DataStructures & Algorithms

By

Praveen Naga

Blog: https://praveennagatech.blogspot.com
Facebook Group: https://www.facebook.com/groups/268426377837151
Github repo: https://github.com/praveennaga
Email: opraveennaga@gmail.com

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1

What is a Datastructure?
It is a way to organize the data and in a way that enables it to be processed in an efficient time.

What is Algorithm?
It is a set of rules to be followed to solve a problem.

Types of Data Structure

1. **Physical Data Structure**
   a) Array
   b) Linked List
2. **Logical Data Structure**

a) Stack
b) Queue
c) Tree
d) Graph

Recursion

Let's consider below example to understand the recursion and then we will discuss the properties of recursion accordingly.

For example,
Let's say we need to search for value in binary tree

Search(root,valueToSearch)
If(root equals null) return null
else if(root.value equals valueToSearch) return root
else if(valueToSearch < root.value) search(root.left,valueToSearch)
else search(root.right,valueToSearch)

Properties of Recursion

a) Same operations is performed multiple times with different inputs.
b) In every step we try to make the problem smaller.
c) We mandatorily need to have a base condition(highlighted in bold in the above example) which tells system when to stop the recursion.

Why we need Recursion

- Because it makes the code easy to write (**compared to iterative**) whenever a given problem can be broken down into **similar** sub-problem.
- Because it is heavily used in Data Structures like Tree, Graph etc.

- It is heavily used in techniques like ‘Divide and Conquer’,’ Greedy’, ’Dynamic Programming’.

Format of a Recursive Function

- **Recursive Case:** Case where the function recur.
- **Base Case**: Case where the function does not recur.

Example,

```
SampleRecursion(parameter){
 If(base case is satisfied){
   return some base case value;
  }else{
  SampleRecursion(modified parameter){
 }
}
```

Recrusion mainly uses stack memory as you know

Let’s see some recursion examples

A. Factorial

```
public class Factorial {

    public static void main(String args[]) {
        Scanner s= new Scanner(System.in);
        System.out.println("Enter number for finding factorial of it");
        int num=s.nextInt();
        System.out.println(factorial(num));
        s.close();
    }

   private static int factorial(int num) {
       if(num==0) {
           return 1;
       }else {
           return num*factorial(num-1);
       }
   }
}
```

B. Fibonacci Series (0 1 1 2 3 5 8 13 21 …..)

```
//0 1 1 2 3 5 8 13 21....
public class Fibonacci {
    public static void main(String args[]) throws Exception {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter number for finding factorial of it");
        int num = s.nextInt();
        System.out.println("Number at Position "+num+ " is "+ fibonacci(num));
        s.close();
    }

    private static int fibonacci(int num) throws Exception {

        if (num < 1) {
            throw new Exception("Please provide number greater than zero");
        }

        if (num <= 1 || num <= 2) {
            return num - 1;
        } else {
            return fibonacci(num - 1) + fibonacci(num - 2);
        }

    }
}
```

Recursive vs Iteration

- Space Efficient? Recursion –No, Iteration – Yes
- Time Efficient? Recursion –No, Iteration – Yes
- Ease of Code (to solve sub-problems)? Recursion –Yes, Iteration – No

When to use/Avoid Recursion?

When to use

- When we can easily breakdown a problem into similar sub-problem
- When we are ok with extra overhead(both time and space) that comes with it
- When we need a quick working solution instead of efficient one

When to Avoid

- If the response to any of the above statement is NO, we should not go with recursion.

Practical use of Recursion

- Stack

- Tree- Traversal/Searching/Insertion/Deletion
- Sorting – Quick Sort and Merge Sort
- Divide and Conquer
- Dynamic Programming
- etc…….

Algorithm runtime time Analysis

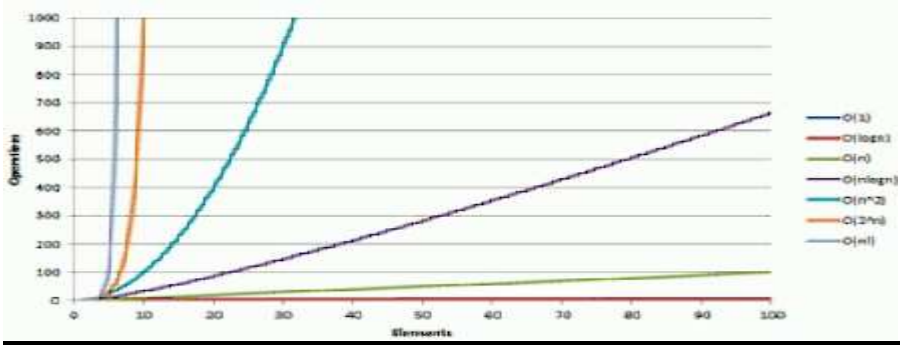
It is used to measure the efficiency of a program.

Notations used in Algorithm runtime analysis are

1. Omega
    - This notation gives the tighter lower bound of a given algorithm
    - Which means for any given input, running time of a given algorithm will not be less than given time.
2. Big O(O)
    - This notation gives the tighter upper bound of a given algorithm
    - Which means for any given input, running time of a given algorithm will not be greater than given time.
3. Theta
    - This notation decides whether the lower and upper bound of a given algorithm is same or not.
    - Which means for any given input, running time of a given algorithm will on an average be equal to given time.

Algorithm runtime complexity

Time Complexity	Name	Example
O(1)	Constant	Adding and element at front of linked list
O(log n)	Logarithmic	Finding an element in sorted array
O(n)	Linear	Findin an elemnet in unsorted array
O(n log n)	Linear Logarithmic	Merge Sort
$O(n^2)$	Quadratic	Shortest path between 2 nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	Tower of Hanoi Problem



How to Calculate Algorithm Time complexity

Find the biggest number in an array

```
public class FindMaxNumberInArray {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32, 12, 98, 2 };
        int max = num[0]; // O(1)

        for (int i = 1; i < num.length - 1; i++) { // O(n)
            if (num[i] > max) { // O(1)
                max = num[i]; // O(1)
            }
        }
        System.out.println("Max Element is " + max);

        // Using recursive
        System.out.println(findMaxNumberInArray(num, num.length - 1));
    }

    // Time Complexity : O(1)+O(n)+O(1) = O(n)

    public static int findMaxNumberInArray(int[] num, int length) {
        int max = Integer.MIN_VALUE;

        if (length == -1) {
            return max;
        } else {

            if (num[length] > max)
                max = num[length];
            return findMaxNumberInArray(num, length - 1);

        }
    }
}
```

Arrays

Arrays <- Random Access

An array is group of like typed variables and of fixed size.
for example, int[] num = new int[10];

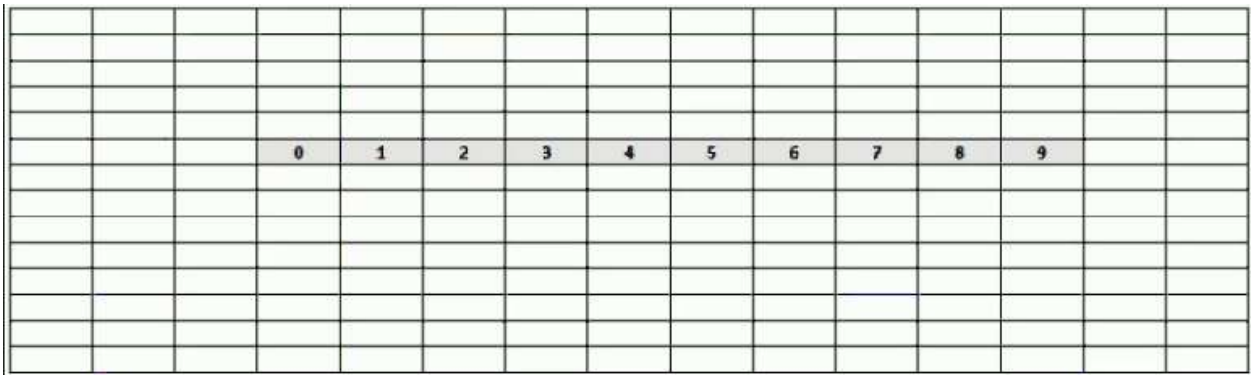
- For comparing two arrays we can use Arrays.equals(a1,a2);
- For comparing two arrays of arrays we can use Arrays.deepEquals(a1,a2);
- To print an array we can use Arrays.toString(a);
- To sort an array we can use Arrays.sort(a);
- Binary search in an sorted array we can use Arrays.binarySearch(int[] a,int key) where key will be index.
- To copy an array we can use Arrays.copyof(int[] original,int newLength)
- To convert an array to List we can use Arrays.asList(a)

Properties of Array

- Array can store data of specific datatype
- It has contiguous memory location
- Every cell of an array has unique index
- Index starts with zero
- Size of array needs to be specified mandatorily and cannot be modified.

How is an Array represented in Memory (RAM)?

It stores the cells data in contiguous memory location.



Limitation of Arrays

1.Need to know size in advance. Too small size cannot add more elements where as Too large size is waste of space.
2.We cannot insert/remove the elements infront of others except at the end.
3.some operations like search will be slow.

How many times of Arrays available?

1. Single Dimensional Array (1D)
2. Two Dimensional Array(2D)
3. Multi Dimensional Array(3D,4D etc)

Single Dimensional Array(1D)

For example,

```
public class SingleDimensionalArray {

    public static void main(String[] args) {
        // declaration and creation
        int[] a = new int[8];
        // As there are no values initialized, default value of int is 0 and for string it is null
        System.out.println(a);
        //retrieval
        for(int i: a) {
            System.out.println(i);
        }
        // declaration,creation and initialization
        int[] num= {90,24,46,35,32,12,98,2};
        System.out.println(num);
        int sum=0;
        // retrieval
        for(int i: num) {
            System.out.println(i);
            sum+=i;
        }
        // sum of elements present in single dimensional Array
        System.out.println("sum is "+ sum);
    }
}
```

Output

```
[I@15db9742
0
0
0
0
0
0
0
0
[I@6d06d69c
90
24
46
35
32
12
98
2
sum is 339
```

Two Dimensional Array

For example,

```
public class TwoDimensionalArray {
    public static void main(String[] args) {
        // declaration,creation and initialization
        int[][] num = { { 90, 24, 46 }, { 35, 32, 12 } };
        System.out.println(num);
        int sum = 0;
        // Retrieval
        for (int i = 0; i < num.length; i++) {
            for (int j = 0; j < num[i].length; j++) {
                System.out.print(num[i][j] + " ");
                sum += num[i][j];
            }
            System.out.println();
        }
        // Sum of all elements present in Two Dimensional Array
        System.out.println("sum is " + sum);

        // The above 2 Dimensional Array is also called Matrix Array as the rows and
        // columns are fixed

    }
}
```

Output

```
[[I@15db9742
90 24 46
35 32 12
sum is 239
```

The above 2 Dimensional Array is also called **Matrix Array** as the rows and columns are fixed.

Jagged Array

Array which has rows as fixed but columns are variable

For example,

```
public class JaggedArray {
    public static void main(String[] args) {
        // Lets see Jagged Array, the rows are fixed but column is variable

        int[][] num1 = new int[5][];
        System.out.println(num1);
        for (int i = 0; i < num1.length; i++) {
            num1[i] = new int[i + 1];
        }
        // Initializing array
        int count = 0;
        for (int i = 0; i < num1.length; i++) {
            for (int j = 0; j < num1[i].length; j++) {
                num1[i][j] = count++;
            }
        }

        // Displaying the values of 2D Jagged array
        for (int i = 0; i < num1.length; i++) {
            for (int j = 0; j < num1[i].length; j++) {
                System.out.print(num1[i][j] + " ");
            }
            System.out.println();
        }

    }
}
```

Output

```
[[I@15db9742
0
1 2
3 4 5
6 7 8 9
10 11 12 13 14
```

MultiDimensional Array

Lets see three dimensional array.

For example,

```
public class ThreeDimensionalArray {

    public static void main(String[] args) {
        // declaration,creation and intialization
        int[][][] num = { { { 90, 24, 46 }, { 35, 32, 12 }, { 98, 2 } } };
        System.out.println(num);
        int sum=0;
        // Retrieval
        for (int i = 0; i < num.length; i++) {
            for (int j = 0; j < num[i].length; j++) {
                for (int k = 0; k < num[i][j].length; k++) {
                    System.out.print(num[i][j][k] + " ");
                    sum+=num[i][j][k];
                }
                System.out.println();
            }
        }
        // sum of all the elements present in 3 Dimensional Array
        System.out.println("sum is "+sum);
    }

}
```

Output

```
[[[I@15db9742
90 24 46
35 32 12
98 2
sum is 339
```

Linked List

A linked list is a liner data structure where each element is a separate object. Each element (node) of a list comprises of two items i.e. data and a reference to next node.

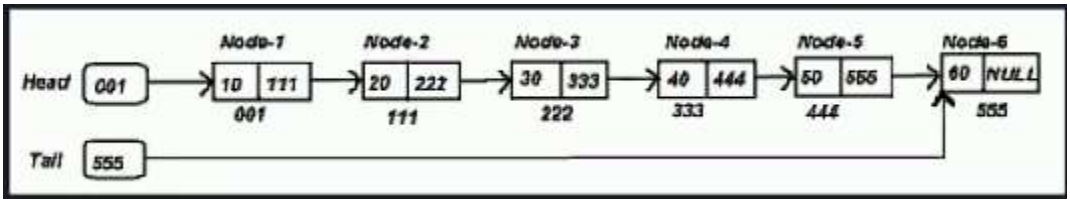
The most powerful feature of linked list is that it is of variable size.

Linked List Vs Array

1. Separate object – Linked List –Yes,Array-No
2. Variable size – Linked List –Yes, Array-No

3. Random Access – Linked List –No, Array-Yes

Linked List Components



Node: Contains both data and reference to next node

Head: Reference to first node in the list

Tail: Reference to last node in the list

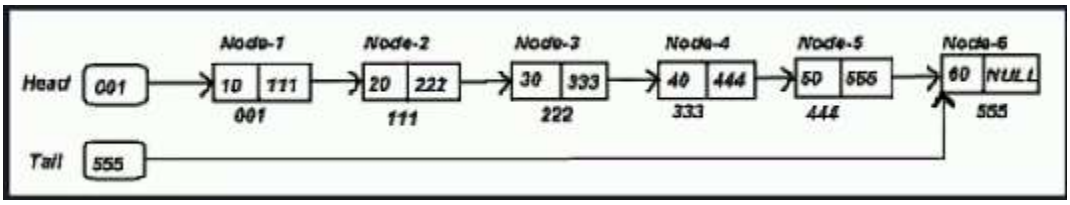
Types of Linked List

1. Single Linked List
2. Circular Single Linked List
3. Double Linked List
4. Circular Double Linked List

Single Linked List

In a single linked list, each node stores the data of the node and reference of the next node in the list. It doesn't store the reference of previous node.

It is the most basic form of linked list which give the flexibility to add/remove nodes at runtime.

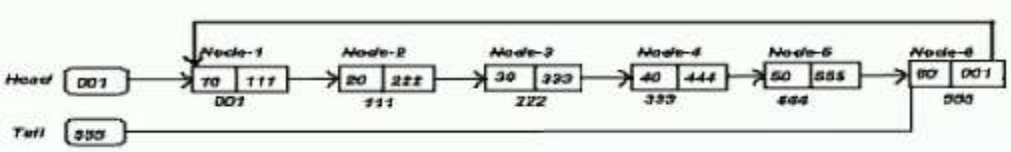


For example, single linked list fails for multiplayer board game if we are tracking players turn in linked list.

Circular Single Linked List

In case of circular single linked list, the only change that occurs is that the end of the given list is linked back to the front.

For example, multiplayer board game if we are tracking players turn in linked list.

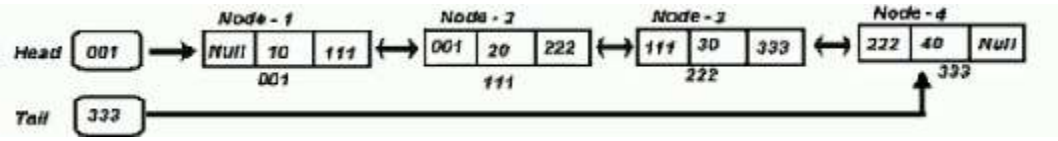


Double Linked List

In case of double linked list each node contains previous and next node references.

This is useful when we want to move in both direction depending on requirement.

For example, Music player has next and previous buttons.

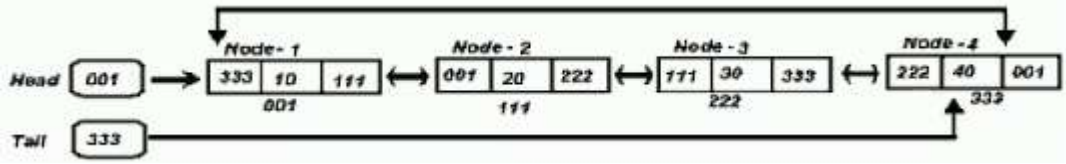


Circular Double Linked List

In case of circular double single linked list, the only change that occurs is that the end of the given list is linked back to the front of the list and vice versa.

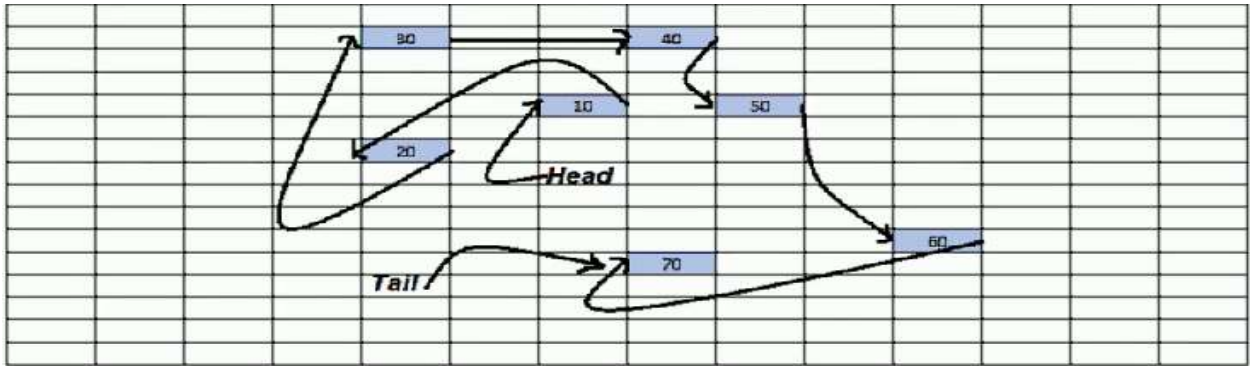
This is useful when we want to loop through the list indefinitely until the list exists. We want to move both forward and backward.

For example, ALT+TAB button in Windows.



How is Linked List stored in memory(RAM)?

As you see the below diagram, In Linked List cells will not be stored in contiguous memory location like an Array and it stores in uneven location.



Common Operations in Linked List

1. Creation of Linked List
2. Insertion of Linked List
3. Traversal of Linked List
4. Searching in a Linked List
5. Deletion of a node from Linked List
6. Deletion of Linked List

SingleLinkedList

```
public class SingleLinkedList {
    private SingleNode head;
    private SingleNode tail;
    private int size;// denotes size of list


    public SingleNode createSingleLinkedList(int nodeValue) {
        head = new SingleNode();
        SingleNode node = new SingleNode();
        node.setValue(nodeValue);
        node.setNext(null);
        head = node;
        tail = node;
        size = 1;// size =1
        return head;
    }

    public SingleNode getHead() {
        return head;
    }

    public void setHead(SingleNode head) {
        this.head = head;
    }

    public SingleNode getTail() {
        return tail;
    }
```

```
public void setTail(SingleNode tail) {
    this.tail = tail;
}

public int getSize() {
    return size;
}

public void setSize(int size) {
    this.size = size;
}


public void insertInLinkedList(int nodeValue, int location) {
    SingleNode node = new SingleNode();
    node.setValue(nodeValue);
    if (!existsLinkedList()) { // Linked List does not exists
        System.out.println("The linked list does not exist!!");
        return;
    } else if (location == 0) {// insert at first position
        node.setNext(head);
        head = node;
    } else if (location >= size) {// insert at last position
        node.setNext(null);
        tail.setNext(node);
        tail = node;
    } else {// insert at specified location
        SingleNode tempNode = head;
        int index = 0;
        while (index < location - 1) {// loop till we reach specified node
            tempNode = tempNode.getNext();
            index++;
        }//tempNode currently references to node after which we should insert new node
        SingleNode nextNode = tempNode.getNext(); //this is the immediate next node after new node
        tempNode.setNext(node);//update reference of tempNode to reference to new node
            node.setNext(nextNode);//update newly added nodes' next.
        }
        setSize(getSize()+1);
    }


    public boolean existsLinkedList() {
        // if head is not null retrun true otherwise return false
        return head != null;
    }


    //Traverses Linked List
    void traverseLinkedList() {
        if (existsLinkedList()) {
            SingleNode tempNode = head;
            for (int i = 0; i < getSize(); i++) {
                System.out.print(tempNode.getValue());
                if (i != getSize() - 1) {
                    System.out.print(" -> ");
                }
                tempNode = tempNode.getNext();
            }
        }else {
            System.out.println("Linked List does not exists !");
        }
        System.out.println("\n");
    }
```

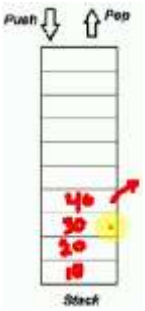
```
    //Deletes entire Linked List
    void deleteLinkedList() {
        System.out.println("\n\nDeleting Linked List...");
        head = null;
        tail = null;
        System.out.println("Linked List deleted successfully !");
    }


    //Searches a node with given value
    boolean searchNode(int nodeValue) {
        if (existsLinkedList()) {
            SingleNode tempNode = head;
            for (int i = 0; i < getSize(); i++) {
                if (tempNode.getValue() == nodeValue) {
                    System.out.print("Found the node at location: "+i+"\n");
                    return true;
                }
                tempNode = tempNode.getNext();
            }
        }
        System.out.print("Node not found!! \n");
        return false;
    }


    //Deletes a node having a given value
    public void deletionOfNode(int location) {
        if (!existsLinkedList()) {
            System.out.println("The linked list does not exist!!");// Linked List does not exists
            return;
        } else if (location == 0) { // we want to delete first element
            head = head.getNext();
            setSize(getSize()-1);
            if(getSize() == 0) { // if there are no more nodes in this list
                tail = null;
            }
        }else if (location >= getSize()){ //If location is not in range or equal, then delete last node
            SingleNode tempNode = head;
            for (int i = 0; i < size - 1; i++) {
                tempNode = tempNode.getNext(); //temp node points to 2nd last node
            }
            if (tempNode == head) { //if this is the only element in the list
                tail = head = null;
                setSize(getSize()-1);
                return;
            }
            tempNode.setNext(null);
            tail= tempNode;
            setSize(getSize()-1);

        }else { //if any inside node is to be deleted
            SingleNode tempNode = head;
            for (int i = 0; i < location - 1; i++) {
                tempNode = tempNode.getNext(); // we need to traverse till we find the location
            }
            tempNode.setNext(tempNode.getNext().getNext()); // delete the required node
            setSize(getSize()-1);
        }//end of else

    }//end of method

}// end of class
```

You can check code for other linked lists in my repository(https://github.com/praveennaga/praveennaga-datastructure-algorithm/tree/master/src/main/java/com/praveennaga/dsalg/linkedlist)

Stack

Properties of Stack

Follows LIFO(Last In First Out) method



For example, implementation of 'back button' in browser.

Common Operations in Stack

1. CreateStack()
2. Push()
3. Pop()
4. Peek()
5. isEmpty()
6. isFull()
7. DeleteStack()

Implementation of Stack

1. Using Arrays

Pros: Easy to Implement

Cons: Fixed Size

You can refer the code in https://github.com/praveennaga/praveennaga-datastructure-algorithm/tree/master/src/main/java/com/praveennaga/dsalg/array

2. Using Linked List

Pros: Variable Size

Cons: Moderate in Implementation.

You can refer the code in my git repository(https://github.com/praveennaga/praveen-java-datastructure-algorithm/tree/master/src/main/java/com/praveen/dsalg/stack/linkedlist)

When to Use/Avoid Stack?

When to Use

- Helps manage data in particular way(LIFO)
- Cannot be easily corrupted(No one can insert data in middle)

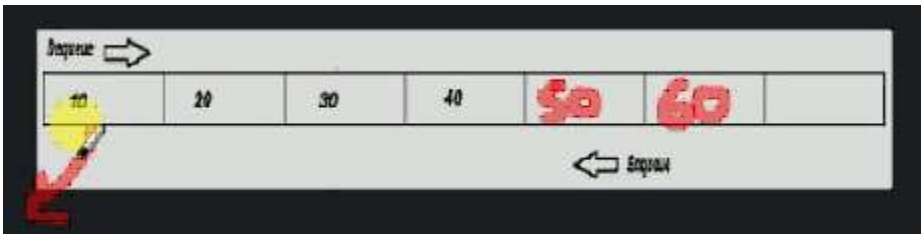
When to Avoid

- Random Access not possible(If we have done some mistake it is costly to rectify)

Queue

Properties of Queue

It follows FIFO(First In First Out) method.



For example ,Implementation of billing Counter

Common Operations of Queue

1. CreateQueue
2. EnQueue
3. deQueue
4. peekInQueue
5. isEmpty()
6. isFull()
7. deleteQueue()

You can refer the code for Linear Queue (Array Implementation) in
https://github.com/praveennaga/praveennaga-datastructure-algorithm/tree/master/src/main/java/com/praveennaga/dsalg/queue/linearqueue/array

What is Circular Queue?

Dequeue operation causes blanks cells Linear Queue(Array Implementation)

In order to avoid this, we need to go for Circular Queue

You can refer the code for Circular Queue(Array Implementation) in
https://github.com/praveennaga/praveennaga-datastructure-algorithm/tree/master/src/main/java/com/praveennaga/dsalg/queue/circularqueue/array

We can also implement the Queue using List and you can refer the code in
https://github.com/praveennaga/praveennaga-datastructure-algorithm/tree/master/src/main/java/com/praveennaga/dsalg/queue/list

When to Use/Avoid Queue

When to Use

- Helps to manage the data in a particular way(FIFO)
- Not easily corrupted(No one can insert the data in middle)

When to Avoid

- Random Access not possible(If we have done some mistake it is costly to rectify)

Sorting Techniques

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merged Sort
5. Quick Sort
6. Heap Sort

Please refer the below screenshot for complexities
operation count(O)
number of elements(n)

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Selection sort	n^2	n^2	n^2	1	Depends
Bubble sort	n	n^2	n^2	1	Yes

Bubble Sort

It compares the adjacent elements and swaps until the sorting is achieved.
For example,
int[] num= {90,24,46,35,32};
Lets see how bubble sort works
First round:
90,24,46,35,32
24,90,46,35,32
24,46,90,35,32
24,46,35,90,32
24,46,35,32,90
Second round:
24,46,35,32,90
24,35,46,32,90
24,35,32,46,90
Third round:
24,35,32,46,90
24,32,35,46,90
That's it after 3 rounds of adjacent element comparison we have achieved the sorted order using Bubble sort.

Lets start coding for bubble sort

```
public class BubbleSort {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        int temp = 0;
        // Comparison of adjacent elements and swap until the sorted order is achieved.
        for (int i = 0; i < num.length; i++) {
            int flag = 0;
            for (int j = 0; j < num.length - 1 - i; j++) {
                if (num[j] > num[j + 1]) {
                    temp = num[j];
                    num[j] = num[j + 1];
                    num[j + 1] = temp;
                    flag = 1;
                }
            }
            // This used  if the adjacent element is not having higher value then there is no need to iterate
            if (flag == 0) {
                break;
            }
        }
        // After sorting
        for (int i : num) {
            System.out.print(i+" ");
        }

    }
}
```

Output: 24 32 35 46 90

Selection Sort

Selection sort is combination of searching and sorting.
It sorts the array by repeatedly finding the minimum element from unsorted part and putting it at the beginning until the sorting is achieved.

For example,
int[] num = { 90, 24, 46, 35, 32 };
Lets see how selection sort works,
90,24,46,35,32
24,90,46,35,32
24,32,46,35,90
24,32,35,46,90

Lets start coding

```
public class SelectionSort {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        int min, temp = 0;
        for (int i = 0; i < num.length; i++) {
            min = i;
            for (int j = i + 1; j < num.length; j++) {
                if (num[j] < num[min]) {
                    min = j;
                }
            }
            temp = num[i];
            num[i] = num[min];
            num[min] = temp;
        }
        // After sorting
        for (int i : num) {
            System.out.print(i + " ");
        }
    }
}
```

Output: 24 32 35 46 90

Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hand.We choose one card and insert in its position.
This is useful only for sorting small set of elements.
Pick element and insert it into sorted sequence
Lets see how insertion sort works

int[] num = { 90, 24, 46, 35, 32 };
90,24,46,35,32
24,90,46,35,32
24,46,90,35,32
24,35,46,90,32
24,32,35,46,90

Let's start coding

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        int temp, j;
        for (int i = 1; i < num.length; i++) {
            temp = num[i];
            j = i;
            while (j > 0 && num[j - 1] > temp) {
                num[j] = num[j - 1];
                j = j - 1;
            }
            num[j] = temp;
        }
        // After sorting
        for (int i : num) {
            System.out.print(i + " ");
        }
    }
}
```

Output: 24 32 35 46 90

Merged Sort

Merged sort is one of the popular sorting algorithms and it is widely used technique. It is better compared to Bubble Sort, Selection Sort and Inserting Sort.
Algorithm: Divide and Conquer

1. Divide the unsorted list into n sublists each containing 1 element.
2. Take adjacent pairs of 2 singleton lists and merge them to form a list of 2 elements. n will now convert into n/2 lists of size 2
3. Repeat the process until it is sorted.

Lets see how Merged Sort works
int[] num = { 90, 24, 46, 35, 32 };
90,24,46,35,32
90,24 46,35,32
90 24 46 35,32

90 24 46 35 32

Now start merging as we have got individual elements

24 90 45 32 35
24 90 32 35 45
24 32 35 45 90

Lets start coding

```
public class MergedSort {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        mergeSort(num, num.length);
        // After sorting
        for (int i : num) {
            System.out.print(i + " ");
        }
    }

    public static void mergeSort(int[] a, int n) {
        if (n < 2) {
            return;
        }
        int mid = n / 2;
        int[] l = new int[mid];
        int[] r = new int[n - mid];

        for (int i = 0; i < mid; i++) {
            l[i] = a[i];
        }
        for (int i = mid; i < n; i++) {
            r[i - mid] = a[i];
        }
        mergeSort(l, mid);
        mergeSort(r, n - mid);

        merge(a, l, r, mid, n - mid);
    }

    public static void merge(int[] a, int[] l, int[] r, int left, int right) {

        int i = 0, j = 0, k = 0;
        while (i < left && j < right) {
            if (l[i] <= r[j]) {
                a[k++] = l[i++];
            } else {
                a[k++] = r[j++];
            }
        }
        while (i < left) {
            a[k++] = l[i++];
        }
        while (j < right) {
            a[k++] = r[j++];
        }
    }
}
```

Output 24 32 35 46 90

Blog: https://praveennagatech.blogspot.com
Facebook Group: https://www.facebook.com/groups/268426377837151
Github repo: https://github.com/praveennaga
Email: opraveennaga@gmail.com

QuickSort

Quicksort is a sorting algorithm, which is leveraging the divide-and-conquer principle. It has an average O(n log n) complexity and it's one of the most used sorting algorithms, especially for big data volumes.

It's important to remember that Quicksort isn't a stable algorithm. A stable sorting algorithm is an algorithm where the elements with the same values appear in the same order in the sorted output as they appear in the input list.

1. We choose an element from the list, called the pivot. We'll use it to divide the list into two sub-lists.
2. We reorder all the elements around the pivot – the ones with smaller value are placed before it, and all the elements greater than the pivot after it. After this step, the pivot is in its final position. This is the important partition step.
3. We apply the above steps recursively to both sub-lists on the left and right of the pivot.

Lets start coding

```
public class QuickSort {
    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        quickSort(num, 0, num.length-1);
        // After sorting
        for (int i : num) {
            System.out.print(i + " ");
        }
    }

    public static void quickSort(int arr[], int begin, int end) {
        if (begin < end) {
            int partitionIndex = partition(arr, begin, end);

            quickSort(arr, begin, partitionIndex-1);
            quickSort(arr, partitionIndex+1, end);
        }
    }

    private static int partition(int arr[], int begin, int end) {
        int pivot = arr[end];
        int i = (begin-1);

        for (int j = begin; j < end; j++) {
            if (arr[j] <= pivot) {
                i++;

                int swapTemp = arr[i];
                arr[i] = arr[j];
                arr[j] = swapTemp;
```

```
            }
        }

        int swapTemp = arr[i+1];
        arr[i+1] = arr[end];
        arr[end] = swapTemp;

        return i+1;
    }
}
```

Output 24 32 35 46 90

Heap Sort

A Heap is a specialized tree-based data structure.
Tree is a non linear data structure that consists of nodes with a parent-child relationship.
Heap must be complete or almost binary tree and child node must be less than parent node.

Lets start coding

```
public class HeapSort {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        heapSort(num);
        // After sorting
        for (int i : num) {
            System.out.print(i + " ");
        }
    }

    private static void heapSort(int[] num) {
        int len = num.length;
        for (int i = len / 2 - 1; i >= 0; i--) {
            heapify(num, len, i);
        }

        // swap the elements and heapify again
        for (int i = len - 1; i >= 0; i--) {
            int temp = num[0];
            num[0] = num[i];
            num[i] = temp;
            heapify(num, i, 0);
        }
    }
```

```
    private static void heapify(int[] num, int len, int i) {
        int largest = i; // parent node index position
        int li = 2 * i + 1; // left child node index position
        int ri = 2 * i + 2; // right child node index position
        if (li < len && num[li] > num[largest]) {
            largest = li;
        }
        if (ri < len && num[ri] > num[largest]) {
            largest = ri;
        }
        if (largest != i) {
            int temp = num[i];
            num[i] = num[largest];
            num[largest] = temp;
            heapify(num, len, largest);
        }
    }
}
```

Output 24 32 35 46 90

Searching Algorithms

1.Linear Search
2.Binary Search

Linear Search

Linear search is a very simple algorithm.In this type of search, sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise search will continue till the end of the data collection.

Lets start coding

```
public class LinearSearch {

    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        int item=32;
        int temp=0;
        for(int i=0;i<num.length;i++) {
            if(num[i]==item) {
                System.out.println("Item "+item +" present at index position "+ i);
                temp=temp+1;
            }
        }
        if(temp==0) {
            System.out.println("Item "+item +" not found");
        }
    }

}
```

Output: Item 32 present at index position 4

Binary Search

Binary Search is the process of searching an element from sorted array by dividing the search interval in half.
Binary Search is faster than Linear Search.
Although Binary Search is a very optimized way of searching a particular element but the array must be sorted on which you want to perform search process.
If Array is unsorted then we have to perform the sort first and then we can perform binary search.

Lets start coding

```
public class BinarySearch {
    public static void main(String[] args) {
        int[] num = { 90, 24, 46, 35, 32 };
        int key=46;
        binarySearch(num,0,num.length,key);
    }

    public static void binarySearch(int arr[], int first, int last, int key){
           int mid = (first + last)/2;
           while( first <= last ){
              if ( arr[mid] < key ){
                first = mid + 1;
              }else if ( arr[mid] == key ){
                System.out.println("Element "+ key+ " is found at index: " + mid);
                break;
              }else{
                 last = mid - 1;
              }
              mid = (first + last)/2;
           }
           if ( first > last ){
              System.out.println("Element is not found!");
           }
        }
}
```

Output: Element 46 is found at index: 2

You can refer complete code in https://github.com/praveennaga/praveennaga-datastructure-algorithm

Please check out my other ebooks in https://github.com/praveennaga/PraveenNaga-Tech-Ebooks

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1