DATA STRUCTURES

Dynamic Array. Amortized complexity. Iterator.

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University, Faculty of Mathematics and Computer Science

In Lecture 1...

- Abstract Data Types
- Data Structures
- Pseudocode conventions
- Algorithm Complexity Analysis
  - O, Ω and Θ notations
  - Best case, worst case, average case
  - Complexity analysis for recursive algorithms

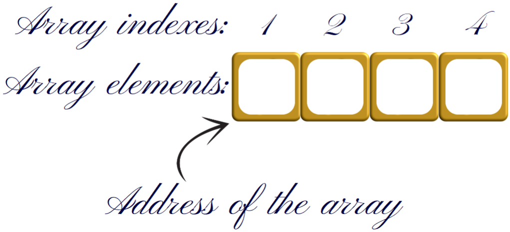
Today

- Dynamic Array
- Amortized complexity analysis
- Iterator

Arrays

The array is one of the simplest and most basic data structures.

An array can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block.



Arrays

- When a new array is created we have to specify two things:

  1. The type of the elements in the array
  2. The maximum number of elements that can be stored in the array (*capacity* of the array)

- The memory occupied by the array is the capacity times the size of one element.

- The array itself is referenced by the address of its first element.

Arrays - C++ Example 1

An array of *boolean* values (which occupy one byte)

```
Size of bool: 1
Address of the array: 0x781a60 7871072
Address of the element from index 0: 0x781a60 7871072
Address of the element from index 1: 0x781a61 7871073
Address of the element from index 2: 0x781a62 7871074
Address of the element from index 3: 0x781a63 7871075
Address of the element from index 4: 0x781a64 7871076
Address of the element from index 5: 0x781a65 7871077
```

Can you guess the address of the element from index 6?

Arrays - C++ Example 2

An array of *integer* values (which occupy 4 bytes)

```
Size of int: 4
Address of the array: 0xf31a60 15932000
Address of the element from index 0: 0xf31a60 15932000
Address of the element from index 1: 0xf31a64 15932004
Address of the element from index 2: 0xf31a68 15932008
Address of the element from index 3: 0xf31a6c 15932012
Address of the element from index 4: 0xf31a70 15932016
Address of the element from index 5: 0xf31a74 15932020
```

Can you guess the address of the element from index 6?

Arrays - advantage

The main **advantage** of arrays is that any element of the array can be accessed in constant time ($\Theta(1)$).

Computing the address of an array's element:

Address of i^{th} element $=$ address of array + $(i-1)$ * size of an element

Obs: If array indexing starts from 0, the above formula is still valid, but with i instead of $i-1$.

Arrays - disadvantage

An array is a **static** data structure: its capacity is fixed.

Disadvantage: we need to know the number of elements from the beginning:

- if the capacity is too small: we cannot store every element we want to
- if the capacity is too large: we waste memory

Dynamic Array

A **dynamic array** is an array with dynamic capacity. Its capacity can grow or shrink, depending on the number of elements that need to be stored in the array.

Dynamic arrays still have the advantage of accessing any element in $\Theta(1)$ time.

Dynamic Array - Representation

- For representing a Dynamic Array we need the following fields:
  - *capacity* - the number of slots allocated for the array
  - *length* - the actual number of elements stored in the array
  - *elems* - the actual array with *capacity* slots allocated

Representation of a Dynamic Array:

Dynamic Array:

capacity: Integer
length: Integer
elems: TElem[]

Dynamic Array - Resize

- When the value of *length* equals the value of *capacity*, the array is full. If more elements need to be added, the *capacity* of the array is increased, the array being *resized*.

- When *resizing* the dynamic array, a new, bigger array is allocated and the existing elements are copied from the old array to the new one.

- Optionally, *resize* can be performed after deletion as well: if the dynamic array becomes "too empty", a resize operation can be performed to shrink its size.

Dynamic Array - DS vs. ADT

Dynamic Array is a data structure:

- It describes how data is actually stored in a the computer's memory and how it can be accessed and processed
- It can be used to implement different container ADTs

Being so frequently used, in most programming languages it exists as a container ADT as well.

- The Dynamic Array is not really an ADT, but we still can treat it as an ADT and discuss its domain and interface.

Dynamic Array - Interface

- **Domain** of ADT Dynamic Array:

$$\mathcal{DA} = \{\textbf{da} |\ da = (capacity, length, e_1 e_2 e_3 ... e_{length}), capacity, length \in N, length \leq capacity, e_i \text{ is of type TElem } \forall i \in \{1, 2, 3, ..., length\}\}$$

Dynamic Array - Interface

- init(da, cp)
  - **description:** creates a new, empty Dynamic Array with initial capacity *cp* (constructor)
  - **pre:** $cp \in N^*$
  - **post:** $da \in \mathcal{DA},\ da.capacity = cp,\ da.length = 0$
  - **throws:** an exception if *cp* is negative or zero

Dynamic Array - Interface

- destroy(da)
  - **description:** destroys a dynamic array
  - **pre:** *da* $\in \mathcal{DA}$
  - **post:** *da* was destroyed (the memory occupied by the dynamic array *da* has been freed)

Dynamic Array - Interface

- size(da)
  - **description:** returns the size (number of elements) of the Dynamic Array *da*
  - **pre:** $da \in \mathcal{DA}$
  - **post:** size = the size of *da* (the number of elements in *da*)

Dynamic Array - Interface

- getElement(da, i)
  - **description:** returns the element from index *i* in the Dynamic Array *da*
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.length$
  - **post:** $getElement = e,\ \ e \in TElem,\ \ e = da.e_i$ (the element from index *i*)
  - **throws:** an exception if *i* is not a valid index for the Dynamic Array *da*

Dynamic Array - Interface

- setElement(da, i, e)
  - **description:** changes the element from a position to another value
  - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.length$, $e \in TElem$
  - **post:** $da' \in \mathcal{DA}, da'.e_i = e$ (the i^{th} element from *da'* becomes *e*), setElement = $da.e_i$ (returns the old value from index *i*)
  - **throws:** an exception if *i* is not a valid index for the Dinamyc Array *da*

Dynamic Array - Interface

- addToEnd(da, e)
  - **description:** adds an element to the end of a Dynamic Array. If the array is full, its capacity will be increased
  - **pre:** $da \in \mathcal{DA},\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.length = da.length + 1; da'.e_{da'.length} = e\ (da.capacity = da.length \Rightarrow da'.capacity = da.capacity * 2)$

Dynamic Array - Interface

- addToPosition(da, i, e)
  - **description:** adds an element to a given position in the Dynamic Array. If the array is full, its capacity will be increased.
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.length + 1,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.length = da.length + 1,\ da'.e_j = da.e_{j-1} \forall j = da'.length, da'.length - 1, ..., i+1, da'.e_i = e,\ da'.e_j = da.e_j\ \forall j = i - 1, ..., 1\ (da.capacity = da.length \Rightarrow da'.capacity = da.capacity * 2)$
  - **throws:** an exception if i is not a valid position (da.length+1 is a valid position when adding a new element)

Dynamic Array - Interface

- deleteFromPosition(da, i)
  - **description:** deletes an element from a given position (index) from the Dynamic Array. Returns the deleted element.
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.length$
  - **post:** $deleteFromPosition = e$,
    $e \in TElem,\ e = da.e_i,\ da' \in \mathcal{DA},\ da'.length = da.length - 1,\ da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.length$,
    $da'.e_j = da.e_j, \forall j = 1, ..., i-1$
  - **throws:** an exception if i is not a valid index

Dynamic Array - Interface

- iterator(da, i)
  - **description:** returns an iterator for a Dynamic Array
  - **pre:** *da* $\in \mathcal{DA}$
  - **post:** *it* $\in \mathcal{I}$, *i* is an iterator over *da*

Dynamic Array - Interface

- Other possible operations:
  - Check if the Dynamic Array is empty or not
  - Delete an element (giving the element itself and not its index)
  - Check if an element appears in the Dynamic Array or not
  - Return the index of a given element
  - etc.

Dynamic Array - Implementation

Most operations from the interface of the Dynamic Array are very easy to implement.

In the following we will discuss the implementation of three operations: *addToEnd*, *addToPosition* and *deleteFromPosition*.

We are going to use the representation discussed earlier.

Representation:

Dynamic Array:

capacity: Integer
length: Integer
elems: TElem[]

Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10

Length: 6

- *Add the element 49 to the end of the dynamic array*

Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 6

- Add the element 49 to the end of the dynamic array

51	32	19	31	47	95	49			
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 7

Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

Capacity: 6

Length: 6

- *Add the element 49 to the end of the dynamic array*

Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

Capacity: 6
Length: 6

- *Add the element 49 to the end of the dynamic array*

51	32	19	31	47	95

51	32	19	31	47	95	49					
1	2	3	4	5	6	7	8	9	10	11	12

Capacity: 12
Length: 7

Dynamic Array - addToEnd

Adding to the end of a Dynamic Array:

```
subalgorithm addToEnd (da, e) is:
    if da.length = da.capacity then
    //The dynamic array is full. We need to resize it first
        da.capacity ← da.capacity * 2
        newElems ← @ an array with da.capacity empty slots
        //We need to copy all the existing elements into newElems
        for index ← 1, da.length execute
            newElems[index] ← da.elems[index]
        end-for
        //Depending on the prog. lang., we may need to free the old elems array
        free(da.elems)
        //We need to replace the old elements array with the new one
        da.elems ← newElems
    end-if
    //Now we certainly have space for the element e
    da.length ← da.length + 1
    da.elems[da.length] ← e
end-subalgorithm
```

Dynamic Array - addToPosition

What is the complexity of *addToEnd*?

Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10

Length: 6

- Add the element 49 to index 3

Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 6

- *Add the element 49 to index 3*

51	32	49	19	31	47	95			
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 7

Dynamic Array - addToPosition

Adding to a given position in a Dynamic Array:

```
subalgorithm addToPosition (da, i, e) is:
    if i > 0 and i ≤ da.length+1 then
        if da.length= da.capacity then //The dynamic array is full. We need to resize it
            da.capacity ← da.capacity * 2
            newElems ← @ an array with da.capacity empty slots
            for index ← 1, da.length execute
                newElems[index] ← da.elems[index]
            end-for
            free(da.elems)
            da.elems ← newElems
        end-if //Now we certainly have space for the element e
        da.length← da.length+1
        for index ← da.length, i+1, -1 execute //Move the elements to the right
            da.elems[index] ← da.elems[index-1]
        end-for
        da.elems[i] ← e
    else
        @throw exception
    end-if
end-subalgorithm
```

Dynamic Array - addToPosition

What is the complexity of *addToPosition*?

Dynamic Array

Obs:

- While it is not mandatory to double the capacity, it is important to compute the new capacity by multiplying the old one with a constant number greater than 1.
- After resizing a dynamic array, its elements will still occupy a contiguous memory block, but a different one.

Dynamic Array - resize

E What are the resize factors in different programming languages?

- C++ - 1.5
- Java - 1.5
- Python - ≈ 1.125
- C# - 2

Dynamic Array - Delete From Position

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10

Length: 6

- *Delete the element at index 3*

Dynamic Array - Delete From Position

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 6

- *Delete the element at index 3*

1 2 3

51	32	31	47	95					
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 5

Dynamic Array - deleteFromPosition

Deleting the element at a given index in a Dynamic Array:

```
subalgorithm deleteFromPosition (da, i) is:
    if i > 0 and i ≤ da.length then
        e ← da.elems[i]
        for index ← i, da.length-1 execute
            da.elems[i] ← da.elems[i+1]
        end-for
        da.length← da.length - 1
        deleteFromPosition ← e
    else
        @throw exception
    end-if
end-subalgorithm
```

What is the complexity of *deleteFromPosition*?

Dynamic Array - Complexity of operations

- size -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd - $\Theta(1)$
- deleteFromPosition -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd - $\Theta(1)$
- deleteFromPosition - $O(n)$
- addToEnd -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd - $\Theta(1)$
- deleteFromPosition - $O(n)$
- addToEnd - $\Theta(1)$ *amortized*

Amortized analysis

The amortized analysis is a method of analyzing algorithms that considers an entire series of operations, computes the cost of the entire series and “charges” each individual operation with a share of the total cost.

We can, at times, obtain tighter bounds using amortized complexity rather than worst-case complexity.

While certain operations may be costly, they cannot occur at a high-enough frequency to weigh down the entire program.

Amortized analysis

addToEnd has the complexity the $O(n)$. Consequently, n calls to the *addToEnd* would have complexity the $O(n^2)$.

But we rarely have to resize a Dynamic Array if we consider a sequence of n operations.

Amortized analysis

- Consider c_i the cost for the i^{th} call to *addToEnd*.
- Supposing that we start with an initial capacity of 1 and we double the capacity at each resizing, at the *i*th operation we perform a resize iff *i-1* is a power of 2. So, the cost of the *i*th operation *i*, c_i, is:

$$c_i = \begin{cases} i, & \text{if i-1 is an exact power of 2} \\ 1, & \text{otherwise} \end{cases}$$

Amortized analysis

The cost of n operations is:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{[log_2 n]} 2^j \leq n + 2^{[log_2 n]+1} - 1 < n + 2n = 3n$$

Since the total cost of n operations is $3n$, we can say that the cost of one operation is 3, which is constant.

Amortized analysis

While the worst case time complexity of *addToEnd* is still $\Theta(n)$, the amortized complexity is $\Theta(1)$.

The amortized complexity is no longer valid if resizing means increasing the capacity by a constant.

In case of *addToPosition*, the complexity remains $O(n)$ - even if resizing is performed rarely, we need to shift elements.

When do we have amortized complexity?

The reason why in case of *addToEnd* we can talk about amortized complexity is that the worst case situation happens rarely.

Whenever you want to determine whether amortized complexity computation is applicable, ask the following question:

- Can we have worst case complexity for two consecutive calls? If the answer is YES, than you do **not** have a situation of amortized complexity.

Amortized analysis

Obs: In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".

How empty should the array become before resizing? Which of the following two strategies do you think is better? Why?

- Wait until the array is only half full (da.length$\approx$ da.capacity/2) and resize it to the half of its capacity
- Wait until the array is only a quarter full (da.length$\approx$ da.capacity/4) and resize it to the half of its capacity

Iterator

An **iterator** is a data type that is used to iterate through the elements of a container.

It offers a generic way of traversing a container, irrespective to its representation.

Every iterable container should include in its interface an operation called *iterator* that will return an iterator over it.

Iterator

An iterator usually contains:

- a reference to the container it iterates over
- a **cursor**, which is reference to a current element from the container

Iterating through the elements of the container means moving the *cursor* from one element to the next until the iterator becomes *invalid*

The exact way of representing the *cursor* of the iterator depends on the data structure used for representing the container.

Iterator - domain

- **Domain** of the Iterator ADT:

$\mathcal{I} = \{$**i** $|$ i is an iterator over a container with elements of type TElem $\}$

Iterator - interface

What operations should we have in the **interface** of the Iterator ADT?

Iterator - interface

- init(i, c)
  - **description:** creates a new iterator for a container
  - **pre:** *c* is a container
  - **post:** $i \in \mathcal{I}$ and *i* refers the first element in *c* if *c* is not empty or *i* is not valid

Iterator - interface

- getCurrent(i)
  - **description:** returns the current element refered by the iterator
  - **pre:** $i \in \mathcal{I}$, i is valid
  - **post:** $getCurrent = e$, $e \in TElem$, e is the current element referred by i
  - **throws:** an exception if i is not valid

Iterator - interface

- next(i)
  - **description:** moves the cursor of the iterator to the next element or makes the iterator invalid if no elements are left
  - **pre:** $i \in \mathcal{I}$, i is valid
  - **post:** $i' \in \mathcal{I}$, the cursor of *i'* points to the next element from the container or, if no more elements are left, i' is invalid
  - **throws:** an exception if i is not valid

Iterator - interface

- valid(i)
  - **description:** verifies if the iterator is valid
  - **pre:** $i \in \mathcal{I}$
  - **post:**

$$valid = \begin{cases} True, & \text{if } i \text{ refers a valid element from the container} \\ False, & \text{otherwise} \end{cases}$$

Iterator - interface

- first(i)
  - **description:** sets the cursor of an iterator to point the first element of the iterated container
  - **pre:** $i \in \mathcal{I}$
  - **post:** $i' \in \mathcal{I}$, the cursor of i' points to the first element of the container, if it is not empty, or i' is invalid, otherwise

Types of iterators

The interface presented presented previously describes the simplest iterator: *unidirectional* and *read-only* .

A *unidirectional* iterator can be used to iterate through a container in one direction only (usually *forward*).

A *read-only* iterator can be used to iterate through the container, but without changing it.

Types of iterators

A **bidirectional** iterator can be used to iterate in both directions. It has a *previous* operation in addition to the *next* operation.

A **random access** iterator allows performing multiple steps at once (not just one step forward or one step backward).

A **read-write** iterator can be used to add/delete elements to/from the container.

Using the iterator

Printing the content of a container using the iterator:

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
    iterator(c, i)
    while valid(i) execute
        //get the current element from the iterator
        elem ← getCurrent(i)
        print elem
        //go to the next element
        next(i)
    end-while
end-subalgorithm
```

Obs: This sub-algorithm is general and can be used to print the elements of any iterable container.

Iterator for a Dynamic Array

- What will be the type of the iterator cursor?

Iterator for a Dynamic Array

- What will be the type of the iterator cursor?
- In case of a Dynamic Array, the simplest way to represent the iterator is to retain the index of the current element.

Representation:

IteratorDA:
    da: Dynamic Array
    current: Integer

Iterator for a Dynamic Array - init

Creating a new iterator:

subalgorithm init(i, da) *is:*
//i is an IteratorDA, da is a Dynamic Array
    i.da ← da
    i.current ← 1
end-subalgorithm

Complexity:

Iterator for a Dynamic Array - init

Creating a new iterator:

subalgorithm init(i, da) *is:*

//i is an IteratorDA, da is a Dynamic Array

    i.da $\leftarrow$ da

    i.current $\leftarrow$ 1

end-subalgorithm

Complexity: $\Theta(1)$

Iterator for a Dynamic Array - getCurrent

Getting the current element:

function getCurrent(i) *is:*
  if not valid(i) **then**
    @throw an exception
  end-if
  getCurrent $\leftarrow$ i.da.elems[i.current]
end-function

Complexity:

Iterator for a Dynamic Array - getCurrent

Getting the current element:

function getCurrent(i) *is:*
  if not valid(i) **then**
    @throw an exception
  end-if
  getCurrent ← i.da.elems[i.current]
end-function

Complexity: $\Theta(1)$

Iterator for a Dynamic Array - next

Moving to the next element in the traversal:

subalgorithm next(i) *is:*
    if not valid(i) **then**
        @throw exception
    end-if
    i.current ← i.current + 1
end-subalgorithm

Complexity:

Iterator for a Dynamic Array - next

Moving to the next element in the traversal:

subalgorithm next(i) *is:*
    if not valid(i) **then**
        @throw exception
    end-if
    i.current $\leftarrow$ i.current + 1
end-subalgorithm

Complexity: $\Theta(1)$

Iterator for a Dynamic Array - valid

Checking if the iterator is valid:

function valid(i) *is:*
    if i.current $<=$ i.da.length **then**
        valid $\leftarrow$ True
    else
        valid $\leftarrow$ False
    end-if
end-function

Complexity: $\Theta(1)$

Iterator for a Dynamic Array - first

Resetting the iterator:

subalgorithm first(i) *is:*
    i.current $\leftarrow$ 1
end-subalgorithm

Complexity: $\Theta(1)$

Iterator for a Dynamic Array

We can print the content of a Dynamic Array in two ways:

1. Using an iterator (as presented above for a container)
2. Using the indexes of elements

Print with Iterator

Printing all the elements of an array using the iterator:

```
subalgorithm printDAWithIterator(da) is:
//pre: da is a DynamicArray
    //we create an iterator using the iterator method of DA
    iterator(da, it)
    while valid(it) execute
        //get the current element from the iterator
        elem ← getCurrent(it)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

What is the complexity of *printDAWithIterator*?

Print with indexes

Printing all the elements of an array using indexes:

subalgorithm printDAWithIndexes(da) **is:**
//pre: da is a Dynamic Array
    for i ← 1, size(da) **execute**
        elem ← getElement(da, i)
        print elem
    end-for
end-subalgorithm

What is the complexity of *printDAWithIndexes*?

Iterator for a Dynamic Array

In case of a Dynamic Array both printing algorithms have $\Theta(n)$ complexity

For other data structures/containers we need iterators because there are no indexes or using them violates the abstraction principle.

Dynamic Array - review

The elements of a dynamic array occupy a contiguous memory block.

Advantages:

- accessing any element in $\Theta(1)$
- adding at the end in $\Theta(1)$ amortized and removing from the end in $\Theta(1)$ (amortized)
- space efficiency

Disadvantages:

- adding and removing from the beginning of the array in $\Theta(n)$

Containers represented using dynamic arrays

Dynamic arrays are used for representing the following containers:

- ADT List
    - ArrayList in Java, Vector in C++ STL, Python lists ([])
- ADT Stack
    - Stack in Java
- Sorted List
- Bag
- Matrix

Arrays - Applications

Applications of arrays:

Machine Learning / Data Mining

- Representing the data instances as an array of characteristics

Speech processing

- Each speech signal is an array. The filters that are used to remove noise in a recording are also arrays.

Multi-player games

- To store the scores and sort them in descending order to clearly make out the rank of each player in the game

Representing strings

- Every string is an array of characters

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