Lecture 4

- Containers
  - Bag
  - Set
  - Map
  - Multimap
  - Stack
  - Queue
  - Deque
  - PriorityQueue

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

Previously, in Lecture 3

- Iterators & Containers

  e.g.: dynamic array

Container Bag

Bag

- allow duplicate elements
- no order is guaranteed

Small examples:

- {a, b} = {b, a}
- {a}   <> {a, a}

Terminology:	bag	(*Smalltalk*)
	multiset	*(C++ STL)*
	collection	*(Java.util)*

- A Bag can be represented using several data structures, one of them being a dynamic array (others will be discussed later)

ADT Bag
(as discussed in sem.1)

Operations:
init(b)
add(b, e)
remove(b, e)
search(b, e)
nrOfOccurrences(b, e)
size(b)
iterator(b, it)
destroy(b)

Bag - representation

A Bag can be represented using several data structures, one of them being a dynamic array (others will be discussed later)

- (R1) Store separately every element that was added.
- (R2) Store each element only once and keep a frequency count for it.

(See sem.1)

Two more representations:

- (R3) store the unique elements (in a dynamic array) and store separately the positions from this array for every element that appears in the Bag
- (R4) If the elements of the Bag are integer numbers: we use an array such that the positions of the array represent the elements and the value from the position is the frequency of the element.

Bag – representation: R3

- (R3) store the unique elements in a dynamic array and store separately the positions from this array for every element that appears in the Bag

<u>**e.g.**</u>: <u>Bag</u>: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9			

m items

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6				

n items

(assume 1-based indexing)

Bag – representation: R3

<u>e.g.</u>: (assume 1-based indexing)

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	1	4	5	2	2	2	6	7	4		

Remove element 6

1	2	3	4	5	6	7	8	9
4	1	-5	7	2	9			

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	4	1	4	5	2	2	2	6	3			

Remove element 1

1	2	3	4	5	6	7	8	9
4	1	-5	7	2	9			

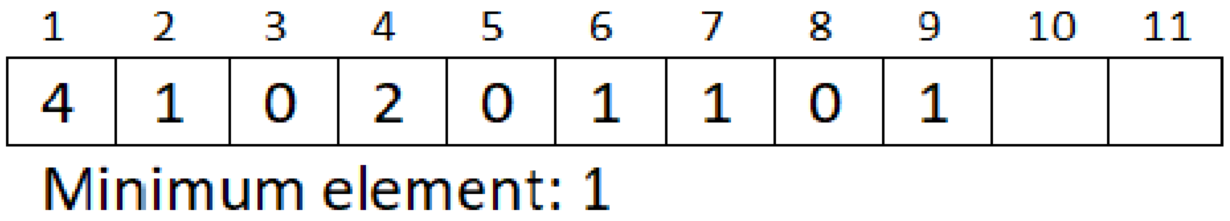
1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	4	1	4	5	2	2	2	6				

Bag – representation: R4

- (R4) If the elements of the Bag are integer numbers: we use an array such that the positions of the array represent the elements and the value from the position is the frequency of the element.
- the frequency of the minimum element is at position 1

  (assume 1-based indexing).

<u>e.g.</u>: Bag: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9



1	2	3	4	5	6	7	8	9	10	11
4	1	0	2	0	1	1	0	1		

Minimum element: 1

Bag – representation: R4

- When indexing starts from 1, the element in the dynamic array that is on position i , represents the actual value:
  minimum + i - 1
- Position of an element e is: *e - minimum + 1*

e.g.: Bag: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9

1	2	3	4	5	6	7	8	9	10	11
4	1	0	2	0	1	1	0	1		

Minimum element: 1

Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

Add element 7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	2	0	1		

Minimum element: -5

Bag – representation: R4

- e.g.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	2	0	1		

Minimum element: -5

Remove element 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

Remove element 1

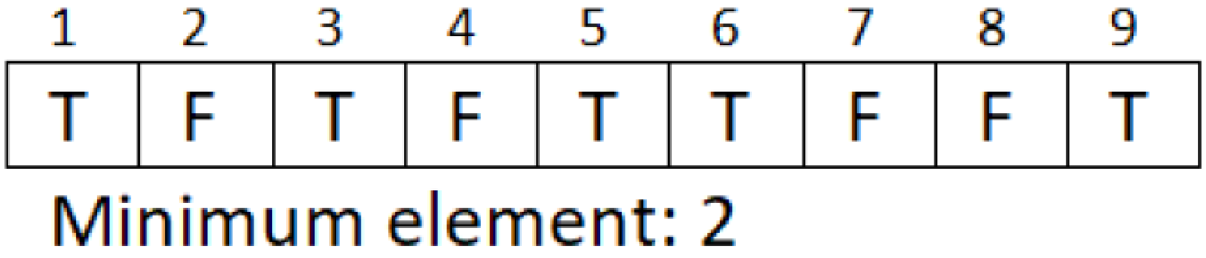
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	3	1	0	2	0	0	2	0	1		

Minimum element: -5

Set – representation: R2

- if elements are integers, we can choose a representation in which the elements are represented by the positions in the dynamic array
- (assume 1-based indexing).

<u>e.g.</u>: <u>Set</u>: 4, 2, 10, 7, 6.



1	2	3	4	5	6	7	8	9
T	F	T	F	T	T	F	F	T

Minimum element: 2

Set – representation: R2

e.g.:

1	2	3	4	5	6	7	8	9
T	F	T	F	T	T	F	F	T

Minimum element: 2

Add element -3

1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	F	F	F	F	T	F	T	F	T	T	F	F	T

Minimum element: -3

Remove element 10

1	2	3	4	5	6	7	8	9	10	11
T	F	F	F	F	T	F	T	F	T	T

Minimum element: -3

Sorted containers

Specific:

- iterator for a sorted container has to return the elements in the order given by a relation.

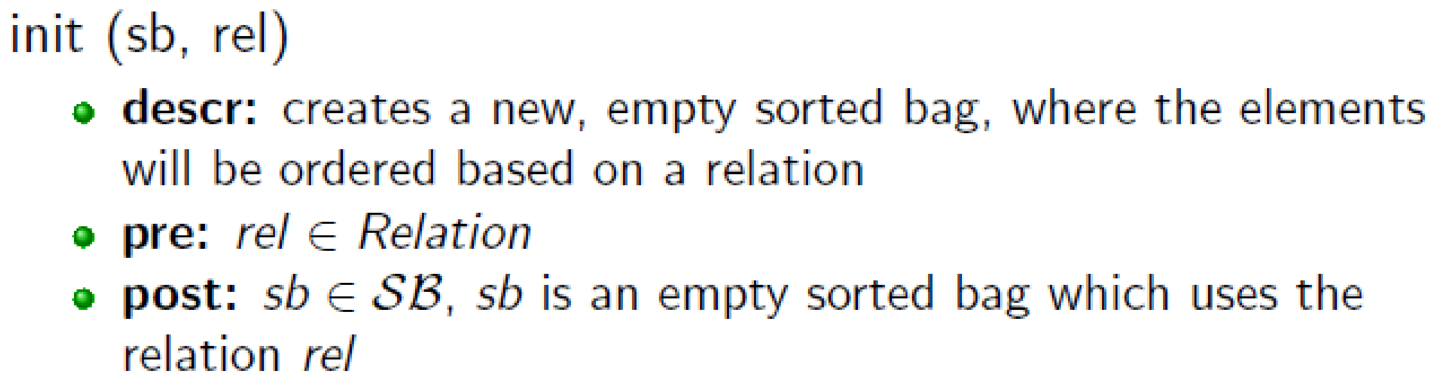
Usually there are two approaches, when we want to order elements:

- Assume that they have a natural ordering, and use this ordering (for ex: alphabetical ordering for strings, ascending ordering for numbers, etc.).
- Sometimes, we want to order the elements in a different way than the natural ordering (or there is no natural ordering) we use a relation. A relation will be considered as a function with two parameters (the two elements that are compared) which returns true if they are in the correct order, or false if they should be reversed.

Sorted containers: SortedBag

SortedBag:

- The only modification in the interface is that the init operation receives a relation as parameter



init (sb, rel)

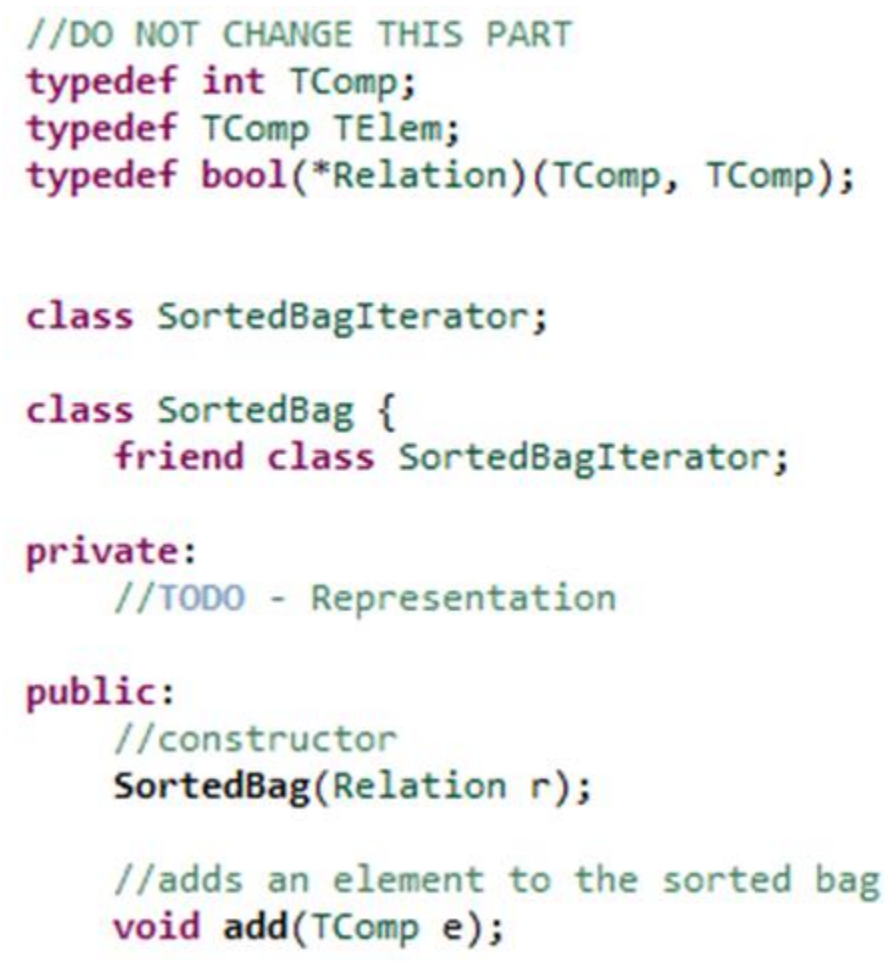
- **descr:** creates a new, empty sorted bag, where the elements will be ordered based on a relation
- **pre:** $rel \in Relation$
- **post:** $sb \in \mathcal{SB}$, sb is an empty sorted bag which uses the relation rel

- The iterator for a SortedBag has to return the elements in the order given by the relation.

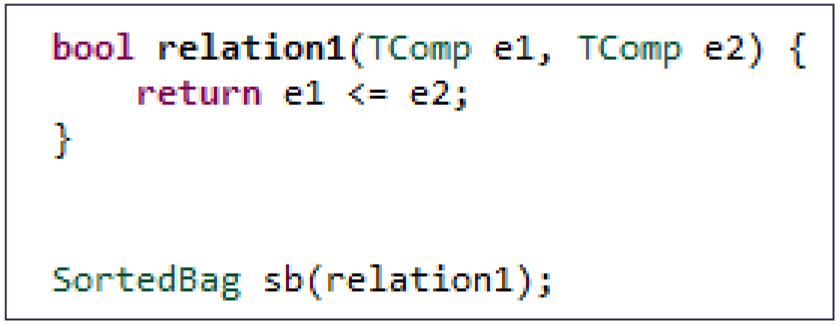
- Do we want the iterator operations to have a **Θ**(1) complexity?

  Consider storing elements ordered based on the relation.

SortedBag at the lab. Small example.



```
//DO NOT CHANGE THIS PART
typedef int TComp;
typedef TComp TElem;
typedef bool(*Relation)(TComp, TComp);


class SortedBagIterator;

class SortedBag {
    friend class SortedBagIterator;

private:
    //TODO - Representation

public:
    //constructor
    SortedBag(Relation r);

    //adds an element to the sorted bag
    void add(TComp e);
```



```
bool relation1(TComp e1, TComp e2) {
    return e1 <= e2;
}


SortedBag sb(relation1);
```

…

SortedBag at the lab. Small example.

```
class SortedBag {

    friend class SortedBagIterator;

private:
    /*representation of SortedBag*/
    int nrElem;
    int cap;
    TComp* elems;
    Relation rel;
    ...
```

```
bool relation1(TComp e1, TComp e2) {
    return e1 <= e2;
}


SortedBag sb(relation1);
```

```
SortedBag::SortedBag(Relation r) {
    this->cap =
    this->nrElem =     ● ● ●
    this->elems =
    this->rel = r;
}

void SortedBag::add(TComp e) {
    if (this->cap == this->nrElem) {
        ● ● ●
    }
    int index = this->nrElem - 1;
    while (index >= 0 && this->rel(this->elems[index], e) == false) {
        this->elems[index + 1] = this->elems[index];
        index--;
    }
    this->elems[index + 1] = e;
    ● ● ●
}
```

See: Lecture04_ADTMap

Map

- init / destroy
- add / remove
- search
- iterator
- size
- isEmpty

- keys
- values
- pairs

Set or Bag ?

- The container in which we store key - value pairs, and where the keys are unique and they are in no particular order

Naming (other):	associative array, dictionary, unique associative container

Sorted Map

- Map **&** an order (a relation) on the set of possible keys.
- The only change in the interface is for the init operation that will receive the relation as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the relation, and the operations keys and pairs return SortedSets.

See: Lecture04_ADTMultiMap

MultiMap

- init / destroy
- add / remove
- search
- iterator
- size
- isEmpty

- keys
- values
- pairs

Set or Bag ?

- The container in which we store key - value pairs, and where the keys are NOT necessarily unique and they are in no particular order.

Sorted MultiMap

- MultiMap **&** an order (a relation) on the set of possible keys. However, if a key has multiple values, they can be in any order (we order the keys only, not the values)
- The only change in the interface is for the init operation that will receive the relation as parameter.
- For a sorted MultiMap, the iterator has to iterate through the pairs in the order given by the relation.

See: Lecture04_ADTStack

Source: https://clipart.wpblink.com/wallpaper-1911442

Stack

- init / destroy
- push / pop
- top
- isEmpty

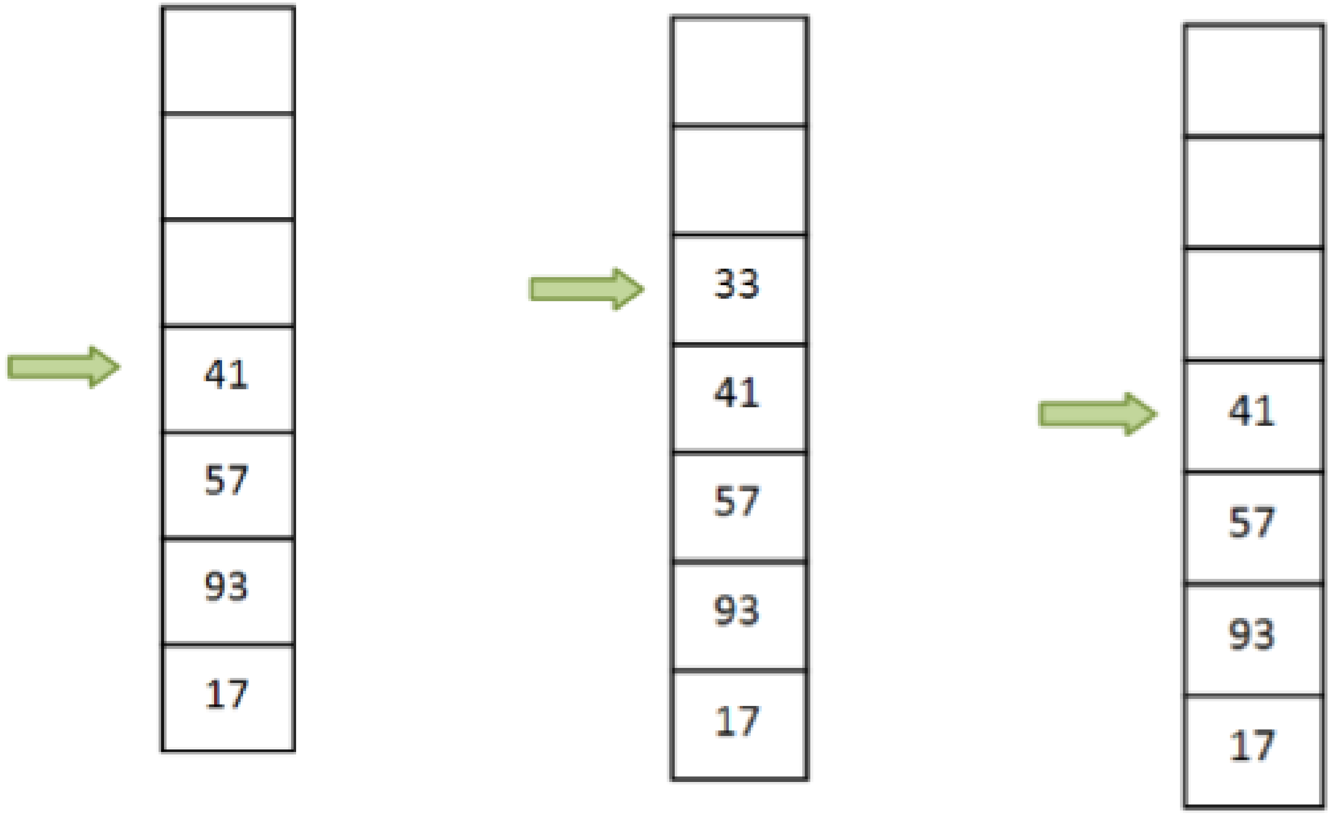
Stack represents a container in which access to the elements is restricted to one end of the container, called the top of the stack.

- 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: Last In, First Out (the last element that was added will be the first element that will be removed)

Stack. Example

- push 33 pop



- How can we implement it over an array ?

See: Lecture04_ADTQueue

Queue

- init / destroy
- push / pop
- top
- isEmpty

http:www.rgbstock.comphotomeZ8AhAQueue+Line

Sometimes, we will use names: enqueue/dequeue instead of push/pop

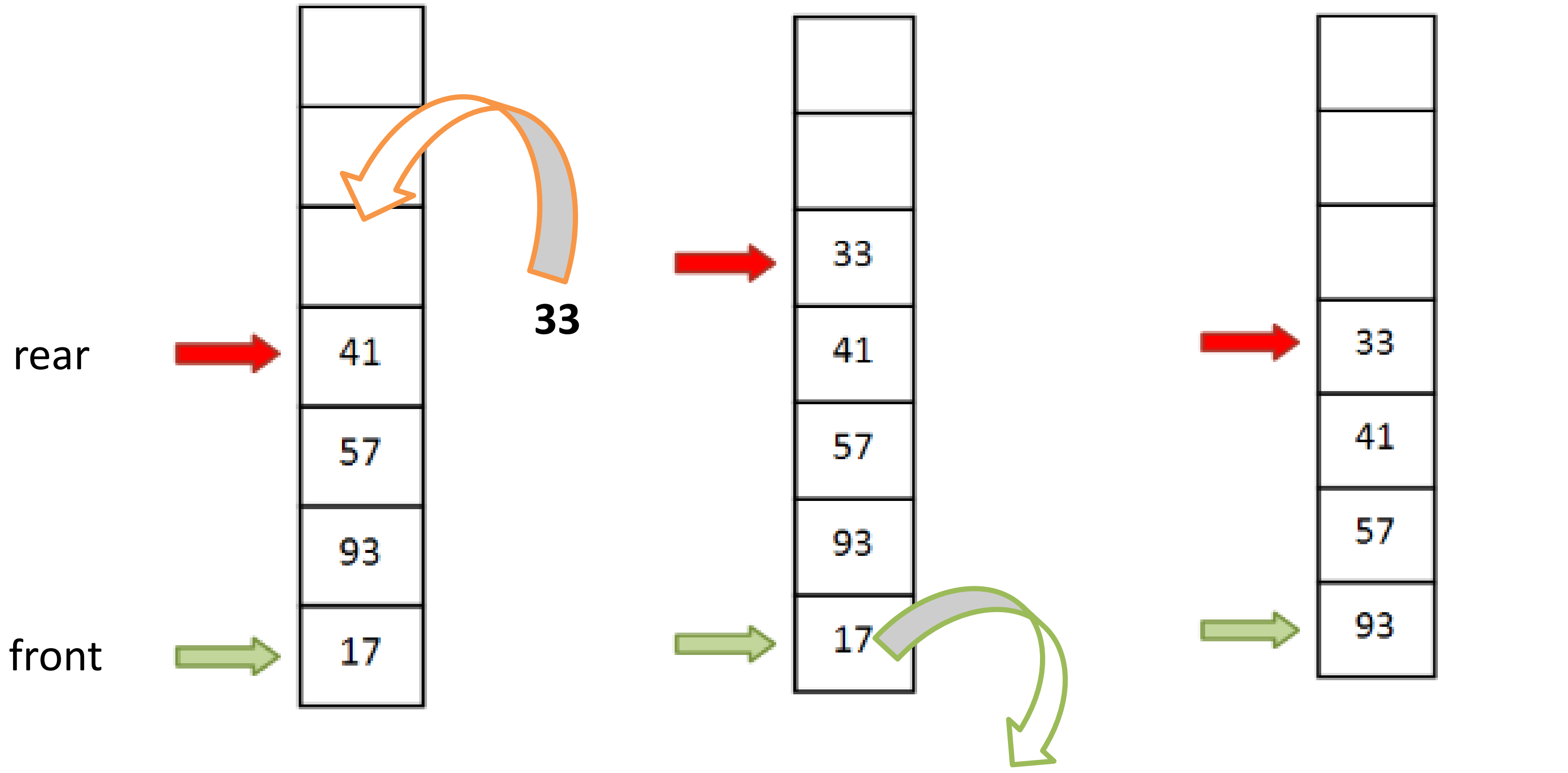
Queue represents a container in which access to the elements is restricted to the two ends of the container, called front and rear.

- When a new element is <u>added</u> (pushed), it has to be added to the <u>rear</u> of the queue.
- When an element is <u>removed</u> (popped), it will be the one at the <u>front</u> of the queue.

Because of this restricted access, the queue is said to have a FIFO policy: First In First Out

Queue. Example

- push pop

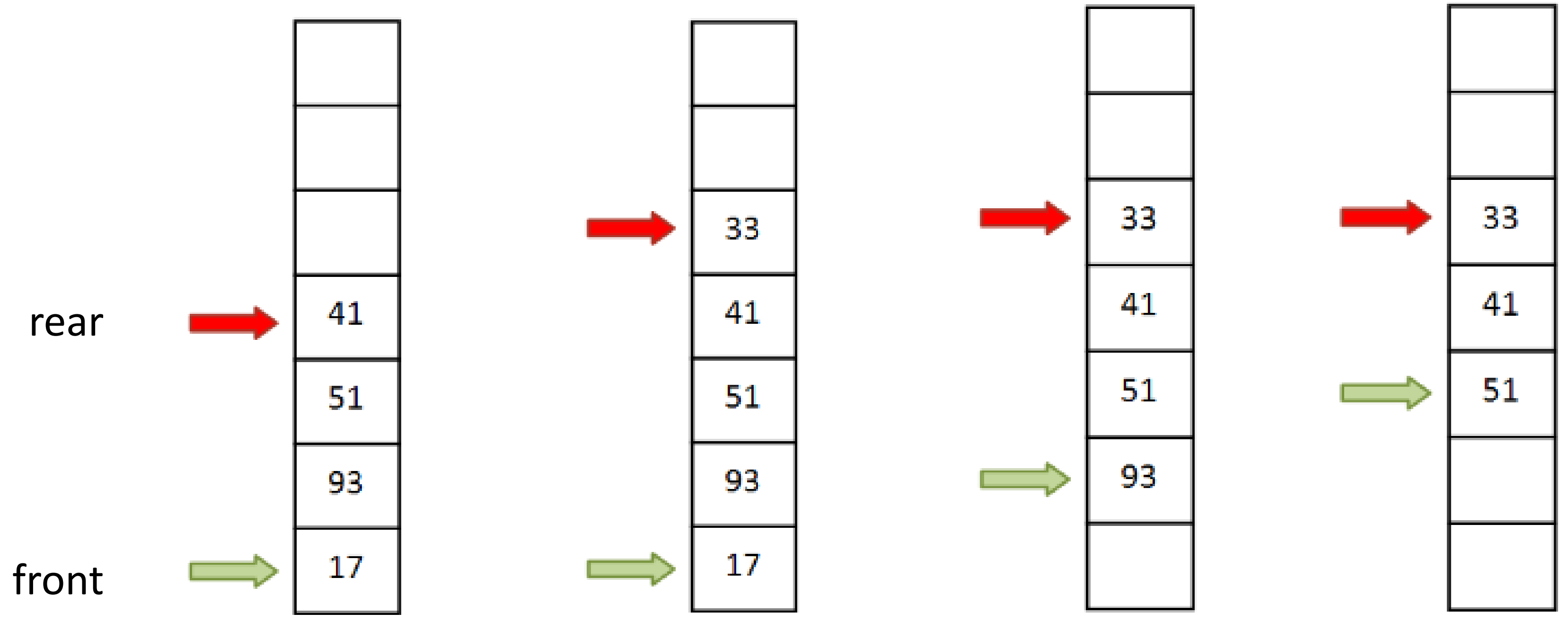


- How can we implement it over an array ?
- What are operations complexities?

(not like this)

Queue. Array based representation

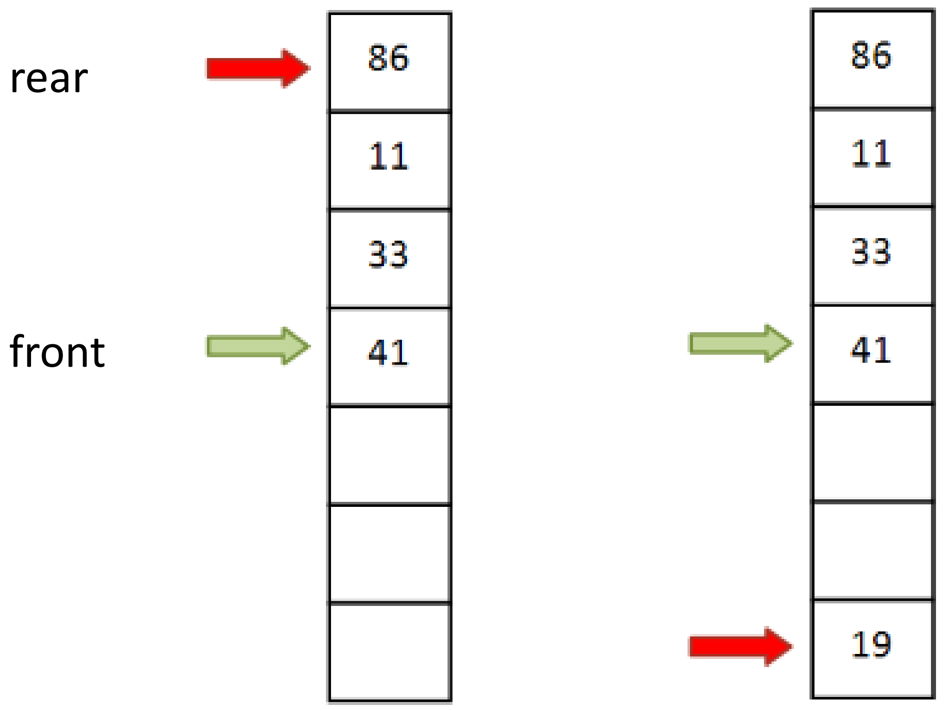
- push 33 pop pop



- How can we implement it over an array ?
- What are operations complexities?

Queue. Array based representation

- push 19

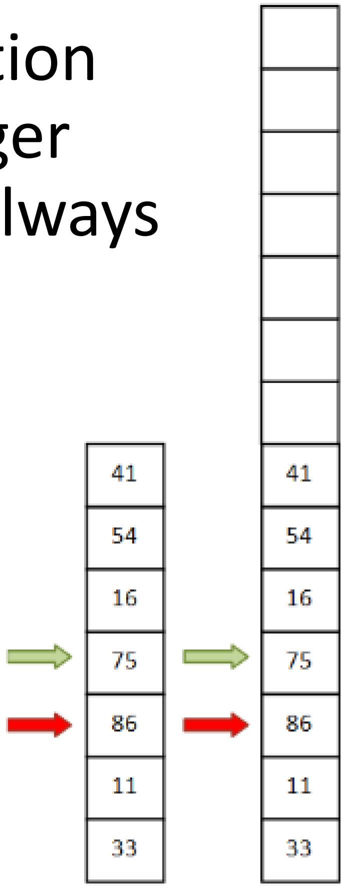


- Use a circular array

Queue. Array based representation

- When pushing a new element we have to check whether the queue is full
- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)

But we have to be careful how we copy the elements in order to avoid having something like:



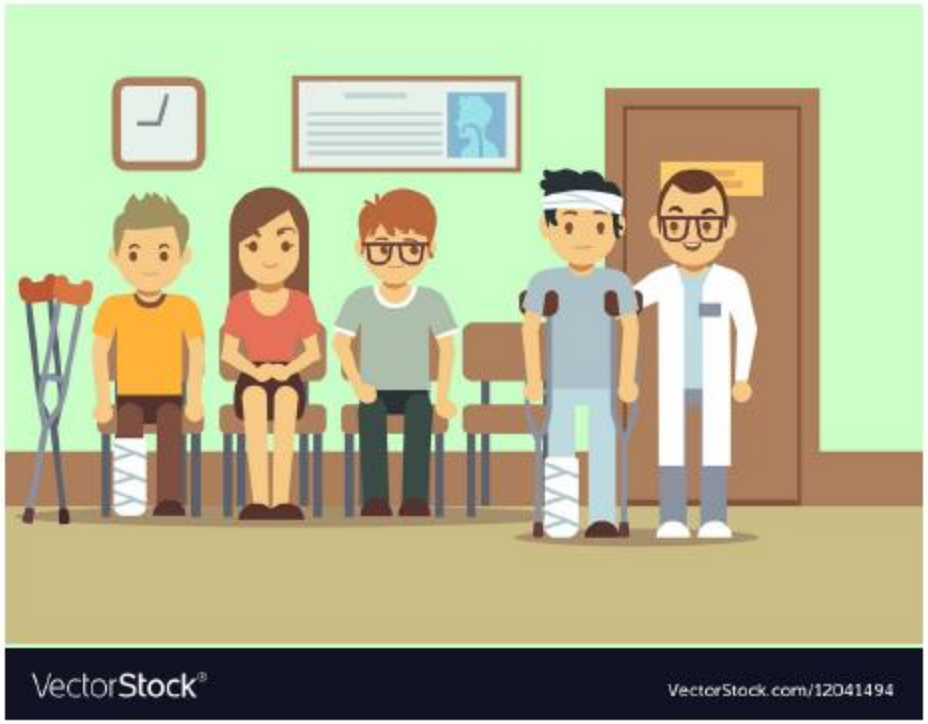
Deque - double ended queue

- insertions/extractions can be made ***from both ends*** **(LIFO + FIFO)**

Operations (naming):

Operation	**Ada**	**C++ STL**	**Java.util**
insert at back	append	***push_back***	addLast
insert at front	appendleft	***push_front***	addFirst
remove last	pop	***pop_back***	removeLast
remove first	popleft	***pop_front***	removeFirst
examine last		back	***getLast***
examine first		front	***getFirst***

- How can we implement it over an array ?
- What are operations complexities?



Source: https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494

Consider the following queue in front of the Emergency Room. Who should be the next person checked by the doctor?

Priority Queue

- Priority Queue is a container

  in which each element has an associated priority (of type TPriority).
- Access to the elements is restricted: we can access only the element with the highest priority. Because of this restricted access, we say that the Priority Queue works based on a <u>Highest Priority First</u> policy.

In order to work in a more general manner:

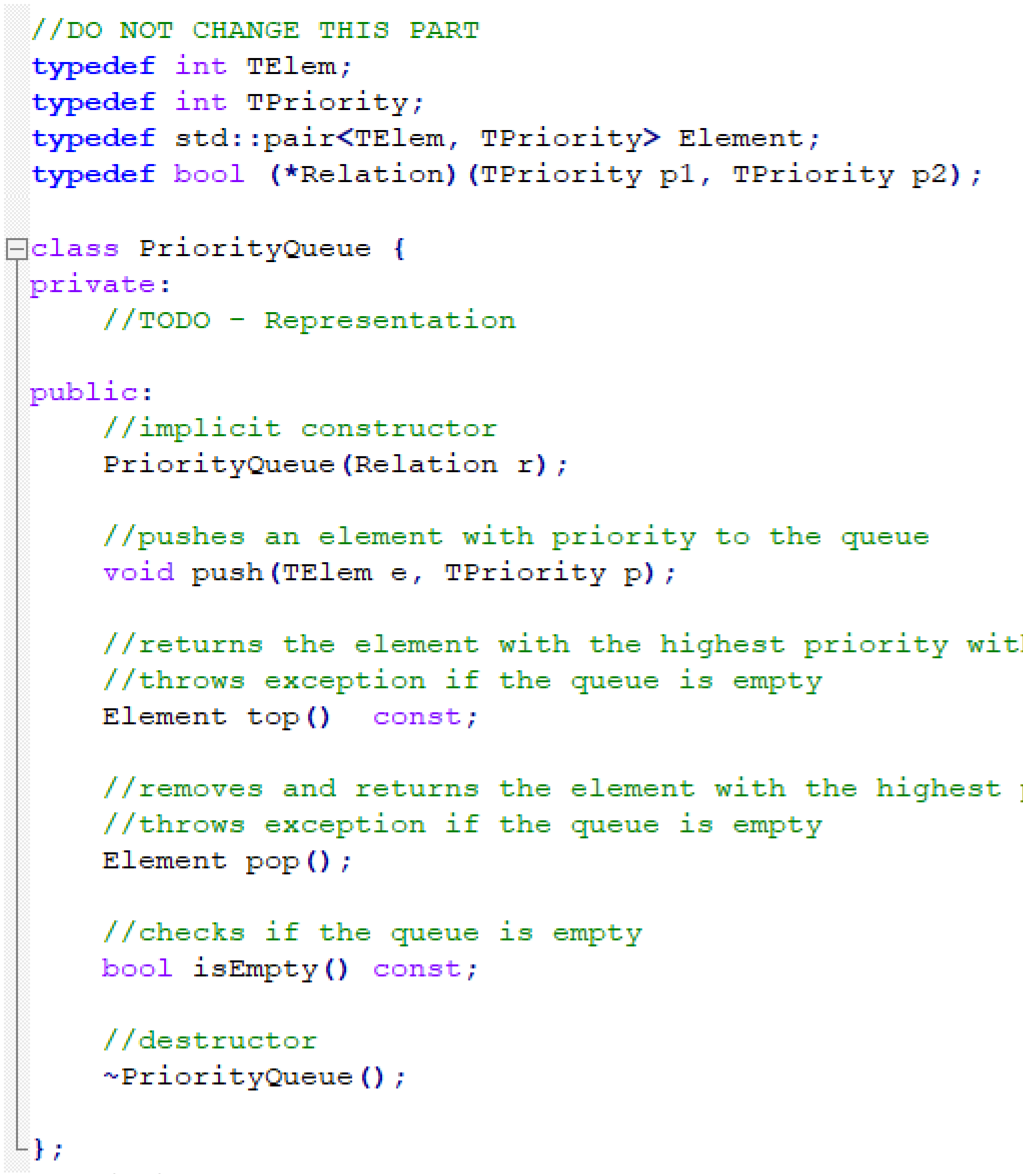
Think about: how can we define the priority ?

Define a relation $\mathfrak{R}$ on the set of priorities:

$$\mathfrak{R} : \text{TPriority} \times \text{TPriority}$$

- When we say the element with the highest priority we will mean that the highest priority is determined using this relation $\mathfrak{R}$.
- If the relation $\mathfrak{R}$ = "$\geq$", the element with the highest priority is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathfrak{R}$ = "$\leq$", the element with the highest priority is the one for which the value of the priority is the lowest (minimum).

Priority Queue



```cpp
//DO NOT CHANGE THIS PART
typedef int TElem;
typedef int TPriority;
typedef std::pair<TElem, TPriority> Element;
typedef bool (*Relation)(TPriority p1, TPriority p2);

class PriorityQueue {
private:
    //TODO - Representation

public:
    //implicit constructor
    PriorityQueue(Relation r);

    //pushes an element with priority to the queue
    void push(TElem e, TPriority p);

    //returns the element with the highest priority with respect to the order relation
    //throws exception if the queue is empty
    Element top()  const;

    //removes and returns the element with the highest priority
    //throws exception if the queue is empty
    Element pop();

    //checks if the queue is empty
    bool isEmpty() const;

    //destructor
    ~PriorityQueue();

};
```

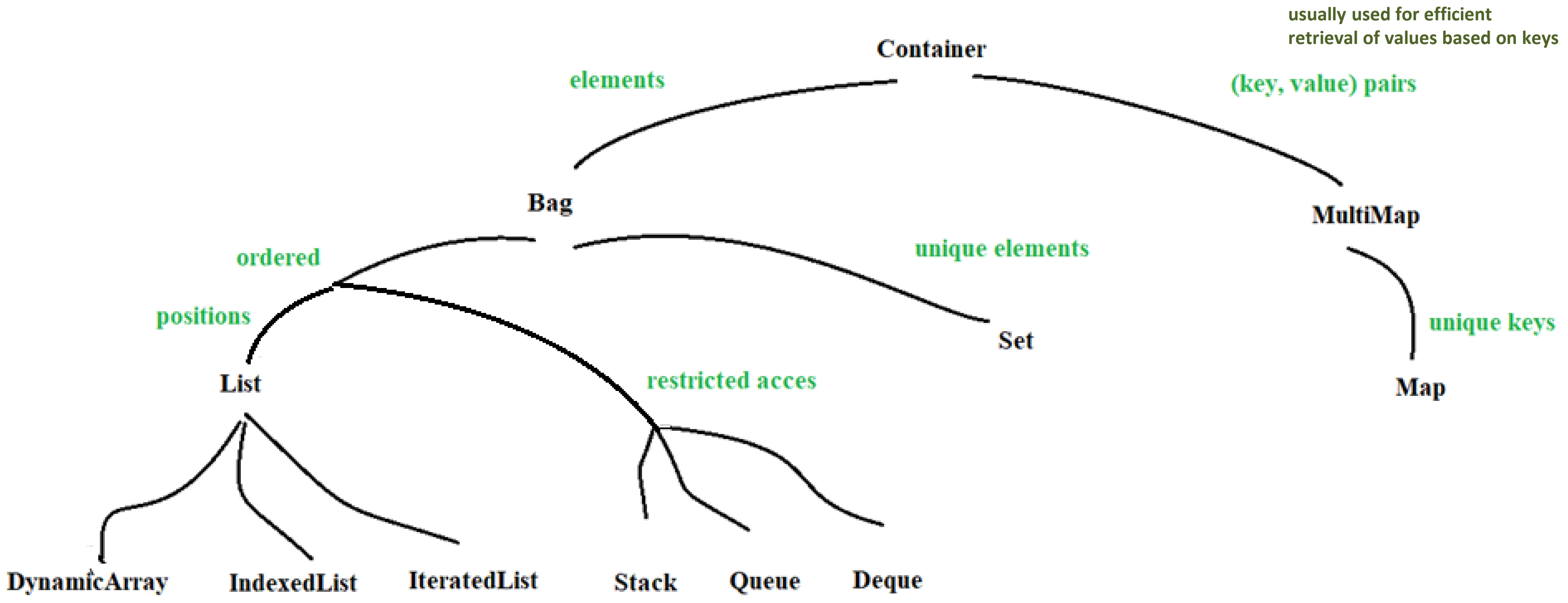
ADT

- See Lecture04_ADTPriority Queue

PriorityQueue.h → lab

Overview

Unsorted containers:


usually used for efficient
retrieval of values based on keys
Container
elements
(key, value) pairs
Bag
MultiMap
ordered
unique elements
positions
Set
unique keys
List
restricted acces
Map
DynamicArray
IndexedList
IteratedList
Stack
Queue
Deque