Ungur Nicoleta
Group 917

Huffman encoding
Binary tree and Priority Queue

Specifications:

❑ **The domain of the ADT Binary Tree:**

BT = { **bt** | **bt** binary tree with nodes containing information of type **e** ∈ **TElem**}

Interface:

- **init** (bt)
    - descr: creates a new, **empty** binary tree
    - pre: **true**
    - post: **bt** ∈ **BT** , **bt** is an empty binary tree

- **initLeaf**(bt, e)
    - descr: 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, e, right)
    - descr: creates a new binary tree, having a given information in the root and two given binary trees as children
    - pre: **left,right** ∈ **BT , e** ∈ **TElem**
    - post: **bt** ∈ **BT** , **bt** is a binary tree with left child equal to left, right child equal to right and the information from the **root** is **e**

- **insertLeftSubtree**(bt, left)
    - descr: 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)
    - descr: 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)
  - descr: returns the information from the root of a binary tree
  - pre: **bt** ∈ **BT , bt 6= Φ**
  - post: **root** ← **e, e** ∈ **TElem**, e is the information from the root of bt
  - throws: an **exception** if bt is empty

- **left**(bt)
  - descr: returns the left subtree of a binary tree
  - pre: **bt** ∈ **BT , bt 6= Φ**
  - post: **left** ←**, l, l** ∈ **BT** , l is the left subtree of bt
  - throws: an **exception** if bt is empty

- **right**(bt)
  - descr: returns the right subtree of a binary tree
  - pre: **bt** ∈ **BT , bt 6= Φ**
  - post: **right** ← **r, r** ∈ **BT** , r is the right subtree of bt
  - throws: an **exception** if bt is empty

- **isEmpty**(bt)
  - descr: checks if a binary tree is empty
  - pre: **bt** ∈ **BT**
  - post: **empty** ←True, if bt = Φ
    ←False, otherwise

- **iterator** (bt, traversal, i)
  - descr: returns an iterator for a binary tree
  - pre: **bt** ∈ **BT** , traversal represents the order in which the tree has to be traversed
  - post: **i** ∈ **I**, i is an iterator over bt that iterates in the order given by traversal

- **initIter**(it, bt)
  - description: creates a new iterator for the binary tree
  - pre: **bt** is a binary tree
  - post: **it** ∈ **I** and it points to the first element in **bt** if **bt** is not empty or it is not valid

- **getCurrent**(it, e)
    - description: returns the current element from the iterator
    - pre: **it ∊ I**, it is valid
    - post: **e ∊ TElem**, e is the current element from it

- **next**(it)
    - description: moves the current element from the container to the next element or makes the iterator invalid if no elements are left
    - pre: **it ∊ I, it** is **valid**
    - post: the current element from it points to the next element from the container

- **valid**(it)
    - description: verifies if the iterator is valid
    - pre: **it ∊ I**
    - post: valid
        - **True**, if it points to a valid element from the container
        - **False** otherwise

- **destroy**(bt)
    - descr: destorys a binary tree
    - pre: **bt ∊ BT**
    - post: **bt** was destroyed

❑ **The domain of the ADT Priority Queue:**

PQ = { **pq** | **pq** is a priority queue with elements **(e, p)**, **e ∊ TElem, p ∊ TPriority**}

Interface:

- **init** (pq, R)
    - description: creates a new empty priority queue
    - pre: **R** is a relation over the priorities, **R : TPriority × TPriority**
    - post: **pq ∊ PQ**, **pq** is an empty priority queue

- **destroy**(pq)
    - description: destroys a priority queue
    - pre: **pq ∊ PQ**
    - post: **pq** was destroyed

- **push**(pq, e, p)
    - description: pushes (adds) a new element to the priority queue
    - pre: **pq ∊ PQ, e ∊ TElem, p ∊ TPriority**
    - post: **pq' ∊ PQ, pq' = pq ⊕ (e, p)**

- **pop** (pq, e, p)
    - description: pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
    - pre: **pq ∊ PQ**
    - post: **e ∊ TElem, p ∊ TPriority**, **e** is the element with the highest priority from pq, p is its priority. **pq' ∊ PQ, pq' = pq -(e, p)**
    - throws: an **exception** if the priority queue is empty

- **top** (pq, e, p)
    - description: returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
    - pre: **pq ∊ PQ**
    - post: **e ∊ TElem, p ∊ 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 (it has no elements)
    - pre: **pq ∊ PQ**
    - post: **isEmpty** ← **true**, if **pq** has no elements | **false**, otherwise

- **isFull** (pq)
    - description: checks if the priority queue is full (not every implementation has this operation)
    - pre: **pq ∊ PQ**
    - post: **isFull** ← **true**, if pq is full | **false**, otherwise

Priority queues cannot be iterated, so they **don't have** an **iterator operation**!

Reprezentation:

- <u>**BTNode:**</u>
  info: TElem
  left: ↑ BTNode
  right: ↑ BTNode

- <u>**BinaryTree:**</u>
  root: ↑ BTNode

- <u>**InorderIterator:**</u>
  bt: Binary Tree
  pq: Priority Queue
  currentNode: ↑ BTNode

- <u>**TElem:**</u>
  letter: String
  frequency: Integer
  code: Integer

- <u>**Pair:**</u>
  e: TElem
  p: TPriority

- <u>**PQ:**</u>
  elems: Pair[]
  len: Integer
  cap: Integer

Problem Statement:

The government just found that a state secret is in danger.All the documents must be kept in safe and all the digital information must be compressed. Their IT department decided to encode the information using Huffman code. When they will be sure that everything is ok in the security center, the information will be decoded.

Justification:

It's a greedy algorithm: at each iteration, the algorithm makes a "greedy" decision to merge the two subtrees with least frequency. We will have a priority when making this decision.

The easiest way to encode a text is using a binary tree because every letter from the text will be a leaf. Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.

The easiest way to decode a text is also using a binary tree. We iterate through the tree starting from the root, if the current bit from the code is 0 go to the left child, otherwise go to the right child, if we are at a leaf node we have decoded a character and have to start over from the root .