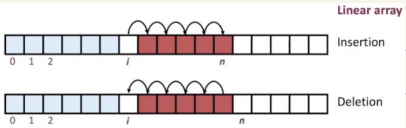
Video 1. Array-based lists, dynamic arrays

```
public interface List<E> {
  int size();
  boolean isEmpty();
  E get(int i) throws IndexOutOfBoundsException;
  E set(int i, E e) throws IndexOutOfBoundsException;
  void add(int i, E e) throws IndexOutOfBoundsException;
  E remove(int i) throws IndexOutOfBoundsException;
}
```

List ADT operations

size( )	returns the number of elements in the list	
isEmpty( )	return true if the list is empty, false otherwise	
get(*i*)	returns the element at index *i* or an error if *i* is not in {0, ..., size()-1}	error if *i* is not in range {0, ..., size( )-1}
set(*i*, *e*)	replaces the element at index *i* with *e* and returns the replaced element	
add(*i*, *e*)	adds *e* at index *i*, moving subsequent elements one index forward	({0, ..., size( )} for add)
remove(*i*)	removes and returns the element at index i, moving subsequent elements an index backward	

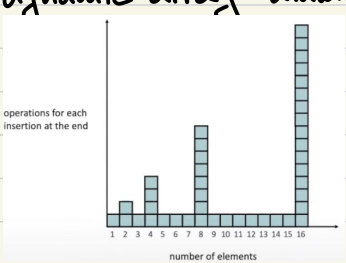


Lists can be implemented using arrays:
- **Linear array:** fixed-capacity array, k^{th} element is always stored at index $i = k-1$.
- **Circular array:** fixed-capacity array, list can wrap around the array

Array-based list operations		Time
Obtain size	size( )	O(1)
Check if it's empty	isEmpty( )	O(1)
Get element at index *i*	get(*i*)	O(1)
Set element *e* at index *i*	set(*i*, *e*)	O(1)
Add element *e* at index *i*	add(*i*, *e*)	O(*n*)
Remove element at index *i*	remove(*i*)	O(*n*)

Efficient operations per implementation		
Linear array		
add(*n*, *e*)	add last	O(1)
remove(*n*-1)	remove last	O(1)
Circular array		
add(0, *e*)	add first	O(1)
add(*n*, *e*)	add last	O(1)
remove(0)	remove first	O(1)
remove(*n*-1)	remove last	O(1)

dynamic array: ammortized analysis



fixed-capacity array, doubles in size

insertion: O(1) when array not full

time < O(n) when array is full

average:

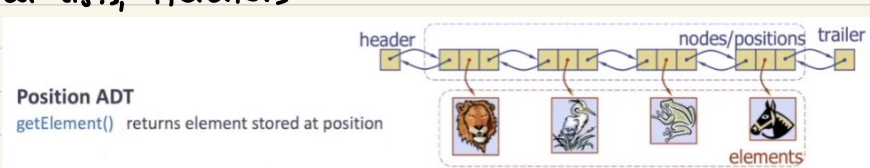
size-doubling $n = 2^i$ $i = \log_2 n$

$$\sum_{i=1}^{\log_2 n} 2^i = \frac{2 \times (1 - 2^{\log_2 n})}{1-2} = 2(n-1)$$

=> insertion time: O(n)

amortized (average) per instruction: O(1)

Video 2. Positional lists, iterators



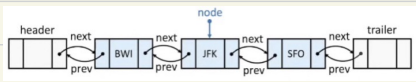
Positional List ADT

Accessor methods

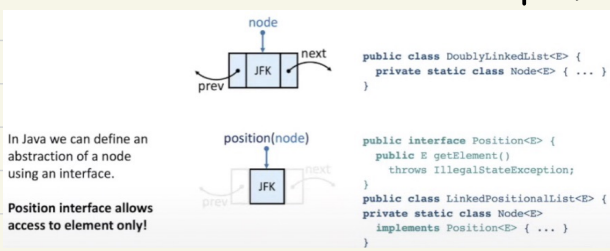
first( )	returns position of first element (**null** if empty)
last( )	returns position of last element (**null** if empty)
before(p)	returns position just before ***p*** (**null** if ***p*** is first)
after(p)	returns position just after ***p*** (**null** if ***p*** is last)
isEmpty( )	returns true if list is empty, false otherwise
size( )	returns the number of elements in list

Update methods

addFirst(*e*)	inserts *e* at the front, returns position
addLast(*e*)	inserts *e* at the back, returns position
addBefore(*p*, *e*)	inserts *e* before *p*
addAfter(*p*, *e*)	inserts *e* after *p*
set(*p*, *e*)	replaces element at position *p* with *e*
remove(*p*)	removes and returns element at *p*

DLL: 

(access to the whole list
position: reference to node



iterator - provides a single pass through a collection

Java.util.Iterator

hasNext( )	returns true if there's at least one additional element in the sequence
next( )	returns the next element in the sequence (or NoSuchElementException)
remove()	optional method, removes the element returned by the most recent call to next()

Java Iterable interface

iterator( )	returns an iterator of the elements in the collection

Snapshot iterator:
- maintains own private (deep) copy of the collection
- unaffected by changes to the primary collection
- O(n) space and O(n) time

Lazy iterator:
- directly traverses the primary data structure (shallow copy)
- affected by changes to the primary collection
- O(1) space and O(1) time