horizontal line

**Matko Gabriel Mihai**

Babes Bolyai University

Computer Science, first year

Gr. 914/1

# CONTENT

1. Problem overview and justification
2. **ADT** - domain and interface (+iterator)
3. Representation for the **ADT** (+**iterator**) and implementation for the operations in pseudocode (+**iterator**)
4. Tests for the **ADT** (C++ code)
5. Implementation of the problem solution in pseudocode
6. Complexities for the operations

Huffman encoding

# Problem overview and justification

When storing characters on a computer, each of them occupies 8 bits of data. For very big data files, it would be inefficient to store the text in its original form because it will use a lot of space.

Here, the **Huffman Encoding** comes in place. It provides a unique way of encoding each character from a text, based on it’s frequency. The most used characters will have a small binary number, and the less used ones will have a bigger number. By doing this, we will reduce the data size of the file by compressing it, fitting more data in the same space.

This method is proven mathematically to be the most efficient way in providing a unique binary encoding to any character from a text.

We will use a priority queue to keep the characters with lower frequency to the top and a binary tree because that way it will be easier to generate the code for each character.

# 

# 

# ADTs - domain and interface

## Priority Queue

The ADT Priority Queue is a container in which each element has an associated priority. The access to the elements is restricted, so we can access only the element with the highest priority. Priority Queues cannot be iterated (because the access is limited to the highest priority element), so they don’t have an iterator operation.

**Domain:**

**PQ** = {pq | pq is a priority queue with elements **TElem↑**}

**Interface:**

init (pq, R):

* **Description**: creates a new empty priority queue
* **Pre**: R is a relation over the priorities
  + R : TPriority x TPriority
* **Post**: pq is an empty **PQ**

destroy (pq):

* **Description**: destroys a priority queue
* **Pre**: pq is a **PQ**
* **Post**: pq was destroyed

push (pq, elem):

* **Description**: pushes (adds) a new element to the priority queue
* **Pre**: pq is a **PQ**, elem is a **TElem↑**, containing information and a priority
* **Post**: pq’ is a **PQ**, pq’ = pq + elem

pop (pq, elem):

* **Description**: pops (removes) from the priority queue and returns the element with the highest priority
* **Pre**: pq is a **PQ**
* **Post**: pop ← elem**, elem** is a **TElem↑**, the element with the highest priority from pq, p is its priority, elem is removed from pq
* **Throws**: an exception if the priority queue is empty

top (pq, elem):

* **Description**: returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue
* **Pre**: pq is a **PQ**
* **Post**: e is a TElem, p is a TPriority, e is the element with the highest priority from pq, p is its priority
* **Throws**: an exception if the priority queue is empty

isEmpty (pq):

* **Description**: checks if the priority queue is empty
* **Pre**: pq is a **PQ**
* **Post**: isEmpty ← **true**, if pq has no elements, **false** otherwise

size(pq):

* **Description**: returns the number of elements in pq
* **Pre**: pq is a **PQ**
* **Post**: size ← number of elements in **pq**

## Binary Tree

The ADT Binary Tree is an ordered tree in which each node has at most two children. We call the children of a node the left child and right child

**Domain:**

**BT** = {bt | bt binary tree with nodes containing information of type TELem}

**Interface:**

Init (bt):

* **Description**: creates a new, empty binary tree
* **Pre**: true
* **Post**: bt ∈ **BT** , bt is an empty binary tree

initLeaf(bt, e):

* **Description**: creates a new binary tree, having only the root with a given value
* **Pre**: e ∈ **TElem↑**
* **Post**: bt ∈ **BT** , bt is a binary tree with only one node (its root) which contains the value e

initTree(bt, left, right, info):

* **Description**: creates a new binary tree, having a given information in the root and two given binary trees as children
* **Pre**: left, right ∈ **BT↑** , info ∈ **TInfo↑**
* **Post**: bt ∈ **BT** , bt is a binary tree with left child equal to left, right child equal to right and the information from the rootNode is e

insertLeftSubtree(bt, left):

* **Description**: sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)
* **Pre**: bt, left ∈ **BT↑**
* **Post**: bt’ ∈ **BT↑** , the left subtree of bt’ is equal to left

insertRightSubtree(bt, right):

* **Description**: sets the right subtree of a binary tree to a given value (if the tree had a right subtree, it will be changed)
* **Pre**: bt,right ∈ **BT**
* **Post**: bt’ ∈ **BT** , the right subtree of bt’ is equal to right

root(bt):

* **Description**: returns the information from the root of a binary tree
* pre: bt ∈ **BT** , bt != **Φ**
* post: root ← info ∈ **TInfo↑**, e is the information from the rootNode of bt
* **Throws**: an exception if bt is empty

left(bt):

* **Description**: returns the left subtree of a binary tree
* **Pre**: bt ∈ **BT** , bt != **Φ**
* **Post**: left ← l, **TNode↑**, l is the starting node of the left subtree of bt
* throws: an exception if bt is empty

right(bt):

* **Description**: returns the right subtree of a binary tree
* **Pre**: bt ∈ **BT** , bt != **Φ**
* **Post**: right ← r, r ∈ **TNode↑** , r is the starting node of the right subtree of bt
* **Throws:** an exception if bt is empty

isEmpty(bt):

* **Description**: checks if a binary tree is empty
* **Pre**: bt ∈ **BT**
* **Post**: empty ← ( True, if bt = Φ False, otherwise)

iterator (bt, traversal, it):

* **Description**: returns an iterator for a binary tree
* **Pre**: bt ∈ **BT** , traversal represents the order in which the tree has to be iterated
* **Post**: it is an iterator over bt that iterates in the order given by traversal

destroy(bt):

* **Description**: destorys a binary tree
* **Pre**: bt ∈ BT
* **Post**: bt was destroyed

### **Iterator for Binary Tree**

**Domain:**

**IT** = {it | it is an iterator for a binary tree }

**Representation:**

Iterator:

* bt: **BinaryTree↑**
* s: **Stack**
* currentNode: **TNode↑**

**Interface:**

init(it, bt):

* **Description**: initialize an iterator for a binary tree
* **Pre**: bt ∈ BT, it ∈ **IT**
* **Post**: it is an iterator for bt

getCurrent(it):

* **Description**: returns the current element from the iterator
* **Pre**: it ∈ **IT**
* **Post**: getCurrent ← the information from the current node

valid(it):

* **Description**: verifies if the current node in the iterator is a valid one
* **Pre**: it ∈ **IT**
* **Post**: valid ← true if currentNode differs from NIL, false otherwise

next(it):

* **Description**: iterates to next element in binary tree
* **Pre**: it is a PreOrderIterator over a binary tree
* **Post**: it will contain now the next element in the binary tree

# Representation for the ADTs

## Priority Queue

The ADT Priority queue will be implemented using a dynamic array, where the elements will be kept sorted by the priority. The first elements is the element with the highest priority.

**PriorityQueue:**

* elems: **TElem↑ []**
* capacity: **Integer**
* len: **Integer**
* relation: **TRelation**

## 

## 

## 

## 

## 

## Binary Tree

The ADT Binary Tree will be implemented using a linked representation with dynamic allocation where we have a structure to represent a node, containing the information and the address of the left and right child (right and left child will be a Node of a binary tree too)..

**BinaryTree:**

* rootNode: **TNode↑**

**TNode↑:**

* left: **TNode↑**
* right: **TNode↑**
* information: **TInfo↑**

**Iterator:**

* bt: **BinaryTree↑**
* nodeStack: **Stack**
* currentNode: **TNode↑**

# Implementation for the operations in pseudocode

## PriorityQueue:

**subalgorithm init**(pq):

**Complexity**: Θ(1) for AC, BC, WC

pq.capacity ← 10

pq.len ← 0

pq.elems ← **@allocate** a new array of **Telem↑** of size pq.capacity

**end-subalgorithm**

**subalgorithm** **push**(pq, elem):

**Complexity**: O(n) for AC, WC, Θ(1) BC when the elements have the highest priority in relation with TRelation(BC)

**if** pq.len = pq.capacity **then**:

newArr ← **@allocate** a new array of **TElem↑** of size 2\*pq.capacity

**@copy** elems from pq.elems to newArr

pq.capacity ← pq.capacity \* 2

**end-if**

pq.elems[pq.len+1] ← elem

pq.len ← pq.len + 1

i ← pq.len-1

**while** i > 1 and TRelation(pq.elems[i], pq.elems[i-1]) = True **do**:

**@swap** pq.elems[i] with pq.elems[i-1]

i ← i-1

**end-while**

**end-subalgorithm**

**subalgorithm pop**(pq, elem):

**Complexity**: Θ(1) for AC, BC, WC

pq.len ← pq.len - 1

elem ← pq.elems[pq.len+1]

**end-subalgorithm**

**subalgorithm top**(pq, elem):

**Complexity**: Θ(1) for AC, BC, WC

elem ← pq.elems[pq.len-1]

**end-subalgorithm**

**function** isEmpty(pq):

**Complexity**: Θ(1) for AC, BC, WC

**if** pq.len = 0 **then**:

isEmpty ← True

**else**

isEmpty ← False

**end-if**

**end-function**

**subalgorithm destroy**(pq):

**Complexity**: Θ(n) for AC, BC, WC, n being the number of elements from pq

**for** i ← 0, pq.len, 1 **execute**:

**@deallocate** pq.elems[i]

**end-for**

**@deallocate** pq.elems

pq.elems ← NIL

**end-subalgorithm**

**function** size(pq):

**Complexity**: Θ(1) for AC, BC, WC

size ← pq.len

**end-function**

## 

## 

## BinaryTree:

**subalgorithm** init(bt):

**Complexity**: Θ(1) for AC, BC, WC

bt.node ← **@allocate** new empty **TNode↑**

**end-subalgorithm**

**subalgorithm** init(bt, info):

**Complexity**: Θ(1) for AC, BC, WC

bt.node ← **@allocate** new empty **TNode↑**

**[**bt.node**]**.information ← info

**end-subalgorithm**

**subalgorithm** initLeaf(bt, info):

**Complexity**: Θ(1) for AC, BC, WC

bt.node ← **@allocate** new empty **TNode↑**

**[**bt.node**]**.information ← info

**end-subalgorithm**

**subalgorithm** insertLeftSubtree(bt, left):

**Complexity**: Θ(1) for AC, BC, WC

**[**bt.rootNode**]**.left ← **[**left**]**.rootNode

**end-subalgorithm**

**subalgorithm** insertRightSubtree(bt, right):

**Complexity**: Θ(1) for AC, BC, WC

**[**bt.rootNode**]**.right ← **[**right**]**.rootNode

**end-subalgorithm**

**function** root(bt):

**Complexity**: Θ(1) for AC, BC, WC

root ← **[**bt.rootNode**]**.info

**end-function**

**function** left(bt):

**Complexity**: Θ(1) for AC, BC, WC

left ← **[**bt.rootNode**]**.left

**end-function**

**function** right(bt):

**Complexity**: Θ(1) for AC, BC, WC

right ← **[**bt.rootNode**]**.right

**end-function**

**function** isEmpty(bt):

**Complexity**: Θ(1) for AC, BC, WC

**if** **[**bt.rootNode**]**.info = NIL **then**:

isEmpty ← True

**else**:

isEmpty ← False

**end-if**

**end-function**

**subalgorithm** iterator (bt, traversal, it):

**Complexity**: Θ(1) for AC, BC, WC

init(it, bt)

//now it is an iterator over bt

**end-subalgorithm**

**subalgorithm** destroy(bt):

**Complexity**: Θ(1) for AC, BC, WC

**@deallocate** bt.rootNode

**end-subalgorithm**

## BinaryTree iterator:

**subalgorithm** init(bt, it):

**Complexity**: O(n) for AC, BC, WC

it.bt ← bt

it.currentNode ← [bt].rootNode

push(it.nodeStack, [bt].rootNode)

next(it)

**end-subalgorithm**

**function** getCurrent(it):

**Complexity**: Θ(1) for AC, BC, WC

getCurrent ← [it.currentNode].information

**end-function**

**function** valid(it):

**Complexity**: Θ(1) for AC, BC, WC

**if** it.currentNode != NIL **then**:

valid ← True

**else**

valid ← False

**end-if**

**end-function**

**subalgorithm** next(it):

**Complexity**: O(n) for AC, BC, WC

**if** empty(it.nodeStack) = True **then**:

it.currentNode = NIL

break //@close the subalgorithm

**end-if**

node ← top(it.nodeStack)

pop(it.nodeStack)

it.currentNode ← node

**if** [node].right = NIL **then**:

push(it.nodeStack, [node].right)

**end-if**

**if** [node].left = NIL **then**:

push(it.nodeStack, [node].left)

**end-if**

**end-subalgorithm**

# Tests for ADTs:

## PriorityQueue:

void testPQ() {

auto \*pq = new PriorityQueue();

srand((unsigned int)time(nullptr));

assert(pq->isEmpty());

pq->push(new BinaryTree(new TNode(new TInfo((char)('a'), 5))));

assert(pq->top()->root()->data == 'a');

assert(pq->top()->root()->freq == 5);

assert(!pq->isEmpty());

pq->push(new BinaryTree(new TNode(new TInfo((char)('b'), 4))));

assert(pq->top()->root()->data == 'b');

assert(pq->top()->root()->freq == 4);

pq->pop();

assert(pq->top()->root()->data == 'a');

assert(pq->top()->root()->freq == 5);

for(int i=0; i<20; i++){

pq->push(new BinaryTree(new TNode(new TInfo('c'+i, i))));

}

assert(pq->top()->root()->data == 'c');

assert(pq->top()->root()->freq == 0);

free(pq);

std::cout << "ALL TESTS PASSED FOR PQ !" << std::endl;

}

## BinaryTree:

void testBT() {

auto \*bt = new BinaryTree();

assert(bt->isEmpty());

bt->initLeaf(new TInfo('!', 3));

assert(bt->root()->data == '!' && bt->root()->freq == 3);

assert(!bt->isEmpty());

free(bt);

bt = new BinaryTree(new TNode(new TInfo('$', 2)));

assert(bt->root()->freq == 2 && bt->root()->data == '$');

bt->insertLeftSubtree(new BinaryTree(new TNode(new TInfo('#', 5))));

assert(bt->left()->info->freq == 5 && bt->left()->info->data == '#');

bt->insertRightSubtree(new BinaryTree(new TNode(new TInfo('#', 5))));

assert(bt->right()->info->freq == 5 && bt->right()->info->data == '#');

free(bt);

bt = new BinaryTree();

bt->initTree(new TInfo('@', 3), new BinaryTree(), new BinaryTree());

assert(bt->left() != nullptr);

assert(bt->right() != nullptr);

free(bt);

std::cout << "ALL TESTS PASSED FOR PQ !" << std::endl;

}

# Problem solution

**subalgorithm** readFile(fileC):

**Description:** reads the content of a file in fileC

**Pre:** fileC is an empty String, fin is a global input file stream

**Post:** fileC contains all the content of fin

**Complexity**: Θ(n) for AC, BC, WC, where n is the number of lines in the file

**if** is\_open(fin) **then**:

**while** fin is not empty **do**:

read ← **@read** line from fin

fileC ← **@append** the content of read to fileC

**end-while**

**end-if**

**end-subalgorithm**

**function** generateHuffmanTree(str, pq):

**Description:** generate the huffman tree corresponding to content of str

**Pre:** str - a string representing the content of the file, pq - priority queue

**Post:** generateHuffmanTree ← the top node of the corresponding huffman tree

**Complexity**: O(n^2) for AC, BC, WC

**@global** freq - a vector representing the frequency of the characters

**makeFrequencies**(str, freq)

**for** i ← 0, 256, 1, **execute**:

**if** freq[i] != 0 **then**:

init(bt) //initialize a new binary tree

info ← **@create** new TInfo pointer

**[**info**].**data ← **@assign** char from ASCII representation with integer code **i**

**[**info**]**.freq ← freq[i]

initLeaf(bt, info)

push(pq, bt)

**end-if**

**end-for**

**while** size(pq) != 1 **do**:

left ← top(pq)

pop(pq)

right ← top(pq)

pop(pq)

**//allocate** newBt- empty binary tree

init(newBt)

info ← **@create** new TInfo pointer

**[**info**].**data ← "$" //or any special character for non-leaf nodes

**[**info**]**.freq ← root(left).freq + root(right).freq

initTree(bt, info, left, right);

push(pq, newBt)

**end-while**

**generateHuffmanCodes([**top(pq)**]**.rootNode, "")

//return value

generateHuffmanTree ← **[**top(pq)**]**.rootNode

**end-function**

**subalgorithm** makeFrequencies(str, freq):

**Description**: generate a vector of frequencies for every character from input file stream

**Pre**: str - string, content of the file, freq - vector of frequencies

**Post**: freq will be a vector of frequencies, where at the position i will be the frequency of the character with ASCII code i in the file

**Complexity**: Θ(n) for AC, BC, WC, n being the number of chars in str

**for** i ← 0, 256, 1 **execute**:

encodings[i] ← ""

freq[i] ← 0;

**end-for**

**for** chr **in** str **execute**:

pos ← ASCII(chr) //converting character chr to its ascii code

freq[pos] ← freq[pos] + 1

**end-for**

**end-subalgoritm**

**subalgorithm** generateHuffmanCodes(root, str):

**Description**: generates the huffman code for every character depending on its frequency

**Pre**: root - **TNode**↑, str - string containing the specifig huffman-code of every character

**Post**: encodings will contain on position ASCII(character) the specific huffman-code for that character

**Complexity**: Θ(2^n) for AC, BC, WC, where n is the number of 0's and 1's in our huffman tree

**if** root = NIL **then:**

**return**

**end-if**

**if [[**root**].**info**].**data != '$' **then**:

pos ← ASCII(**[**root**].**info**].**data)

encodings**[**pos**] ←** str

**end-if**

generateHuffmanCodes(**[**root**]**.left, str + "0")

generateHuffmanCodes(**[**root**]**.right, str + "1")

**end-subalgorithm**

**subalgorithm** printEncoded(str, bt):

**Description**: Outputs the encoded message to the output stream

**Pre**: str - a string containing the original message, bt - huffman tree of the message from str

**Post**: -

**Complexity**: Θ(n) for AC, BC, WC where n is the size of str

**@print** " >>> Letter encoding of : " + str

**for** i ← 0, 256, 1 **execute**:

//encodings[] is a vector containing strings, representing encoding of every ASCII character with integer code i

**if** not encodings[i].empty() **then**: //if the character is present in our message, print it and its huffman specific encoding

**@print** char(i) + encodings[i]

**end-if**

**end-for**

**@print** ">>> Original message: \n"+ str

**@print** ">>> Encoded message: \n" + str

encoded ← ""

**for** chr **in** str **execute**:

encoded ← encoded + encodings[int(chr)] // int(chr) represent the ascii code of chr

**@print** encodings[int(chr)]

**end-for**

decodeFromTree(encoded, bt)

**end-subalgorithm**

**subalgorithm** decodeFromTree(encodedMessage, bt)

**Description**: decode the encodedMessage from huffman codes to human-readable ASCII characters

**Pre**: encodedMessage - string containing a huffman message, bt - a huffman binary tree

**Post**: the decoded message will be outputted to the output stream

**Complexity**: Θ(n) for AC, BC, WC, where n is the number of characters in encoded message

**@print** ">>> Decoded message:"

node ← **[**bt**]**.rootNode

**for** chr **in** encodedMessage **execute**:

**if** chr **=** '0' **then**:

node ← **[**node**]**.left

**else**

node ← **[**node**]**.right

**end-if**

**if [[**node**]**.info**]**.data != **'$' then**:

**@print [[**node**]**.info**]**.data

node ← **[**bt**]**.rootNode

**end-if**

**end-for**

**end-subalgorithm**

**subalgorithm** iterateOverTree(bt)**:**

**Description**: this subalgorithm iterates over a binary tree and print it's values

**Pre**: bt is a binary tree

**Post**: values from bt are printed to normal output stream

**Complexity**: Θ(n) for AC, BC, WC, where n is the number of nodes from bt

iterator ← iterator(bt)

**while** valid(iterator) **do**:

**@print** "char: " + [getCurrent(iterator)].data + " freq: **" +** [getCurrent(iterator)].freq

next(it)

**end-while**

**end-subalgorithm**

**subalgorithm** main():

**Description**: the main function of the program which starts the encoding and decoding of the message

**Pre**: initialize variables which will be used during program execution

**Post**: after its execution, the message from fileContent will be encoded/decoded using a huffman tree

**Complexity**: Θ(1) for AC, BC, WC, the subalgorithm containing only function calls

testBT()

testPQ()

init(p\_queue)

readFile(fileContent)

init(bt)

**[**bt**]**.rootNode ← generateHuffmanTree(fileContent, p\_queue)

printEncoded(fileContent, bt)

destroy(p\_queue)

destroy(bt)

**@print** "Encoding/decoding was successful!"

**end-subalgorithm**