DATA STRUCTURES AND ALGORITHMS
LECTURE 6

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

In Lecture 5...

- Containers
  - Doubly Linked List
  - Sorted Lists
  - Circular Lists
  - XOR Lists
  - Skip Lists
- Linked Lists on Array

Today

1. Linked Lists on Arrays
2. Iterator
3. Stacks, Queue, Deque and Priority Queues

Linked Lists on Arrays

- Usually, when we work with arrays, we store the elements in the array starting from the leftmost position and place them one after the other (no empty spaces in the middle of the list are allowed).

- The order of the elements is given by the order in which they are placed in the array.

elems	46	78	11	6	59	19				

- Order of the elements: 46, 78, 11, 6, 59, 19

Linked Lists on Arrays

- We can define a linked data structure on an array, if we consider that the order of the elements is not given by their relative positions in the array, but by an integer number associated with each element, which shows the index of the next element in the array (thus we have a singly linked list).

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				

head = 3

- Order of the elements: 11, 46, 59, 78, 19, 6

Linked Lists on Arrays

- Now, if we want to delete the number 46 (which is actually the second element of the list), we do not have to move every other element to the left of the array, we just need to modify the links:

elems		78	11	6	59	19				
next		6	5	-1	2	4				

head = 3

- Order of the elements: 11, 59, 78, 19, 6

Linked Lists on Arrays

- If we want to insert a new element, for example 44, at the 3^{rd} position in the list, we can put the element anywhere in the array, the important part is setting the links correctly:

elems		78	11	6	59	19		44		
next		6	5	-1	8	4		2		

head = 3

- Order of the elements: 11, 59, 44, 78, 19, 6

Linked Lists on Arrays

- When a new element needs to be inserted, it can be put to any empty position in the array. However, finding an empty position has $O(n)$ complexity, which will make the complexity of any insert operation (anywhere in the list) $O(n)$. In order to avoid this, we will keep a linked list of the empty positions as well.

elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

firstEmpty = 1

Linked Lists on Arrays

- In a more formal way, we can simulate a singly linked list on an array with the following:
    - an array in which we will store the elements.
    - an array in which we will store the links (indexes to the next elements).
    - the capacity of the arrays (the two arrays have the same capacity, so we need only one value).
    - an index to tell where the *head* of the list is.
    - an index to tell where the first empty position in the array is.

SLL on Array - Representation

- The representation of a singly linked list on an array is the following:

SLLA:
  elems: TElem[]
  next: Integer[]
  cap: Integer
  head: Integer
  firstEmpty: Integer

SLLA - Operations

- We can implement for a SLLA any operation that we can implement for a SLL:
  - insert at the beginning, end, at a position, before/after a given value
  - delete from the beginning, end, from a position, a given element
  - search for an element
  - get an element from a position

SLLA - Init

subalgorithm init(slla) **is:**
//pre: true; post: slla is an empty SLLA
  slla.cap ← INIT_CAPACITY

SLLA - Init

subalgorithm init(slla) **is:**
//pre: true; post: slla is an empty SLLA
    slla.cap ← INIT_CAPACITY
    slla.elems ← @an array with slla.cap positions
    slla.next ← @an array with slla.cap positions
    slla.head ← -1
    for i ← 1, slla.cap-1 **execute**
        slla.next[i] ← i + 1
    end-for
    slla.next[slla.cap] ← -1
    slla.firstEmpty ← 1
end-subalgorithm

- Complexity:

SLLA - Init

subalgorithm init(slla) **is:**
//pre: true; post: slla is an empty SLLA
  slla.cap ← INIT_CAPACITY
  slla.elems ← @an array with slla.cap positions
  slla.next ← @an array with slla.cap positions
  slla.head ← -1
  for i ← 1, slla.cap-1 **execute**
    slla.next[i] ← i + 1
  end-for
  slla.next[slla.cap] ← -1
  slla.firstEmpty ← 1
end-subalgorithm

- Complexity: $\Theta(n)$ - where n is the initial capacity

SLLA - Search

function search (slla, elem) **is:**
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
    current ← slla.head
    while current ≠ -1 **and** slla.elems[current] ≠ elem **execute**
        current ← slla.next[current]
    end-while
    if current ≠ -1 **then**
        search ← True
    else
        search ← False
    end-if
end-function

- Complexity:

SLLA - Search

```
function search (slla, elem) is:
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
   current ← slla.head
   while current ≠ -1 and slla.elems[current] ≠ elem execute
      current ← slla.next[current]
   end-while
   if current ≠ -1 then
      search ← True
   else
      search ← False
   end-if
end-function
```

- Complexity: $O(n)$

SLLA - Search

- From the *search* function we can see how we can go through the elements of a SLLA (and how similar this traversal is to the one done for a SLL):
  - We need a *current* element used for traversal, which is initialized to the index of the *head* of the list.
  - We stop the traversal when the value of *current* becomes -1
  - We go to the next element with the instruction: $current \leftarrow slla.next[current]$.

SLLA -InsertPosition

subalgorithm insertPosition(slla, elem, poz) **is:**
//pre: slla is an SLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted into slla at position pos
//throws an exception if the position is not valid
  if (pos $<$ 1) **then**
    @error, invalid position
  end-if

SLLA -InsertPosition

subalgorithm insertPosition(slla, elem, poz) **is:**
//pre: slla is an SLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted into slla at position pos
//throws an exception if the position is not valid
  if (pos < 1) **then**
    @error, invalid position
  end-if
  if slla.firstEmpty = -1 **then**
    newElems ← @an array with slla.cap * 2 positions
    newNext ← @an array with slla.cap * 2 positions
    for i ← 1, slla.cap **execute**
      newElems[i] ← slla.elems[i]
      newNext[i] ← slla.next[i]
    end-for
//continued on the next slide...

```
    for i ← slla.cap + 1, slla.cap*2 - 1 execute
        newNext[i] ← i + 1
    end-for
    newNext[slla.cap*2] ← -1
    //free slla.elems and slla.next if necessary
    slla.elems ← newElems
    slla.next ← newNext
    slla.firstEmpty ← slla.cap+1
    slla.cap ← slla.cap * 2
end-if
```

```
    for i ← slla.cap + 1, slla.cap*2 - 1 execute
      newNext[i] ← i + 1
    end-for
    newNext[slla.cap*2] ← -1
    //free slla.elems and slla.next if necessary
    slla.elems ← newElems
    slla.next ← newNext
    slla.firstEmpty ← slla.cap+1
    slla.cap ← slla.cap * 2
end-if
if poz = 1 then
    newPosition ← slla.firstEmpty
    slla.elems[newPosition] ← elem
    slla.firstEmpty ← slla.next[slla.firstEmpty]
    slla.next[newPosition] ← slla.head
    slla.head ← newPosition
else
```

pozCurrent ← 1
nodCurrent ← slla.head
while nodCurrent $\neq$ -1 **and** pozCurrent $<$ poz - 1 **execute**
  pozCurrent ← pozCurrent + 1
  nodCurrent ← slla.next[nodCurrent]
end-while
if nodCurrent $\neq$ -1 **atunci**
  newElem ← slla.firstEmpty
  slla.firstEmpty ← slla.next[firstEmpty]
  slla.elems[newElem] ← elem
  slla.next[newElem] ← slla.next[nodCurrent]
  slla.next[nodCurrent] ← newElem
else
//continued on the next slide...

SLLA - InsertPosition

```
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity:

SLLA - InsertPosition

```
      @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$

SLLA - InsertPosition

- Observations regarding the *insertPosition* subalgorithm
  - Similar to the SLL, we iterate through the list until we find the element *after* which we insert (denoted in the code by *nodCurrent* - which is an index in the array).
  - We treat as a special case the situation when we insert at the first position (no node to insert after).

SLLA - DeleteElement

subalgorithm deleteElement(slla, elem) **is:**
//pre: slla is a SLLA; elem is a TElem
//post: the element elem is deleted from SLLA
  nodC ← slla.head
  prevNode ← -1
  while nodC ≠ -1 **and** slla.elems[nodC] ≠ elem **execute**
    prevNode ← nodC
    nodC ← slla.next[nodC]
  end-while
  if nodC ≠ -1 **then**
    if nodC = slla.head **then**
      slla.head ← slla.next[slla.head]
    else
      slla.next[prevNode] ← slla.next[nodC]
    end-if
//continued on the next slide...

SLLA - DeleteElement

//add the nodC position to the list of empty spaces
    slla.next[nodC] ← slla.firstEmpty
    slla.firstEmpty ← nodC
  else
    @the element does not exist
  end-if
end-subalgorithm

- Complexity: $O(n)$

SLLA - Iterator

- Iterator for a SSLA is a combination of an iterator for an array and of an iterator for a singly linked list:
- Since the elements are stored in an array, the *currentElement* will be an index from the array.
- But since we have a linked list, going to the next element will not be done by incrementing the *currentElement* by one; we have to follow the *next* links.

DLLA

- Obviously, we can define a doubly linked list as well without pointers, using arrays.
- For the DLLA we will see another way of representing a linked list on arrays:
  - The main idea is the same, we will use array indexes as links between elements
  - We are using the same information, but we are going to structure it differently
  - However, we can make it look more similar to linked lists with dynamic allocation

DLLA - Node

- 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:

<u>DLLANode:</u>
  info: TElem
  next: Integer
  prev: Integer

DLLA

- 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

DLLA - Allocate and free

- To make the representation and implementation even more similar to a dynamically allocated linked list, we can define the *allocate* and *free* functions as well.

```
function allocate(dlla) is:
//pre: dlla is a DLLA
//post: a new element will be allocated and its position returned
  newElem ← dlla.firstEmpty
  if newElem ≠ -1 then
    dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next
    if dlla.firstEmpty ≠ -1 then
      dlla.nodes[dlla.firstEmpty].prev ← -1
    end-if
    dlla.nodes[newElem].next ← -1
    dlla.nodes[newElem].prev ← -1
  end-if
  allocate ← newElem
end-function
```

DLLA - Allocate and free

subalgorithm free (dlla, poz) **is:**
//pre: dlla is a DLLA, poz is an integer number
//post: the position poz was freed
  dlla.nodes[poz].next ← dlla.firstEmpty
  dlla.nodes[poz].prev ← -1
  if dlla.fistEmpty ≠ -1 **then**
    dlla.nodes[dlla.firstEmpty].prev ← poz
  end-if
  dlla.firstEmpty ← poz
end-subalgorithm

DLLA - InsertPosition

subalgorithm insertPosition(dlla, elem, poz) **is:**
//pre: dlla is a DLLA, elem is a TElem, poz is an integer number
//we assume that poz is a valid position
//post: the element elem is inserted in dlla at position poz
    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
            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...
```

DLLA - InsertPosition

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 ← 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: $\Theta(1)$

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: $\Theta(1)$

Iterator - why do we need it? I

- Most containers have iterators and for every data structure we will discuss how we can implement an iterator for a container defined on that data structure.

- Why are iterators so important?

Iterator - why do we need it? II

- They offer a uniform way of iterating through the elements of any container

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
   iterator(c, it)
   while valid(it) execute
      //get the current element from the iterator
      elem ← getCurrent(it)
      print elem
      //go to the next element
      next(it)
   end-while
end-subalgorithm
```

Iterator - why do we need it? III

- For most containers the iterator is the only thing we have to *see* the content of the container.
    - ADT List is the only container that has positions, for other containers we can use only the iterator.

Iterator - why do we need it? IV

- Giving up positions, we can gain performance.
  - Containers that do not have positions can be represented on data structures where some operations have good complexities, but where the notion of a position does not naturally exist and where enforcing positions is really complicated (ex. hash tables).

Iterator - why do we need it? V

- Even if we have positions, using an iterator might be faster.
    - Going through the elements of a linked list with an iterator is faster than going through every position one-by-one.

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?

Stack - Representation on SLL

- 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.

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)?

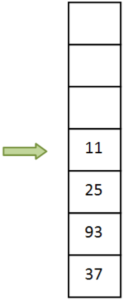
GetMinimum in constant time

- 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)?

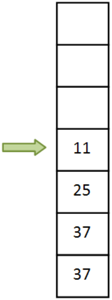
- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time - Example

- If this is the *element stack*:



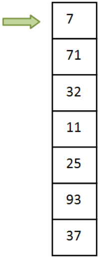
- This is the corresponding *min stack*:



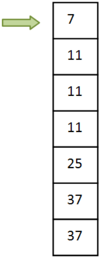
GetMinimum in constant time - Example

- 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:* $\Theta(1)$

SpecialStack - Notes / Think about it

- 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.

Queue - Representation

- 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.

Queue - representation on a SLL

- 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?

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?

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

ADT Deque

- The ADT Deque (Double Ended Queue) is a container in which we can insert and delete from both ends:
  - We have *push_front* and *push_back*
  - We have *pop_front* and *pop_back*
  - We have *top_front* and *top_back*
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

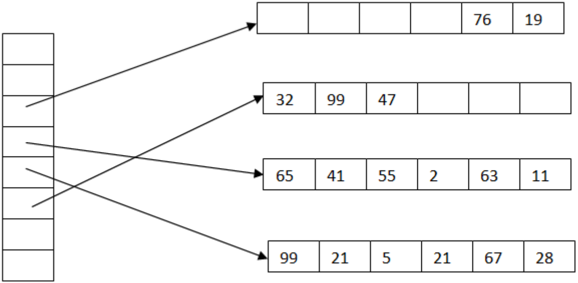
ADT Deque

- Possible (good) representations for a Deque:
  - Circular Array
  - Doubly Linked List
  - A dynamic array of constant size arrays

ADT Deque - Representation

- An interesting representation for a deque is to use a dynamic array of fixed size arrays:
  - Place the elements in fixed size arrays (blocks).
  - Keep a dynamic array with the addresses of these blocks.
  - Every block is full, except for the first and last ones.
  - The first block is filled from right to left.
  - The last block is filled from left to right.
  - If the first or last block is full, a new one is created and its address is put in the dynamic array.
  - If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved).

Deque - Example



- Elements of the deque: 76, 19, 65, ..., 11, 99, ..., 28, 32, 99, 47

Deque - Example

- Information (fields) we need to represent a deque using a dynamic array of blocks:
  - Block size
  - The dynamic array with the addresses of the blocks
  - Capacity of the dynamic array
  - First occupied position in the dynamic array
  - First occupied position in the first block
  - Last occupied position in the dynamic array
  - Last occupied position in the last block

  - The last two fields are not mandatory if we keep count of the total number of elements in the deque.