Sorting and Searching

Nguyễn Công Thành—BI9-210
Nguyễn Gia Phong—BI9-184
Nguyễn Văn Tùng—BI9-229
Trần Minh Vương—BI9-239
Trần Hồng Minh—BI8-114

University of Science and Technology of Hà Nội

November 30, 2019

Contents

Introduction

- Sorting & Searching are *important*

Introduction

- Sorting & Searching are *important*
- Object-Oriented Programming

Introduction

- Sorting & Searching are *important*
- Object-Oriented Programming
- Implementation in Java

Introduction

- Sorting & Searching are *important*
- Object-Oriented Programming
- Implementation in Java
- Generic Programming

Searching

Given a value x, return the [zero-based] index of x in the array, if such x exists. Otherwise, return `NOT_FOUND` (-1).

Linear Search

- Checks sequentially till

Linear Search

- Checks sequentially till
  - match found

Linear Search

- Checks sequentially till
  - match found
  - whole list searched

Linear Search

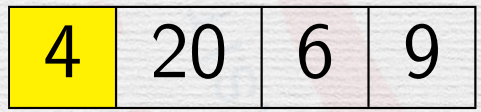
- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity

Linear Search

- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity
- `java.util.List.indexOf`

Linear Search

- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity
- `java.util.List.indexOf`
- Example: Search for number 7



Linear Search

- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity
- `java.util.List.indexOf`
- Example: Search for number 7

Linear Search

- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity
- `java.util.List.indexOf`
- Example: Search for number 7

Linear Search

- Checks sequentially till
  - match found
  - whole list searched
- Linear time complexity
- `java.util.List.indexOf`
- Example: Search for number 7

4	20	6	9

Implementation

```
import java.util.List;

public class Search
{
  public static final int NOT_FOUND = -1;

  public static linear(List l, Object o)
  {
    for (int i = 0; i < l.size(); ++i)
      if (o == null ? l.get(i) == null
                    : o.equals(l.get(i)))
        return i;
    return NOT_FOUND;
  }
}
```

Binary Search

- For sorted arrays only

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval

Binary Search

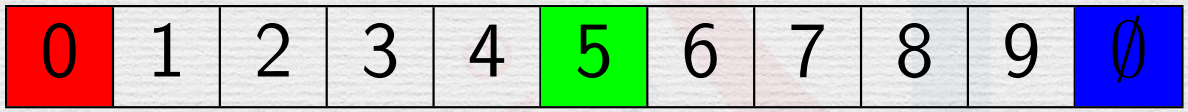
- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity
- `java.util.Collections.binarySearch`

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity
- `java.util.Collections.binarySearch`
- Example: Search for number 7



Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity
- `java.util.Collections.binarySearch`
- Example: Search for number 7

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity
- `java.util.Collections.binarySearch`
- Example: Search for number 7

Binary Search

- For sorted arrays only
- Repeat halving interval cannot have x till
  - match found
  - invalid interval
- Logarithmic time complexity
- `java.util.Collections.binarySearch`
- Example: Search for number 7

Implementation

```
public class Search
{
  private static <T> int binary(
    List<? extends Comparable<? super T>> list,
    T key, int low, int high)
  {
    if (high < low)
      return NOT_FOUND;
    var mid = (low + high) / 2;
    var cmp = list.get(mid).compareTo(key);
    if (cmp < 0)
      return binary(list, key, mid + 1, high);
    if (cmp > 0)
      return binary(list, key, low, mid - 1);
    return mid;
  }
}
```

Wrapper

```
public class Search
{
  public static <T> int binary(
    List<? extends Comparable<? super T>> list,
    T key)
  {
    return binary(list, key, 0, list.size());
  }
}
```

Sorting

Given an array of n values, arrange the values into ascending order.

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

6	9	4	2	0

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

0	9	4	2	6

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

0	2	4	9	6

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

Selection Sort

- Iterate through every position,
  select the minimum from there to array's end
- Quadratic time complexity
- Example:

0	2	4	6	9

Implementation

```
import static java.util.Collections.swap;

public class Sort
{
  public static <T extends Comparable<? super T>>
  void selection(List<T> list)
  {
    int i, j, m, n = list.size();
    for (i = 0; i < n; ++i)
      {
        for (m = j = i; j < n; ++j)
          if (list.get(j).compareTo(list.get(m)) < 0)
            m = j;
        swap(list, i, m);
      }
  }
}
```

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

6	9	4	2	0

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

6	9	4	2	0

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

6	4	9	2	0

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

6	4	2	9	0

Bubble Sort

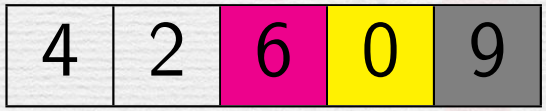
- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

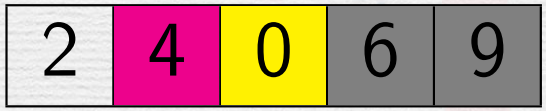


Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:



Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

Bubble Sort

- Repeatedly iterate through the array, swap adjacent elements in wrong order
- Quadratic time complexity
- Example:

0	2	4	6	9

Implementation

```
public class Sort
{
  public static <T extends Comparable<? super T>>
  void bubble(List<T> list)
  {
    for (int n = list.size(), m = 0;
         n > 1; n = m, m = 0)
      for (int i = 1; i < n; ++i)
        if (list.get(i).compareTo(list.get(i-1)) < 0)
          swap(list, m = i, i - 1);
  }
}
```

Bubble v Selection: Dawn of Sort

C. Thomas Wu (2010) claimed that

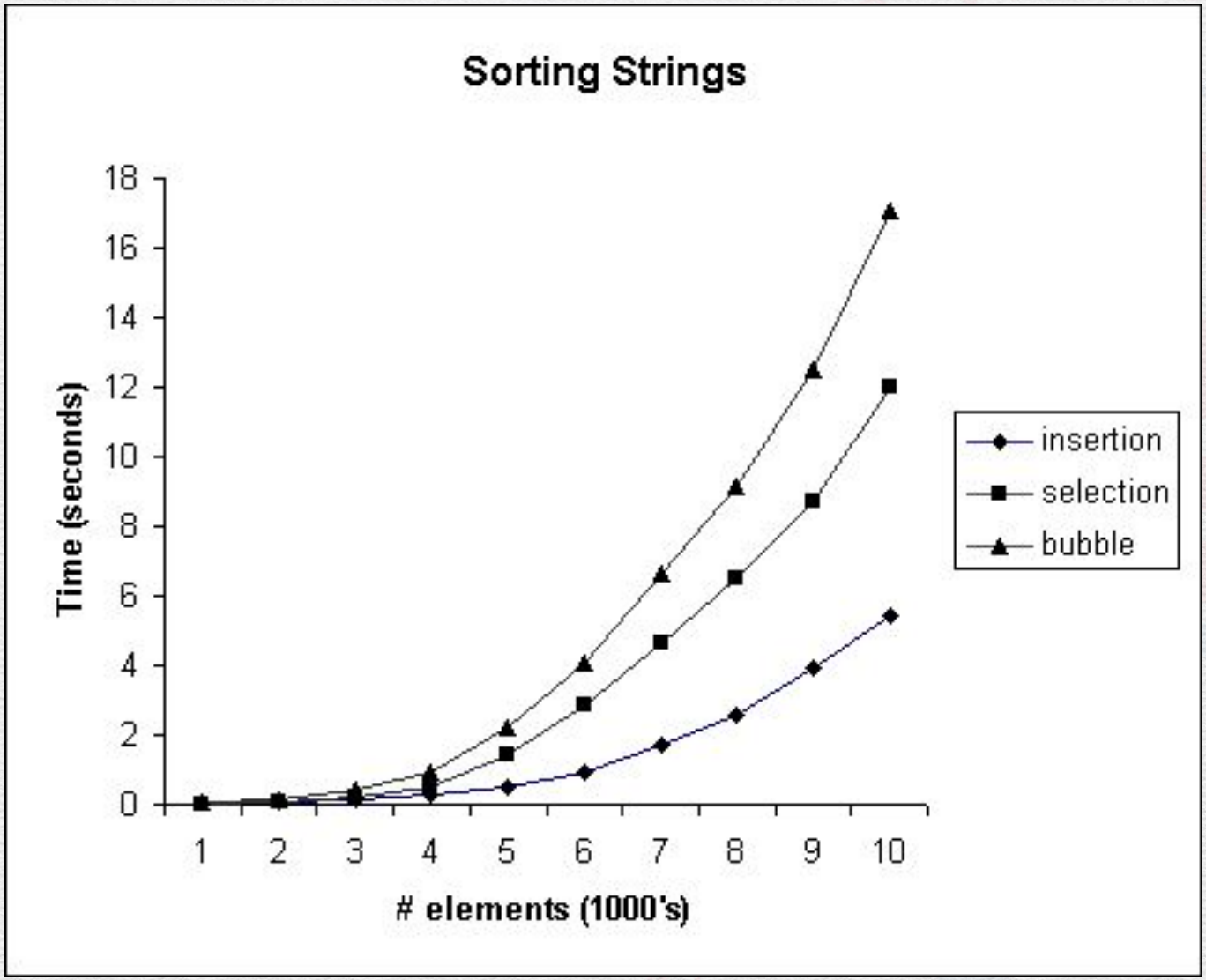
> *On average, we expect the bubble sort to finish sorting sooner than the selection sort, because there will be more data movements for the same number of comparisons, and there is a test to exit the method when the array gets sorted.*

Bubble v Selection: Dawn of Sort

BvS: Time Complexity

Case	Selection Sort		Bubble Sort	
	Comparisons	Swaps	Comparisons	Swaps
Best	$\Omega(n^2)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
Average	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$

BvS: Average Case in Practice



Heapsort

- Selection sort, but use heap for selection

Heapsort

- Selection sort, but use heap for selection
- Linearithmic time complexity

Using PriorityQueue (Min-Heap)

```
import java.util.PriorityQueue;

public class Sort
{
  public static <T extends Comparable<? super T>>
  void pq(List<T> list)
  {
    var q = new PriorityQueue<T>(list);
    for (int i = 0; i < list.size(); ++i)
      list.set(i, q.poll());
  }
}
```

Using PriorityQueue (Min-Heap)

```
import java.util.PriorityQueue;

public class Sort
{
  public static <T extends Comparable<? super T>>
  void pq(List<T> list)
  {
    var q = new PriorityQueue<T>(list);
    for (int i = 0; i < list.size(); ++i)
      list.set(i, q.poll());
  }
}
```

But hey, there is also `List.sort`!

Binary Max-Heap

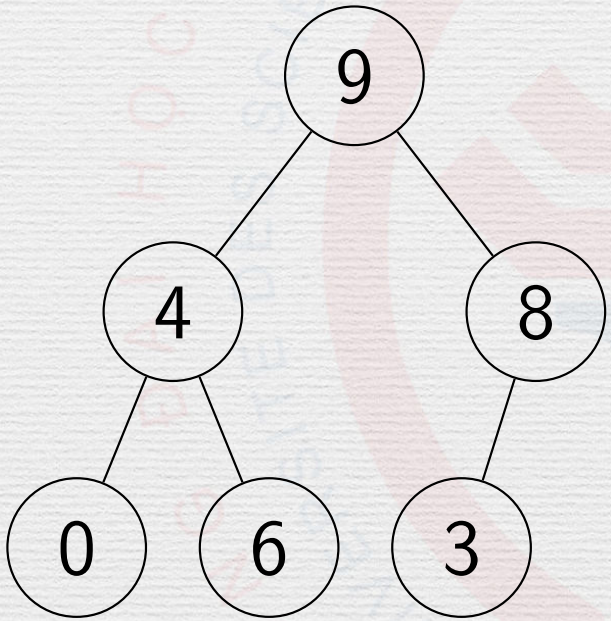
- Nearly complete binary tree

Binary Max-Heap

- Nearly complete binary tree
- Parent $\geq$ Children $\Longrightarrow$ Root is max!

Binary Max-Heap

- Nearly complete binary tree
- Parent $\geq$ Children $\Longrightarrow$ Root is max!
- Example:



Linear Binary Max-Heap

- $length$ of inner representation

Linear Binary Max-Heap

- $length$ of inner representation
- $size$ of heap $(0 \leq size \leq length)$

Linear Binary Max-Heap

- $length$ of inner representation
- $size$ of heap $(0 \leq size \leq length)$
- Index within $[0 \,..\, size)$

$$\begin{aligned} \text{parent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor \\ \text{left}(i) &= 2i+1 \\ \text{right}(i) &= 2i+2 \end{aligned}$$

Heap Declaration

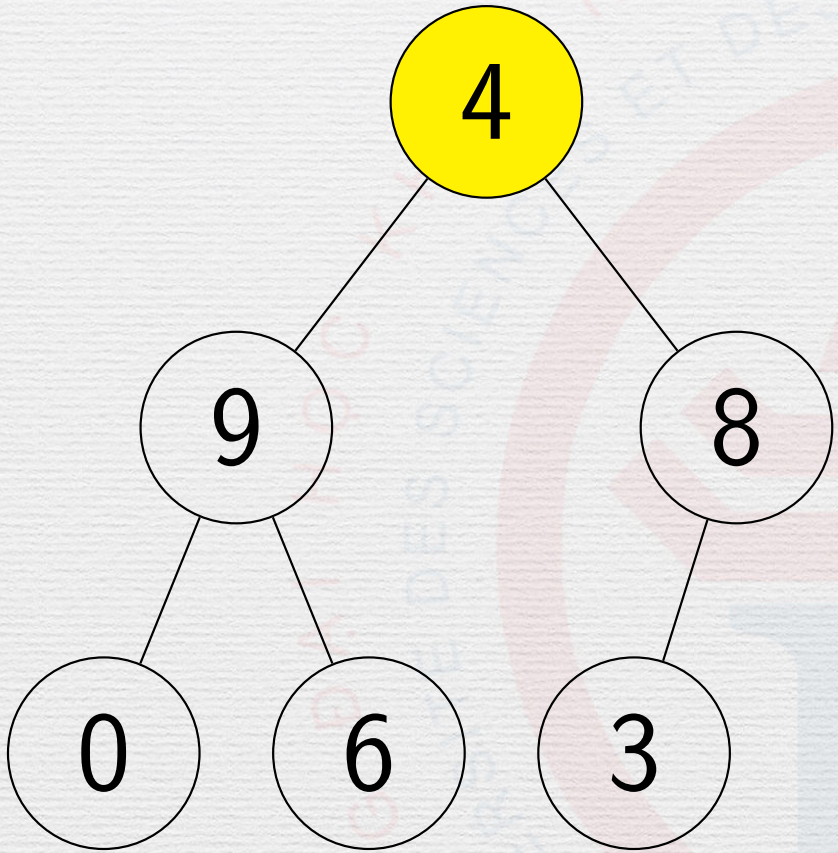
```
public class Heap<T extends Comparable<? super T>>
{
  private List<T> list;
  private int size;

  public int getSize() { return size; }
  public int getLength() { return list.size(); }
  public T get(int i) { return list.get(i); }
}
```

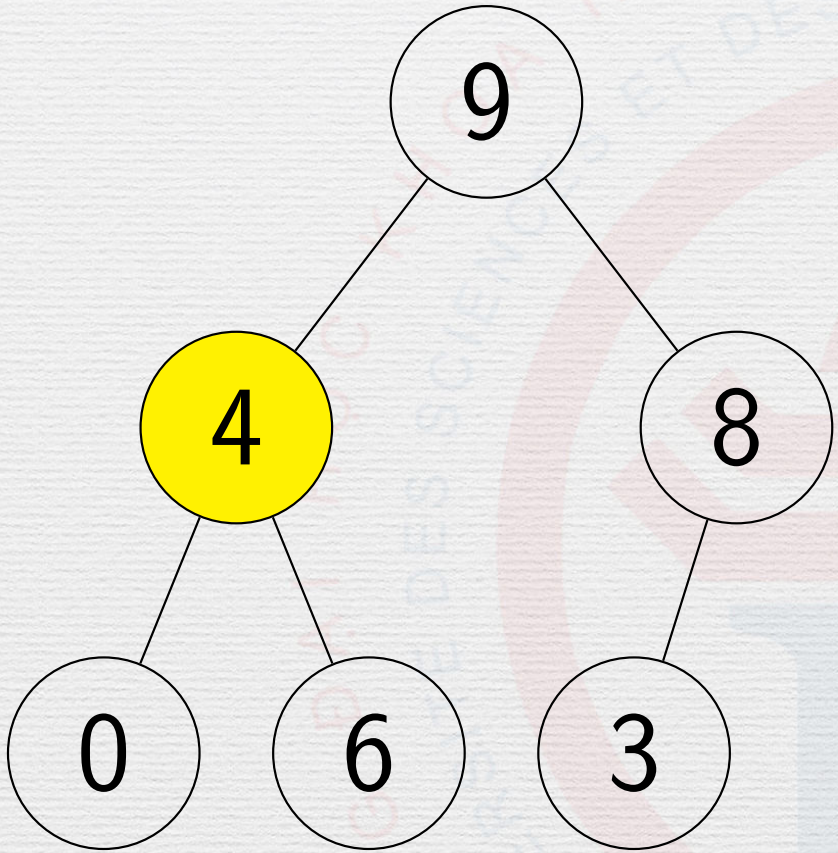
void Heap::heapify(int i)

```
int right = i + 1 << 1;
int left = right - 1;
int largest = i;
if (left < size
    && get(left).compareTo(get(largest)) > 0)
  largest = left;
if (right < size
    && get(right).compareTo(get(largest)) > 0)
  largest = right;
if (largest != i)
  {
    swap(list, i, largest);
    heapify(largest);
  }
```

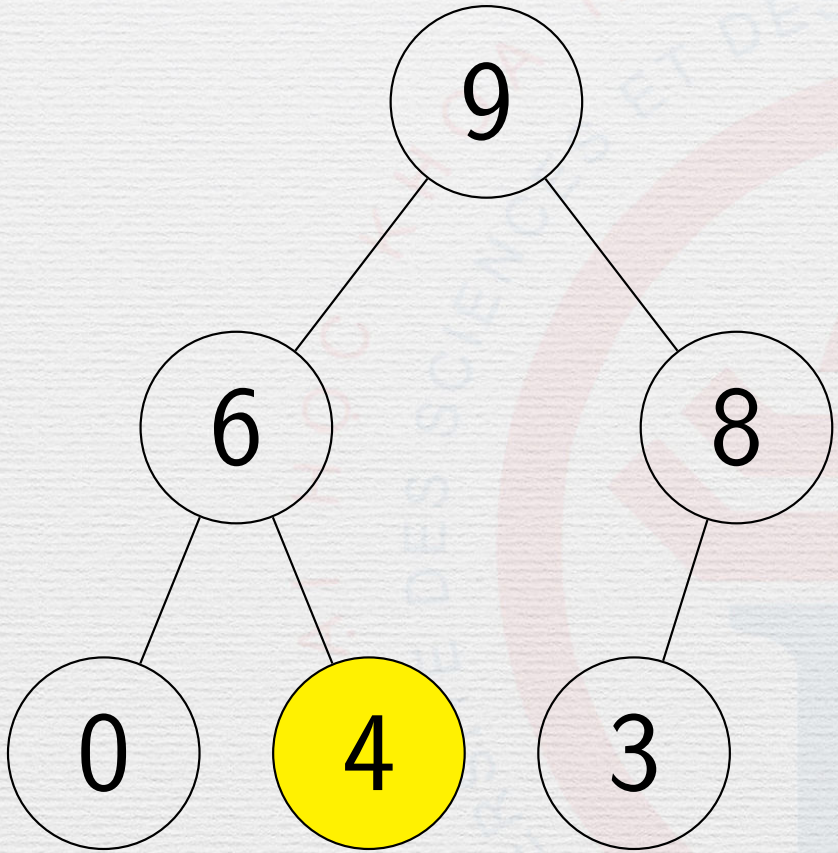
Heapification



Heapification



Heapification



The Loop Invariant

For $i = \lfloor n/2 \rfloor - 1$ downto 0, `heapify(i)`:

- **Initialization**: For every array, each node $\lfloor n/2 \rfloor .. n - 1$ is trival max-heap (leaf).

The Loop Invariant

For $i = \lfloor n/2 \rfloor - 1$ downto 0, `heapify(i)`:

- **Initialization**: For every array, each node $\lfloor n/2 \rfloor \,..\, n-1$ is trival max-heap (leaf).
- **Maintenance**: If nodes $i+1 \,..\, n-1$ are max-heaps, after `heapify(i)`, all nodes $i \,..\, n-1$ are max-heaps.

The Loop Invariant

For $i = \lfloor n/2 \rfloor - 1$ downto 0, `heapify(i)`:

- **Initialization**: For every array, each node $\lfloor n/2 \rfloor \,..\, n-1$ is trival max-heap (leaf).
- **Maintenance**: If nodes $i+1 \,..\, n-1$ are max-heaps, after `heapify(i)`, all nodes $i \,..\, n-1$ are max-heaps.
- **Termininination**: After `heapify(0)`, the whole array is a max-heap.

Heap Constructor

```
public class Heap<T extends Comparable<? super T>>
{
  public Heap(List<T> a)
  {
    list = a;
    size = a.size();
    for (int i = size >> 1; i-- > 0;)
      heapify(i);
  }
}
```

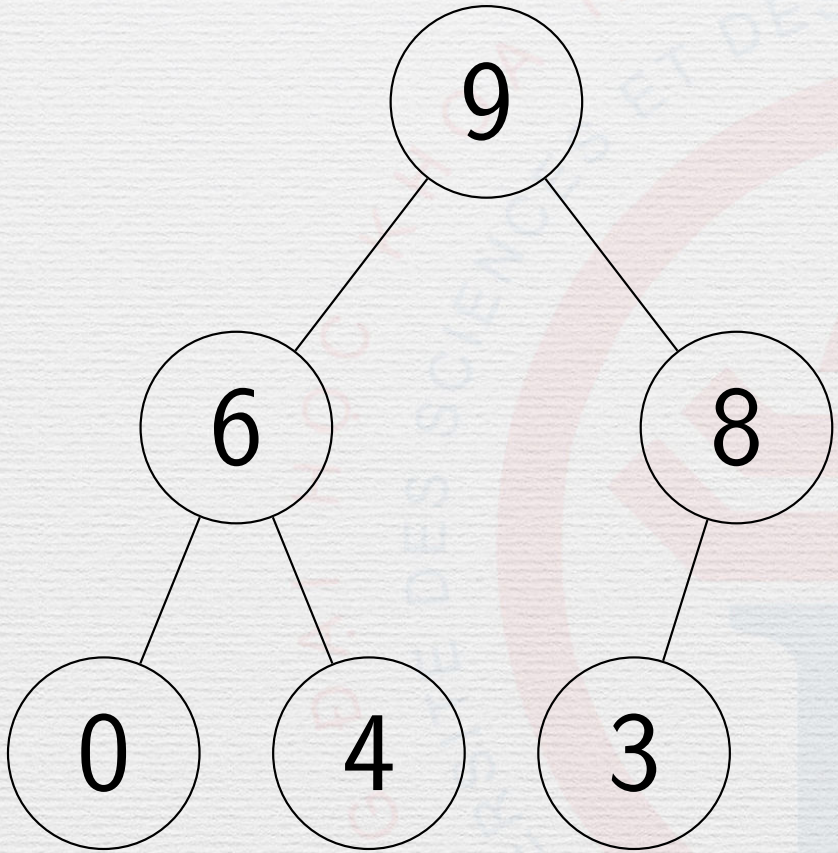
Maximum Selection

```
public class Heap<T extends Comparable<? super T>>
{
  public T pop() throws RuntimeException
  {
    if (size < 1)
      throw new RuntimeException("heap underflow");
    swap(list, 0, --size);
    heapify(0);
    return get(size);
  }
}
```

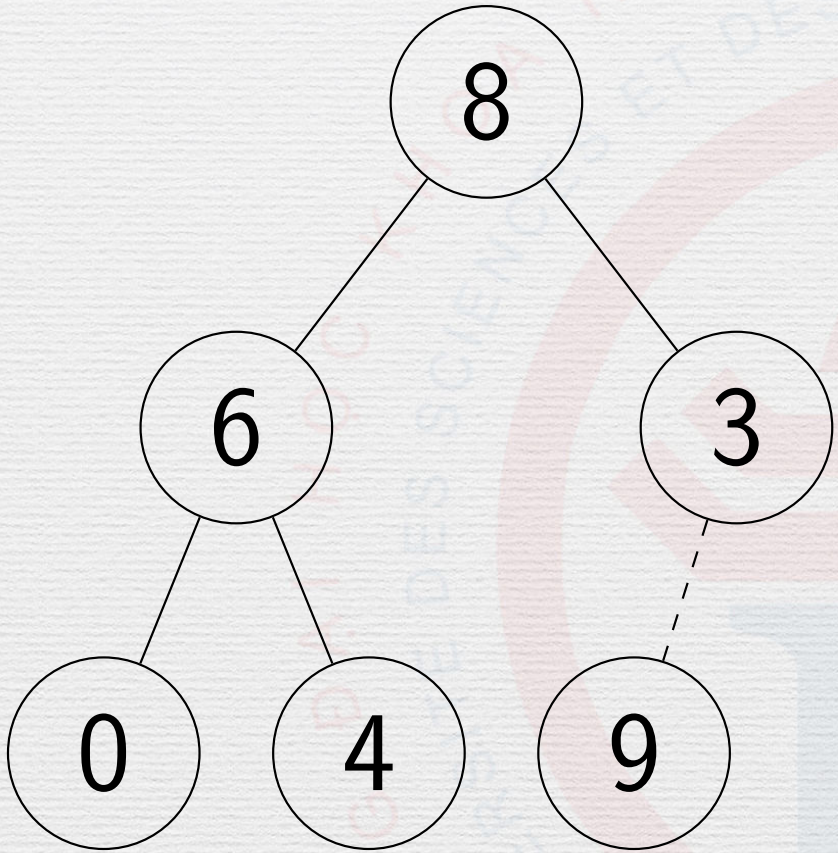
Heapsort Implementation

```
public class Sort
{
  public static <T extends Comparable<? super T>>
  void heap(List<T> list)
  {
    var heap = new Heap<T>(list);
    for (int i = 1; i < list.size(); ++i)
      heap.pop();
  }
}
```

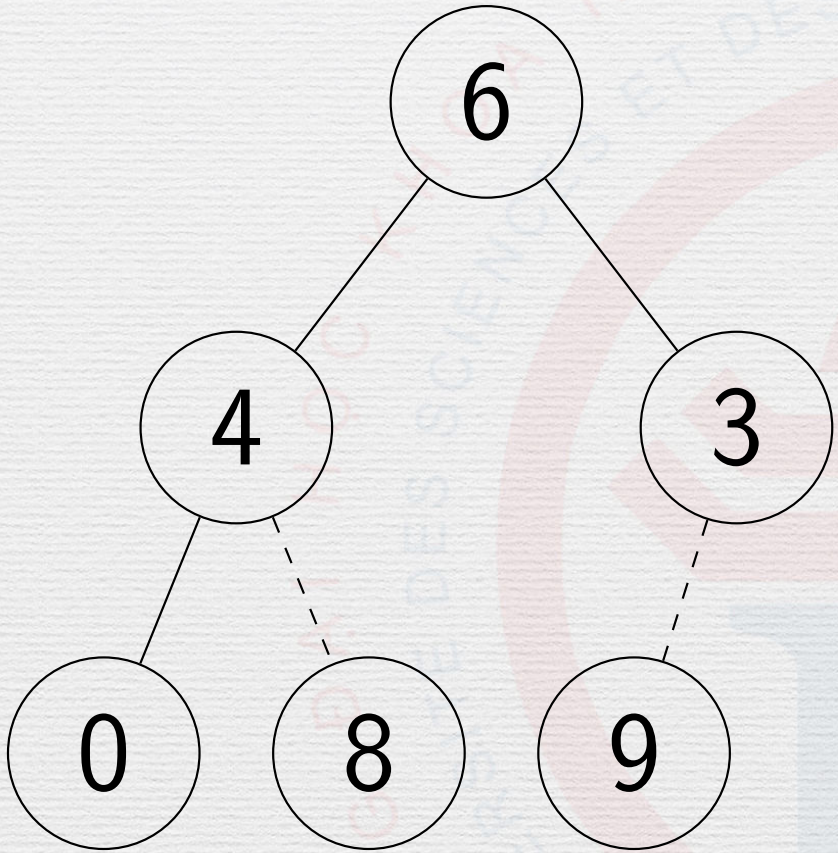
Sorting a Heap



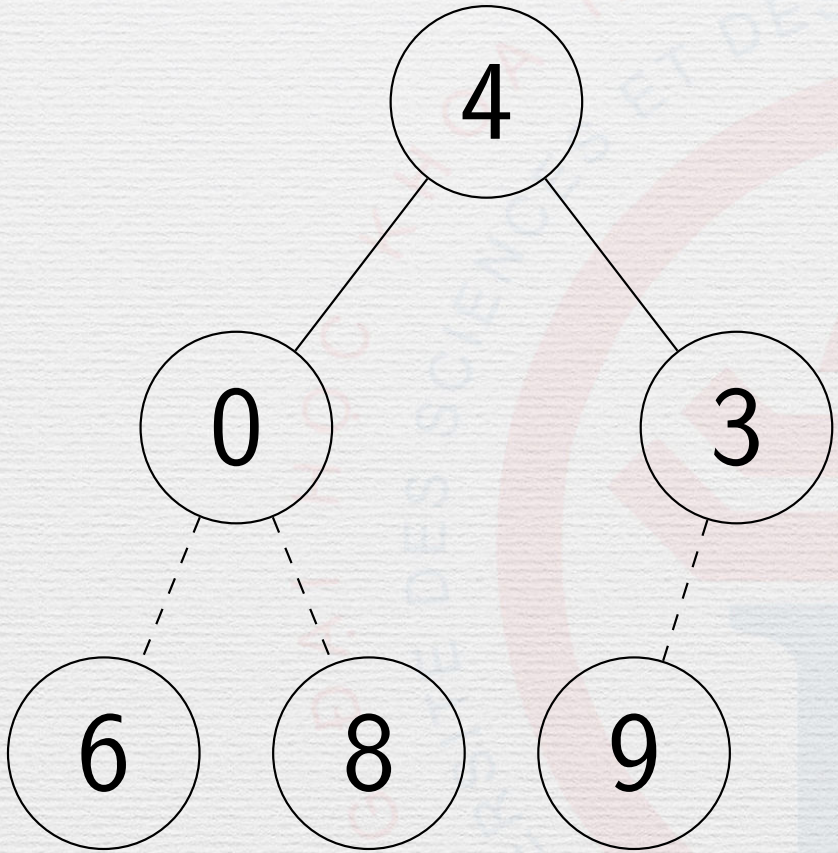
Sorting a Heap



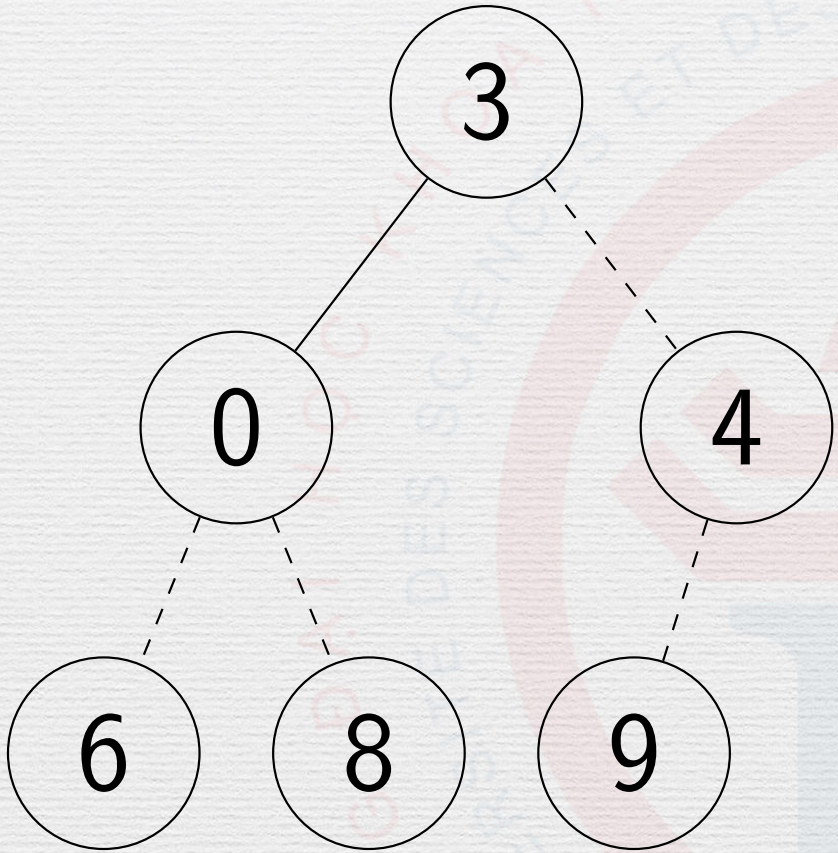
Sorting a Heap



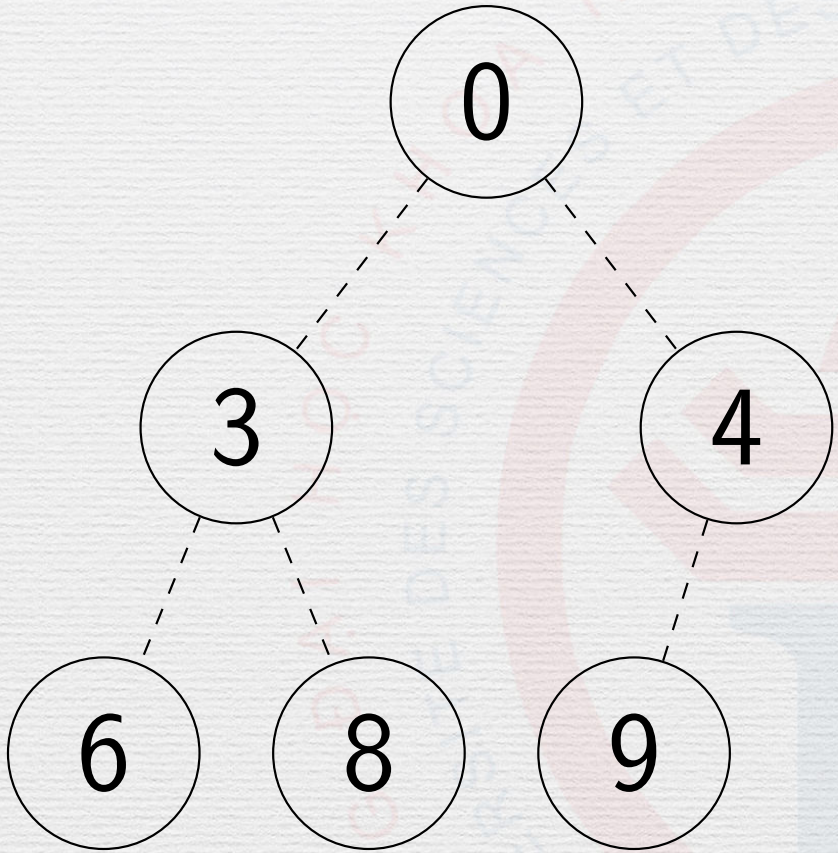
Sorting a Heap



Sorting a Heap



Sorting a Heap



Comparing

- $<$, $\leq$, $=$, $\geq$, $>$ or $\neq$?

Comparing

- $<, \leq, =, \geq, >$ or $\neq$?
- e.g. 420 > 69

Comparing

- $<, \leq, =, \geq, >$ or $\neq$?
- e.g. 420 > 69
- But "420" < "69"!

Comparing

- $<, \leq, =, \geq, >$ or $\neq$?
- e.g. 420 > 69
- But "420" < "69"!
- How do we sort any collection of data?

Comparable

- *Natural* increasing order

Comparable

- *Natural* increasing order
- Define `int compareTo(T other)`

Comparable

- *Natural* increasing order
- Define `int compareTo(T other)`
- Negative: less than; Zero: equal; Positve: greater than.

Example Element

```
public class Person implements Comparable<Person>
{
  private String name;
  private Integer age;
  private Character gender;

  public int compareTo(Person other)
  {
    return this.name.compareTo(other.name);
  }
}
```

Example Element (misc.)

```
public class Person implements Comparable<Person>
{
  public Person(String name, Integer age,
                Character gender)
  {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }

  public String toString()
  {
    return String.format("%s (%d%c)",
                         name, age, gender);
  }
}
```

Implementation

```
import static java.util.Collections.swap;

public class Sort
{
  public static <T extends Comparable<? super T>>
  void selection(List<T> list)
  {
    int i, j, m, n = list.size();
    for (i = 0; i < n; ++i)
      {
        for (m = j = i; j < n; ++j)
          if (list.get(j).compareTo(list.get(m)) < 0)
            m = j;
        swap(list, i, m);
      }
  }
}
```

Sorting People

```
var list = java.util.Arrays.asList(
  new Person("Mahathir Mohamad", 94, 'M'),
  new Person("Elizabeth II", 93, 'F'),
  new Person("Paul Biya", 86, 'M'),
  new Person("Michel Aoun", 84, 'M'),
  new Person("Mahmoud Abbas", 83, 'M'),
  new Person("Francis", 82, 'M'));
Sort.selection(list);
list.forEach(System.out::println);
```

Sort by Name Output

Elizabeth II (93F)
Francis (82M)
Mahathir Mohamad (94M)
Mahmoud Abbas (83M)
Michel Aoun (84M)
Paul Biya (86M)

Comparator

- How about reverse order?

Comparator

- How about reverse order?
- Sort by another *key*?

Comparator

- How about reverse order?
- Sort by another *key*?
- `compareTo` (or any other) method cannot be overriden without subclassing.

java.util.Comparator

- Define `int compare(T one, T another)`

java.util.Comparator

- Define `int compare(T one, T another)`
- Negative: less than; Zero: equal;
  Positve: greater than.

Refactored Selection Sort

```
public class Sort
{
  public static <T>
  void selection(List<T> list,
                 Comparator<T> comparator)
  {
    int i, j, m, n = list.size();
    for (i = 0; i < n; ++i)
      {
        for (m = j = i; j < n; ++j)
          if (comparator.compare(list.get(j),
                                 list.get(m)) < 0)
            m = j;
        swap(list, i, m);
      }
  }
}
```

Exposing Attributes

```
public class Person implements Comparable<Person>
{
  public String getName() { return name; }
  public Integer getAge() { return age; }
  public Character getGender() { return gender; }
}
```

Sorting by Age

```
Sort.heap(list, new Comparator<Person>()
{
  public int compare(Person a, Person b)
  {
    return a.getAge().compareTo(b.getAge());
  }
});
list.forEach(System.out::println);
```

Sorting by Age Output

Francis (82M)
Mahmoud Abbas (83M)
Michel Aoun (84M)
Paul Biya (86M)
Elizabeth II (93F)
Mahathir Mohamad (94M)

Backward Compatibility

```
public class Compare<T extends Comparable<? super T>>
implements Comparator<T>
{
  public int compare(T a, T b)
  {
    return a.compareTo(b);
  }
}

public class Sort
{
  public static <T extends Comparable<? super T>>
  void selection(List<T> list)
  {
    selection(list, new Compare<T>());
  }
}
```

Conclusion

Though the topic is more algorithmic than OOP:

- **Encapsulation**: Intuitive interface and concise code, e.g. binary search, heap.

Conclusion

Though the topic is more algorithmic than OOP:

- **Encapsulation**: Intuitive interface and concise code, e.g. binary search, heap.
- **Polymorphism**: Generic, convenient libraries, thus more *code reuse* and more effective development.

Conclusion

Though the topic is more algorithmic than OOP:

- **Encapsulation**: Intuitive interface and concise code, e.g. binary search, heap.
- **Polymorphism**: Generic, convenient libraries, thus more *code reuse* and more effective development.
- **Inheritance**: Extend objects' functionalities, hence even more generalization.

Conclusion

Though the topic is more algorithmic than OOP:

- **Encapsulation**: Intuitive interface and concise code, e.g. binary search, heap.
- **Polymorphism**: Generic, convenient libraries, thus more *code reuse* and more effective development.
- **Inheritance**: Extend objects' functionalities, hence even more generalization.
- However, shoving every self-contained function into a class is rather redundant.

Copying

- For the list of references, see our report.

Copying

- For the list of references, see our report.
- The report also contains more explainations and examples.

Copying

- For the list of references, see our report.
- The report also contains more explainations and examples.
-