DATA STRUCTURES

ADT List. ADT Stack. ADT Queue.

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In Lecture 6...

- Sorted Linked Lists
- Linked Lists on Array

Today

- ADT List
- ADT Stack
- ADT Queue

ADT List

A **List** is a container which is either *empty* or

- it has a unique *first* element
- it has a unique *last* element
- every element (except for the last) has a unique *successor*
- every element (except for the first) has a unique *predecessor*

ADT List - Positions

Every element from a list has a unique position in the list that:

- identifies the element in the list
- determines the positions of its successor and predecessor (if they exist)

For generality, we will consider that positions are of type *TPosition*.

ADT - List - Positions

A position *p* will be considered *valid* if it the position of an actual element from the list:

- if *p* is a pointer, *p* is valid if it is the address of an element from a list and not NIL
- if *p* is the rank of the element from the list, *p* is valid if it is between 1 and the size of the list.

For an invalid position we will use the following notation: $\perp$

ADT List

- Domain of the ADT List:

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

ADT List

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

ADT List

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

ADT List

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

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

ADT List

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

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

ADT List

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

ADT List

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

$$next = q \in TPosition, q = \begin{cases} \text{the position after p} & \text{if p is not the last position} \\ \perp & \textit{otherwise} \end{cases}$$

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

ADT List

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

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

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

ADT List

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

ADT List

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

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

ADT List

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

ADT List

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT List

- addToEnd(l, e)
  - **descr:**adds a new element to the end of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT List

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

ADT List

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

ADT List

- remove(l, p)
  - **descr:** removes an element from a given position from a list
  - **pre:** $l \in \mathcal{L}, p \in \mathit{TPosition}, \mathit{valid}(l, p)$
  - **post:** $\mathit{remove} = e$, $e \in \mathit{TElem}$, 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

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove = \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT List

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

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

ADT List

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

ADT List

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

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

ADT List

- 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*, referring to the first element from *l*. If *l* is empty, then *it* is invalid.

TPosition - Integer

In Python and Java, TPosition is Integer, a position being represented by an index.

Python:

insert(int index, E object)
index(E object)

- Returns an integer value, position of the element (or exception if *object* is not in the list)

Java:

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

- Returns the removed element

ADT IndexedList

If we consider that TPosition is an Integer (similar to Python and Java), we have an **Indexed List**.

In the case of an *Indexed List* the operations that work with positions take as parameters integers representing those positions.

There are less operations in the interface of the *Indexe d List*. The operations *first*, *last*, *next*, *previous*, *valid* do not exist.

ADT IndexedList

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

ADT IndexedList

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

ADT IndexedList

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

ADT IndexedList

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

$$position = i \in \mathcal{N}$$

$$\text{i = } \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ -1 & otherwise \end{cases}$$

ADT IndexedList

- setElement(l, i, e)
  - **descr:** replaces an element from a position with another
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, *i* is a valid position
  - **post:** $l' \in \mathcal{L}$, the element from position *i* from *l'* is e, $setElement = el$, $el \in TElem$, *el* is the element from position *i* from *l* (returns the previous value from the position)
  - **throws:** exception if *i* is not valid

ADT IndexedList

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IndexedList

- addToEnd(l, e)
  - **descr:**adds a new element to the end of a list
  - **pre:** $l \in \mathcal{L}$, $e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT IndexedList

- addToPosition(l, i, e)
  - **descr:** inserts a new element at a given position (it is the same as *addBeforePosition*)
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position (size + 1 is valid for adding an element)
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l at the position i
  - **throws:** exception if i is not valid

ADT IndexedList

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

ADT IndexedList

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove = \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT IndexedList

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

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

ADT IndexedList

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

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

ADT IndexedList

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

ADT IndexedList

- 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*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

TPosition - Iterator

In STL (C++), TPosition is represented by an iterator.

list:

iterator insert(iterator position, const value_type& val)

- Insert the element *val* before the element referred by the iterator and returns an iterator that points to the newly inserted element

iterator erase(iterator position);

- Deletes the element referred by the iterator and returns an iterator that points to element that followed the element erased by the function call.

ADT IteratedList

If we consider that TPosition is an Iterator (similar to C++) we have an **Iterated List**.

In case of an *Iterated List* the operations that take as parameters positions work with iterators.

Operations *valid*, *next*, *previous* no longer exist in the interface (they are operations for the Iterator).

ADT IteratedList

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

ADT IteratedList

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

ADT IteratedList

- first(l)
  - **descr:** returns an Iterator set to the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first = it \in Iterator$

$$it = \begin{cases} \text{an iterator set to the first element} & \text{if } \mathrm{l} \neq \emptyset \\ \text{an invalid iterator} & \textit{otherwise} \end{cases}$$

ADT IteratedList

- last(l)
  - **descr:** returns an Iterator set to the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last = it \in Iterator$

$$it = \begin{cases} \text{an iterator set to the last element} & \text{if } l \neq \emptyset \\ \text{an invalid iterator} & \textit{otherwise} \end{cases}$$

ADT IteratedList

- getElement(l, it)
  - **descr:** returns the element from the position denoted by an Iterator
  - **pre:** $l \in \mathcal{L}$, $it \in Iterator$, $valid(it)$
  - **post:** $getElement = e$, $e \in TElem$, e = the element from l from the current position
  - **throws:** exception if *it* is not valid

ADT IteratedList

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

$$position = it \in Iterator$$

$$it = \begin{cases} \text{an iterator set to the first position of element e from l} & \text{if } e \in l \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList

- setElement(l, it, e)
  - **descr:** replaces the element from the position denoted by an Iterator with another element
  - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem$, valid(it)
  - **post:** $l' \in \mathcal{L}$, the element from the position denoted by *it* from *l'* is e, $setElement = el$, $el \in TElem$, *el* is the element from the current position from *it* from *l* (returns the previous value from the position)
  - **throws:** exception if *it* is not valid

ADT IteratedList

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IteratedList

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

ADT IteratedList

- addToPosition(l, it, e)
  - **descr:** inserts a new element at a given position specified by the iterator (it is the same as *addAfterPosition*)
  - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem$, $valid(it)$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element *e* was added in l at the position specified by *it*
  - **throws:** exception if *it* is not valid

ADT IteratedList

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

ADT IteratedList

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove = \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT IteratedList

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

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

ADT IteratedList

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

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

ADT IteratedList

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

ADT List - Applications

Applications of ADT List:

Music player

- A music player can use a list to allow switching to the next/previous or playing the song at a given position

Web browsers

- To keep track of the visited pages

File history

- To store the version history of a file

Multi-player games

- To keep the track of turns

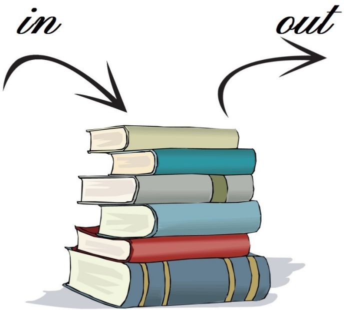
ADT Stack

The ADT **Stack** represents a container in which the access to the elements is restricted to one end of the container, called the *top* of the stack.

- A new element is automatically added to the top.
- The only element that can be removed is the one from the top.
- Only the element from the top can be accessed.

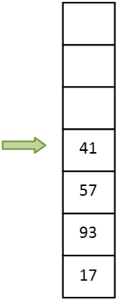
ADT Stack

Due to this restricted access, the stack is said to have a **L**ast **I**n, **F**irst **O**ut **LIFO** policy.



ADT Stack Example

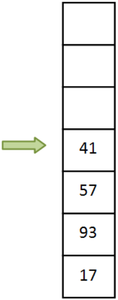
- Suppose that we have the following stack (green arrow shows the top of the stack):



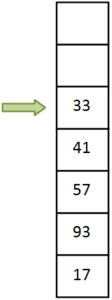
- We *push* the number 33:

ADT Stack Example

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



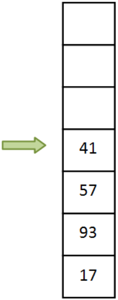
- We *push* the number 33:



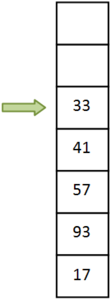
- We *pop* an element:

ADT Stack Example

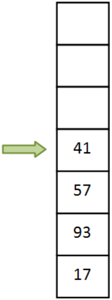
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

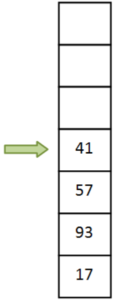


- We *pop* an element:



ADT Stack Example

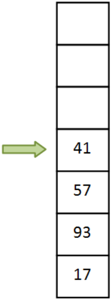
- This is our stack:



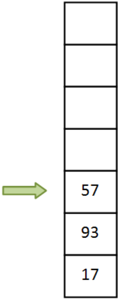
- We *pop* another element:

ADT Stack Example

- This is our stack:



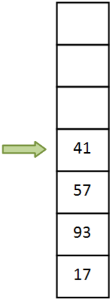
- We *pop* another element:



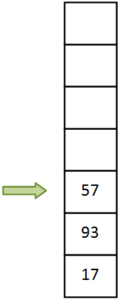
- We *push* the number 72:

ADT Stack Example

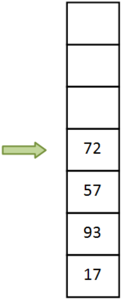
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



ADT Stack - Interface

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

ADT Stack - Interface

- init(s)
  - **descr:** creates a new empty stack
  - **pre:** True
  - **post:** $s \in \mathcal{S}$, s is an empty stack

ADT Stack - Interface

- destroy(s)
  - **descr:** destroys a stack
  - **pre:** $s \in \mathcal{S}$
  - **post:** s was destroyed

ADT Stack - Interface

- push(s, e)
  - **descr:** 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

ADT Stack - Interface

- pop(s)
  - **descr:** pops (removes) the most recent element from the stack
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $pop = e$, e is a *TElem*, e is the most recent element from s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **throws:** an *underflow* exception if the stack is empty

ADT Stack - Interface

- top(s)
  - **descr:** returns the most recent element from the stack (but it does not change the stack)
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $top = e$, e is a *TElem*, e is the most recent element from s
  - **throws:** an *underflow* exception if the stack is empty

ADT Stack - Interface

- isEmpty(s)
  - **descr:** checks if the stack is empty (has no elements)
  - **pre:** $s \in \mathcal{S}$
  - **post:**

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

ADT Stack - Interface

Note: Stacks cannot be iterated, so they don't have an *iterator* operation.

Representation for Stack

Data structures that can be used to represent 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 the *top* at the beginning of the array $\Rightarrow$ every push and pop operation needs to shift every other element to the right or left.

  Place the *top* at the end of the array $\Rightarrow$ push and pop without moving the other elements.

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 ⇒ for every push, pop and top 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 using an array) $\Rightarrow$ we can push and pop elements without iterating through the list.
- Place it at the beginning of the list (like we did when using a SLL) $\Rightarrow$ 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.

Stack - Applications

Applications of stacks:

Expression evaluations

- For evaluating arithmetic expressions

Programming languages

- Function Call Stack

Trees

- For Depth First traversal

Backtracking

- For storing the states in the search space

Considering the queue above, if a new person arrives, where should he/she stand?

When the blue lady finishes, who is going to be the next at the ATM?

ADT Queue

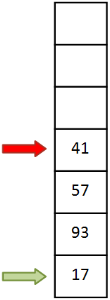
The ADT **Queue** represents a container in which the access to the elements is restricted to the two ends of the container, called *front* and *rear*.

- 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 **F**irst **I**n **F**irst **O**ut (**FIFO**) policy.

ADT Queue - Example

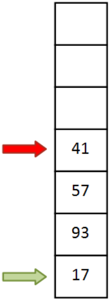
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



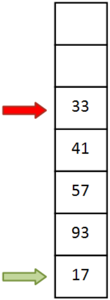
- Push number 33:

ADT Queue - Example

- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



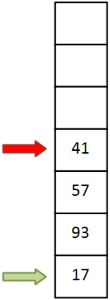
- Push number 33:



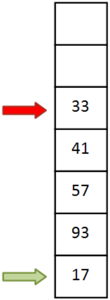
- Pop an element:

ADT Queue - Example

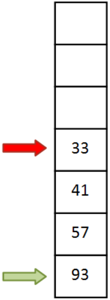
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

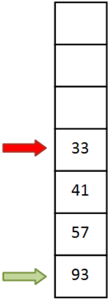


- Pop an element:



ADT Queue - Example

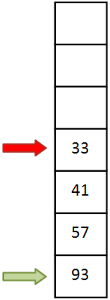
- This is our queue:



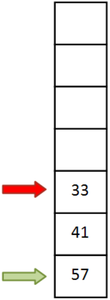
- Pop an element:

ADT Queue - Example

- This is our queue:



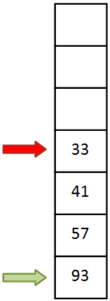
- Pop an element:



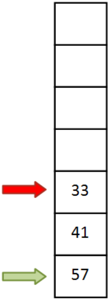
- Push number 72:

ADT Queue - Example

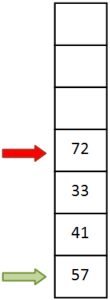
- This is our queue:



- Pop an element:



- Push number 72:



ADT Queue - Interface

- The domain of the ADT Queue:
  $\mathcal{Q} = \{q \mid q \textit{ is a queue with elements of type TElem}\}$

ADT Queue - Interface

- init(q)
  - **descr:** creates a new empty queue
  - **pre:** True
  - **post:** $q \in \mathcal{Q}$, q is an empty queue

ADT Queue - Interface

- destroy(q)
  - **descr:** destroys a queue
  - **pre:** $q \in \mathcal{Q}$
  - **post:** q was destroyed

ADT Queue - Interface

- push(q, e)
  - **descr:** pushes (adds) a new element to the rear of the queue
  - **pre:** $q \in \mathcal{Q}$, e is a *TElem*
  - **post:** $q' \in \mathcal{Q}$, $q' = q \oplus e$, e is the element at the rear of the queue

ADT Queue - Interface

- pop(q)
  - **descr:** pops (removes) the element from the front of the queue
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $pop = e$, e is a *TElem*, e is the element at the front of q, $q' \in \mathcal{Q}$, $q' = q \ominus e$
  - **throws:** an *underflow* exception if the queue is empty

ADT Queue - Interface

- top(q)
  - **descr:** returns the element from the front of the queue (but it does not change the queue)
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $top = e$, e is a *TElem*, e is the element from the front of q
  - **throws:** an *underflow* exception if the queue is empty

ADT Queue - Interface

- isEmpty(q)
  - **descr:** checks if the queue is empty (has no elements)
  - **pre:** $q \in \mathcal{Q}$
  - **post:**

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

ADT Queue - Interface

! Queues cannot be iterated, so they do not have an *iterator* operation.

Queue - representation on a SLL

If we want to represent a Queue using a SLL, where should we place the *front* and the *rear* of the queue?

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

Queue - representation on a SLL

In either case we have one operation (push or pop) with $\Theta(n)$ 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.

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?

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

Queue - Applications

Applications of queues:

Operating systems
- For job scheduling

Websites
- A virtual waiting room implemented using a queue prevents website slow-downs

E-mail queue
- An email queue enables asynchronous communication by creating a buffer of outgoing emails.

Print queue
- To store all active and pending print jobs

Trees
- For Breadth-First traversal

Media player queues
- For sequential playing of the songs in a playlist

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you