Appendix A

Multi-level index operations

Algoritmo A.1 B-Tree look up algorithm

Require: $k : int \neq null$ //looking up within the node
1: **for all** $e \in Elements$ **do**
2: **if** $e.getKey() = k$ **then**
3: **return** e
4: **end if**
5: **end for**
6: **if** node is leaf **then**
7: **return** $null$
8: **end if**
9: $i \Leftarrow 0$ //looking up within child nodes
10: **for all** $e \in Elements$ **do**
11: **if** $k < e.getHey()$ **then**
12: **return** $Childs\,[i]\,.lookup(k)$//left child's lookup result is returned
13: **end if**
14: $i \Leftarrow i + 1$
15: **end for**
16: **return** $Childs\,[iright]\,.lookup(k)$ //last child's lookup result is returned

Algoritmo A.2 B-Tree insertion algorithm

```
1: if node is leaf then
2:   if node is not full then
3:     newElement is inserted in Elements keeping the ascendant order
4:   else
5:     The set of Elements is ordered with newElement and the median element is chosen
       as the new parent of less and greater elements contained in new nodes, given as result
       the nodes newLeftChild and newRightChild as children.
6:     return newElement, newLeftChild, newRightChild
7:   end if
8: end if
9: for all e ∈ Elements do
10:  if newElement.getKey() < e.getKey() then
11:    The newElement is inserted into the left child
12:    Exit for all
13:  end if
14: end for
15: if newElement have not been inserted then
16:  The newElement is inserted into the last child of node
17: end if
18: if insertion have produced a splitting then
19:  if node is not full then
20:    newElement is inserted into Elements keeping the ascendant order
21:  else
22:    The set of Elements is ordered with newElement and the median element is chosen
       as the new parent of less and greater elements contained in new nodes, given as result
       the nodes newLeftChild and newRightChild as children.
23:    return newElement, newLeftChild, newRightChild
24:  end if
25: end if
```

Algoritmo A.3 B-Tree deletion algorithm

```
1: if key = e_i.getKey() ∧ e ∈ Elements then
2:    if node is a leaf then
3:       Delete element from leaf node
4:       return
5:    else
6:       leftChild ⇐ Childs[i]
7:       rightChild ⇐ Childs[i + 1]
8:       if ( number of elements of leftChild+ number of elements of rightChild) ≤
         BTree.grade − 1 then
9:          Merge leftChild and rigtChild into leftChild
10:      else
11:         lookup the greatest element of leftChild's subtree as a newSeparator
12:         The newSeparator is deleted from his node
13:         delete(newSeparetor.getKey())
14:         and chosen as a parent of the leftChild's subtree and rigthChild's subtree.
15:         newSeparetor takes up the place of deleted element
16:      end if
17:   end if
18: else
19:   for all e ∈ Elements do
20:      if key < e.getKey() then
21:         Childs[i].delete(k)
22:         i ⇐ i + 1
23:         Exit for all
24:      end if
25:      i ⇐ i + 1
26:   end for
27:   Childs[i].delete(k)
28:   leftChild ⇐ Childs[i − 1]
29:   rightChild ⇐ Childs[i]
30:   if number of elements of leftChild < BTree.grade or number of elements of
      rightChild < BTree.grade then
31:      hola
32:   end if
33: end if
```