PROJECT DOCUMENTATION

DATA STRUCTURES AND ALGORITHMS

MOIS BRIGITTE-IZABELLA
GROUP 812
MATHEMATICS AND COMPUTER SCIENCE

Project documentation

Data structures and algorithms

Name: Mois Brigitte-Izabella
Group: 812

Contents

Mois Brigitte-Izabella
Group 812
Task 21

1) Task

Task 21 - ADT Priority Queue - implementation on a binary search tree

2) ADT Specification

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

TElem -> the general element of the container
The interface of the TElem contains the following operations:

- *Assignment (* $e_1 \leftarrow e_2$ *)*
  - ***Pre**:* $e_1, e_2 \in$ *TElem*
  - ***Post**:* $e_1 = e_2$
- *Equality test (* $e_1 = e_2$*)*
  - ***Pre**:* $e_1, e_2 \in$ *TElem*
  - ***Post**: equal =* $\begin{cases} \text{True, if } e_1 = e_2 \\ \text{False, otherwise} \end{cases}$

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

3) ADT 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 ∈ PQ, pq is an empty priority queue
  - **Throws:** -

- destroy (pq)
  - **Description:** destroys a priority queue
  - **Pre:** pq ∈ PQ
  - **Post:** pq was destroyed
  - **Throws:** -

- 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)
  - **Throws: -**

- 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 $\leftarrow \begin{cases} true, if\ pq\ has\ no\ elements \\ false, otherwise \end{cases}$
  - **Throws: -**

- isFull (pq)
  - **Description:** checks if the priority queue is full
  - **Pre:** pq ∈ PQ
  - **Post:** isFull $\leftarrow \begin{cases} true, if\ pq\ is\ full \\ false, otherwise \end{cases}$
  - **Throws: -**

4)ADT Representation – *binary search tree*

TNode

info: *TElement*

left: ↑*TNode*

right: ↑*TNode*

TElement

p: *int* // every TElement has a priority level

info: *string*

r: *rel {R: TPriority X TPriority}* // I have overridden the < operator

PQBST

root: ↑*TNode*

m: *int* // The given size of the container

5)Pseudocode implementation

```
Subalgorithm init(pq) is:

      @allocate root of type Node
      pq.root = NIL;
      pq.s = 0;

end-subalgorithm
```

Complexity: **θ(1)**

```
Subalgorithm destroy(pq) is:
      @recursively deallocate left, right nodes and lastly the root, while we
      check if they’re NIL or not
end-subalgorithm
```

Complexity: **O(n)**

```
Subalgorithm push(pq,e,p) is:   //wrapper method
      if pq.root = NIL then
            @allocate newNode of type Node
            pq.root ← newNode(e);
      else
            push(e,root);
      end-if
end-subalgorithm
```

Complexity: **O(n)**

```
Subalgorithm push(pq,e,node) is:  //private method
      if e.info < [node.info].info then
            if node.left = NIL then
                  @allocate newNode of type Node
                  node.left ← newNode(e);
            else
                  push(e,node.left);
            end-if
      else
            if node.right = NIL then
                  @allocate newNode of type Node
                  Node.right ← newNode(e);
            else
                  push(e,node.right);
            end-if
      end-if
end-subalgorithm
```

Complexity: **O(n)**

```
Function pop(pq,e,p) is:    //wrapper method

      elem : TElement;
      if pq.root = NIL then
            @throw exception
      else
            if [pq.root].right = NIL then
                  elem ← [pq.root].info;
                  if [pq.root].left = NIL then
                        @deallocate pq.root
                        root ← NIL;
                  else
                        @allocate temp of type Node
                        temp ← pq.root;
                        pq.root ← [pq.root].left;
                        @deallocate temp
                  end-if
            else
                  elem ← pop([pq.root].right, pq.root);
            end-if
      end-if
      pop ← elem;

end-function
```

Complexity: **O(n)**

```
Function pop(node, parent) is:   //private method

      elem : TElement;
      if node.right != NIL then
            elem ← pop(node.right, node);
      else
            elem ← node.info;
            if node.left = NIL then
                  parent.right ← NIL;
            else
                  parent.right ← node.left;
            end-if
            @deallocate node;
      end-if
      pop ← elem;

end-function
```

Complexity: **O(n)**

```
Function top(pq,e,p) is:

      if pq.root = NIL then
            @throw exception
      else
            @allocate currentNode of type Node
            currentNode ← pq.root;
            while currentNode.right != NIL execute
                  currentNode ← currentNode.right;
            end-while
            top ← currentNode.info;
      end-if

end-function
```

Complexity: **O(n)**

```
Function isEmpty(pq) is:

      isEmpty ← pq.root = NIL;

end-function
```

Complexity: **θ(1)**

```
Function isFull(pq) is:

      isFull ← false;

end-function
```

Complexity: **θ(1)**

6)Tests for the container

```
PQBST pqbst{};  //we're checking the init method
assert((pqbst.isEmpty() == true)); //we're checking the isEmpty method
Human h(2, "Mois Brigitte");
Human h2(3, "Michael Jackson");
Human h3(0, "Mois Kiwi");
Human h4(1, "Muresan Ioana");
Human h5(4, "Elena Ceausescu");
pqbst.push(h); //we're checking the push method
pqbst.push(h2);
assert((pqbst.top() == h2)); //we're checking the top method
assert((pqbst.pop() == h2)); //we're checking the pop method
PQBST pqbst2(NULL, 5);
pqbst.setM(3); //we're checking the setM method
pqbst.push(h3);
assert((pqbst2.isEmpty() == true)); //double-check
assert((pqbst.pop() == h)); //in the following lines I'll be checking all the sub-
                            branches of the pop method (the right son having a
                            left son and so on)
assert((pqbst.pop() == h3));
pqbst.push(h);
pqbst.push(h5);
pqbst.push(h2);
pqbst.push(h4);
pqbst.push(h3);
assert((pqbst.pop() == h5));
assert((pqbst.pop() == h2));
assert((pqbst.pop() == h));
assert((pqbst.pop() == h4));
assert((pqbst.pop() == h3));
pqbst.push(h);
pqbst.push(h2);
pqbst.push(h5);
pqbst.push(h4);
pqbst.push(h3);
assert((pqbst.pop() == h5));
assert((pqbst.pop() == h2));
assert((pqbst.pop() == h));
assert((pqbst.pop() == h4));
assert((pqbst.pop() == h3));
pqbst.push(h5);   //we're not leaving the queue empty so that the destroy function
                    can be asserted automatically
```

7)Operations complexity

`init(pq) ->` **θ(1)**

`destroy(pq) ->` **O(n)**

`push(pq,e,p) ->` **O(n)**

`pop(pq,e,p) ->` **O(n)**

`top(pq,e,p) ->` **O(n)**

`isEmpty(pq) ->` **θ(1)**

`isFull(pq) ->` **θ(1)**

I will make the computations for the complexity of the function push(pq,e,p).

Best case: **θ(1)** – when the tree is empty

Worst case: **θ(n)** – when we have to traverse all the nodes

Average case: $T(n) = (n-1) + \frac{2}{n}(T(0)+T(1)+\ldots+T(n-1))$

$T(n)/n \in \theta(\log n)$

8)Problem statement - Rescuing the human race based on status

An extraterrestrial species has attacked planet Earth and it is about to be destroyed. Unfortunately, only *m* humans can embark on the rescue spaceship, based on their priority levels, as follows:

Politicians – priority level 4

Scientists – priority level 3

Doctors – priority level 2

Priests – priority level 1

Commoners – priority level 0

Simulate the process of rescuing the human race by adding a human to the spaceship, removing a human from the spaceship and displaying all the humans that have been saved.

9) Problem explanation

My assigned ADT, the priority queue is suitable for solving the chosen problem because each human has a priority level and they can only embark the spaceship if they have the highest priority level.

In other words, we start with an empty priority queue (an empty spaceship) and first we "push" the politicians, then the scientists, then the doctors, then the priests and then the commoners, while we check not to have a full spaceship.

10) Problem solution

```
Algorithm rescueHumanRace() is:

	createPQWithHumans(pq);
	@read m
	displayRescuedHumans(pq,m);

end-algorithm
```

Complexity: **θ(1)**

```
Subalgorithm createPQWithHumans(pq) is:

	@read n;
	init(pq,">=");
	for 1 ← 1,n execute
		@read human and priority
		push(pq, human, priority);
	end-for

end-subalgorithm
```

Complexity: $\mathbf{O(n^2)}$

```
Subalgorithm displayRescuedHumans(pq,m) is:

       While m>0 and  ¬ isEmpty(pq) execute
              pop(pq, human, priority);
              @print human and priority
              m ← m-1;
       end-while
       destroy(pq);

end-subalgorithm
```

Complexity: **O(n*m)**

11) Solution complexity

rescueHumanRace() -> **O(n^2)**

createPQWithHumans(pq) -> **O(n^2)**

displayRescuedHumans(pq,m) -> **O(n*m)**