Selection Sort

```
for i = START to (END - 1)
   minIndex = i;
   for j = i+1 to END
      if (data[j] < data[minIndex])
         minIndex = j
   tmp = data[minIndex]
   data[minIndex] = data[i]
   data[i] = tmp
```


Insertion Sort

Pseudo-code :

```
for i = (START+1) to END
   current = A[ i ]
   j = i - 1
      while j >=0 && A[j] > current
         A[j+1] = A[j]
         j = j – 1
   A[j+1] = current
```

Quick Sort Algorithm:

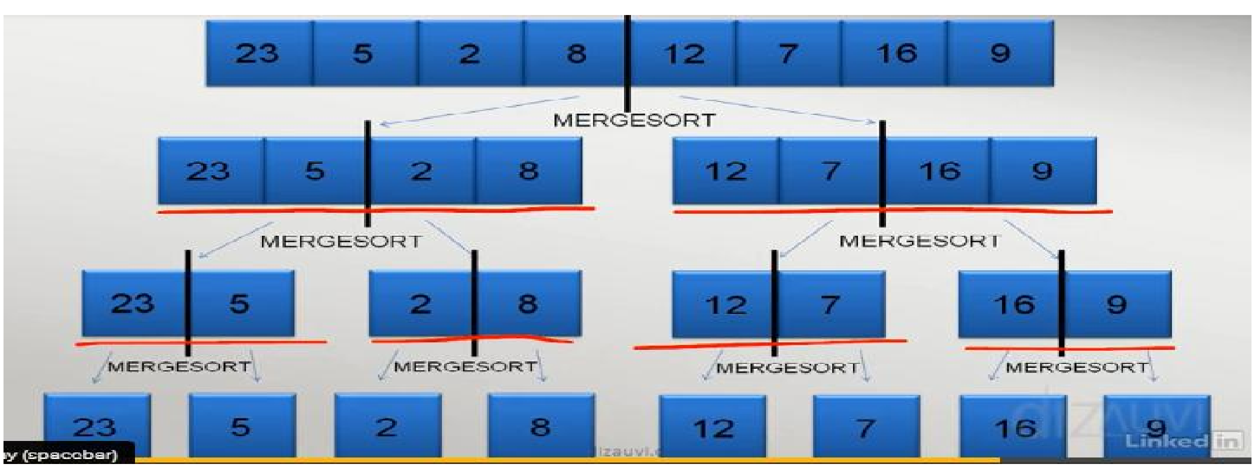
1. Average complexity: O (nlogn)
2. Worst Case: O(n2)

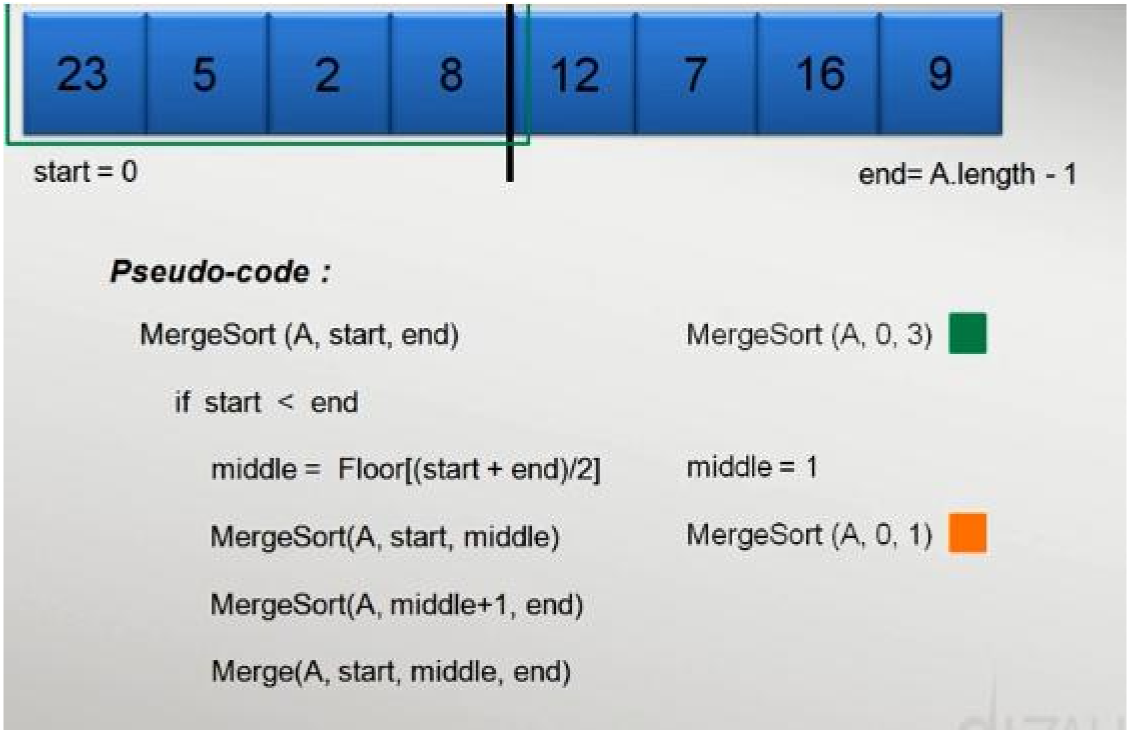
<u>Algorithm:</u>

```
Pseudo-code :
  QuickSort (A, start, end)
    if start < end
    pivot = Partition(A, start, end)
    QuickSort(A, start, pivot-1)
    QuickSort(A, pivot+1, end)
```

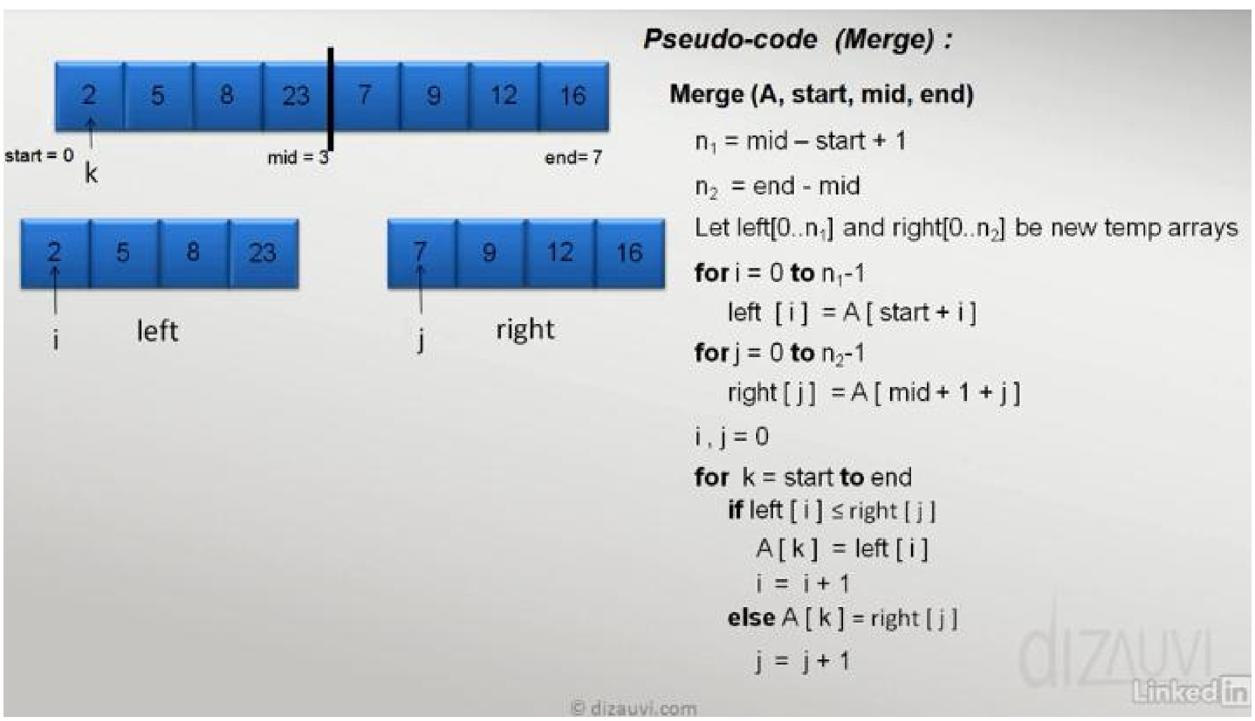
```
Pseudo-code :
  Partition (A, start, end)
    pivot = A[end]
    i = start
    for j = start to end-1
      if A[j] ≤ pivot
        exchange A[i] with A[j]
        i = i + 1
    exchange A[i] with A[end]
```

Merge Sort Algorithm:





23
5
2
8
12
7
16
9
start = 0
end= A.length - 1
Pseudo-code :
MergeSort (A, start, end)
if start < end
middle = Floor[(start + end)/2]
MergeSort(A, start, middle)
MergeSort(A, middle+1, end)
Merge(A, start, middle, end)
MergeSort (A, 0, 3)
middle = 1
MergeSort (A, 0, 1)



2
5
8
23
7
9
12
16
start = 0
k
mid = 3
end= 7
2
5
8
23
i
left
7
9
12
16
j
right
Pseudo-code (Merge) :
Merge (A, start, mid, end)
n_1 = mid – start + 1
n_2 = end - mid
Let left[0..n_1] and right[0..n_2] be new temp arrays
for i = 0 to n_1-1
left [i] = A [start + i]
for j = 0 to n_2-1
right [j] = A [mid + 1 + j]
i , j = 0
for k = start to end
if left [i] ≤ right [j]
A [k] = left [i]
i = i + 1
else A [k] = right [j]
j = j + 1
© dizauvi.com

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)

Binary Search Algorithm:

Recursive:

```
public static boolean binarySearchRecursive(int[] array, int x, int left, int right) {
    if (left > right) {
        return false;
    }

    int mid = (left + right) / 2;
    if (array[mid] == x) {
        return true;
    } else if (x < array[mid]) {
        return binarySearchRecursive(array, x, left, mid - 1);
    } else {
        return binarySearchRecursive(array, x, mid + 1, right);
    }
}

public static boolean binarySearchRecursive(int[] array, int x) {
    return binarySearchRecursive(array, x, 0, array.length - 1);
}
```

Iterative:

```
public static boolean binarySearchIterative(int[] array, int x) {
    int left = 0;
    int right = array.length - 1;
    while (left <= right) {
        int mid = left + ((right - left) / 2);
        if (array[mid] == x) {
            return true;
        } else if (x < array[mid]) {
            right = mid - 1;
            //return binarySearchRecursive(array, x, left, mid - 1);
        } else {
            left = mid + 1;
            //return binarySearchRecursive(array, x, mid + 1, right);
        }
    }
    return false;
```

Stack:

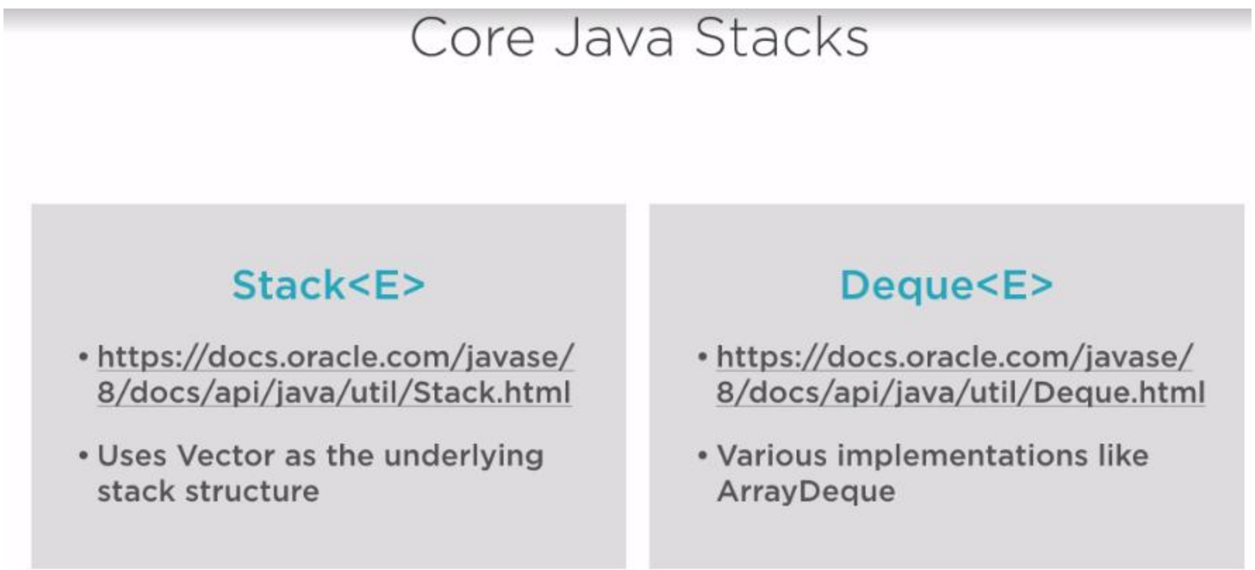
```java
public class BasicStack<X> {
    private X [] data;
    private int stackPointer;

    public BasicStack() {
        data = (X[]) new Object[1000];
        stackPointer = 0;
    }

    public void push(X newItem) {
        data[stackPointer++] = newItem;
    }

    public X pop() {
        if(stackPointer == 0) {
            throw new IllegalStateException("No more items on the stack");
        }

        return data[--stackPointer];
    }
}
```



Queue:

```java
public class BasicQueue<X> {
    private X[] data;
    private int front;
    private int end;


    public BasicQueue() {
        this(1000);
    }

    public BasicQueue(int size) {
        this.front = -1;
        this.end = -1;
        data = (X[])new Object[size];
    }

    public int size() {
        //if the queue is empty return 0
        if(front == -1 && end == -1) {
            return 0;
        }
        //otherwise we add one to get the inclusive subtraction value rather than excluding the
        else {
            return end - front + 1;
        }
    }

    public void enQueue(X item) {
        //first see if the queue is full
        if((end + 1) % data.length == front) {
            throw new IllegalStateException("The Queue is full!");
        }
        //otherwise check to see if any items have been added to the queue yet
        else if(size() == 0) {
            front++;
            end++;
            data[end] = item;
        }
        //otherwise add the item to the end of the queue
        else {
            end++;
            data[end] = item;
        }
    }

    public X deQueue() {
        X item = null;

        //if the queue is empty we can't dequeue anything
        if(size() == 0) {
            throw new IllegalStateException("Can't dequeue because the queue is empty!");
        }
        //otherwise if this is the last item on the queue, the queue needs to get reset to empt
        else if(front == end) {
            item = data[front];
            front = -1;
            end = -1;
        }
        //otherwise grab the front of the queue, return it and adjust the front pointer
        else {
            item = data[front];
            front++;
        }

        return item;
    }
```

LinkedList:

```java
public class BasicLinkedList<X> {
    private Node first;
    private Node last;

    public BasicLinkedList() {
        first = null;
        last = null;
    }

    private class Node {
        private Node nextNode;
        private X nodeItem;

        public Node(X item) {
            this.nextNode = null;
            this.nodeItem = item;
        }

        public void setNextNode(Node nextNode) {
            this.nextNode = nextNode;
        }

        public Node getNextNode() {
            return nextNode;
        }

        public X getNodeItem() {
            return nodeItem;
```

Add Element To LinkedList:

```java
public class BasicLinkedList<X> {
    private Node first;
    private Node last;
    private int nodeCount;

    public BasicLinkedList() {
        first = null;
        last = null;
        nodeCount = 0;
    }

    public void add(X item) {
        //this condition means we are adding something for the first time.
        if(first == null) {
            first = new Node(item);
            last = first;
        }
        //otherwise, we want to grab the last node and update it's value
        else {
            Node newLastNode = new Node(item);
            last.setNextNode(newLastNode);
            last = newLastNode;
        }
        nodeCount++;
    }
```

Remove Item from LinkedList

```
public X remove() {
    if(first == null) {
        throw new IllegalStateException("The LinkedList is empty and there are no items to
    }

    X nodeItem = first.getNodeItem();

    //now update the first pointer and throw away the old first
    first = first.getNextNode();
    nodeCount--;
    return nodeItem;
}
```

Insert Item into LinkedList

```
public void insert(X item, int position) {
    if(size() < position) {
        throw new IllegalStateException("The LinkedList is smaller than the position you ar
    }

    Node currentNode = first;

    //start at 1 because we are already on the first node
    for(int i = 1; i < position && currentNode != null; i++) {
        currentNode = currentNode.getNextNode();
    }

    //severs the link chain and reconnects with the new node
    Node newNode = new Node(item);
    Node nextNode = currentNode.getNextNode();
    currentNode.setNextNode(newNode);
    newNode.setNextNode(nextNode);
    nodeCount++;
}
```

Remove Item from LinkedList

```
public X removeAt(int position) {
    if(first == null) {
        throw new IllegalStateException("The LinkedList is empty and there are no items to remove");
    }

    Node currentNode = first;
    Node prevNode = first;

    //start at 1 because we are already on the first node
    for(int i = 1; i < position && currentNode != null; i++) {
        prevNode = currentNode;
        currentNode = currentNode.getNextNode();
    }

    //now update the pointers and throw away the old first
    X nodeItem = currentNode.getNodeItem();
    prevNode.setNextNode(currentNode.getNextNode());
    nodeCount--;
    return nodeItem;

}
```

Hash Data Structure:

```
public class BasicHashTable<X, Y> {
    private HashEntry[] data;
    private int capacity;

    public BasicHashTable(int tableSize) {
        this.capacity = tableSize;
        this.data = new HashEntry[this.capacity];
    }
```

```
private class HashEntry<X, Y> {
    private X key;
    private Y value;

    public HashEntry(X key, Y value) {
        this.key = key;
        this.value = value;
    }

    public X getKey() {
        return key;
    }
    public void setKey(X key) {
        this.key = key;
    }
    public Y getValue() {
        return value;
    }
    public void setValue(Y value) {
        this.value = value;
    }
```

Create Size() of Hash Table:

```
public class BasicHashTable<X, Y> {
    private HashEntry[] data;
    private int capacity;
    private int size;

    public BasicHashTable(int tableSize) {
        this.capacity = tableSize;
        this.data = new HashEntry[this.capacity];
        this.size = 0;
    }

    public int size() {
        return this.size;
    }
```

Get value by Key from HashTable:

```
public Y get(X key) {
    int hash = calculateHash(key);

    // if we don't have anything for the given key, just return null
    if(data[hash] == null) {
        return null;
    }
    // otherwise get the hashentry for the key and return its value
    else {
        return (Y)data[hash].getValue();
    }
}
```

```java
public Y get(X key) {
}

public int size() {
    return this.size;
}

private int calculateHash(X key) {
    int hash = (key.hashCode() % this.capacity);
    // this is necessary to deal with collisions
    while (data[hash] != null && !data[hash].getKey().equals(key)) {
        hash = (hash + 1) % this.capacity;
    }
    return hash;
}
```

Put Value into HashTable:

```java
public void put(X key, Y value) {
    int hash = calculateHash(key);

    data[hash] = new HashEntry<X, Y>(key, value);
    size++;
}
```

Binary Tree:

```java
public class BasicBinaryTree<X extends Comparable<X>> {
    private Node root;

    public BasicBinaryTree() {
        this.root = null;
    }

    private class Node {
        private Node left;
        private Node right;
        private Node parent;
        private X item;

        public Node(X item) {
            this.item = item;
            this.left = null;
            this.right = null;
            this.parent = null;
        }

        public Node getLeft() {
            return left;
        }

        public void setLeft(Node left) {
            this.left = left;
        }
```

Other getter & setter

Size() of Binary Tree:

```java
public class BasicBinaryTree<X extends Comparable<X>> {
    private Node root;
    private int size;

    public BasicBinaryTree() {
        this.root = null;
    }

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

Add Note into Binary Tree

```java
    public void add(X item) {
        Node node = new Node(item);

        //if this is the first item in the tree, set it as root
        if(root == null) {
            this.root = node;
            System.out.println("Set root: " + node.getItem());
            this.size++;
        }
        //otherwise we need to insert the item into the tree using the binary tree insert algorithm
        else {
            insert(this.root, node);
        }

    }

    private void insert(Node parent, Node child) {
        //if the child is less than the parent, it goes on the left
        if(child.getItem().compareTo(parent.getItem()) < 0) {
            //if the left node is null, we've found our spot
            if(parent.getLeft() == null) {
                parent.setLeft(child);
                child.setParent(parent);
                this.size++;
            }
            //otherwise we need to call insert again and test for left or right (recursion)
            else {
                insert(parent.getLeft(), child);
            }
        }
        //if the child is greater than the parent, it goes on the right
        else if(child.getItem().compareTo(parent.getItem()) > 0) {
            //if the right node is null, we've found our spot
            if(parent.getRight() == null) {
                parent.setRight(child);
                child.setParent(parent);
                this.size++;
            }
            //otherwise we need to call insert again and test for left or right (recursion)
            else {
                insert(parent.getRight(), child);
            }
        }
    }
```