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

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

CONTENT

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

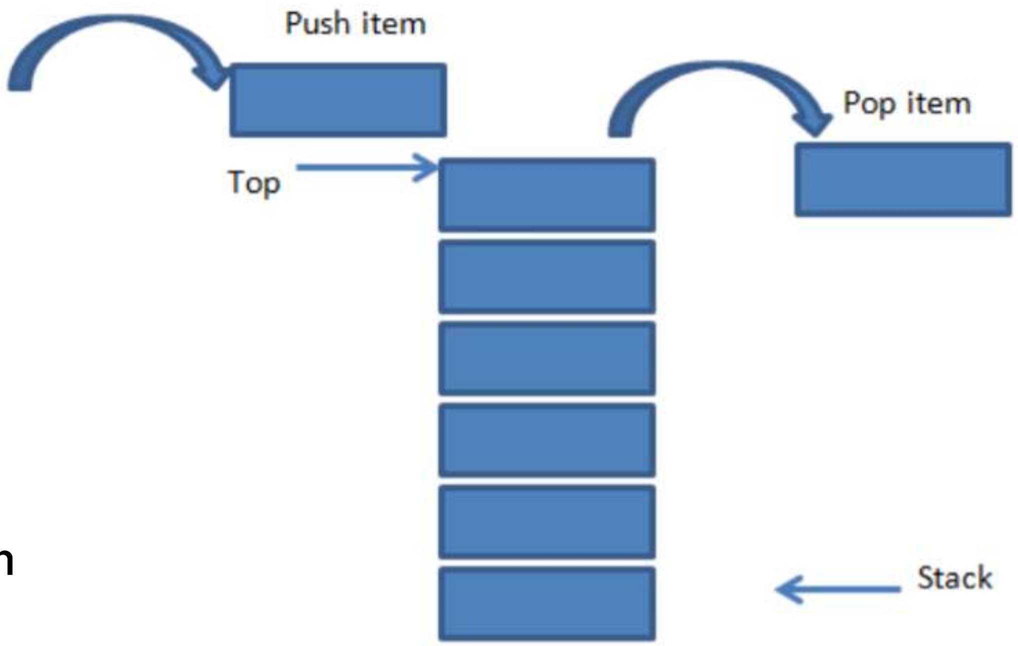
It is a linear data structure that follows the **LIFO** (Last-In-First-Out) principle

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)

```java
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)

```java
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

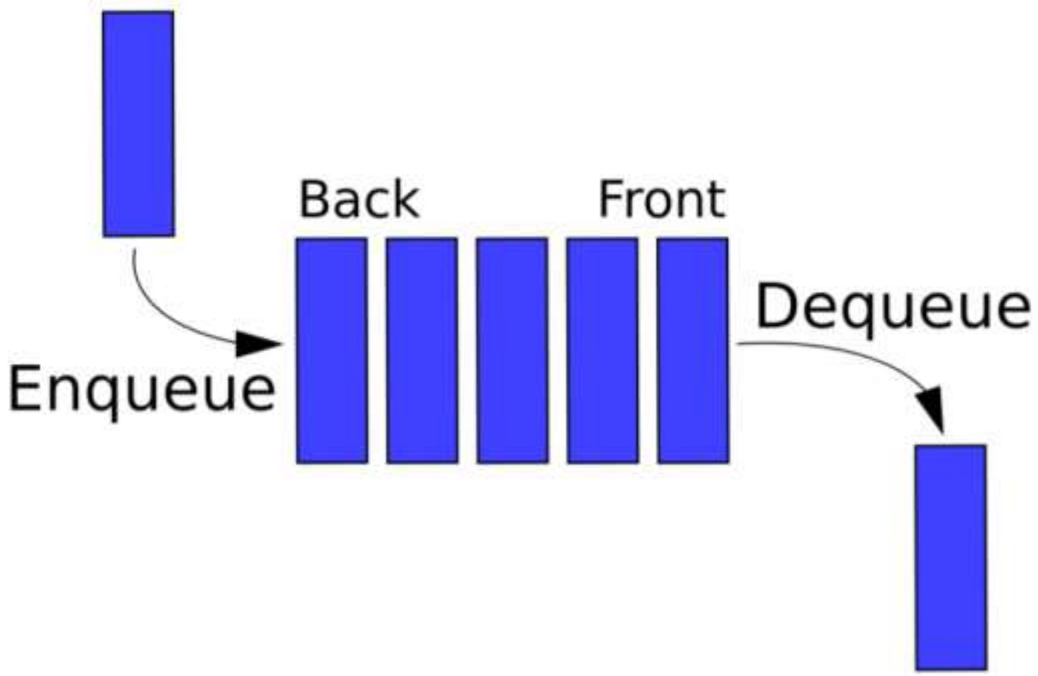
It is a linear data structure that follows the FIFO (First-In-First-Out) principle

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

It is a complete binary tree

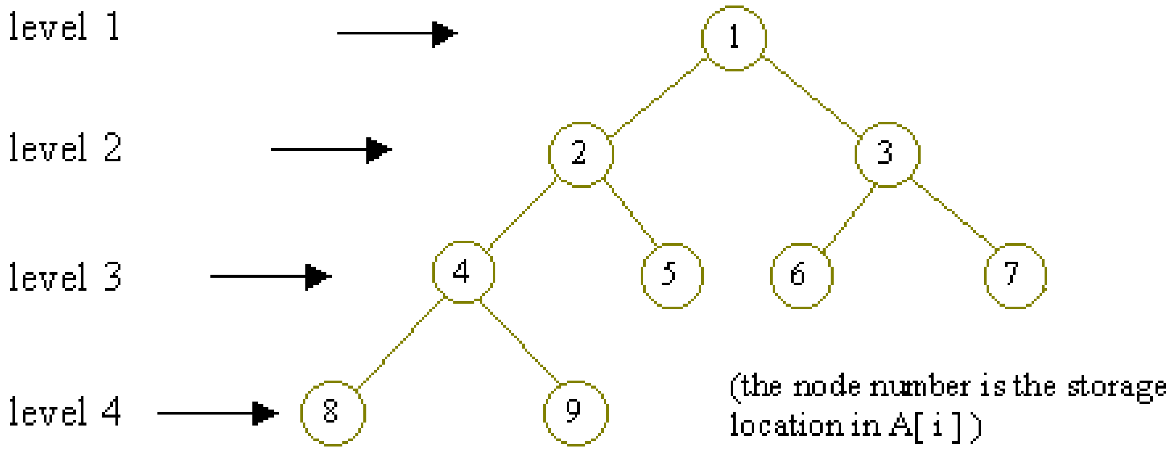
- Each level of the tree is filled, except the last one
- Each level is filled from left to right

Types:

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

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

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

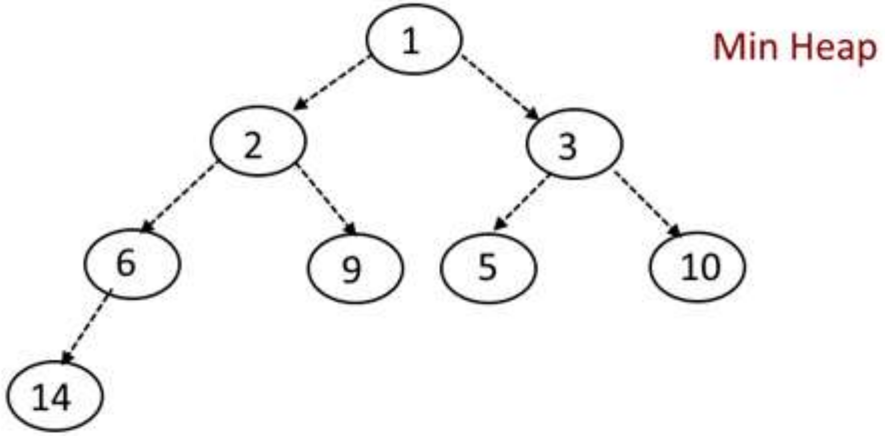
Extracting the *largest/smallest element from the heap (i.e. finding and removing it) takes O(log n) time

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].

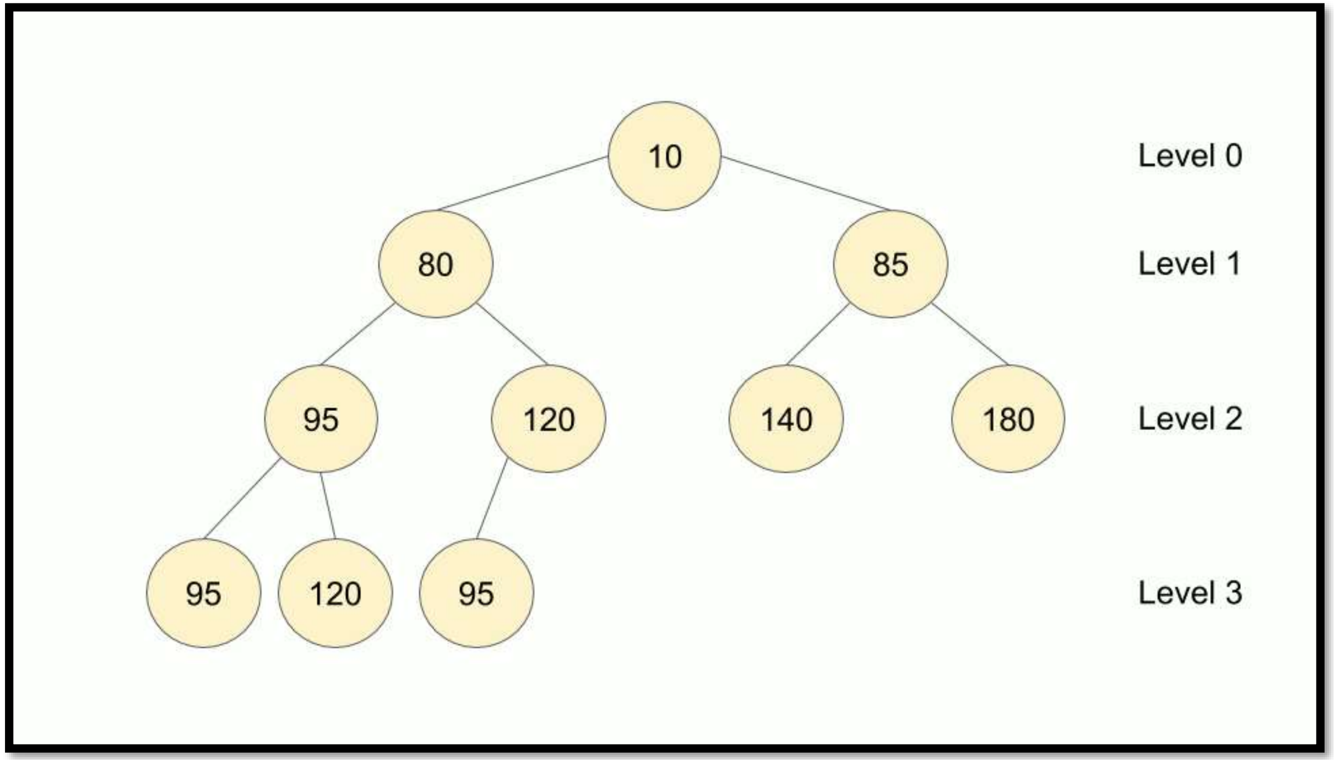
*largest/smallest – largest for Max Heap and smallest for Min Heap

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

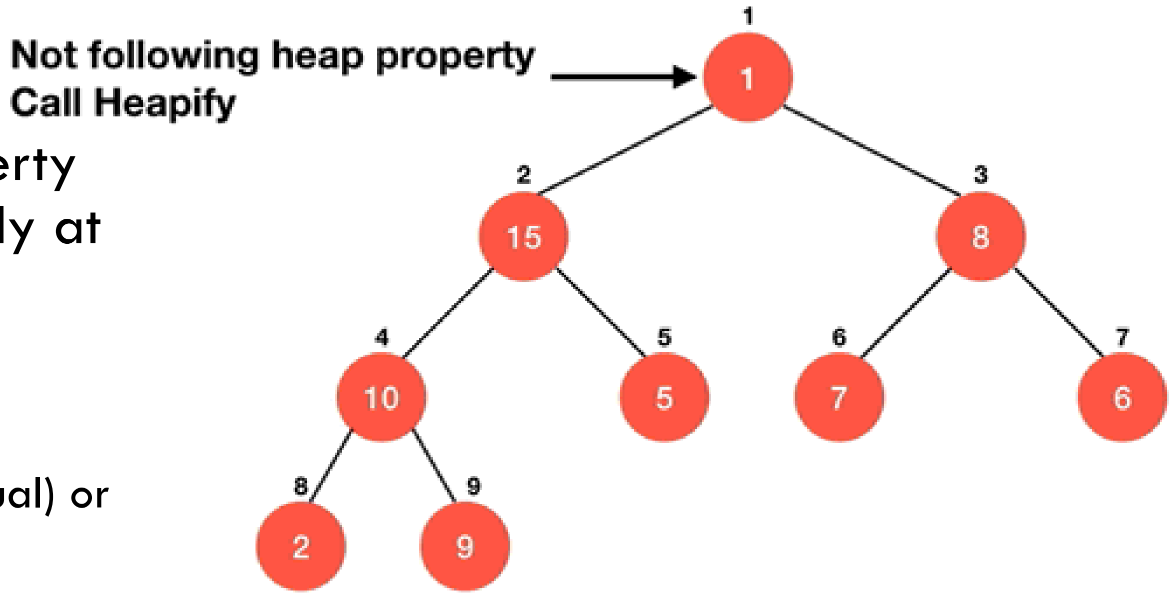


HEAP:HEAPIFY – O(LOG(N))

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



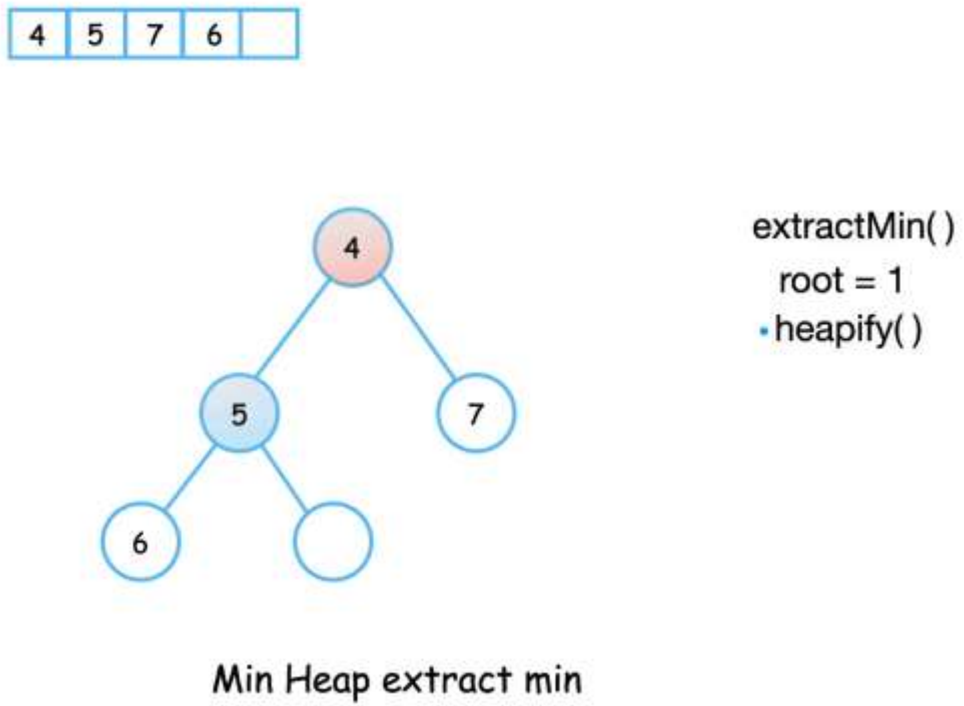
HEAP:EXTRACT_MIN – O(LOG(N))

Min Heap example

A root item is replaced with the last element

Recursive actions:

- Heapify(rootIndex)



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

GOOD LUCK!