ALGORITHMS AND DATA STRUCTURES
LECTURE 3 - PHYSICAL DATA STRUCTURES

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

CONTENT

1. Array
2. ArrayList
3. ArrayList<T>
4. LinkedList
5. LinkedList<T>
6. Performance difference

ARRAY

Array is a data structure of related data items

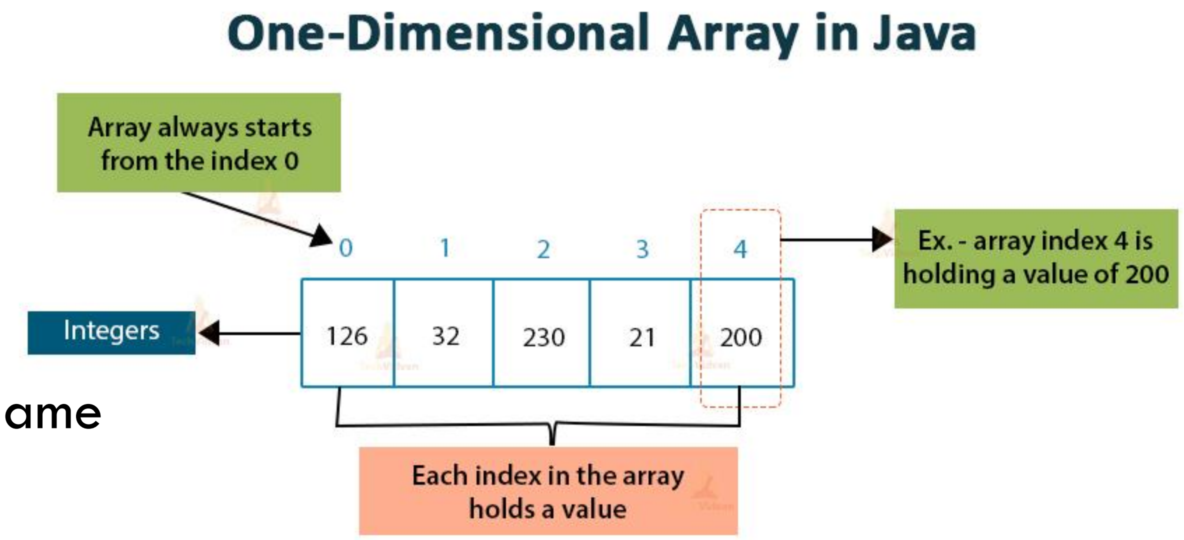
Static entity (**same size** throughout program)

A set of variables of the same type

Each element is referred to through a common name

Specific element is accessed using index

The set of data is stored in contiguous memory locations in index order



ARRAY: DISADVANTAGES

Array is fixed size data structure

It cannot increase by itself

Additional data requires an increase in size

Why not to create a class that could take care of it

The complexity of increasing in size is O(N)

Should it be required for each insert?

```
int[] arr = new int[] {1, 2, 3, 4, 5};

// Try to add 6
arr[5] = 6; // Out of range error
```

```
public static void main(String[] args) {

    int[] arr = new int[] {1, 2, 3, 4, 5};

    // Try to add 6
    arr = add(arr, x: 6);
    System.out.println(arr[5]);
}

public static int[] add(int[] arr, int x) {
    int[] arr2 = new int[arr.length + 1]; // Bigger array
    int i;
    for (i = 0; i < arr.length; i++) {
        arr2[i] = arr[i]; // Copy the content
    }
    arr2[i] = x; // Insertion
    return arr2;
}
```

ARRAYLIST

ArrayList is a variable length Collection class

A class that provides a better API for working with array

- Insertion
- Deletion
- Altering
- Searching
- Etc.

Increasing process does not happen on every insert, since we use capacity (*buffer*)

The capacity is increased according to some formula, which is completely chosen by the designer

```java
public class MyArrayList {
    private int[] array;
    private int size = 0;
    private int capacity = 5;

    public MyArrayList() { array = new int[capacity]; }

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

    public void add(int newItem) {
        if (size == capacity) {
            increaseBuffer();
        }
        array[size++] = newItem;
    }

    private void increaseBuffer() {
        capacity = (int) (1.5 * capacity);
        int[] array2 = new int[capacity];
        for (int i = 0; i < size; i++) {
            array2[i] = array[i];
        }

        array = array2;
    }

    public int getSize() { return size; }
}
```

ARRAYLIST<T>

ArrayList can also be of generic type <T>

In this case, array references must be of more abstract type **Object** *(language-specific)*

It is needed to cast data item to type T for retrieving

The object creation for MyArrayList<T> is as follows:

```
MyArrayList<Integer> list = new MyArrayList<>();
```

The type for which you are creating an ArrayList must be a **reference type**

Since, it is not possible to handle primitives when using generics *(language-specific)*

```
public class MyArrayList<T> {
    private Object[] array;
    private int size = 0;
    private int capacity = 5;

    public MyArrayList() {
        array = new Object[capacity];
    }

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

    public void add(T newItem) {
        if (size == capacity) {
            increaseBuffer();
        }
        array[size++] = newItem;
    }

    private void increaseBuffer() {
        capacity = (int) (1.5 * capacity);
        Object[] array2 = new Object[capacity];
        for (int i = 0; i < size; i++) {
            array2[i] = array[i];
        }

        array = array2;
    }

    public int getSize() { return size; }
}
```

ARRAYLIST<T>:ITERATORS

An Iterator is an object that can be used to loop through collections

MyArrayList can also implement an **Iterable<T>** interface

It needs iterator() method to be implemented

That method should return an instance of **Iterator**

Create a private class which implements **Iterator<T>**

Implement hasNext() and next() methods

See next slide for results…

```java
public class MyArrayList<T> implements Iterable<T> {
    // ....

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

    // ....

    public int getSize() { return size; }

    @Override
    public Iterator<T> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        int cursor = 0;

        @Override
        public boolean hasNext() {
            return cursor != getSize();
        }

        @Override
        public T next() {
            T nextItem = get(cursor);
            cursor++;
            return nextItem;
        }
    }
}
```

ARRAYLIST<T>:ITERATORS

```java
public static void main(String[] args) {
    MyArrayList<Integer> list = new MyArrayList<>();

    for (int i = 0; i < 30; i++) {
        list.add(i);
    }

    Iterator<Integer> it = list.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

```java
public static void main(String[] args) {
    MyArrayList<Integer> list = new MyArrayList<>();

    for (int i = 0; i < 30; i++) {
        list.add(i);
    }

    for (Integer num : list) {
        System.out.println(num);
    }
}
```

```java
public class MyArrayList<T> implements Iterable<T> {
    // ....

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

    // ....

    public int getSize() { return size; }

    @Override
    public Iterator<T> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        int cursor = 0;

        @Override
        public boolean hasNext() {
            return cursor != getSize();
        }

        @Override
        public T next() {
            T nextItem = get(cursor);
            cursor++;
            return nextItem;
        }
    }
}
```

ARRAYLIST<T>:DISADVANTAGES

Imagine that you need to add a new item at position 0

Its complexity is O(N), because we need to shift all elements to the right

The same happens for removing an item at position 0 (shifting all elements to the left)

Hint: There is no need to reduce the capacity (buffer)

Any other solution?

```
public void addForward(T newItem) {
    if (size == capacity) {
        increaseBuffer();
    }

    for (int i = size; i > 0; i--) {
        array[i] = array[i - 1]; // moving right
    }
    size++;
    array[0] = newItem; // Insertion
}
```

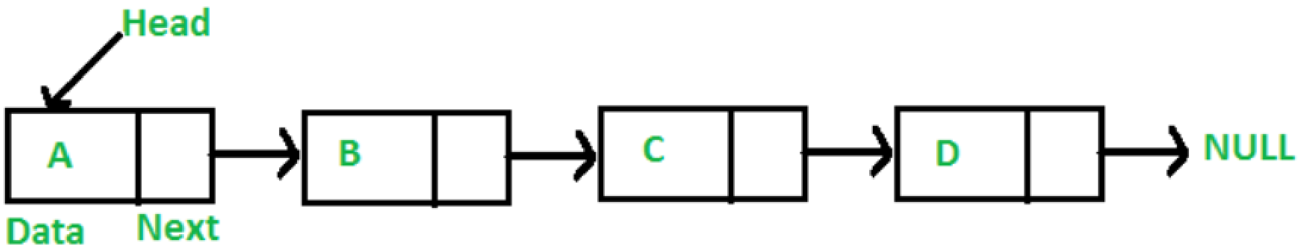
```
public void removeLast() {
    size--;
}

public void removeFirst() {
    for (int i = 0; i < size - 1; i++) {
        array[i] = array[i + 1]; // moving right
    }

    size--;
}
```

LINKEDLIST

A linked list is a series of connected **nodes**

Each node contains at least

- A piece of data (any type)
- Reference (or pointer) to the next node in the list



A node is an object that must hold a value and some references to *other nodes*

Depending on that, the LinkedList has 3 types

- Singly-linked (each node points to the next node)
- Doubly-linked (each node points to the next and previous nodes)
- Circular-linked (Last node points to the first)

```
class MyNode {
    int data;
    MyNode next;
    // MyNode prev;

    MyNode(int data) {
        this.data = data;
        next = null;
    }
}
```

LINKEDLIST

A linked list must have its **head** (entry point) and sometimes **tail** (last element) for better performance

The head must refer to NULL when the linked list is just created

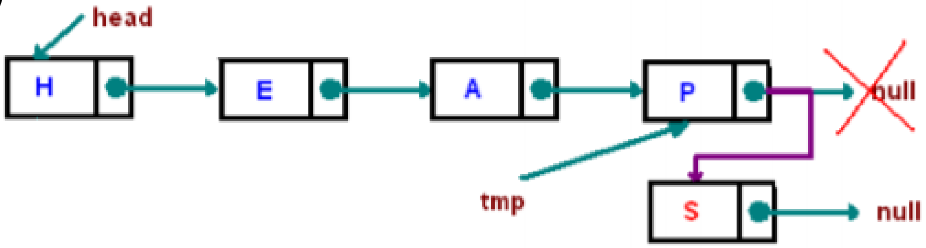
In order to iterate to the next element, we use:

```
node = node.next;
```

Adding a new node to the end (when **tail** is not stored)

Adding a new node when tail is stored

- (complexity is O(1))




```
public void add(int newItem) {
    MyNode newNode = new MyNode(newItem);
    if (head == null) {
        head = newNode;
    } else {
        MyNode current = head;
        while (current.next != null) {
            current = current.next;
        }

        current.next = newNode;
    }

    size++;
}
```

```
public void add(int newItem) {
    MyNode newNode = new MyNode(newItem);
    if (head == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }

    size++;
}
```

```java
public class MyLinkedList {
    private MyNode head; // entry point
    private MyNode tail; // last node
    private int size;

    public MyLinkedList() {
//          head = null; --> these are
//          size = 0;    --> redundant
    }

    public void add(int newItem) {
        MyNode newNode = new MyNode(newItem);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
    }

    public int get(int index) {
        MyNode current = head;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.data;
    }

    private static class MyNode {
        int data;
        MyNode next;
        // MyNode prev; // for doubly-linked

        MyNode(int data) {
            this.data = data;
            // next = null; // redundant
        }
    }
}
```

```java
public static void main(String[] args) {
    MyLinkedList list = new MyLinkedList();

    for (int i = 0; i < 30; i++) {
        list.add(i);
    }

    for (int i = 0; i < 30; i++) {
        System.out.print(list.get(i) + " ");
    }
}
```

Main

```
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
Process finished with exit code 0
```

LINKEDLIST<T>

LinkedList can also be of generic type <T>

In this case, each node`s data item must be of type **T**

Then the Linked list can be created for any type:

```java
public static void main(String[] args) {
    MyLinkedList<String> list = new MyLinkedList<>();

    list.add("Almaty");
    list.add("is the best");
    list.add("city!");

    for (int i = 0; i < list.getSize(); i++) {
        System.out.print(list.get(i) + " ");
    }
}
```

Main ×

```
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
Almaty is the best city!
Process finished with exit code 0
```

```java
public class MyLinkedList<T> {
    private MyNode<T> head; // entry point
    private MyNode<T> tail; // last node
    private int size;

    public MyLinkedList() {
//          head = null; --> these are
//          size = 0;    --> redundant
    }

    public void add(T newItem) {
        MyNode<T> newNode = new MyNode<>(newItem);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
    }

    public T get(int index) {
        MyNode<T> current = head;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.data;
    }

    private static class MyNode<E> {
        E data;
        MyNode<E> next;
        // MyNode prev; // for doubly-linked

        MyNode(E data) {
            this.data = data;
            // next = null; // redundant
        }
    }
}
```

LINKEDLIST<T>:COMPLEXITIES

Adding an element to the end – O(N)

```
public void add(T newItem) {
    MyNode<T> newNode = new MyNode<>(newItem);
    if (head == null) {
        head = newNode;
    } else {
        MyNode<T> current = head;
        while (current.next != null)
            current = current.next;

        current.next = newNode;
    }
    size++;
}
```

Adding an element to the end (with tail) – O(1)

```
public void add(T newItem) {
    MyNode<T> newNode = new MyNode<>(newItem);
    if (head == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
    size++;
}
```

Adding an element at the beginning – O(1)

```
public void addForward(T newItem) {
    if (head == null) {
        add(newItem);
        return;
    }
    MyNode<T> newNode = new MyNode<>(newItem);
    newNode.next = head;
    head = newNode;
    size++;
}
```

LINKEDLIST<T>:COMPLEXITIES

Removing the last element – O(N)

```
public void removeLast() {
    if (head == tail) {
        head = tail = null;
    } else {
        MyNode<T> current = head;
        while (current.next != tail) {
            current = current.next;
        }
        tail = current;
        tail.next = null;
    }
    size--;
}
```

Removing the last element (with previous) – O(1)

```
public void removeLast() {
    if (head == tail) {
        head = tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }
    size--;
}
```

Removing the first element – O(1)

```
public void removeFirst() {
    if (head == tail) {
        head = tail = null;
    } else {
        head = head.next;
    }
    size--;
}
```

LINKEDLIST<T>

Missed something?

- It is better to check before the action
- It is better to **return** an element that is removed

get(int index) method`s complexity:

- O(1) for ArrayList
- O(N) for LinkedList

```
public T removeFirst() {
    if (head == null)
        throw new IndexOutOfBoundsException("Linked list is empty!");

    T removedElement = head.data;

    if (head == tail) {
        head = tail = null;
    } else {
        head = head.next;
    }
    size--;

    return removedElement;
}
```

LinkedList is a linear collection of data elements whose order is not given by their physical placement in memory

LinkedList is our choice when the access to the element at specific position is rarely used

*The performance enhancement is revealed during the frequent insertion, deletion and retrieval of both the **first** and the **last** elements ONLY*

LINKEDLIST<T>:ITERATORS

MyLinkedList can also implement an **Iterable<T>** interface

It needs iterator() method to be implemented

That method should return an instance of **Iterator**

Create a private class which implements **Iterator<T>**

Implement hasNext() and next() methods

See next slide for results…

```java
@Override
public Iterator<T> iterator() {
    return new MyIterator();
}

private class MyIterator implements Iterator<T> {
    MyNode<T> cursor = head;

    @Override
    public boolean hasNext() {
        return cursor != null;
    }

    @Override
    public T next() {
        T nextItem = cursor.data;
        cursor = cursor.next;
        return nextItem;
    }
}
```

LINKEDLIST<T>:ITERATORS

```java
public static void main(String[] args) {
    MyLinkedList<Integer> list = new MyLinkedList<>();
    int n = scanner.nextInt();
    for (int i = 0; i < n; i++) {
        list.add(i);
    }

    for (int i = 0; i < list.getSize(); i++) {
        System.out.print(list.get(i) + " ");
    }
}
```

$O(N^2)$

$O(N)$

```java
public static void main(String[] args) {
    MyLinkedList<Integer> list = new MyLinkedList<>();
    int n = scanner.nextInt();
    for (int i = 0; i < n; i++) {
        list.add(i);
    }

    for (Integer item : list) {
        System.out.print(item + " ");
    }
}
```

$O(N)$

```
Main
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
10
0 1 2 3 4 5 6 7 8 9
Process finished with exit code 0
```

```java
public class MyLinkedList<T> implements Iterable<T>{
    private MyNode<T> head; // entry point
    private MyNode<T> tail; // last node
    private int size;

    // ...

    @Override
    public Iterator<T> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        MyNode<T> cursor = head;

        @Override
        public boolean hasNext() {
            return cursor != null;
        }

        @Override
        public T next() {
            T nextItem = cursor.data;
            cursor = cursor.next;
            return nextItem;
        }
    }

    private static class MyNode<E> {...}
}
```

PERFORMANCE DIFFERENCE

ArrayList`s advantages over LinkedList:

- Accessing an element at specific position
- Less memory usage
- ArrayList is better for storing and accessing data

LinkedList`s advantages over ArrayList:

- Increasing the size
- Adding and removing an element comparatively at the beginning (also at the end when it is doubly-linked)
- LinkedList is better for manipulating data

```java
MyArrayList<Integer> arrayList = new MyArrayList<>();
MyLinkedList<Integer> linkedList = new MyLinkedList<>();
int n = 100000;
for (int i = 0; i < n; i++) {
    arrayList.add(i);
    linkedList.add(i);
}

long time1 = System.nanoTime();
arrayList.get(50000);
long time2 = System.nanoTime();
linkedList.get(50000);
long time3 = System.nanoTime();

System.out.println("get(50000) : ArrayList: " + (time2 - time1));
System.out.println("get(50000) : LinkedList: " + (time3 - time2));

time1 = System.nanoTime();
arrayList.removeFirst();
time2 = System.nanoTime();
linkedList.removeFirst();
time3 = System.nanoTime();

System.out.println("removeFirst() : ArrayList: " + (time2 - time1));
System.out.println("removeFirst() : LinkedList: " + (time3 - time2));
```

```
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
get(50000) : ArrayList: 300
get(50000) : LinkedList: 244700
removeFirst() : ArrayList: 2114000
removeFirst() : LinkedList: 13100

Process finished with exit code 0
```

FURTHER MODIFICATIONS (HOMEWORK)

MyArrayList: public void add(T newItem, int index)

MyArrayList: public int find(T keyItem) – returns index or -1

MyArrayList: public T remove(int index) – returns removed element

MyArrayList: public void reverse() – reverses the ArrayList (1,2,3,4 becomes 4,3,2,1)

MyLinkedList: public void add(T newItem, int index)

MyLinkedList: public int find(T keyItem) – returns index or -1

MyLinkedList: public T remove(int index) – returns removed element

MyLinkedList: public void reverse() – reverses the LinkedList

Implement everything (including example methods from this lecture) for MyDoublyLinkedList<T>

LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 1.3

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 2 – Arrays and Linked Lists

GOOD LUCK!