DATA STRUCTURES AND ALGORITHMS
LECTURE 6

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 5...

- Skip Lists
- ADT Set, Map, Matrix
- Heap

Today

1. Heap
2. ADT List
3. ADT Stack

Heap

- A binary heap is a data structure that can be used as an efficient representation for Priority Queues (will be discussed later).
- A binary heap is a kind of hybrid between a dynamic array and a binary tree.
- The elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree.

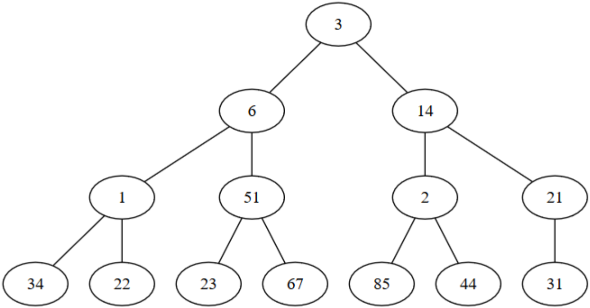
Heap

- Assume that we have the following array (upper row contains positions, lower row contains elements):

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31

Heap

- We can visualize this array as a binary tree, in which each node has exactly 2 children, except for the last two rows, but there the children of the nodes are completed from left to right.



Heap

- If the elements of the array are: $a_1, a_2, a_3, ..., a_n$, we know that:
    - a_1 is the root of the heap
    - for an element from position i, its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than n)
    - for an element from positions i (i $>$ 1), the parent of the element is on position $[i/2]$ (integer part of i/2)

Heap

- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.

  - *Heap structure:* in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.

  - *Heap property:* $a_i \geq a_{2*i}$ (if $2 * i \leq n$) and $a_i \geq a_{2*i+1}$ (if $2 * i + 1 \leq n$)
  - The $\geq$ relation between a node and both its descendants can be generalized (other relations can be used as well).

Heap - Notes

- If we use the $\geq$ relation, we will have a *MAX-HEAP*.
- If we use the $\leq$ relation, we will have a *MIN-HEAP*.
- The height of a heap with n elements is $log_2 n$, so the operations performed on the heap have $O(log_2 n)$ complexity.

Heap - operations

- A heap can be used as representation for a Priority Queue and it has two specific operations:
  - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
  - remove (we always remove the root of the heap - no other element can be removed).

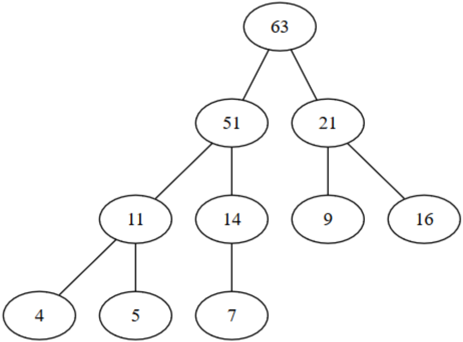
Heap - representation

Heap:
  cap: Integer
  len: Integer
  elems: TElem[]

- For the implementation we will assume that we have a MAX-HEAP.

Heap - Add - example

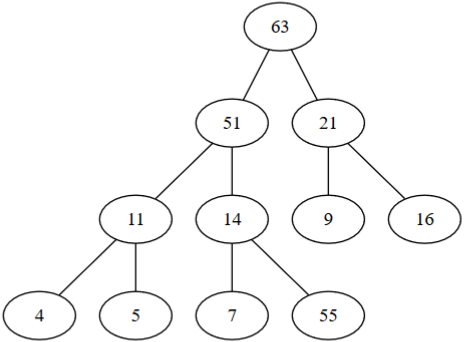
- Consider the following (MAX) heap:



- Let's add the number 55 to the heap.

Heap - Add - example

- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).

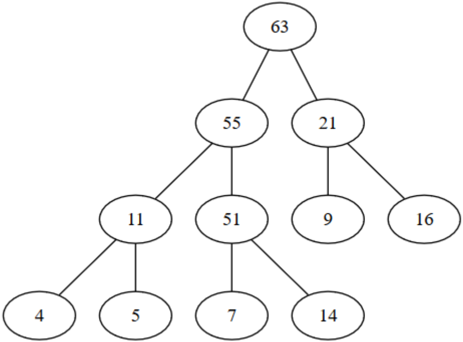


Heap - Add - example

- *Heap property* is not kept: 14 has as child node 55 (since it is a MAX-heap, each node has to be greater or equal than its descendants).
- In order to restore the heap property, we will start a *bubble-up* process: we will keep swapping the value of the new node with the value of its parent node, until it gets to its final place. No other node from the heap is changed.

Heap - Add - example

- When *bubble-up* ends:



Heap - add

subalgorithm add(heap, e) **is:**
//heap - a heap
//e - the element to be added
  if heap.len = heap.cap **then**
    @ resize
  end-if
  heap.elems[heap.len+1] ← e
  heap.len ← heap.len + 1
  bubble-up(heap, heap.len)
end-subalgorithm

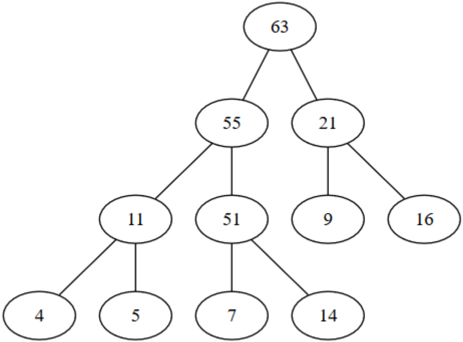
Heap - add

subalgorithm bubble-up (heap, p) **is:**
//heap - a heap
//p - position from which we bubble the new node up
    poz ← p
    elem ← heap.elems[p]
    parent ← p / 2
    while poz > 1 **and** elem > heap.elems[parent] **execute**
        //move parent down
        heap.elems[poz] ← heap.elems[parent]
        poz ← parent
        parent ← poz / 2
    end-while
    heap.elems[poz] ← elem
end-subalgorithm

- Complexity: $O(log_2 n)$

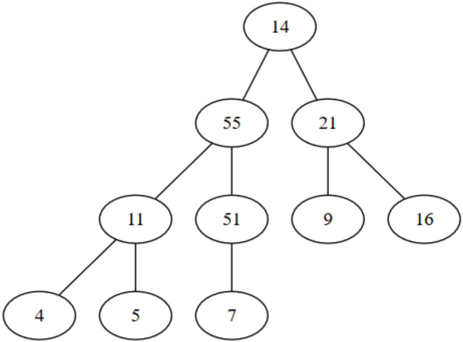
Heap - Remove - example

- From a heap we can only remove the root element.



Heap - Remove - example

- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.

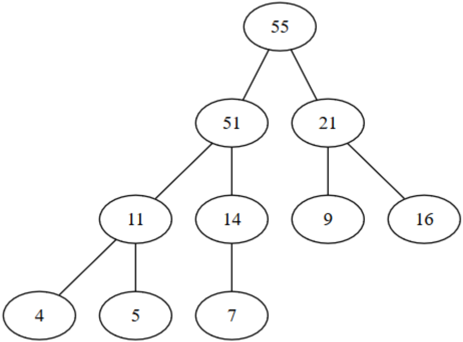


Heap - Remove - example

- *Heap property* is not kept: the root is no longer the maximum element.
- In order to restore the heap property, we will start a *bubble-down* process, where the new node will be swapped with its maximum child, until it becomes a leaf, or until it will be greater than both children.

Heap - Remove - example

- When the bubble-down process ends:



Heap - remove

function remove(heap) **is:**
//heap - is a heap
  if heap.len = 0 **then**
    @ error - empty heap
  end-if
  deletedElem ← heap.elems[1]
  heap.elems[1] ← heap.elems[heap.len]
  heap.len ← heap.len - 1
  bubble-down(heap, 1)
  remove ← deletedElem
end-function

Heap - remove

subalgorithm bubble-down(heap, p) **is:**
//heap - is a heap
//p - position from which we move down the element
  poz ← p
  elem ← heap.elems[p]
  while poz < heap.len **execute**
    maxChild ← -1
    if poz * 2 ≤ heap.len **then**
    //it has a left child
      maxChild ← poz*2
    end-if
    if poz*2+1 ≤ heap.len **and** heap.elems[2*poz+1]>heap.elems[2*poz] **th**
    //it has two children and right is greater
      maxChild ← poz*2 + 1
    end-if
//continued on the next slide...

Heap - remove

```
        if maxChild ≠ -1 and heap.elems[maxChild] > elem then
            heap.elems[poz] ← heap.elems[maxChild]
            poz ← maxChild
        else
            poz ← heap.len + 1
            //to stop the while loop
        end-if
    end-while
end-subalgorithm
```

- Complexity: $O(log_2 n)$

Question

- In a max-heap where can we find the:
  - maximum element of the array?

Question

- In a max-heap where can we find the:
  - maximum element of the array?
  - minimum element of the array?

Heap-sort

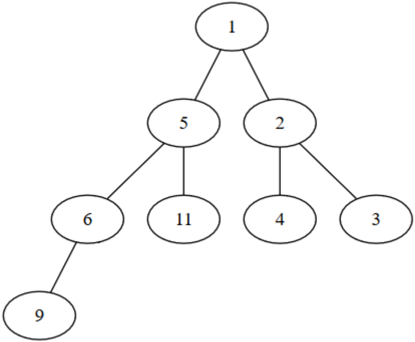
- There is a sorting algorithm, called *Heap-sort*, that is based on the use of a heap.
- In the following we are going to assume that we want to sort a sequence in ascending order.
- Let's sort the following sequence: [6, 1, 3, 9, 11, 4, 2, 5]

Heap-sort - Naive approach

- Based on what we know so far, we can guess how heap-sort works:
  - Build a min-heap adding elements one-by-one to it.
  - Start removing elements from the min-heap: they will be removed in the sorted order.

Heap-sort - Naive approach

- The heap when all the elements were added:



- When we remove the elements one-by-one we will have: 1, 2, 3, 4, 5, 6, 9, 11.

Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?

Heap-sort - Naive approach

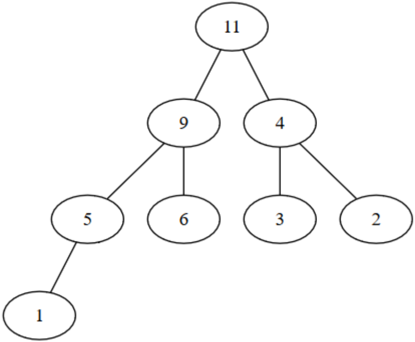
- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2 n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?

Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?
- The extra space complexity of the algorithm is $\Theta(n)$ - we need an extra array.

Heap-sort - Better approach

- If instead of building a min-heap, we build a max-heap (even if we want to do ascending sorting), we do not need the extra array.



Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.

Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.
  - If we have an unsorted array, we can transform it easier into a heap: the second half of the array will contain leaves, they can be left where they are.
  - Starting from the first non-leaf element (and going towards the beginning of the array), we will just call *bubble-down* for every element.
  - Time complexity of this approach: $O(n)$ (but removing the elements from the heap is still $O(nlog_2n)$)

ADT List

- A *list* can be seen as a sequence of elements of the same type, $< l_1, l_2, ..., l_n >$, where there is an order of the elements, and each element has a *position* inside the list.
- In a list, the order of the elements is important (positions are important).
- The number of elements from a list is called the length of the list. A list without elements is called *empty*.

ADT List

- A List is a container which is either *empty* or
  - it has a unique *first* element
  - it has a unique *last* element
  - for every element (except for the last) there is a unique *successor* element
  - for every element (except for the first) there is a unique *predecessor* element
- In a list, we can insert elements (using positions), remove elements (using positions), we can access the successor and predecessor of an element from a given position, we can access an element from a position.

ADT List - Positions

- Every element from a list has a unique position in the list:
  - positions are relative to the list (but important for the list)
  - the position of an element:
    - identifies the element from the list
    - determines the position of the successor and predecessor element (if they exist).

ADT List - Positions

- Position of an element can be seen in different ways:
  - as the *rank* of the element in the list (first, second, third, etc.)
    - similarly to an array, the position of an element is actually its index
  - as a *reference* to the memory location where the element is stored.
    - for example a pointer to the memory location
- For a general treatment, we will consider in the following the *position* of an element in an abstract manner, and we will consider that positions are of type *TPosition*

ADT - List - Positions

- A position p will be considered *valid* if it denotes the position of an actual element from the list:
    - if p is a pointer to a memory location, p is valid if it is the address of an element from a list (not NIL or some other address that is not the address of any element)
    - if p is the rank of the element from the list, p is valid if it is between 1 and the number of elements.
- For an invalid position we will use the following notation: $\perp$

ADT List I

- Domain of the ADT List:

  $\mathcal{L} = \{l|$l is a list with elements of type TElem, each having a unique position in l of type TPosition$\}$

ADT List II

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

ADT List III

- first(l)
  - **descr:** returns the TPosition of the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the first element from l} & \text{if l} \neq \emptyset \\ \perp & otherwise \end{cases}$$

ADT List IV

- last(l)
  - **descr:** returns the TPosition of the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last \leftarrow p \in TPosition$

    $$p = \begin{cases} \text{the position of the last element from l} & \text{if l} \neq \emptyset \\ \perp & \textit{otherwise} \end{cases}$$

ADT List V

- valid(l, p)
  - **descr:** checks whether a TPosition is valid in a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$
  - **post:** $valid \leftarrow \begin{cases} true & \text{if p is a valid position in l} \\ false & otherwise \end{cases}$

ADT List VI

- next(l, p)
  - **descr:** goes to the next TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$
  - **post:**

$$next \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the next element after p} & \text{if p is not the last position} \\ \perp & otherwise \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VII

- previous(l, p)
  - **descr:** goes to the previous TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$
  - **post:**

$$previous \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the element before p} & \text{if p is not the first position} \\ \perp & \textit{otherwise} \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VIII

- getElement(l, p, e)
  - **descr:** returns the element from a given TPosition
  - **pre:** $l \in \mathcal{L}, p \in TPosition$, valid(p)
  - **post:** $e \in TElem$, e = the element from position p from l
  - **throws:** exception if p is not valid

ADT List IX

- position(l, e)
  - **descr:** returns the TPosition of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow p \in TPosition$$

$$p = \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ \perp & otherwise \end{cases}$$

ADT List X

- modify(l, p, e)
  - **descr:** replaces an element from a TPosition with another
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, valid(p)
  - **post:** $l' \in \mathcal{L}$, the element from position p from l' is e
  - **throws:** exception if p is not valid

ADT List XI

- insertFirst(l, e)
  - **descr:** inserts a new element at the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, the element e was added at the beginning of l

ADT List XII

- insertLast(l, e)
    - **descr:** inserts a new element at the end of a list
    - **pre:** $l \in \mathcal{L}, e \in TElem$
    - **post:** $l' \in \mathcal{L}$, the element e was added at the end of l

ADT List XIII

- insertAfter(l, p, e)
  - **descr:** inserts a new element after a given position
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, valid(p)
  - **post:**$l' \in \mathcal{L}$, the element e was added in l after the position p (position(l', e) $=$ next(l', p) - if e is not already in the list)
  - **throws:** exception if p is not valid

ADT List XIV

- insertBefore(l, p, e)
  - **descr:** inserts a new element before a given position
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, valid(p)
  - **post:** $l' \in \mathcal{L}$, the element e was added in l before the position p (position(l', e) = previous(l', p) - if e is not already in the list)
  - **throws:** exception if p is not valid

ADT List XV

- remove(l, p, e)
  - **descr:** removes an element froma given position from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$, valid(p)
  - **post:** $e \in TElem$, e is the element from position p from l, $l' \in \mathcal{L}$, l' = l - e.
  - **throws:** exception if p is not valid

ADT List XVI

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT List XVII

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT List XVIII

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT List XIX

- destroy(l)
    - **descr:** destroys a list
    - **pre:** $l \in \mathcal{L}$
    - **post:** l was destroyed

ADT List XX

- iterator(l, it)
  - **descr:** returns an iterator for a list
  - **pre:** $l \in \mathcal{L}$
  - **post:**$it \in \mathcal{I}$, it is an iterator over l

TPosition

- Using TPositions in the interface of the ADT List can have disadvantages:
  - The exact type of a TPosition might differ if we use different representations for the list.
  - We have a large interface with many operations.

TPosition - C++

- In STL, TPosition is represented by an iterator.
- The operations *valid*, *next*, *previous*, *getElement* are actually operations for the iterator.
- For example - vector:

  iterator insert(iterator position, const value_type& val)
  iterator erase (iterator position);

- For example - list:

  iterator insert(iterator position, const value_type& val)
  iterator erase (iterator position);

TPosition - Java

- In Java, TPosition is represented by an index.
- We can add and remove using index and we can access elements using their index.
- There are fewer operations in the interface of the list
- For example:

  void add(int index, E element)
  E get(int index)
  E remove(int index)

ADT SortedList

- We can define the ADT SortedList, in which the elements are memorized in a given order, based on a relation.
- Elements still have positions, we can access elements by position.
- Differences in the interface:
  - init takes as parameter a relation
  - only one insert operation exists
  - no modify operation

ADT List - representation

- If we want to implement the ADT List (or ADT SortedList) we can use the following data structures are representation:
  - a (dynamic) array - elements are kept in a contiguous memory location - we have direct access to any element
  - a linked list - elements are kept in nodes, we do not have direct access to any element
- Demo

ADT Stack

Stack of books

Source: www.clipartfest.com

- The word stack might be familiar from expressions like: *stack of books*, *stack of paper* or from the *call stack* that you usually see in debug windows.

Stack II

- 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 at the top.
  - When an element is removed it will be removed automatically from the top.
  - Only the element from the top can be accessed.

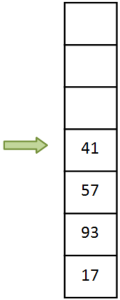
- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

Stack III

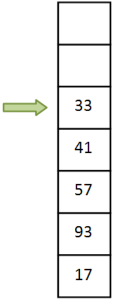
- When a new stack is created, it can have a fixed capacity. If the number of elements in the stack is equal to this capacity, we say that the *stack is full*.
- A stack with no elements is called an *empty stack*.

Stack Example

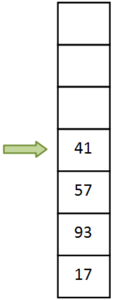
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

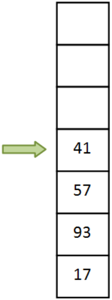


- We *pop* an element:

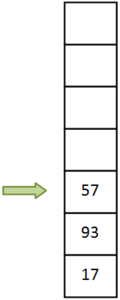


Stack Example II

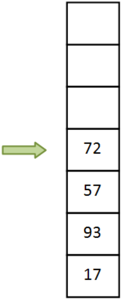
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



Stack Interface I

- The domain of the ADT Stack:
  $\mathcal{S} = \{s | s \textit{ is a stack with elements of type TElem}\}$
- The interface of the ADT Stack contains the following operations:

Stack Interface II

- init(s)
  - **Description:** creates a new empty stack
  - **Pre:** True
  - **Post:** $s \in \mathcal{S}$, s is an empty stack

Stack Interface III

- destroy(s)
    - **Description:** destroys a stack
    - **Pre:** $s \in \mathcal{S}$
    - **Post:** s was destroyed

Stack Interface IV

- push(s, e)
  - **Description:** pushes (adds) a new element onto the stack
  - **Pre:** $s \in \mathcal{S}$, e is a *TElem*
  - **Post:** $s' \in \mathcal{S}$, $s' = s \oplus e$, e is the most recent element added to the stack
  - **Throws:** an *overflow* error if the stack is full

Stack Interface V

- pop(s)
  - **Description:** pops (removes) the most recent element from the stack
  - **Pre:** $s \in \mathcal{S}$
  - **Post:** $pop \leftarrow e$, e is a $TElem$, e is the most recent element from s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **Throws:** an *underflow* error if the stack is empty

Stack Interface VI

- top(s)
  - **Description:** returns the most recent element from the stack (but it does not change the stack)
  - **Pre:** $s \in \mathcal{S}$
  - **Post:** $top \leftarrow e$, e is a $TElem$, e is the most recent element from s
  - **Throws:** an *underflow* error if the stack is empty

Stack Interface VII

- isEmpty(s)
  - **Description:** checks if the stack is empty (has no elements)
  - **Pre:** $s \in \mathcal{S}$
  - **Post:**

$$isEmpty \leftarrow \begin{cases} true, & if\ \ s\ has\ no\ elements \\ false, & otherwise \end{cases}$$

Stack Interface VIII

- isFull(s)
  - **Description:** checks if the stack is full - not every representation has this operation
  - **Pre:** $s \in \mathcal{S}$
  - **Post:**

$$isFull \leftarrow \begin{cases} true, & if \ s \ is \ full \\ false, & otherwise \end{cases}$$

Stack Interface IX

- **Note:** stacks cannot be iterated, so they don't have an *iterator* operation!

Representation for Stack

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

Static Array-based representation II

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

Static Array-based representation II

? 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.

Static Array-based representation

Stack:
- capacity: Integer
- top: Integer
- elements: TElem[0...capacity-1]

Init - Implementation using a static array

subalgorithm init(s) **is:**
  s.capacity ← MAX_CAPACITY
  //MAX_CAPACITY is a constant with the maximum capacity
  s.top ← 0
  @allocate memory for the *elements* array
end-subalgorithm

- Complexity: $\Theta(1)$

Push - Implementation using a static array

subalgorithm push(s, e) **is:**
  if s.capacity $=$ s.top **then** *//check if s is full*
    @throw overflow (full stack) exception
  end-if
  s.elements[s.top] $\leftarrow$ e
  s.top $\leftarrow$ s.top $+ 1$
end-subalgorithm

- Complexity: $\Theta(1)$

Pop - Implementation using a static array

function pop(s) **is:**
  if s.top $= 0$ **then** *//check if s is empty*
    @throw underflow(empty stack) exception
  end-if
  topElem $\leftarrow$ s.elements[s.top-1]
  s.top $\leftarrow$ s.top -1
  pop$\leftarrow$ topElem
end-function

- Complexity: $\Theta(1)$

Top - Implementation using a static array

function top(s) **is:**
    if s.top = 0 **then** *//check if s is empty*
        @throw underflow(empty stack) exception
    end-if
    topElem ← s.elements[s.top-1]
    top← topElem
end-function

- Complexity: $\Theta(1)$

IsEmpty - Implementation using a static array

function isEmpty(s) **is:**
  if s.top = 0 **then**
    isEmpty ← True
  else
    isEmpty ← False
  end-if
end-function

- Complexity: $\Theta(1)$

IsFull - Implementation using a static array

function isFull(s) **is:**
  if s.top = s.capacity **then**
    isFull ← True
  else
    isFull ← False
  end-if
end-function

- Complexity: $\Theta(1)$

Implementation using a dynamic array

? Which operations change if we use a dynamic array instead of a static one for implementing a Stack?

Implementation using a dynamic array

? Which operations change if we use a dynamic array instead of a static one for implementing a Stack?

- If we use a Dynamic Array we can change the capacity of the Stack as elements are pushed onto it, so the Stack will never be full (except when there is no memory at all).
- The *push* operation does not throw an exception, it resizes the array if needed (doubles the capacity).
- The *isFull* operation will always return false.

Singly-Linked List-based representation

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

Singly-Linked List-based representation

?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.

Singly-Linked List-based representation

Node:
elem: TElem
next: ↑ Node

Stack
top: ↑ Node

Init - Implementation using a singly-linked list

subalgorithm init(s) **is:**
  s.top $\leftarrow$ NIL
end-subalgorithm

- Complexity: $\Theta(1)$

Destroy - Implementation using a singly-linked list

subalgorithm destroy(s) **is:**
  while s.top $\neq$ NIL **execute**
    firstNode $\leftarrow$ s.top
    s.top $\leftarrow$ [s.top].next
    @deallocate firstNode
  end-while
end-subalgorithm

- Complexity: $\Theta(n)$ - where n is the number of elements from s

Push - Implementation using a singly-linked list

subalgorithm push(s, e) **is:**
    //allocate a new Node and set its fields
    @allocate *newnode* of type Node
    [newnode].elem ← e
    [newnode].next ← NIL
    if s.top = NIL **then**
        s.top ← newnode
    else
        [newnode].next ← s.top
        s.top ← newnode
    end-if
end-subalgorithm

- Complexity: $\Theta(1)$

Pop - Implementation using a singly-linked list

function pop(s) **is:**
  if s.top = NIL **then** *//check if s is empty*
    @throw underflow(empty stack) exception
  end-if
  firstNode ←s.top
  topElem ← [firstNode].elem
  s.top ← [s.top].next
  @deallocate firstNode
  pop← topElem
end-function

- Complexity: $\Theta(1)$

Top - Implementation using a singly-linked list

function top(s) **is:**
  if s.top = NIL **then** *//check if s is empty*
    @throw underflow(empty stack) exception
  end-if
  topElem ← [s.top].elem
  top← topElem
end-function

- Complexity: $\Theta(1)$

IsEmpty - Implementation using a singly-linked list

function isEmpty(s) **is:**
  if s.top = NIL **then**
    isEmpty ← True
  else
    isEmpty ← False
  end-if
end-function

- Complexity: $\Theta(1)$

IsFull - Implementation using a singly-linked list

- We don't have a maximum capacity in case of a linked list, so our stack will never be full. If we still want to implement this method, we can make it to always return false.

function isFull(s) **is:**
  isFull $\leftarrow$ False
end-function

- Complexity: $\Theta(1)$

Fixed capacity stack with singly-linked list

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

Fixed capacity stack with singly-linked list

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

- Similar to the implementation with a static array, we can keep in the *Stack* structure two integer values: maximum capacity and current size.