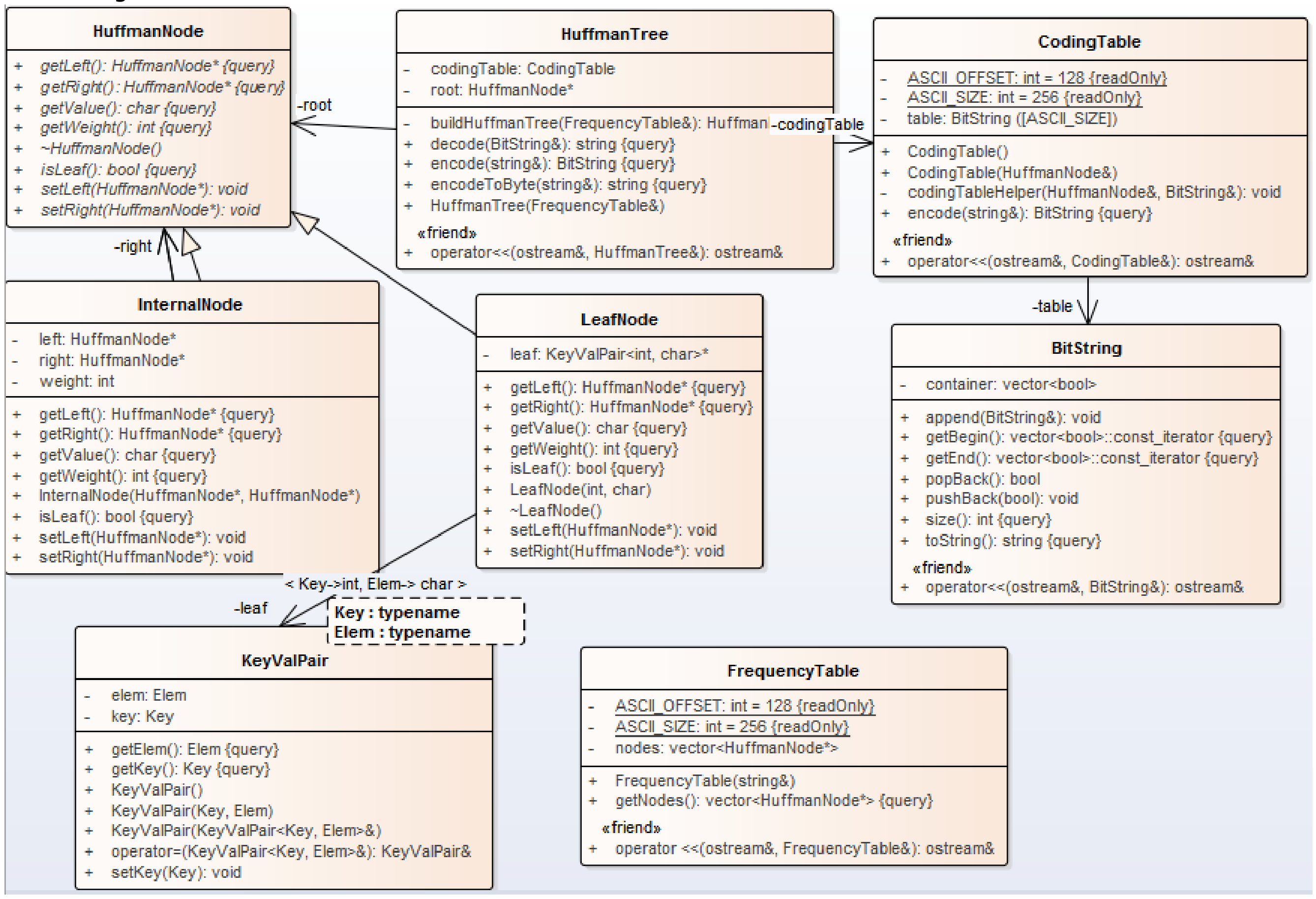
Data compression using Huffman algorithm

This is how it works, you create a Huffman tree using a FrequencyTable or directly from a string, because FrequencyTable is constructed from a string. A FrequencyTable on creation analyzes the string and stores a vector of HuffmanNodes, every node contains a character and how many times that character is observed in the string (weight of node). In the constructor of the HuffmanTree is called the helper function builldHuffmanTree which builds a min heap from the FrequencyTable using the weight of the nodes as priority values, after that the first two nodes are merged into an InternalNode, this process is repeated until only one node is left, which is our root node. After that with a recursive traversal from the root node a CodingTable is created (Coding table – maps a given character with a BitString (BitString - a wrapper class around vector<bool> for convenience)). And our tree is ready to rumble.

Class Diagram:


HuffmanNode
+ getLeft(): HuffmanNode* {query}
+ getRight(): HuffmanNode* {query}
+ getValue(): char {query}
+ getWeight(): int {query}
+ ~HuffmanNode()
+ isLeaf(): bool {query}
+ setLeft(HuffmanNode*): void
+ setRight(HuffmanNode*): void
-root
HuffmanTree
- codingTable: CodingTable
- root: HuffmanNode*
- buildHuffmanTree(FrequencyTable&): Huffman
+ decode(BitString&): string {query}
+ encode(string&): BitString {query}
+ encodeToByte(string&): string {query}
+ HuffmanTree(FrequencyTable&)
«friend»
+ operator<<(ostream&, HuffmanTree&): ostream&
-codingTable
CodingTable
- ASCII_OFFSET: int = 128 {readOnly}
- ASCII_SIZE: int = 256 {readOnly}
- table: BitString ([ASCII_SIZE])
+ CodingTable()
+ CodingTable(HuffmanNode&)
- codingTableHelper(HuffmanNode&, BitString&): void
+ encode(string&): BitString {query}
«friend»
+ operator<<(ostream&, CodingTable&): ostream&
-right
InternalNode
- left: HuffmanNode*
- right: HuffmanNode*
- weight: int
+ getLeft(): HuffmanNode* {query}
+ getRight(): HuffmanNode* {query}
+ getValue(): char {query}
+ getWeight(): int {query}
+ InternalNode(HuffmanNode*, HuffmanNode*)
+ isLeaf(): bool {query}
+ setLeft(HuffmanNode*): void
+ setRight(HuffmanNode*): void
LeafNode
- leaf: KeyValPair<int, char>*
+ getLeft(): HuffmanNode* {query}
+ getRight(): HuffmanNode* {query}
+ getValue(): char {query}
+ getWeight(): int {query}
+ isLeaf(): bool {query}
+ LeafNode(int, char)
+ ~LeafNode()
+ setLeft(HuffmanNode*): void
+ setRight(HuffmanNode*): void
-table
BitString
- container: vector<bool>
+ append(BitString&): void
+ getBegin(): vector<bool>::const_iterator {query}
+ getEnd(): vector<bool>::const_iterator {query}
+ popBack(): bool
+ pushBack(bool): void
+ size(): int {query}
+ toString(): string {query}
«friend»
+ operator<<(ostream&, BitString&): ostream&
< Key->int, Elem-> char >
-leaf
Key : typename
Elem : typename
KeyValPair
- elem: Elem
- key: Key
+ getElem(): Elem {query}
+ getKey(): Key {query}
+ KeyValPair()
+ KeyValPair(Key, Elem)
+ KeyValPair(KeyValPair<Key, Elem>&)
+ operator=(KeyValPair<Key, Elem>&): KeyValPair&
+ setKey(Key): void
FrequencyTable
- ASCII_OFFSET: int = 128 {readOnly}
- ASCII_SIZE: int = 256 {readOnly}
- nodes: vector<HuffmanNode*>
+ FrequencyTable(string&)
+ getNodes(): vector<HuffmanNode*> {query}
«friend»
+ operator <<(ostream&, FrequencyTable&): ostream&