June 2018
Cluj-Napoca

DOCUMENTATION - DSA

Facultatea de Matematică și Informatică
UNIVERSITATEA BABEŞ BOLYAI

Huffman encoding
Binary tree and Priority Queue

Student Ungur Nicoleta
Group 917
Project 32

Contents:

<u>Huffman encoding</u>
<u>Binary tree and Priority Queue</u>

I. Task

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 .

II. Specification and interface

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

- **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

Specifications:

- ❑ **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, index)
    - description: pops (removes) from the priority queue the element with the highest priority. It returns the element.The index will be always 1, meaning the element with the highest priority.
    - pre: **pq ∊ PQ , index ∊ Integer(=1)**

- post: **e ∊ ↑Node** is the element with the highest priority from pq, p is its priority. **pq' ∊ PQ, pq' = pq -(e)**

- **top** (pq, index)
  - 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 , index(will be always set to 1- the indexes are the priority, because the pq will always be sorted by the priority)**
  - post: **e ∊ ↑Node**, **e** is the element with the highest priority from **pq**

- **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

- **compare(**a, b)
  - description: compare 2 nodes by priority(this is the function that sets the relation between the elements)
  - pre: **a ∊ ↑Node, b ∊ ↑Node**
  - post: **1** if [a].frequency > [b].frequency, **0** otherwise

Priority queues cannot be iterated, so they **don't have** an **iterator operation**!
The Priority Queue representation will be on a binary heap.

III. ADT Representation and operation implementation in pseudocode

Reprezentation:

- **Node:**
  letter: Char
  frequency: Integer
  code: String
  left: ↑ Node
  right: ↑ Node

- **BinaryTree:**
  root: ↑ Node

- **InorderIterator:**
  bt: Binary Tree
  st: Stack
  currentNode: ↑Node

- **PQ:**
  arr: ↑ Node[]
  len: Integer
  cap: Integer

In main:
frequencies[255]: Integer
text[1000] : char
k = 0 : Integer
pq: PriorityQueue
bt: BinaryTree
nodes[255]: ↑Node

Pseudocode

BINARY TREE:

```
subalgorithm init(bt) is:
        bt.tree = NIL
end-subalgorithm


function initLeft(bt) is:
        [root].left <- left
        initLeft <- root
end-function:


function initRight(bt) is:
        [root].right <- right
        initRight <- root
end-function:


function insertLeftSubree(bt, left) is:
        [root].left <- left
        initLeftSubree <-  root;
end-function:


function insertRightSubree(bt, right) is:
[root].right <- right
        initRightSubree <-  root;
end-function:


function initTree(bt,left, right) is:
        [root].left <- left
```

```
        [root].right <- right
        initTree <-  root;
end-function:


function root(bt) is:
        initTree <-  root;
end-function:

function initLeaf(bt,node) is:
        root <- node
        initLeaf <- rootend-function:
end-function:


function isEmpty(bt) is:
        if ([root].left = NIL && [root].right = NIL)
                isEmpty<-true
        isEmpty<- false
end-function:

function isLeaf(node)is:
        if ([node].left = NIL && [node].right = NIL)
                isLeaf<-true
        isLeaf<- false
end-function:
```

ITERATOR:

```
function getCurrent(it,bt) is:
        getCurrent <- [it].current
end-function:

function init(it,bt) is:
```

```
        [it].current<- [bt].root
end-function:


function isValid(it) is:
        if ([it].current = NIL)
                isValid<- false
        isValid<-true
end-function:


function next(param) is:
        if ([[it].current].letter != '_')
                [it].current <- tree.root

        end-if
        if ([it].current].left != NIL && param =0)
                [it].current = [it].current].left
                next <- true
        end-if

        if ([it].current].right != NIL && param =1)
                [it].current = [it].current].right
                next <- true
        end-if
end-function:
```

PRIORITY QUEUE:

```
function isEmpty(pq) is:
        if ([pq].len = 0)
                isEmpty<-true
        isEmpty<- false
```

```
end-function:


function isFull(pq) is:
        if ([pq].len = [pq].cap)
                isFull<-true
        isFull<- false
end-function:
```

```
function push(pq,newElem, index) is:
        if (index = pq.len)
                arr[++len] <- newElem
                index <- pq.len
        end-if

        if (index <= 1)
                push <- newElem
        end-if

        if (compare(arr[index / 2], arr[index]) = 1)
                aux <- arr[index / 2]
                arr[index / 2] <- arr[index]
                arr[index] <- aux
                push <- push(arr[index / 2], index / 2)
        else
                push <- newElem;
        end-if:
end-function:
```

```
function pop(pq, index) is:   //(index =1)
        ↑Node st <- arr[index * 2]
        ↑Node dr <- arr[index * 2 + 1]
        ↑Node aux
```

```
        if (index == 1)
                aux <- arr[len]
                arr[len] <- arr[1]
                arr[1] <- aux
        end-if

        if ((compare(arr[index], st) && (index * 2)<pq.len))
                aux <- arr[index]
                arr[index] <- st
                arr[2 * index] <- aux
                pop <- pop(index * 2)
        end-if

        if ((compare(arr[index], dr) && (index * 2 + 1) < pq.len))
                aux <- arr[index]
                arr[index] <- dr
                arr[2 * index + 1] <- aux
                pop <- pop(index * 2 + 1)
        else
                pop <- arr[pq.len]
        end-if
end-function:
```

MAIN:

subalgorithm **citeste()** is:

```
        print Introduceti textul: "
        read (text, 1000);
        textLen <- len(text)
        print ("Sirul citit are ‘%d’ caractere", textLen)
        for (i <- 0,textLen,1)
                frequencies[int(text[i])]++

        for ( i<- 0,255,1)
```

```
        if (frequencies[i] != 0)
            @allocate new node
            pq.push(newNode, pq.len)
end-subalgorithm
```

subalgorithm **HuffmanTree**() is:

```
    while (pq.len > 1)
        ↑Node* elem;
        ↑Node* elem2;
        elem <- pq.pop(1);
        pq.len--;
        elem2 <- pq.pop(1);
        pq.len--;
        int sumFreq <- [elem].freq + [elem2].freq;
        Node* elem3 <- new Node{ (char)('_'), sumFreq };
        [elem3].left <- elem;
        [elem3].right <- elem2;
        pq.push(elem3,pq.len)
    end-while
end-subalgorithm:
```

subalgorithm **preorder_recursive**(node) is:

```
    if (node.isLeaf(node) = false)
        if ([node].left != NIL)
            [[node].left].code += [node].code;
            [[node].left].code += "0";
            preorder_recursive([node].left);

        if (node->right != NULL)
            [[node].right].code += [node].code;
            [[node].right].code += "1";
            preorder_recursive([node].right);
    else
```

```
                Print([node].code)
                nodes[k] = node
                k++
        end-if
end-subalgorithm:


subalgorithm preorderRec(tree) is:
        preorder_recursive([tree].root)
end-subalgorithm:


function encoding() is:
        string str = "";
        for (i <- 0,len(text),1)
                for (j <- 0,k,1)
                        if (text[i] = (char)([nodes[j]].letter))
                                str += [nodes[j]].code
                        end-if
                end-for
        end-for
        encoding <- str
end-function


function decoding(code)
        decode = ""
        Iterator it{ bt }
        node = it.getCurrent()
        for (i <- 0,i < code.length(),1)
                if (code[i] = '0')
                        it.next(0)
                        node <- it.getCurrent()

                if (code[i] = '1')
```

```
                it.next(1)
                node <- it.getCurrent()

            if (node->isLeaf(node))
                decode += (char)([node].letter)
    decode <-decode
end-function
```

```
function main() is:
    citeste()
    HuffmanTree();

    bt.root = pq.top(1);
    Print("Root: ")
    Print([bt.root].toString(bt.root))
    preorderRec(bt)
    string encode = encoding()
    string decode = decoding(encode)
    Print(encode)
    Print(decode)
end-function
```

IV. Tests

Binary Tree:

```
    BinaryTree bt{};
    Iterator it{ bt };
    assert(bt.root == NULL);
    bt.rightTree()->letter = 'n';
    assert(bt.rightTree()->letter == 'n');
    bt.rightTree()->letter = 'm';
```

```
assert(bt.leftTree()->letter == 'm');
Node* node1 = new Node{ 'a', 1 };
bt.initLeaf(node1);
Node* node2 = new Node{ 'b', 1 };
assert(bt.initLeaf(node1) == node1);
assert(bt.initTree(node1, node2) == bt.root);
```

Priority Queue:

```
PriorityQueue pq{};
Node* zero = new Node{ '-', 0 };
Node* node1 = new Node{ 'a', 1};
Node* node2 = new Node{ 'b', 2 };
Node* zr = pq.push(zero, pq.len);
assert(pq.len == 1);
Node* a = pq.push(node1, pq.len);
assert(pq.len == 2);
Node* b = pq.push(node2, pq.len);
assert(pq.len == 3);

Node* elem = pq.pop(1, pq.len);
assert(pq.len == 2);
Node* elem2 = pq.top(1);
assert(pq.len == 2);
```

Iterator:

```
BinaryTree bt{};
Iterator it{ bt };

Node* node1 = new Node{ 'a', 1 };
Node* node2 = new Node{ 'b', 1 };
Node* node3 = new Node{ 'c', 1 };
bt.root = node1;
bt.root->right = node2;
```

```
assert(it.getCurrent() == bt.root);
it.next(1);
assert(it.getCurrent() == node2);
assert(it.isValid() == true);
```

V. ADT Complexities

Priority Queue:

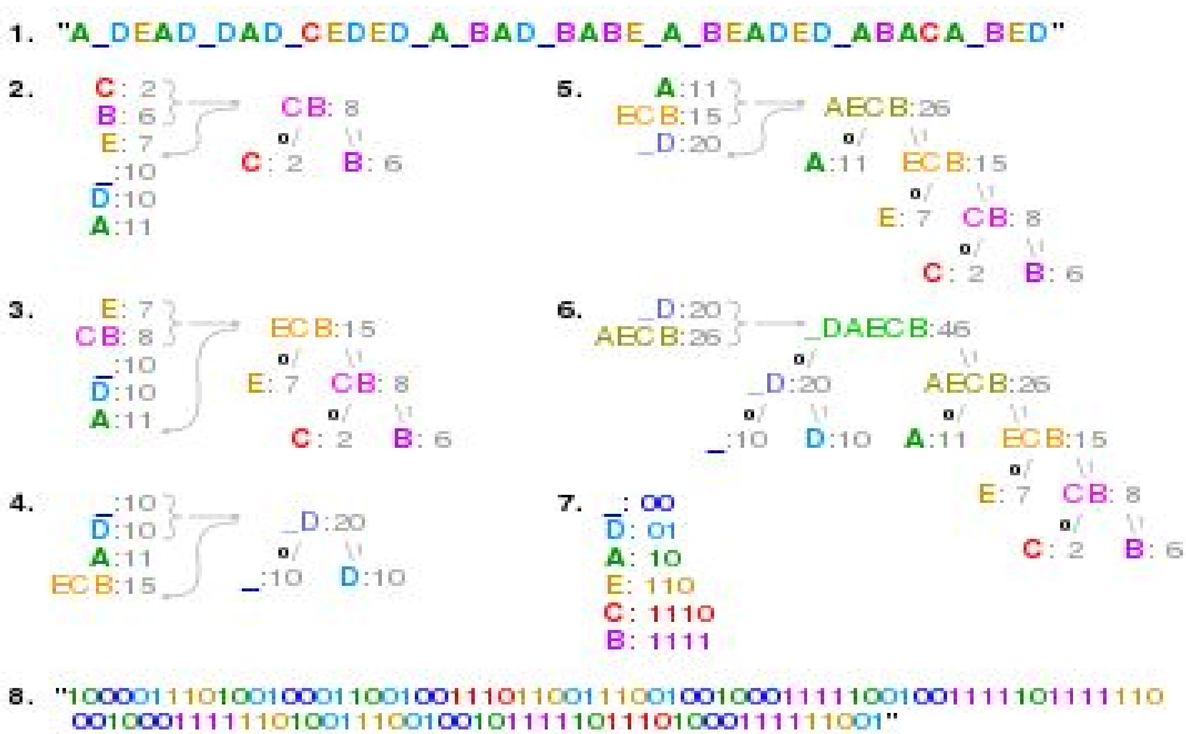
Operation	
push	$O(\log 2n)$
pop	$O(\log 2n)$
top	$\Theta(1)$

Iterator:

Operation	
init	$\Theta(1)$
getCurrent	$\Theta(1)$
next	$\Theta(1)$
isValid	$\Theta(1)$

Binary Tree:

Operation	
root	$\Theta(1)$
leftTree	$\Theta(1)$
rightTree	$\Theta(1)$
initLeaf	$\Theta(1)$
initTree	$\Theta(1)$
initLeftSubree	$\Theta(1)$
initRightSubree	$\Theta(1)$
isEmpty	$\Theta(1)$



Thank you for your review!

The End