# Lecture 6
# ADTs Lined Lists

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

# Abstract Data Type (ADT) vs. Data Structure

## Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Defines the input and outputs, not the implementations
- Can be expressed as an interface
- Examples: List, Map, Set

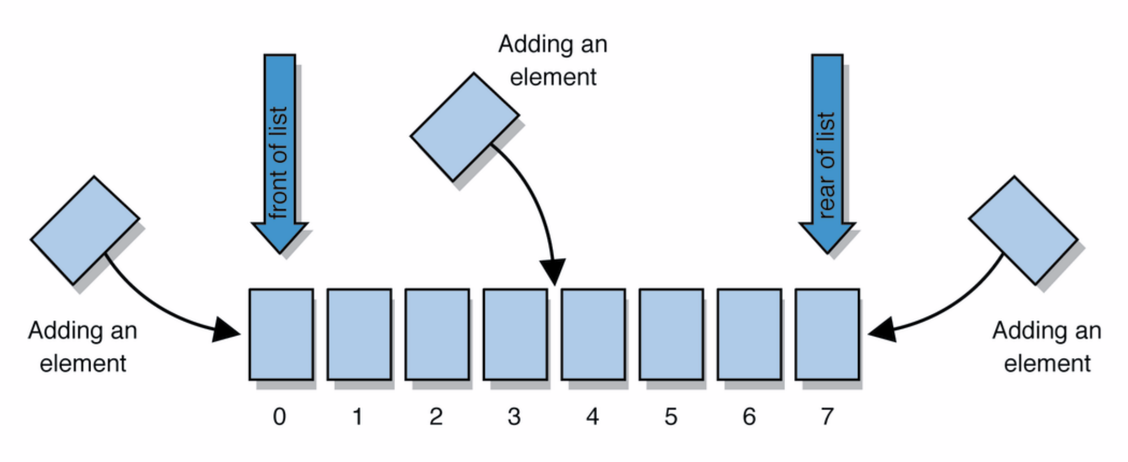
## Data Structure

- *A way of organizing and storing related data items*
- An object that implements the functionality of a specified ADT
- Describes how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

# Abstract Data Types (ADT): List Example

*Review:* List – a collection storing an ordered sequence of elements

- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- the ADT of a list can be implemented many ways through different Data Structures
    - in Java, a list can be represented as an ArrayList object



# *Review:* Interfaces

**interface**: a construct in Java that defines a set of methods that a class promises to implement

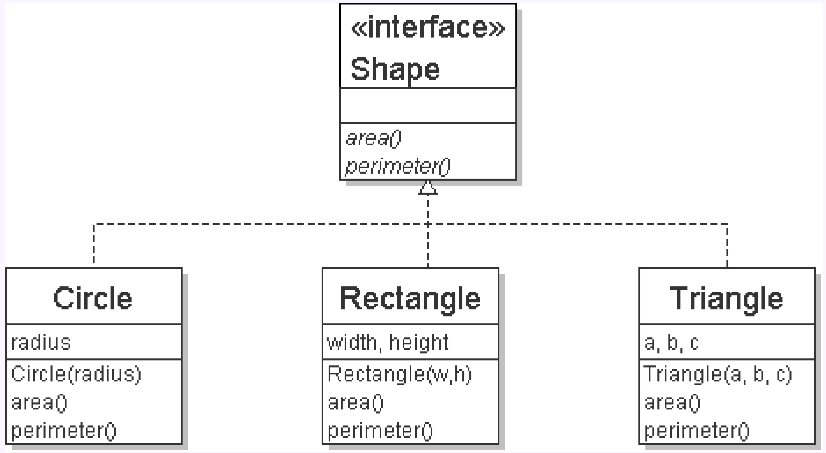
```
public interface name {
    public type name(type name, …, type name );
    public type name(type name, …, type name );
    …
}
```

- Interfaces give you an is-a relationship *without* code sharing.
  - A `Rectangle` object can be treated as a `Shape` but inherits no code.
- Analogous to non-programming idea of roles/certifications:
  - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know **how** to compute my area and perimeter."
  - "I'm ‘certified’ as a CPA accountant. This assures you I know **how** to do taxes, audits, and consulting."

## Example

```
// Describes features common to all
// shapes.

public interface Shape {
        public double area();
        public double perimeter();
}
```



«interface»
Shape

*area()*
*perimeter()*

Circle

radius

Circle(radius)
area()
perimeter()

Rectangle

width, height

Rectangle(w,h)
area()
perimeter()

Triangle

a, b, c

Triangle(a, b, c)
area()
perimeter()

# *Review:* Interfaces: List Example

**interface**: a construct in Java that defines a set of methods that a class promises to implement

In terms of ADTs, interfaces help us make sure that our implementations of that ADT are doing what they need to

For example, we could define an interface for the ADT `List<E>` and any class that implements it must have implementations for all of the defined methods

```
// Describes features common to all lists.

public interface List<E> {
    public E get(int index);
    public void set(E element, int index);
    public void append(E element);
    public E remove(int index);
    …

    // many more methods
}
```

# *Review:* Java Collections

Java provides some DS implementations of ADTs for you!

| ADTs | Data Structures |
|---|---|
| Lists | `List<Integer> a = new ArrayList<Integer>();` |
| Stacks | `Stack<Character> c = new Stack<Character>();` |
| Queues | `Queue<String> b = new LinkedList<String>();` |
| Maps | `Map<String, String> d = new TreeMap<String, String>();` |

But some data structures you made from scratch

Linked Lists - LinkedIntList was a collection of ListNode
Binary Search Trees – SearchTree was a collection of SearchTreeNodes

# ADTs and Data Structures: (Loose) Analogy

An ADT may be implemented with different data structures with different tradeoffs:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- …

Abstract Data Type (ADT)

| Mode of Transportation |
|---|
| Must be able to move |
| Must be able to be steered |

Data Structures

| Car | Airplane | Bike |
|---|---|---|
| Tires | Engines/wings | Wheels |
| Steering wheel | Control column | Handlebars |

# ADTs and Data Structures: List Example

## List - Abstract Data Type (ADT)

```
// Describes features common to all lists.

public interface List<E> {
    public E get(int index);
    public void set(E element, int index);
    public void append(E element);
    public E remove(int index);
    …
}
```

## ArrayIntList – Data Structure

```
public class ArrayIntList extends List<E>{
       private int[] list;
       private int size;

    public ArrayIntList(){
        //initialize fields
    }

    public int get(int index){
        return list[index];
    }

    public void set(E element, int index){
        list[index] = element;
    }
    …
}
```

# ADTs Examples

- List: an ordered sequence of elements
- Set: an unordered collection of elements
- Map: a collection of “keys” and associated “values”
- Stack: a sequence of elements that can only go in or out from one end
- Queue: a sequence of elements that go in one end and exit the other
- Priority Queue: a sequence of elements that is ordered by “priority”
- Graph: a collection of points/vertices and edges between points
- Disjoint Set: a collection of sets of elements with no overlap

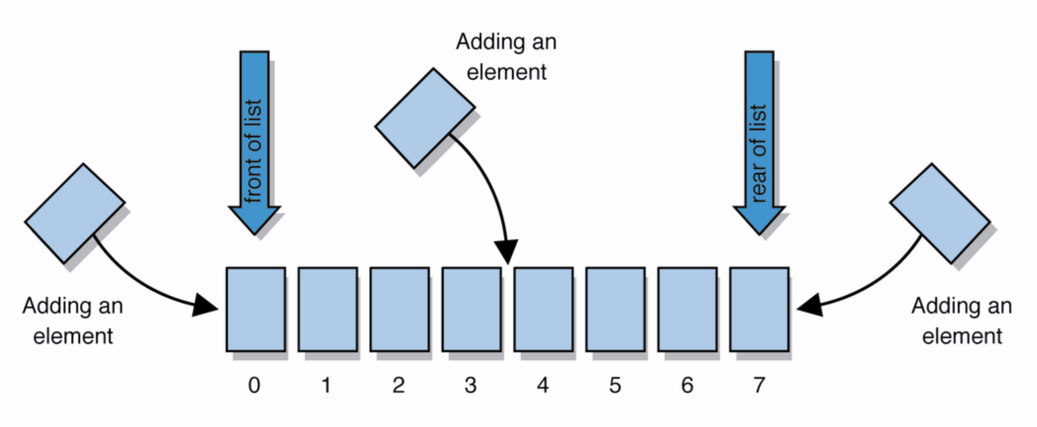
# Case Study: The List ADT

**list:** a collection storing an ordered sequence of elements

- Each item is accessible by an index
- A list has a size defined as the number of elements in the list

## Expected Behavior:

- **`get(index)`:** returns the item at the given index
- **`set(value, index)`:** sets the item at the given index to the given value
- **`append(value)`:** adds the given item to the end of the list
- **`insert(value, index)`:** insert the given item at the given index maintaining order
- **`delete(index)`:** removes the item at the given index maintaining order
- **`size()`:** returns the number of elements in the list



```
List<String> names = new ArrayList<>();
names.add("Anish");
names.add("Amanda");
names.add(0, "Brian");
```

# Case Study: List Implementations

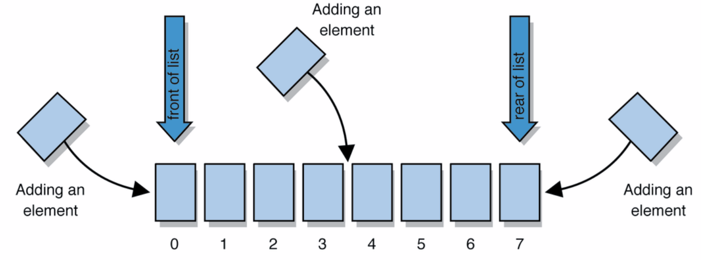
## List ADT

**state**

Set of ordered items
Count of items

**behavior**

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items



## ArrayList

uses an Array as underlying storage

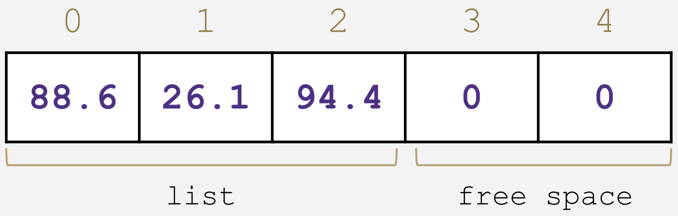
### ArrayList<E>

**state**

```
data[]
size
```

**behavior**

```
get return data[index]
set data[index] = value
append data[size] =
value, if out of space
grow data
insert shift values to
make hole at index,
data[index] = value, if
out of space grow data
delete shift following
values forward
size return size
```



## LinkedList

uses nodes as underlying storage

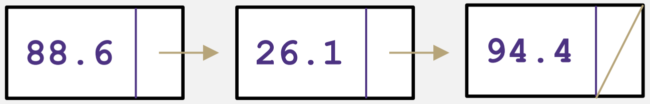
### LinkedList<E>

**state**

```
Node front
size
```

**behavior**

```
get loop until index,
return node's value
set loop until index,
update node's value
append create new node,
update next of last node
insert create new node,
loop until index, update
next fields
delete loop until index,
skip node
size return size
```



# Implementing Insert

## ArrayList<E>

`insert(element, index)` with shifting

`insert(10, 0)`

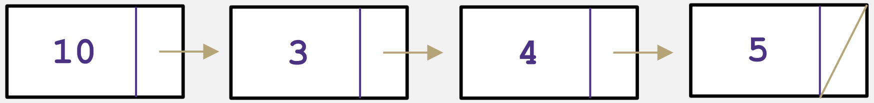
| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 10 | 3 | 4 | 5 |

`numberOfItems = 4`

## LinkedList<E>

`insert(element, index)` with shifting

`insert(10, 0)`



`numberOfItems = 4`

# Implementing Delete

## ArrayList<E>

`delete(index)` with shifting

`delete(0)`

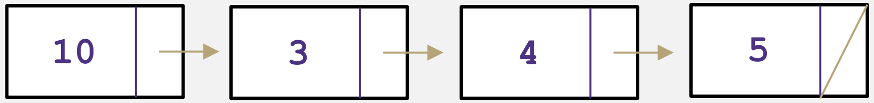
| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 3 | 4 | 5 | 5 |

`numberOfItems = 4`

## LinkedList<E>

`delete(index)` with shifting

`delete(0)`



`numberOfItems = 4`

# Implementing Append

## ArrayList<E>

`append(element)` with growth

`append(2)`

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 10 | 3 | 4 | 5 |

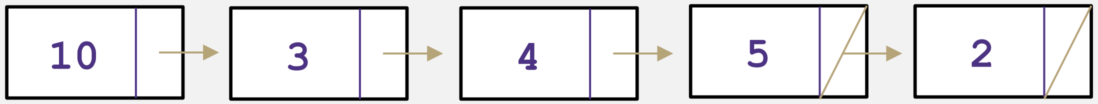
`numberOfItems` = 5

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 3 | 4 | 5 | 2 |  |  |  |

## LinkedList<E>

`append(element)` with growth

`append(2)`

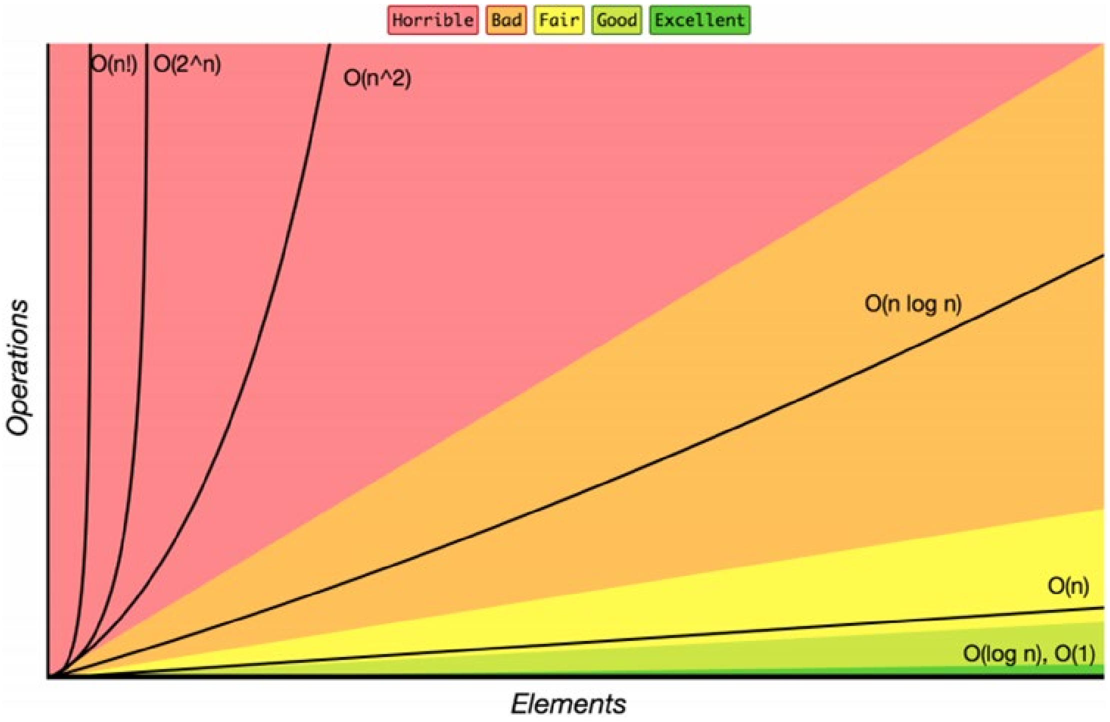


`numberOfItems` = 5

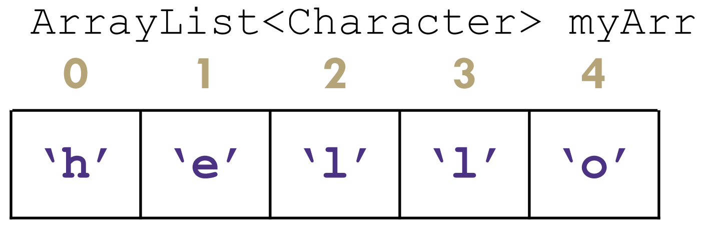
# *Review:* Complexity Class

**complexity class:** A category of algorithm efficiency based on the algorithm's relationship to the input size N.

| Complexity Class | Big-O | Runtime if you double N | Example Algorithm |
|---|---|---|---|
| constant | O(1) | unchanged | Accessing an index of an array |
| logarithmic | $O(\log_2 N)$ | increases slightly | Binary search |
| linear | O(N) | doubles | Looping over an array |
| log-linear | $O(N \log_2 N)$ | slightly more than doubles | Merge sort algorithm |
| quadratic | $O(N^2)$ | quadruples | Nested loops |
| ... | ... | ... | ... |
| exponential | $O(2^N)$ | multiplies drastically | Fibonacci with recursion |

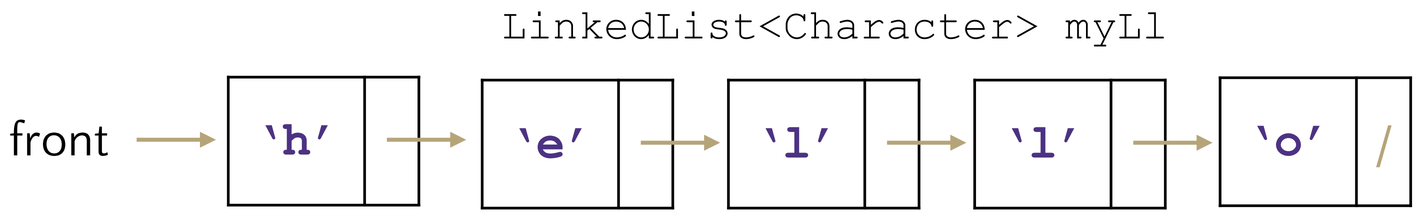


# List ADT tradeoffs


ArrayList<Character> myArr

0 1 2 3 4

'h' 'e' 'l' 'l' 'o'


LinkedList<Character> myLl

front

'h' 'e' 'l' 'l' 'o' /

Time needed to access Nth element:

- ArrayList: O(1) constant time
- LinkedList: O(N) linear time

Time needed to insert at Nth element (if the array is full!)

- ArrayList: O(N) linear time
- LinkedList: O(N) linear time

Amount of space used overall/across all elements

- ArrayList: sometimes wasted space at end of array
- LinkedList: compact, one node for each entry

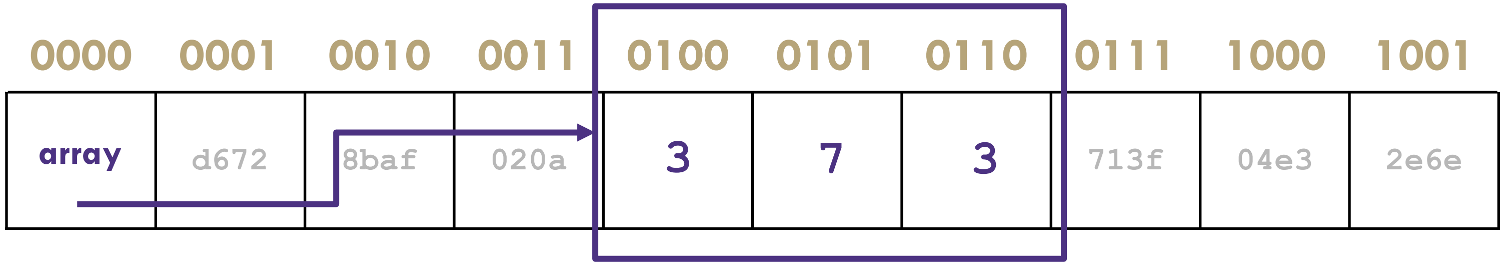
Amount of space used per element

- ArrayList: minimal, one element of array
- LinkedList: tiny bit extra, object with two fields

# A quick aside: Types of memory

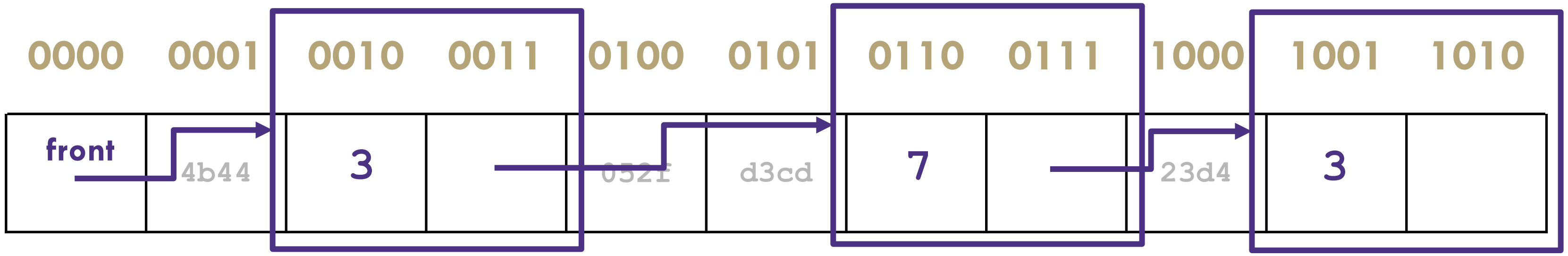
Arrays - **contiguous memory**: when the “new” keyword is used on an array the OS allocates a single, right-sized block of computer memory → Good cache locality

```
int[] array = new int[3];
array[0] = 3;
array[1] = 7;
array[2] = 3;
```



Nodes- **non-contiguous memory**: when the “new” keyword is used on a single node the OS allocates memory space for that object at the next available memory location → Poor cache locality

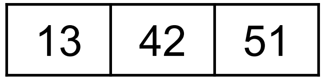
```
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```



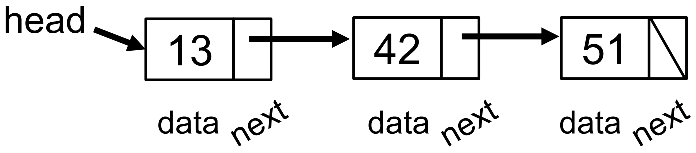
# ArrayList vs. Linkedlist

| ArrayList | LinkedList |
|---|---|
| ArrayList uses **dynamic array** to store data items. | LinkedList uses **doubly linked list** to store data items. |
| Manipulation with ArrayList is **slow**. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **fast**, since no bit shifting is required. |
| ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| ArrayList has **less memory overhead**, and each index only holds actual data. | LinkedList has **more memory overhead**, and each node holds both data and references/pointers |
| ArrayList has **good cache locality** due to contiguous memory allocation | LinkedList has **poor cache locality** due to non-contiguous memory allocation |

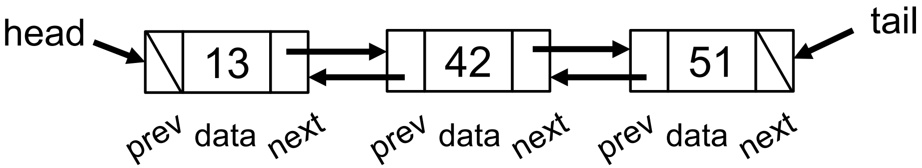
Array



Singly Linked List



Doubly Linked List



# References

Array vs. Single Linked List (In Terms of Representation)

- https://www.youtube.com/watch?v=R9PTBwOzceo&list=PLBlnK6fEyqRj9lld8sWIUNwlKfdUoPd1Y&index=31

Linked lists in 4 minutes

- https://www.youtube.com/watch?v=F8AbOfQwl1c