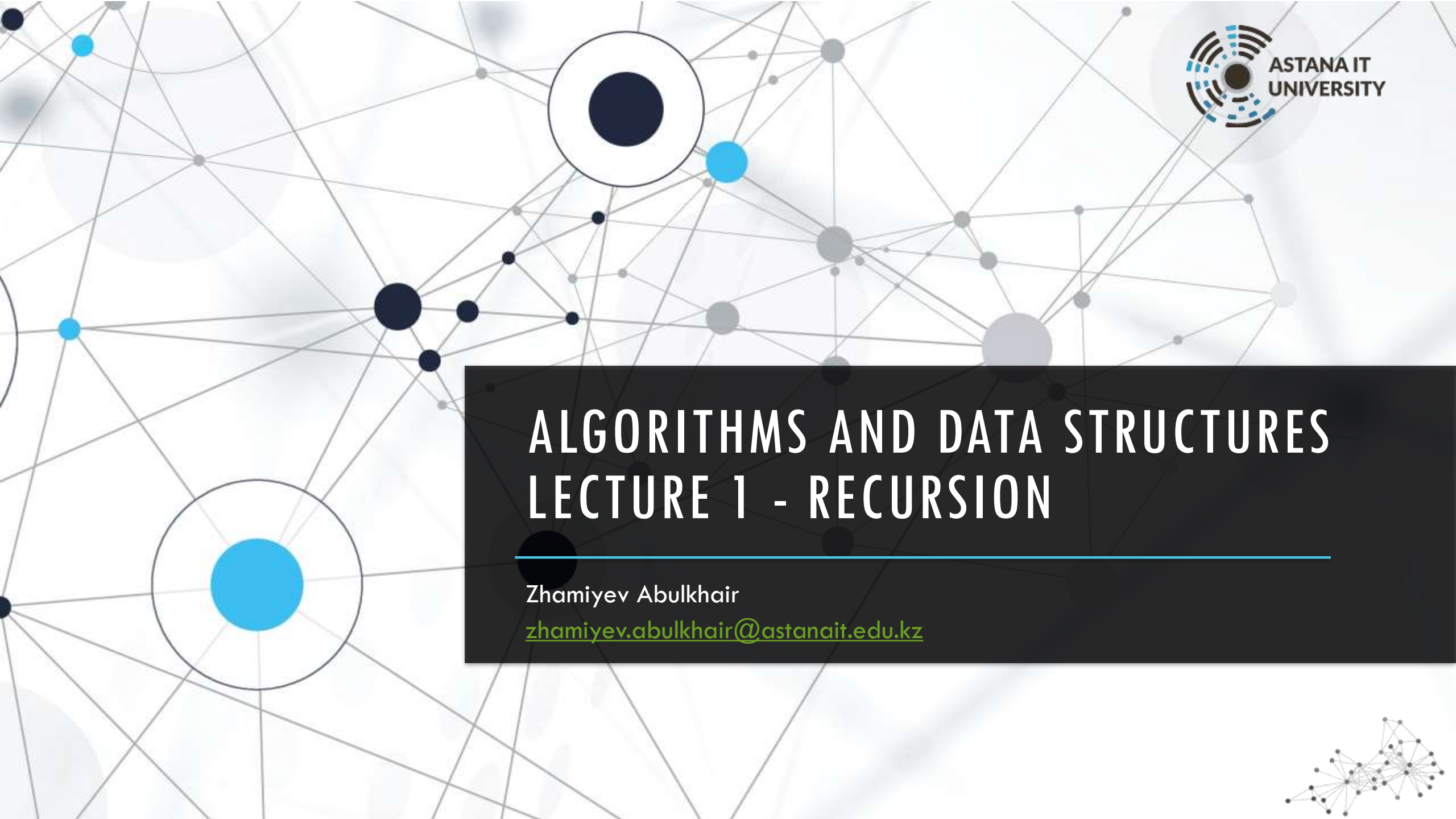


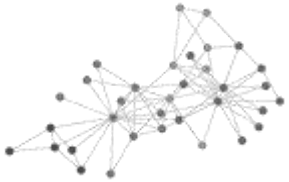
ALGORITHMS AND DATA STRUCTURES LECTURE 1 - RECURSION

Zhamiyev Abulkhair
zhamiyev.abulkhair@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Recursion Overview
2. Simple example
3. How it works?
4. Function call and Stack
5. Iteration vs Recursion
6. How to create a recursive algorithm?
7. Fibonacci solution



RECURSION OVERVIEW

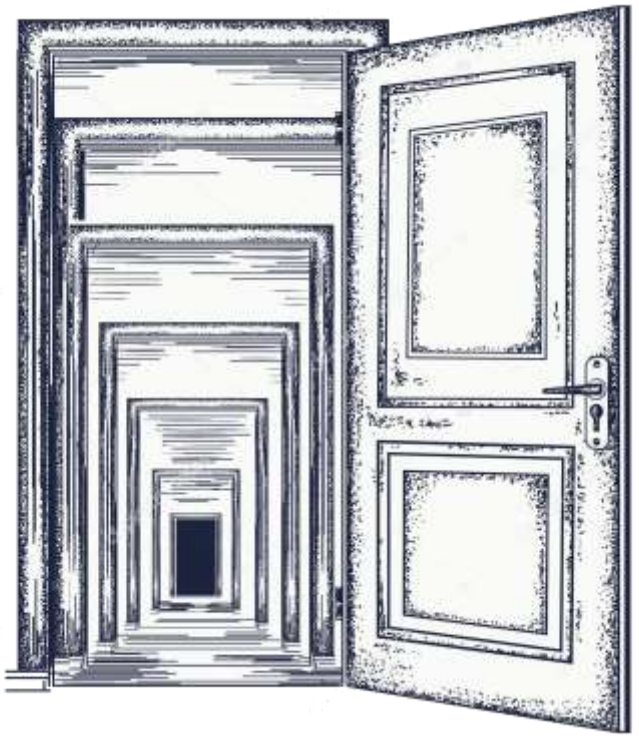
Recursion is the process of repeating items in a self-similar way

A way to design solutions by Divide-and-Conquer

- Reduce a problem to simpler versions of the same problem

A programming technique where a function calls itself

- Must have at least 1 **base case**
- **Base case** means that there exist one or more inputs for which the function produces a result trivially (without recurring)
- Must solve the same problem on some other input with the goal of simplifying the larger problem input



ASTANA IT UNIVERSITY

SIMPLE EXAMPLE

N! = 1•2•3•4•5•6•7•…•N fact(N)

N! = 1•2•3•4•5•6•7•…•(N-1) •N fact(N-1) * N

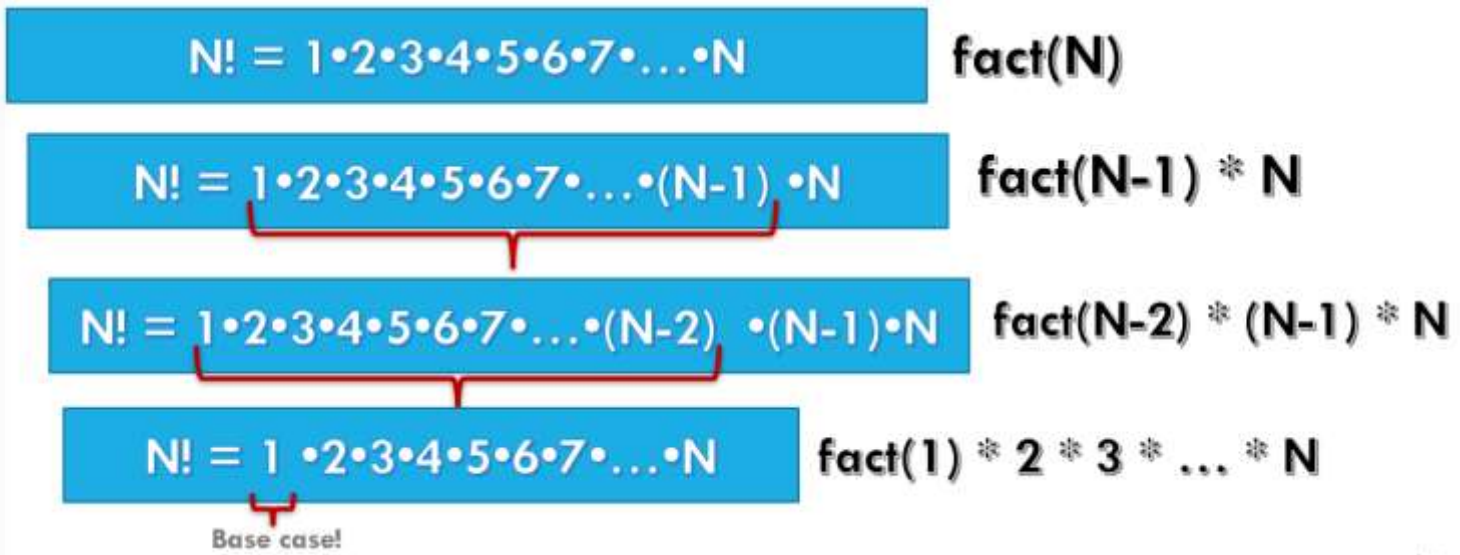
N! = 1•2•3•4•5•6•7•…•(N-2) •(N-1)•N fact(N-2) * (N-1) * N

N! = 1 •2•3•4•5•6•7•…•N fact(1) * 2 * 3 * … * N

Base case!

SIMPLE EXAMPLE

```java
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt();

    int result = factorial(n);

    System.out.println(result);
}
```



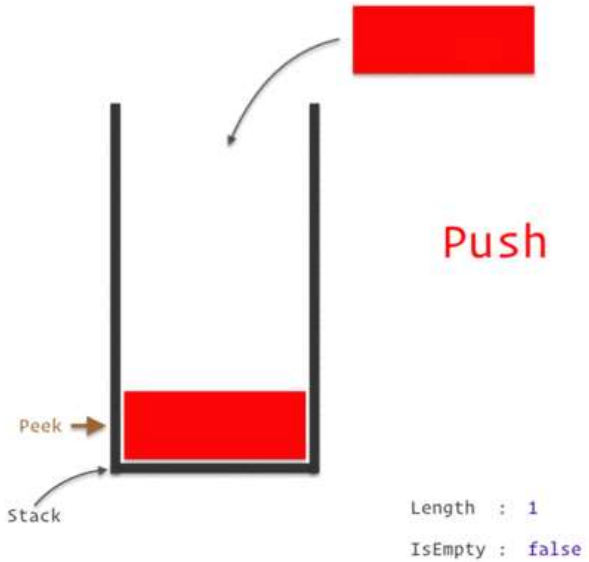
HOW IT WORKS?

Recursion is no different than a function call

Every function call creates a new frame (block) inside the stack

The system keeps track of the sequence of method calls that have been started but not finished yet (active calls)

- order matters



Recursion pitfalls

- miss base-case (infinite recursion, stack overflow)
- no convergence (solve recursively a problem that is not simpler than the original one)

FUNCTION CALL AND STACK

When you run a program, the computer creates a **stack** for you

Each time you invoke a method, the method is placed to the stack

A stack is a **last-in/first-out** memory structure. The first item referenced or removed from a stack is always the last item entered into the stack

If some function call has produced an excessively long chain of recursive calls, it can lead to **stack overflow**

```
int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;
}

void main() {
    int result = factorial( N: 3);

    System.out.println(result);
}
```

ITERATION VS RECURSION

Iteration

- Uses repetition structures (for, while or do…while)
- Repetition through explicitly use of repetition structure
- Terminates when loop-continuation condition fails
- Controls repetition by using a counter

```
public static int factorial(int N) {
    int product = 1;
    for (int i = 1; i <= N; i++) {
        product *= i;
    }

    return product;
}
```

Recursion

- Uses selection structures (if, if…else or switch)
- Repetition through repeated method calls
- Terminates when base case is satisfied
- Controls repetition by dividing problem into simpler one

```
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}
```

ASTANA IT UNIVERSITY

ITERATION VS RECURSION

Repetition

- Iteration: explicit loop
- Recursion: repeated function calls

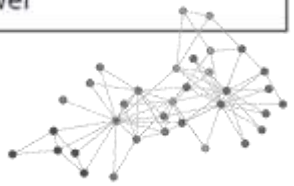
Termination

- Iteration: loop condition fails
- Recursion: base case recognized

Both can have infinite loops

Balance between performance (iteration) and good software engineering (recursion)

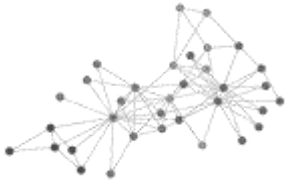
Criteria	Iteration	Recursion
Mode of implementation	Implemented using loops	Function calls itself
State	Defined by the control variable's value	Defined by the parameter values stored in stack
Progression	The value of control variable moves towards the value in condition	The function state converges towards the base case
Termination	Loop ends when control variable's value satisfies the condition	Recursion ends when base case becomes true
Code Size	Iterative code tends to be bigger in size	Recursion decreases the size of code
No Termination State	Infinite Loops uses CPU Cycles	Infinite Recursion may cause Stack Overflow error or it might crash the system
Execution	Execution is faster	Execution is slower



ASTANA IT UNIVERSITY

HOW TO CREATE A RECURSIVE ALGORITHM?

1. Think about a problem at a high level of abstraction
2. Figure out the **base case** for the program
3. Redefine the answer in terms of a simpler sub-problem
4. Combine the results in the formulation of the answer



FIBONACCI SOLUTION

Fibonacci sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …
- Each element is the sum of previous two
- Starts from 0 and 1

Task: Find the Fibonacci number at the given position

Example:

- 3rd element is 5
- 6th element is 8

Solution:

fib(n) = fib(n-2) + fib(n-1)

fib(0) = 0 and fib(1) = 1 // this is a base case

```
public static int fib(int n) {
    if (n <= 1) return n; // base case
    // no need to write "else", since the
    // previous one will return
    return fib( n: n-2) + fib( n: n-1);
}
```

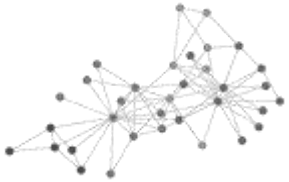
LITERATURE

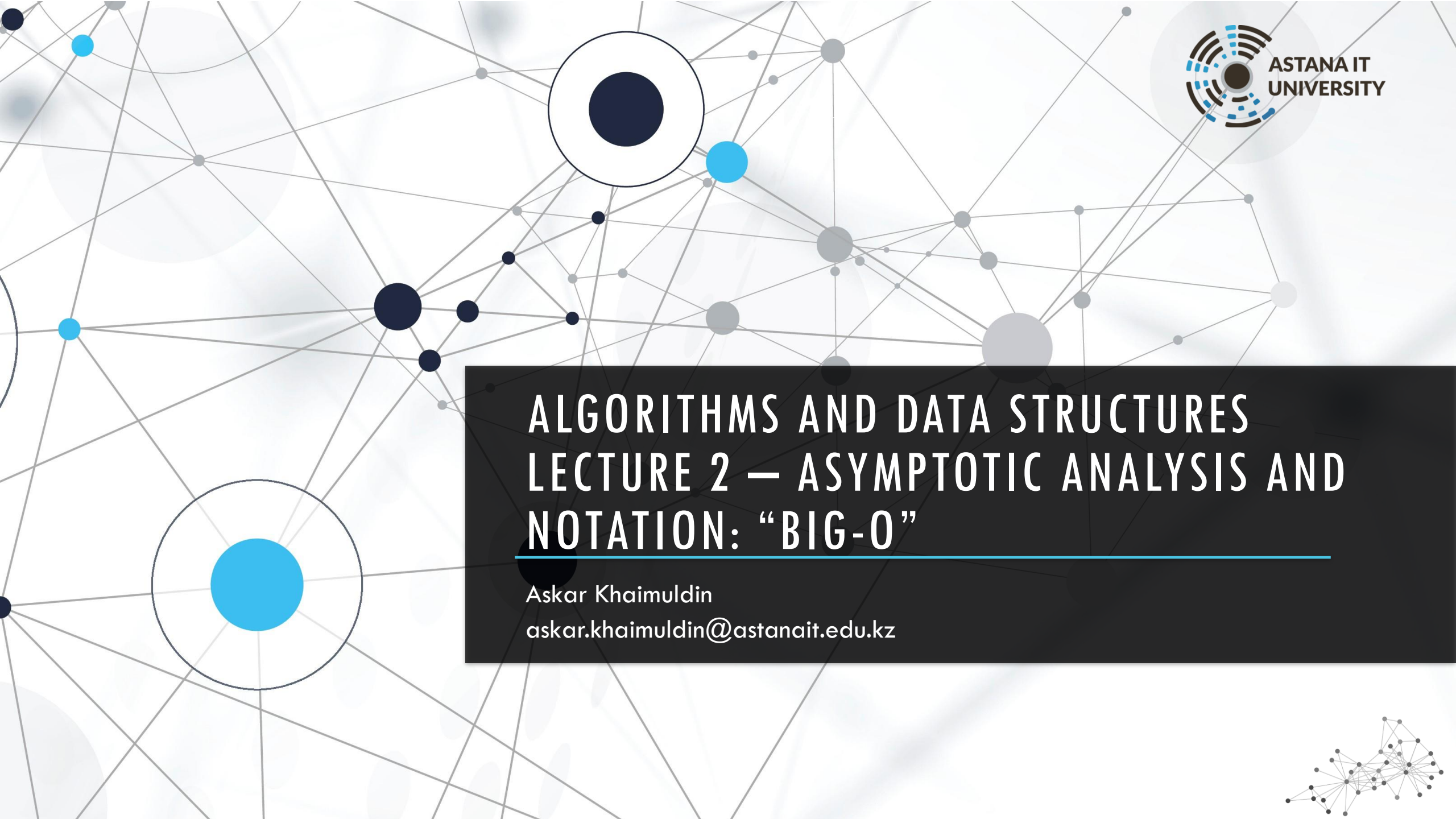
Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 1.1

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 3





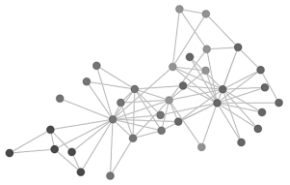
ALGORITHMS AND DATA STRUCTURES LECTURE 2 — ASYMPTOTIC ANALYSIS AND NOTATION: "BIG-O"

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Algorithm efficiency
2. Order of growth
3. Big-O notation
4. Analyzing the running time of a program (Example)
5. Complexity classes
6. Constant complexity
7. Logarithmic complexity
8. Linear complexity
9. Exponential complexity



ALGORITHM EFFICIENCY

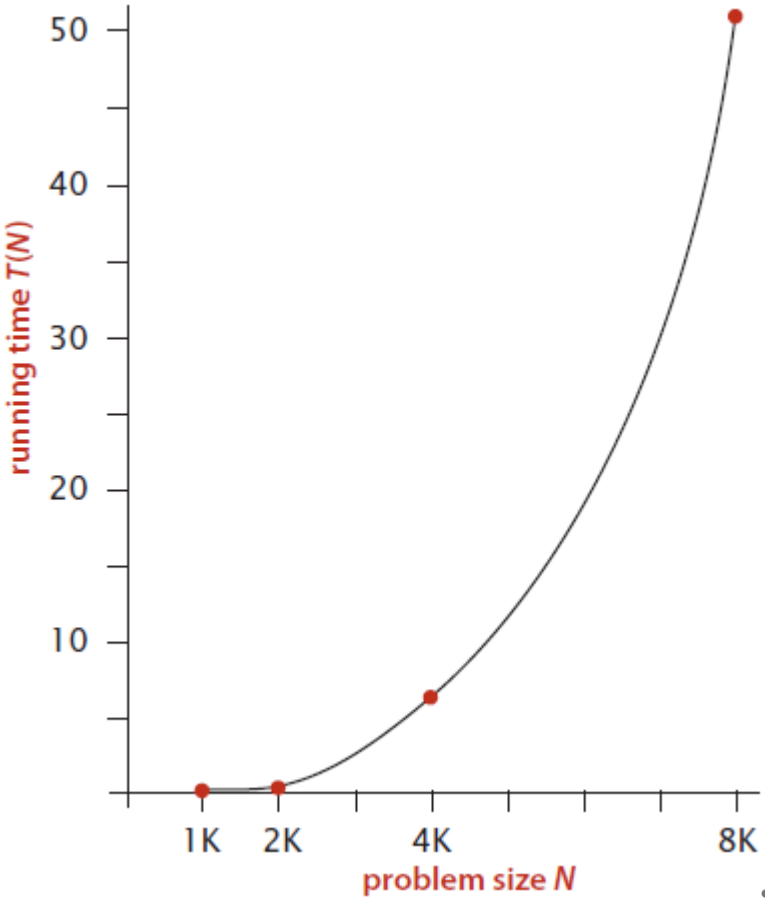
Computers are getting faster

- Does the efficiency matter?
- How about a very large dataset?

A program can be implemented in several ways

How to measure the efficiency?

- Using timer
- Count the number of operations
- **Order of growth**



ALGORITHM EFFICIENCY : USING TIMER

Timers are inconsistent

- Time varies for different inputs but cannot express a relationship between inputs and time
- Can be significantly different in various cases (because of CPU, memory, GC, operating system, network, etc.)

Our first challenge is to determine how to make quantitative measurements of the running time of our program using random numbers for input

Hint: Math.random()

```
long time1 = System.currentTimeMillis();
int triplesNum = countZeroTriples(arr);
long time2 = System.currentTimeMillis();

System.out.println(time2 - time1 + " ms");
```

5000	1000	100
30990 ms	256 ms	3 ms

```
// Calculate how many triples sum
// to exactly zero
public static int countZeroTriples(int[] arr) {
    int n = arr.length, count = 0;

    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            for (int k = j+1; k < n; k++)
                if (arr[i] + arr[j] + arr[k] == 0)
                    count++;
    return count;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt(); // number of elements

    int[] arr = new int[n];

    for (int i = 0; i < arr.length; i++) {
        arr[i] = sc.nextInt();
    }

    System.out.println(countZeroTriples(arr));
}
```

ALGORITHM EFFICIENCY : COUNT OPERATIONS

Assume that any of comparison, addition, value setting and retrieving, etc. is just 1 operation

How many of those operations do I use in my algorithm?

This could be used to give us a sense of efficiency

2 + 2 + (n+1) + n + n + 2n and 1 more for “return”

= **5n + 6 (worst case)**

The worst-case scenario: all elements are zeros

```
public static int countZeros(int[] arr) {
    int count = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == 0) {
            count++;
        }
    }

    return count;
}
```

operation	Sample cost in ns	frequency
variable declaration	2/5	2
assignment statement	1/5	2
less than compare	1/5	$n+1$
equal to compare	1/10	n
array access	10	n
increment	1/10	n to $2n$

ORDER OF GROWTH

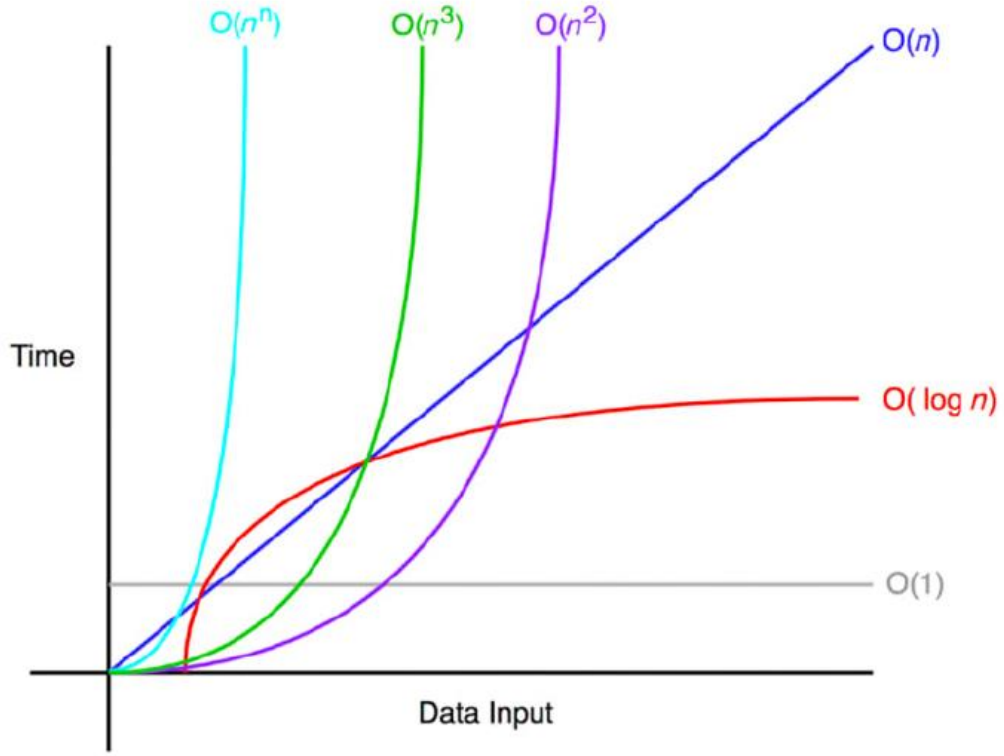
If f(n) ~ Cg(n) for some constant C > 0, then the order of growth of f(n) is g(n)

- Ignores leading coefficient
- Ignores lower-order terms
- Focuses on dominant terms

Examples:

1) $2n^3 + 5n^2 + 0.15n + 7 \approx n^3$

2) $2n^3 + 3^n \approx 3^n$



*Big-O notation is a mathematical notation that describes the **limiting behavior** of a **function** when the **argument** tends towards a particular value or infinity.*

BIG-O NOTATION

Law of addition: $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

- Used with sequential statements

$$O(n) + O(n^2) = O(n + n^2) = O(n^2)$$

```
for (int number : arr) {
    System.out.println(number);
}

int N = arr.length;
for (int i = 0; i < N * N; i++) {
    System.out.println(i);
}
```

$O(n)$

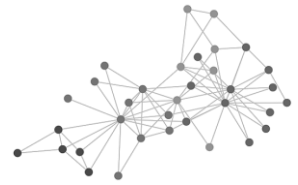
$O(n^2)$

Law of multiplication $O(f(n)) * O(g(n)) = O(f(n) * g(n))$

- Used with nested statements/loops or recursion

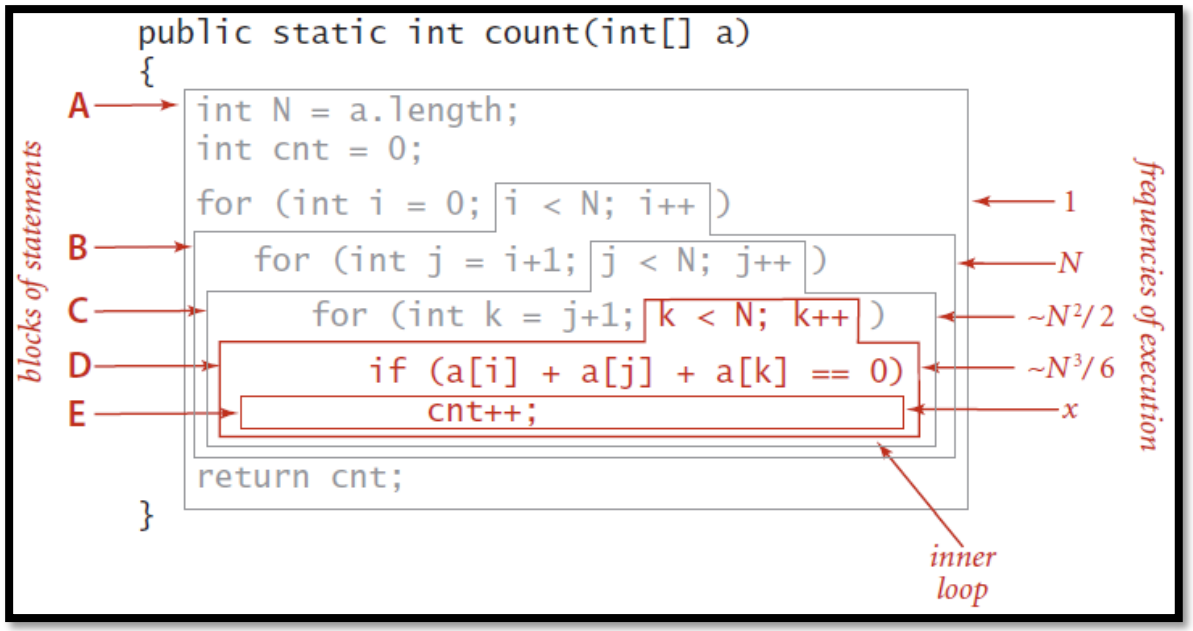
$$O(n) * O(n) = O(n^2)$$

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        System.out.println( i + j);
    }
}
```



ANALYZING THE RUNNING TIME OF A PROGRAM

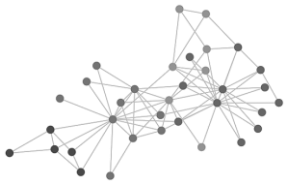
statement block	time in seconds	frequency	total time
E	t_0	x *(depends on input)*	$t_0 x$
D	t_1	$N^3/6 - N^2/2 + N/3$	$t_1 (N^3/6 - N^2/2 + N/3)$
C	t_2	$N^2/2 - N/2$	$t_2 (N^2/2 - N/2)$
B	t_3	N	$t_3 N$
A	t_4	1	t_4
		grand total	$(t_1/6)\, N^3 + (t_2/2 - t_1/2)\, N^2 + (t_1/3 - t_2/2 + t_3)\, N + t_4 + t_0 x$
		tilde approximation	$\sim (t_1/6)\, N^3$ *(assuming x is small)*
		order of growth	N^3



ASTANA IT UNIVERSITY

COMPLEXITY CLASSES

order of growth	name	typical code framework	description	example	$T(2n) / T(n)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log n$	**logarithmic**	`while (n > 1) { n = n/2; ... }`	divide in half	binary search	~ 1
n	**linear**	`for (int i = 0; i < n; i++) { ... }`	single loop	find the maximum	2
$n \log n$	**linearithmic**	*see mergesort*	divide and conquer	mergesort	~ 2
n^2	**quadratic**	`for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) { ... }`	double loop	check all pairs	4
n^3	**cubic**	`for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) for (int k = 0; k < n; k++) { ... }`	triple loop	check all triples	8
2^n	**exponential**	*see combinatorial search*	exhaustive search	check all subsets	2^n



CONSTANT COMPLEXITY

Complexity independent of inputs

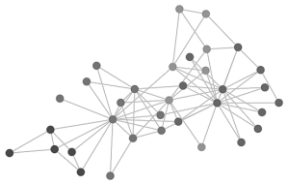
- No matter how many operations it performs exactly as long as it doesn`t depend on the number of inputs
- Time complexity is constant

```
public static int add(int a, int b) {
    return a + b;
}
```

Can have loops or recursive calls, but **only if** number of iterations or calls independent of input size

- It is still O(1) if it performs constant number of operations

```
public static void sayHello() {
    for (int i = 0; i < 1000; i++) {
        System.out.println("Hello");
    }
}
```



LOGARITHMIC COMPLEXITY

Complexity grows as log of size of one of its inputs

- When the number of iterations is divided to any constant K > 1 at each iteration (or halved in most cases)

```
public static void outputMiddles(int[] arr) {
    int middle = arr.length / 2;
    int sum;

    while (middle > 0) {
        System.out.println(arr[middle]);
        middle /= 2;
    }
}
```

```
10
43 -35 49 3 12 -39 -23 42 15 16
-39
49
-35
```

How about this?

```
public static int sumHalf(int[] arr) {
    int n = arr.length;
    int sum = 0;

    for (int i = 0; i < n / 2; i++) {
        sum += arr[i];
    }

    return sum;
}
```

It is not decreasing at each iteration

$$O\left(\frac{n}{2}\right) = O(n)$$

LINEAR COMPLEXITY

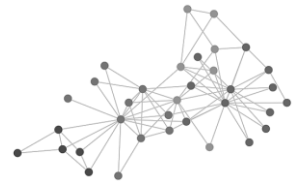
Can be in form of iterative loops or recursion

Iterative loops: depends on number of iterations

Recursion: depends on number of recursive function calls

```java
public static int factorial(int N) {
    int product = 1;
    for (int i = 1; i <= N; i++) {
        product *= i;
    }

    return product;
}
```

```java
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}
```

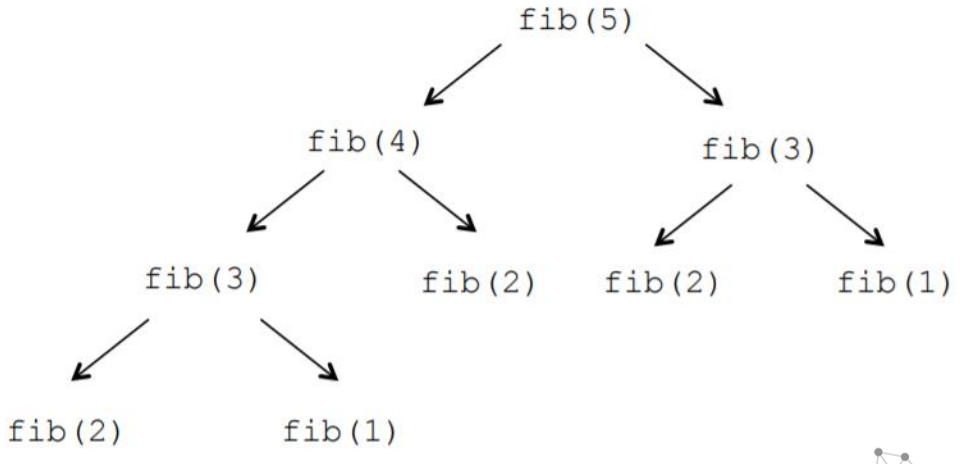


EXPONENTIAL COMPLEXITY

Exponential Time complexity denotes an algorithm whose growth is increasing exponentially (doubles in most cases) with each addition to the input data set

A good example is the recursive solution for Fibonacci Complexity is 2^n

```
public static int fib(int n) {
    if (n <= 1) return n; // base case
    // no need to write "else", since the
    // previous one will return
    return fib( n: n-2) + fib( n: n-1);
}
```

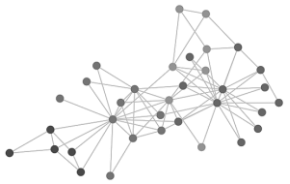


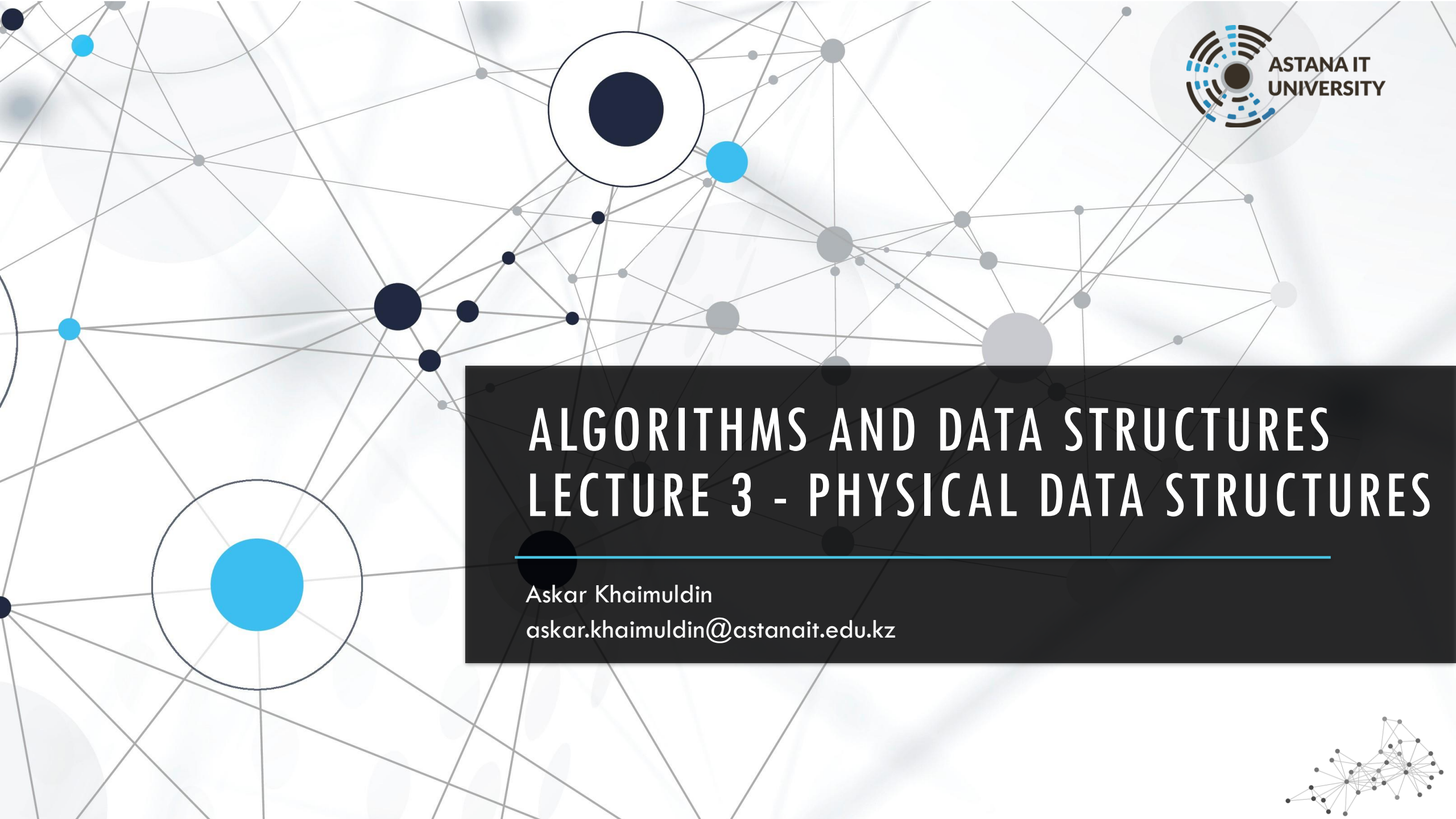
ASTANA IT UNIVERSITY

LITERATURE

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

- Chapter 1.4





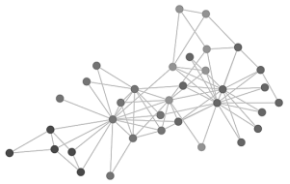
ALGORITHMS AND DATA STRUCTURES
LECTURE 3 - PHYSICAL DATA STRUCTURES

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

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

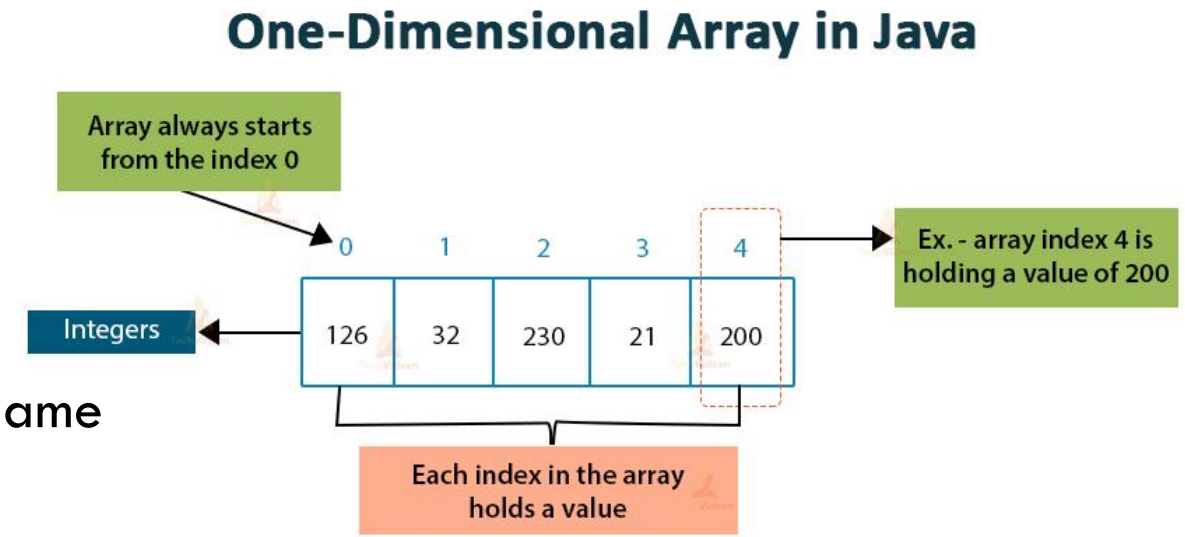
Static entity (**same size** throughout program)

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?

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

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

```java
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

```
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…

```
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
}
```

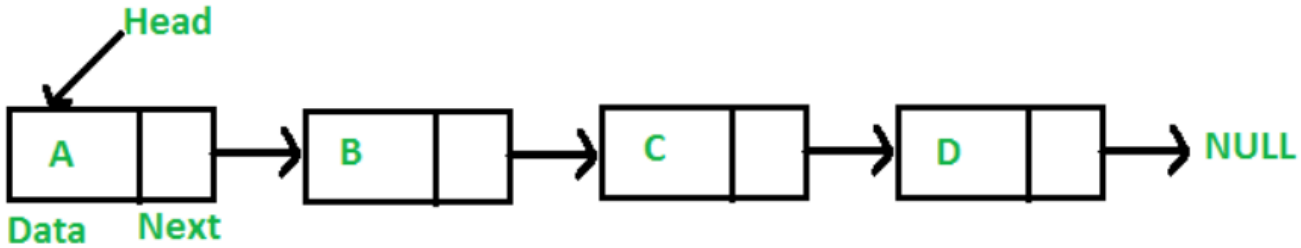
```
public void removeLast() {
    size--;
}

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

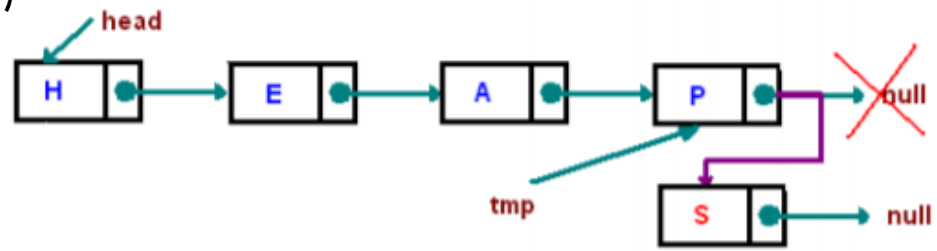
In order to iterate to the next element, we use:

```
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++;
}
```

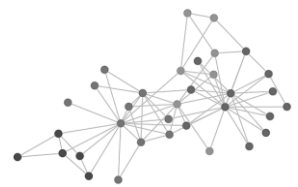
```
public void add(int newItem) {
    MyNode newNode = new MyNode(newItem);
    if (head == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }

    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:

```
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
```

```
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

```
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;
}
```

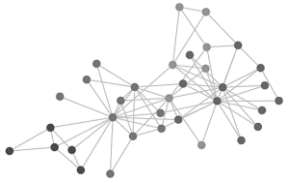
get(int index) method`s complexity:

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

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

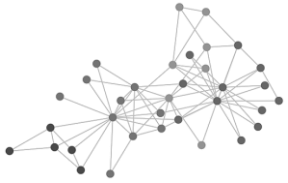
That method should return an instance of **Iterator**

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

Implement hasNext() and next() methods

See next slide for results…

```
@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

```
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)

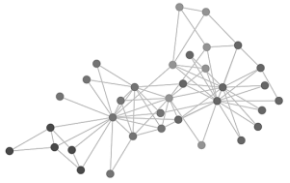
MyLinkedList: public void add(T newItem, int index)

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>



ASTANA IT UNIVERSITY

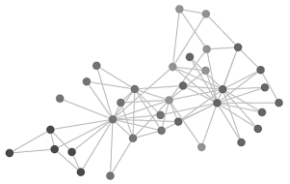
LITERATURE

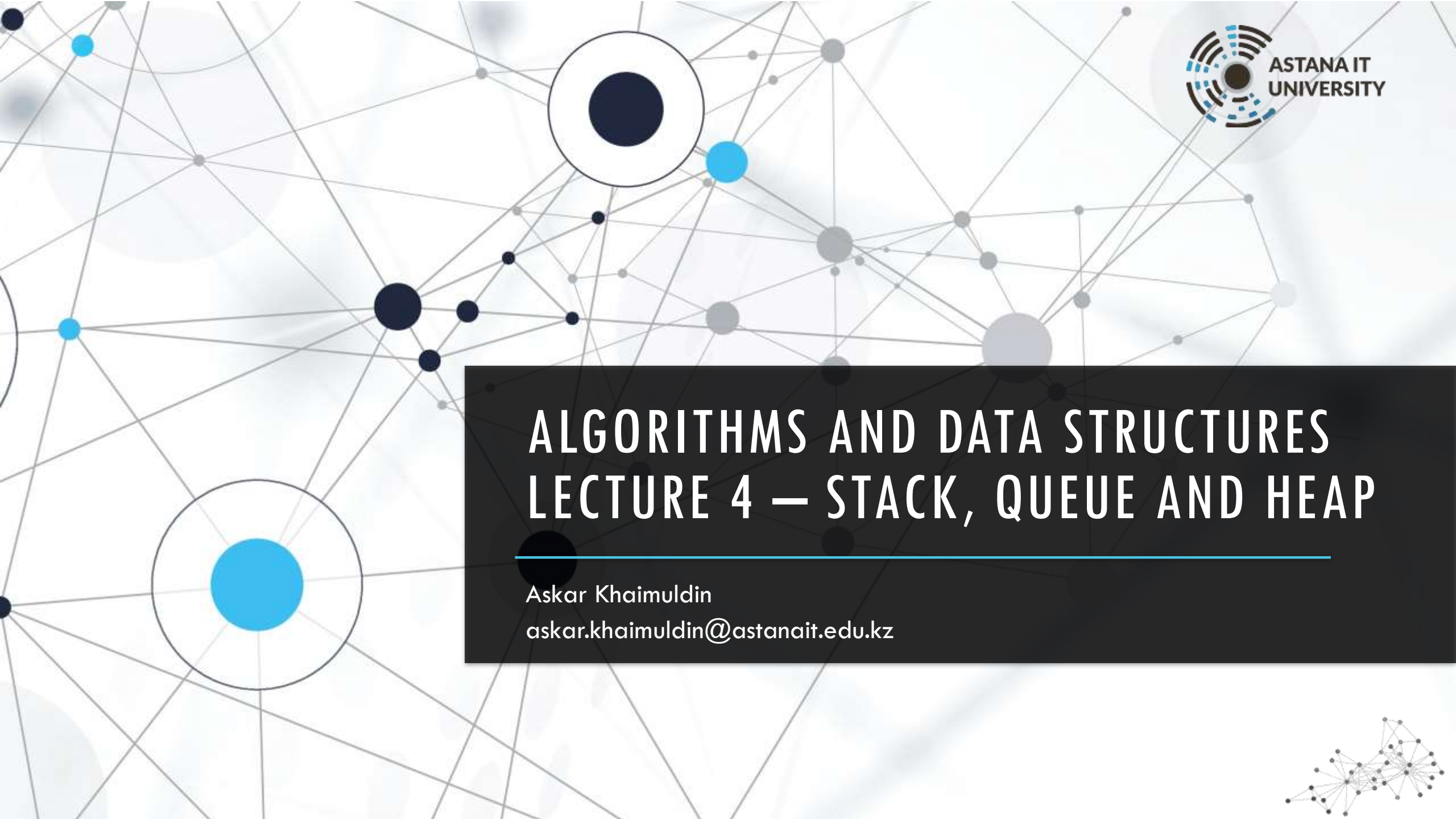
Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 1.3

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 2 – Arrays and Linked Lists





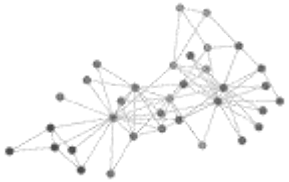
ALGORITHMS AND DATA STRUCTURES LECTURE 4 — STACK, QUEUE AND HEAP

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Preface
2. Stack
3. Queue
4. Heap
5. Heap<T extends Comparable<T>>



PREFACE

Logical Data Structures

- Linear (Stack, Queue, etc.)
- Non-linear (Tree, Hash-Table, Graph, etc.)

A Linear data structure has data elements arranged in a **sequential manner** and each member element is connected to its previous and next element

Data structures where data elements are attached in hierarchical manner are called non-linear data structures. One element could have several paths to another element

Logical Data Structures are implemented using either an array, a linked list, or a combination of both

STACK

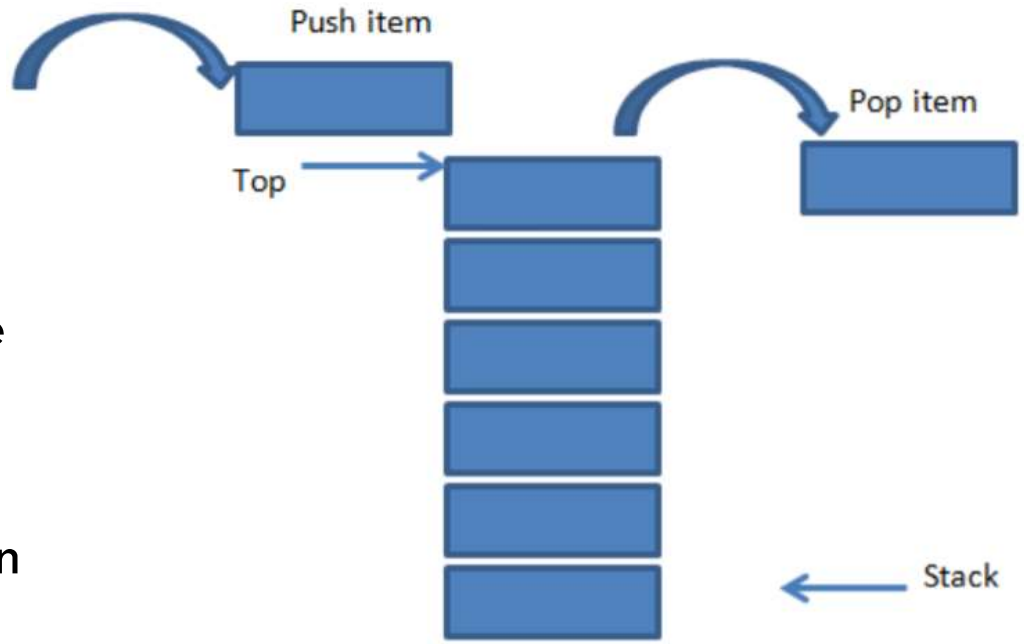
It is a linear data structure that follows the **LIFO** (Last-In-First-Out) principle

Last added item will be served first

It has **only one** end (named as 'top')

Insertion and deletion operations are performed at the top only

A stack can be implemented using linked list as well as an array. However, extra restrictions must be applied in order to follow LIFO



STACK:API

boolean empty() – Returns whether the stack is empty – Time Complexity : O(1)

int size() – Returns the size of the stack – Time Complexity : O(1)

T peek() – Returns a reference to the topmost element of the stack – Time Complexity : O(1)

T push(T) – Adds the element at the top of the stack – Time Complexity : O(1)

T pop() – Retrieves and deletes the topmost element of the stack – Time Complexity : O(1)

STACK:EXAMPLE

Topmost item at position n-1 (Array)

```
public T push(T newItem) {
    // Add a new item to the end
    // of the list
    addLast(newItem);

    // Return just added item
    return newItem;
}

public T peek() {
    // Get last element
    return get(size - 1);
}

public T pop() {
    // Get topmost item
    T removingItem = peek();

    // Remove topmost item
    removeLast();

    // Return just removed item
    return removingItem;
}
```

Topmost item at position 0 (Linked List)

```
public T push(T newItem) {
    // Add a new item to the front
    // of the list
    addFront(newItem);

    // Return just added item
    return newItem;
}

public T peek() {
    // Get front element
    return get(0);
}

public T pop() {
    // Get topmost item
    T removingItem = peek();

    // Remove topmost item
    removeFront();

    // Return just removed item
    return removingItem;
}
```

QUEUE

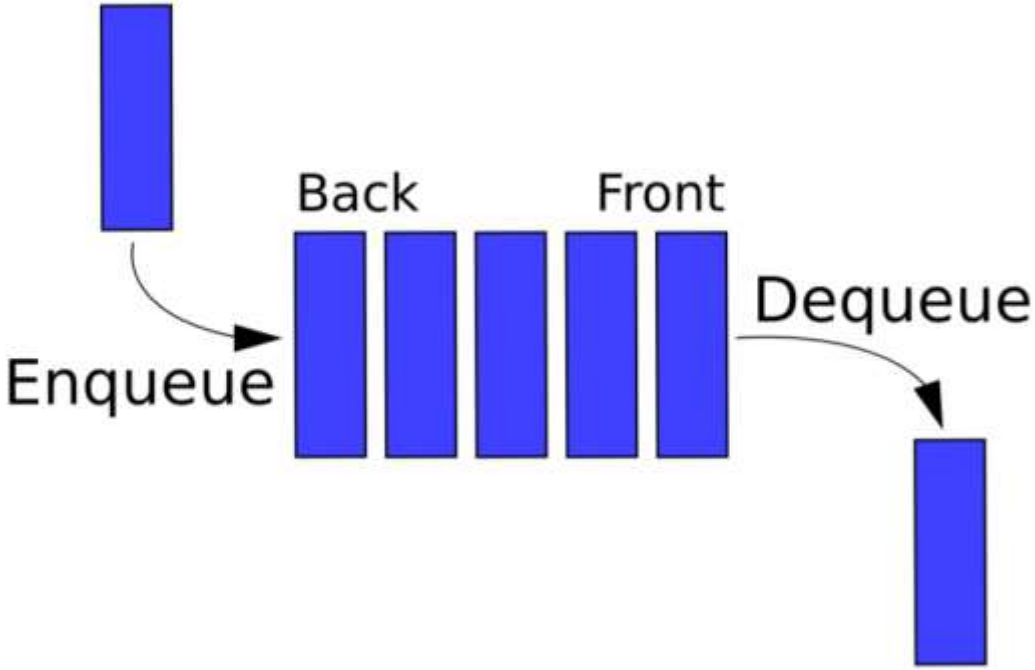
It is a linear data structure that follows the FIFO (First-In-First-Out) principle

First added item will be served first

It has two ends (named as ‘Front’ and ‘Back’)

Insertion (enqueue) and deletion (dequeue) operations are performed at different sides

A queue can be implemented using linked list as well as an array. However, it shows better performance with linked list, which has both head and tail references



QUEUE:API

boolean empty() – Returns whether the queue is empty

int size() – Returns the size of the queue

T peek() – Returns a reference to the front element of the queue

T enqueue(T) – Adds the element at the end of the queue

T dequeue() – Retrieves and deletes the front element of the queue

QUEUE:EXAMPLE

It is also possible to provide two methods for each of the followings:

Peek

- peek() – returns null when queue is empty
- element() – throws an exception when queue is empty

Enqueue

- boolean offer(T) – returns false if it fails to insert
- add(T) – throws an exception if it fails to insert

Dequeue

- remove() – returns null when queue is empty
- poll() – throws an exception when queue is empty

```
public T peek() {
    // Get front element
    return get(0);
    // can be get(n-1)
    // it depends which side is Front
}

public T enqueue(T newItem) {
    // Add a new item to the end
    // of the queue
    addBack(newItem);

    // Return just added item
    return newItem;
}

public T dequeue() {
    // Get front item
    T removingItem = peek();

    // Remove topmost item
    removeFront();

    // Return just removed item
    return removingItem;
}
```

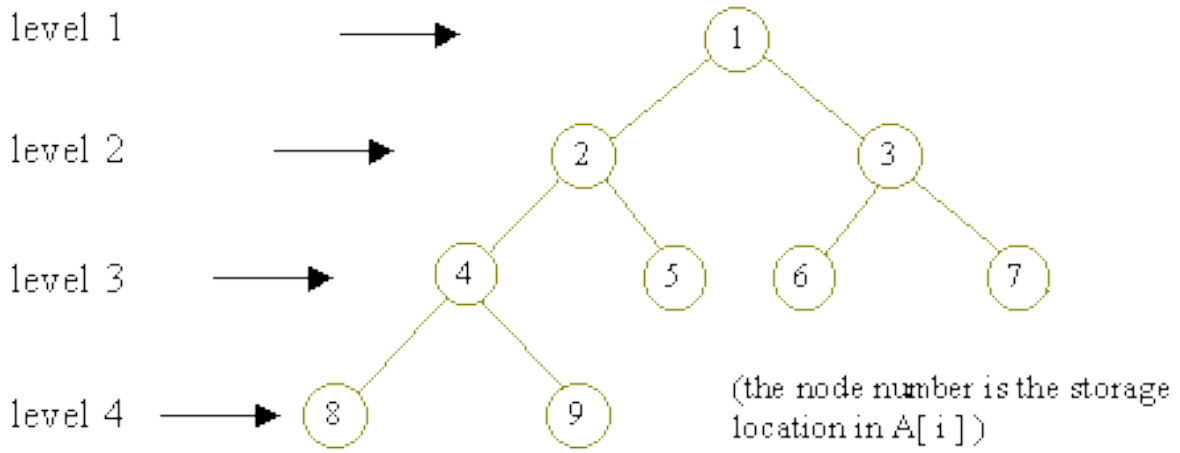
HEAP DATA STRUCTURE

It is a complete binary tree

- Each level of the tree is filled, except the last one
- Each level is filled from left to right

Types:

- Min Heap – $A[parent[i]] \geq A[i]$
- Max Heap – $A[parent[i]] \leq A[i]$



It satisfies the heap-order property

- The data item stored in each node is **smaller** than or equal to any of the data items stored in its children (**Min Heap**)
- The data item stored in each node is **greater** than or equal to any of the data items stored in its children (**Max Heap**)

HEAP DATA STRUCTURE

It allows you to find the *largest/smallest element in the heap in O(1) time

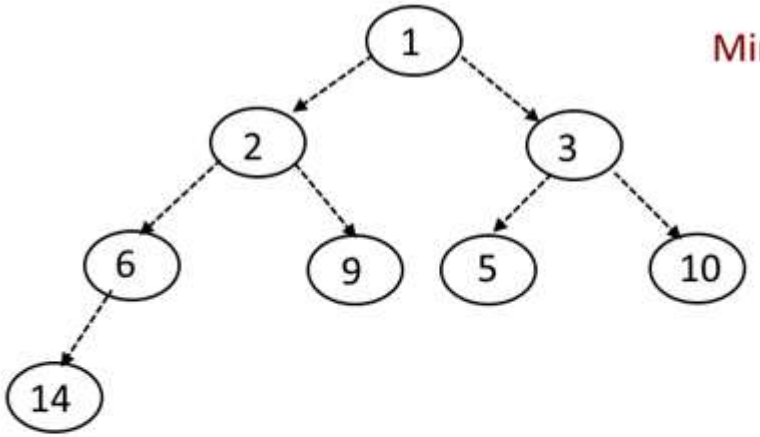
Extracting the *largest/smallest element from the heap (i.e. finding and removing it) takes O(log n) time

Heap can be implemented using:

- Array (manipulating its indices)
- Nodes with references to their right and left children (not covered)

The root is stored at index 1, and if a node is at index i, then

- Its left child has index 2i
- Its right child has index 2i+ 1
- Its parent has index i/2



0	1	2	3	4	5	6	7	8
	1	2	3	6	9	5	10	14

for Node at i : Left child will be 2i and right child will be at 2i+1 and parent node will be at [i/2].

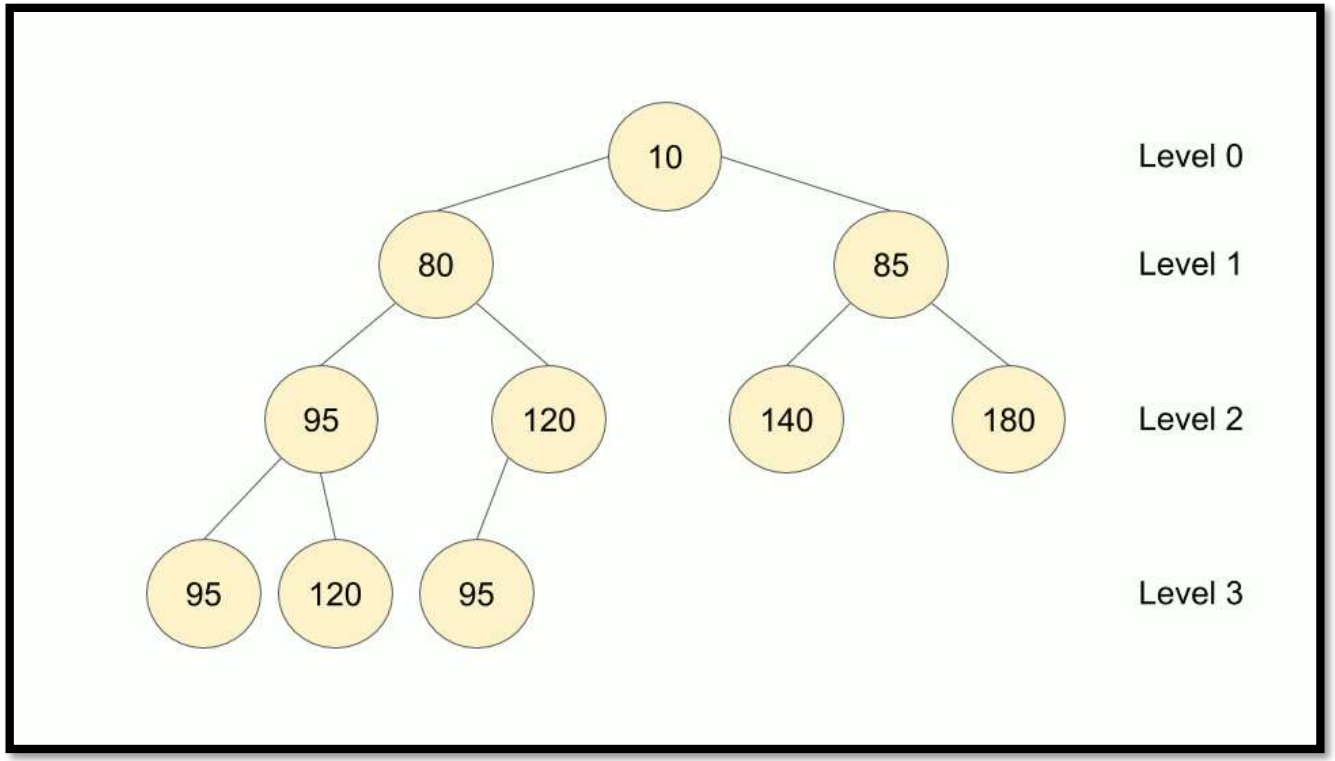
*largest/smallest – largest for Max Heap and smallest for Min Heap

HEAP:INSERTION – O(LOG(N))

A new item is added as the last element

Recursive actions (**traverse up**):

- Compare with parent
- Exchange if it violates the **property**
- Stops when no other violations or it has reached the root

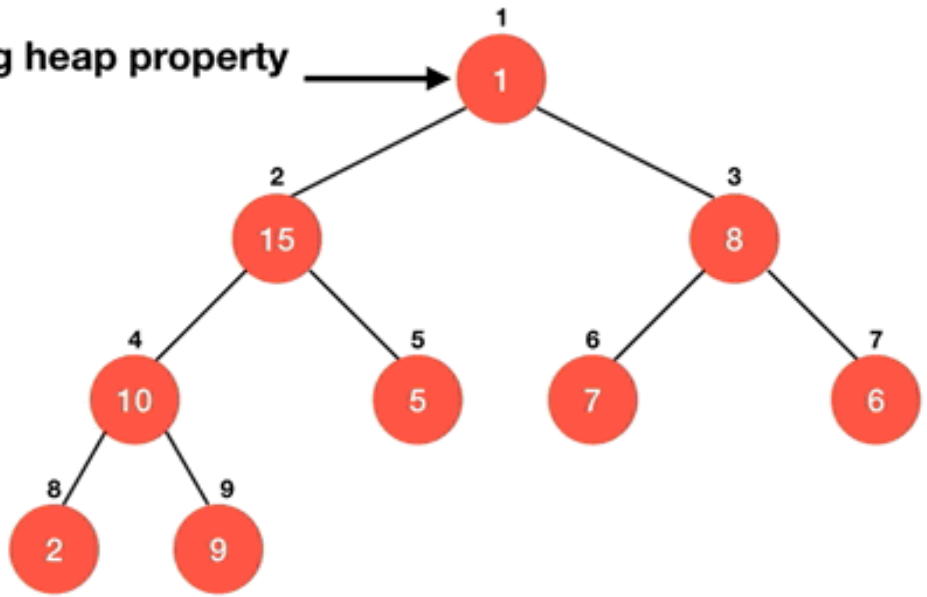


HEAP:HEAPIFY – O(LOG(N))

Max Heap example

Heapify(i) – fixes the violation of heap property at any position ***i*** (assuming that violation is only at i`th position)

- Replace an element at i with the largest of children
- Recall Heapify(largestIndex)
- Stops when current item is larger than children (or equal) or there`s no other child items



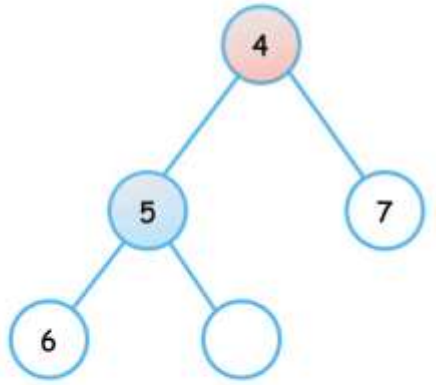
HEAP:EXTRACT_MIN – O(LOG(N))

Min Heap example

A root item is replaced with the last element

Recursive actions:

- Heapify(rootIndex)



extractMin( )
root = 1
- heapify( )

Min Heap extract min

HEAP:METHODS

Public:

- empty() – Returns whether the heap is empty
- size() – Returns the size of the heap
- T getMax() or getMin() – Returns a reference to the root element of the heap
- T extractMax() or extractMin() – Retrieves and deletes the root element of the heap
- insert(T) – Adds the element to the heap

Private:

- heapify(index) – can perform heapify actions starting from position 'index'
- traverseUp(index) – can perform traverseUp actions starting from position 'index'
- leftChildOf(index) – returns the index of the left child item
- rightChildOf(index) – returns the index of the right child item
- parentOf(index) - returns the index of the parent item
- swap(index1, index2) – exchanges two elements by their positions

HEAP<T EXTENDS COMPARABLE<T>>

There are several comparisons in Heap

It is not possible to use >, <, <=, etc. operators when dealing with objects (not primitives)

Comparable<T> is an interface that provides a method obj1.**compareTo**(obj2), which returns a number

- More than 0 when obj1 is greater than obj2
- Less than 0 when obj1 is smaller than obj2
- Exactly 0 when obj1 is equal to obj2

That comparison is defined in object itself

- Classes that are already Comparable: Integer, Double, String, etc.
- If heap stores objects of user-defined type, then that type should implement Comparable<T> interface

HEAP<T EXTENDS COMPARABLE<T>>

```java
public class Student implements Comparable<Student> {
    private String name;
    private int grade;

    // other code

    // example
    @Override
    public int compareTo(Student another) {
        int diff = this.grade - another.grade;
        if (diff == 0)
            return this.name.compareTo(another.name);

        return diff;
    }
}
```

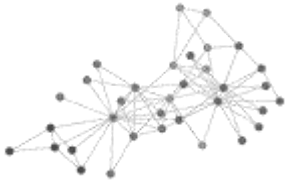
```java
public static void main(String[] args) {
    // other code

    MyMinHeap<Student> heap = new MyMinHeap<>();

    // another code
}
```

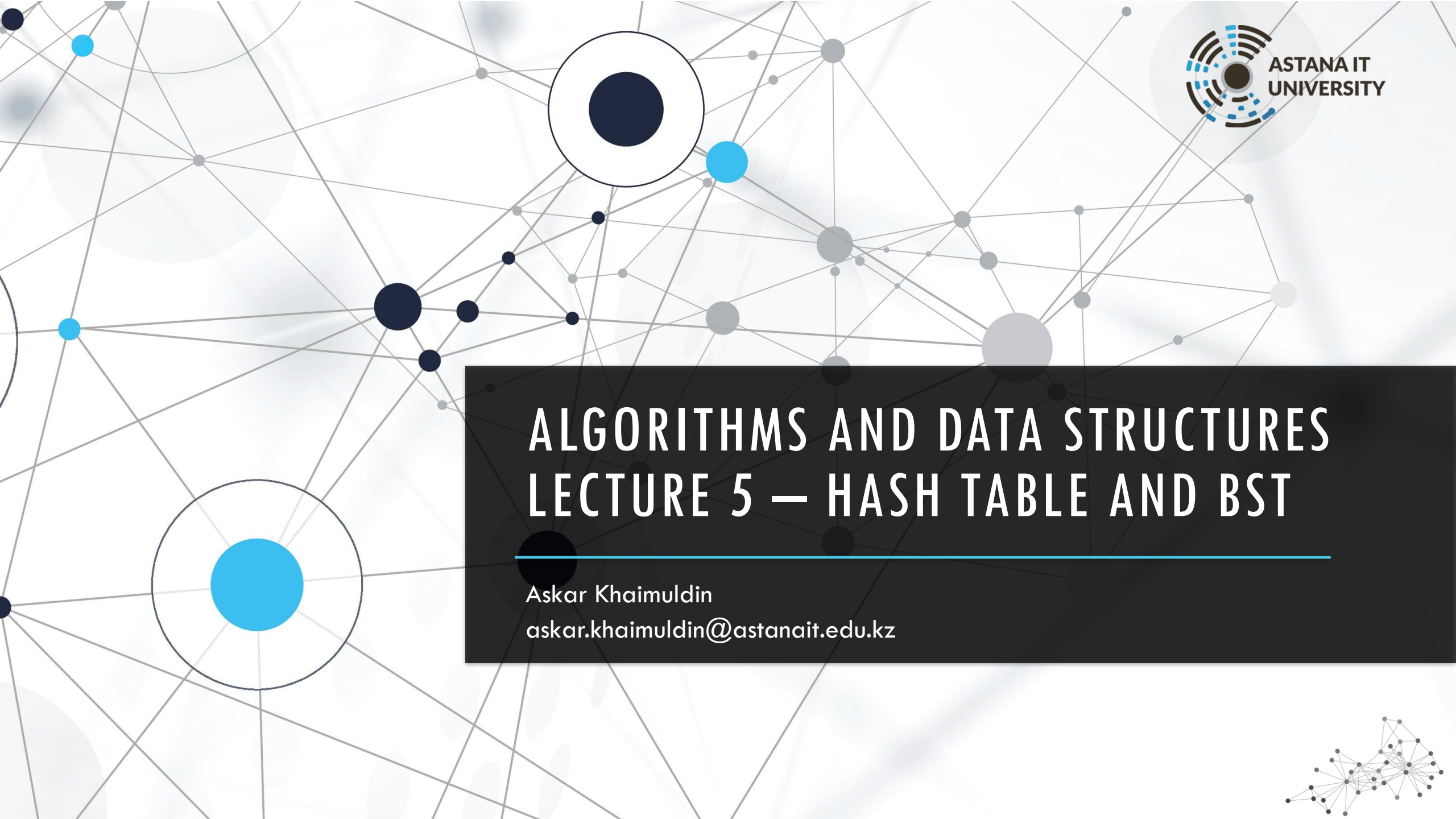
```java
public class MyMinHeap<T extends Comparable<T>> {
    private Object[] array;
    private int size = 0;
    private int capacity = 5;


    // other code

    public T getMin() {
        return get(1); // or get(0)
        // depends on the index of root
    }

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

    public void anyMethodWithCompare(int index) {
        T left = get(leftChildInd(index));
        T right = get(rightChildInd(index));
        if (left.compareTo(right) > 0) {
            // another code
        }
    }

    private int leftChildInd(int index) { return 2 * index; }

    private int rightChildInd(int index) { return 2 * index + 1; }
}
```

LITERATURE

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

- Chapter 1.3, 2.4





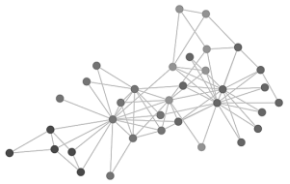
ALGORITHMS AND DATA STRUCTURES LECTURE 5 – HASH TABLE AND BST

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Hashing
2. Hash Table
3. Binary Search Tree
4. BST: Inorder Traversal



ASTANA IT UNIVERSITY

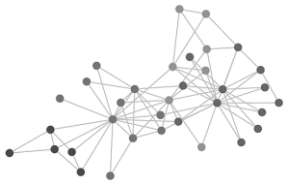
HASHING

Hashing means using some function or algorithm to map object data to some representative integer value

This so-called hash code (or simply hash) can then be used as a way to narrow down our search when looking for the item in the set

Object class contains hashCode() method with its default implementation

Recommended: Each class provides its own implementation of hashCode()



HASHING: STRING EXAMPLE

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

ith character of s

```
String s = "call";
int code = s.hashCode();
```

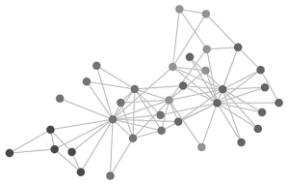
$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

(Horner's method)

Horner's method to hash string of length L: L multiplies/adds.

Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$



HASHING: 'STANDARD' RECIPE

Combine each **significant** field using the 31x + y rule.

If field is a primitive type, use wrapper type hashCode()

If field is null, return 0

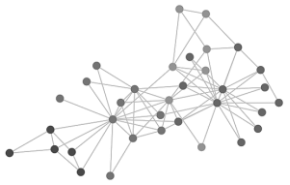
If field is a reference type, use hashCode()

If field is an array, apply to each entry

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  amount;

   ...

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime



HASH TABLE

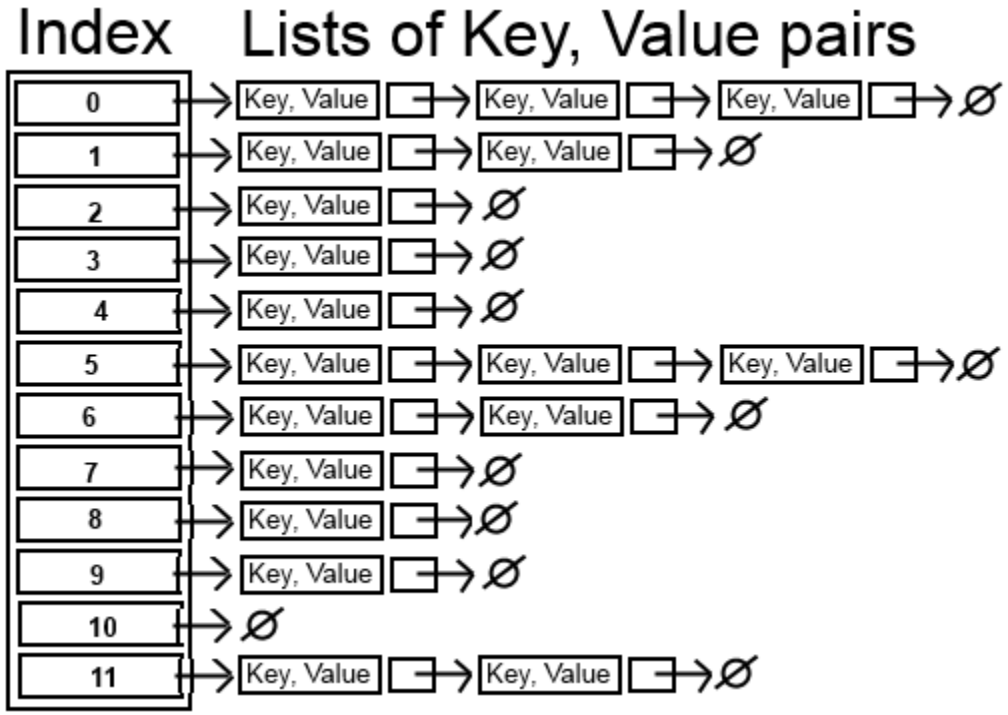
Hash table maps keys to values. Any non-null object can be used as a key or as a value

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

It looks like “an array of singly-linked lists (**chains**)”

Each linked list is accepted as **bucket**

Array size indicates number of buckets



Average case:

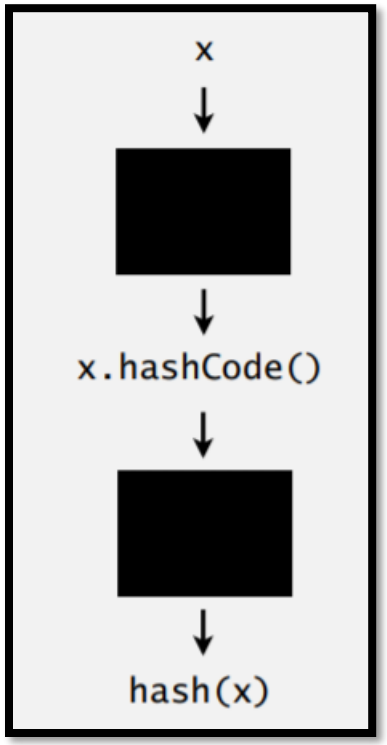
- Insertion = deletion = retrieving = searching = **O(1)**

HASH TABLE

The **capacity** (number of buckets - **M**) and **load factor** are parameters that affect to its performance

The **load factor** is a measure of how full the hash table is allowed to get before its capacity is automatically increased (LF should be around 0.75)

The hashCode is used to get an index of *chain* by **hash()** method (**Modular hashing**)




```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

correct

HASH TABLE

Collision – having same index for several nodes (cannot be avoided)

- A new node should be added to the same chain (bucket)

Challenge: Deal with collisions efficiently

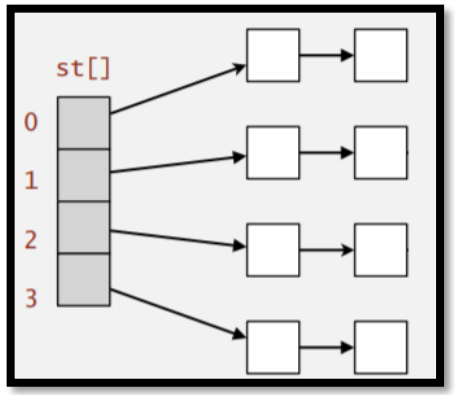
Target: Uniform distribution

Analysis: Number of probes for search/insert is proportional to N/M

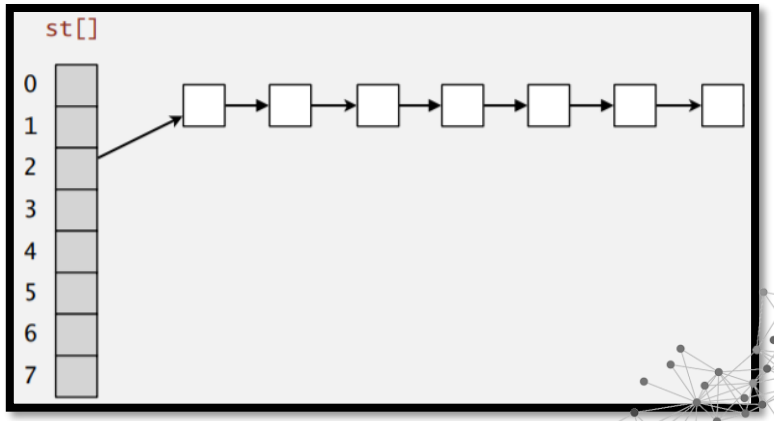
- M too large ⇒ too many empty chains
- M too small ⇒ chains too long
- Typical choice: M ~ N / 4 ⇒ constant-time ops

Once a hash table has passed its load factor - it has to rehash [create a new bigger table, and re-insert each element to the table]

Best case



Worst case



HASH TABLE: EXAMPLE

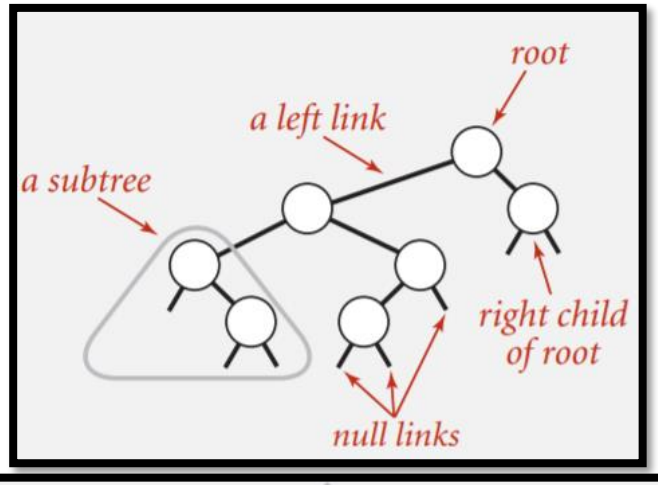
```java
public class MyHashTable<K, V> {

    private class HashNode<K, V> {...}

    private HashNode<K, V>[] chainArray; // or Object[]
    private int M = 11; // default number of chains
    private int size;

    public MyHashTable() {...}

    public MyHashTable(int M) {...}

    private int hash(K key) {...}

    public void put(K key, V value) {...}

    public V get(K key) {...}

    public V remove(K key) {...}

    public boolean contains(V value) {...}

    public K getKey(V value) {...}
}
```

```java
private class HashNode<K, V> {
    private K key;
    private V value;
    private HashNode<K, V> next;

    public HashNode(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString() {
        return "{" + key + " " + value + "}";
    }
}
```

BINARY SEARCH TREE

A BST is a binary tree in symmetric order.

Each node has two references to left and right nodes

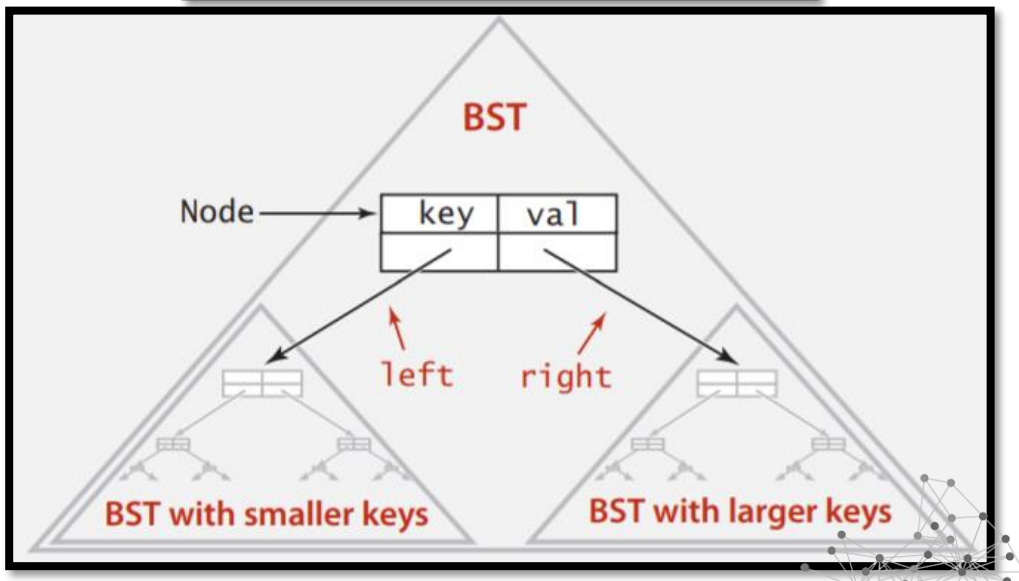


Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree
- Smaller than all keys in its right subtree

A Node is composed of four fields

- Key and Value
- Left and right subtree references



BINARY SEARCH TREE

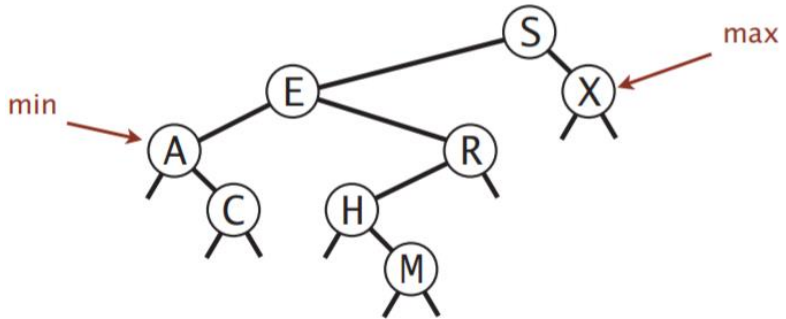
A BST uses O(log(N)) for most manipulations

Search: If less, go left; if greater, go right; if equal, search hit

Insert: If less, go left; if greater, go right; if null, insert

GetMin(): Most left node

GetMax(): Most right node




```
public class BST<K extends Comparable<K>, V> {
    private Node root;
    private class Node
    {
        private K key;
        private V val;
        private Node left, right;
        public Node(K key, V val)
        {
            this.key = key;
            this.val = val;
        }
    }
    public void put(K key, V val) {...}

    public V get(K key) {...}

    public void delete(K key) {...}

    public Iterable<K> iterator() {...}
}
```

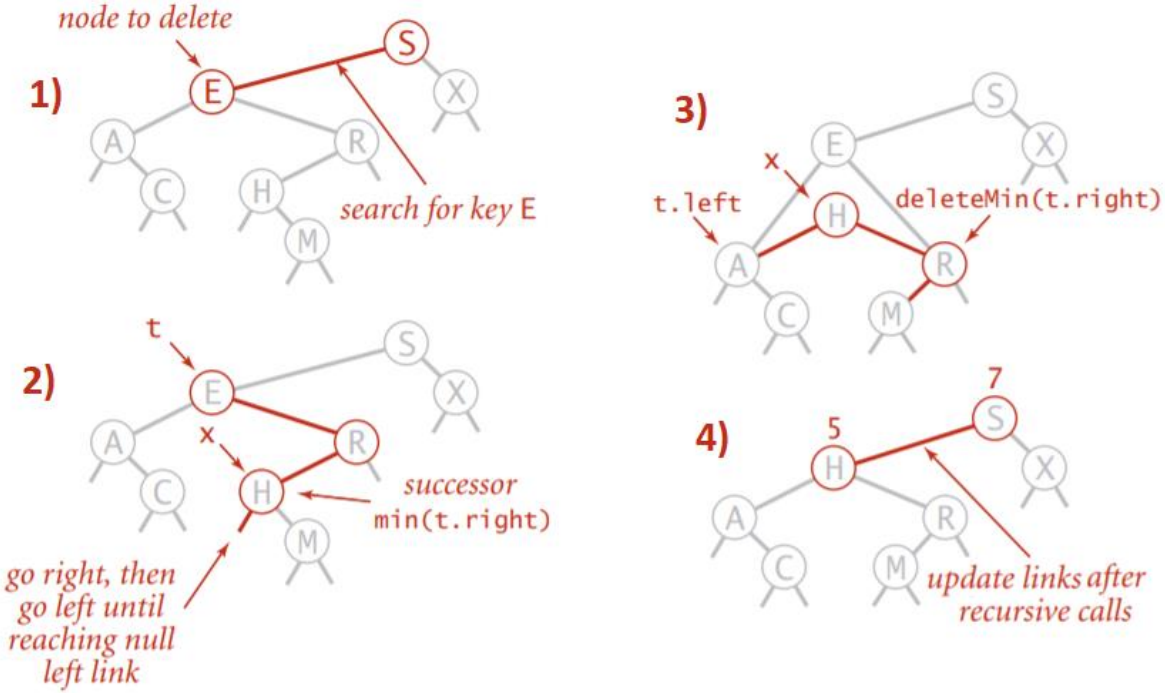
BINARY SEARCH TREE: DELETE

To delete a node with key k: search for node t containing key k

Case 1 (1 child): Delete t by replacing parent link

Case 2 (2 children):

- Find successor x of t
- Delete the minimum in t's right subtree
- Put x in t's spot



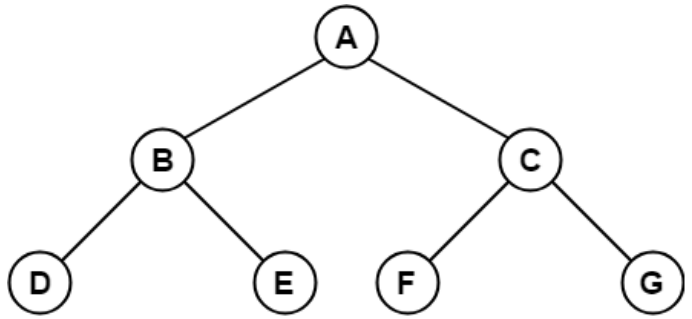
guarantee			average case			ordered ops?	operations on keys
search	insert	delete	search hit	insert	delete		
N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	compareTo()

BST: INORDER TRAVERSAL

In BST, **inorder traversal** is used to get nodes in increasing order (Left-Root-Right)

Ordered iteration

- Traverse left subtree
- Enqueue key
- Traverse right subtree



Inorder Traversal : D , B , E , A , F , C , G

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

ASTANA IT UNIVERSITY

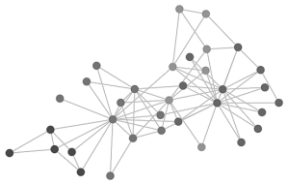
LITERATURE

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

- Chapters 3.2, 3.4

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 5



ASTANA IT UNIVERSITY

GOOD LUCK!