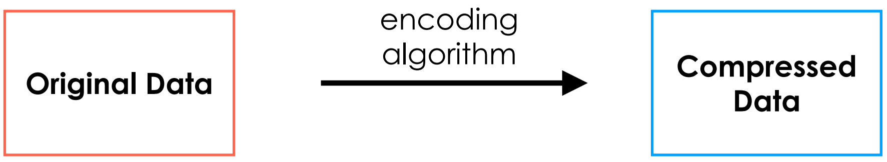
Compression is when we take data and make it into a smaller size (in memory). This is useful if we are transmitting data because smaller sizes of data are transmitted faster.



Most operating systems have in built compression tools to compress files into smaller sizes. There are two steps to compression. Encoding, and decoding.

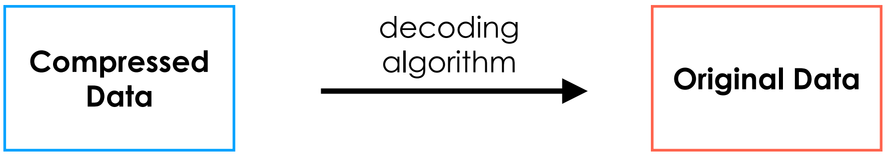
Encoding

Encoding is the process of converting data into a smaller size:



Decoding

Decoding is the process of converting back the compressed data back to its original state, which usually occurs on the receiver's end.



In this PSET, you will be implementing the encoding function of a popular compression technique: **Huffman compression**.

Huffman Compression

Huffman compression is a compression technique for string data. Strings are arrays of characters. Originally, each character in a string takes up 1 byte, and moreover, no matter how frequent a character appears, it takes up the same amount of space. How Huffman compression works is that it:

1. Assigns characters to smaller codes which take up smaller sizes. For instance, the character `'A'` might take the code of `011`, which is only 3 bits, instead of a **byte**, which is **8 bits**
2. Assigns characters which **appear more frequently** codes with smaller bit lengths. For instance, given the string `"HUFFMAN ENCODING"`, the character `'N'` which appears 3 times, will be assigned a code of `00` while `'H'`, which only appears once, will be assigned a code of `11110`

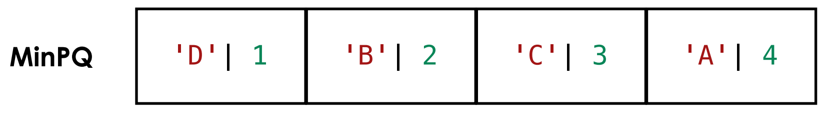
Huffman Encoding Algorithm

Let's learn how the Huffman encoding algorithm works using the example string: `'AAABBACCCD'`

Step 1: Count the frequencies of each character in the string. We can store this in a map data structure like a dictionary or hash table.

```
{
    'A': 4,
    'B': 2,
    'C': 3,
    'D': 1
}
```

Step 2: Store each character as a **TreeNode with left and right attributes (set to None)** in a **minimum priority queue sorted according to frequency of each character.** We'll understand why we use a TreeNode in the next step.



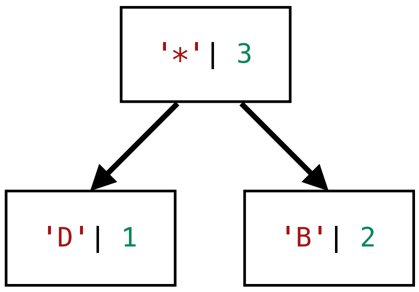
Step 3: While our min priority queue has more than 1 item

- **3.1** Extract the 2 min nodes and create a new tree node where its left is the node with smallest frequency and right is the node with second smallest frequency.
- **3.2** Insert the new node back into the priority queue.

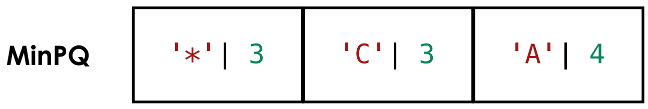
(continue to next page for visualisation of step 3)

Iteration 1

3.1: Create Node

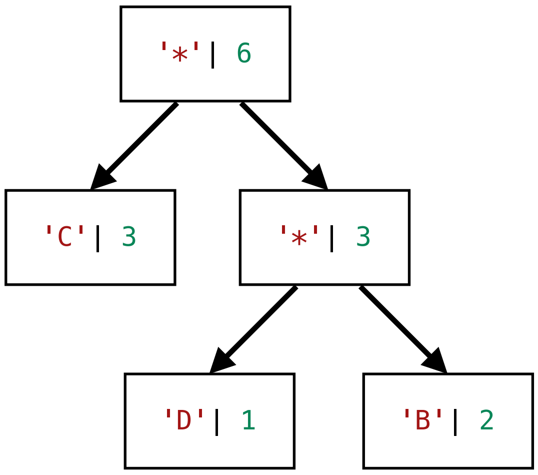


3.2: Insert back into PQ

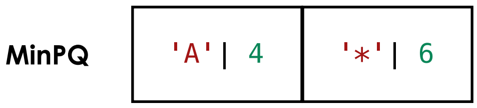


Iteration 2

3.1: Create Node



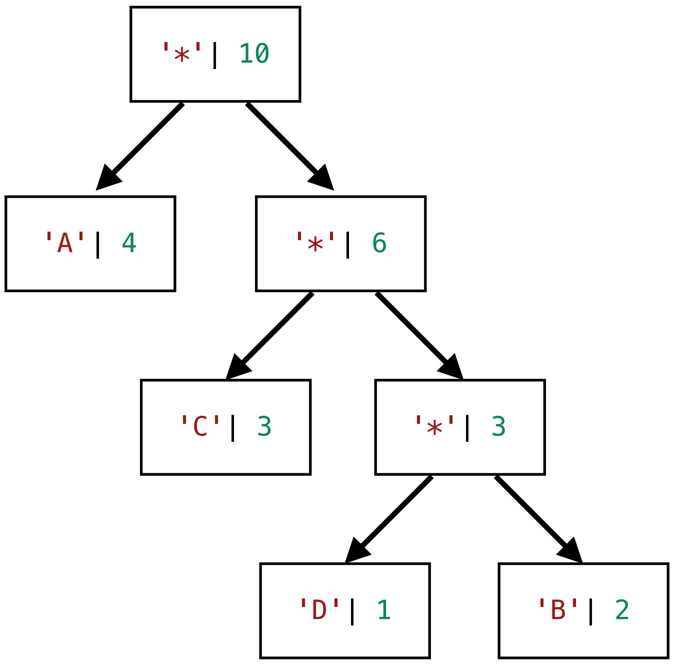
3.2: Insert back into PQ



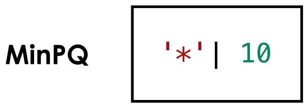
Note: Since `'C'` and the **subtree** have a frequency of 3, it doesn't matter which goes to the left or right.

Iteration 3

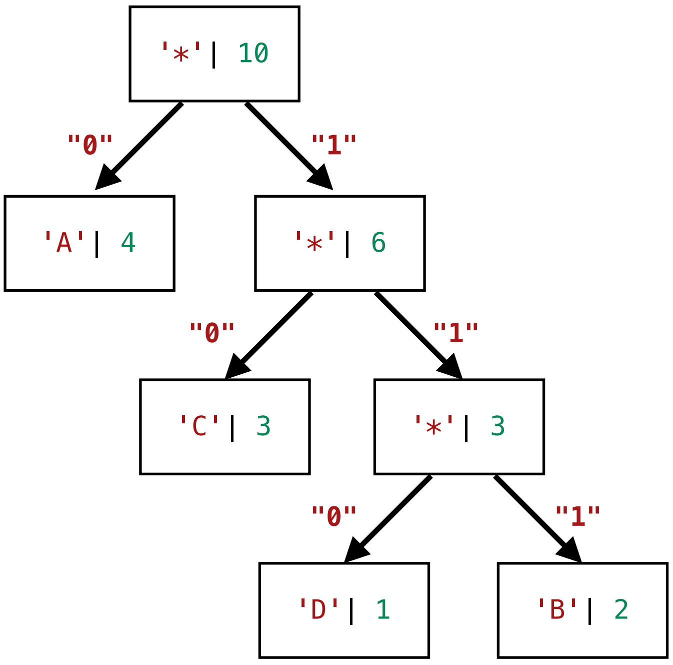
3.1: Create Node



3.2: Insert back into PQ



Step 4: At this point, we have constructed our **Huffman Tree**, and we now need to **traverse it** to assign codes to each character. To do so, we will assign each left path `'0'` and right path `'1'`. Then, traverse the tree, and whenever we reach a leaf node (which means we've encountered a single character), that character's assigned code will be it's **path**



According to the above **tree**, our characters' codes will look like so:

```
{
    'A': '0',
    'B': '111',
    'C': '10',
    'D': '110'
}
```

As you can see, the characters with higher frequencies, like `'A'`, have shorter bit lengths! Additionally, another feature to note is that no character is **a prefix of another character**. This is an important feature in Huffman encoding!

Step 5: The last step is to simply iterate through the original string and replace each character with its assigned code:

- original data: `"AAABBACCCD"`
- encoded data: `0001111110101010110`
- space used before compression (in bits): **80**
- space used after compression (in bits): **19**

A. Starting Code

`HuffmanEncoding`

```
def HuffmanEncoding(data):

    # 1. Determine frequencies of characters
    charFrequencies = calculateFrequencies(data)

    # 2. Build Huffman tree
    huffmanRoot = createHuffmanTree(charFrequencies)

    # 3. Traverse Huffman tree to determine code for each char
    codes = calculateCodes(huffmanRoot)

    # 4. Build encoded string using codes
    encodedOutput = encodeString(data, codes)

    return encodedOutput, min
```

The overall Huffman algorithm has been written for you. Your job will be to implement each function called! Additionally, we have implemented a main function to help you test your code.

TreeNode

```
class TreeNode:
    def __init__(self, char, frequency, left=None, right=None):
        self.char = char
        self.frequency = frequency
        self.left = left
        self.right = right
```

We have defined a class called **TreeNode** for you which takes in:

- **char**: specific character (e.g. 'A')
- **frequency**: frequency at which that character appears in the string. If the node is not a leaf node, then frequency will be the sum of it's child nodes' frequencies
- **left**: left child node
- **right**: right child node

MinHeap

```
class HeapItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value

 class MinHeap:
     def __init__(self, maxsize):
         self.maxsize = maxsize
         self.size = 0
         self.heap = [None] * (maxsize + 1)
         self.positions = {}

     def insert(self, newKey, newValue):

     def getMin(self) -> HeapItem:
```

We have imported the implementation of MinHeap done during the lecture. Above are the methods that you will likely use in this PSET. Do note that items are stored as **HeapItems** in the heap.

B. Your Task

Implement the following functions which make up the Huffman Encoding algorithm:

1. calculateFrequencies

- This function takes in the **data string** and returns a **dictionary** containing each character in the string as keys and it's frequency as the value

2. createHuffmanTree

- This function takes in **char frequencies (dictionary)** and inserts each character as a TreeNode into a MinPQ sorted according to character frequency
- Then, while the PQ has more than one item:

  - Extract the 2 min nodes and create a new tree node where its left is the node with smallest frequency and right is the node with second smallest frequency. Remember that when you call the GetMin() function, it returns a HeapItem. However, each **TreeNode** should contain **TreeNodes** as it's left and right
  - Insert the new node back into the priority queue
- Once the tree has been build, this function should return the root node of the tree as a **TreeNode**

3. `calculateCodes`

- This function takes in the root node from the Huffman tree
- It should traverse the tree, and determine the code for each character in the string:
  - A left path is considered `'0'`
  - A right path is considered `'1'`
  - Traverse a tree until you find a leaf node, and that character's code will be its path.
  - For instance, to get to C, we first go right, then left. Thus, C's code will be `'10'`
- This function should should return a dictionary with each character as keys and it's code as the value

4. `encodeString`

- This function takes in the original string (data) and the dictionary of codes for each char
- It should return the string with each character replaced with its corresponding code

C. Sample Output

```
python huffman.py
input a string to encode via huffman: AAABBACCCD
encoded data: 0001111110101010110
space usage before compression: 80
space usage after compression: 19

python huffman.py
input a string to encode via huffman: HUFFMAN ENCODING
encoded data: 1111011111100100010011010001110101001011011011001110001010
space usage before compression: 128
space usage after compression: 58

python huffman.py
input a string to encode via huffman: HELLO WORLD
encoded data: 11101111101011001100111000010010
space usage before compression: 88
space usage after compression: 32
```

D. Submission

To test your code, run the following command:

```
python utils/test.py
```

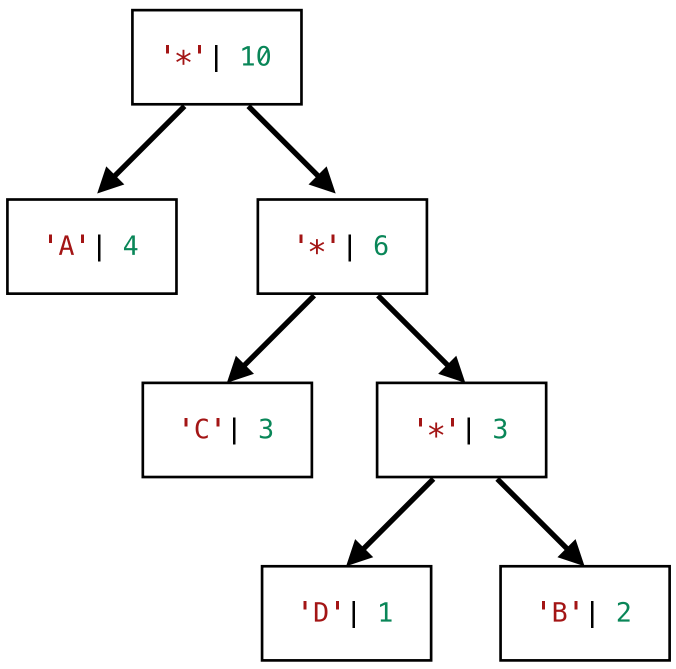
The following strings are used to test your code:

```
string1 = "AAAA BBB CCCCC DD"
string2 = "HHHELLLLOOWWWWWWOOOOOOORRRRRRRRRLLLLLLLLLLDDDDDDDDDDD"
string3 = "AAAABBBBBCCCCCCDDDDDDDEEEEEEEE"
```

Note: For the `createHuffmanTree` function test, we perform a **preorder traversal** to flatten your tree and compare it against the answer. For instance, the tree below, when tested by the auto grader, will return:

```
{
      'treeShape': [1, 0, 1, 0, 1, 0, 0],
      'chars': ['A', 'C', 'D', 'B']
}
```

where `treeShape` is the preorder traversal (0 represents leaf nodes) from the root, and `chars` represent the char stored at each **leaf node**.



To submit your code, run the following command:

```
python utils/submit.py
```