DATA STRUCTURES
LECTURE 8

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 7...

- Linked Lists on Array

Today

- Linked List on Array
- Stack, Queue, Priority Queue

DLLA - Node - Recap

- Linked Lists with dynamic allocation are made of nodes. We can define a structure to represent a node, even if we are working with arrays.

- A node (for a doubly linked list) contains the information and links towards the previous and the next nodes:

DLLANode:
    info: TElem
    next: Integer
    prev: Integer

DLLA - Recap

- Having defined the *DLLANode* structure, we only need one array, which will contain *DLLANodes*.
- Since it is a doubly linked list, we keep both the head and the tail of the list.

DLLA:
    nodes: DLLANode[]
    cap: Integer
    head: Integer
    tail: Integer
    firstEmpty: Integer
    size: Integer *//it is not mandatory, but useful*

DLLA - InsertPosition

```
subalgorithm insertPosition(dlla, elem, poz) is:
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted in dlla at position poz
   if poz < 1 OR poz > dlla.size + 1 execute
      @throw exception
   end-if
   newElem ← alocate(dlla)
   if newElem = -1 then
      @resize
      newElem ← alocate(dlla)
   end-if
   dlla.nodes[newElem].info ← elem
   if poz = 1 then
      if dlla.head = -1 then
         dlla.head ← newElem
         dlla.tail ← newElem
      else
//continued on the next slide...
```

DLLA - InsertPosition

```
        dlla.nodes[newElem].next ← dlla.head
        dlla.nodes[dlla.head].prev ← newElem
        dlla.head ← newElem
    end-if
  else
    nodC ← dlla.head
    pozC ← 1
    while nodC ≠ -1 and pozC < poz - 1 execute
        nodC ← dlla.nodes[nodC].next
        pozC ← pozC + 1
    end-while
    if nodC ≠ -1 then //it should never be -1, the position is correct
        nodNext ← dlla.nodes[nodC].next
        dlla.nodes[newElem].next ← nodNext
        dlla.nodes[newElem].prev ← nodC
        dlla.nodes[nodC].next ← newElem
//continued on the next slide...
```

```
            if nodNext = -1 then
                dlla.tail ← newElem
            else
                dlla.nodes[nodNext].prev ← newElem
            end-if
        end-if
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

DLLA - Iterator

- The iterator for a DLLA contains as *current element* the index of the current node from the array.

DLLAIterator:
  list: DLLA
  currentElement: Integer

DLLAIterator - init

subalgorithm init(it, dlla) **is:**
//pre: dlla is a DLLA
//post: it is a DLLAIterator for dlla
  it.list ← dlla
  it.currentElement ← dlla.head
end-subalgorithm

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).

- Complexity:

DLLAIterator - init

subalgorithm init(it, dlla) **is:**
//pre: dlla is a DLLA
//post: it is a DLLAIterator for dlla
  it.list $\leftarrow$ dlla
  it.currentElement $\leftarrow$ dlla.head
end-subalgorithm

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).
- Complexity: $\Theta(1)$

DLLAIterator - getCurrent

```
subalgorithm getCurrent(it) is:
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
//throws exception if the iterator is not valid
   if it.currentElement = -1 then
      @throw exception
   end-if
   getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm
```

- Complexity:

DLLAIterator - getCurrent

subalgorithm getCurrent(it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: e is a TElem, e is the current element from it
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  getCurrent ← it.list.nodes[it.currentElement].info
end-subalgorithm

- Complexity: $\Theta(1)$

DLLAIterator - next

subalgoritm next (it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm

- In case a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.
- Complexity:

DLLAIterator - next

subalgoritm next (it) **is:**
//pre: it is a DLLAIterator, it is valid
//post: the current elements from it is moved to the next element
//throws exception if the iterator is not valid
  if it.currentElement = -1 **then**
    @throw exception
  end-if
  it.currentElement ← it.list.nodes[it.currentElement].next
end-subalgorithm

- In case a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.
- Complexity: $\Theta(1)$

DLLAIterator - valid

function valid (it) **is:**
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false otherwise
  if it.currentElement = -1 **then**
    valid ← False
  else
    valid ← True
  end-if
end-function

- Complexity:

DLLAIterator - valid

function valid (it) **is:**
//pre: it is a DLLAIterator
//post: valid return true is the current element is valid, false otherwise
  if it.currentElement = -1 **then**
    valid ← False
  else
    valid ← True
  end-if
end-function

- Complexity: $\Theta(1)$

ADT Stack

- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed, the one from the top is automatically removed.
  - Only the element from the top can be accessed.

- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

Representation for Stack

- Data structures that can be used to implement a stack:
  - Arrays
    - Static Array - if we want a fixed-capacity stack
    - Dynamic Array
  - Linked Lists
    - Singly-Linked List
    - Doubly-Linked List

Array-based representation

- Where should we place the top of the stack for optimal performance?

Array-based representation

- Where should we place the top of the stack for optimal performance?
- We have two options:
  - Place top at the beginning of the array - every push and pop operation needs to shift every element to the right or left.
  - Place top at the end of the array - push and pop elements without moving the other ones.

Stack - Representation on SLL

- Where should we place the top of the stack for optimal performance?

- Where should we place the top of the stack for optimal performance?
- We have two options:
  - Place it at the end of the list (like we did when we used an array) - for every push, pop and top operation we have to iterate through every element to get to the end of the list.
  - Place it at the beginning of the list - we can push and pop elements without iterating through the list.

Stack - Representation on DLL

- Where should we place the top of the stack for optimal performance?

Stack - Representation on DLL

- Where should we place the top of the stack for optimal performance?

- We have two options:
  - Place it at the end of the list (like we did when we used an array) - we can push and pop elements without iterating through the list.
  - Place it at the beginning of the list - we can push and pop elements without iterating through the list.

Fixed capacity stack with linked list

- How could we implement a stack with a fixed maximum capacity using a linked list?

Fixed capacity stack with linked list

- How could we implement a stack with a fixed maximum capacity using a linked list?
- Similar to the implementation with a static array, we can keep in the *Stack* structure two integer values (besides the top node): maximum capacity and current size.

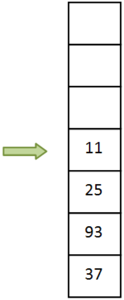
- How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

GetMinimum in constant time

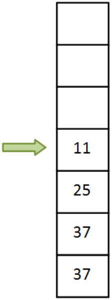
- How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time - Example

- If this is the *element stack*:



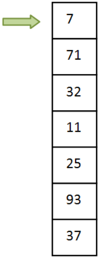
- This is the corresponding *min stack*:



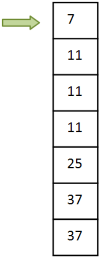
GetMinimum in constant time - Example

- When a new element is pushed to the *element stack*, we push a new element to the *min stack* as well. This element is the minimum between the top of the *min stack* and the newly added element.

- The *element stack*:



- The corresponding *min stack*:



GetMinimum in constant time

- When an element si popped from the *element stack*, we will pop an element from the *min stack* as well.
- The *getMinimum* operation will simply return the *top* of the *min stack*.
- The other stack operations remain unchanged (except *init*, where you have to create two stacks).

GetMinimum in constant time

- Let's implement the *push* operation for this *SpecialStack*, represented in the following way:

```
SpecialStack:
    elementStack: Stack
    minStack: Stack
```

- We will use an existing implementation for the stack and work only with the operations from the interface.

Push for SpecialStack

```
subalgorithm push(ss, e) is:
   if isFull(ss.elementStack) then
      @throw overflow (full stack) exception
   end-if
   if isEmpty(ss.elementStack) then//the stacks are empty, just push the elem
      push(ss.elementStack, e)
      push(ss.minStack, e)
   else
      push(ss.elementStack, e)
      currentMin ← top(ss.minStack)
      if currentMin < e then //find the minim to push to minStack
         push(ss.minStack, currentMin)
      else
         push(ss.minStack, e)
      end-if
   end-if
end-subalgorithm //Complexity: Θ(1)
```

- We designed the special stack in such a way that all the operations have a $\Theta(1)$ time complexity.
- The disadvantage is that we occupy twice as much space as with the regular stack.

- Think about how can we reduce the space occupied by the *min stack* to O(n) (especially if the minimum element of the stack rarely changes). *Hint: If the minimum does not change, we don't have to push a new element to the min stack.* How can we implement the *push* and *pop* operations in this case? What happens if the minimum element appears more than once in the *element stack*?

ADT Queue

- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
  - When a new element is added (pushed), it has to be added to the *rear* of the queue.
  - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

- What data structures can be used to implement a Queue?
    - Dynamic Array - circular array (already discussed)
    - Singly Linked List
    - Doubly Linked List

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

- We can improve the complexity of the operations if, even though the list is singly linked, we keep both the head and the tail of the list.
- What should the tail of the list be: the *front* or the *rear* of the queue?

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
    - Put *front* at the beginning of the list and *rear* at the end
    - Put *front* at the end of the list and *rear* at the beginning

- 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.

- What data structures can be used to implement a priority queue?
  - Dynamic Array
  - Linked List
  - (Binary) Heap - will be discussed later

Priority Queue - Representation

- If the representation is a Dynamic Array or a Linked List we have to decide how we store the elements in the array/list:
  - we can keep the elements ordered by their priorities
    - Where would you put the element with the highest priority?
  - we can keep the elements in the order in which they were inserted

Priority Queue - Representation

- Complexity of the main operations for the two representation options:

Operation	**Sorted**	**Non-sorted**
push	O(n)	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

- What happens if we keep in a separate field the element with the highest priority?