DATA STRUCTURES AND ALGORITHMS

Extra reading 6

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

Content

- This extra reading contains my solution for the *WashingMachine* problem from Extra reading 5. If you are not familiar with the problem, read that extra reading first.
- In the first part I will describe how to solve the problem using containers (that we assume are already implemented), while in the second part, I will present how to solve the problem using a data structure.

Requirements - Recap I

- We want to implemented the class *WashingMachine*, which stores a fixed list of programs for the washing machine, and when the machine is turned on has a current program which is displayed.

Requirements - Recap II

- *WashingMachine* has the following operations:
  - *init* - the constructor. It receives a list of programs, which are the programs that this washing machine has.
  - *turnOn* - operation which turns on the washing machine. At this moment the *Eco 40-60* program is the current program.
  - *start* - start the current program (this will influence the frequency of using the current program)
  - *turnLeft* - turn the program selector knob to the left, which will change the current program to the previous one. If the current program was the first one, it will become the last program.
  - *turnRight* - turn the program selector knob to the right, which will change the current program to the next one. If the current program was the last one, it will become the first program.
  - *getCurrentProgram* - returns the current program (so that it can be displayed).

Requirements - Recap III

- The order in which the programs are considered has to be based on how many times the program was started since the creation of the washing machine.
- If two or more programs were started the same number of times, they can appear in any order.
- **Notes:** Normally, the washing machine should have a *turnOff* operation as well, and *start*, *turnLeft*, *turnRight* and *getCurrentProgram* can only be called if the washing machine is turned on. Since we focus on the part which is relevant from a container/data structure perspective, we are going to ignore these restrictions.

Components of the solution

- If we want to solve the problem, there are two issues that we need to handle:
  - How to store the programs such that we can display them based on the number of times they were started.
  - How to represent that *current program*.

Container - level solution I

- If we want to be able to display the programs based on how many times they were started, first of all we need to store for each program the number of times it was started $\rightarrow$ we need *program name - nr. of times started* pairs (which will be called from now on *program - frequency* to simplify things).
- At this point it sounds like a map with *key - value* pairs, where the program is the key (since it is unique) and the frequency is the value.
- The only problem with this approach is that we need to sort the programs by the frequency, and this is not possible in case of a map and not possible in case of a sorted map, which is sorted by the keys.

Container - level solution II

- What if we reverse things and have the *frequency* as key, to be able to sort by it, and the program name as value? It is possible, but then the keys are not going to be unique. This does not fit the characteristics of a (sorted)map, but it fits a (sorted)multimap.
- The SortedMultiMap (*SMM* from now on) will handle the details of keeping the programs in the right order, but we still need to handle the *Eco 40-60* program separately, since, by the requirement, this is always displayed as the first program, no matter how many times it was started.
- The easiest way to do this, is to assign a *frequency* to the *Eco 40-60* program, so that it will always be the first pair in the SMM.

Container - level solution III

- Now that we have decided how to store the elements, we can focus on the second component of the problem: the *current program*. If we read the requirements related to the *current program*, we can realize that it sounds very similar to an iterator. So, the simplest way to represent it is to take an iterator over the SMM.

- However, since the knob can turn both to left and right, it needs to be a bidirectional iterator.

Container - level solution IV

- Let's see then how the representation of the WashingMachine looks like:

WashingMachine:
elements: SMM
currentP: BidirectionalSMMIterator

- And let's see the implementation of the operations for the WashingMachine: *init*, *turnOn*, *start*, *turnLeft*, *turnRight*, *getCurrentProgram*.

Container - level solution V

- Implementation of *init*
  - We get as parameter the list of existing programs, we just need to create the SMM with the frequency and program pairs.
  - We need to define the relation for the SMM as well. We simply want to order descendingly by the frequencies.
  - Initial frequencies are 0, except for the *Eco 40-60* program, for which we will take a large value, for example, 100 000 as frequency, to make sure that it will have the highest frequency.
  - **Note:** Hard-coding values (like the 100 000 for frequency) is not a good practice. Nevertheless, a frequency of 100000 will probably be enough. If you run the same program once every day, you still need more than 270 years to achieve that frequency.

Container - level solution VI

```
function relation(v1, v2):
    relation ← v1 ≥ v2
end-function
```

```
subalgorithm init(wm, programs)
    //pre: wm is a WashingMachine, programs is a list of programs
    init(wm.smm, relation)
    for i ← 1, size(programs) execute
        pr ← getElement(programs, i)
        if pr = "Eco 40-60" then
            add(wm.smm, 100000, pr)
        else
            add(wm.smm, 0, pr)
        end-if
    end-for
end-subalgorithm
```

Container - level solution VII

- Implementation of *turnOn*
  - When the washing machine is turned on, the *currentP* iterator needs to be initialized. When it is initialized, it will automatically point to the first element of the SMM, which is the *Eco 40-60* program

```
subalgorithm turnOn(wm):
    iterator(wm.smm, wm.currentP)
end-subalgorithm
```

Container - level solution VIII

- Implementation of *turnRight*
  - Turning the knob to right means changing the current program to the next one. This is simple, since we have an iterator for the current program and iterators have a function to go to the next element.
  - However, a special case is when the current program is the last one. By the specifications of the WashingMachine, in this case, the current program should jump back to the first one, while an iterator becomes invalid when next is called while it is positioned on the last element.
  - We will solve this by checking after next whether the iterator is invalid, and if it is, we just reset it to the beginning, using the *first* function.

Container - level solution IX

```
subalgorithm turnRight(wm):
    next(wm.currentP)
    if not valid(wm.currentP) then
        first(wm.currentP)
    end-if
end-subalgorithm
```

Container - level solution X

- Implementation of *turnLeft*
  - Very similar to the implementation of turnRight. Since it is a bidirectional iterator, *currentP* has an operation to go to the previous element.
  - Again, special care is needed to be taken when the iterator is set to the first element. Calling previous on it will make it invalid, and we need to set it to the last element. For this, we will assume that we have a function called *last* (the pair of *first*), which sets the current element of the iterator to the last element.

Container - level solution XI

- What if we do not have an operation *last*? In this case, we either need to know the number of programs (something we do not have now, but should be kept as part of the WashingMachine if there is no *last* function in the iterator) so that we can create an iterator and call *next* on it the corresponding number of times to be set on the last program, or we can do something similar to the method of stick: create an iterator and call next on it, then reintialize currentP to be the first element and call next on the two iterators in parallel. Since the first is one step in front of the second, when the first is invalid, the second is set on the last element.

Container - level solution XII

```
subalgorithm turnLeft(wm):
    previous(wm.currentP)
    if not valid(wm.currentP) then
        last(wm.currentP)
    end-if
end-subalgorithm
```

Container - level solution XIII

- Implementation of *getCurrentProgram*
  - This is easy, we just get the current element from the iterator and return its value, which is the actual program. Nobody cares for the key, the frequency.
  - Since *turnLeft* and *turnRight* are implemented in such a way that they never leave the iterator invalid, we can just directly call *getCurrent*, no need to check if it is valid.

```
function getCurrentProgram(wm):
    <fr, pr> ← getCurrent(wm.currentP)
    getCurrentProgram ← pr
end-function
```

Container - level solution XIV

- Implementation of *start*
  - When we start the current program, its frequency needs to be incremented. In a multimap we cannot just change the key of a pair (like we can do with the value of a key), we need to remove the current element, increment the frequency and add it again to the smm.
  - But what happens with the current program? You should not modify a container if you are iterating over it. However, when a program is started, we can no longer turn the knob to left or right. If you think about a real washing machine, you need to turn it on again to be able to select programs. But just to be safe, we will reset the iterator to the beginning.

Container - level solution XV

```
subalgorithm start(wm):
    < fr, pr >← getCurrent(wm.currentP)
    remove(wm.smm, fr, pr)
    add(wm.smm, fr+1, pr)
    iterator(wm.smm, wm.currentP)
end-subalgorithm
```

Container - level solution alternatives

- Instead of using a SMM, we could have used one of two other containers:
  - ADT List
  - ADT SortedList
- These containers are not designed for *key-value* pairs, but you can always define a structure to hold a key and a value together and have lists where the elements are of that type.
- In both cases, the *currentP* could have been an index (or an iterator just like in case of the SMM)
- If we used a SortedList (obviously, sorted by the frequencies), the idea of the code would have been the same (especially in case of start: remove the current element, increase frequency and add it back)
- If we used a List, when the current program is started we could increment the frequency and then swap the current element with its previous one(s) until the list is sorted.

Data structure - level solutions I

- If we want to implement the WashingMachine directly on a data structure, we have two alternatives:
  - Dynamic array
  - Doubly - linked list (due to the fact that we need to traverse the program list in both directions, a singly - linked list is not a good option).
- Both data structures are equally good, we can implement the operations with good complexity in both cases. Possibly dynamic arrays have the advantage of caching, and since we never need to insert elements to the middle or beginning we can use a dynamic array efficiently.

Data structure - level solutions II

- **Obs:** In case of the SMM we did remove and insert key - value pairs, but when we work on a data structure level, this is not required, it is enough to swap elements from different positions and this can be done in case of a DLL as well. Moreover, in case of a DLL we actually have two options: we can swap info between nodes, or we can actually rearrange the nodes, if the information stored in it is large and it is inefficient to copy it.

- In the following we will discuss the implementation on a DLL, simply because it is more interesting from an algorithmic perspective.

Data structure - level solutions III

- So, for the representation of the Washing Machine, we will have the following structures:

WMNode:
info: String //I only retain the name of the program
fr: Integer
next: ↑ WMNode
prev: ↑ WMNode

WashingMachine:
head: ↑ WMNode
tail: ↑ WMNode
currentP: ↑ WMNode

- And let's see the implementation of the operations for the WashingMachine: *init*, *turnOn*, *start*, *turnLeft*, *turnRight*, *getCurrentProgram*.

Data structure - level solutions IV

- Implementation of *init*
  - We get as parameter the list of existing programs, we just need to create the a DLL from it.
  - Since this is not a generic DLL or generic container, and we know that we are going to compare frequencies, we do not need a relation.
  - Since we have access to the nodes, we could treat the *Eco 40-60* program separately, but it is a lot easier if we follow the same strategy as in the first implementation: make its frequency very large.
  - Initial frequencies are 0, except for the *Eco 40-60* program, for which we will take a large value, ex. 100 000 as frequency, to make sure that it will have the highest frequency.

Data structure - level solutions V

- An alternative approach would be to have a frequency of the *Eco 40-60* program which is not 0 initially, but not that big either, but whenever a program is started, besides incrementing its frequency, we can increment the frequency of *Eco 40-60*. *Eco 40-60* beeing the head of the list, we have direct access to it. This way, it will always have a frequency larger than any other program.

- The above approach would have been a little more complicated in case of the SMM, where we do not have direct access to the elements. We can access a value by key, which is the frequency, and, especially when we keep incrementing the frequency of the *Eco 40-60* program, who knows what its frequency is, to increment it? We could have accessed it by creating an iterator over the SMM (other than the *currentP*), which, when created, points to the first element.

Data structure - level solutions VI

- The *Eco 40-60* program will be added as the first node, the rest will be added at the end, and since all have initially frequency 0, the list is sorted.

Data structure - level solutions VII

```
subalgorithm init(wm, programs)
    //pre: wm is a WashingMachine, programs is a list of programs
    wm.head ← NIL
    for i ← 1, size(programs), 1 execute
        pr ← getElement(programs, i)
        newNode ← allocate()
        [newNode].info ← pr
        [newNode].next ← NIL
        [newNode].prev ← NIL
        if pr = "Eco 40-60" then //add as first
            [newNode].fr ← 10
            [newNode].next ← wm.head
            if wm.head = NIL then
                wm.head ← newNode
                wm.tail ← newNode
            else
//continued on the next slide
```

Data structure - level solutions VIII

```
            [wm.head].prev ← newNode
            wm.head ← newNode
        end-if
    else //not the Eco 40-60 program, add as last
        [newNode].fr ← 0
        if wm.head = NIL then
            wm.head ← newNode
            wm.tail ← newNode
        else
            [newNode].prev ← wm.tail
            [wm.tail].next ← newNode
            wm.tail ← newNode
        end-if
    end-if
  end-for
  currentP ← wm.head
end-subalgorithm
```

Data structure - level solutions IX

- Complexity of *init*: $\Theta(nr\ of\ programs)$
- **Obs.** When the WashingMachine was implemented on a SMM, we could not talk about the complexities of the operations, because we did not know how the SMM was implemented. However now, when we work directly on DS-level, we can discuss the complexities of the operations.

Data structure - level solutions X

- Implementation of *turnOn*
  - When the washing machine is turned on, the *currentP* node needs to be initialized. We simply make it point to the head of the list.

```
subalgorithm turnOn(wm):
    wm.currentP ← wm.head
end-subalgorithm
```

- Complexity: $\Theta(1)$

Data structure - level solutions XI

- Implementation of *turnRight*
  - Turning the knob to right means changing the current program to the next one. This is simple, we just go to the next node.
  - However, a special case is when the current program is the last one. By the specifications of the WashingMachine, in this case, the current program should jump back to the first one.
  - This can be solved in two very similar ways:
    - Check if *currentP* is the tail, if it is, set it to head
    - Go to the next node with *currentP*, check if it is NIL, if it is, set it to head

Data structure - level solutions XII

```
subalgorithm turnRight(wm):
    wm.currentP ← [wm.currentP].next
    if wm.currentP = NIL then
        wm.currentP ← wm.head
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

Data structure - level solutions XIII

- Implementation of *turnLeft*
  - Very similar to the implementation of turnRight. Since we have a doubly linked list, *currentP* can go to the previous node.
  - Again, special case is when the current program is set to the first element, since in this case it needs to be reset to the tail.

Data structure - level solutions XIV

```
subalgorithm turnLeft(wm):
    wm.currentP ← [wm.currentP].prev
    if wm.currentP = NIL then
        wm.currentP ← wm.tail
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

Data structure - level solutions XV

- Implementation of *getCurrentProgram*
  - This is easy, we just get the program from the currentP node.
  - Since *turnLeft* and *turnRight* are implemented in such a way that they never leave *currentP* equal to NIL, we can just access its content.

```
function getCurrentProgram(wm):
    getCurrentProgram ← [wm.curretP].info
end-function
```

- Complexity: $\Theta(1)$

Data structure - level solutions XVI

- Implementation of *start*
  - When we start the current program, its frequency needs to be incremented. This is easy to do, we have the *currentP* node. But then, we need to make sure that the nodes are still ordered by their frequencies. In our case this means that the *currentP* node might need to be moved towards the beginning of the list. We can do this in two ways:
    - Leave the structure as it is, just swap the frequency and info parts of the nodes.
    - Rearrange the nodes (change links), so that the current program node actually is moved in front.
  - Can current program become the head?
    - If the initial frequency is 100000, this is highly unlikely, since that frequency means that we can start the same program 27 times a day for 10 years and then it will have a frequency close to that of the *Eco 40-60* program.
    - If we increment the frequency of the program every time another program is started, this will never happen.

Data structure - level solutions XVII

```
subalgorithm start(wm):
    [wm.currentP].fr ← [wm.currentP].fr + 1
    if [wm.currentP] ≠ wm.head then
        [wm.head].fr ← [wm.head].fr + 1
        while [[wm.currentP].prev].fr < [wm.currentP].fr execute
            oldFr ← [[wm.currentP].prev].fr
            oldPr ← [[wm.currentP].prev].pr
            [[wm.currentP].prev].fr ← [wm.currentP].fr
            [[wm.currentP].prev].pr ← [wm.currentP].pr
            [wm.currentP].fr ← oldFr
            [wm.currentP].pr ← oldPr
            wm.currentP ← [wm.currentP].prev
        end-while
    end-if
    wm.currentP ← wm.head
end-subalgorithm
```

Data structure - level solutions XVIII

- Complexity: $O(number\ of\ programs)$ - we might have all but the first program with the same frequency and if we start the last one, it needs to be moved to be the second.