DATA STRUCTURES AND ALGORITHMS

LECTURE 8

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 7...

- Linked Lists on Arrays
- ADT Set
- ADT Map
- Iterator

Today

1. ADT Matrix
2. ADT List
3. ADT Stack
4. Written test

ADT Matrix

- A *Matrix* is a container that represents a two-dimensional array.
- Each element has a unique position, determined by two indexes: its line and column.

ADT Matrix

- A *Matrix* is a container that represents a two-dimensional array.
- Each element has a unique position, determined by two indexes: its line and column.
- The operations for a Matrix are different from the operations that exist for most other containers, because in a Matrix we cannot add elements, and we cannot delete an element from a Matrix, we can only change the value of an element.

Matrix - Operations

- The minimum set of operations that should exist for the ADT Matrix is:
  - init(matrix, nrL, nrC) - create a new matrix with *nrL* lines and *nrC* columns
  - nrLine(matrix) - return the number of lines from the matrix
  - nrColumns(matrix) - return the number of columns from the matrix
  - element(matrix, i, j) - return the element from the line *i* and column *j*
  - modify(matrix, i, j, val) - change the values of the element from line *i* and column *j* into *val*

Matrix - Operations

- Other possible operations:
  - get the position of a given element
  - create an iterator that goes through the elements by columns
  - create an iterator the goes through the elements by lines
  - etc.

Matrix - representation

- Usually a sequential representation is used for a Matrix (we memorize all the lines one after the other in a consecutive memory block).
- If the Matrix contains many values of 0 (or 0_{TElem}), we have a *sparse matrix*, where it is more (space) efficient to memorize only the elements that are different from 0.

Sparse Matrix example

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 5 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 \\ 1 & 0 & 0 & 7 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 5 \\ 0 & 0 & 9 & 1 & 0 & 0 \end{bmatrix}$$

- Out of the 36 elements, only 10 are different from 0.

Sparse Matrix - representation

- We can memorize (line, column, value) triples, where value is different from 0 (or 0_{TElem}). For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column).
- Triples can be stored in:
  - (dynamic) arrays
  - linked lists
  - (balanced) binary trees

Sparse Matrix - representation example

- For the previous example we would keep the following triples: $< 1, 3, 3 >$, $< 1, 5, 5 >$, $< 2, 1, 2 >$, $< 3, 6, 4 >$, $< 4, 1, 1 >$, $< 4, 4, 7 >$, $< 5, 2, 6 >$, $< 5, 6, 5 >$, $< 6, 3, 9 >$, $< 6, 4, 1 >$.
- We need to retain the dimensions of the matrix as well (we might have last line(s) or column(s) with only 0 values).

Sparse Matrix - operations

- Operations of a sparse matrix are exactly the same as the operations for a *regular* matrix. The most difficult operation is *modify*, because here we have 4 different cases, based on the current value at line *i* and column *j* (we will call it *old_value*) and the value we want to put there (*new_value*).

Sparse Matrix - operations

- Operations of a sparse matrix are exactly the same as the operations for a *regular* matrix. The most difficult operation is *modify*, because here we have 4 different cases, based on the current value at line i and column j (we will call it *old_value*) and the value we want to put there (*new_value*).
  - $old\_value = 0$ and $new\_value = 0 \Rightarrow$ do nothing
  - $old\_value = 0$ and $new\_value \neq 0 \Rightarrow$ add a new triple/node with *new_value*
  - $old\_value \neq 0$ and $new\_value = 0 \Rightarrow$ delete the triple/node with *old_value*
  - $old\_value \neq 0$ and $new\_value \neq 0 \Rightarrow$ modify the value from the triple/node to *new_value*

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:** $last \leftarrow p \in TPosition$

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

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:**

$$previous \leftarrow q \in TPosition$$

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

  - **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 (but we have iterator as well for the List).
- 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.

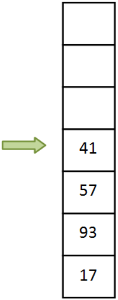
- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

Stack III

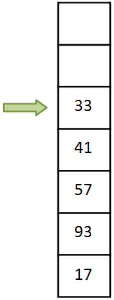
- A stack with no elements is called an *empty stack*.
- When a new stack is created, it can have a fixed capacity. If the number of elements in the stack is equal to this capacity, we say that the *stack is full*.

Stack Example

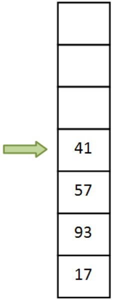
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

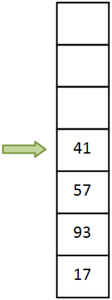


- We *pop* an element:

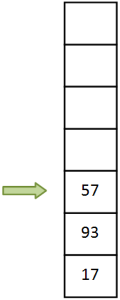


Stack Example II

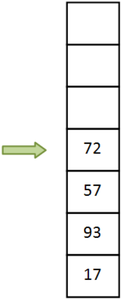
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



Stack Interface I

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

- 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[]

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+1] ← e
  s.top ← 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 ← s.elements[s.top]
   s.top ← s.top -1
   pop← 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]
  top← topElem
end-function

- Complexity: $\Theta(1)$

IsEmpty - Implementation using a static array

function isEmpty(s) **is:**
  if s.top $=$ 0 **then**
    isEmpty $\leftarrow$ True
  else
    isEmpty $\leftarrow$ 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.

Written test - General Info

- Will be at the first half of seminar 5
- Will last 50 minutes
- **Everybody has to participate at the test with his/her own group!**

Written test - Subjects I

- Each subject will be of the form: Container (ADT) + Representation (DS) + Operation
- Containers:
  - Bag
  - Set
  - Map
  - MultiMap
  - List
  - sorted version of the above containers
  - (Sparse) Matrix

Written test - Subjects II

- Data structures
  - Singly Linked List with Dynamic Allocation
  - Doubly Linked List with Dynamic Allocation
  - Singly Linked List on an Array
  - Doubly Linked List on an Array

Written test - Subjects III

- Operation
  - Add
  - Remove
  - Search
  - For Sparse Matrix: Modify value (from 0 to non-zero or from non-zero to 0), search for element from position (i,j)
- Example of a subject: **ADT Set - represented on a doubly linked list on an array - operation: add**.

Written test - Grading

- **1p - Start**
- **0.5p - Specification of the operation** - header, preconditions, postconditions
- **0.5p - Short description of the container**
- **1.25p - representation**
  - 1p - structure(s) needed for the container
  - 0.25p - structure for the iterator for the container

Written test - Grading II

- **4.5p - Implementation of the operation in pseudocode**
  - If you have a data structure with dynamic allocation, you can use the *allocate* and *deallocate/free* operations. If you call any other function(s) in the implementation, you have to implement them.
  - If you have a data structure on an array, you do not need to write code for *resize*-ing the data structure, but you need to show where the resize part would be:

```
...
if s.firstEmpty = -1 then
    @resize
end-if
```

Written test - Grading III

- **1.25p - Complexity**
  - 0.25p - Best Case - with explanations
  - 0.5p - Worst Case - with computations
  - 0.5p - Average Case - with computations
- **1p - Style**
  - Is it general (uses TElem, TComp, a generic Relation)?
  - Is it efficient?
  - etc.