WORKSHEET WEEK 03: ADTS AND IMPLEMENTATIONS

Abstract Data Types (ADT) allow to ignore implementation details of data structures when using them in larger algorithms. Just like interfaces (API) in programming – you only need to know which operations are supported. ADTs also allow to compare different implementations for stacks, lists, priority queues, sets, etc.

3.1 Concepts and Facts

Definition: A data container C that stores items imposes *extrinsic order* iff this container ensures predictable order of how they can be visited or retrieved. Namely, for any two elements $a, b \in C$, either that a *preceeds* b or a *succeeds* b in this container.

Example: Items stored in a linked list or in an array can be visited in a predictable order such as $A[0], A[1], \ldots$. This extrinsic order does not need to be related with possible ordering of items themselves (items in an array are not always stored in increasing or decreasing order).

Definition: Items a, b stored in a data container have *intrinsic order* iff they can be compared by themselves. Such as $a = b$, $a < b$, or $a > b$. (Some data structures such as Binary Search Trees (BSTs) store keys according to their intrinsic order).

There may be data structures (such as sets stored in a hashtable) that do not use either intrinsic or extrinsic order. We can iterate over such containers, but will get the elements in some unpredictable order.

Definition: A data structure is *immutable*, if it cannot change its state after its creation Other data structures that can be changed are called *mutable*.

Example: A `string` data structure is immutable in many programming languages – a string object stays unchanged throughout its lifetime. Appending new characters or modifying the string is possible, but the result is always a new string object.

Immutable data structures can be easily passed as parameters to functions (just by copying its address or reference, but without cloning the data). Immutable data structures also can serve as keys in larger data structures.

Definition: Some property for a mutable data structure is a *representation invariant*, if it is necessary for any consistent state of this data structure and it is preserved during the ADT operations. Namely, if the property was true before the operation (insertion, deletion etc.), then it must be also true after that operation.

Example: Some mutable data structures (such as `java.util.HashSet`) allow to insert other mutable objects such as `ArrayList` objects into the hashtable. If some object is modified after being inserted, it is not rehashed and its location becomes invalid. At that time the representation invariant of the hashtable is broken – the object cannot be located in its appropriate hashing bucket anymore.

Definition A *stack* is a data structure with extrinsic order – it allows to access (to read or to delete) the element which was the latest to be inserted (LIFO policy). A stack supports the following operations:

S.INITIALIZE() *(create a new empty stack or clear existing contents)*
S.PUSH(x) *(add item x to the top of the stack)*
S.POP() *(remove and return an item from the top of the stack)*
S.ISEMPTY() *(true iff the stack is empty)*

Optionally stacks can support some non-essential operations (can be expressed with the above operations considered essential):

S.TOP() *(return, but do not remove the top element)*
S.SIZE() *(return the number of elements in the stack)*

In practice stacks may also need S.ISFULL() operation to find, if the container has place for one more item. Abstract stacks may assume that the stack is never full.

Definition A *queue* is a data structure with extrinsic order – it allows to access (seeing or deleting) the single element which was the first to be inserted (FIFO policy). A queue supports the following operations:

Q.INITIALIZE() *(create a new empty queue or clear existing contents)*
Q.ENQUEUE(x) *(add item x to the end of the queue)*
Q.DEQUEUE() *(remove and return an item from the front of the queue)*
Q.ISEMPTY() *(true iff the queue is empty)*

Optionally queues can support other operations, but many queue applications do not need them:

Q.FRONT() *(return the front element in the queue without dequeuing)*
S.SIZE() *(return the number of elements in the queue)*

Definition: A *double-ended queue* or a *deque* is a data structure with extrinsic order allowing elements to be added to the front (as in a stack) and also to the back (as in a queue). It supports the following operations:

D.INITIALIZE() *(create a new empty deque or clear existing contents)*
D.PUSHFRONT(x) *(add item x to the front of the deque)*
D.POPFRONT() *(remove and return an item from the front of the deque)*
D.PUSHBACK(x) *(add item x to the back of the deque)*
D.POPBACK() *(remove and return an item from the back of the deque)*
D.FRONT() *(return the front element in the deque without removing it)*
D.BACK() *(return the back element in the deque without removing it)*
D.ISEMPTY() *(true iff the deque is empty)*
D.SIZE() *(return the number of elements in the deque)*

Optionally, a deque can be used to search for an item x with a given key $x.k$ or to find the predecessor or successor of an item x by its pointer/reference.

D.SEARCH(k) *(return an item x with key k or "nil")*
D.PREDECESSOR(x) *(return the predecessor of x)*
D.SUCCESSOR(x) *(return the successor of x)*

Deques are often implemented as doubly linked lists with front and back pointers (and the current size maintained in a separate variable). In this case pushing and popping items happens in constant time, searching requires scanning through the list in $O(n)$ time. And predecessors/successors can be found by following the `next` and `prev` pointers.

STL class `vector` in C++ supports the deque operations, but also allows random access just as in array (returning an element by its index). Similar things are supported by Python lists or by :math:`java.util.List` interface. Such data structures are more powerful than either stacks, queues or deques – they behave like arrays (without strict limit on size).

3.2 Problems

Problem 1: In some Pythons implementations, the dynamic array is grown by $n/8$ whenever the list overflows. Assume that r is the ratio between inserts and reads for this dynamic array. Find the value r for which this growth factor is the optimal one.

Problem 2: Reverse the order of elements in stack S in the following ways:

(A) Use two additional stacks, but no auxiliary variables.

(B) Use one additional stack and some additional non-array variables (i.e. cannot use lists, sequences, stacks or queues).

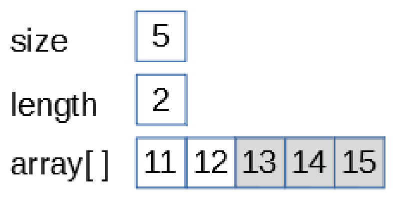
Problem 3: Read elements from an iterator and place them on a stack S in ascending order using one additional stack and some additional non-array variables.

Problem 4: Given a data structure implementing the Sequence ADT, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)

Problem 5: What are the costs for each ADT operation, if a queue is implemented as dynamic array?

Problem 6: Which operations become asymptotically faster, if list ADT is implemented as a doubly linked list (instead of a singly linked list)?

Problem 7: (Stack Implementation as Array): Stack is implemented as an array. In our case the array has size $n = 5$. Stack contains integer numbers; initially the array has the following content.



Stack has the physical representation with `length` $= 2$ (the number of elements in the stack), `size` $= 5$ (maximal number of elements contained in the stack). We have the following fragment:

```
pop();
push(21);
push(22);
pop();
push(23);
```

(continues on next page)

(continued from previous page)

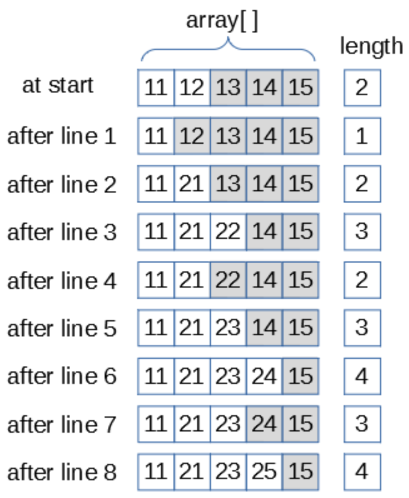
```
push(24);
pop();
push(25);
```

Draw the state of the array after every command. (Every `push(elt)` command assigns a new element into the element `array[length]`, then increments `length` by 1. The command `pop()` does not modify the array, but decreases `length` by 1.

If the command cannot be executed (`pop()` on an empty stack; `push(elt)` on a full stack), then the stack structure does not change at all (`array` or `length` are not modified). To help imagine the state of this stack, you can shade those cells that do not belong to the array.

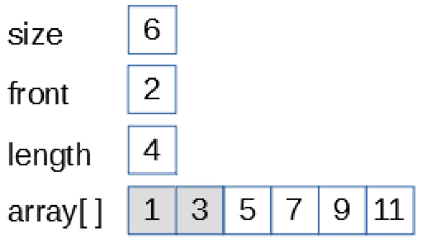
Answer:

The figure below shows stack memory states after each of the 8 operations above are completed:



	array[ ]					length
at start	11	12	13	14	15	2
after line 1	11	12	13	14	15	1
after line 2	11	21	13	14	15	2
after line 3	11	21	22	14	15	3
after line 4	11	21	22	14	15	2
after line 5	11	21	23	14	15	3
after line 6	11	21	23	24	15	4
after line 7	11	21	23	24	15	3
after line 8	11	21	23	25	15	4

Problem 8 (Queue Implementation as a Circular Array): A queue is implemented as an array with `size` elements; it has two extra variables `front` (pointer to the first element) and `length` (the current number of elements in the queue). Current state is shown in the figure:



Enumeration of array elements starts with 0. The array is filled in a circular fashion. The command `enqueue(elt)` inserts a new element at

$$(\texttt{front} + \texttt{length}) \bmod \texttt{size},$$

where mod means the remainder when dividing by `size`. It also increments the `length` element.

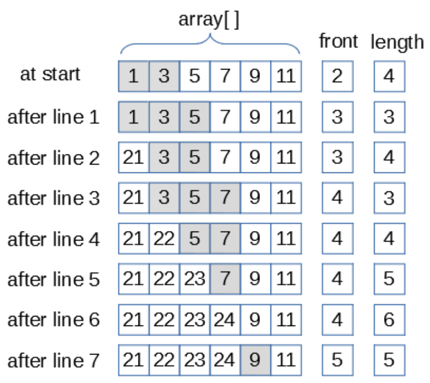
The command `dequeue()` does not change anything in the array, but increments `front` by 1 and decreases $length$ by 1. Thus the queue becomes shorter by 1.

```
dequeue();
enqueue(21);
dequeue();
enqueue(22);
enqueue(23);
enqueue(24);
dequeue();
```

Show the state of the array after every command – `array`, `length`, `front` variables after every line. (Shade the unused cells.)

Answer:

The figure below shows queue memory states after each of the operations above are completed:



	array[ ]						front	length
at start	▨ 1	▨ 3	5	7	9	11	2	4
after line 1	▨ 1	▨ 3	▨ 5	7	9	11	3	3
after line 2	21	▨ 3	▨ 5	7	9	11	3	4
after line 3	21	▨ 3	▨ 5	▨ 7	9	11	4	3
after line 4	21	22	▨ 5	▨ 7	9	11	4	4
after line 5	21	22	23	▨ 7	9	11	4	5
after line 6	21	22	23	24	9	11	4	6
after line 7	21	22	23	24	▨ 9	11	5	5