Problem statement + Justification:

There is a market. Clients come to buy things. Everytime a client comes he recieves a random number. This number is compared to others number(who are already in priority queue) by a relation which is given. The relation can be "<" or ">". With this relation we know who has priority to buy . The owner of the market calls the number with the highest priority(if the relation is "<" the smallest number, if the relation is ">" the largest number) out of all the numbers taken at that moment. The person with that number can do his buyings.Then he calls another number.

It is a good problem for priority queue because as soon as someone comes to the store we place him somewhere based on the priority he recieves. We use a priority queue and not a queue because in a queue if someone comes he will do his buyings based on how early he reached the store and not based on the random number he recieves.

Interface:

The domain of the ADT Priority Queue:

PQ = {pq|pq is a priority queue with elements (e, p), e $\in$

TElem, p $\in$ TPriority}

The interface of the ADT Priority Queue contains the

following operations:

init (pq, R)

Description: creates a new empty priority queue

Pre: R is a relation over the priorities,

R : TPriority $\times$ TPriority

Post: pq $\in$ PQ, pq is an empty priority queue

$\Theta(1)$

destroy(pq)

Description: destroys a priority queue

Pre: pq ∈ PQ

Post: pq was destroyed

$\Theta(1)$

push(pq, e, p)

Description: pushes (adds) a new element to the priority queue

Pre: pq ∈ PQ, e ∈ TElem, p ∈ TPriority

Post: pq0 ∈ PQ, pq0 = pq ⨁ (e, p)

$O(\log_2 n)$

n-number of elements in the priority queue

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, pq is not empty

Post:e ∈ TElem, p ∈ TPriority, e is the element with the highest priority from pq, p is its priority.

pq0 ∈ PQ, pq0 = pq - (e, p)

Throws: an exception if the priority queue is empty.

$O(\log_2 n)$

n-number of elements in the priority queue 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, pq is not empty

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.

$\Theta(1)$

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

$\Theta(1)$

Representation:

Node:

e:TElem

p:TPriority

left: ↑ Elem

right: ↑ Elem

PriorityQueue:

rel: ↑ Relation

root: ↑ Node

Specifications:

The ADT Priority Queue is a container in which each element has an associated priority (of type TPriority).

In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.

Because of this restricted access, we say that the Priority Queue works based on a HPF - Highest Priority First policy.

In order to work in a more general manner, we can define a relation R on the set of priorities: R : TPriority × TPriority

When we say the element with the highest priority we will mean that the highest priority is determined using this relation

R.

If the relation R = ”≥”, the element with the highest priority

is the one for which the value of the priority is the largest

(maximum).

Similarly, if the relation R = ”≤”, the element with the

highest priority is the one for which the value of the priority is

the lowest (minimum).

Pseudocode:

Subalgorithm init(pq,r) is:

    head ← NIL

    rel ← r

End-subalgorithm

Complexity:θ(1)

Subalgorithm destroy(pq) is:

    head ← NIL

End-subalgorithm

Complexity:θ(1)

function push(pq,e,p) is:

    if root = NIL then

```
        root ← node
        push ← true
    end-if
    else
        ok ← 1
        cn ↑ BSTNode(e,p)
        cn ← root
        while cn.getLeft() != nullptr or cn.getRight() != NIL and ok = 1 execute:
            if node.getTPriority().relation(cn.getTPriority(), rel) = 1 and cn->getLeft() = NIL then
                cn.setLeft(node)
                ok ← 0
                push ← true
            end-if
            else if node.getTPriority().relation(cn.getTPriority(), rel) = 0 and cn.getRight() = NIL then
                cn.setRight(node)
                ok ← 0
                push ←true
            end-if
            else if (node.getTPriority().relation(cn.getTPriority(), rel) = 1) then
                cn ← cn.getLeft()
            end-if
            else if (node.getTPriority().relation(cn.getTPriority(), rel) = 0) then
                cn ← cn.getRight()
            end-if
        end-while
```

```
        if ok = 1 then
            if node.getTPriority().relation(cn.getTPriority(), rel) = 1 then
                cn.setLeft(node)
                push ← true
            end-if
            else
                cn.setRight(node)
                push ← true
            end-if
        end-if
    end-if
    push ← false
End-subalgorithm
Complexity:O(log2n)

function pop(pq,e,p) is:
    if root = NIL then
        /*throw std::exception("Value not found");*/
        pop   ← false
    end-if
    else
        cn ↑ BSTNode(e,p)
        if root.getRight() = NIL and root.getLeft() = NIL then
            root   ← nullptr
            return true
        end-if
```

```
else if root.getRight() != NIL then
        cn ← root.getRight()
        if cn.getLeft() = NIL then
                cn.setLeft(root.getLeft())
                root ← cn
        end-if
        pop ← true
end-if
else if root.getLeft() != NIL then
        cn ← root->getLeft()
        if cn.getRight() = NIL then
                cn.setRight(roo.getRight())
                root ← cn
        end-if
        pop ← true
end-if
else
if root.getRight() = NIL and root.getLeft() != NIL then
        cn ← root.getLeft()
        while cn.getRight().getRight() != NIL execute:
                cn ← cn.getRight()
        end-while
        cn.getRight().setLeft(root.getLeft())
        cn.getRight().setRight(root.getRight())
        root ← cn.getRight()
        cn.setRight(NIL)
```

```
                pop ← true
        end-if
        else if root.getLeft() = NIL and root.getRight() != NIL then
                cn ← root.getRight()
                while cn.getLeft().getLeft() !=NIL execute:
                end-while
                        cn ← cn.getLeft()
                end-while
                cn.getLeft().setLeft(root.getLeft())
                cn.getLeft().setRight(root.getRight())
                root ← cn.getLeft()
                cn.setLeft(NIL)
                pop ← true
        end-if
    end-if
    pop   ← false
End-function
```

Complexity:O(log2n)

```
function top(pq,e,p) is:
    if root = NIL then
        @throw an exception
    else
        cn ← root
        top ← cn
    end-if
```

```
End-function
Complexity:θ(1)


function isEmpty(pq) is:
        if root = NIL then
                isEmpty ← true
        else
                isEmpty ← false
        end-if
End-function
Complexity:θ(1)


Subalgorithm setRoot(pq,node) is:
        root ← node
End-subalgorithm
Complexity:θ(1)

function getRoot(pq) is:
        getRoot ← root
End-function
Complexity:θ(1)

Subalgorithm setRelation(pq,r) is:
        rel ← r
```

End-subalgorithm

Complexity:θ(1)

function getRelation(pq) is:

getRelation ← rel

End-function

Complexity:θ(1)

function search(pq,p) is:

```
    if root = NIL then
        search ← NIL
    else
        cn ↑ BSTNode(e,p)
        cn ← root
        while cn != NIL execute:
            if p.relationEqual(cn.getTPriority()) = 1 then
                search ← cn
            end-if
            if p.relation(cn->getTPriority(), rel) = 1 then
                cn ← cn.getLeft()
            end-if
            else if p.relation(cn->getTPriority(), rel) = 0 then
                cn ← cn.getRight()
            end-if
        end-while
    end-if
```

```
        search ← NIL
End-function
Complexity:O(log2n)


function getParent(pq,node) is:
        cn ↑ BSTNode
        cn ↑ root
        if parent = node then
                getParent ← NIL
        end-if
        else
                while parent != NIL and parent.getLeft() != node and parent.getRight() != node
execute:
                        if node.getTPriority().relation(parent.getTPriority(), rel) = 1 then
                                parent ← parent.getLeft()
                        else
                                parent ← parent.getRight()
                getParent ← parent
        end-if
End-function
Complexity:O(log2n)


function getMin(pq) is:
        currentNode ↑ BSTNode
```

```
        currentNode ↑ root
        while currentNode.getLeft() != NIL execute:
                currentNode ← currentNode->getLeft()
        getMin ← currentNode
End-function
```

Complexity:O(log2n)

```
function getMax(pq) is:
        currentNode ↑ BSTNode
        currentNode ↑ root
        while currentNode.getRight() != NIL execute:
                currentNode ← currentNode->getRight()
        getMax ← currentNode
End-function
```

Complexity:O(log2n)